z3c.form Documentation

Release 5.1

Zope Foundation and Contributors

Jul 19, 2023

CONTENTS

`z3c.form` provides an implementation for both HTML and JSON forms and according widgets. Its goal is to provide a simple API but with the ability to easily customize any data or steps.

There are currently two maintained branches:

- master with the latest changes
- 3.x without the object widget overhaul and still including the `ObjectSubForm` and the `SubformAdapter`.

Documentation on this implementation and its API can be found at https://z3cform.readthedocs.io/

The documents are ordered in the way they should be read:

CHAPTER

ONE

MUST READ

1.1 Forms

The purpose of this package is to make development of forms as simple as possible, while still providing all the hooks to do customization at any level as required by our real-world use cases. Thus, once the system is set up with all its default registrations, it should be trivial to develop a new form.

The strategy of this document is to provide the most common, and thus simplest, case first and then demonstrate the available customization options. In order to not overwhelm you with our set of well-chosen defaults, all the default component registrations have been made prior to doing those examples:

```
>>> from z3c.form import testing
>>> testing.setupFormDefaults()
```

Note, since version 2.4.2 the IFormLayer doesn't provide IBrowserRequest anymore. This is useful if you like to use z3c.form components for other requests than the IBrowserRequest.

```
>>> from zope.publisher.interfaces.browser import IBrowserRequest
>>> import z3c.form.interfaces
>>> z3c.form.interfaces.IFormLayer.isOrExtends(IBrowserRequest)
False
```

Before we can start writing forms, we must have the content to work with:

```
>>> import zope.interface
>>> import zope.schema
>>> class IPerson(zope.interface.Interface):
...
...     id = zope.schema.TextLine(
...         title='ID',
...         readonly=True,
...         required=True)
...
...     name = zope.schema.TextLine(
...         title='Name',
...         required=True)
...
...     gender = zope.schema.Choice(
...         title='Gender',
...         values=('male', 'female'),
...         required=False)
```

(continues on next page)

(continued from previous page)

```
...
...     age = zope.schema.Int(
...         title='Age',
...         description=u"The person's age.",
...         min=0,
...         default=20,
...         required=False)
...
...     @zope.interface.invariant
...     def ensureIdAndNameNotEqual(person):
...         if person.id == person.name:
...             raise zope.interface.Invalid(
...                 "The id and name cannot be the same.")
```

```
>>> from zope.schema.fieldproperty import FieldProperty
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...     id = FieldProperty(IPerson['id'])
...     name = FieldProperty(IPerson['name'])
...     gender = FieldProperty(IPerson['gender'])
...     age = FieldProperty(IPerson['age'])
...
...     def __init__(self, id, name, gender=None, age=None):
...         self.id = id
...         self.name = name
...         if gender:
...             self.gender = gender
...         if age:
...             self.age = age
...
...     def __repr__(self):
...         return '<%s %r>' % (self.__class__.__name__, self.name)
```

Okay, that should suffice for now.

What's next? Well, first things first. Let's create an add form for the person. Since practice showed that the `IAdding` interface is overkill for most projects, the default add form of `z3c.form` requires you to define the creation and adding mechanism.

Note:

If it is not done, `NotImplementedError[s]` are raised:

```
>>> from z3c.form.testing import TestRequest
>>> from z3c.form import form, field
```

```
>>> abstract = form.AddForm(None, TestRequest())
```

```
>>> abstract.create({})
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> abstract.add(1)
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> abstract.nextURL()
Traceback (most recent call last):
...
NotImplementedError
```

Thus let's now create a working add form:

```
>>> class PersonAddForm(form.AddForm):
...
...     fields = field.Fields(IPerson)
...
...     def create(self, data):
...         return Person(**data)
...
...     def add(self, object):
...         self.context[object.id] = object
...
...     def nextURL(self):
...         return 'index.html'
```

This is as simple as it gets. We explicitly define the pieces that are custom to every situation and let the default setup of the framework do the rest. This is intentionally similar to `zope.formlib`, because we really like the simplicity of `zope.formlib`'s way of dealing with the common use cases.

Let's try to add a new person object to the root folder (which was created during test setup). For this add form, of course, the context is now the root folder:

```
>>> request = TestRequest()
>>> addForm = PersonAddForm(root, request)
```

Since forms are not necessarily pages – in fact often they are not – they must not have a `__call__` method that does all the processing and rendering at once. Instead, we use the update/render pattern. Thus, we first call the `update()` method.

```
>>> addForm.update()
```

Actually a lot of things happen during this stage. Let us step through it one by one pointing out the effects.

1.1.1 Find a widget manager and update it

The default widget manager knows to look for the `fields` attribute in the form, since it implements `IFieldsForm`:

```
>>> from z3c.form import interfaces
>>> interfaces.IFieldsForm.providedBy(addForm)
True
```

The widget manager is then stored in the `widgets` attribute as promised by the `IForm` interface:

```
>>> addForm.widgets
FieldWidgets([...])
```

The widget manager will have four widgets, one for each field:

```
>>> list(addForm.widgets.keys())
['id', 'name', 'gender', 'age']
```

When the widget manager updates itself, several sub-tasks are processed. The manager goes through each field, trying to create a fully representative widget for the field.

Field Availability

Just because a field is requested in the field manager, does not mean that a widget has to be created for the field. There are cases when a field declaration might be ignored. The following reasons come to mind:

- No widget is created if the data are not accessible in the content.
- A custom widget manager has been registered to specifically ignore a field.

In our simple example, all fields will be converted to widgets.

Widget Creation

During the widget creation process, several pieces of information are transferred from the field to the widget:

```
>>> age = addForm.widgets['age']
```

field.title -> age.label

```
>>> age.label
'Age'
```

field.required -> age.required

```
>>> age.required
False
```

All these values can be overridden at later stages of the updating process.

Widget Value

The next step is to determine the value that should be displayed by the widget. This value could come from three places (looked up in this order):

1. The field's default value.
2. The content object that the form is representing.
3. The request in case a form has not been submitted or an error occurred.

Since we are currently building an add form and not an edit form, there is no content object to represent, so the second step is not applicable. The third step is also not applicable as we do not have anything in the request. Therefore, the value should be the field's default value, or be empty. In this case the field provides a default value:

```
>>> age.value
'20'
```

While the default of the age field is actually the integer `20`, the widget has converted the value to the output-ready string `'20'` using a data converter.

Widget Mode

Now the widget manager looks at the field to determine the widget mode – in other words whether the widget is a display or edit widget. In this case all fields are input fields:

```
>>> age.mode
'input'
```

Deciding which mode to use, however, might not be a trivial operation. It might depend on several factors (items listed later override earlier ones):

- The global `mode` flag of the widget manager
- The permission to the content's data value
- The `readonly` flag in the schema field
- The `mode` flag in the field

Widget Attribute Values

As mentioned before, several widget attributes are optionally overridden when the widget updates itself:

- label
- required
- mode

Since we have no customization components registered, all of those fields will remain as set before.

1.1.2 Find an action manager, update and execute it

After all widgets have been instantiated and the `update()` method has been called successfully, the actions are set up. By default, the form machinery uses the button declaration on the form to create its actions. For the add form, an add button is defined by default, so that we did not need to create our own. Thus, there should be one action:

```
>>> len(addForm.actions)
1
```

The add button is an action and a widget at the same time:

```
>>> addAction = addForm.actions['add']
>>> addAction.title
'Add'
>>> addAction.value
'Add'
```

After everything is set up, all pressed buttons are executed. Once a submitted action is detected, a special action handler adapter is used to determine the actions to take. Since the add button has not been pressed yet, no action occurred.

1.1.3 Rendering the form

Once the update is complete we can render the form using one of two methods reder or json. If we want to generate json data to be consumed by the client all we need to do is call json():

```
>>> import json
>>> from pprint import pprint
>>> pprint(json.loads(addForm.json()))
{'errors': [],
 'fields': [{'error': '',
             'id': 'form-widgets-id',
             'label': 'ID',
             'mode': 'input',
             'name': 'form.widgets.id',
             'required': True,
             'type': 'text',
             'value': ''},
            {'error': '',
             'id': 'form-widgets-name',
             'label': 'Name',
             'mode': 'input',
             'name': 'form.widgets.name',
             'required': True,
             'type': 'text',
             'value': ''},
            {'error': '',
             'id': 'form-widgets-gender',
             'label': 'Gender',
             'mode': 'input',
             'name': 'form.widgets.gender',
             'options': [{'content': 'No value',
                          'id': 'form-widgets-gender-novalue',
                          'selected': True,
                          'value': '--NOVALUE--'},
                         {'content': 'male',
                          'id': 'form-widgets-gender-0',
                          'selected': False,
                          'value': 'male'},
                         {'content': 'female',
                          'id': 'form-widgets-gender-1',
                          'selected': False,
                          'value': 'female'}],
             'required': False,
             'type': 'select',
             'value': []},
            {'error': '',
             'id': 'form-widgets-age',
             'label': 'Age',
             'mode': 'input',
             'name': 'form.widgets.age',
             'required': False,
             'type': 'text',
             'value': '20'}],
```

(continues on next page)

(continued from previous page)

```
 'label': '',
 'mode': 'input',
 'prefix': 'form.',
 'status': ''}
```

The other way we can render the form is using the render() method.

The render method requires us to specify a template, we have to do this now. We have prepared a small and very simple template as part of this example:

```
>>> import os
>>> from zope.browserpage.viewpagetemplatefile import BoundPageTemplate
>>> from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile
>>> from z3c.form import tests
>>> def addTemplate(form):
...     form.template = BoundPageTemplate(
...         ViewPageTemplateFile(
...             'simple_edit.pt', os.path.dirname(tests.__file__)), form)
>>> addTemplate(addForm)
```

Let's now render the page:

```
>>> print(addForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-id">ID</label>
        <input type="text" id="form-widgets-id"
               name="form.widgets.id"
               class="text-widget required textline-field"
               value="" />
      </div>
      <div class="row">
        <label for="form-widgets-name">Name</label>
        <input type="text" id="form-widgets-name" name="form.widgets.name"
               class="text-widget required textline-field"
               value="" />
      </div>
      <div class="row">
        <label for="form-widgets-gender">Gender</label>
        <select id="form-widgets-gender" name="form.widgets.gender:list"
                class="select-widget choice-field" size="1">
          <option id="form-widgets-gender-novalue" selected="selected"
                  value="--NOVALUE--">No value</option>
          <option id="form-widgets-gender-0" value="male">male</option>
          <option id="form-widgets-gender-1" value="female">female</option>
        </select>
        <input name="form.widgets.gender-empty-marker" type="hidden"
               value="1" />
      </div>
      <div class="row">
        <label for="form-widgets-age">Age</label>
```

(continues on next page)

(continued from previous page)

```
        <input type="text" id="form-widgets-age" name="form.widgets.age"
               class="text-widget int-field" value="20" />
      </div>
      <div class="action">
        <input type="submit" id="form-buttons-add" name="form.buttons.add"
               class="submit-widget button-field" value="Add" />
      </div>
    </form>
  </body>
</html>
```

The update()/render() cycle is what happens when the form is called, i.e. when it is published:

```
>>> print(addForm())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-id">ID</label>
        <input type="text" id="form-widgets-id"
               name="form.widgets.id"
               class="text-widget required textline-field"
               value="" />
      </div>
      <div class="row">
        <label for="form-widgets-name">Name</label>
        <input type="text" id="form-widgets-name" name="form.widgets.name"
               class="text-widget required textline-field"
               value="" />
      </div>
      <div class="row">
        <label for="form-widgets-gender">Gender</label>
        <select id="form-widgets-gender" name="form.widgets.gender:list"
                class="select-widget choice-field" size="1">
          <option id="form-widgets-gender-novalue" selected="selected"
                  value="--NOVALUE--">No value</option>
          <option id="form-widgets-gender-0" value="male">male</option>
          <option id="form-widgets-gender-1" value="female">female</option>
        </select>
        <input name="form.widgets.gender-empty-marker" type="hidden"
               value="1" />
      </div>
      <div class="row">
        <label for="form-widgets-age">Age</label>
        <input type="text" id="form-widgets-age" name="form.widgets.age"
               class="text-widget int-field" value="20" />
      </div>
      <div class="action">
        <input type="submit" id="form-buttons-add" name="form.buttons.add"
               class="submit-widget button-field" value="Add" />
      </div>
    </form>
```

(continues on next page)

(continued from previous page)

```
  </body>
</html>
```

Note that we don't actually call render if the response has been set to a 3xx type status code (e.g. a redirect or not modified response), since the browser would not render it anyway:

```
>>> request.response.setStatus(304)
>>> print(addForm())
```

Let's go back to a normal status to continue the test.

```
>>> request.response.setStatus(200)
```

1.1.4 Registering a custom event handler for the DataExtractedEvent

```
>>> data_extracted_eventlog = []
>>> from z3c.form.events import DataExtractedEvent
>>> @zope.component.adapter(DataExtractedEvent)
... def data_extracted_logger(event):
...     data_extracted_eventlog.append(event)
>>> zope.component.provideHandler(data_extracted_logger)
```

1.1.5 Submitting an add form successfully

Initially the root folder of the application is empty:

```
>>> sorted(root)
[]
```

Let's now fill the request with all the right values so that upon submitting the form with the "Add" button, the person should be added to the root folder:

```
>>> request = TestRequest(form={
...     'form.widgets.id': 'srichter',
...     'form.widgets.name': 'Stephan Richter',
...     'form.widgets.gender': ['male'],
...     'form.widgets.age': '20',
...     'form.buttons.add': 'Add'}
...     )
```

```
>>> addForm = PersonAddForm(root, request)
>>> addForm.update()
```

```
>>> sorted(root)
['srichter']
>>> stephan = root['srichter']
>>> stephan.id
'srichter'
>>> stephan.name
```

(continues on next page)

(continued from previous page)

```
'Stephan Richter'
>>> stephan.gender
'male'
>>> stephan.age
20
```

1.1.6 Check, if DataExtractedEvent was thrown

```
>>> event = data_extracted_eventlog[0]
>>> 'name' in event.data
True
```

```
>>> event.errors
()
```

```
>>> event.form
<PersonAddForm object at ...
```

1.1.7 Submitting an add form with invalid data

Next we try to submit the add form with the required name missing. Thus, the add form should not complete with the addition, but return with the add form pointing out the error.

```
>>> request = TestRequest(form={
...     'form.widgets.id': 'srichter',
...     'form.widgets.gender': ['male'],
...     'form.widgets.age': '23',
...     'form.buttons.add': 'Add'}
...     )
```

```
>>> addForm = PersonAddForm(root, request)
>>> addForm.update()
```

The widget manager and the widget causing the error should have an error message:

```
>>> [(error.widget.__name__, error) for error in addForm.widgets.errors]
[('name', <ErrorViewSnippet for RequiredMissing>)]
```

```
>>> addForm.widgets['name'].error
<ErrorViewSnippet for RequiredMissing>
```

Check, if event was thrown:

```
>>> event = data_extracted_eventlog[-1]
>>> 'id' in event.data
True
```

```
>>> event.errors
(<ErrorViewSnippet for RequiredMissing>,)
```

```
>>> event.form
<PersonAddForm object at ...
```

Let's now render the form:

```
>>> addTemplate(addForm)
>>> print(addForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <i>There were some errors.</i>
    <ul>
      <li>
        Name: <div class="error">Required input is missing.</div>
      </li>
    </ul>
    <form action=".">
      <div class="row">
        <label for="form-widgets-id">ID</label>
        <input type="text" id="form-widgets-id"
               name="form.widgets.id"
               class="text-widget required textline-field"
               value="srichter" />
      </div>
      <div class="row">
        <b><div class="error">Required input is missing.</div>
        </b><label for="form-widgets-name">Name</label>
        <input type="text" id="form-widgets-name" name="form.widgets.name"
               class="text-widget required textline-field" value="" />
      </div>
      <div class="row">
        <label for="form-widgets-gender">Gender</label>
        <select id="form-widgets-gender" name="form.widgets.gender:list"
                class="select-widget choice-field" size="1">
          <option id="form-widgets-gender-novalue"
                  value="--NOVALUE--">No value</option>
          <option id="form-widgets-gender-0" value="male"
                  selected="selected">male</option>
          <option id="form-widgets-gender-1" value="female">female</option>
        </select>
        <input name="form.widgets.gender-empty-marker" type="hidden"
               value="1" />
      </div>
      <div class="row">
        <label for="form-widgets-age">Age</label>
        <input type="text" id="form-widgets-age" name="form.widgets.age"
               class="text-widget int-field" value="23" />
      </div>
      <div class="action">
        <input type="submit" id="form-buttons-add" name="form.buttons.add"
               class="submit-widget button-field" value="Add" />
      </div>
    </form>
  </body>
```

(continues on next page)

(continued from previous page)

```
</html>
```

Notice the errors are present in the json output of the form as well

```
>>> import json
>>> from pprint import pprint
>>> pprint(json.loads(addForm.json()))
{'errors': [],
 'fields': [{'error': '',
             'id': 'form-widgets-id',
             'label': 'ID',
             'mode': 'input',
             'name': 'form.widgets.id',
             'required': True,
             'type': 'text',
             'value': 'srichter'},
            {'error': 'Required input is missing.',
             'id': 'form-widgets-name',
             'label': 'Name',
             'mode': 'input',
             'name': 'form.widgets.name',
             'required': True,
             'type': 'text',
             'value': ''},
            {'error': '',
             'id': 'form-widgets-gender',
             'label': 'Gender',
             'mode': 'input',
             'name': 'form.widgets.gender',
             'options': [{'content': 'No value',
                          'id': 'form-widgets-gender-novalue',
                          'selected': False,
                          'value': '--NOVALUE--'},
                         {'content': 'male',
                          'id': 'form-widgets-gender-0',
                          'selected': True,
                          'value': 'male'},
                         {'content': 'female',
                          'id': 'form-widgets-gender-1',
                          'selected': False,
                          'value': 'female'}],
             'required': False,
             'type': 'select',
             'value': ['male']},
            {'error': '',
             'id': 'form-widgets-age',
             'label': 'Age',
             'mode': 'input',
             'name': 'form.widgets.age',
             'required': False,
             'type': 'text',
             'value': '23'}],
```

(continues on next page)

(continued from previous page)

```
 'label': '',
 'mode': 'input',
 'prefix': 'form.',
 'status': 'There were some errors.'}
```

Note that the values of the field are now extracted from the request.

Another way to receive an error is by not fulfilling the invariants of the schema. In our case, the id and name cannot be the same. So let's provoke the error now:

```
>>> request = TestRequest(form={
...     'form.widgets.id': 'Stephan',
...     'form.widgets.name': 'Stephan',
...     'form.widgets.gender': ['male'],
...     'form.widgets.age': '23',
...     'form.buttons.add': 'Add'}
...     )
```

```
>>> addForm = PersonAddForm(root, request)
>>> addTemplate(addForm)
>>> addForm.update()
```

and see how the form looks like:

```
>>> print(addForm.render())
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <i>There were some errors.</i>
    <ul>
      <li>
        <div class="error">The id and name cannot be the same.</div>
      </li>
    </ul>
    ...
  </body>
</html>
```

and through as json:

```
>>> import json
>>> from pprint import pprint
>>> pprint(json.loads(addForm.json()))
 {'errors': ['The id and name cannot be the same.'],
  'fields': [{'error': '',
              'id': 'form-widgets-id',
              'label': 'ID',
              'mode': 'input',
              'name': 'form.widgets.id',
              'required': True,
              'type': 'text',
              'value': 'Stephan'},
```

(continues on next page)

(continued from previous page)

```
             {'error': '',
              'id': 'form-widgets-name',
              'label': 'Name',
              'mode': 'input',
              'name': 'form.widgets.name',
              'required': True,
              'type': 'text',
              'value': 'Stephan'},
             {'error': '',
              'id': 'form-widgets-gender',
              'label': 'Gender',
              'mode': 'input',
              'name': 'form.widgets.gender',
              'options': [{'content': 'No value',
                           'id': 'form-widgets-gender-novalue',
                           'selected': False,
                           'value': '--NOVALUE--'},
                          {'content': 'male',
                           'id': 'form-widgets-gender-0',
                           'selected': True,
                           'value': 'male'},
                          {'content': 'female',
                           'id': 'form-widgets-gender-1',
                           'selected': False,
                           'value': 'female'}],
              'required': False,
              'type': 'select',
              'value': ['male']},
             {'error': '',
              'id': 'form-widgets-age',
              'label': 'Age',
              'mode': 'input',
              'name': 'form.widgets.age',
              'required': False,
              'type': 'text',
              'value': '23'}],
  'label': '',
  'mode': 'input',
  'prefix': 'form.',
  'status': 'There were some errors.'}
```

Let's try to provide a negative age, which is not possible either:

```
>>> request = TestRequest(form={
...     'form.widgets.id': 'srichter',
...     'form.widgets.gender': ['male'],
...     'form.widgets.age': '-5',
...     'form.buttons.add': 'Add'}
...     )
```

```
>>> addForm = PersonAddForm(root, request)
>>> addForm.update()
```

```
>>> [(view.widget.label, view) for view in addForm.widgets.errors]
[('Name', <ErrorViewSnippet for RequiredMissing>),
 ('Age', <ErrorViewSnippet for TooSmall>)]
```

But the error message for a negative age is too generic:

```
>>> print(addForm.widgets['age'].error.render())
<div class="error">Value is too small</div>
```

It would be better to say that negative values are disallowed. So let's register a new error view snippet for the `TooSmall` error:

```
>>> from z3c.form import error
```

```
>>> class TooSmallView(error.ErrorViewSnippet):
...     zope.component.adapts(
...         zope.schema.interfaces.TooSmall, None, None, None, None, None)
...
...     def update(self):
...         super(TooSmallView, self).update()
...         if self.field.min == 0:
...             self.message = 'The value cannot be a negative number.'
```

```
>>> zope.component.provideAdapter(TooSmallView)
```

```
>>> addForm = PersonAddForm(root, request)
>>> addForm.update()
>>> print(addForm.widgets['age'].error.render())
<div class="error">The value cannot be a negative number.</div>
```

Note: The `adapts()` declaration might look strange. An error view snippet is actually a multiadapter that adapts a combination of 6 objects – error, request, widget, field, form, content. By specifying only the error, we tell the system that we do not care about the other discriminators, which then can be anything. We could also have used `zope.interface.Interface` instead, which would be equivalent.

1.1.8 Additional Form Attributes and API

Since we are talking about HTML forms here, add and edit forms support all relevant FORM element attributes as attributes on the class.

```
>>> addForm.method
'post'
>>> addForm.enctype
'multipart/form-data'
>>> addForm.acceptCharset
>>> addForm.accept
```

The `action` attribute is computed. By default it is the current URL:

```
>>> addForm.action
'http://127.0.0.1'
```

The name is also computed. By default it takes the prefix and removes any trailing ".".

```
>>> addForm.name
'form'
```

The id is computed from the name, replacing dots with hyphens. Let's set the prefix to something containing more than one final dot and check how it works.

```
>>> addForm.prefix = 'person.form.add.'
>>> addForm.id
'person-form-add'
```

The template can then use those attributes, if it likes to.

In the examples previously we set the template manually. If no template is specified, the system tries to find an adapter. Without any special configuration, there is no adapter, so rendering the form fails:

```
>>> addForm.template = None
>>> addForm.render()
Traceback (most recent call last):
...
ComponentLookupError: ((...), <InterfaceClass ...IPageTemplate>, '')
```

The form module provides a simple component to create adapter factories from templates:

```
>>> factory = form.FormTemplateFactory(
...     testing.getPath('../tests/simple_edit.pt'), form=PersonAddForm)
```

Let's register our new template-based adapter factory:

```
>>> zope.component.provideAdapter(factory)
```

Now the factory will be used to provide a template:

```
>>> print(addForm.render())
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
...
</html>
```

Since a form can also be used as a page itself, it is callable. When you call it will invoke both the `update()` and `render()` methods:

```
>>> print(addForm())
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
...
</html>
```

The form also provides a label for rendering a required info. This required info depends by default on the given requiredInfo label and if at least one field is required:

```
>>> addForm.requiredInfo
'<span class="required">*</span>&ndash; required'
```

If we set the labelRequired to None, we do not get a requiredInfo label:

```
>>> addForm.labelRequired = None
>>> addForm.requiredInfo is None
True
```

1.1.9 Changing Widget Attribute Values

It frequently happens that a customer comes along and wants to slightly or totally change some of the text shown in forms or make optional fields required. It does not make sense to always have to adjust the schema or implement a custom schema for these use cases. With the z3c.form framework all attributes – for which it is sensible to replace a value without touching the code – are customizable via an attribute value adapter.

To demonstrate this feature, let's change the label of the name widget from "Name" to "Full Name":

```
>>> from z3c.form import widget
>>> NameLabel = widget.StaticWidgetAttribute(
...     'Full Name', field=IPerson['name'])
>>> zope.component.provideAdapter(NameLabel, name='label')
```

When the form renders, the label has now changed:

```
>>> addForm = PersonAddForm(root, TestRequest())
>>> addTemplate(addForm)
>>> addForm.update()
>>> print(testing.render(addForm, './/xmlns:div[2][@class="row"]'))
<div class="row">
  <label for="form-widgets-name">Full Name</label>
    <input id="form-widgets-name" name="form.widgets.name" class="text-widget required␣
↪textline-field" value="" type="text" />
</div>
...
```

1.1.10 Adding a "Cancel" button

Let's say a client requests that all add forms should have a "Cancel" button. When the button is pressed, the user is forwarded to the next URL of the add form. As always, the goal is to not touch the core implementation of the code, but make those changes externally.

Adding a button/action is a little bit more involved than changing a value, because you have to insert the additional action and customize the action handler. Based on your needs of flexibility, multiple approaches could be chosen. Here we demonstrate the simplest one.

The first step is to create a custom action manager that always inserts a cancel action:

```
>>> from z3c.form import button
>>> class AddActions(button.ButtonActions):
...     zope.component.adapts(
...         interfaces.IAddForm,
...         zope.interface.Interface,
...         zope.interface.Interface)
...
...     def update(self):
```

(continues on next page)

(continued from previous page)

```
...         self.form.buttons = button.Buttons(
...             self.form.buttons,
...             button.Button('cancel', 'Cancel'))
...         super(AddActions, self).update()
```

After registering the new action manager,

```
>>> zope.component.provideAdapter(AddActions)
```

the add form should display a cancel button:

```
>>> addForm.update()
>>> print(testing.render(addForm, './/xmlns:div[@class="action"]'))
<div class="action">
  <input id="form-buttons-add" name="form.buttons.add" class="submit-widget button-field
↪" value="Add" type="submit" />
</div>
<div class="action">
  <input id="form-buttons-cancel" name="form.buttons.cancel" class="submit-widget button-
↪field" value="Cancel" type="submit" />
</div>
...
```

But showing the button does not mean it does anything. So we also need a custom action handler to handle the cancel action:

```
>>> class AddActionHandler(button.ButtonActionHandler):
...     zope.component.adapts(
...         interfaces.IAddForm,
...         zope.interface.Interface,
...         zope.interface.Interface,
...         button.ButtonAction)
...
...     def __call__(self):
...         if self.action.name == 'form.buttons.cancel':
...            self.form._finishedAdd = True
...            return
...         super(AddActionHandler, self).__call__()
```

After registering the action handler,

```
>>> zope.component.provideAdapter(AddActionHandler)
```

we can press the cancel button and we will be forwarded:

```
>>> request = TestRequest(form={'form.buttons.cancel': 'Cancel'})
```

```
>>> addForm = PersonAddForm(root, request)
>>> addTemplate(addForm)
>>> addForm.update()
>>> addForm.render()
''
```

```
>>> request.response.getStatus()
302
>>> request.response.getHeader('Location')
'index.html'
```

Eventually, we might have action managers and handlers that are much more powerful and some of the manual labor in this example would become unnecessary.

1.1.11 Creating an Edit Form

Now that we have exhaustively covered the customization possibilities of add forms, let's create an edit form. Edit forms are even simpler than add forms, since all actions are completely automatic:

```
>>> class PersonEditForm(form.EditForm):
...
...     fields = field.Fields(IPerson)
```

We can use the created person from the successful addition above.

```
>>> editForm = PersonEditForm(root['srichter'], TestRequest())
```

After adding a template, we can look at the form:

```
>>> addTemplate(editForm)
>>> editForm.update()
>>> print(editForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
          <label for="form-widgets-id">ID</label>
          <span id="form-widgets-id"
                class="text-widget textline-field">srichter</span>
      </div>
      <div class="row">
        <label for="form-widgets-name">Full Name</label>
        <input type="text" id="form-widgets-name" name="form.widgets.name"
               class="text-widget required textline-field"
               value="Stephan Richter" />
      </div>
      <div class="row">
        <label for="form-widgets-gender">Gender</label>
        <select id="form-widgets-gender" name="form.widgets.gender:list"
                class="select-widget choice-field" size="1">
          <option id="form-widgets-gender-novalue"
                  value="--NOVALUE--">No value</option>
          <option id="form-widgets-gender-0" value="male"
                  selected="selected">male</option>
          <option id="form-widgets-gender-1" value="female">female</option>
        </select>
        <input name="form.widgets.gender-empty-marker" type="hidden"
               value="1" />
```

(continues on next page)

(continued from previous page)

```
      </div>
      <div class="row">
        <label for="form-widgets-age">Age</label>
        <input type="text" id="form-widgets-age" name="form.widgets.age"
               class="text-widget int-field" value="20" />
      </div>
      <div class="action">
        <input type="submit" id="form-buttons-apply" name="form.buttons.apply"
               class="submit-widget button-field" value="Apply" />
      </div>
    </form>
  </body>
</html>
```

As you can see, the data are being pulled in from the context for the edit form. Next we will look at the behavior when submitting the form.

1.1.12 Failure Upon Submission of Edit Form

Let's now submit the form having some invalid data.

```
>>> request = TestRequest(form={
...     'form.widgets.name': 'Claudia Richter',
...     'form.widgets.gender': ['female'],
...     'form.widgets.age': '-1',
...     'form.buttons.apply': 'Apply'}
...     )
```

```
>>> editForm = PersonEditForm(root['srichter'], request)
>>> addTemplate(editForm)
>>> editForm.update()
>>> print(editForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <i>There were some errors.</i>
    <ul>
      <li>
        Age: <div class="error">The value cannot be a negative number.</div>
      </li>
    </ul>
    <form action=".">
      <div class="row">
          <label for="form-widgets-id">ID</label>
          <span id="form-widgets-id"
                class="text-widget textline-field">srichter</span>
      </div>
      <div class="row">
        <label for="form-widgets-name">Full Name</label>
        <input type="text" id="form-widgets-name" name="form.widgets.name"
               class="text-widget required textline-field"
               value="Claudia Richter" />
```

(continues on next page)

(continued from previous page)

```
      </div>
      <div class="row">
        <label for="form-widgets-gender">Gender</label>
        <select id="form-widgets-gender" name="form.widgets.gender:list"
                class="select-widget choice-field" size="1">
          <option id="form-widgets-gender-novalue"
                  value="--NOVALUE--">No value</option>
          <option id="form-widgets-gender-0" value="male">male</option>
          <option id="form-widgets-gender-1" value="female"
                  selected="selected">female</option>
        </select>
        <input name="form.widgets.gender-empty-marker" type="hidden"
               value="1" />
      </div>
      <div class="row">
        <b><div class="error">The value cannot be a negative number.</div>
        </b><label for="form-widgets-age">Age</label>
        <input type="text" id="form-widgets-age" name="form.widgets.age"
               class="text-widget int-field" value="-1" />
      </div>
      <div class="action">
        <input type="submit" id="form-buttons-apply" name="form.buttons.apply"
               class="submit-widget button-field" value="Apply" />
      </div>
    </form>
  </body>
</html>
```

1.1.13 Successfully Editing Content

Let's now resubmit the form with valid data, so the data should be updated.

```
>>> request = TestRequest(form={
...     'form.widgets.name': 'Claudia Richter',
...     'form.widgets.gender': ['female'],
...     'form.widgets.age': '27',
...     'form.buttons.apply': 'Apply'}
...     )
```

```
>>> editForm = PersonEditForm(root['srichter'], request)
>>> addTemplate(editForm)
>>> editForm.update()
>>> print(testing.render(editForm, './/xmlns:i'))
<i >Data successfully updated.</i>
...
```

```
>>> stephan = root['srichter']
>>> stephan.name
'Claudia Richter'
>>> stephan.gender
```

(continues on next page)

(continued from previous page)

```
'female'
>>> stephan.age
27
```

When an edit form is successfully committed, a detailed object-modified event is sent out telling the system about the changes. To see the error, let's create an event subscriber for object-modified events:

```
>>> eventlog = []
>>> import zope.lifecycleevent
>>> @zope.component.adapter(zope.lifecycleevent.ObjectModifiedEvent)
... def logEvent(event):
...     eventlog.append(event)
>>> zope.component.provideHandler(logEvent)
```

Let's now submit the form again, successfully changing the age:

```
>>> request = TestRequest(form={
...     'form.widgets.name': 'Claudia Richter',
...     'form.widgets.gender': ['female'],
...     'form.widgets.age': '29',
...     'form.buttons.apply': 'Apply'}
...     )
```

```
>>> editForm = PersonEditForm(root['srichter'], request)
>>> addTemplate(editForm)
>>> editForm.update()
```

We can now look at the event:

```
>>> event = eventlog[-1]
>>> event
<zope...ObjectModifiedEvent object at ...>
```

```
>>> attrs = event.descriptions[0]
>>> attrs.interface
<InterfaceClass builtins.IPerson>
>>> attrs.attributes
('age',)
```

1.1.14 Successful Action with No Changes

When submitting the form without any changes, the form will tell you so.

```
>>> request = TestRequest(form={
...     'form.widgets.name': 'Claudia Richter',
...     'form.widgets.gender': ['female'],
...     'form.widgets.age': '29',
...     'form.buttons.apply': 'Apply'}
...     )
```

```
>>> editForm = PersonEditForm(root['srichter'], request)
>>> addTemplate(editForm)
>>> editForm.update()
>>> print(testing.render(editForm, './/xmlns:i'))
<i >No changes were applied.</i>
...
```

1.1.15 Changing Status Messages

Depending on the project, it is often desirable to change the status messages to fit the application. In `zope.formlib` this was hard to do, since the messages were buried within fairly complex methods that one did not want to touch. In this package all those messages are exposed as form attributes.

There are three messages for the edit form:

- `formErrorsMessage` – Indicates that an error occurred while applying the changes. This message is also available for the add form.
- `successMessage` – The form data was successfully applied.
- `noChangesMessage` – No changes were found in the form data.

Let's now change the `noChangesMessage`:

```
>>> editForm.noChangesMessage = 'No changes were detected in the form data.'
>>> editForm.update()
>>> print(testing.render(editForm, './/xmlns:i'))
<i >No changes were detected in the form data.</i>
...
```

When even more flexibility is required within a project, one could also implement these messages as properties looking up an attribute value. However, we have found this to be a rare case.

1.1.16 Creating Edit Forms for Dictionaries

Sometimes it is not desirable to edit a class instance that implements the fields, but other types of object. A good example is the need to modify a simple dictionary, where the field names are the keys. To do that, a special data manager for dictionaries is available:

```
>>> from z3c.form import datamanager
>>> zope.component.provideAdapter(datamanager.DictionaryField)
```

The only step the developer has to complete is to re-implement the form's `getContent()` method to return the dictionary:

```
>>> personDict = {'id': 'rineichen', 'name': 'Roger Ineichen',
...               'gender': None, 'age': None}
>>> class PersonDictEditForm(PersonEditForm):
...     def getContent(self):
...         return personDict
```

We can now use the form as usual:

```
>>> editForm = PersonDictEditForm(None, TestRequest())
>>> addTemplate(editForm)
>>> editForm.update()
>>> print(editForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-id">ID</label>
        <span id="form-widgets-id"
              class="text-widget textline-field">rineichen</span>
      </div>
      <div class="row">
        <label for="form-widgets-name">Full Name</label>
        <input type="text" id="form-widgets-name"
               name="form.widgets.name"
               class="text-widget required textline-field"
       value="Roger Ineichen" />
      </div>
      <div class="row">
        <label for="form-widgets-gender">Gender</label>
        <select id="form-widgets-gender" name="form.widgets.gender:list"
                class="select-widget choice-field" size="1">
          <option id="form-widgets-gender-novalue"
                  value="--NOVALUE--" selected="selected">No value</option>
          <option id="form-widgets-gender-0" value="male">male</option>
          <option id="form-widgets-gender-1" value="female">female</option>
        </select>
        <input name="form.widgets.gender-empty-marker" type="hidden"
               value="1" />
      </div>
      <div class="row">
        <label for="form-widgets-age">Age</label>
        <input type="text" id="form-widgets-age"
               name="form.widgets.age" class="text-widget int-field"
               value="20" />
      </div>
      <div class="action">
        <input type="submit" id="form-buttons-apply"
               name="form.buttons.apply" class="submit-widget button-field"
               value="Apply" />
      </div>
    </form>
  </body>
</html>
```

Note that the name displayed in the form is identical to the one in the dictionary. Let's now submit a form to ensure that the data are also written to the dictionary:

```
>>> request = TestRequest(form={
...     'form.widgets.name': 'Jesse Ineichen',
...     'form.widgets.gender': ['male'],
...     'form.widgets.age': '5',
```

(continues on next page)

(continued from previous page)

```
...     'form.buttons.apply': 'Apply'}
...     )
>>> editForm = PersonDictEditForm(None, request)
>>> editForm.update()
```

```
>>> len(personDict)
4
>>> personDict['age']
5
>>> personDict['gender']
'male'
>>> personDict['id']
'rineichen'
>>> personDict['name']
'Jesse Ineichen'
```

1.1.17 Creating a Display Form

Creating a display form is simple; just instantiate, update and render it:

```
>>> class PersonDisplayForm(form.DisplayForm):
...     fields = field.Fields(IPerson)
...     template = ViewPageTemplateFile(
...         'simple_display.pt', os.path.dirname(tests.__file__))
```

```
>>> display = PersonDisplayForm(stephan, TestRequest())
>>> display.update()
>>> print(display.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <div class="row">
      <span id="form-widgets-id"
            class="text-widget textline-field">srichter</span>
    </div>
    <div class="row">
      <span id="form-widgets-name"
            class="text-widget textline-field">Claudia Richter</span>
    </div>
    <div class="row">
      <span id="form-widgets-gender"
            class="select-widget choice-field"><span
          class="selected-option">female</span></span>
    </div>
    <div class="row">
      <span id="form-widgets-age" class="text-widget int-field">29</span>
    </div>
  </body>
</html>
```

1.1.18 Simple Form Customization

The form exposes several of the widget manager's attributes as attributes on the form. They are: `mode`, `ignoreContext`, `ignoreRequest`, and `ignoreReadonly`.

Here are the values for the display form we just created:

```
>>> display.mode
'display'
>>> display.ignoreContext
False
>>> display.ignoreRequest
True
>>> display.ignoreReadonly
False
```

These values should be equal to the ones of the widget manager:

```
>>> display.widgets.mode
'display'
>>> display.widgets.ignoreContext
False
>>> display.widgets.ignoreRequest
True
>>> display.widgets.ignoreReadonly
False
```

Now, if we change those values before updating the widgets, ...

```
>>> display.mode = interfaces.INPUT_MODE
>>> display.ignoreContext = True
>>> display.ignoreRequest = False
>>> display.ignoreReadonly = True
```

... the widget manager will have the same values after updating the widgets:

```
>>> display.updateWidgets()
```

```
>>> display.widgets.mode
'input'
>>> display.widgets.ignoreContext
True
>>> display.widgets.ignoreRequest
False
>>> display.widgets.ignoreReadonly
True
```

We can also set the widget prefix when we update the widgets:

```
>>> display.updateWidgets(prefix="person")
>>> display.widgets.prefix
'person'
```

This will affect the individual widgets' names:

```
>>> display.widgets['id'].name
'form.person.id'
```

To use unqualified names, we must clear both the form prefix and the widgets prefix:

```
>>> display.prefix = ""
>>> display.updateWidgets(prefix="")
>>> display.widgets['id'].name
'id'
```

1.1.19 Extending Forms

One very common use case is to extend forms. For example, you would like to use the edit form and its defined "Apply" button, but add another button yourself. Unfortunately, just inheriting the form is not enough, because the new button and handler declarations will override the inherited ones. Let me demonstrate the problem:

```
>>> class BaseForm(form.Form):
...     fields = field.Fields(IPerson).select('name')
...
...     @button.buttonAndHandler('Apply')
...     def handleApply(self, action):
...         print('success')
```

```
>>> list(BaseForm.fields.keys())
['name']
>>> list(BaseForm.buttons.keys())
['apply']
>>> BaseForm.handlers
<Handlers [<Handler for <Button 'apply' 'Apply'>>]>
```

Let's now derive a form from the base form:

```
>>> class DerivedForm(BaseForm):
...     fields = field.Fields(IPerson).select('gender')
...
...     @button.buttonAndHandler('Cancel')
...     def handleCancel(self, action):
...         print('cancel')
```

```
>>> list(DerivedForm.fields.keys())
['gender']
>>> list(DerivedForm.buttons.keys())
['cancel']
>>> DerivedForm.handlers
<Handlers [<Handler for <Button 'cancel' 'Cancel'>>]>
```

The obvious method to "inherit" the base form's information is to copy it over:

```
>>> class DerivedForm(BaseForm):
...     fields = BaseForm.fields.copy()
...     buttons = BaseForm.buttons.copy()
```

(continues on next page)

(continued from previous page)

```
...     handlers = BaseForm.handlers.copy()
...
...     fields += field.Fields(IPerson).select('gender')
...
...     @button.buttonAndHandler('Cancel')
...     def handleCancel(self, action):
...         print('cancel')
```

```
>>> list(DerivedForm.fields.keys())
['name', 'gender']
>>> list(DerivedForm.buttons.keys())
['apply', 'cancel']
>>> DerivedForm.handlers
<Handlers
    [<Handler for <Button 'apply' 'Apply'>>,
     <Handler for <Button 'cancel' 'Cancel'>>]>
```

But this is pretty clumsy. Instead, the `form` module provides a helper method that will do the extending for you:

```
>>> class DerivedForm(BaseForm):
...     form.extends(BaseForm)
...
...     fields += field.Fields(IPerson).select('gender')
...
...     @button.buttonAndHandler('Cancel')
...     def handleCancel(self, action):
...         print('cancel')
```

```
>>> list(DerivedForm.fields.keys())
['name', 'gender']
>>> list(DerivedForm.buttons.keys())
['apply', 'cancel']
>>> DerivedForm.handlers
<Handlers
    [<Handler for <Button 'apply' 'Apply'>>,
     <Handler for <Button 'cancel' 'Cancel'>>]>
```

If you, for example do not want to extend the buttons, you can turn that off:

```
>>> class DerivedForm(BaseForm):
...     form.extends(BaseForm, ignoreButtons=True)
...
...     fields += field.Fields(IPerson).select('gender')
...
...     @button.buttonAndHandler('Cancel')
...     def handleCancel(self, action):
...         print('cancel')
```

```
>>> list(DerivedForm.fields.keys())
['name', 'gender']
>>> list(DerivedForm.buttons.keys())
['cancel']
```

(continues on next page)

(continued from previous page)

```
>>> DerivedForm.handlers
<Handlers
    [<Handler for <Button 'apply' 'Apply'>>,
     <Handler for <Button 'cancel' 'Cancel'>>]>
```

If you, for example do not want to extend the handlers, you can turn that off:

```
>>> class DerivedForm(BaseForm):
...     form.extends(BaseForm, ignoreHandlers=True)
...
...     fields += field.Fields(IPerson).select('gender')
...
...     @button.buttonAndHandler('Cancel')
...     def handleCancel(self, action):
...         print('cancel')
```

```
>>> list(DerivedForm.fields.keys())
['name', 'gender']
>>> list(DerivedForm.buttons.keys())
['apply', 'cancel']
>>> DerivedForm.handlers
<Handlers [<Handler for <Button 'cancel' 'Cancel'>>]>
```

1.1.20 Custom widget factories

Another important part of a form is that we can use custom widgets. We can do this in a form by defining a widget factory for a field. We can get the field from the fields collection e.g. `fields['foo']`. This means, we can define new widget factories by defining `fields['foo'].widgetFactory = MyWidget`. Let's show a sample and define a custom widget:

```
>>> from z3c.form.browser import text
>>> class MyWidget(text.TextWidget):
...     """My new widget."""
...     klass = 'MyCSS'
```

Now we can define a field widget factory:

```
>>> def MyFieldWidget(field, request):
...     """IFieldWidget factory for MyWidget."""
...     return widget.FieldWidget(field, MyWidget(request))
```

We register the `MyWidget` in a form like:

```
>>> class MyEditForm(form.EditForm):
...
...     fields = field.Fields(IPerson)
...     fields['name'].widgetFactory = MyFieldWidget
```

We can see that the custom widget gets used in the rendered form:

```
>>> myEdit = MyEditForm(root['srichter'], TestRequest())
>>> addTemplate(myEdit)
>>> myEdit.update()
>>> print(testing.render(myEdit, './/xmlns:input[@id="form-widgets-name"]'))
<input type="text" id="form-widgets-name"
       name="form.widgets.name" class="MyCSS required textline-field"
       value="Claudia Richter" />
```

1.1.21 Hidden fields

Another important part of a form is that we can generate hidden widgets. We can do this in a form by defining a widget mode. We can do this by override the setUpWidgets method.

```
>>> class HiddenFieldEditForm(form.EditForm):
...
...     fields = field.Fields(IPerson)
...     fields['name'].widgetFactory = MyFieldWidget
...
...     def updateWidgets(self):
...         super(HiddenFieldEditForm, self).updateWidgets()
...         self.widgets['age'].mode = interfaces.HIDDEN_MODE
```

We can see that the widget gets rendered as hidden:

```
>>> hiddenEdit = HiddenFieldEditForm(root['srichter'], TestRequest())
>>> addTemplate(hiddenEdit)
>>> hiddenEdit.update()
>>> print(testing.render(hiddenEdit, './/xmlns:input[@id="form-widgets-age"]'))
<input type="hidden" id="form-widgets-age"
       name="form.widgets.age" class="hidden-widget"
       value="29" />
```

1.1.22 Actions with Errors

Even though the data might be validated correctly, it sometimes happens that data turns out to be invalid while the action is executed. In those cases a special action execution error can be raised that wraps the original error.

```
>>> class PersonAddForm(form.AddForm):
...
...     fields = field.Fields(IPerson).select('id')
...
...     @button.buttonAndHandler('Check')
...     def handleCheck(self, action):
...         data, errors = self.extractData()
...         if data['id'] in self.getContent():
...             raise interfaces.WidgetActionExecutionError(
...                 'id', zope.interface.Invalid('Id already exists'))
```

In this case the action execution error is specific to a widget. The framework will attach a proper error view to the widget and the widget manager:

```
>>> request = TestRequest(form={
...     'form.widgets.id': 'srichter',
...     'form.buttons.check': 'Check'}
...     )
```

```
>>> addForm = PersonAddForm(root, request)
>>> addForm.update()
```

```
>>> addForm.widgets.errors
(<InvalidErrorViewSnippet for Invalid>,)
>>> addForm.widgets['id'].error
<InvalidErrorViewSnippet for Invalid>
>>> addForm.status
'There were some errors.'
```

If the error is non-widget specific, then we can simply use the generic action execution error:

```
>>> class PersonAddForm(form.AddForm):
...
...     fields = field.Fields(IPerson).select('id')
...
...     @button.buttonAndHandler('Check')
...     def handleCheck(self, action):
...         raise interfaces.ActionExecutionError(
...             zope.interface.Invalid('Some problem occurred.'))
```

Let's have a look at the result:

```
>>> addForm = PersonAddForm(root, request)
>>> addForm.update()
```

```
>>> addForm.widgets.errors
(<InvalidErrorViewSnippet for Invalid>,)
>>> addForm.status
'There were some errors.'
```

Note:

The action execution errors are connected to the form via an event listener called `handlerActionError`. This event listener listens for `IActionErrorEvent` events. If the event is called for an action associated with a form, the listener does its work as seen above. If the action is not coupled to a form, then event listener does nothing:

```
>>> from z3c.form import action
```

```
>>> cancel = action.Action(request, 'Cancel')
>>> event = action.ActionErrorOccurred(cancel, ValueError(3))
```

```
>>> form.handleActionError(event)
```

1.1.23 Applying Changes

When applying the data of a form to a content component, the function `applyChanges()` is called. It simply iterates through the fields of the form and uses the data managers to store the values. The output of the function is a list of changes:

```
>>> roger = Person('roger', 'Roger')
>>> roger
<Person 'Roger'>
```

```
>>> class BaseForm(form.Form):
...     fields = field.Fields(IPerson).select('name')
>>> myForm = BaseForm(roger, TestRequest())
```

```
>>> form.applyChanges(myForm, roger, {'name': 'Roger Ineichen'})
{<InterfaceClass builtins.IPerson>: ['name']}
```

```
>>> roger
<Person 'Roger Ineichen'>
```

When a field is missing from the data, it is simply skipped:

```
>>> form.applyChanges(myForm, roger, {})
{}
```

If the new and old value are identical, storing the data is skipped as well:

```
>>> form.applyChanges(myForm, roger, {'name': 'Roger Ineichen'})
{}
```

In some cases the data converter for a field-widget pair returns the `NOT_CHANGED` value. In this case, the field is skipped as well:

```
>>> form.applyChanges(myForm, roger, {'name': interfaces.NOT_CHANGED})
{}
```

```
>>> roger
<Person 'Roger Ineichen'>
```

1.1.24 Refreshing actions

Sometimes, it's useful to update actions again after executing them, because some conditions could have changed. For example, imagine we have a sequence edit form that has a delete button. We don't want to show delete button when the sequence is empty. The button condition would handle this, but what if the sequence becomes empty as a result of execution of the delete action that was available? In that case we want to refresh our actions to new conditions to make our delete button not visible anymore. The `refreshActions` form variable is intended to handle this case.

Let's create a simple form with an action that clears our context sequence.

```
>>> class SequenceForm(form.Form):
...
...     @button.buttonAndHandler('Empty', condition=lambda form:bool(form.context))
```

(continues on next page)

(continued from previous page)

```
...     def handleEmpty(self, action):
...         self.context[:] = []
...         self.refreshActions = True
```

First, let's illustrate simple cases, when no button is pressed. The button will be available when context is not empty.

```
>>> context = [1, 2, 3, 4]
>>> request = TestRequest()
>>> myForm = SequenceForm(context, request)
>>> myForm.update()
>>> addTemplate(myForm)
>>> print(testing.render(myForm, './/xmlns:div[@class="action"]'))
<div class="action">
  <input id="form-buttons-empty" name="form.buttons.empty" class="submit-widget button-
↪field" value="Empty" type="submit" />
</div>
...
```

The button will not be available when the context is empty.

```
>>> context = []
>>> request = TestRequest()
>>> myForm = SequenceForm(context, request)
>>> myForm.update()
>>> addTemplate(myForm)
>>> print(testing.render(myForm, './/xmlns:form'))
<form action=".">
</form>
```

Now, the most interesting case when context is not empty, but becomes empty as a result of pressing the "empty" button. We set the `refreshActions` flag in the action handler, so our actions should be updated to new conditions.

```
>>> context = [1, 2, 3, 4, 5]
>>> request = TestRequest(form={
...     'form.buttons.empty': 'Empty'}
...     )
>>> myForm = SequenceForm(context, request)
>>> myForm.update()
>>> addTemplate(myForm)
>>> print(testing.render(myForm, './/xmlns:form'))
<form action=".">
</form>
```

1.1.25 Integration tests

Identifying the different forms can be important if it comes to layout template lookup. Let's ensure that we support the right interfaces for the different forms.

Form

```
>>> from zope.interface.verify import verifyObject
>>> from z3c.form import interfaces
>>> obj = form.Form(None, None)
>>> verifyObject(interfaces.IForm, obj)
True
```

```
>>> interfaces.IForm.providedBy(obj)
True
```

```
>>> from z3c.form import interfaces
>>> interfaces.IDisplayForm.providedBy(obj)
False
```

```
>>> from z3c.form import interfaces
>>> interfaces.IEditForm.providedBy(obj)
False
```

```
>>> from z3c.form import interfaces
>>> interfaces.IAddForm.providedBy(obj)
False
```

DisplayForm

```
>>> from z3c.form import interfaces
>>> obj = form.DisplayForm(None, None)
>>> verifyObject(interfaces.IDisplayForm, obj)
True
```

```
>>> interfaces.IForm.providedBy(obj)
True
```

```
>>> from z3c.form import interfaces
>>> interfaces.IDisplayForm.providedBy(obj)
True
```

```
>>> from z3c.form import interfaces
>>> interfaces.IEditForm.providedBy(obj)
False
```

```
>>> from z3c.form import interfaces
>>> interfaces.IAddForm.providedBy(obj)
False
```

EditForm

```
>>> from z3c.form import interfaces
>>> obj = form.EditForm(None, None)
>>> verifyObject(interfaces.IEditForm, obj)
True
```

```
>>> interfaces.IForm.providedBy(obj)
True
```

```
>>> from z3c.form import interfaces
>>> interfaces.IDisplayForm.providedBy(obj)
False
```

```
>>> from z3c.form import interfaces
>>> interfaces.IEditForm.providedBy(obj)
True
```

```
>>> from z3c.form import interfaces
>>> interfaces.IAddForm.providedBy(obj)
False
```

AddForm

```
>>> from z3c.form import interfaces
>>> obj = form.AddForm(None, None)
>>> verifyObject(interfaces.IAddForm, obj)
True
```

```
>>> interfaces.IForm.providedBy(obj)
True
```

```
>>> from z3c.form import interfaces
>>> interfaces.IDisplayForm.providedBy(obj)
False
```

```
>>> from z3c.form import interfaces
>>> interfaces.IEditForm.providedBy(obj)
False
```

```
>>> from z3c.form import interfaces
>>> interfaces.IAddForm.providedBy(obj)
True
```

1.2 Group Forms

Group forms allow you to split up a form into several logical units without much overhead. To the parent form, groups should be only dealt with during coding and be transparent on the data extraction level.

For the examples to work, we have to bring up most of the form framework:

```
>>> from z3c.form import testing
>>> testing.setupFormDefaults()
```

So let's first define a complex content component that warrants setting up multiple groups:

```
>>> import zope.interface
>>> import zope.schema
```

```
>>> class IVehicleRegistration(zope.interface.Interface):
...     firstName = zope.schema.TextLine(title='First Name')
...     lastName = zope.schema.TextLine(title='Last Name')
...
...     license = zope.schema.TextLine(title='License')
...     address = zope.schema.TextLine(title='Address')
...
...     model = zope.schema.TextLine(title='Model')
...     make = zope.schema.TextLine(title='Make')
...     year = zope.schema.TextLine(title='Year')
```

```
>>> @zope.interface.implementer(IVehicleRegistration)
... class VehicleRegistration(object):
...
...     def __init__(self, **kw):
...         for name, value in kw.items():
...             setattr(self, name, value)
```

The schema above can be separated into basic, license, and car information, where the latter two will be placed into groups. First we create the two groups:

```
>>> from z3c.form import field, group
```

```
>>> class LicenseGroup(group.Group):
...     label = 'License'
...     fields = field.Fields(IVehicleRegistration).select(
...         'license', 'address')
```

```
>>> class CarGroup(group.Group):
...     label = 'Car'
...     fields = field.Fields(IVehicleRegistration).select(
...         'model', 'make', 'year')
```

Most of the group is setup like any other (sub)form. Additionally, you can specify a label, which is a human-readable string that can be used for layout purposes.

Let's now create an add form for the entire vehicle registration. In comparison to a regular add form, you only need to add the `GroupForm` as one of the base classes. The groups are specified in a simple tuple:

```
>>> import os
>>> from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile
>>> from z3c.form import form, tests
```

```
>>> class RegistrationAddForm(group.GroupForm, form.AddForm):
...     fields = field.Fields(IVehicleRegistration).select(
...         'firstName', 'lastName')
...     groups = (LicenseGroup, CarGroup)
...
...     template = ViewPageTemplateFile(
...         'simple_groupedit.pt', os.path.dirname(tests.__file__))
...
...     def create(self, data):
...         return VehicleRegistration(**data)
...
...     def add(self, object):
...         self.getContent()['obj1'] = object
...         return object
```

Note: The order of the base classes is very important here. The `GroupForm` class must be left of the `AddForm` class, because the `GroupForm` class overrides some methods of the `AddForm` class.

Now we can instantiate the form:

```
>>> request = testing.TestRequest()
```

```
>>> add = RegistrationAddForm(None, request)
>>> add.update()
```

After the form is updated the tuple of group classes is converted to group instances:

```
>>> add.groups
(<LicenseGroup object at ...>, <CarGroup object at ...>)
```

If we happen to update the add form again, the groups that have already been converted to instances ares skipped.

```
>>> add.update()
>>> add.groups
(<LicenseGroup object at ...>, <CarGroup object at ...>)
```

We can now render the form:

```
>>> print(add.render())
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-firstName">First Name</label>
        <input type="text" id="form-widgets-firstName"
               name="form.widgets.firstName"
               class="text-widget required textline-field"
```

(continues on next page)

(continued from previous page)

```
          value="" />
    </div>
    <div class="row">
      <label for="form-widgets-lastName">Last Name</label>
      <input type="text" id="form-widgets-lastName"
             name="form.widgets.lastName"
             class="text-widget required textline-field"
             value="" />
    </div>
    <fieldset>
      <legend>License</legend>
      <div class="row">
        <label for="form-widgets-license">License</label>
        <input type="text" id="form-widgets-license"
               name="form.widgets.license"
               class="text-widget required textline-field"
               value="" />
      </div>
      <div class="row">
        <label for="form-widgets-address">Address</label>
        <input type="text" id="form-widgets-address"
               name="form.widgets.address"
               class="text-widget required textline-field"
               value="" />
      </div>
    </fieldset>
    <fieldset>
      <legend>Car</legend>
      <div class="row">
        <label for="form-widgets-model">Model</label>
        <input type="text" id="form-widgets-model"
               name="form.widgets.model"
               class="text-widget required textline-field"
               value="" />
      </div>
      <div class="row">
        <label for="form-widgets-make">Make</label>
        <input type="text" id="form-widgets-make"
               name="form.widgets.make"
               class="text-widget required textline-field"
               value="" />
      </div>
      <div class="row">
        <label for="form-widgets-year">Year</label>
        <input type="text" id="form-widgets-year"
               name="form.widgets.year"
               class="text-widget required textline-field"
               value="" />
      </div>
    </fieldset>
    <div class="action">
      <input type="submit" id="form-buttons-add"
```

(continues on next page)

(continued from previous page)

```
              name="form.buttons.add" class="submit-widget button-field"
              value="Add" />
      </div>
    </form>
  </body>
</html>
```

1.2.1 Registering a custom event handler for the DataExtractedEvent

```
>>> data_extracted_eventlog = []
>>> from z3c.form.events import DataExtractedEvent
>>> @zope.component.adapter(DataExtractedEvent)
... def data_extracted_logger(event):
...     data_extracted_eventlog.append(event)
>>> zope.component.provideHandler(data_extracted_logger)
```

Let's now submit the form, but forgetting to enter the address:

```
>>> request = testing.TestRequest(form={
...     'form.widgets.firstName': 'Stephan',
...     'form.widgets.lastName': 'Richter',
...     'form.widgets.license': 'MA 40387',
...     'form.widgets.model': 'BMW',
...     'form.widgets.make': '325',
...     'form.widgets.year': '2005',
...     'form.buttons.add': 'Add'
...     })
```

```
>>> add = RegistrationAddForm(None, request)
>>> add.update()
>>> print(testing.render(add, './/xmlns:i'))
<i >There were some errors.</i>
...
```

```
>>> print(testing.render(add, './/xmlns:fieldset[1]/xmlns:ul'))
<ul >
  <li>
    Address: <div class="error">Required input is missing.</div>
  </li>
</ul>
...
```

As you can see, the template is clever enough to just report the errors at the top of the form, but still report the actual problem within the group.

Check, if DataExtractedEvent was thrown:

```
>>> len(data_extracted_eventlog) > 0
True
```

So what happens, if errors happen inside and outside a group?

```
>>> request = testing.TestRequest(form={
...     'form.widgets.firstName': 'Stephan',
...     'form.widgets.license': 'MA 40387',
...     'form.widgets.model': 'BMW',
...     'form.widgets.make': '325',
...     'form.widgets.year': '2005',
...     'form.buttons.add': 'Add'
...     })
```

```
>>> add = RegistrationAddForm(None, request)
>>> add.update()
>>> print(testing.render(add, './/xmlns:i'))
<i >There were some errors.</i>
...
```

```
>>> print(testing.render(add, './/xmlns:ul'))
<ul >
  <li>
  Last Name:
    <div class="error">Required input is missing.</div>
  </li>
</ul>
...
<ul >
  <li>
  Address:
    <div class="error">Required input is missing.</div>
  </li>
</ul>
...
```

```
>>> print(testing.render(add, './/xmlns:fieldset[1]/xmlns:ul'))
<ul >
  <li>
    Address: <div class="error">Required input is missing.</div>
  </li>
</ul>
...
```

Let's now successfully complete the add form.

```
>>> from zope.container import btree
>>> context = btree.BTreeContainer()
```

```
>>> request = testing.TestRequest(form={
...     'form.widgets.firstName': 'Stephan',
...     'form.widgets.lastName': 'Richter',
...     'form.widgets.license': 'MA 40387',
...     'form.widgets.address': '10 Main St, Maynard, MA',
...     'form.widgets.model': 'BMW',
...     'form.widgets.make': '325',
...     'form.widgets.year': '2005',
```

(continues on next page)

(continued from previous page)

```
...     'form.buttons.add': 'Add'
...     })
```

```
>>> add = RegistrationAddForm(context, request)
>>> add.update()
```

The object is now added to the container and all attributes should be set:

```
>>> reg = context['obj1']
>>> reg.firstName
'Stephan'
>>> reg.lastName
'Richter'
>>> reg.license
'MA 40387'
>>> reg.address
'10 Main St, Maynard, MA'
>>> reg.model
'BMW'
>>> reg.make
'325'
>>> reg.year
'2005'
```

Let's now have a look at an edit form for the vehicle registration:

```
>>> class RegistrationEditForm(group.GroupForm, form.EditForm):
...     fields = field.Fields(IVehicleRegistration).select(
...         'firstName', 'lastName')
...     groups = (LicenseGroup, CarGroup)
...
...     template = ViewPageTemplateFile(
...         'simple_groupedit.pt', os.path.dirname(tests.__file__))
```

```
>>> request = testing.TestRequest()
```

```
>>> edit = RegistrationEditForm(reg, request)
>>> edit.update()
```

After updating the form, we can render the HTML:

```
>>> print(edit.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-firstName">First Name</label>
        <input type="text" id="form-widgets-firstName"
               name="form.widgets.firstName"
               class="text-widget required textline-field"
               value="Stephan" />
      </div>
```

(continues on next page)

(continued from previous page)

```
      <div class="row">
        <label for="form-widgets-lastName">Last Name</label>
        <input type="text" id="form-widgets-lastName"
               name="form.widgets.lastName"
               class="text-widget required textline-field"
               value="Richter" />
       </div>
      <fieldset>
        <legend>License</legend>
        <div class="row">
          <label for="form-widgets-license">License</label>
          <input type="text" id="form-widgets-license"
                 name="form.widgets.license"
                 class="text-widget required textline-field"
                 value="MA 40387" />
        </div>
        <div class="row">
          <label for="form-widgets-address">Address</label>
          <input type="text" id="form-widgets-address"
                 name="form.widgets.address"
                 class="text-widget required textline-field"
                 value="10 Main St, Maynard, MA" />
        </div>
      </fieldset>
      <fieldset>
        <legend>Car</legend>
        <div class="row">
          <label for="form-widgets-model">Model</label>
          <input type="text" id="form-widgets-model"
                 name="form.widgets.model"
                 class="text-widget required textline-field"
                 value="BMW" />
        </div>
        <div class="row">
          <label for="form-widgets-make">Make</label>
          <input type="text" id="form-widgets-make"
                 name="form.widgets.make"
                 class="text-widget required textline-field"
                 value="325" />
        </div>
        <div class="row">
          <label for="form-widgets-year">Year</label>
          <input type="text" id="form-widgets-year"
                 name="form.widgets.year"
                 class="text-widget required textline-field"
                 value="2005" />
        </div>
      </fieldset>
      <div class="action">
        <input type="submit" id="form-buttons-apply"
               name="form.buttons.apply" class="submit-widget button-field"
               value="Apply" />
```

(continues on next page)

(continued from previous page)

```
      </div>
    </form>
  </body>
</html>
```

The behavior when an error occurs is identical to that of the add form:

```
>>> request = testing.TestRequest(form={
...     'form.widgets.firstName': 'Stephan',
...     'form.widgets.lastName': 'Richter',
...     'form.widgets.license': 'MA 40387',
...     'form.widgets.model': 'BMW',
...     'form.widgets.make': '325',
...     'form.widgets.year': '2005',
...     'form.buttons.apply': 'Apply'
...     })
```

```
>>> edit = RegistrationEditForm(reg, request)
>>> edit.update()
>>> print(testing.render(edit, './/xmlns:i'))
<i >There were some errors.</i>
...
```

```
>>> print(testing.render(edit, './/xmlns:ul'))
<ul >
  <li>
  Address:
    <div class="error">Required input is missing.</div>
  </li>
</ul>
...
```

```
>>> print(testing.render(edit, './/xmlns:fieldset/xmlns:ul'))
<ul >
  <li>
    Address: <div class="error">Required input is missing.</div>
  </li>
</ul>
...
```

When an edit form with groups is successfully committed, a detailed object-modified event is sent out telling the system about the changes. To see the error, let's create an event subscriber for object-modified events:

```
>>> eventlog = []
>>> import zope.lifecycleevent
>>> @zope.component.adapter(zope.lifecycleevent.ObjectModifiedEvent)
... def logEvent(event):
...     eventlog.append(event)
>>> zope.component.provideHandler(logEvent)
```

Let's now complete the form successfully:

```
>>> request = testing.TestRequest(form={
...     'form.widgets.firstName': 'Stephan',
...     'form.widgets.lastName': 'Richter',
...     'form.widgets.license': 'MA 4038765',
...     'form.widgets.address': '11 Main St, Maynard, MA',
...     'form.widgets.model': 'Ford',
...     'form.widgets.make': 'F150',
...     'form.widgets.year': '2006',
...     'form.buttons.apply': 'Apply'
...     })
```

```
>>> edit = RegistrationEditForm(reg, request)
>>> edit.update()
```

The success message will be shown on the form, ...

```
>>> print(testing.render(edit, './/xmlns:i'))
<i >Data successfully updated.</i>
...
```

and the data are correctly updated:

```
>>> reg.firstName
'Stephan'
>>> reg.lastName
'Richter'
>>> reg.license
'MA 4038765'
>>> reg.address
'11 Main St, Maynard, MA'
>>> reg.model
'Ford'
>>> reg.make
'F150'
>>> reg.year
'2006'
```

Let's look at the event:

```
>>> event = eventlog[-1]
>>> event
<zope...ObjectModifiedEvent object at ...>
```

The event's description contains the changed Interface and the names of all changed fields, even if they where in different groups:

```
>>> attrs = event.descriptions[0]
>>> attrs.interface
<InterfaceClass builtins.IVehicleRegistration>
>>> attrs.attributes
('license', 'address', 'model', 'make', 'year')
```

1.2.2 Group form as instance

It is also possible to use group instances in forms. Let's setup our previous form and assing a group instance:

```
>>> class RegistrationEditForm(group.GroupForm, form.EditForm):
...     fields = field.Fields(IVehicleRegistration).select(
...         'firstName', 'lastName')
...
...     template = ViewPageTemplateFile(
...         'simple_groupedit.pt', os.path.dirname(tests.__file__))
```

```
>>> request = testing.TestRequest()
```

```
>>> edit = RegistrationEditForm(reg, request)
```

Instanciate the form and use a group class and a group instance:

```
>>> carGroupInstance = CarGroup(edit.context, request, edit)
>>> edit.groups = (LicenseGroup, carGroupInstance)
>>> edit.update()
>>> print(edit.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-firstName">First Name</label>
        <input id="form-widgets-firstName"
               name="form.widgets.firstName"
               class="text-widget required textline-field"
               value="Stephan" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-lastName">Last Name</label>
        <input id="form-widgets-lastName"
               name="form.widgets.lastName"
               class="text-widget required textline-field"
               value="Richter" type="text" />
      </div>
      <fieldset>
        <legend>License</legend>
      <div class="row">
        <label for="form-widgets-license">License</label>
        <input id="form-widgets-license"
               name="form.widgets.license"
               class="text-widget required textline-field"
               value="MA 4038765" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-address">Address</label>
        <input id="form-widgets-address"
               name="form.widgets.address"
               class="text-widget required textline-field"
               value="11 Main St, Maynard, MA" type="text" />
```

(continues on next page)

(continued from previous page)

```
      </div>
      </fieldset>
      <fieldset>
        <legend>Car</legend>
      <div class="row">
        <label for="form-widgets-model">Model</label>
        <input id="form-widgets-model" name="form.widgets.model"
               class="text-widget required textline-field"
               value="Ford" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-make">Make</label>
        <input id="form-widgets-make" name="form.widgets.make"
               class="text-widget required textline-field"
               value="F150" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-year">Year</label>
        <input id="form-widgets-year" name="form.widgets.year"
               class="text-widget required textline-field"
               value="2006" type="text" />
          </div>
      </fieldset>
      <div class="action">
        <input id="form-buttons-apply" name="form.buttons.apply"
               class="submit-widget button-field" value="Apply"
               type="submit" />
      </div>
    </form>
  </body>
</html>
```

1.2.3 Groups with Different Content

You can customize the content for a group by overriding a group's `getContent` method. This is a very easy way to get around not having object widgets. For example, suppose we want to maintain the vehicle owner's information in a separate class than the vehicle. We might have an `IVehicleOwner` interface like so.

```
>>> class IVehicleOwner(zope.interface.Interface):
...     firstName = zope.schema.TextLine(title='First Name')
...     lastName = zope.schema.TextLine(title='Last Name')
```

Then out `IVehicleRegistration` interface would include an object field for the owner instead of the `firstName` and `lastName` fields.

```
>>> class IVehicleRegistration(zope.interface.Interface):
...     owner = zope.schema.Object(title='Owner', schema=IVehicleOwner)
...
...     license = zope.schema.TextLine(title='License')
...     address = zope.schema.TextLine(title='Address')
...
```

(continues on next page)

(continued from previous page)

```
...     model = zope.schema.TextLine(title='Model')
...     make = zope.schema.TextLine(title='Make')
...     year = zope.schema.TextLine(title='Year')
```

Now let's create simple implementations of these two interfaces.

```
>>> @zope.interface.implementer(IVehicleOwner)
... class VehicleOwner(object):
...
...     def __init__(self, **kw):
...         for name, value in kw.items():
...             setattr(self, name, value)
```

```
>>> @zope.interface.implementer(IVehicleRegistration)
... class VehicleRegistration(object):
...
...     def __init__(self, **kw):
...         for name, value in kw.items():
...             setattr(self, name, value)
```

Now we can create a group just for the owner with its own `getContent` method that simply returns the `owner` object field of the `VehicleRegistration` instance.

```
>>> class OwnerGroup(group.Group):
...     label = 'Owner'
...     fields = field.Fields(IVehicleOwner, prefix='owner')
...
...     def getContent(self):
...         return self.context.owner
```

When we create an Edit form for example, we should omit the `owner` field which is taken care of with the group.

```
>>> class RegistrationEditForm(group.GroupForm, form.EditForm):
...     fields = field.Fields(IVehicleRegistration).omit(
...         'owner')
...     groups = (OwnerGroup,)
...
...     template = ViewPageTemplateFile(
...         'simple_groupedit.pt', os.path.dirname(tests.__file__))
```

```
>>> reg = VehicleRegistration(
...                license='MA 40387',
...                address='10 Main St, Maynard, MA',
...                model='BMW',
...                make='325',
...                year='2005',
...                owner=VehicleOwner(firstName='Stephan',
...                                   lastName='Richter'))
>>> request = testing.TestRequest()
```

```
>>> edit = RegistrationEditForm(reg, request)
>>> edit.update()
```

When we render the form, the group appears as we would expect but with the `owner` prefix for the fields.

```
>>> print(edit.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-license">License</label>
        <input type="text" id="form-widgets-license"
               name="form.widgets.license"
               class="text-widget required textline-field"
               value="MA 40387" />
      </div>
      <div class="row">
        <label for="form-widgets-address">Address</label>
        <input type="text" id="form-widgets-address"
               name="form.widgets.address"
               class="text-widget required textline-field"
               value="10 Main St, Maynard, MA" />
      </div>
      <div class="row">
        <label for="form-widgets-model">Model</label>
        <input type="text" id="form-widgets-model"
               name="form.widgets.model"
               class="text-widget required textline-field"
               value="BMW" />
      </div>
      <div class="row">
        <label for="form-widgets-make">Make</label>
        <input type="text" id="form-widgets-make"
               name="form.widgets.make"
               class="text-widget required textline-field"
               value="325" />
      </div>
      <div class="row">
        <label for="form-widgets-year">Year</label>
        <input type="text" id="form-widgets-year"
               name="form.widgets.year"
               class="text-widget required textline-field" value="2005" />
      </div>
      <fieldset>
        <legend>Owner</legend>
        <div class="row">
          <label for="form-widgets-owner-firstName">First Name</label>
          <input type="text" id="form-widgets-owner-firstName"
                 name="form.widgets.owner.firstName"
                 class="text-widget required textline-field"
                 value="Stephan" />
        </div>
        <div class="row">
          <label for="form-widgets-owner-lastName">Last Name</label>
          <input type="text" id="form-widgets-owner-lastName"
                 name="form.widgets.owner.lastName"
```

(continues on next page)

(continued from previous page)

```
                class="text-widget required textline-field"
                value="Richter" />
        </div>
      </fieldset>
      <div class="action">
        <input type="submit" id="form-buttons-apply"
               name="form.buttons.apply"
               class="submit-widget button-field" value="Apply" />
      </div>
    </form>
  </body>
</html>
```

Now let's try and edit the owner. For example, suppose that Stephan Richter gave his BMW to Paul Carduner because he is such a nice guy.

```
>>> request = testing.TestRequest(form={
...     'form.widgets.owner.firstName': 'Paul',
...     'form.widgets.owner.lastName': 'Carduner',
...     'form.widgets.license': 'MA 4038765',
...     'form.widgets.address': 'Berkeley',
...     'form.widgets.model': 'BMW',
...     'form.widgets.make': '325',
...     'form.widgets.year': '2005',
...     'form.buttons.apply': 'Apply'
...     })
>>> edit = RegistrationEditForm(reg, request)
>>> edit.update()
```

We'll see if everything worked on the form side.

```
>>> print(testing.render(edit, './/xmlns:i'))
<i >Data successfully updated.</i>
...
```

Now the owner object should have updated fields.

```
>>> reg.owner.firstName
'Paul'
>>> reg.owner.lastName
'Carduner'
>>> reg.license
'MA 4038765'
>>> reg.address
'Berkeley'
>>> reg.model
'BMW'
>>> reg.make
'325'
>>> reg.year
'2005'
```

1.2.4 Nested Groups

The group can contains groups. Let's adapt the previous RegistrationEditForm:

```
>>> class OwnerGroup(group.Group):
...     label = 'Owner'
...     fields = field.Fields(IVehicleOwner, prefix='owner')
...
...     def getContent(self):
...         return self.context.owner
```

```
>>> class VehicleRegistrationGroup(group.Group):
...     label = 'Registration'
...     fields = field.Fields(IVehicleRegistration).omit(
...         'owner')
...     groups = (OwnerGroup,)
...
...     template = ViewPageTemplateFile(
...         'simple_groupedit.pt', os.path.dirname(tests.__file__))
```

```
>>> class RegistrationEditForm(group.GroupForm, form.EditForm):
...     groups = (VehicleRegistrationGroup,)
...
...     template = ViewPageTemplateFile(
...         'simple_nested_groupedit.pt', os.path.dirname(tests.__file__))
```

```
>>> reg = VehicleRegistration(
...                license='MA 40387',
...                address='10 Main St, Maynard, MA',
...                model='BMW',
...                make='325',
...                year='2005',
...                owner=VehicleOwner(firstName='Stephan',
...                                   lastName='Richter'))
>>> request = testing.TestRequest()
```

```
>>> edit = RegistrationEditForm(reg, request)
>>> edit.update()
```

Now let's try and edit the owner. For example, suppose that Stephan Richter gave his BMW to Paul Carduner because he is such a nice guy.

```
>>> request = testing.TestRequest(form={
...     'form.widgets.owner.firstName': 'Paul',
...     'form.widgets.owner.lastName': 'Carduner',
...     'form.widgets.license': 'MA 4038765',
...     'form.widgets.address': 'Berkeley',
...     'form.widgets.model': 'BMW',
...     'form.widgets.make': '325',
...     'form.widgets.year': '2005',
...     'form.buttons.apply': 'Apply'
...     })
```

(continues on next page)

(continued from previous page)

```
>>> edit = RegistrationEditForm(reg, request)
>>> edit.update()
```

We'll see if everything worked on the form side.

```
>>> print(testing.render(edit, './/xmlns:i'))
<i >Data successfully updated.</i>
...
```

Now the owner object should have updated fields.

```
>>> reg.owner.firstName
'Paul'
>>> reg.owner.lastName
'Carduner'
>>> reg.license
'MA 4038765'
>>> reg.address
'Berkeley'
>>> reg.model
'BMW'
>>> reg.make
'325'
>>> reg.year
'2005'
```

So what happens, if errors happen inside a nested group? Let's use an empty invalid object for the test missing input errors:

```
>>> reg = VehicleRegistration(owner=VehicleOwner())
```

```
>>> request = testing.TestRequest(form={
...     'form.widgets.owner.firstName': '',
...     'form.widgets.owner.lastName': '',
...     'form.widgets.license': '',
...     'form.widgets.address': '',
...     'form.widgets.model': '',
...     'form.widgets.make': '',
...     'form.widgets.year': '',
...     'form.buttons.apply': 'Apply'
...     })
```

```
>>> edit = RegistrationEditForm(reg, request)
>>> edit.update()
>>> data, errors = edit.extractData()
>>> print(testing.render(edit, './/xmlns:i'))
<i >There were some errors.</i>
...
```

```
>>> print(testing.render(edit, './/xmlns:fieldset/xmlns:ul'))
<ul >
  <li>
```

(continues on next page)

(continued from previous page)

```
  License:
    <div class="error">Required input is missing.</div>
  </li>
  <li>
  Address:
    <div class="error">Required input is missing.</div>
  </li>
  <li>
  Model:
    <div class="error">Required input is missing.</div>
  </li>
  <li>
  Make:
    <div class="error">Required input is missing.</div>
  </li>
  <li>
  Year:
    <div class="error">Required input is missing.</div>
  </li>
</ul>
...
<ul >
  <li>
  First Name:
    <div class="error">Required input is missing.</div>
  </li>
  <li>
  Last Name:
    <div class="error">Required input is missing.</div>
  </li>
</ul>
...
```

1.2.5 Group instance in nested group

Let's also test if the Group class can handle group objects as instances:

```
>>> reg = VehicleRegistration(
...                license='MA 40387',
...                address='10 Main St, Maynard, MA',
...                model='BMW',
...                make='325',
...                year='2005',
...                owner=VehicleOwner(firstName='Stephan',
...                                   lastName='Richter'))
>>> request = testing.TestRequest()
```

```
>>> edit = RegistrationEditForm(reg, request)
>>> vrg = VehicleRegistrationGroup(edit.context, request, edit)
>>> ownerGroup = OwnerGroup(edit.context, request, edit)
```

Now build the group instance object chain:

```
>>> vrg.groups = (ownerGroup,)
>>> edit.groups = (vrg,)
```

Also use refreshActions which is not needed but will make coverage this additional line of code in the update method:

```
>>> edit.refreshActions = True
```

Update and render:

```
>>> edit.update()
>>> print(edit.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <fieldset>
        <legend>Registration</legend>
      <div class="row">
        <label for="form-widgets-license">License</label>
        <input id="form-widgets-license"
               name="form.widgets.license"
               class="text-widget required textline-field"
               value="MA 40387" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-address">Address</label>
        <input id="form-widgets-address"
               name="form.widgets.address"
               class="text-widget required textline-field"
               value="10 Main St, Maynard, MA" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-model">Model</label>
        <input id="form-widgets-model" name="form.widgets.model"
               class="text-widget required textline-field"
               value="BMW" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-make">Make</label>
        <input id="form-widgets-make" name="form.widgets.make"
               class="text-widget required textline-field"
               value="325" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-year">Year</label>
        <input id="form-widgets-year" name="form.widgets.year"
               class="text-widget required textline-field"
               value="2005" type="text" />
      </div>
        <fieldset>
        <legend>Owner</legend>
      <div class="row">
        <label for="form-widgets-owner-firstName">First Name</label>
```

(continues on next page)

(continued from previous page)

```
        <input id="form-widgets-owner-firstName"
               name="form.widgets.owner.firstName"
               class="text-widget required textline-field"
               value="Stephan" type="text" />
      </div>
      <div class="row">
        <label for="form-widgets-owner-lastName">Last Name</label>
        <input id="form-widgets-owner-lastName"
               name="form.widgets.owner.lastName"
               class="text-widget required textline-field"
               value="Richter" type="text" />
      </div>
      </fieldset>
      </fieldset>
      <div class="action">
        <input id="form-buttons-apply" name="form.buttons.apply"
               class="submit-widget button-field" value="Apply"
               type="submit" />
        </div>
    </form>
  </body>
</html>
```

Now test the error handling if just one missing value is given in a group:

```
>>> request = testing.TestRequest(form={
...      'form.widgets.owner.firstName': 'Paul',
...      'form.widgets.owner.lastName': '',
...      'form.widgets.license': 'MA 4038765',
...      'form.widgets.address': 'Berkeley',
...      'form.widgets.model': 'BMW',
...      'form.widgets.make': '325',
...      'form.widgets.year': '2005',
...      'form.buttons.apply': 'Apply'
...      })
```

```
>>> edit = RegistrationEditForm(reg, request)
>>> vrg = VehicleRegistrationGroup(edit.context, request, edit)
>>> ownerGroup = OwnerGroup(edit.context, request, edit)
>>> vrg.groups = (ownerGroup,)
>>> edit.groups = (vrg,)
```

```
>>> edit.update()
>>> data, errors = edit.extractData()
>>> print(testing.render(edit, './/xmlns:i'))
<i >There were some errors.</i>
...
```

```
>>> print(testing.render(edit, './/xmlns:fieldset/xmlns:ul'))
<ul >
  <li>
```

(continues on next page)

(continued from previous page)

```
  Last Name:
    <div class="error">Required input is missing.</div>
  </li>
</ul>
...
```

Just check whether we fully support the interface:

```
>>> from z3c.form import interfaces
>>> from zope.interface.verify import verifyClass
>>> verifyClass(interfaces.IGroup, group.Group)
True
```

1.3 Sub-Forms

Traditionally, the Zope community talks about sub-forms in a generic manner without defining their purpose, restrictions and assumptions. When we initially talked about sub-forms for this package, we quickly noticed that there are several classes of sub-forms with different goals.

Of course, we need to setup our defaults for this demonstration as well:

```
>>> from z3c.form import testing
>>> testing.setupFormDefaults()
```

1.3.1 Class I: The form within the form

This class of sub-forms provides a complete form within a form, including its own actions. When an action of the sub-form is submitted, the parent form usually does not interact with that action at all. The same is true for the reverse; when an action of the parent form is submitted, the sub-form does not react.

A classic example for this type of sub-form is uploading an image. The subform allows uploading a file and once the file is uploaded the image is shown as well as a "Delete"/"Clear" button. The sub-form will store the image in the session and when the main form is submitted it looks in the session for the image.

This scenario was well supported in `zope.formlib` and also does not require special support in `z3c.form`. Let me show you, how this can be done.

In this example, we would like to describe a car and its owner:

```
>>> import zope.interface
>>> import zope.schema
```

```
>>> class IOwner(zope.interface.Interface):
...     name = zope.schema.TextLine(title='Name')
...     license = zope.schema.TextLine(title='License')
```

```
>>> class ICar(zope.interface.Interface):
...     model = zope.schema.TextLine(title='Model')
...     make = zope.schema.TextLine(title='Make')
...     owner = zope.schema.Object(title='Owner', schema=IOwner)
```

Let's now implement the two interfaces and create instances, so that we can create edit forms for it:

```
>>> @zope.interface.implementer(IOwner)
... class Owner(object):
...     def __init__(self, name, license):
...         self.name = name
...         self.license = license
```

```
>>> @zope.interface.implementer(ICar)
... class Car(object):
...     def __init__(self, model, make, owner):
...         self.model = model
...         self.make = make
...         self.owner = owner
```

```
>>> me = Owner('Stephan Richter', 'MA-1231FW97')
>>> mycar = Car('Nissan', 'Sentra', me)
```

We define the owner sub-form as we would any other form. The only difference is the template, which should not render a form-tag:

```
>>> import os
>>> from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile
>>> from z3c.form import form, field, tests
```

```
>>> templatePath = os.path.dirname(tests.__file__)
```

```
>>> class OwnerForm(form.EditForm):
...     template = ViewPageTemplateFile(
...         'simple_owneredit.pt', templatePath)
...     fields = field.Fields(IOwner)
...     prefix = 'owner'
```

Next we define the car form, which has the owner form as a sub-form. The car form also needs a special template, since it needs to render the sub-form at some point. For the simplicity of this example, I have duplicated a lot of template code here, but you can use your favorite template techniques, such as METAL macros, viewlets, or pagelets to make better reuse of some code.

```
>>> class CarForm(form.EditForm):
...     fields = field.Fields(ICar).select('model', 'make')
...     template = ViewPageTemplateFile(
...         'simple_caredit.pt', templatePath)
...     prefix = 'car'
...     def update(self):
...         self.owner = OwnerForm(self.context.owner, self.request)
...         self.owner.update()
...         super(CarForm, self).update()
```

Let's now instantiate the form and render it:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> carForm = CarForm(mycar, request)
>>> carForm.update()
>>> print(carForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="car-widgets-model">Model</label>
        <input type="text" id="car-widgets-model" name="car.widgets.model"
               class="text-widget required textline-field" value="Nissan" />
      </div>
      <div class="row">
        <label for="car-widgets-make">Make</label>
        <input type="text" id="car-widgets-make" name="car.widgets.make"
               class="text-widget required textline-field" value="Sentra" />
      </div>
      <fieldset>
        <legend>Owner</legend>
        <div class="row">
          <label for="owner-widgets-name">Name</label>
          <input type="text" id="owner-widgets-name" name="owner.widgets.name"
                 class="text-widget required textline-field"
                 value="Stephan Richter" />
        </div>
        <div class="row">
          <label for="owner-widgets-license">License</label>
          <input type="text" id="owner-widgets-license"
                 name="owner.widgets.license"
                 class="text-widget required textline-field"
                 value="MA-1231FW97" />
        </div>
        <div class="action">
          <input type="submit" id="owner-buttons-apply"
                 name="owner.buttons.apply"
                 class="submit-widget button-field"
                 value="Apply" />
        </div>
      </fieldset>
      <div class="action">
        <input type="submit" id="car-buttons-apply"
               name="car.buttons.apply"
               class="submit-widget button-field"
               value="Apply" />
      </div>
    </form>
  </body>
</html>
```

I can now submit the owner form, which should not submit any car changes I might have made in the form:

```
>>> request = TestRequest(form={
...     'car.widgets.model': 'BMW',
...     'car.widgets.make': '325',
```

(continues on next page)

(continued from previous page)

```
...     'owner.widgets.name': 'Stephan Richter',
...     'owner.widgets.license': 'MA-97097A87',
...     'owner.buttons.apply': 'Apply'
...     })
```

```
>>> carForm = CarForm(mycar, request)
>>> carForm.update()
```

```
>>> mycar.model
'Nissan'
>>> mycar.make
'Sentra'
```

```
>>> me.name
'Stephan Richter'
>>> me.license
'MA-97097A87'
```

Also, the form should say that the data of the owner has changed:

```
>>> print(carForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="car-widgets-model">Model</label>
        <input type="text" id="car-widgets-model"
               name="car.widgets.model"
               class="text-widget required textline-field"
               value="BMW" />
      </div>
      <div class="row">
        <label for="car-widgets-make">Make</label>
        <input type="text" id="car-widgets-make"
               name="car.widgets.make"
               class="text-widget required textline-field"
               value="325" />
      </div>
      <fieldset>
        <legend>Owner</legend>
        <i>Data successfully updated.</i>
        <div class="row">
          <label for="owner-widgets-name">Name</label>
          <input type="text" id="owner-widgets-name"
                 name="owner.widgets.name"
                 class="text-widget required textline-field"
                 value="Stephan Richter" />
        </div>
        <div class="row">
          <label for="owner-widgets-license">License</label>
          <input type="text" id="owner-widgets-license"
```

(continues on next page)

(continued from previous page)

```
                name="owner.widgets.license"
                class="text-widget required textline-field"
                value="MA-97097A87" />
        </div>
        <div class="action">
          <input type="submit" id="owner-buttons-apply"
                name="owner.buttons.apply"
                class="submit-widget button-field"
                value="Apply" />
        </div>
      </fieldset>
      <div class="action">
        <input type="submit" id="car-buttons-apply"
               name="car.buttons.apply"
               class="submit-widget button-field"
               value="Apply" />
      </div>
    </form>
  </body>
</html>
```

The same is true the other way around as well. Submitting the overall form does not submit the owner form:

```
>>> request = TestRequest(form={
...     'car.widgets.model': 'BMW',
...     'car.widgets.make': '325',
...     'car.buttons.apply': 'Apply',
...     'owner.widgets.name': 'Claudia Richter',
...     'owner.widgets.license': 'MA-123403S2',
...     })
```

```
>>> carForm = CarForm(mycar, request)
>>> carForm.update()
```

```
>>> mycar.model
'BMW'
>>> mycar.make
'325'
```

```
>>> me.name
'Stephan Richter'
>>> me.license
'MA-97097A87'
```

1.3.2 Class II: The logical unit

In this class of sub-forms, a sub-form is often just a collection of widgets without any actions. Instead, the sub-form must be able to react to the actions of the parent form. A good example of those types of sub-forms is actually the example I chose above.

So let's redevelop our example above in a way that the owner sub-form is just a logical unit that shares the action with its parent form. Initially, the example does not look very different, except that we use `EditSubForm` as a base class:

```
>>> from z3c.form import subform
```

```
>>> class OwnerForm(subform.EditSubForm):
...     template = ViewPageTemplateFile(
...         'simple_subedit.pt', templatePath)
...     fields = field.Fields(IOwner)
...     prefix = 'owner'
```

The main form also is pretty much the same, except that a subform takes three constructor arguments, the last one being the parent form:

```
>>> class CarForm(form.EditForm):
...     fields = field.Fields(ICar).select('model', 'make')
...     template = ViewPageTemplateFile(
...         'simple_caredit.pt', templatePath)
...     prefix = 'car'
...
...     def update(self):
...         super(CarForm, self).update()
...         self.owner = OwnerForm(self.context.owner, self.request, self)
...         self.owner.update()
```

Rendering the form works as before:

```
>>> request = TestRequest()
>>> carForm = CarForm(mycar, request)
>>> carForm.update()
>>> print(carForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="car-widgets-model">Model</label>
        <input type="text" id="car-widgets-model"
               name="car.widgets.model"
               class="text-widget required textline-field"
               value="BMW" />
      </div>
      <div class="row">
        <label for="car-widgets-make">Make</label>
        <input type="text" id="car-widgets-make"
               name="car.widgets.make"
               class="text-widget required textline-field"
               value="325" />
      </div>
```

(continues on next page)

(continued from previous page)

```
      <fieldset>
        <legend>Owner</legend>
        <div class="row">
          <label for="owner-widgets-name">Name</label>
          <input type="text" id="owner-widgets-name"
                 name="owner.widgets.name"
                 class="text-widget required textline-field"
                 value="Stephan Richter" />
        </div>
        <div class="row">
          <label for="owner-widgets-license">License</label>
          <input type="text" id="owner-widgets-license"
                 name="owner.widgets.license"
                 class="text-widget required textline-field"
                 value="MA-97097A87" />
        </div>
      </fieldset>
      <div class="action">
        <input type="submit" id="car-buttons-apply"
               name="car.buttons.apply"
               class="submit-widget button-field"
               value="Apply" />
      </div>
    </form>
  </body>
</html>
```

The interesting part of this setup is that the "Apply" button calls the action handlers for both, the main and the sub-form:

```
>>> request = TestRequest(form={
...     'car.widgets.model': 'Ford',
...     'car.widgets.make': 'F150',
...     'car.buttons.apply': 'Apply',
...     'owner.widgets.name': 'Claudia Richter',
...     'owner.widgets.license': 'MA-991723FDG',
...     })
```

```
>>> carForm = CarForm(mycar, request)
>>> carForm.update()
```

```
>>> mycar.model
'Ford'
>>> mycar.make
'F150'
>>> me.name
'Claudia Richter'
>>> me.license
'MA-991723FDG'
```

Let's now have a look at cases where an error happens. If an error occurs in the parent form, the sub-form is still submitted:

```
>>> request = TestRequest(form={
...     'car.widgets.model': 'Volvo\n~',
...     'car.widgets.make': '450',
...     'car.buttons.apply': 'Apply',
...     'owner.widgets.name': 'Stephan Richter',
...     'owner.widgets.license': 'MA-991723FDG',
...     })
```

```
>>> carForm = CarForm(mycar, request)
>>> carForm.update()
```

```
>>> mycar.model
'Ford'
>>> mycar.make
'F150'
>>> me.name
'Stephan Richter'
>>> me.license
'MA-991723FDG'
```

Let's look at the rendered form:

```
>>> print(carForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <i>There were some errors.</i>
    <ul>
      <li>
        Model: <div class="error">Constraint not satisfied</div>
      </li>
    </ul>
    <form action=".">
      <div class="row">
        <b><div class="error">Constraint not satisfied</div>
        </b><label for="car-widgets-model">Model</label>
        <input type="text" id="car-widgets-model"
               name="car.widgets.model"
               class="text-widget required textline-field"
               value="Volvo ~" />
      </div>
      <div class="row">
        <label for="car-widgets-make">Make</label>
        <input type="text" id="car-widgets-make"
               name="car.widgets.make"
               class="text-widget required textline-field"
               value="450" />
      </div>
      <fieldset>
        <legend>Owner</legend>
        <i>Data successfully updated.</i>
        <div class="row">
          <label for="owner-widgets-name">Name</label>
```

(continues on next page)

(continued from previous page)

```
          <input type="text" id="owner-widgets-name"
                 name="owner.widgets.name"
                 class="text-widget required textline-field"
                 value="Stephan Richter" />
        </div>
        <div class="row">
          <label for="owner-widgets-license">License</label>
          <input type="text" id="owner-widgets-license"
                 name="owner.widgets.license"
                 class="text-widget required textline-field"
                 value="MA-991723FDG" />
        </div>
      </fieldset>
      <div class="action">
        <input type="submit" id="car-buttons-apply"
               name="car.buttons.apply" class="submit-widget button-field"
               value="Apply" />
      </div>
    </form>
  </body>
</html>
```

Now, we know, we know. This might not be the behavior that *you* want. But remember how we started this document. We started with the recognition that there are many classes and policies surrounding subforms. So while this package provides some sensible default behavior, it is not intended to be comprehensive.

Let's now create an error in the sub-form, ensuring that an error message occurs:

```
>>> request = TestRequest(form={
...     'car.widgets.model': 'Volvo',
...     'car.widgets.make': '450',
...     'car.buttons.apply': 'Apply',
...     'owner.widgets.name': 'Claudia\n Richter',
...     'owner.widgets.license': 'MA-991723F12',
...     })
```

```
>>> carForm = CarForm(mycar, request)
>>> carForm.update()
```

```
>>> mycar.model
'Volvo'
>>> mycar.make
'450'
>>> me.name
'Stephan Richter'
>>> me.license
'MA-991723FDG'
```

```
>>> print(carForm.render())
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

(continues on next page)

(continued from previous page)

```
    ...
    <fieldset>
      <legend>Owner</legend>
      <i>There were some errors.</i>
      <ul>
         <li>
           Name:
           <div class="error">Constraint not satisfied</div>
         </li>
      </ul>
     ...
   </fieldset>
   ...
</html>
```

If the data did not change, it is also locally reported:

```
>>> request = TestRequest(form={
...     'car.widgets.model': 'Ford',
...     'car.widgets.make': 'F150',
...     'car.buttons.apply': 'Apply',
...     'owner.widgets.name': 'Stephan Richter',
...     'owner.widgets.license': 'MA-991723FDG',
...     })
```

```
>>> carForm = CarForm(mycar, request)
>>> carForm.update()
>>> print(carForm.render())
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  ...
    <fieldset>
    <legend>Owner</legend>
    <i>No changes were applied.</i>
    ...
    </fieldset>
  ...
</html>
```

Final Note: With `zope.formlib` and `zope.app.form` people usually wrote complex object widgets to handle objects within forms. We never considered this a good way of programming, since one loses control over the layout too easily.

1.3.3 Context-free subforms

Ok, that was easy. But what about writing a form including a subform without a context? Let's show how we can write a form without any context using the sample above. Note, this sample form does not include actions which store the form input. You can store the values like in any other forms using the forms widget method `self.widgets.extract()` which will return the form and subform input values.

```
>>> from z3c.form.interfaces import IWidgets
>>> class OwnerAddForm(form.EditForm):
...     template = ViewPageTemplateFile(
...         'simple_owneredit.pt', templatePath)
...     fields = field.Fields(IOwner)
...     prefix = 'owner'
...
...     def updateWidgets(self):
...         self.widgets = zope.component.getMultiAdapter(
...             (self, self.request, self.getContent()), IWidgets)
...         self.widgets.ignoreContext = True
...         self.widgets.update()
```

Next we define the car form, which has the owner form as a sub-form.

```
>>> class CarAddForm(form.EditForm):
...     fields = field.Fields(ICar).select('model', 'make')
...     template = ViewPageTemplateFile(
...         'simple_caredit.pt', templatePath)
...     prefix = 'car'
...
...     def updateWidgets(self):
...         self.widgets = zope.component.getMultiAdapter(
...             (self, self.request, self.getContent()), IWidgets)
...         self.widgets.ignoreContext = True
...         self.widgets.update()
...
...     def update(self):
...         self.owner = OwnerAddForm(None, self.request)
...         self.owner.update()
...         super(CarAddForm, self).update()
```

Let's now instantiate the form and render it. but first set up a simple container which we can use for the add form context:

```
>>> class Container(object):
...     """Simple context simulating a container."""
>>> container = Container()
```

Set up a test request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

And render the form. As you can see, the widgets get rendered without any *real* context.

```
>>> carForm = CarAddForm(container, request)
>>> carForm.update()
>>> print(carForm.render())
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
<form action=".">
  <div class="row">
    <label for="car-widgets-model">Model</label>
    <input type="text" id="car-widgets-model"
           name="car.widgets.model"
           class="text-widget required textline-field"
           value="" />
  </div>
  <div class="row">
    <label for="car-widgets-make">Make</label>
    <input type="text" id="car-widgets-make"
           name="car.widgets.make"
           class="text-widget required textline-field"
           value="" />
  </div>
  <fieldset>
  <legend>Owner</legend>
  <div class="row">
    <label for="owner-widgets-name">Name</label>
    <input type="text" id="owner-widgets-name"
           name="owner.widgets.name"
           class="text-widget required textline-field"
           value="" />
  </div>
  <div class="row">
    <label for="owner-widgets-license">License</label>
    <input type="text" id="owner-widgets-license"
         name="owner.widgets.license"
         class="text-widget required textline-field"
         value="" />
  </div>
  <div class="action">
    <input type="submit" id="owner-buttons-apply"
         name="owner.buttons.apply" class="submit-widget button-field"
         value="Apply" />
  </div>
  </fieldset>
  <div class="action">
    <input type="submit" id="car-buttons-apply"
         name="car.buttons.apply" class="submit-widget button-field"
         value="Apply" />
  </div>
</form>
</body>
</html>
```

Let's show how we can extract the input values of the form and the subform. First give them some input:

```
>>> request = TestRequest(form={
...     'car.widgets.model': 'Ford',
...     'car.widgets.make': 'F150',
...     'owner.widgets.name': 'Stephan Richter',
...     'owner.widgets.license': 'MA-991723FDG',
...     })
>>> carForm = CarAddForm(container, request)
>>> carForm.update()
```

Now get the form values. This is normally done in a action handler:

```
>>> pprint(carForm.widgets.extract())
({'make': 'F150', 'model': 'Ford'}, ())
```

```
>>> pprint(carForm.owner.widgets.extract())
({'license': 'MA-991723FDG', 'name': 'Stephan Richter'}, ())
```

1.4 Field Managers

One of the features in `zope.formlib` that works really well is the syntax used to define the contents of the form. The formlib uses form fields, to describe how the form should be put together. Since we liked this way of working, this package offers this feature as well in a very similar way.

A field manager organizes all fields to be displayed within a form. Each field is associated with additional meta-data. The simplest way to create a field manager is to specify the schema from which to extract all fields.

Thus, the first step is to create a schema:

```
>>> import zope.interface
>>> import zope.schema
```

```
>>> class IPerson(zope.interface.Interface):
...     id = zope.schema.Int(
...         title='Id',
...         readonly=True)
...
...     name = zope.schema.TextLine(
...         title='Name')
...
...     country = zope.schema.Choice(
...         title='Country',
...         values=('Germany', 'Switzerland', 'USA'),
...         required=False)
```

We can now create the field manager:

```
>>> from z3c.form import field
>>> manager = field.Fields(IPerson)
```

Like all managers in this package, it provides the enumerable mapping API:

```
>>> manager['id']
<Field 'id'>
>>> manager['unknown']
Traceback (most recent call last):
...
KeyError: 'unknown'
```

```
>>> manager.get('id')
<Field 'id'>
>>> manager.get('unknown', 'default')
'default'
```

```
>>> 'id' in manager
True
>>> 'unknown' in manager
False
```

```
>>> list(manager.keys())
['id', 'name', 'country']
```

```
>>> [key for key in manager]
['id', 'name', 'country']
```

```
>>> list(manager.values())
[<Field 'id'>, <Field 'name'>, <Field 'country'>]
```

```
>>> list(manager.items())
[('id', <Field 'id'>),
 ('name', <Field 'name'>),
 ('country', <Field 'country'>)]
```

```
>>> len(manager)
3
```

You can also select the fields that you would like to have:

```
>>> manager = manager.select('name', 'country')
>>> list(manager.keys())
['name', 'country']
```

Changing the order is simply a matter of changing the selection order:

```
>>> manager = manager.select('country', 'name')
>>> list(manager.keys())
['country', 'name']
```

Selecting a field becomes a little bit more tricky when field names overlap. For example, let's say that a person can be adapted to a pet:

```
>>> class IPet(zope.interface.Interface):
...       id = zope.schema.TextLine(
```

(continues on next page)

(continued from previous page)

```
...           title='Id')
...
...       name = zope.schema.TextLine(
...           title='Name')
```

The pet field(s) can only be added to the fields manager with a prefix:

```
>>> manager += field.Fields(IPet, prefix='pet')
>>> list(manager.keys())
['country', 'name', 'pet.id', 'pet.name']
```

When selecting fields, this prefix has to be used:

```
>>> manager = manager.select('name', 'pet.name')
>>> list(manager.keys())
['name', 'pet.name']
```

However, sometimes it is tedious to specify the prefix together with the field; for example here:

```
>>> manager = field.Fields(IPerson).select('name')
>>> manager += field.Fields(IPet, prefix='pet').select('pet.name', 'pet.id')
>>> list(manager.keys())
['name', 'pet.name', 'pet.id']
```

It is easier to specify the prefix as an afterthought:

```
>>> manager = field.Fields(IPerson).select('name')
>>> manager += field.Fields(IPet, prefix='pet').select(
...     'name', 'id', prefix='pet')
>>> list(manager.keys())
['name', 'pet.name', 'pet.id']
```

Alternatively, you can specify the interface:

```
>>> manager = field.Fields(IPerson).select('name')
>>> manager += field.Fields(IPet, prefix='pet').select(
...     'name', 'id', interface=IPet)
>>> list(manager.keys())
['name', 'pet.name', 'pet.id']
```

Sometimes it is easier to simply omit a set of fields instead of selecting all the ones you want:

```
>>> manager = field.Fields(IPerson)
>>> manager = manager.omit('id')
>>> list(manager.keys())
['name', 'country']
```

Again, you can solve name conflicts using the full prefixed name, ...

```
>>> manager = field.Fields(IPerson).omit('country')
>>> manager += field.Fields(IPet, prefix='pet')
>>> list(manager.omit('pet.id').keys())
['id', 'name', 'pet.name']
```

using the prefix keyword argument, ...

```
>>> manager = field.Fields(IPerson).omit('country')
>>> manager += field.Fields(IPet, prefix='pet')
>>> list(manager.omit('id', prefix='pet').keys())
['id', 'name', 'pet.name']
```

or, using the interface:

```
>>> manager = field.Fields(IPerson).omit('country')
>>> manager += field.Fields(IPet, prefix='pet')
>>> list(manager.omit('id', interface=IPet).keys())
['id', 'name', 'pet.name']
```

You can also add two field managers together:

```
>>> manager = field.Fields(IPerson).select('name', 'country')
>>> manager2 = field.Fields(IPerson).select('id')
>>> list((manager + manager2).keys())
['name', 'country', 'id']
```

Adding anything else to a field manager is not well defined:

```
>>> manager + 1
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Fields' and 'int'
```

You also cannot make any additions that would cause a name conflict:

```
>>> manager + manager
Traceback (most recent call last):
...
ValueError: ('Duplicate name', 'name')
```

When creating a new form derived from another, you often want to keep existing fields and add new ones. In order to not change the super-form class, you need to copy the field manager:

```
>>> list(manager.keys())
['name', 'country']
>>> list(manager.copy().keys())
['name', 'country']
```

1.4.1 More on the Constructor

The constructor does not only accept schemas to be passed in; one can also just pass in schema fields:

```
>>> list(field.Fields(IPerson['name']).keys())
['name']
```

However, the schema field has to have a name:

```
>>> email = zope.schema.TextLine(title='E-Mail')
>>> field.Fields(email)
Traceback (most recent call last):
...
ValueError: Field has no name
```

Adding a name helps:

```
>>> email.__name__ = 'email'
>>> list(field.Fields(email).keys())
['email']
```

Or, you can just pass in other field managers, which is the feature that the add mechanism uses:

```
>>> list(field.Fields(manager).keys())
['name', 'country']
```

Last, but not least, the constructor also accepts form fields, which are used by `select()` and `omit()`:

```
>>> list(field.Fields(manager['name'], manager2['id']).keys())
['name', 'id']
```

If the constructor does not recognize any of the types above, it raises a `TypeError` exception:

```
>>> field.Fields(object())
Traceback (most recent call last):
...
TypeError: ('Unrecognized argument type', <object object at ...>)
```

Additionally, you can specify several keyword arguments in the field manager constructor that are used to set up the fields:

- `omitReadOnly`

  When set to `True` all read-only fields are omitted.

  ```
  >>> list(field.Fields(IPerson, omitReadOnly=True).keys())
  ['name', 'country']
  ```

- `keepReadOnly`

  Sometimes you want to keep a particular read-only field around, even though in general you want to omit them. In this case you can specify the fields to keep:

  ```
  >>> list(field.Fields(
  ...     IPerson, omitReadOnly=True, keepReadOnly=('id',)).keys())
  ['id', 'name', 'country']
  ```

- `prefix`

  Sets the prefix of the fields. This argument is passed on to each field.

  ```
  >>> manager = field.Fields(IPerson, prefix='myform.')
  >>> manager['myform.name']
  <Field 'myform.name'>
  ```

- `interface`

  Usually the interface is inferred from the field itself. The interface is used to determine whether an adapter must be looked up for a given context.

  But sometimes fields are generated in isolation to an interface or the interface of the field is not the one you want. In this case you can specify the interface:

```
>>> class IMyPerson(IPerson):
...     pass
```

```
>>> manager = field.Fields(email, interface=IMyPerson)
>>> manager['email'].interface
<InterfaceClass builtins.IMyPerson>
```

- `mode`

  The mode in which the widget will be rendered. By default there are two available, "input" and "display". When mode is not specified, "input" is chosen.

```
>>> from z3c.form import interfaces
>>> manager = field.Fields(IPerson, mode=interfaces.DISPLAY_MODE)
>>> manager['country'].mode
'display'
```

- `ignoreContext`

  While the `ignoreContext` flag is usually set on the form, it is sometimes desirable to set the flag for a particular field.

```
>>> manager = field.Fields(IPerson)
>>> manager['country'].ignoreContext
```

```
>>> manager = field.Fields(IPerson, ignoreContext=True)
>>> manager['country'].ignoreContext
True
```

```
>>> manager = field.Fields(IPerson, ignoreContext=False)
>>> manager['country'].ignoreContext
False
```

- `showDefault`

  The `showDefault` can be set on fields.

```
>>> manager = field.Fields(IPerson)
>>> manager['country'].showDefault
```

```
>>> manager = field.Fields(IPerson, showDefault=True)
>>> manager['country'].showDefault
True
```

```
>>> manager = field.Fields(IPerson, showDefault=False)
>>> manager['country'].showDefault
False
```

1.4.2 Fields Widget Manager

When a form (or any other widget-using view) is updated, one of the tasks is to create the widgets. Traditionally, generating the widgets involved looking at the form fields (or similar) of a form and generating the widgets using the information of those specifications. This solution is good for the common (about 85%) use cases, since it makes writing new forms very simple and allows a lot of control at a class-definition level.

It has, however, its limitations. It does not, for example, allow for customization without rewriting a form. This can range from omitting fields on a particular form to generically adding a new widget to the form, such as an "object name" button on add forms. This package solves this issue by providing a widget manager, which is responsible providing the widgets for a particular view.

The default widget manager for forms is able to look at a form's field definitions and create widgets for them. Thus, let's create a schema first:

```
>>> import zope.interface
>>> import zope.schema
```

```
>>> class LastNameTooShort(zope.schema.interfaces.ValidationError):
...     """The last name is too short."""
```

```
>>> def lastNameConstraint(value):
...     if value and value == value.lower():
...         raise zope.interface.Invalid(u"Name must have at least one capital letter")
...     return True
```

```
>>> class IPerson(zope.interface.Interface):
...     id = zope.schema.TextLine(
...         title='ID',
...         description=u"The person's ID.",
...         readonly=True,
...         required=True)
...
...     lastName = zope.schema.TextLine(
...         title='Last Name',
...         description=u"The person's last name.",
...         default='',
...         required=True,
...         constraint=lastNameConstraint)
...
...     firstName = zope.schema.TextLine(
...         title='First Name',
...         description=u"The person's first name.",
...         default='-- unknown --',
...         required=False)
...
...     @zope.interface.invariant
...     def twiceAsLong(person):
...         # note: we're protecting here values against being None
...         # just in case ignoreRequiredOnExtract lets that through
...         if len(person.lastName or '') >= 2 * len(person.firstName or ''):
...             raise LastNameTooShort()
```

Next we need a form that specifies the fields to be added:

```
>>> from z3c.form import field
```

```
>>> class PersonForm(object):
...     prefix = 'form.'
...     fields = field.Fields(IPerson)
>>> personForm = PersonForm()
```

For more details on how to define fields within a form, see *Forms*. We can now create the fields widget manager. Its discriminators are the form for which the widgets are created, the request, and the context that is being manipulated. In the simplest case the context is None and ignored, as it is true for an add form.

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> context = object()
```

```
>>> manager = field.FieldWidgets(personForm, request, context)
>>> manager.ignoreContext = True
```

Widget Mapping

The main responsibility of the manager is to provide the IEnumerableMapping interface and an update() method. Initially the mapping, going from widget id to widget value, is empty:

```
>>> from zope.interface.common.mapping import IEnumerableMapping
>>> IEnumerableMapping.providedBy(manager)
True
```

```
>>> list(manager.keys())
[]
```

Only by "updating" the manager, will the widgets become available; before we can use the update method, however, we have to register the IFieldWidget adapter for the ITextLine field:

```
>>> from z3c.form import interfaces, widget
```

```
>>> @zope.component.adapter(zope.schema.TextLine, TestRequest)
... @zope.interface.implementer(interfaces.IFieldWidget)
... def TextFieldWidget(field, request):
...     return widget.FieldWidget(field, widget.Widget(request))
```

```
>>> zope.component.provideAdapter(TextFieldWidget)
```

```
>>> from z3c.form import converter
>>> zope.component.provideAdapter(converter.FieldDataConverter)
>>> zope.component.provideAdapter(converter.FieldWidgetDataConverter)
```

```
>>> manager.update()
```

Other than usual mappings in Python, the widget manager's widgets are always in a particular order:

```
>>> list(manager.keys())
['id', 'lastName', 'firstName']
```

As you can see, if we call update twice, we still get the same amount and order of keys:

```
>>> manager.update()
>>> list(manager.keys())
['id', 'lastName', 'firstName']
```

Let's make sure that all enumerable mapping functions work correctly:

```
>>> manager['lastName']
<Widget 'form.widgets.lastName'>
```

```
>>> manager['unknown']
Traceback (most recent call last):
...
KeyError: 'unknown'
```

```
>>> manager.get('lastName')
<Widget 'form.widgets.lastName'>
```

```
>>> manager.get('unknown', 'default')
'default'
```

```
>>> 'lastName' in manager
True
>>> 'unknown' in manager
False
```

```
>>> [key for key in manager]
['id', 'lastName', 'firstName']
```

```
>>> list(manager.values())
[<Widget 'form.widgets.id'>,
 <Widget 'form.widgets.lastName'>,
 <Widget 'form.widgets.firstName'>]
```

```
>>> list(manager.items())
[('id', <Widget 'form.widgets.id'>),
 ('lastName', <Widget 'form.widgets.lastName'>),
 ('firstName', <Widget 'form.widgets.firstName'>)]
```

```
>>> len(manager)
3
```

It is also possible to delete widgets from the manager:

```
>>> del manager['firstName']
>>> len(manager)
2
```

(continues on next page)

(continued from previous page)

```
>>> list(manager.values())
[<Widget 'form.widgets.id'>, <Widget 'form.widgets.lastName'>]
>>> list(manager.keys())
['id', 'lastName']
>>> list(manager.items())
[('id', <Widget 'form.widgets.id'>),
('lastName', <Widget 'form.widgets.lastName'>)]
```

Note that deleting a non-existent widget causes a `KeyError` to be raised:

```
>>> del manager['firstName']
Traceback (most recent call last):
...
KeyError: 'firstName'
```

Also, the field widget manager, like any selection manager, can be cloned:

```
>>> clone = manager.copy()
>>> clone is not manager
True
>>> clone.form == manager.form
True
>>> clone.request == manager.request
True
>>> clone.content == manager.content
True
>>> list(clone.items()) == list(manager.items())
True
```

Properties of widgets within a manager

When a widget is added to the widget manager, it is located:

```
>>> lname = manager['lastName']
```

```
>>> lname.__name__
'lastName'
>>> lname.__parent__
FieldWidgets([...])
```

All widgets created by this widget manager are context aware:

```
>>> interfaces.IContextAware.providedBy(lname)
True
>>> lname.context is context
True
```

Determination of the widget mode

By default, all widgets will also assume the mode of the manager:

```
>>> manager['lastName'].mode
'input'
```

```
>>> manager.mode = interfaces.DISPLAY_MODE
>>> manager.update()
```

```
>>> manager['lastName'].mode
'display'
```

The exception is when some fields specifically desire a different mode. In the first case, all "readonly" fields will be shown in display mode:

```
>>> manager.mode = interfaces.INPUT_MODE
>>> manager.update()
```

```
>>> manager['id'].mode
'display'
```

An exception is made when the flag, "ignoreReadonly" is set:

```
>>> manager.ignoreReadonly = True
>>> manager.update()
```

```
>>> manager['id'].mode
'input'
```

In the second case, the last name will inherit the mode from the widget manager, while the first name will want to use a display widget:

```
>>> personForm.fields = field.Fields(IPerson).select('lastName')
>>> personForm.fields += field.Fields(
...     IPerson, mode=interfaces.DISPLAY_MODE).select('firstName')
```

```
>>> manager.mode = interfaces.INPUT_MODE
>>> manager.update()
```

```
>>> manager['lastName'].mode
'input'
>>> manager['firstName'].mode
'display'
```

In a third case, the widget will be shown in display mode, if the attribute of the context is not writable. Clearly this can never occur in add forms, since there the context is ignored, but is an important use case in edit forms.

Thus, we need an implementation of the `IPerson` interface including some security declarations:

```
>>> from zope.security import checker
```

```
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...
...     def __init__(self, firstName, lastName):
...         self.id = firstName[0].lower() + lastName.lower()
...         self.firstName = firstName
...         self.lastName = lastName
```

```
>>> PersonChecker = checker.Checker(
...     get_permissions = {'id': checker.CheckerPublic,
...                        'firstName': checker.CheckerPublic,
...                        'lastName': checker.CheckerPublic},
...     set_permissions = {'firstName': 'test.Edit',
...                        'lastName': checker.CheckerPublic}
...     )
```

```
>>> srichter = checker.ProxyFactory(
...     Person('Stephan', 'Richter'), PersonChecker)
```

In this case the last name is always editable, but for the first name the user will need the edit ("test.Edit") permission.

We also need to register the data manager and setup a new security policy:

```
>>> from z3c.form import datamanager
>>> zope.component.provideAdapter(datamanager.AttributeField)
```

```
>>> from zope.security import management
>>> from z3c.form import testing
>>> management.endInteraction()
>>> newPolicy = testing.SimpleSecurityPolicy()
>>> oldpolicy = management.setSecurityPolicy(newPolicy)
>>> management.newInteraction()
```

Now we can create the widget manager:

```
>>> personForm = PersonForm()
>>> request = TestRequest()
>>> manager = field.FieldWidgets(personForm, request, srichter)
```

After updating the widget manager, the fields are available as widgets, the first name being in display and the last name is input mode:

```
>>> manager.update()
>>> manager['id'].mode
'display'
>>> manager['firstName'].mode
'display'
>>> manager['lastName'].mode
'input'
```

However, explicitly overriding the mode in the field declaration overrides this selection for you:

```
>>> personForm.fields['firstName'].mode = interfaces.INPUT_MODE
```

```
>>> manager.update()
>>> manager['id'].mode
'display'
>>> manager['firstName'].mode
'input'
>>> manager['lastName'].mode
'input'
```

1.4.3 showDefault

showDefault by default is True:

```
>>> manager['firstName'].showDefault
True
```

showDefault gets set on the widget based on the field's setting.

```
>>> personForm.fields['firstName'].showDefault = False
```

```
>>> manager.update()
>>> manager['firstName'].showDefault
False
```

```
>>> personForm.fields['firstName'].showDefault = True
```

```
>>> manager.update()
>>> manager['firstName'].showDefault
True
```

1.4.4 Required fields

There is a flag for required fields. This flag get set if at least one field is required. This let us render a required info legend in forms if required fields get used.

```
>>> manager.hasRequiredFields
True
```

Data extraction and validation

Besides managing widgets, the widget manager also controls the process of extracting and validating extracted data. Let's start with the validation first, which only validates the data as a whole, assuming each individual value being already validated.

Before we can use the method, we have to register a "manager validator":

```
>>> from z3c.form import validator
>>> zope.component.provideAdapter(validator.InvariantsValidator)
```

```
>>> personForm.fields = field.Fields(IPerson)
>>> manager.update()
```

```
>>> manager.validate(
...     {'firstName': 'Stephan', 'lastName': 'Richter'})
()
```

The result of this method is a tuple of errors that occurred during the validation. An empty tuple means the validation succeeded. Let's now make the validation fail:

```
>>> errors = manager.validate(
...     {'firstName': 'Stephan', 'lastName': 'Richter-Richter'})
```

```
>>> [error.doc() for error in errors]
['The last name is too short.']
```

A special case occurs when the schema fields are not associated with an interface:

```
>>> name = zope.schema.TextLine(__name__='name')
```

```
>>> class PersonNameForm(object):
...     prefix = 'form.'
...     fields = field.Fields(name)
>>> personNameForm = PersonNameForm()
```

```
>>> manager = field.FieldWidgets(personNameForm, request, context)
```

In this case, the widget manager's `validate()` method should simply ignore the field and not try to look up any invariants:

```
>>> manager.validate({'name': 'Stephan'})
()
```

Let's now have a look at the widget manager's `extract()`, which returns a data dictionary and the collection of errors. Before we can validate, we have to register a validator for the widget:

```
>>> zope.component.provideAdapter(validator.SimpleFieldValidator)
```

When all goes well, the data dictionary is complete and the error collection empty:

```
>>> request = TestRequest(form={
...     'form.widgets.id': 'srichter',
...     'form.widgets.firstName': 'Stephan',
...     'form.widgets.lastName': 'Richter'})
>>> manager = field.FieldWidgets(personForm, request, context)
>>> manager.ignoreContext = True
>>> manager.update()
```

```
>>> data, errors = manager.extract()
>>> data['firstName']
'Stephan'
>>> data['lastName']
```

(continues on next page)

(continued from previous page)

```
'Richter'
>>> errors
()
```

Since all errors are immediately converted to error view snippets, we have to provide the adapter from a validation error to an error view snippet first:

```
>>> from z3c.form import error
>>> zope.component.provideAdapter(error.ErrorViewSnippet)
>>> zope.component.provideAdapter(error.InvalidErrorViewSnippet)
```

Let's now cause a widget-level error by not submitting the required last name:

```
>>> request = TestRequest(form={
...     'form.widgets.firstName': 'Stephan', 'form.widgets.id': 'srichter'})
>>> manager = field.FieldWidgets(personForm, request, context)
>>> manager.ignoreContext = True
>>> manager.update()
>>> manager.extract()
({'firstName': 'Stephan'}, (<ErrorViewSnippet for RequiredMissing>,))
```

We can also turn off `required` checking for data extraction:

```
>>> request = TestRequest(form={
...     'form.widgets.firstName': 'Stephan', 'form.widgets.id': 'srichter'})
>>> manager = field.FieldWidgets(personForm, request, context)
>>> manager.ignoreContext = True
>>> manager.ignoreRequiredOnExtract = True
>>> manager.update()
```

Here we get the required field as `None` and no errors:

```
>>> pprint(manager.extract())
({'firstName': 'Stephan', 'lastName': None}, ())
```

```
>>> manager.ignoreRequiredOnExtract = False
```

Or, we could violate a constraint. This constraint raises Invalid, which is a convenient way to raise errors where we mainly care about providing a custom error message.

```
>>> request = TestRequest(form={
...     'form.widgets.firstName': 'Stephan',
...     'form.widgets.lastName': 'richter',
...     'form.widgets.id': 'srichter'})
>>> manager = field.FieldWidgets(personForm, request, context)
>>> manager.ignoreContext = True
>>> manager.update()
>>> extracted = manager.extract()
>>> extracted
({'firstName': 'Stephan'}, (<InvalidErrorViewSnippet for Invalid>,))
```

```
>>> extracted[1][0].createMessage()
'Name must have at least one capital letter'
```

Finally, let's ensure that invariant failures are also caught:

```
>>> request = TestRequest(form={
...     'form.widgets.id': 'srichter',
...     'form.widgets.firstName': 'Stephan',
...     'form.widgets.lastName': 'Richter-Richter'})
>>> manager = field.FieldWidgets(personForm, request, context)
>>> manager.ignoreContext = True
>>> manager.update()
>>> data, errors = manager.extract()
>>> errors[0].error.doc()
'The last name is too short.'
```

Note that the errors coming from invariants are all error view snippets as well, just as it is the case for field-specific validation errors. And that's really all there is!

By default, the `extract()` method not only returns the errors that it catches, but also sets them on individual widgets and on the manager:

```
>>> manager.errors
(<ErrorViewSnippet for LastNameTooShort>,)
```

This behavior can be turned off. To demonstrate, let's make a new request that causes a widget-level error:

```
>>> request = TestRequest(form={
...     'form.widgets.firstName': 'Stephan', 'form.widgets.id': 'srichter'})
>>> manager = field.FieldWidgets(personForm, request, context)
>>> manager.ignoreContext = True
>>> manager.update()
```

We have to set the setErrors property to False before calling extract, we still get the same result from the method call, ...

```
>>> manager.setErrors = False
>>> manager.extract()
({'firstName': 'Stephan'}, (<ErrorViewSnippet for RequiredMissing>,))
```

but there are no side effects on the manager and the widgets:

```
>>> manager.errors
()
>>> manager['lastName'].error is None
True
```

Customization of Ignoring the Context

Note that you can also manually control ignoring the context per field.

```
>>> class CustomPersonForm(object):
...     prefix = 'form.'
...     fields = field.Fields(IPerson).select('id')
...     fields += field.Fields(IPerson, ignoreContext=True).select(
...                     'firstName', 'lastName')
>>> customPersonForm = CustomPersonForm()
```

Let's now create a manager and update it:

```
>>> customManager = field.FieldWidgets(customPersonForm, request, context)
>>> customManager.update()
```

```
>>> customManager['id'].ignoreContext
False
>>> customManager['firstName'].ignoreContext
True
>>> customManager['lastName'].ignoreContext
True
```

1.4.5 Fields – Custom Widget Factories

It is possible to declare custom widgets for fields within the field's declaration.

Let's have a look at the default form first. Initially, the standard registered widgets are used:

```
>>> manager = field.FieldWidgets(personForm, request, srichter)
>>> manager.update()
```

```
>>> manager['firstName']
<Widget 'form.widgets.firstName'>
```

Now we would like to have our own custom input widget:

```
>>> class CustomInputWidget(widget.Widget):
...     pass
```

```
>>> def CustomInputWidgetFactory(field, request):
...     return widget.FieldWidget(field, CustomInputWidget(request))
```

It can be simply assigned as follows:

```
>>> personForm.fields['firstName'].widgetFactory = CustomInputWidgetFactory
>>> personForm.fields['lastName'].widgetFactory = CustomInputWidgetFactory
```

Now this widget should be used instead of the registered default one:

```
>>> manager = field.FieldWidgets(personForm, request, srichter)
>>> manager.update()
>>> manager['firstName']
<CustomInputWidget 'form.widgets.firstName'>
```

In the background the widget factory assignment really just registered the default factory in the `WidgetFactories` object, which manages the custom widgets for all modes. Now all modes show this input widget:

```
>>> manager = field.FieldWidgets(personForm, request, srichter)
>>> manager.mode = interfaces.DISPLAY_MODE
>>> manager.update()
>>> manager['firstName']
<CustomInputWidget 'form.widgets.firstName'>
```

However, we can also register a specific widget for the display mode:

```
>>> class CustomDisplayWidget(widget.Widget):
...     pass
```

```
>>> def CustomDisplayWidgetFactory(field, request):
...     return widget.FieldWidget(field, CustomDisplayWidget(request))
```

```
>>> personForm.fields['firstName']\
...     .widgetFactory[interfaces.DISPLAY_MODE] = CustomDisplayWidgetFactory
>>> personForm.fields['lastName']\
...     .widgetFactory[interfaces.DISPLAY_MODE] = CustomDisplayWidgetFactory
```

Now the display mode should produce the custom display widget, ...

```
>>> manager = field.FieldWidgets(personForm, request, srichter)
>>> manager.mode = interfaces.DISPLAY_MODE
>>> manager.update()
>>> manager['firstName']
<CustomDisplayWidget 'form.widgets.firstName'>
>>> manager['lastName']
<CustomDisplayWidget 'form.widgets.lastName'>
```

... while the input mode still shows the default custom input widget on the `lastName` field but not on the `firstName` field since we don't have the `test.Edit` permission:

```
>>> manager = field.FieldWidgets(personForm, request, srichter)
>>> manager.mode = interfaces.INPUT_MODE
>>> manager.update()
>>> manager['firstName']
<CustomDisplayWidget 'form.widgets.firstName'>
>>> manager['lastName']
<CustomInputWidget 'form.widgets.lastName'>
```

The widgets factories component,

```
>>> factories = personForm.fields['firstName'].widgetFactory
>>> factories
{'display': <function CustomDisplayWidgetFactory at ...>}
```

is pretty much a standard dictionary that also manages a default value:

```
>>> factories.default
<function CustomInputWidgetFactory at ...>
```

When getting a value for a key, if the key is not found, the default is returned:

```
>>> sorted(factories.keys())
['display']
```

```
>>> factories[interfaces.DISPLAY_MODE]
<function CustomDisplayWidgetFactory at ...>
>>> factories[interfaces.INPUT_MODE]
<function CustomInputWidgetFactory at ...>
```

```
>>> factories.get(interfaces.DISPLAY_MODE)
<function CustomDisplayWidgetFactory at ...>
>>> factories.get(interfaces.INPUT_MODE)
<function CustomInputWidgetFactory at ...>
```

If no default is specified,

```
>>> factories.default = None
```

then the dictionary behaves as usual:

```
>>> factories[interfaces.DISPLAY_MODE]
<function CustomDisplayWidgetFactory at ...>
>>> factories[interfaces.INPUT_MODE]
Traceback (most recent call last):
...
KeyError: 'input'
```

```
>>> factories.get(interfaces.DISPLAY_MODE)
<function CustomDisplayWidgetFactory at ...>
>>> factories.get(interfaces.INPUT_MODE)
```

1.5 Buttons

Buttons are a method to declare actions for a form. Like fields describe widgets within a form, buttons describe actions. The symmetry goes even further; like fields, buttons are schema fields within schema. When the form is instantiated and updated, the buttons are converted to actions.

```
>>> from z3c.form import button
```

1.5.1 Schema Defined Buttons

Let's now create a schema that describes the buttons of a form. Having button schemas allows one to more easily reuse button declarations and to group them logically. `Button` objects are just a simple extension to `Field` objects, so they behave identical within a schema:

```
>>> import zope.interface
>>> class IButtons(zope.interface.Interface):
...     apply = button.Button(title='Apply')
...     cancel = button.Button(title='Cancel')
```

In reality, only the title and name is relevant. Let's now create a form that provides those buttons.

```
>>> from z3c.form import interfaces
>>> @zope.interface.implementer(
...     interfaces.IButtonForm, interfaces.IHandlerForm)
... class Form(object):
...     buttons = button.Buttons(IButtons)
...     prefix = 'form'
...
```

(continues on next page)

(continued from previous page)

```
...     @button.handler(IButtons['apply'])
...     def apply(self, action):
...         print('successfully applied')
...
...     @button.handler(IButtons['cancel'])
...     def cancel(self, action):
...         self.request.response.redirect('index.html')
```

Let's now create an action manager for the button manager in the form. To do that we first need a request and a form instance:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> form = Form()
```

We also have to register a button action factory for the buttons:

```
>>> zope.component.provideAdapter(
...     button.ButtonAction, provides=interfaces.IButtonAction)
```

Action managers are instantiated using the form, request, and context/content. A special button-action-manager implementation is available in the `button` package:

```
>>> actions = button.ButtonActions(form, request, None)
>>> actions.update()
```

Once the action manager is updated, the buttons should be available as actions:

```
>>> list(actions.keys())
['apply', 'cancel']
```

```
>>> actions['apply']
<ButtonAction 'form.buttons.apply' 'Apply'>
```

It is possible to customize how a button is transformed into an action by registering an adapter for the request and the button that provides `IButtonAction`.

```
>>> import zope.component
>>> from zope.publisher.interfaces.browser import IBrowserRequest
>>> class CustomButtonAction(button.ButtonAction):
...     """Custom Button Action Class."""
```

```
>>> zope.component.provideAdapter(
...     CustomButtonAction, provides=interfaces.IButtonAction)
```

Now if we rerun update we will get this other ButtonAction implementation. Note, there are two strategies what now could happen. We can remove the existing action and get the new adapter based action or we can reuse the existing action. Since the ButtonActions class offers an API for remove existing actions, we reuse the existing action because it very uncommon to replace existing action during an for update call with an adapter. If someone really will add an action adapter during process time via directly provided interface, he is also responsible for remove existing actions.

As you can see we still will get the old button action if we only call update:

```
>>> actions.update()
>>> list(actions.keys())
['apply', 'cancel']
```

```
>>> actions['apply']
<ButtonAction 'form.buttons.apply' 'Apply'>
```

This means we have to remove the previous action before we call update:

```
>>> del actions['apply']
>>> actions.update()
```

Make sure we do not append a button twice to the key and value lists by calling update twice:

```
>>> list(actions.keys())
['apply', 'cancel']
```

```
>>> actions['apply']
<CustomButtonAction 'form.buttons.apply' 'Apply'>
```

Alternatively, customize an individual button by setting its actionFactory attribute.

```
>>> def customButtonActionFactory(request, field):
...     print("This button factory creates a button only once.")
...     button = CustomButtonAction(request, field)
...     button.css = "happy"
...     return button
```

```
>>> form.buttons['apply'].actionFactory = customButtonActionFactory
```

Again, remove the old button action befor we call update:

```
>>> del actions['apply']
>>> actions.update()
This button factory creates a button only once.
```

```
>>> actions.update()
>>> actions['apply'].css
'happy'
```

Since we only create a button once from an adapter or a factory, we can change the button attributes without to lose changes:

```
>>> actions['apply'].css = 'very happy'
>>> actions['apply'].css
'very happy'
```

```
>>> actions.update()
>>> actions['apply'].css
'very happy'
```

But let's not digress too much and get rid of this customization

```
>>> form.buttons['apply'].actionFactory = None
>>> actions.update()
```

Button actions are locations:

```
>>> apply = actions['apply']
>>> apply.__name__
'apply'
>>> apply.__parent__
<ButtonActions None>
```

A button action is also a submit widget. The attributes translate as follows:

```
>>> interfaces.ISubmitWidget.providedBy(apply)
True
```

```
>>> apply.value == apply.title
True
>>> apply.id == apply.name.replace('.', '-')
True
```

Next we want to display our button actions. To be able to do this, we have to register a template for the submit widget:

```
>>> from z3c.form import testing, widget
>>> templatePath = testing.getPath('submit_input.pt')
>>> factory = widget.WidgetTemplateFactory(templatePath, 'text/html')
```

```
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, TestRequest, None, None,
...      interfaces.ISubmitWidget),
...     IPageTemplate, name='input')
```

A widget template has many discriminators: context, request, view, field, and widget. We can now render each action:

```
>>> print(actions['apply'].render())
<input type="submit" id="form-buttons-apply"
       name="form.buttons.apply" class="submit-widget button-field"
       value="Apply" />
```

So displaying is nice, but how do button handlers get executed? The action manager provides attributes and method to check whether actions were executed. Initially there are no executed actions:

```
>>> list(actions.executedActions)
[]
```

So in this case executing the actions does not do anything:

```
>>> actions.execute()
```

But if the request contains the information that the button was pressed, the execution works:

```
>>> request = TestRequest(form={'form.buttons.apply': 'Apply'})
```

```
>>> actions = button.ButtonActions(form, request, None)
>>> actions.update()
>>> actions.execute()
```

Aehm, something should have happened. But in order for the system to look at the handlers declared in the form, a special action handler has to be registered with the system:

```
>>> zope.component.provideAdapter(button.ButtonActionHandler)
```

And voila, the execution works:

```
>>> actions.execute()
successfully applied
```

Finally, if there is no handler for a button, then the button click is silently ignored:

```
>>> form.handlers = button.Handlers()
>>> actions.execute()
```

While this might seem awkward at first, this is an intended feature. Sometimes there are several sub-forms that listen to a particular button and one form or another might simply not care about the button at all and not provide a handler.

1.5.2 In-Form Button Declarations

Some readers might find it cumbersome to declare a full schema just to create some buttons. A faster method is to write simple arguments to the button manager:

```
>>> @zope.interface.implementer(
...     interfaces.IButtonForm, interfaces.IHandlerForm)
... class Form(object):
...     buttons = button.Buttons(
...         button.Button('apply', title='Apply'))
...     prefix = 'form.'
...
...     @button.handler(buttons['apply'])
...     def apply(self, action):
...         print('successfully applied')
```

The first argument of the `Button` class constructor is the name of the button. Optionally, this can also be one of the following keyword arguments:

```
>>> button.Button(name='apply').__name__
'apply'
>>> button.Button(__name__='apply').__name__
'apply'
```

If no name is specified, the button will not have a name immediately, ...

```
>>> button.Button(title='Apply').__name__
''
```

because if the button is created within an interface, the name is assigned later:

```
>>> class IActions(zope.interface.Interface):
...     apply = button.Button(title='Apply')
```

```
>>> IActions['apply'].__name__
'apply'
```

However, once the button is added to a button manager, a name will be assigned:

```
>>> btns = button.Buttons(button.Button(title='Apply'))
>>> btns['apply'].__name__
'apply'
```

```
>>> btns = button.Buttons(button.Button(title='Apply and more'))
>>> btns['4170706c7920616e64206d6f7265'].__name__
'4170706c7920616e64206d6f7265'
```

This declaration behaves identical to the one before:

```
>>> form = Form()
>>> request = TestRequest()
```

```
>>> actions = button.ButtonActions(form, request, None)
>>> actions.update()
>>> actions.execute()
```

When sending in the right information, the actions are executed:

```
>>> request = TestRequest(form={'form.buttons.apply': 'Apply'})
>>> actions = button.ButtonActions(form, request, None)
>>> actions.update()
>>> actions.execute()
successfully applied
```

An even simpler method – resembling closest the API provided by formlib – is to create the button and handler at the same time:

```
>>> @zope.interface.implementer(
...     interfaces.IButtonForm, interfaces.IHandlerForm)
... class Form(object):
...     prefix = 'form.'
...
...     @button.buttonAndHandler('Apply')
...     def apply(self, action):
...         print('successfully applied')
```

In this case the `buttonAndHandler` decorator creates a button and a handler for it. By default the name is computed from the title of the button, which is required. All (keyword) arguments are forwarded to the button constructor. Let's now render the form:

```
>>> request = TestRequest(form={'form.buttons.apply': 'Apply'})
>>> actions = button.ButtonActions(form, request, None)
>>> actions.update()
```

(continues on next page)

(continued from previous page)

```
>>> actions.execute()
successfully applied
```

If the title is a more complex string, then the name of the button becomes a hex-encoded string:

```
>>> class Form(object):
...
...     @button.buttonAndHandler('Apply and Next')
...     def apply(self, action):
...         print('successfully applied')
```

```
>>> list(Form.buttons.keys())
['4170706c7920616e64204e657874']
```

Of course, you can use the `__name__` argument to specify a name yourself. The decorator, however, also allows the keyword `name`:

```
>>> class Form(object):
...
...     @button.buttonAndHandler('Apply and Next', name='applyNext')
...     def apply(self, action):
...         print('successfully applied')
```

```
>>> list(Form.buttons.keys())
['applyNext']
```

This helper function also supports a keyword argument `provides`, which allows the developer to specify a sequence of interfaces that the generated button should directly provide. Those provided interfaces can be used for a multitude of things, including handler discrimination and UI layout:

```
>>> class IMyButton(zope.interface.Interface):
...     pass
```

```
>>> class Form(object):
...
...     @button.buttonAndHandler('Apply', provides=(IMyButton,))
...     def apply(self, action):
...         print('successfully applied')
```

```
>>> IMyButton.providedBy(Form.buttons['apply'])
True
```

1.5.3 Button Conditions

Sometimes it is desirable to only show a button when a certain condition is fulfilled. The `Button` field supports conditions via a simple argument. The `condition` argument must be a callable taking the form as argument and returning a truth-value. If the condition is not fulfilled, the button will not be converted to an action:

```
>>> class Form(object):
...     prefix = 'form'
...     showApply = True
...
...     @button.buttonAndHandler(
...         'Apply', condition=lambda form: form.showApply)
...     def apply(self, action):
...         print('successfully applied')
```

In this case a form variable specifies the availability. Initially the button is available as action:

```
>>> myform = Form()
>>> actions = button.ButtonActions(myform, TestRequest(), None)
>>> actions.update()
>>> list(actions.keys())
['apply']
```

If we set the show-apply attribute to false, the action will not be available.

```
>>> myform.showApply = False
>>> actions.update()
>>> list(actions.keys())
[]
>>> list(actions.values())
[]
```

This feature is very helpful in multi-forms and wizards.

1.5.4 Customizing the Title

As for widgets, it is often desirable to change attributes of the button actions without altering any original code. Again we will be using attribute value adapters to complete the task. Originally, our title is as follows:

```
>>> myform = Form()
>>> actions = button.ButtonActions(myform, TestRequest(), None)
>>> actions.update()
>>> actions['apply'].title
'Apply'
```

Let's now create a custom label for the action:

```
>>> ApplyLabel = button.StaticButtonActionAttribute(
...     'Apply now', button=myform.buttons['apply'])
>>> zope.component.provideAdapter(ApplyLabel, name='title')
```

Once the button action manager is updated, the new title is chosen:

```
>>> actions.update()
>>> actions['apply'].title
'Apply now'
```

1.5.5 The Button Manager

The button manager contains several additional API methods that make the management of buttons easy.

First, you are able to add button managers:

```
>>> bm1 = button.Buttons(IButtons)
>>> bm2 = button.Buttons(button.Button('help', title='Help'))
```

```
>>> bm1 + bm2
Buttons([...])
>>> list(bm1 + bm2)
['apply', 'cancel', 'help']
```

The result of the addition is another button manager. Also note that the order of the buttons is preserved throughout the addition. Adding anything else is not well-defined:

```
>>> bm1 + 1
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Buttons' and 'int'
```

Second, you can select the buttons in a particular order:

```
>>> bm = bm1 + bm2
>>> list(bm)
['apply', 'cancel', 'help']
```

```
>>> list(bm.select('help', 'apply', 'cancel'))
['help', 'apply', 'cancel']
```

The `select()` method can also be used to eliminate another button:

```
>>> list(bm.select('help', 'apply'))
['help', 'apply']
```

Of course, in the example above we eliminated one and reorganized the buttons.

Third, you can omit one or more buttons:

```
>>> list(bm.omit('cancel'))
['apply', 'help']
```

Finally, while the constructor is very flexible, you cannot just pass in anything:

```
>>> button.Buttons(1, 2)
Traceback (most recent call last):
...
TypeError: ('Unrecognized argument type', 1)
```

When creating a new form derived from another, you often want to keep existing buttons and add new ones. In order not to change the super-form class, you need to copy the button manager:

```
>>> list(bm.keys())
['apply', 'cancel', 'help']
>>> list(bm.copy().keys())
['apply', 'cancel', 'help']
```

1.5.6 The Handlers Object

All handlers of a form are collected in the `handlers` attribute, which is a `Handlers` instance:

```
>>> isinstance(form.handlers, button.Handlers)
True
>>> form.handlers
<Handlers [<Handler for <Button 'apply' 'Apply'>>]>
```

Internally the object uses an adapter registry to manage the handlers for buttons. If a handler is registered for a button, it simply behaves as an instance-adapter.

The object itself is pretty simple. You can get a handler as follows:

```
>>> apply = form.buttons['apply']
>>> form.handlers.getHandler(apply)
<Handler for <Button 'apply' 'Apply'>>
```

But you can also register handlers for groups of buttons, either by interface or class:

```
>>> class SpecialButton(button.Button):
...     pass
```

```
>>> def handleSpecialButton(form, action):
...     return 'Special button action'
```

```
>>> form.handlers.addHandler(
...     SpecialButton, button.Handler(SpecialButton, handleSpecialButton))
```

```
>>> form.handlers
<Handlers
    [<Handler for <Button 'apply' 'Apply'>>,
     <Handler for <class 'SpecialButton'>>]>
```

Now all special buttons should use that handler:

```
>>> button1 = SpecialButton(name='button1', title='Button 1')
>>> button2 = SpecialButton(name='button2', title='Button 2')
```

```
>>> form.handlers.getHandler(button1)(form, None)
'Special button action'
>>> form.handlers.getHandler(button2)(form, None)
'Special button action'
```

However, registering a more specific handler for button 1 will override the general handler:

```
>>> def handleButton1(form, action):
...     return 'Button 1 action'
```

```
>>> form.handlers.addHandler(
...     button1, button.Handler(button1, handleButton1))
```

```
>>> form.handlers.getHandler(button1)(form, None)
'Button 1 action'
>>> form.handlers.getHandler(button2)(form, None)
'Special button action'
```

You can also add handlers objects:

```
>>> handlers2 = button.Handlers()
```

```
>>> button3 = SpecialButton(name='button3', title='Button 3')
>>> handlers2.addHandler(
...     button3, button.Handler(button3, None))
```

```
>>> form.handlers + handlers2
<Handlers
    [<Handler for <Button 'apply' 'Apply'>>,
     <Handler for <class 'SpecialButton'>>,
     <Handler for <SpecialButton 'button1' 'Button 1'>>,
     <Handler for <SpecialButton 'button3' 'Button 3'>>]>
```

However, adding other components is not supported:

```
>>> form.handlers + 1
Traceback (most recent call last):
...
NotImplementedError
```

The handlers also provide a method to copy the handlers to a new instance:

```
>>> copy = form.handlers.copy()
>>> isinstance(copy, button.Handlers)
True
>>> copy is form.handlers
False
```

This is commonly needed when one wants to extend the handlers of a super-form.

1.5.7 Image Buttons

A special type of button is the image button. Instead of creating a "submit"- or "button"-type input, an "image" button is created. An image button is a simple extension of a button, requiring an *image* argument to the constructor:

```
>>> imgSubmit = button.ImageButton(
...     name='submit',
...     title='Submit',
...     image='submit.png')
>>> imgSubmit
<ImageButton 'submit' 'submit.png'>
```

Some browsers do not submit the value of the input, but only the coordinates of the image where the mouse click occurred. Thus we also need a special button action:

```
>>> from zope.publisher.browser import TestRequest
>>> request = TestRequest()
```

```
>>> imgSubmitAction = button.ImageButtonAction(request, imgSubmit)
>>> imgSubmitAction
<ImageButtonAction 'submit' 'Submit'>
```

Initially, we did not click on the image:

```
>>> imgSubmitAction.isExecuted()
False
```

Now the button is clicked:

```
>>> request = TestRequest(form={'submit.x': '3', 'submit.y': '4'})
```

```
>>> imgSubmitAction = button.ImageButtonAction(request, imgSubmit)
>>> imgSubmitAction.isExecuted()
True
```

The "image" type of the "input"-element also requires there to be a *src* attribute, which is the URL to the image to be used. The attribute is also supported by the Python API. However, in order for the attribute to work, the image must be available as a resource, so let's do that now:

> # Traversing setup >>> from zope.traversing import testing >>> testing.setUp()
>
> # Resource namespace >>> import zope.component >>> from zope.traversing.interfaces import ITraversable >>> from zope.traversing.namespace import resource >>> zope.component.provideAdapter(... resource, (None,), ITraversable, name="resource") >>> zope.component.provideAdapter(... resource, (None, None), ITraversable, name="resource")
>
> # New absolute URL adapter for resources, if available >>> from zope.browserresource.resource import AbsoluteURL >>> zope.component.provideAdapter(AbsoluteURL)
>
> # Register the "submit.png" resource >>> from zope.browserresource.resource import Resource >>> testing.browserResource('submit.png', Resource)

Now the attribute can be called:

```
>>> imgSubmitAction.src
'http://127.0.0.1/@@/submit.png'
```

1.6 Directives

1.6.1 Widget template directive

Show how we can use the widget template directive. Register the meta configuration for the directive.

```
>>> import sys
>>> from zope.configuration import xmlconfig
>>> import z3c.form
>>> context = xmlconfig.file('meta.zcml', z3c.form)
```

We need a custom widget template

```
>>> import os, tempfile
>>> temp_dir = tempfile.mkdtemp()
>>> widget_file = os.path.join(temp_dir, 'widget.pt')
>>> with open(widget_file, 'w') as file:
...     _ = file.write('''
... <html xmlns="http://www.w3.org/1999/xhtml"
...       xmlns:tal="http://xml.zope.org/namespaces/tal"
...       tal:omit-tag="">
...    <input type="text" id="" name="" value="" size=""
...           tal:attributes="id view/id;
...                           name view/name;
...                           size view/size;
...                           value view/value;" />
... </html>
... ''')
```

and a interface

```
>>> import zope.interface
>>> from z3c.form import interfaces
>>> class IMyWidget(interfaces.IWidget):
...     """My widget interface."""
```

and a widget class:

```
>>> from z3c.form.testing import TestRequest
>>> from z3c.form.browser import text
>>> @zope.interface.implementer(IMyWidget)
... class MyWidget(text.TextWidget):
...     pass
>>> request = TestRequest()
>>> myWidget = MyWidget(request)
```

Make them available under the fake package `custom`:

```
>>> sys.modules['custom'] = type(
...     'Module', (),
...     {'IMyWidget': IMyWidget})()
```

and register them as a widget template within the `z3c:widgetTemplate` directive:

```
>>> context = xmlconfig.string("""
... <configure
...     xmlns:z3c="http://namespaces.zope.org/z3c">
...   <z3c:widgetTemplate
...       template="%s"
...       widget="custom.IMyWidget"
...       />
... </configure>
... """ % widget_file, context=context)
```

Let's get the template

```
>>> import zope.component
>>> from z3c.template.interfaces import IPageTemplate
>>> template = zope.component.queryMultiAdapter((None, request, None, None,
...     myWidget), interface=IPageTemplate, name='input')
```

and check it:

```
>>> from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile
>>> isinstance(template, ViewPageTemplateFile)
True
```

Let's use the template within the widget.

```
>>> print(template(myWidget))
<input type="text" value="" />
```

We normally render the widget which returns the registered template.

```
>>> print(myWidget.render())
<input type="text" value="" />
```

If the template does not exist, then the widget directive should fail immediately:

```
>>> unknownFile = os.path.join(temp_dir, 'unknown.pt')
>>> context = xmlconfig.string("""
... <configure
...     xmlns:z3c="http://namespaces.zope.org/z3c">
...   <z3c:widgetTemplate
...       template="%s"
...       widget="custom.IMyWidget"
...       />
... </configure>
... """ % unknownFile, context=context)
Traceback (most recent call last):
...
ConfigurationError: ('No such file', '...unknown.pt')
```

1.6.2 Object Widget template directive

Show how we can use the objectwidget template directive.

The big difference between the 'simple' Widget template and the Object Widget directive is that the Object Widget template takes the field's schema into account. That makes it easy to register different widget templates for different sub-schemas. You can use this together with SubformAdapter to get a totally custom subwidget.

We need a custom widget template

```
>>> widget_file = os.path.join(temp_dir, 'widget.pt')
>>> with open(widget_file, 'w') as file:
...     _ = file.write('''
... <html xmlns="http://www.w3.org/1999/xhtml"
...       xmlns:tal="http://xml.zope.org/namespaces/tal"
...       tal:omit-tag="">
...    <div class="object-widget" tal:attributes="class view/klass">
...    yeah, this can get complex
...    </div>
... </html>
... ''')
```

and a interface

```
>>> class IMyObjectWidget(interfaces.IObjectWidget):
...     """My objectwidget interface."""
```

and a widget class:

```
>>> from z3c.form.browser import object
>>> @zope.interface.implementer(IMyObjectWidget)
... class MyObjectWidget(object.ObjectWidget):
...     pass
>>> request = TestRequest()
>>> myObjectWidget = MyObjectWidget(request)
```

```
>>> from z3c.form.testing import IMySubObject
>>> import zope.schema
>>> field = zope.schema.Object(
...     __name__='subobject',
...     title=u'my object widget',
...     schema=IMySubObject)
>>> myObjectWidget.field = field
```

Make them available under the fake package `custom`:

```
>>> sys.modules['custom'] = type(
...     'Module', (),
...     {'IMyObjectWidget': IMyObjectWidget})()
```

and register them as a widget template within the `z3c:objectWidgetTemplate` directive:

```
>>> context = xmlconfig.string("""
... <configure
...     xmlns:z3c="http://namespaces.zope.org/z3c">
```

(continues on next page)

(continued from previous page)

```
...   <z3c:objectWidgetTemplate
...       template="%s"
...       widget="custom.IMyObjectWidget"
...       />
... </configure>
... """ % widget_file, context=context)
```

Let's get the template

```
>>> template = zope.component.queryMultiAdapter((None, request, None, None,
...     myObjectWidget, None), interface=IPageTemplate, name='input')
```

and check it:

```
>>> isinstance(template, ViewPageTemplateFile)
True
```

Let's use the template within the widget.

```
>>> print(template(myObjectWidget))
<div class="object-widget">yeah, this can get complex</div>
```

We normally render the widget which returns the registered template.

```
>>> print(myObjectWidget.render())
<div class="object-widget">yeah, this can get complex</div>
```

If the template does not exist, then the widget directive should fail immediately:

```
>>> unknownFile = os.path.join(temp_dir, 'unknown.pt')
>>> context = xmlconfig.string("""
... <configure
...     xmlns:z3c="http://namespaces.zope.org/z3c">
...   <z3c:objectWidgetTemplate
...       template="%s"
...       widget="custom.IMyObjectWidget"
...       />
... </configure>
... """ % unknownFile, context=context)
Traceback (most recent call last):
...
ConfigurationError: ('No such file', '...unknown.pt')
```

Register a specific template for a schema:

We need a custom widget template

```
>>> widgetspec_file = os.path.join(temp_dir, 'widgetspec.pt')
>>> with open(widgetspec_file, 'w') as file:
...     _ = file.write('''
... <html xmlns="http://www.w3.org/1999/xhtml"
...       xmlns:tal="http://xml.zope.org/namespaces/tal"
...       tal:omit-tag="">
...    <div class="object-widget" tal:attributes="class view/klass">
```

(continues on next page)

(continued from previous page)

```
...    this one is specific
...    </div>
... </html>
... ''')
```

```
>>> context = xmlconfig.string("""
... <configure
...     xmlns:z3c="http://namespaces.zope.org/z3c">
...   <z3c:objectWidgetTemplate
...       template="%s"
...       widget="custom.IMyObjectWidget"
...       schema="z3c.form.testing.IMySubObject"
...       />
... </configure>
... """ % widgetspec_file, context=context)
```

Let's get the template

```
>>> template = zope.component.queryMultiAdapter((None, request, None, None,
...     myObjectWidget, None), interface=IPageTemplate, name='input')
```

and check it:

```
>>> print(myObjectWidget.render())
<div class="object-widget">this one is specific</div>
```

1.6.3 Cleanup

Now we need to clean up the custom module.

```
>>> del sys.modules['custom']
```

Also let's not leave temporary files lying around

```
>>> import shutil
>>> shutil.rmtree(temp_dir)
```

CHAPTER

TWO

ADVANCED USERS

2.1 Validators

Validators are components that validate submitted data. This is certainly not a new concept, but in the previous form frameworks validation was hidden in many places:

- Field/Widget Validation

  The schema field consists of a `validate()` method. Validation is automatically invoked when converting a unicode string to a field value using `fromUnicode()`. This makes it very hard to customize the field validation. No hooks were provided to exert additional restriction at the presentation level.

- Schema/Form Validation

  This type of validation was not supported at all initially. `zope.formlib` fixed this problem by validating against schema invariants. While this was a first good step, it still made it hard to customize validators, since it required touching the base implementations of the forms.

- Action Validation

  `zope.formlib` supports the notion of action validators. Actions have a success and failure handler. If the validation succeeds, the success handler is called, otherwise the failure handler is chosen. We believe that this design was ill-conceived, especially the default, which required the data to completely validate in order for the action to successful. There are many actions that do not even care about the data in the form, such as "Help", "Cancel" and "Reset" buttons. Thus, validation should be part of the data retrieval process and not the action.

For me, the primary goals of the validator framework are as follows:

- Assert additional restrictions on the data at the presentation level.

  There are several use cases for this. Sometimes clients desire additional restrictions on data for their particular version of the software. It is not always desirable to adjust the model for this client, since the framework knows how to handle the less restrictive case anyways. In another case, additional restrictions might be applied to a particular form due to limited restrictions.

- Make validation pluggable.

  Like most other components of this package, it should be possible to control the validation adapters at a fine grained level.

  - Widgets: context, request, view, field[1], widget
  - Widget Managers: context, request, view, schema[2], manager

  [1].. This is optional, since widgets must not necessarily have fields. [2].. This is optional, since widget managers must not necessarily have manage field widgets and thus know about schemas.

- Provide good defaults that behave sensibly.

  Good defaults are, like in anywhere in this package, very important. We have chosen to implement the `zope.formlib` behavior as the default, since it worked very well – with exception of action validation, of course.

For this package, we have decided to support validators at the widget and widget manager level. By default, the framework only supports field widgets, since the validation of field-absent widgets is generally not well-defined. Thus, we first need to create a schema.

```
>>> import zope.interface
>>> import zope.schema
>>> class IPerson(zope.interface.Interface):
...     login = zope.schema.TextLine(
...         title=u'Login',
...         min_length=1,
...         max_length=10,
...         required=True)
...
...     email = zope.schema.TextLine(
...         title=u'E-mail')
...
...     @zope.interface.invariant
...     def isLoginPartOfEmail(person):
...         if not person.email.startswith(person.login):
...             raise zope.interface.Invalid("The login not part of email.")
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...     login = None
...     email = None
```

2.1.1 Widget Validators

Widget validators only validate the data of one particular widget. The validated value is always assumed to be an internal value and not a widget value.

By default, the system uses the simple field validator, which simply uses the `validate()` method of the field. For instantiation, all validators have the following signature for its discriminators: context, request, view, field, and widget

```
>>> from z3c.form import validator
>>> simple = validator.SimpleFieldValidator(
...     None, None, None, IPerson['login'], None)
```

A validator has a single method `validate()`. When the validation is successful, `None` is returned:

```
>>> simple.validate(u'srichter')
```

A validation error is raised, when the validation fails:

```
>>> simple.validate(u'StephanCaveman3')
Traceback (most recent call last):
...
TooLong: (u'StephanCaveman3', 10)
```

Let's now create a validator that also requires at least 1 numerical character in the login name:

```
>>> import re
>>> class LoginValidator(validator.SimpleFieldValidator):
...
...     def validate(self, value):
...         super(LoginValidator, self).validate(value)
...         if re.search('[0-9]', value) is None:
...             raise zope.interface.Invalid('No numerical character found.')
```

Let's now try our new validator:

```
>>> login = LoginValidator(None, None, None, IPerson['login'], None)
```

```
>>> login.validate(u'srichter1')
```

```
>>> login.validate(u'srichter')
Traceback (most recent call last):
...
Invalid: No numerical character found.
```

We can now register the validator with the component architecture, ...

```
>>> import zope.component
>>> zope.component.provideAdapter(LoginValidator)
```

and look up the adapter using the usual way:

```
>>> from z3c.form import interfaces
```

```
>>> zope.component.queryMultiAdapter(
...     (None, None, None, IPerson['login'], None),
...     interfaces.IValidator)
<LoginValidator for IPerson['login']>
```

Unfortunately, the adapter is now registered for all fields, so that the E-mail field also has this restriction (which is okay in this case, but not generally):

```
>>> zope.component.queryMultiAdapter(
...     (None, None, None, IPerson['email'], None),
...     interfaces.IValidator)
<LoginValidator for IPerson['email']>
```

The validator module provides a helper function to set the discriminators for a validator, which can include instances:

```
>>> validator.WidgetValidatorDiscriminators(
...     LoginValidator, field=IPerson['login'])
```

Let's now clean up the component architecture and register the login validator again:

```
>>> from zope.testing import cleanup
>>> cleanup.cleanUp()
```

```
>>> zope.component.provideAdapter(LoginValidator)
```

```
>>> zope.component.queryMultiAdapter(
...     (None, None, None, IPerson['login'], None),
...     interfaces.IValidator)
<LoginValidator for IPerson['login']>
```

```
>>> zope.component.queryMultiAdapter(
...     (None, None, None, IPerson['email'], None),
...     interfaces.IValidator)
```

Ignoring unchanged values

Most of the time we want to ignore unchanged fields/values at validation. A common usecase for this is if a value went away from a vocabulary and we want to keep the old value after editing. In case you want to strict behaviour, register `StrictSimpleFieldValidator` for your layer.

```
>>> simple = validator.SimpleFieldValidator(
...     None, None, None, IPerson['login'], None)
```

NOT_CHANGED never gets validated.

```
>>> simple.validate(interfaces.NOT_CHANGED)
```

Current value gets extracted by `IDataManager` via the widget, field and context

```
>>> from z3c.form.datamanager import AttributeField
>>> zope.component.provideAdapter(AttributeField)
```

```
>>> import z3c.form.testing
>>> request = z3c.form.testing.TestRequest()
>>> import z3c.form.widget
>>> widget = z3c.form.widget.Widget(request)
>>> context = Person()
```

```
>>> widget.context = context
>>> from z3c.form import interfaces
>>> zope.interface.alsoProvides(widget, interfaces.IContextAware)
```

```
>>> simple = validator.SimpleFieldValidator(
...     context, request, None, IPerson['login'], widget)
```

OK, let's see checking after setup. Works like a StrictSimpleFieldValidator until we have to validate a different value:

```
>>> context.login = u'john'
>>> simple.validate(u'carter')
>>> simple.validate(u'hippocratiusxy')
Traceback (most recent call last):
...
TooLong: (u'hippocratiusxy', 10)
```

Validating the unchanged value works despite it would be an error.

```
>>> context.login = u'hippocratiusxy'
>>> simple.validate(u'hippocratiusxy')
```

Unless we want to force validation:

```
>>> simple.validate(u'hippocratiusxy', force=True)
Traceback (most recent call last):
...
TooLong: (u'hippocratiusxy', 10)
```

Some exceptions:

`missing_value` gets validated

```
>>> simple.validate(IPerson['login'].missing_value)
Traceback (most recent call last):
...
RequiredMissing: login
```

Widget Validators and File-Uploads

File-Uploads behave a bit different than the other form elements. Whether the user did not choose a file to upload `interfaces.NOT_CHANGED` is set as value. But the validator knows how to handle this.

The example has two bytes fields where File-Uploads are possible, one field is required the other one not:

```
>>> class IPhoto(zope.interface.Interface):
...     data = zope.schema.Bytes(
...         title=u'Photo',
...         required=True)
...
...     thumb = zope.schema.Bytes(
...         title=u'Thumbnail',
...         required=False)
```

There are several possible cases to differentiate between:

No widget

If there is no widget or the widget does not provide `interfaces.IContextAware`, no value is looked up from the context. So the not required field validates successfully but the required one has a required missing error, as the default value of the field is looked up on the field:

```
>>> simple_thumb = validator.StrictSimpleFieldValidator(
...     None, None, None, IPhoto['thumb'], None)
>>> simple_thumb.validate(interfaces.NOT_CHANGED)
```

```
>>> simple_data = validator.StrictSimpleFieldValidator(
...     None, None, None, IPhoto['data'], None)
>>> simple_data.validate(interfaces.NOT_CHANGED)
Traceback (most recent call last):
RequiredMissing: data
```

Widget which ignores context

If the context is ignored in the widget - as in the add form - the behavior is the same as if there was no widget:

```
>>> import z3c.form.widget
>>> widget = z3c.form.widget.Widget(None)
>>> zope.interface.alsoProvides(widget, interfaces.IContextAware)
>>> widget.ignoreContext = True
>>> simple_thumb = validator.StrictSimpleFieldValidator(
...     None, None, None, IPhoto['thumb'], widget)
>>> simple_thumb.validate(interfaces.NOT_CHANGED)
```

```
>>> simple_data = validator.StrictSimpleFieldValidator(
...     None, None, None, IPhoto['data'], widget)
>>> simple_data.validate(interfaces.NOT_CHANGED)
Traceback (most recent call last):
RequiredMissing: data
```

Look up value from default adapter

When the value is `interfaces.NOT_CHANGED` the validator tries to look up the default value using a `interfaces.IValue` adapter. Whether the adapter is found, its value is used as default, so the validation of the required field is successful here:

```
>>> data_default = z3c.form.widget.StaticWidgetAttribute(
...     b'data', context=None, request=None, view=None,
...     field=IPhoto['data'], widget=widget)
>>> zope.component.provideAdapter(data_default, name='default')
>>> simple_data.validate(interfaces.NOT_CHANGED)
```

Look up value from context

If there is a context aware widget which does not ignore its context, the value is looked up on the context using a data manager:

```
>>> @zope.interface.implementer(IPhoto)
... class Photo(object):
...     data = None
...     thumb = None
>>> photo = Photo()
>>> widget.ignoreContext = False
>>> zope.component.provideAdapter(z3c.form.datamanager.AttributeField)
```

```
>>> simple_thumb = validator.StrictSimpleFieldValidator(
...     photo, None, None, IPhoto['thumb'], widget)
>>> simple_thumb.validate(interfaces.NOT_CHANGED)
```

If the value is not set on the context it is a required missing as neither context nor input have a valid value:

```
>>> simple_data = validator.StrictSimpleFieldValidator(
...     photo, None, None, IPhoto['data'], widget)
>>> simple_data.validate(interfaces.NOT_CHANGED)
Traceback (most recent call last):
RequiredMissing: data
```

After setting the value validation is successful:

```
>>> photo.data = b'data'
>>> simple_data.validate(interfaces.NOT_CHANGED)
```

Clean-up

```
>>> gsm = zope.component.getGlobalSiteManager()
>>> gsm.unregisterAdapter(z3c.form.datamanager.AttributeField)
True
>>> gsm.unregisterAdapter(data_default, name='default')
True
```

Ignoring required

Sometimes we want to ignore `required` checking. That's because we want to have *all* fields extracted from the form regardless whether required fields are filled. And have no required-errors displayed.

```
>>> class IPersonRequired(zope.interface.Interface):
...     login = zope.schema.TextLine(
...         title=u'Login',
...         required=True)
...
...     email = zope.schema.TextLine(
...         title=u'E-mail')
```

```
>>> simple = validator.SimpleFieldValidator(
...     None, None, None, IPersonRequired['login'], None)
```

```
>>> simple.validate(None)
Traceback (most recent call last):
...
RequiredMissing: login
```

Ooops we need a widget too.

```
>>> widget = z3c.form.widget.Widget(None)
>>> widget.field = IPersonRequired['login']
```

```
>>> simple = validator.SimpleFieldValidator(
...     None, None, None, IPersonRequired['login'], widget)
```

```
>>> simple.validate(None)
Traceback (most recent call last):
...
RequiredMissing: login
```

Meeeh, need to signal that we need to ignore `required`:

```
>>> widget.ignoreRequiredOnValidation = True
```

```
>>> simple.validate(None)
```

2.1.2 Widget Manager Validators

The widget manager validator, while similar in spirit, works somewhat different. The discriminators of the widget manager validator are: context, request, view, schema, and manager.

A simple default implementation is provided that checks the invariants of the schemas:

```
>>> invariants = validator.InvariantsValidator(
...     None, None, None, IPerson, None)
```

Widget manager validators have the option to validate a data dictionary,

```
>>> invariants.validate(
...     {'login': u'srichter', 'email': u'srichter@foo.com'})
()
```

or an object implementing the schema:

```
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...     login = u'srichter'
...     email = u'srichter@foo.com'
>>> stephan = Person()
```

```
>>> invariants.validateObject(stephan)
()
```

Since multiple errors can occur during the validation process, all errors are collected in a tuple, which is returned. If the tuple is empty, the validation was successful. Let's now generate a failure:

```
>>> errors = invariants.validate(
...     {'login': u'srichter', 'email': u'strichter@foo.com'})
```

```
>>> for e in errors:
...     print(e.__class__.__name__ + ':', e)
Invalid: The login not part of email.
```

Let's now have a look at writing a custom validator. In this case, we want to ensure that the E-mail address is at most twice as long as the login:

```
>>> class CustomValidator(validator.InvariantsValidator):
...     def validateObject(self, obj):
...         errors = super(CustomValidator, self).validateObject(obj)
...         if len(obj.email) > 2 * len(obj.login):
...             errors += (zope.interface.Invalid('Email too long.'),)
...         return errors
```

Since the `validate()` method of `InvatiantsValidator` simply uses `validateObject()` it is enough to only override `validateObject()`. Now we can use the validator:

```
>>> custom = CustomValidator(
...     None, None, None, IPerson, None)
```

```
>>> custom.validate(
...     {'login': u'srichter', 'email': u'srichter@foo.com'})
()
>>> errors = custom.validate(
...     {'login': u'srichter', 'email': u'srichter@foobar.com'})
>>> for e in errors:
...     print(e.__class__.__name__ + ':', e)
Invalid: Email too long.
```

To register the custom validator only for this schema, we have to use the discriminator generator again.

```
>>> from z3c.form import util
>>> validator.WidgetsValidatorDiscriminators(
...     CustomValidator, schema=util.getSpecification(IPerson, force=True))
```

Note: Of course we could have used the `zope.component.adapts()` function
from within the class, but I think it is too tedious, since you have to specify all discriminators and not only the specific ones you are interested in.

After registering the validator,

```
>>> zope.component.provideAdapter(CustomValidator)
```

it becomes the validator for this schema:

```
>>> zope.component.queryMultiAdapter(
...     (None, None, None, IPerson, None), interfaces.IManagerValidator)
<CustomValidator for IPerson>
```

```
>>> class ICar(zope.interface.Interface):
...     pass
>>> zope.component.queryMultiAdapter(
...     (None, None, None, ICar, None), interfaces.IManagerValidator)
```

2.1.3 The Data Wrapper

The Data class provides a wrapper to present a dictionary as a class instance. This is used to check for invariants, which always expect an object. While the common use cases of the data wrapper are well tested in the code above, there are some corner cases that need to be addressed.

So let's start by creating a data object:

```
>>> context = object()
>>> data = validator.Data(IPerson, {'login': 'srichter', 'other': 1}, context)
```

When we try to access a name that is not in the schema, we get an attribute error:

```
>>> data.address
Traceback (most recent call last):
...
AttributeError: address
```

```
>>> data.other
Traceback (most recent call last):
...
AttributeError: other
```

If the field found is a method, then a runtime error is raised:

```
>>> class IExtendedPerson(IPerson):
...     def compute():
...         """Compute something."""
```

```
>>> data = validator.Data(IExtendedPerson, {'compute': 1}, context)
>>> data.compute
Traceback (most recent call last):
...
RuntimeError: ('Data value is not a schema field', 'compute')
```

Finally, the context is available as attribute directly:

```
>>> data.__context__ is context
True
```

It is used by the validators (especially invariant validators) to provide a context of validation, for example to look up a vocabulary or access the parent of an object. Note that the context will be different between add and edit forms.

2.1.4 Validation of interface variants when not all fields are displayed in form

We need to register the data manager to access the data on the context object:

```
>>> from z3c.form import datamanager
>>> zope.component.provideAdapter(datamanager.AttributeField)
```

Sometimes you might leave out fields in the form which need to compute the invariant. An exception should be raised. The data wrapper is used to test the invariants and looks up values on the context object that are left out in the form.

```
>>> invariants = validator.InvariantsValidator(
...     stephan, None, None, IPerson, None)
>>> errors = invariants.validate({'email': 'foo@bar.com'})
>>> errors[0].__class__.__name__
'Invalid'
>>> errors[0].args[0]
'The login not part of email.'
```

2.2 Widgets

Widgets are small UI components that accept and process the textual user input. The only responsibility of a widget is to represent a value to the user, allow it to be modified and then return a new value. Good examples of widgets include the Qt widgets and HTML widgets. The widget is not responsible for converting its value to the desired internal value or validate the incoming data. These responsibilities are passed data converters and validators, respectively.

There are several problems that can be identified in the original Zope 3 widget implementation located at `zope.app.form`.

(1) Field Dependence – Widgets are always views of fields. While this might be a correct choice for a high-level API, it is fundamentally wrong. It disallows us to use widgets without defining fields. This also couples certain pieces of information too tightly to the field, especially, value retrieval from and storage to the context, validation and raw data conversion.

(2) Form Dependence – While widgets do not have to be located within a form, they are usually tightly coupled to it. It is very difficult to use widgets outside the context of a form.

(3) Traversability – Widgets cannot be traversed, which means that they cannot interact easily using Javascript. This is not a fundamental problem, but simply a lack of the current design to recognize that small UI components must also be traversable and thus have a URI.

(4) Customizability – A consequence of issue (1) is that widgets are not customizable enough. Implementing real-world projects has shown that widgets often want a very fine-grained ability to customize values. A prime example is the label. Because the label of a widget is retrieved from the field title, it is impossible to provide an alternative label for a widget. While the label could be changed from the form, this would require rewriting the entire form to change a label. Instead, we often endde up writing cusom schemas.

(5) Flexibility – Oftentimes it is desired to have one widget, but multiple styles of representation. For example, in one scenario the widget uses a plain HTML widget and in another a fancy JavaScript widget is used. The current implementation makes it very hard to provide alternative styles for a widget.

2.2.1 Creating and Using Simple Widgets

When using the widget API by itself, the simplest way to use it is to just instantiate it using the request:

```
>>> from z3c.form.testing import TestRequest
>>> from z3c.form import widget
>>> request = TestRequest()
>>> age = widget.Widget(request)
```

In this case we instantiated a generic widget. A full set of simple browser-based widgets can be found in the `browser/` package. Since no helper components are around to fill the attributes of the widget, we have to do it by hand:

```
>>> age.name = 'age'
>>> age.label = 'Age'
>>> age.value = '39'
```

The most important attributes are the "name" and the "value". The name is used to identify the widget within the form. The value is either the value to be manipulated or the default value. The value must be provided in the form the widget needs it. It is the responsibility of a data converter to convert between the widget value and the desired internal value.

Before we can render the widget, we have to register a template for the widget. The first step is to define the template:

```
>>> import tempfile
>>> textWidgetTemplate = tempfile.mktemp('text.pt')
>>> with open(textWidgetTemplate, 'w') as file:
...     _ = file.write('''
... <html xmlns="http://www.w3.org/1999/xhtml"
...       xmlns:tal="http://xml.zope.org/namespaces/tal"
...       tal:omit-tag="">
...    <input type="text" name="" value=""
...           tal:attributes="name view/name; value view/value;" />
... </html>
... ''')
```

Next, we have to create a template factory for the widget:

```
>>> from z3c.form.widget import WidgetTemplateFactory
>>> factory = WidgetTemplateFactory(
...     textWidgetTemplate, widget=widget.Widget)
```

The first argument, which is also required, is the path to the template file. An optional `content_type` keyword argument allows the developer to specify the output content type, usually "text/html". Then there are five keyword arguments that specify the discriminators of the template:

- `context` – This is the context in which the widget is displayed. In a simple widget like the one we have now, the context is `None`.
- `request` – This discriminator allows you to specify the type of request for which the widget will be available. In our case this would be a browser request. Note that browser requests can be further broken into layer, so you could also specify a layer interface here.
- `view` – This is the view from which the widget is used. The simple widget at hand, does not have a view associated with it though.
- `field` – This is the field for which the widget provides a representation. Again, this simple widget does not use a field, so it is `None`.
- `widget` – This is the widget itself. With this discriminator you can specify for which type of widget you are providing a template.

We can now register the template factory. The name of the factory is the mode of the widget. By default, there are two widget modes: "input" and "display". However, since the mode is just a string, one can develop other kinds of modes as needed for a project. The default mode is "input":

```
>>> from z3c.form import interfaces
>>> age.mode is interfaces.INPUT_MODE
True
```

```
>>> import zope.component
>>> zope.component.provideAdapter(factory, name=interfaces.INPUT_MODE)
```

Once everything is set up, the widget is updated and then rendered:

```
>>> age.update()
>>> print(age.render())
<input type="text" name="age" value="39" />
```

If a value is found in the request, it takes precedence, since the user entered the value:

```
>>> age.request = TestRequest(form={'age': '25'})
>>> age.update()
>>> print(age.render())
<input type="text" name="age" value="25" />
```

However, there is an option to turn off all request data:

```
>>> age.value = '39'
>>> age.ignoreRequest = True
>>> age.update()
>>> print(age.render())
<input type="text" name="age" value="39" />
```

Additionally the widget provides a dictionary representation of its data through a json_data() method:

```
>>> from pprint import pprint
>>> pprint(age.json_data())
{'error': '',
 'id': '',
 'label': 'Age',
 'mode': 'input',
 'name': 'age',
 'required': False,
 'type': 'text',
 'value': '39'}
```

2.2.2 Creating and Using Field Widgets

An extended form of the widget allows fields to control several of the widget's properties. Let's create a field first:

```
>>> ageField = zope.schema.Int(
...     __name__ = 'age',
...     title = 'Age',
...     min = 0,
...     max = 130)
```

We can now use our simple widget and create a field widget from it:

```
>>> ageWidget = widget.FieldWidget(ageField, age)
```

Such a widget provides `IFieldWidget`:

```
>>> interfaces.IFieldWidget.providedBy(ageWidget)
True
```

Of course, this is more commonly done using an adapter. Commonly those adapters look like this:

```
>>> @zope.component.adapter(zope.schema.Int, TestRequest)
... @zope.interface.implementer(interfaces.IFieldWidget)
... def IntWidget(field, request):
...     return widget.FieldWidget(field, widget.Widget(request))
```

```
>>> zope.component.provideAdapter(IntWidget)
>>> ageWidget = zope.component.getMultiAdapter((ageField, request),
...     interfaces.IFieldWidget)
```

Now we just have to update and render the widget:

```
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" />
```

There is no initial value for the widget, since there is no value in the request and the field does not provide a default. Let's now give our field a default value and see what happens:

```
>>> ageField.default = 30
>>> ageWidget.update()
Traceback (most recent call last):
...
TypeError: ('Could not adapt', <Widget 'age'>,
            <InterfaceClass z3c.form.interfaces.IDataConverter>)
```

In order for the widget to be able to take the field's default value and use it to provide an initial value the widget, we need to provide a data converter that defines how to convert from the field value to the widget value.

```
>>> from z3c.form import converter
>>> zope.component.provideAdapter(converter.FieldWidgetDataConverter)
>>> zope.component.provideAdapter(converter.FieldDataConverter)
```

```
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" value="30" />
```

Again, the request value is honored above everything else:

```
>>> ageWidget.request = TestRequest(form={'age': '25'})
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" value="25" />
```

2.2.3 Creating and Using Context Widgets

When widgets represent an attribute value of an object, then this object must be set as the context of the widget:

```
>>> class Person(object):
...     age = 45
>>> person = Person()
```

```
>>> ageWidget.context = person
>>> zope.interface.alsoProvides(ageWidget, interfaces.IContextAware)
```

The result is that the context value takes over precendence over the default value:

```
>>> ageWidget.request = TestRequest()
>>> ageWidget.update()
Traceback (most recent call last):
...
ComponentLookupError: ((...), <InterfaceClass ...IDataManager>, '')
```

This call fails because the widget does not know how to extract the value from the context. Registering a data manager for the widget does the trick:

```
>>> from z3c.form import datamanager
>>> zope.component.provideAdapter(datamanager.AttributeField)
```

```
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" value="45" />
```

If the context value is unknown (None), the default value kicks in.

```
>>> person.age = None
```

```
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" value="30" />
```

Unless the widget is explicitely asked to not to show defaults. This is handy for EditForms.

```
>>> ageWidget.showDefault = False
```

```
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" value="" />
```

```
>>> ageWidget.showDefault = True
>>> person.age = 45
```

The context can be explicitely ignored, making the widget display the default value again:

```
>>> ageWidget.ignoreContext = True
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" value="30" />
```

Again, the request value is honored above everything else:

```
>>> ageWidget.request = TestRequest(form={'age': '25'})
>>> ageWidget.ignoreContext = False
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" value="25" />
```

But what happens if the object we are working on is security proxied? In particular, what happens, if the access to the attribute is denied. To see what happens, we have to create a proxied person:

```
>>> from zope.security import checker
>>> PersonChecker = checker.Checker({'age': 'Access'}, {'age': 'Edit'})
```

```
>>> ageWidget.request = TestRequest()
>>> ageWidget.context = checker.ProxyFactory(Person(), PersonChecker)
```

After changing the security policy, ...

```
>>> from zope.security import management
>>> from z3c.form import testing
>>> management.endInteraction()
>>> newPolicy = testing.SimpleSecurityPolicy()
>>> oldPolicy = management.setSecurityPolicy(newPolicy)
>>> management.newInteraction()
```

it is not possible anymore to update the widget:

```
>>> ageWidget.update()
Traceback (most recent call last):
...
Unauthorized: (<Person object at ...>, 'age', 'Access')
```

If no security declaration has been made at all, we get a `ForbiddenAttribute` error:

```
>>> ageWidget.context = checker.ProxyFactory(Person(), checker.Checker({}))
>>> ageWidget.update()
Traceback (most recent call last):
...
ForbiddenAttribute: ('age', <Person object at ...>)
```

Let's clean up the setup:

```
>>> management.endInteraction()
>>> newPolicy = management.setSecurityPolicy(oldPolicy)
>>> management.newInteraction()
```

```
>>> ageWidget.context = Person()
```

2.2.4 Dynamically Changing Attribute Values

Once widgets are used within a framework, it is very tedious to write Python code to adjust certain attributes, even though hooks exist. The easiest way to change those attribute values is actually to provide an adapter that provides the custom value.

We can create a custom label for the age widget:

```
>>> AgeLabel = widget.StaticWidgetAttribute(
...     'Current Age',
...     context=None, request=None, view=None, field=ageField, widget=None)
```

Clearly, this code does not require us to touch the orginal form and widget code, given that we have enough control over the selection. In the example above, all the selection discriminators are listed for demonstration purposes. Of course, the label in this case can be created as follows:

```
>>> AgeLabel = widget.StaticWidgetAttribute('Current Age', field=ageField)
```

Much better, isn't it? Initially the label is the title of the field:

```
>>> ageWidget.label
'Age'
```

Let's now simply register the label as a named adapter; the name is the name of the attribute to change:

```
>>> zope.component.provideAdapter(AgeLabel, name='label')
```

Asking the widget for the label now will return the newly registered label:

```
>>> ageWidget.update()
>>> ageWidget.label
'Current Age'
```

Of course, simply setting the label or changing the label extraction via a sub-class are other options you might want to consider. Furthermore, you could also create a computed attribute value or implement your own component.

Overriding other attributes, such as `required`, is done in the same way. If any widget provides new attributes, they are also overridable this way. For example, the selection widget defines a label for the option that no value was selected. We often want to override this, because the German translation sucks or the wording is often too generic. Widget implementation should add names of overridable attributes to their "_adapterValueAttributes" internal attribute.

Let's try to override the `required` attribute. By default the widget is required, because the field is required as well:

```
>>> ageWidget.required
True
```

Let's provide a static widget attribute adapter with name "required":

```
>>> AgeNotRequired = widget.StaticWidgetAttribute(False, field=ageField)
>>> zope.component.provideAdapter(AgeNotRequired, name="required")
```

Now, let's check if it works:

```
>>> ageWidget.update()
>>> ageWidget.required
False
```

Overriding the default value is somewhat special due to the complexity of obtaining the value. So let's register one now:

```
>>> AgeDefault = widget.StaticWidgetAttribute(50, field=ageField)
>>> zope.component.provideAdapter(AgeDefault, name="default")
```

Let's now instantiate, update and render the widget to see the default value:

```
>>> ageWidget = zope.component.getMultiAdapter((ageField, request),
...     interfaces.IFieldWidget)
>>> ageWidget.update()
>>> print(ageWidget.render())
<input type="text" name="age" value="50" />
```

This value is also respected by the json_data method:

```
>>> from pprint import pprint
>>> pprint(ageWidget.json_data())
{'error': '',
 'id': 'age',
 'label': 'Current Age',
 'mode': 'input',
 'name': 'age',
 'required': False,
 'type': 'text',
 'value': '50'}
```

2.2.5 Sequence Widget

A common use case in user interfaces is to ask the user to select one or more items from a set of options/choices. The `widget` module provides a basic widget implementation to support this use case.

The options available for selections are known as terms. Initially, there are no terms:

```
>>> request = TestRequest()
>>> seqWidget = widget.SequenceWidget(request)
>>> seqWidget.name = 'seq'
```

```
>>> seqWidget.terms is None
True
```

There are two ways terms can be added, either manually or via an adapter. Those term objects must provide `ITerms`. There is no simple default implementation, so we have to provide one ourselves:

```
>>> from zope.schema import vocabulary
>>> @zope.interface.implementer(interfaces.ITerms)
... class Terms(vocabulary.SimpleVocabulary):
...     def getValue(self, token):
...         return self.getTermByToken(token).value
```

```
>>> terms = Terms(
...   [Terms.createTerm(1, 'v1', 'Value 1'),
...    Terms.createTerm(2, 'v2', 'Value 2'),
```

(continues on next page)

(continued from previous page)

```
...     Terms.createTerm(3, 'v3', 'Value 3')])
>>> seqWidget.terms = terms
```

Once the `terms` attribute is set, updating the widgets does not change the terms:

```
>>> seqWidget.update()
>>> [term.value for term in seqWidget.terms]
[1, 2, 3]
```

The value of a sequence widget is a tuple/list of term tokens. When extracting values from the request, the values must be valid tokens, otherwise the default value is returned:

```
>>> seqWidget.request = TestRequest(form={'seq': ['v1']})
>>> seqWidget.extract()
('v1',)
```

```
>>> seqWidget.request = TestRequest(form={'seq': ['v4']})
>>> seqWidget.extract()
<NO_VALUE>
```

```
>>> seqWidget.request = TestRequest(form={'seq-empty-marker': '1'})
>>> seqWidget.extract()
()
```

Note that we also support single values being returned outside a sequence. The extracted value is then wrapped by a tuple. This feature is useful when integrating with third-party client frameworks that do not know about the Zope naming conventions.

```
>>> seqWidget.request = TestRequest(form={'seq': 'v1'})
>>> seqWidget.extract()
('v1',)
```

If the no-value token has been selected, it is returned without further verification:

```
>>> seqWidget.request = TestRequest(form={'seq': [seqWidget.noValueToken]})
>>> seqWidget.extract()
('--NOVALUE--',)
```

Since the value of the widget is a tuple of tokens, when displaying the values, they have to be converted to the title of the term:

```
>>> seqWidget.value = ('v1', 'v2')
>>> seqWidget.displayValue
['Value 1', 'Value 2']
```

Unknown values/terms get silently ignored.

```
>>> seqWidget.value = ('v3', 'v4')
>>> seqWidget.displayValue
['Value 3']
```

When input forms are directly switched to display forms within the same request, it can happen that the value contains the "–NOVALUE–" token entry. This entry should be silently ignored:

```
>>> seqWidget.value = (seqWidget.noValueToken,)
>>> seqWidget.displayValue
[]
```

To demonstrate how the terms is automatically chosen by a widget, we should instantiate a field widget. Let's do this with a choice field:

```
>>> seqField = zope.schema.Choice(
...     title='Sequence Field',
...     vocabulary=terms)
```

Let's now create the field widget:

```
>>> seqWidget = widget.FieldWidget(seqField, widget.SequenceWidget(request))
>>> seqWidget.terms
```

The terms should be available as soon as the widget is updated:

```
>>> seqWidget.update()
Traceback (most recent call last):
...
ComponentLookupError: ((...), <InterfaceClass ...ITerms>, '')
```

This failed, because we did not register an adapter for the terms yet. After the adapter is registered, everything should work as expected:

```
>>> from z3c.form import term
>>> zope.component.provideAdapter(term.ChoiceTermsVocabulary)
>>> zope.component.provideAdapter(term.ChoiceTerms)
```

```
>>> seqWidget.update()
>>> seqWidget.terms
<z3c.form.term.ChoiceTermsVocabulary object at ...>
```

The representation of this widget as json looks a bit different:

```
>>> from pprint import pprint
>>> pprint(seqWidget.json_data())
{'error': '',
 'id': '',
 'label': 'Sequence Field',
 'mode': 'input',
 'name': '',
 'required': True,
 'type': 'sequence',
 'value': ()}
```

So that's it. Everything else is the same from then on.

2.2.6 Multi Widget

A common use case in user interfaces is to ask the user to define one or more items. The `widget` module provides a basic widget implementation to support this use case.

The *MultiWidget* allows to store none, one or more values for a sequence or dictionary field. Don't get confused by the term sequence. The sequence used in *SequenceWidget* means that the widget can choose from a sequence of values which is really a collection. The *MultiWidget* can collect values to build and store a sequence of values like those used in *ITuple* or *IList* field.

```
>>> request = TestRequest()
>>> multiWidget = widget.MultiWidget(request)
>>> multiWidget.name = 'multi.name'
>>> multiWidget.id = 'multi-id'
```

```
>>> multiWidget.value
[]
```

Let's define a field for our multi widget:

```
>>> multiField = zope.schema.List(
...     value_type=zope.schema.Int(default=42))
>>> multiWidget.field = multiField
```

If the multi is used with a schema.List the value of a multi widget is always list. When extracting values from the request, the values must be a list of valid values based on the value_type field used from the used sequence field. The widget also uses a counter which is required for processing the input from a request. The counter is a marker for build the right amount of enumerated widgets.

If we provide no request we will get no value:

```
>>> multiWidget.extract()
<NO_VALUE>
```

If we provide an empty counter we will get an empty list. This is accordance with Widget.extract(), where a missing request value is <NO_VALUE> and an empty ('') request value is ''.

```
>>> multiWidget.request = TestRequest(form={'multi.name.count':'0'})
>>> multiWidget.extract()
[]
```

If we provide real values within the request, we will get it back:

```
>>> multiWidget.request = TestRequest(form={'multi.name.count':'2',
...                                         'multi.name.0':'42',
...                                         'multi.name.1':'43'})
>>> multiWidget.extract()
['42', '43']
```

If we provide a bad value we will get the bad value within the extract method. Our widget update process will validate this bad value later:

```
>>> multiWidget.request = TestRequest(form={'multi.name.count':'1',
...                                         'multi.name.0':'bad'})
>>> multiWidget.extract()
['bad']
```

Storing a widget value forces to update the (sub) widgets. This forces also to validate the (sub) widget values. To show this we need to register a validator:

```
>>> from z3c.form.validator import SimpleFieldValidator
>>> zope.component.provideAdapter(SimpleFieldValidator)
```

Since the value of the widget is a list of (widget) value items, when displaying the values, they can be used as they are:

```
>>> multiWidget.request = TestRequest(form={'multi.name.count':'2',
...                                         'multi.name.0':'42',
...                                         'multi.name.1':'43'})
>>> multiWidget.value = multiWidget.extract()
>>> multiWidget.value
['42', '43']
```

Each widget normally gets first processed by it's update method call after initialization. This update call forces to call extract, which first will get the right amount of (sub) widgets by the given counter value. Based on that counter value the right amount of widgets will get created. Each widget will return it's own value and this collected values get returned by the extract method. The multi widget update method will then store this values if any given as multi widget value argument. If extract doesn't return a value the multi widget update method will use it's default value. If we store a given value from the extract as multi widget value, this will force to setup the multi widget widgets based on the given values and apply the right value for them. After that the multi widget is ready for rendering. The good thing about that pattern is that it is possible to set a value before or after the update method is called. At any time if we change the multi widget value the (sub) widgets get updated within the new relevant value.

```
>>> multiRequest = TestRequest(form={'multi.name.count':'2',
...                                  'multi.name.0':'42',
...                                  'multi.name.1':'43'})
```

```
>>> multiWidget = widget.FieldWidget(multiField, widget.MultiWidget(
...     multiRequest))
>>> multiWidget.name = 'multi.name'
>>> multiWidget.value
[]
```

```
>>> multiWidget.update()
```

```
>>> multiWidget.widgets[0].value
'42'
```

```
>>> multiWidget.widgets[1].value
'43'
```

```
>>> multiWidget.value
['42', '43']
```

MultiWidget also declares the `allowAdding` and `allowRemoving` attributes that can be used in browser presentation to control add/remove button availability. To ease working with common cases, the `updateAllowAddRemove` method provided that will set those attributes in respect to field's min_length and max_length, if the field provides zope.schema.interfaces.IMinMaxLen interface.

Let's define a field with min and max length constraints and create a widget for it.

```
>>> multiField = zope.schema.List(
...     value_type=zope.schema.Int(),
...     min_length=2,
...     max_length=5)
```

```
>>> request = TestRequest()
>>> multiWidget = widget.FieldWidget(multiField, widget.MultiWidget(request))
```

Lets ensure that the minimum number of widgets are created.

```
>>> multiWidget.update()
>>> len(multiWidget.widgets)
2
```

Now, let's check if the function will do the right thing depending on the value:

No value:

```
>>> multiWidget.updateAllowAddRemove()
>>> multiWidget.allowAdding, multiWidget.allowRemoving
(True, False)
```

Minimum length:

```
>>> multiWidget.value = ['3', '5']
>>> multiWidget.updateAllowAddRemove()
>>> multiWidget.allowAdding, multiWidget.allowRemoving
(True, False)
```

Some allowed length:

```
>>> multiWidget.value = ['3', '5', '8', '6']
>>> multiWidget.updateAllowAddRemove()
>>> multiWidget.allowAdding, multiWidget.allowRemoving
(True, True)
```

Maximum length:

```
>>> multiWidget.value = ['3', '5', '8', '6', '42']
>>> multiWidget.updateAllowAddRemove()
>>> multiWidget.allowAdding, multiWidget.allowRemoving
(False, True)
```

Over maximum length:

```
>>> multiWidget.value = ['3', '5', '8', '6', '42', '45']
>>> multiWidget.updateAllowAddRemove()
>>> multiWidget.allowAdding, multiWidget.allowRemoving
(False, True)
```

I know a guy who once switched widget mode in the middle. All simple widgets are easy to hack, but multiWidget needs to update all subwidgets:

```
>>> [w.mode for w in multiWidget.widgets]
['input', 'input', 'input', 'input', 'input', 'input']
```

Switch the multiWidget mode:

```
>>> multiWidget.mode = interfaces.DISPLAY_MODE
```

Yes, all subwidgets switch mode:

```
>>> [w.mode for w in multiWidget.widgets]
['display', 'display', 'display', 'display', 'display', 'display']
```

The json data representing the multi widget:

```
>>> from pprint import pprint
>>> pprint(multiWidget.json_data())
{'error': '',
 'id': '',
 'label': '',
 'mode': 'display',
 'name': '',
 'required': True,
 'type': 'multi',
 'value': ['3', '5', '8', '6', '42', '45'],
 'widgets': [{'error': '',
              'id': '-0',
              'label': '',
              'mode': 'display',
              'name': '.0',
              'required': True,
              'type': 'text',
              'value': '3'},
             {'error': '',
              'id': '-1',
              'label': '',
              'mode': 'display',
              'name': '.1',
              'required': True,
              'type': 'text',
              'value': '5'},
             {'error': '',
              'id': '-2',
              'label': '',
              'mode': 'display',
              'name': '.2',
              'required': True,
              'type': 'text',
              'value': '8'},
             {'error': '',
              'id': '-3',
              'label': '',
              'mode': 'display',
              'name': '.3',
              'required': True,
              'type': 'text',
              'value': '6'},
             {'error': '',
```

(continues on next page)

(continued from previous page)

```
              'id': '-4',
              'label': '',
              'mode': 'display',
              'name': '.4',
              'required': True,
              'type': 'text',
              'value': '42'},
             {'error': '',
              'id': '-5',
              'label': '',
              'mode': 'display',
              'name': '.5',
              'required': True,
              'type': 'text',
              'value': '45'}]}
```

2.2.7 Multi Dict Widget

We can also use a multiWidget in Dict mode by just using a field which a Dict:

```
>>> multiField = zope.schema.Dict(
...     key_type=zope.schema.Int(),
...     value_type=zope.schema.Int(default=42))
>>> multiWidget.field = multiField
>>> multiWidget.name = 'multi.name'
```

Now if we set the value to a list we get an error:

```
>>> multiWidget.value = ['3', '5', '8', '6', '42', '45']
Traceback (most recent call last):
...
ValueError: need more than 1 value to unpack
```

but a dictionary is good.

```
>>> multiWidget.value = [('1', '3'), ('2', '5'), ('3', '8'), ('4', '6'), ('5', '42'), ('6
↪', '45')]
```

and our requests now have to include keys as well as values

```
>>> multiWidget.request = TestRequest(form={'multi.name.count':'2',
...                                         'multi.name.key.0':'1',
...                                         'multi.name.0':'42',
...                                         'multi.name.key.1':'2',
...                                         'multi.name.1':'43'})
>>> multiWidget.extract()
[('1', '42'), ('2', '43')]
```

Let's define a field with min and max length constraints and create a widget for it.

```
>>> multiField = zope.schema.Dict(
...     key_type=zope.schema.Int(),
```

(continues on next page)

(continued from previous page)

```
...     value_type=zope.schema.Int(default=42),
...     min_length=2,
...     max_length=5)
```

```
>>> request = TestRequest()
>>> multiWidget = widget.FieldWidget(multiField, widget.MultiWidget(request))
```

Lets ensure that the minimum number of widgets are created.

```
>>> multiWidget.update()
>>> len(multiWidget.widgets)
2
```

We can add new items

```
>>> multiWidget.appendAddingWidget()
>>> multiWidget.appendAddingWidget()
```

```
>>> multiWidget.update()
>>> len(multiWidget.widgets)
4
```

The json data representing the Multi Dict Widget is the same as the Multi widget:

2.2.8 Widget Events

Widget-system interaction can be very rich and wants to be extended in unexpected ways. Thus there exists a generic widget event that can be used by other code.

```
>>> event = widget.WidgetEvent(ageWidget)
>>> event
<WidgetEvent <Widget 'age'>>
```

These events provide the `IWidgetEvent` interface:

```
>>> interfaces.IWidgetEvent.providedBy(event)
True
```

There exists a special event that can be send out after a widget has been updated, ...

```
>>> afterUpdate = widget.AfterWidgetUpdateEvent(ageWidget)
>>> afterUpdate
<AfterWidgetUpdateEvent <Widget 'age'>>
```

which provides another special interface:

```
>>> interfaces.IAfterWidgetUpdateEvent.providedBy(afterUpdate)
True
```

This event should be used by widget-managing components and is not created and sent out internally by the widget's `update()` method. The event was designed to provide an additional hook between updating the widget and rendering it.

2.2.9 Cleanup

Let's not leave temporary files lying around

```
>>> import os
>>> os.remove(textWidgetTemplate)
```

2.3 Content Providers

We want to mix fields and content providers.

This allow to enrich the form by interlacing html snippets produced by content providers.

For instance, we might want to render the table of results in a search form.

We might also need to render HTML close to a widget as a handle used when improving UI with Ajax.

Adding HTML outside the widgets avoids the systematic need of subclassing or changing the full widget rendering.

2.3.1 Test setup

Before we can use a widget manager, the `IFieldWidget` adapter has to be registered for the `ITextLine` field:

```
>>> import zope.component
>>> import zope.interface
>>> from z3c.form import interfaces, widget
>>> from z3c.form.browser import text
>>> from z3c.form.testing import TestRequest

>>> @zope.component.adapter(zope.schema.TextLine, TestRequest)
... @zope.interface.implementer(interfaces.IFieldWidget)
... def TextFieldWidget(field, request):
...     return widget.FieldWidget(field, text.TextWidget(request))

>>> zope.component.provideAdapter(TextFieldWidget)

>>> from z3c.form import converter
>>> zope.component.provideAdapter(converter.FieldDataConverter)
>>> zope.component.provideAdapter(converter.FieldWidgetDataConverter)
```

We define a simple test schema with fields:

```
>>> import zope.interface
>>> import zope.schema

>>> class IPerson(zope.interface.Interface):
...
...     id = zope.schema.TextLine(
...         title=u'ID',
...         description=u"The person's ID.",
...         required=True)
...
...     fullname = zope.schema.TextLine(
```

(continues on next page)

(continued from previous page)

```
...         title=u'FullName',
...         description=u"The person's name.",
...         required=True)
...
```

A class that implements the schema:

```
>>> class Person(object):
...     id = 'james'
...     fullname = 'James Bond'
```

The usual request instance:

```
>>> request = TestRequest()
```

We want to insert a content provider in between fields. We define a test content provider that renders extra help text:

```
>>> from zope.publisher.browser import BrowserView
>>> from zope.contentprovider.interfaces import IContentProvider
>>> class ExtendedHelp(BrowserView):
...   def __init__(self, context, request, view):
...       super(ExtendedHelp, self).__init__(context, request)
...       self.__parent__ = view
...
...   def update(self):
...       self.person = self.context.id
...
...   def render(self):
...       return '<div class="ex-help">Help about person %s</div>' % self.person
```

2.3.2 Form definition

The meat of the tests begins here.

We define a form as usual by inheriting from `form.Form`:

```
>>> from z3c.form import field, form
>>> from zope.interface import implementer
```

To enable content providers, the form class must :

1. implement `IFieldsAndContentProvidersForm`
2. have a `contentProviders` attribute that is an instance of the `ContentProviders` class.

```
>>> from z3c.form.interfaces import IFieldsAndContentProvidersForm
>>> from z3c.form.contentprovider import ContentProviders
```

Content provider assignment

Content providers classes (factories) can be assigned directly to the `ContentProviders` container:

```
>>> @implementer(IFieldsAndContentProvidersForm)
... class PersonForm(form.Form):
...     fields = field.Fields(IPerson)
...     ignoreContext = True
...     contentProviders = ContentProviders()
...     contentProviders['longHelp'] = ExtendedHelp
...     contentProviders['longHelp'].position = 1
```

Let's instantiate content and form instances:

```
>>> person = Person()
>>> personForm = PersonForm(person, request)
```

Once the widget manager has been updated, it holds the content provider:

```
>>> from z3c.form.contentprovider import FieldWidgetsAndProviders
>>> manager = FieldWidgetsAndProviders(personForm, request, person)
>>> manager.ignoreContext = True
>>> manager.update()
>>> widgets = manager
>>> ids = sorted(widgets.keys())
>>> ids
['fullname', 'id', 'longHelp']
>>> widgets['longHelp']
<ExtendedHelp object at ...>
>>> widgets['id']
<TextWidget 'form.widgets.id'>
>>> widgets['fullname']
<TextWidget 'form.widgets.fullname'>
>>> manager.get('longHelp').render()
'<div class="ex-help">Help about person james</div>'
```

Content provider lookup

Forms can also refer by name to content providers.

Let's register a content provider by name as usual:

```
>>> from zope.component import provideAdapter
>>> from zope.contentprovider.interfaces import IContentProvider
>>> from z3c.form.interfaces import IFormLayer
>>> provideAdapter(ExtendedHelp,
...                (zope.interface.Interface,
...                 IFormLayer,
...                 zope.interface.Interface),
...                provides=IContentProvider, name='longHelp')
```

Let the form refer to it:

```
>>> @implementer(IFieldsAndContentProvidersForm)
... class LookupPersonForm(form.Form):
...     prefix = 'form.'
...     fields = field.Fields(IPerson)
...     ignoreContext = True
...     contentProviders = ContentProviders(['longHelp'])
...     contentProviders['longHelp'].position = 2

>>> lookupForm = LookupPersonForm(person, request)
```

After update, the widget manager refers to the content provider:

```
>>> from z3c.form.contentprovider import FieldWidgetsAndProviders
>>> manager = FieldWidgetsAndProviders(lookupForm, request, person)
>>> manager.ignoreContext = True
>>> manager.update()
>>> widgets = manager
>>> ids = sorted(widgets.keys())
>>> ids
['fullname', 'id', 'longHelp']
>>> widgets['longHelp']
<ExtendedHelp object at ...>
>>> widgets['id']
<TextWidget 'form.widgets.id'>
>>> widgets['fullname']
<TextWidget 'form.widgets.fullname'>
>>> manager.get('longHelp').render()
'<div class="ex-help">Help about person james</div>'
```

Providers position

Until here, we have defined position for content providers without explaining how it is used.

A position needs to be defined for each provider. Let's forget to define a position:

```
>>> @implementer(IFieldsAndContentProvidersForm)
... class UndefinedPositionForm(form.Form):
...     prefix = 'form.'
...     fields = field.Fields(IPerson)
...     ignoreContext = True
...     contentProviders = ContentProviders(['longHelp'])

>>> form = UndefinedPositionForm(person, request)
>>> manager = FieldWidgetsAndProviders(form, request, person)
>>> manager.ignoreContext = True
```

When updating the widget manager, we get an exception:

```
>>> manager.update()
Traceback (most recent call last):
...
ValueError: Position of the following content provider should be an integer: 'longHelp'.
```

Let's check positioning of content providers:

```
>>> LookupPersonForm.contentProviders['longHelp'].position = 0
>>> manager = FieldWidgetsAndProviders(lookupForm, request, person)
>>> manager.ignoreContext = True
>>> manager.update()
>>> list(manager.values())
[<ExtendedHelp object at ...>, <TextWidget 'form.widgets.id'>, <TextWidget 'form.widgets.
↪fullname'>]

>>> LookupPersonForm.contentProviders['longHelp'].position = 1
>>> manager = FieldWidgetsAndProviders(lookupForm, request, person)
>>> manager.ignoreContext = True
>>> manager.update()
>>> list(manager.values())
[<TextWidget 'form.widgets.id'>, <ExtendedHelp object at ...>, <TextWidget 'form.widgets.
↪fullname'>]

>>> LookupPersonForm.contentProviders['longHelp'].position = 2
>>> manager = FieldWidgetsAndProviders(lookupForm, request, person)
>>> manager.ignoreContext = True
>>> manager.update()
>>> list(manager.values())
[<TextWidget 'form.widgets.id'>, <TextWidget 'form.widgets.fullname'>, <ExtendedHelp␣
↪object at ...>]
```

Using value larger than sequence length implies end of sequence:

```
>>> LookupPersonForm.contentProviders['longHelp'].position = 3
>>> manager = FieldWidgetsAndProviders(lookupForm, request, person)
>>> manager.ignoreContext = True
>>> manager.update()
>>> list(manager.values())
[<TextWidget 'form.widgets.id'>, <TextWidget 'form.widgets.fullname'>, <ExtendedHelp␣
↪object at ...>]
```

A negative value is interpreted same as `insert` method of Python lists:

```
>>> LookupPersonForm.contentProviders['longHelp'].position = -1
>>> manager = FieldWidgetsAndProviders(lookupForm, request, person)
>>> manager.ignoreContext = True
>>> manager.update()
>>> list(manager.values())
[<TextWidget 'form.widgets.id'>, <ExtendedHelp object at ...>, <TextWidget 'form.widgets.
↪fullname'>]
```

2.3.3 Rendering the form

Once the form has been updated, it can be rendered.

Since we have not assigned a template yet, we have to do it now. We have a small template as part of this example:

```
>>> import os
>>> from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile
>>> from zope.browserpage.viewpagetemplatefile import BoundPageTemplate
>>> from z3c.form import tests
>>> def personTemplate(form):
...     form.template = BoundPageTemplate(
...         ViewPageTemplateFile(
...             'simple_edit_with_providers.pt',
...             os.path.dirname(tests.__file__)), form)
>>> personTemplate(personForm)
```

To enable form updating, all widget adapters must be registered:

```
>>> from z3c.form.testing import setupFormDefaults
>>> setupFormDefaults()
```

`FieldWidgetsAndProviders` is registered as widget manager for `IFieldsAndContentProvidersForm`:

```
>>> personForm.update()
>>> personForm.widgets
FieldWidgetsAndProviders([...])
```

Let's render the form:

```
>>> print(personForm.render())
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-id">ID</label>
        <input id="form-widgets-id" name="form.widgets.id"
               class="text-widget required textline-field"
               value="" type="text" />
      </div>
      <div class="row">
        <div class="ex-help">Help about person james</div>
      </div>
      <div class="row">
        <label for="form-widgets-fullname">FullName</label>
        <input id="form-widgets-fullname"
               name="form.widgets.fullname"
               class="text-widget required textline-field"
               value="" type="text" />
      </div>
    </form>
  </body>
</html>
```

2.4 Action Managers

Action managers are components that manage all actions that can be taken within a view, usually a form. They are also responsible for executing actions when asked to do so.

2.4.1 Creating an action manager

An action manager is a form-related adapter that has the following discriminator: form, request, and content. While there is a base implementation for an action manager, the `action` module does not provide a full implementation.

So we first have to build a simple implementation based on the `Actions` manager base class which allows us to add actions. Note that the following implementation is for demonstration purposes. If you want to see a real action manager implementation, then have a look at `ButtonActions`. Let's now implement our simple action manager:

```
>>> from z3c.form import action
>>> class SimpleActions(action.Actions):
...     """Simple sample."""
...
...     def append(self, name, action):
...         """See z3c.form.interfaces.IActions."""
...         self[name] = action
```

Before we can initialise the action manager, we have to create instances for our three discriminators, just enough to get it working:

```
>>> import zope.interface
>>> from z3c.form import interfaces
>>> @zope.interface.implementer(interfaces.IForm)
... class Form(object):
...     pass
>>> form = Form()
```

```
>>> @zope.interface.implementer(zope.interface.Interface)
... class Content(object):
...     pass
>>> content = Content()
```

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

We are now ready to create the action manager, which is a simple triple-adapter:

```
>>> manager = SimpleActions(form, request, content)
>>> manager
<SimpleActions None>
```

As we can see in the manager representation above, the name of the manager is `None`, since we have not specified one:

```
>>> manager.__name__ = 'example'
>>> manager
<SimpleActions 'example'>
```

2.4.2 Managing and Accessing Actions

Initially there are no actions in the manager:

```
>>> list(manager.keys())
[]
```

Our simple implementation of has an additional `append()` method, which we will use to add actions:

```
>>> apply = action.Action(request, 'Apply')
>>> manager.append(apply.name, apply)
```

The action is added immediately:

```
>>> list(manager.keys())
['apply']
```

However, you should not rely on it being added, and always update the manager once all actions were defined:

```
>>> manager.update()
```

Note: If the title of the action is a more complex unicode string and no name is specified for the action, then a hexadecimal name is created from the title:

```
>>> action.Action(request, 'Apply Now!').name
'4170706c79204e6f7721'
```

Since the action manager is an enumerable mapping, ...

```
>>> from zope.interface.common.mapping import IEnumerableMapping
>>> IEnumerableMapping.providedBy(manager)
True
```

there are several API methods available:

```
>>> manager['apply']
<Action 'apply' 'Apply'>
>>> manager['foo']
Traceback (most recent call last):
...
KeyError: 'foo'
```

```
>>> manager.get('apply')
<Action 'apply' 'Apply'>
>>> manager.get('foo', 'default')
'default'
```

```
>>> 'apply' in manager
True
>>> 'foo' in manager
False
```

```
>>> list(manager.values())
[<Action 'apply' 'Apply'>]
```

```
>>> list(manager.items())
[('apply', <Action 'apply' 'Apply'>)]
```

```
>>> len(manager)
1
```

2.4.3 Executing actions

When an action is executed, an execution adapter is looked up. If there is no adapter, nothing happens. So let's create a request that submits the apply button:

```
>>> request = TestRequest(form={'apply': 'Apply'})
>>> manager = SimpleActions(form, request, content)
```

We also want to have two buttons in this case, so that we can ensure that only one is executed:

```
>>> apply = action.Action(request, 'Apply')
>>> manager.append(apply.name, apply)
```

```
>>> cancel = action.Action(request, 'Cancel')
>>> manager.append(cancel.name, cancel)
>>> manager.update()
```

Now that the manager is updated, we can ask it for the "executed" actions:

```
>>> manager.executedActions
[<Action 'apply' 'Apply'>]
```

Executing the actions does nothing, because there are no handlers yet:

```
>>> manager.execute()
```

Let's now register an action handler that listens to the "Apply" action. An action handler has four discriminators: form, request, content, and action. All those objects are available to the handler under those names. When using the base action handler from the `action` module, `__call__()` is the only method that needs to be implemented:

```
>>> from z3c.form import util
```

```
>>> class SimpleActionHandler(action.ActionHandlerBase):
...     zope.component.adapts(
...         None, TestRequest, None, util.getSpecification(apply))
...     def __call__(self):
...         print('successfully applied')
```

```
>>> zope.component.provideAdapter(SimpleActionHandler)
```

As you can see, we registered the action specifically for the apply action. Now, executing the actions calls this handler:

```
>>> manager.execute()
successfully applied
```

Of course it only works for the "Apply" action and not ""Cancel":

```
>>> request = TestRequest(form={'cancel': 'Cancel'})
>>> manager.request = apply.request = cancel.request = request
>>> manager.execute()
```

Further, when a handler is successfully executed, an event is sent out, so let's register an event handler:

```
>>> eventlog = []
>>> @zope.component.adapter(interfaces.IActionEvent)
... def handleEvent(event):
...     eventlog.append(event)
```

```
>>> zope.component.provideHandler(handleEvent)
```

Let's now execute the "Apply" action again:

```
>>> request = TestRequest(form={'apply': 'Apply'})
>>> manager.request = apply.request = cancel.request = request
>>> manager.execute()
successfully applied
```

```
>>> eventlog[-1]
<ActionSuccessful for <Action 'apply' 'Apply'>>
```

Action handlers, however, can also raise action errors. These action errors are caught and an event is created notifying the system of the problem. The error is not further propagated. Other errors are not handled by the system to avoid hiding real failures of the code.

Let's see how action errors can be used by implementing a handler for the cancel action:

```
>>> class ErrorActionHandler(action.ActionHandlerBase):
...     zope.component.adapts(
...         None, TestRequest, None, util.getSpecification(cancel))
...     def __call__(self):
...         raise interfaces.ActionExecutionError(
...             zope.interface.Invalid('Something went wrong'))
```

```
>>> zope.component.provideAdapter(ErrorActionHandler)
```

As you can see, the action execution error wraps some other execption, in this case a simple invalid error.

Executing the "Cancel" action now produces the action error event:

```
>>> request = TestRequest(form={'cancel': 'Cancel'})
>>> manager.request = apply.request = cancel.request = request
>>> manager.execute()
```

```
>>> eventlog[-1]
<ActionErrorOccurred for <Action 'cancel' 'Cancel'>>
```

```
>>> eventlog[-1].error
<ActionExecutionError wrapping ...Invalid...>
```

2.5 Browser support

The `z3c.form` library provides a form framework and widgets. This document ensures that we implement a widget for each field defined in `zope.schema`. Take a look at the different widget doctest files for more information about the widgets.

```
>>> import zope.schema
>>> from z3c.form import browser
```

Let's setup all required adapters using zcml. This makes sure we test the real configuration.

```
>>> from zope.configuration import xmlconfig
>>> import zope.component
>>> import zope.security
>>> import zope.i18n
>>> import zope.browserresource
>>> import z3c.form
>>> xmlconfig.XMLConfig('meta.zcml', zope.component)()
>>> xmlconfig.XMLConfig('meta.zcml', zope.security)()
>>> xmlconfig.XMLConfig('meta.zcml', zope.i18n)()
>>> xmlconfig.XMLConfig('meta.zcml', zope.browserresource)()
>>> xmlconfig.XMLConfig('meta.zcml', z3c.form)()
>>> xmlconfig.XMLConfig('configure.zcml', z3c.form)()
```

This utility is setup by hand, since its ZCML loads to many unwanted files:

```
>>> import zope.component
>>> import zope.i18n.negotiator
>>> zope.component.provideUtility(zope.i18n.negotiator.negotiator)
```

also define a helper method for test the widgets:

```
>>> from z3c.form import interfaces
>>> from z3c.form.testing import TestRequest
>>> def setupWidget(field):
...     request = TestRequest()
...     widget = zope.component.getMultiAdapter((field, request),
...         interfaces.IFieldWidget)
...     widget.id = 'foo'
...     widget.name = 'bar'
...     return widget
```

2.5.1 ASCII

```
>>> field = zope.schema.ASCII(default='This is\n ASCII.')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.textarea.TextAreaWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from ASCII to TextAreaWidget>
```

```
>>> print(widget.render())
<textarea id="foo" name="bar" class="textarea-widget required ascii-field">This is
 ASCII.</textarea>
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="textarea-widget required ascii-field">This is
 ASCII.</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="textarea-widget required ascii-field">This is
 ASCII.</span>
</div>
</div>
```

As you can see, we will get an additional error class called `row-error` if we render a widget with an error view assinged:

```
>>> class DummyErrorView(object):
...     def render(self):
...         return 'Dummy Error'
>>> widget.error = (DummyErrorView())
>>> print(widget())
<div id="foo-row" class="row-error row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="textarea-widget required ascii-field">This is
 ASCII.</span>
</div>
  <div class="error">
    Dummy Error
  </div>
</div>
```

2.5.2 ASCIILine

```
>>> field = zope.schema.ASCIILine(default='An ASCII line.')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from ASCIILine to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar"
       class="text-widget required asciiline-field" value="An ASCII line." />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required asciiline-field">An ASCII line.</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required asciiline-field">An ASCII line.</span>
</div>
</div>
```

2.5.3 Bool

```
>>> field = zope.schema.Bool(default=True, title=u"Check me", required=True)
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.radio.RadioWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<SequenceDataConverter converts from Bool to RadioWidget>
```

```
>>> print(widget.render())
<span class="option">
```

(continues on next page)

(continued from previous page)

```
  <label for="foo-0">
    <input type="radio" id="foo-0" name="bar"
           class="radio-widget required bool-field" value="true"
           checked="checked" />
    <span class="label">yes</span>
  </label>
</span><span class="option">
  <label for="foo-1">
    <input type="radio" id="foo-1" name="bar"
           class="radio-widget required bool-field" value="false" />
    <span class="label">no</span>
  </label>
</span>
<input name="bar-empty-marker" type="hidden" value="1" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="radio-widget required bool-field"><span
    class="selected-option">yes</span></span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span>Check me</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
<span id="foo" class="radio-widget required bool-field"><span class="selected-option">yes
↪</span></span>
</div>
</div>
```

For the boolean, the checkbox widget can be used as well:

```
>>> from z3c.form.browser import checkbox
>>> widget = checkbox.CheckBoxFieldWidget(field, TestRequest())
>>> widget.id = 'foo'
>>> widget.name = 'bar'
>>> widget.update()
```

```
>>> print(widget.render())
<span id="foo">
 <span class="option">
  <input type="checkbox" id="foo-0" name="bar:list"
         class="checkbox-widget required bool-field" value="true"
         checked="checked" />
  <label for="foo-0">
    <span class="label">yes</span>
```

(continues on next page)

(continued from previous page)

```
  </label>
</span><span class="option">
  <input type="checkbox" id="foo-1" name="bar:list"
         class="checkbox-widget required bool-field" value="false" />
  <label for="foo-1">
    <span class="label">no</span>
  </label>
 </span>
</span>
<input name="bar-empty-marker" type="hidden" value="1" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="checkbox-widget required bool-field"><span
    class="selected-option">yes</span></span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span>Check me</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
<span id="foo" class="checkbox-widget required bool-field"><span class="selected-option">
↪yes</span></span>
</div>
</div>
```

We can also have a single checkbox button for the boolean.

```
>>> widget = checkbox.SingleCheckBoxFieldWidget(field, TestRequest())
>>> widget.id = 'foo'
>>> widget.name = 'bar'
>>> widget.update()
```

```
>>> print(widget.render())
<span class="option" id="foo">
  <input type="checkbox" id="foo-0" name="bar:list"
         class="single-checkbox-widget required bool-field"
         value="selected" checked="checked" />
  <label for="foo-0">
    <span class="label">Check me</span>
  </label>
</span>
<input name="bar-empty-marker" type="hidden" value="1" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
```

(continues on next page)

(continued from previous page)

```
<span id="foo"
      class="single-checkbox-widget required bool-field"><span
    class="selected-option">Check me</span></span>
```

Note that the widget label is not repeated twice:

```
>>> widget.label
''
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
<span id="foo" class="single-checkbox-widget required bool-field"><span class="selected-
↪option">Check me</span></span>
</div>
</div>
```

2.5.4 Button

```
>>> from z3c.form import button
>>> field = button.Button(title='Press me!')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.submit.SubmitWidget'>
```

```
>>> print(widget.render())
<input type="submit" id="foo" name="bar"
       class="submit-widget button-field" value="Press me!" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<input type="submit" id="foo" name="bar"
       class="submit-widget button-field" value="Press me!"
       disabled="disabled" />
```

There exists an alternative widget for the button field, the button widget. It is not used by default, but available for use:

```
>>> from z3c.form.browser.button import ButtonFieldWidget
>>> widget = ButtonFieldWidget(field, TestRequest())
>>> widget.id = "foo"
>>> widget.name = "bar"
```

```
>>> widget.update()
>>> print(widget.render())
<input type="button" id="foo" name="bar"
       class="button-widget button-field" value="Press me!" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<input type="button" id="foo" name="bar"
       class="button-widget button-field" value="Press me!"
       disabled="disabled" />
```

2.5.5 Bytes

```
>>> field = zope.schema.Bytes(default=b'Default bytes')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.file.FileWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FileUploadDataConverter converts from Bytes to FileWidget>
```

```
>>> print(widget.render())
<input type="file" id="foo" name="bar" class="file-widget required bytes-field" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> widget.render().strip('\r\n')
'<span id="foo" class="file-widget required bytes-field"></span>'
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
<span id="foo" class="file-widget required bytes-field"></span>
</div>
</div>
```

2.5.6 BytesLine

```
>>> field = zope.schema.BytesLine(default=b'A Bytes line.')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from BytesLine to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required bytesline-field"
       value="A Bytes line." />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required bytesline-field">A Bytes line.</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required bytesline-field">A Bytes line.</span>
</div>
</div>
```

2.5.7 Choice

```
>>> from zope.schema import vocabulary
>>> terms = [vocabulary.SimpleTerm(*value) for value in
...          [(True, 'yes', 'Yes'), (False, 'no', 'No')]]
>>> vocabulary = vocabulary.SimpleVocabulary(terms)
>>> field = zope.schema.Choice(default=True, vocabulary=vocabulary)
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.select.SelectWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<SequenceDataConverter converts from Choice to SelectWidget>
```

```
>>> print(widget.render())
<select id="foo" name="bar:list" class="select-widget required choice-field"
        size="1">
  <option id="foo-0" value="yes" selected="selected">Yes</option>
  <option id="foo-1" value="no">No</option>
</select>
<input name="bar-empty-marker" type="hidden" value="1" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="select-widget required choice-field"><span
  class="selected-option">Yes</span></span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
<span id="foo" class="select-widget required choice-field"><span class="selected-option">
↪Yes</span></span>
</div>
</div>
```

2.5.8 Date

```
>>> import datetime
>>> field = zope.schema.Date(default=datetime.date(2007, 4, 1))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<DateDataConverter converts from Date to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required date-field"
       value="07/04/01" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required date-field">07/04/01</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required date-field">07/04/01</span>
</div>
</div>
```

2.5.9 Datetime

```
>>> field = zope.schema.Datetime(default=datetime.datetime(2007, 4, 1, 12))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<DatetimeDataConverter converts from Datetime to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required datetime-field"
       value="07/04/01 12:00" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required datetime-field">07/04/01 12:00</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required datetime-field">07/04/01 12:00</span>
</div>
</div>
```

2.5.10 Decimal

```
>>> import decimal
>>> field = zope.schema.Decimal(default=decimal.Decimal('1265.87'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<DecimalDataConverter converts from Decimal to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required decimal-field"
       value="1,265.87" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required decimal-field">1,265.87</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required decimal-field">1,265.87</span>
</div>
</div>
```

2.5.11 Dict

There is no default widget for this field, since the sematics are fairly complex.

2.5.12 DottedName

```
>>> field = zope.schema.DottedName(default='z3c.form')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from DottedName to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required dottedname-field"
       value="z3c.form" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required dottedname-field">z3c.form</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required dottedname-field">z3c.form</span>
</div>
</div>
```

2.5.13 Float

```
>>> field = zope.schema.Float(default=1265.8)
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FloatDataConverter converts from Float to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required float-field"
       value="1,265.8" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required float-field">1,265.8</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
```

(continues on next page)

(continued from previous page)

```
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required float-field">1,265.8</span>
</div>
</div>
```

2.5.14 FrozenSet

```
>>> field = zope.schema.FrozenSet(
...     value_type=zope.schema.Choice(values=(1, 2, 3, 4)),
...     default=frozenset([1, 3]) )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.select.SelectWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<CollectionSequenceDataConverter converts from FrozenSet to SelectWidget>
```

```
>>> print(widget.render())
<select id="foo" name="bar:list" class="select-widget required frozenset-field"
        multiple="multiple" size="5">
  <option id="foo-0" value="1" selected="selected">1</option>
  <option id="foo-1" value="2">2</option>
  <option id="foo-2" value="3" selected="selected">3</option>
  <option id="foo-3" value="4">4</option>
</select>
<input name="bar-empty-marker" type="hidden" value="1" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="select-widget required frozenset-field"><span
  class="selected-option">1</span>, <span
  class="selected-option">3</span></span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
```

(continues on next page)

(continued from previous page)

```
    </label>
  </div>
  <div class="widget">
<span id="foo" class="select-widget required frozenset-field"><span class="selected-
↪option">1</span>, <span class="selected-option">3</span></span>
</div>
</div>
```

2.5.15 Id

```
>>> field = zope.schema.Id(default='z3c.form')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from Id to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required id-field"
       value="z3c.form" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required id-field">z3c.form</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required id-field">z3c.form</span>
</div>
</div>
```

2.5.16 ImageButton

Let's say we have a simple image field that uses the `pressme.png` image.

```
>>> from z3c.form import button
>>> field = button.ImageButton(
...     image='pressme.png',
...     title='Press me!')
```

When the widget is created, the system converts the relative image path to an absolute image path by looking up the resource. For this to work, we have to setup some of the traversing machinery:

> # Traversing setup >>> from zope.traversing import testing >>> testing.setUp()
>
> # Resource namespace >>> import zope.component >>> from zope.traversing.interfaces import ITraversable >>> from zope.traversing.namespace import resource >>> zope.component.provideAdapter(... resource, (None,), ITraversable, name="resource") >>> zope.component.provideAdapter(... resource, (None, None), ITraversable, name="resource")
>
> # New absolute URL adapter for resources, if available >>> from zope.browserresource.resource import AbsoluteURL >>> zope.component.provideAdapter(AbsoluteURL)
>
> # Register the "pressme.png" resource >>> from zope.browserresource.resource import Resource >>> testing.browserResource('pressme.png', Resource)

Now we are ready to instantiate the widget:

```
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.image.ImageWidget'>
```

```
>>> print(widget.render())
<input type="image" id="foo" name="bar"
       class="image-widget imagebutton-field"
       src="http://127.0.0.1/@@/pressme.png"
       value="Press me!" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<input type="image" id="foo" name="bar"
       class="image-widget imagebutton-field"
       src="http://127.0.0.1/@@/pressme.png"
       value="Press me!" disabled="disabled" />
```

2.5.17 Int

```
>>> field = zope.schema.Int(default=1200)
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<IntegerDataConverter converts from Int to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required int-field"
       value="1,200" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required int-field">1,200</span>
```

2.5.18 List - ASCII

```
>>> field = zope.schema.List(
...     value_type=zope.schema.ASCII(
...         title='ASCII',
...         default='This is\n ASCII.'),
...     default=['foo\nfoo', 'bar\nbar'])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>ASCII</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
```

(continues on next page)

(continued from previous page)

```
                    id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-0" name="bar.0"
               class="textarea-widget required ascii-field">foo
foo</textarea>
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>ASCII</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-1" name="bar.1"
               class="textarea-widget required ascii-field">bar
bar</textarea>
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.19 List - ASCIILine

```
>>> field = zope.schema.List(
...     value_type=zope.schema.ASCIILine(
...         title='ASCIILine',
...         default='An ASCII line.'),
...     default=['foo', 'bar'])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>ASCIILine</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required asciiline-field"
               value="foo" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>ASCIILine</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required asciiline-field"
               value="bar" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.20 List - Choice

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Choice(values=(1, 2, 3, 4)),
...     default=[1, 3] )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.orderedselect.OrderedSelectWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<CollectionSequenceDataConverter converts from List to OrderedSelectWidget>
```

```
>>> print(widget.render())
<script src="++resource++orderedselect_input.js" type="text/javascript"></script>
<table border="0" class="ordered-selection-field" id="foo">
  <tr>
    <td>
      <select id="foo-from" name="bar.from" size="5"
              multiple="multiple"
              class="required list-field">
        <option value="2">2</option>
        <option value="4">4</option>
      </select>
    </td>
    <td>
      <button name="from2toButton" type="button"
              value="&rarr;"
              onclick="javascript:from2to('foo')">&rarr;</button>
      <br />
      <button name="to2fromButton" type="button"
              value="&larr;"
              onclick="javascript:to2from('foo')">&larr;</button>
    </td>
    <td>
      <select id="foo-to" name="bar.to" size="5"
              multiple="multiple" class="required list-field">
        <option value="1">1</option>
        <option value="3">3</option>
      </select>
      <input name="bar-empty-marker" type="hidden" />
      <span id="foo-toDataContainer" style="display: none">
        <script type="text/javascript">
          copyDataForSubmit('foo');</script>
      </span>
    </td>
    <td>
      <button name="upButton" type="button" value="&uarr;"
              onclick="javascript:moveUp('foo')">&uarr;</button>
      <br />
      <button name="downButton" type="button" value="&darr;"
```

(continues on next page)

(continued from previous page)

```
              onclick="javascript:moveDown('foo')">&darr;</button>
    </td>
  </tr>
</table>
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="required list-field"><span
    class="selected-option">1</span>, <span
    class="selected-option">3</span></span>
```

2.5.21 List - Date

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Date(
...         title='Date',
...         default=datetime.date(2007, 4, 1)),
...     default=[datetime.date(2008, 9, 27), datetime.date(2008, 9, 28)])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Date</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required date-field"
               value="08/09/27" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
```

(continues on next page)

(continued from previous page)

```
        <div class="label">
          <label for="foo-1">
            <span>Date</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required date-field"
               value="08/09/28" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.22 List - Datetime

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Datetime(
...         title='Datetime',
...         default=datetime.datetime(2007, 4, 1, 12)),
...     default=[datetime.datetime(2008, 9, 27, 12),
...              datetime.datetime(2008, 9, 28, 12)])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
```

(continues on next page)

(continued from previous page)

```
        <div class="label">
          <label for="foo-0">
            <span>Datetime</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required datetime-field"
               value="08/09/27 12:00" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Datetime</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required datetime-field"
               value="08/09/28 12:00" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.23 List - Decimal

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Decimal(
...         title='Decimal',
...         default=decimal.Decimal('1265.87')),
...     default=[decimal.Decimal('123.456'), decimal.Decimal('1')])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Decimal</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required decimal-field"
               value="123.456" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Decimal</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required decimal-field" value="1" />
```

(continues on next page)

(continued from previous page)

```
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.24 List - DottedName

```
>>> field = zope.schema.List(
...     value_type=zope.schema.DottedName(
...         title='DottedName',
...         default='z3c.form'),
...     default=['z3c.form', 'z3c.wizard'])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>DottedName</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required dottedname-field"
               value="z3c.form" />
        </div>
```

(continues on next page)

(continued from previous page)

```
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>DottedName</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required dottedname-field"
               value="z3c.wizard" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.25 List - Float

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Float(
...         title='Float',
...         default=123.456),
...     default=[1234.5, 1])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
```

(continues on next page)

(continued from previous page)

```
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Float</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required float-field"
               value="1,234.5" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Float</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required float-field" value="1.0" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.26 List - Id

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Id(
...         title='Id',
...         default='z3c.form'),
...     default=['z3c.form', 'z3c.wizard'])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Id</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required id-field"
               value="z3c.form" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Id</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required id-field"
```

(continues on next page)

(continued from previous page)

```
               value="z3c.wizard" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.27 List - Int

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Int(
...         title='Int',
...         default=666),
...     default=[42, 43])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Int</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required int-field" value="42" />
        </div>
```

(continues on next page)

(continued from previous page)

```
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Int</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required int-field" value="43" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.28 List - Password

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Password(
...         title='Password',
...         default='mypwd'),
...     default=['pwd', 'pass'])
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
```

(continues on next page)

(continued from previous page)

```
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Password</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="password" id="foo-0" name="bar.0"
              class="password-widget required password-field" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Password</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="password" id="foo-1" name="bar.1"
              class="password-widget required password-field" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.29 List - SourceText

```
>>> field = zope.schema.List(
...     value_type=zope.schema.SourceText(
...         title='SourceText',
...         default='<source />'),
...     default=['<html></body>foo</body></html>', '<h1>bar</h1>'] )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>SourceText</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-0" name="bar.0"
              class="textarea-widget required sourcetext-field">&lt;html&gt;&lt;/body&
↪gt;foo&lt;/body&gt;&lt;/html&gt;</textarea>
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>SourceText</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-1" name="bar.1"
              class="textarea-widget required sourcetext-field">&lt;h1&gt;bar&lt;/h1&gt;
```

(continues on next page)

(continued from previous page)

```
↪</textarea>
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.30 List - Text

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Text(
...         title='Text',
...         default='Some\n Text.'),
...     default=['foo\nfoo', 'bar\nbar'] )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Text</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-0" name="bar.0"
          class="textarea-widget required text-field">foo
foo</textarea>
```

(continues on next page)

(continued from previous page)

```
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Text</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-1" name="bar.1"
          class="textarea-widget required text-field">bar
bar</textarea>
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.31 List - TextLine

```
>>> field = zope.schema.List(
...     value_type=zope.schema.TextLine(
...         title='TextLine',
...         default='Some Text line.'),
...     default=['foo', 'bar'] )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>TextLine</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required textline-field"
               value="foo" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>TextLine</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required textline-field"
               value="bar" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.32 List - Time

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Time(
...         title='Time',
...         default=datetime.time(12, 0)),
...     default=[datetime.time(13, 0), datetime.time(14, 0)] )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Time</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required time-field" value="13:00" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Time</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required time-field" value="14:00" />
        </div>
```

(continues on next page)

(continued from previous page)

```
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.33 List - Timedelta

```
>>> field = zope.schema.List(
...     value_type=zope.schema.Timedelta(
...         title='Timedelta',
...         default=datetime.timedelta(days=3)),
...     default=[datetime.timedelta(days=4), datetime.timedelta(days=5)] )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Timedelta</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required timedelta-field"
               value="4 days, 0:00:00" />
        </div>
      </div>
```

(continues on next page)

(continued from previous page)

```
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Timedelta</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required timedelta-field"
               value="5 days, 0:00:00" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.34 List - URI

```
>>> field = zope.schema.List(
...     value_type=zope.schema.URI(
...         title='URI',
...         default='http://zope.org'),
...     default=['http://www.python.org', 'http://www.zope.com'] )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from List to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
```

(continues on next page)

(continued from previous page)

```
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>URI</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required uri-field"
               value="http://www.python.org" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>URI</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required uri-field"
               value="http://www.zope.com" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.35 Object

By default, we are not going to provide widgets for an object, since we believe this is better done using sub-forms.

2.5.36 Password

```
>>> field = zope.schema.Password(default='mypwd')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.password.PasswordWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from Password to PasswordWidget>
```

```
>>> print(widget.render())
<input type="password" id="foo" name="bar"
       class="password-widget required password-field" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="password-widget required password-field">mypwd</span>
```

2.5.37 Set

```
>>> field = zope.schema.Set(
...     value_type=zope.schema.Choice(values=(1, 2, 3, 4)),
...     default=set([1, 3]) )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.select.SelectWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<CollectionSequenceDataConverter converts from Set to SelectWidget>
```

```
>>> print(widget.render())
<select id="foo" name="bar:list" class="select-widget required set-field"
        multiple="multiple"  size="5">
  <option id="foo-0" value="1" selected="selected">1</option>
  <option id="foo-1" value="2">2</option>
  <option id="foo-2" value="3" selected="selected">3</option>
  <option id="foo-3" value="4">4</option>
</select>
<input name="bar-empty-marker" type="hidden" value="1" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="select-widget required set-field"><span
    class="selected-option">1</span>, <span
    class="selected-option">3</span></span>
```

2.5.38 SourceText

```
>>> field = zope.schema.SourceText(default='<source />')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.textarea.TextAreaWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from SourceText to TextAreaWidget>
```

```
>>> print(widget.render())
<textarea id="foo" name="bar"
          class="textarea-widget required sourcetext-field">&lt;source /&gt;</textarea>
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="textarea-widget required sourcetext-field">&lt;source /&gt;</span>
```

2.5.39 Text

```
>>> field = zope.schema.Text(default='Some\n Text.')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.textarea.TextAreaWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from Text to TextAreaWidget>
```

```
>>> print(widget.render())
<textarea id="foo" name="bar" class="textarea-widget required text-field">Some
 Text.</textarea>
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="textarea-widget required text-field">Some
  Text.</span>
```

2.5.40 TextLine

```
>>> field = zope.schema.TextLine(default='Some Text line.')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from TextLine to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required textline-field"
       value="Some Text line." />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required textline-field">Some Text line.</span>
```

2.5.41 Time

```
>>> field = zope.schema.Time(default=datetime.time(12, 0))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<TimeDataConverter converts from Time to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required time-field"
       value="12:00" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required time-field">12:00</span>
```

2.5.42 Timedelta

```
>>> field = zope.schema.Timedelta(default=datetime.timedelta(days=3))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<TimedeltaDataConverter converts from Timedelta to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required timedelta-field"
       value="3 days, 0:00:00" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required timedelta-field">3 days, 0:00:00</span>
```

2.5.43 Tuple - ASCII

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.ASCII(
...         title='ASCII',
...         default='This is\n ASCII.'),
...     default=('foo\nfoo', 'bar\nbar'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>ASCII</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
```

(continues on next page)

(continued from previous page)

```
                    id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-0" name="bar.0"
               class="textarea-widget required ascii-field">foo
foo</textarea>
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>ASCII</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-1" name="bar.1"
               class="textarea-widget required ascii-field">bar
bar</textarea>
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.44 Tuple - ASCIILine

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.ASCIILine(
...         title='ASCIILine',
...         default='An ASCII line.'),
...     default=('foo', 'bar'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>ASCIILine</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required asciiline-field"
               value="foo" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>ASCIILine</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required asciiline-field"
               value="bar" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.45 Tuple - Choice

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Choice(values=(1, 2, 3, 4)),
...     default=(1, 3) )
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.orderedselect.OrderedSelectWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<CollectionSequenceDataConverter converts from Tuple to OrderedSelectWidget>
```

```
>>> print(widget.render())
<script src="++resource++orderedselect_input.js" type="text/javascript"></script>
<table border="0" class="ordered-selection-field" id="foo">
  <tr>
    <td>
      <select id="foo-from" name="bar.from" size="5"
              multiple="multiple"
              class="required tuple-field">
        <option value="2">2</option>
        <option value="4">4</option>
      </select>
    </td>
    <td>
      <button name="from2toButton" type="button"
              value="&rarr;"
              onclick="javascript:from2to('foo')">&rarr;</button>
      <br />
      <button name="to2fromButton" type="button"
              value="&larr;"
              onclick="javascript:to2from('foo')">&larr;</button>
    </td>
    <td>
      <select id="foo-to" name="bar.to" size="5"
              multiple="multiple" class="required tuple-field">
        <option value="1">1</option>
        <option value="3">3</option>
      </select>
      <input name="bar-empty-marker" type="hidden" />
      <span id="foo-toDataContainer" style="display: none">
        <script type="text/javascript">
          copyDataForSubmit('foo');</script>
      </span>
    </td>
    <td>
      <button name="upButton" type="button" value="&uarr;"
              onclick="javascript:moveUp('foo')">&uarr;</button>
      <br />
      <button name="downButton" type="button" value="&darr;"
```

(continues on next page)

(continued from previous page)

```
              onclick="javascript:moveDown('foo')">&darr;</button>
    </td>
  </tr>
</table>
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="required tuple-field"><span
  class="selected-option">1</span>, <span
  class="selected-option">3</span></span>
```

2.5.46 Tuple - Date

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Date(
...         title='Date',
...         default=datetime.date(2007, 4, 1)),
...     default=(datetime.date(2008, 9, 27), datetime.date(2008, 9, 28)))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Date</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required date-field"
               value="08/09/27" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
```

(continues on next page)

(continued from previous page)

```
        <div class="label">
          <label for="foo-1">
            <span>Date</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required date-field"
               value="08/09/28" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.47 Tuple - Datetime

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Datetime(
...         title='Datetime',
...         default=datetime.datetime(2007, 4, 1, 12)),
...     default=(datetime.datetime(2008, 9, 27, 12),
...              datetime.datetime(2008, 9, 28, 12)))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
```

(continues on next page)

(continued from previous page)

```
        <div class="label">
          <label for="foo-0">
            <span>Datetime</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required datetime-field"
               value="08/09/27 12:00" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Datetime</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required datetime-field"
               value="08/09/28 12:00" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.48 Tuple - Decimal

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Decimal(
...         title='Decimal',
...         default=decimal.Decimal('1265.87')),
...     default=(decimal.Decimal('123.456'), decimal.Decimal('1')))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Decimal</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required decimal-field"
               value="123.456" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Decimal</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required decimal-field" value="1" />
```

(continues on next page)

(continued from previous page)

```
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.49 Tuple - DottedName

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.DottedName(
...         title='DottedName',
...         default='z3c.form'),
...     default=('z3c.form', 'z3c.wizard'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>DottedName</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required dottedname-field"
               value="z3c.form" />
        </div>
```

(continues on next page)

(continued from previous page)

```
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>DottedName</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required dottedname-field"
               value="z3c.wizard" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.50 Tuple - Float

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Float(
...         title='Float',
...         default=123.456),
...     default=(1234.5, 1))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
```

(continues on next page)

(continued from previous page)

```
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Float</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required float-field"
               value="1,234.5" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Float</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required float-field" value="1.0" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.51 Tuple - Id

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Id(
...         title='Id',
...         default='z3c.form'),
...     default=('z3c.form', 'z3c.wizard'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Id</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required id-field"
               value="z3c.form" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Id</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required id-field"
```

(continues on next page)

(continued from previous page)

```
               value="z3c.wizard" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.52 Tuple - Int

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Int(
...         title='Int',
...         default=666),
...     default=(42, 43))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Int</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required int-field" value="42" />
        </div>
```

(continues on next page)

(continued from previous page)

```
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Int</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required int-field" value="43" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.53 Tuple - Password

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Password(
...         title='Password',
...         default='mypwd'),
...     default=('pwd', 'pass'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
```

(continues on next page)

(continued from previous page)

```
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Password</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="password" id="foo-0" name="bar.0"
              class="password-widget required password-field" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Password</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="password" id="foo-1" name="bar.1"
              class="password-widget required password-field" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.54 Tuple - SourceText

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.SourceText(
...         title='SourceText',
...         default='<source />'),
...     default=('<html></body>foo</body></html>', '<h1>bar</h1>'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>SourceText</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-0" name="bar.0"
              class="textarea-widget required sourcetext-field">&lt;html&gt;&lt;/body&
↪gt;foo&lt;/body&gt;&lt;/html&gt;</textarea>
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>SourceText</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-1" name="bar.1"
              class="textarea-widget required sourcetext-field">&lt;h1&gt;bar&lt;/h1&gt;
```

(continues on next page)

(continued from previous page)

```
↪</textarea>
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.55 Tuple - Text

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Text(
...         title='Text',
...         default='Some\n Text.'),
...     default=('foo\nfoo', 'bar\nbar'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Text</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-0" name="bar.0"
          class="textarea-widget required text-field">foo
foo</textarea>
```

(continues on next page)

(continued from previous page)

```
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Text</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><textarea id="foo-1" name="bar.1"
          class="textarea-widget required text-field">bar
bar</textarea>
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.56 Tuple - TextLine

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.TextLine(
...         title='TextLine',
...         default='Some Text line.'),
...     default=('foo', 'bar'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>TextLine</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required textline-field"
               value="foo" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>TextLine</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required textline-field"
               value="bar" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.57 Tuple - Time

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Time(
...         title='Time',
...         default=datetime.time(12, 0)),
...     default=(datetime.time(13, 0), datetime.time(14, 0)))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Time</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required time-field" value="13:00" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Time</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required time-field" value="14:00" />
        </div>
```

(continues on next page)

(continued from previous page)

```
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.58 Tuple - Timedelta

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.Timedelta(
...         title='Timedelta',
...         default=datetime.timedelta(days=3)),
...     default=(datetime.timedelta(days=4), datetime.timedelta(days=5)))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>Timedelta</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required timedelta-field"
               value="4 days, 0:00:00" />
        </div>
      </div>
```

(continues on next page)

(continued from previous page)

```
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>Timedelta</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required timedelta-field"
               value="5 days, 0:00:00" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.59 Tuple - URI

```
>>> field = zope.schema.Tuple(
...     value_type=zope.schema.URI(
...         title='URI',
...         default='http://zope.org'),
...     default=('http://www.python.org', 'http://www.zope.com'))
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.multi.MultiWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<MultiConverter converts from Tuple to MultiWidget>
```

```
>>> print(widget.render())
<div class="multi-widget required">
```

(continues on next page)

(continued from previous page)

```
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>URI</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-0-remove" name="bar.0.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-0" name="bar.0"
               class="text-widget required uri-field"
               value="http://www.python.org" />
        </div>
      </div>
    </div>
    <div id="foo-1-row" class="row">
        <div class="label">
          <label for="foo-1">
            <span>URI</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="foo-1-remove" name="bar.1.remove" />
          </div>
          <div class="multi-widget-input"><input type="text" id="foo-1" name="bar.1"
               class="text-widget required uri-field"
               value="http://www.zope.com" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="bar-buttons-add"
       name="bar.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="bar-buttons-remove"
       name="bar.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="bar.count" value="2" />
```

2.5.60 URI

```
>>> field = zope.schema.URI(default='http://zope.org')
>>> widget = setupWidget(field)
>>> widget.update()
```

```
>>> widget.__class__
<class 'z3c.form.browser.text.TextWidget'>
```

```
>>> interfaces.IDataConverter(widget)
<FieldDataConverter converts from URI to TextWidget>
```

```
>>> print(widget.render())
<input type="text" id="foo" name="bar" class="text-widget required uri-field"
       value="http://zope.org" />
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="foo" class="text-widget required uri-field">http://zope.org</span>
```

Calling the widget will return the widget including the layout

```
>>> print(widget())
<div id="foo-row" class="row-required row">
  <div class="label">
    <label for="foo">
      <span></span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <span id="foo" class="text-widget required uri-field">http://zope.org</span>
</div>
</div>
```

CHAPTER

THREE

INFORMATIVE

3.1 Attribute Value Adapters

In advanced, highly customized projects it is often the case that a property wants to be overridden for a particular customer in a particular case. A prime example is the label of a widget. Until this implementation of a form framework was written, widgets only could get their label from the field they were representing. Thus, wanting to change the label of a widget meant implementing a custom schema and re-registering the form in question for the custom schema. It is needless to say that this was very annoying.

For this form framework, we are providing multiple levels of customization. The user has the choice to change the value of an attribute through attribute assignment or adapter lookup. The chronological order of an attribute value assignment is as follows:

1. During initialization or right thereafter, the attribute value can be set by direct attribute assignment, i.e. `obj.attr = value`
2. While updating the object, an adapter is looked up for the attribute. If an adapter is found, the attribute value will be overridden. Of course, if the object does not have an `update()` method, one can choose another location to do the adapter lookup.
3. After updating, the developer again has the choice to override the attribute allowing granularity above and beyond the adapter.

The purpose of this module is to implement the availability of an attribute value using an adapter.

```
>>> from z3c.form import value
```

The module provides helper functions and classes, to create those adapters with as little code as possible.

3.1.1 Static Value Adapter

To demonstrate the static value adapter, let's go back to our widget label example. Let's create a couple of simple widgets and forms first:

```
>>> class TextWidget(object):
...     label = u'Text'
>>> tw = TextWidget()
```

```
>>> class CheckboxWidget(object):
...     label = u'Checkbox'
>>> cbw = CheckboxWidget()
```

```
>>> class Form1(object):
...     pass
>>> form1 = Form1()
```

```
>>> class Form2(object):
...     pass
>>> form2 = Form2()
```

We can now create a generic widget property adapter:

```
>>> WidgetAttribute = value.StaticValueCreator(
...     discriminators = ('widget', 'view')
...     )
```

Creating the widget attribute object, using the helper function above, allows us to define the discriminators (or the granulatrity) that can be used to control a widget attribute by an adapter. In our case this is the widget itself and the form/view in which the widget is displayed. In other words, it will be possible to register a widget attribute value specifically for a particular widget, a particular form, or a combination thereof.

Let's now create a label attribute adapter for the text widget, since our customer does not like the default label:

```
>>> TextLabel = WidgetAttribute(u'My Text', widget=TextWidget)
```

The first argument of any static attribute value is the value itself, in our case the string "My Text". The following keyword arguments are the discriminators specified in the property factory. Since we only specify the widget, the label will be available to all widgets. But first we have to register the adapter:

```
>>> import zope.component
>>> zope.component.provideAdapter(TextLabel, name='label')
```

The name of the adapter is the attribute name of the widget. Let's now see how we can get the label:

```
>>> from z3c.form import interfaces
>>> staticValue = zope.component.getMultiAdapter(
...     (tw, form1), interfaces.IValue, name='label')
>>> staticValue
<StaticValue u'My Text'>
```

The resulting value object has one public method `get()`, which returns the actual value:

```
>>> staticValue.get()
u'My Text'
```

As we said before, the value should be available to all forms, ...

```
>>> zope.component.getMultiAdapter(
...     (tw, form2), interfaces.IValue, name='label')
<StaticValue u'My Text'>
```

... but only to the `TextWidget`:

```
>>> zope.component.getMultiAdapter(
...     (cbw, form2), interfaces.IValue, name='label')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ComponentLookupError: ((<CheckboxWidget...>, <Form2...>),
                       <InterfaceClass ...IValue>, 'label')
```

By the way, the attribute adapter factory notices, if you specify a discriminator that was not specified:

```
>>> WidgetAttribute(u'My Text', form=Form2)
Traceback (most recent call last):
...
ValueError: One or more keyword arguments did not match the discriminators.
```

```
>>> WidgetAttribute.discriminators
('widget', 'view')
```

3.1.2 Computed Value Adapter

A second implementation of the value adapter in the evaluated value, where one can specify a function that computes the value to be returned. The only argument to the function is the value adapter instance itself, which then contains all the discriminators as specified when creating the generic widget attribute factory. Let's take the same use case as before, but generating the value as follows:

```
>>> def getLabelValue(adapter):
...     return adapter.widget.label + ' (1)'
```

Now we create the value adapter for it:

```
>>> WidgetAttribute = value.ComputedValueCreator(
...     discriminators = ('widget', 'view')
...     )
```

```
>>> TextLabel = WidgetAttribute(getLabelValue, widget=TextWidget)
```

After registering the adapter, ...

```
>>> zope.component.provideAdapter(TextLabel, name='label')
```

we now get the answers:

```
>>> from z3c.form import interfaces
>>> zope.component.getMultiAdapter(
...     (tw, form1), interfaces.IValue, name='label')
<ComputedValue u'Text (1)'>
```

__Note__: The two implementations of the attribute value adapters are not
meant to be canonical features that must always be used. The API is kept simple to allow you to quickly implement your own value adapter.

3.1.3 Automatic Interface Assignment

Oftentimes it is desirable to register an attribute value adapter for an instance. A good example is a field, so let's create a small schema:

```
>>> import zope.interface
>>> import zope.schema
>>> class IPerson(zope.interface.Interface):
...     firstName = zope.schema.TextLine(title=u'First Name')
...     lastName = zope.schema.TextLine(title=u'Last Name')
```

The customer now requires that the title – which is the basis of the widget label for field widgets – of the last name should be "Surname". Until now the option was to write a new schema changing the title. With this attribute value module, as introduced thus far, we would need to provide a special interface for the last name field, since registering a label adapter for all text fields would also change the first name.

Before demonstrating the solution to this problem, let's first create a field attribute value:

```
>>> FieldAttribute = value.StaticValueCreator(
...     discriminators = ('field',)
...     )
```

We can now create the last name title, changing only the title of the `lastName` field. Instead of passing in an interface of class as the field discriminator, we pass in the field instance:

```
>>> LastNameTitle = FieldAttribute(u'Surname', field=IPerson['lastName'])
```

The attribute value factory will automatically detect instances, create an interface on the fly, directly provide it on the field and makes it the discriminator interface for the adapter registratioon.

So after registering the adapter, ...

```
>>> zope.component.provideAdapter(LastNameTitle, name='title')
```

the adapter is only available to the last name field and not the first name:

```
>>> zope.component.queryMultiAdapter(
...     (IPerson['lastName'],), interfaces.IValue, name='title')
<StaticValue u'Surname'>
```

```
>>> zope.component.queryMultiAdapter(
...     (IPerson['firstName'],), interfaces.IValue, name='title')
```

3.2 Data Managers

For the longest time the way widgets retrieved and stored their values on the actual content/model was done by binding the field to a context and then setting and getting the attribute from it. This has several distinct design shortcomings:

1. The field has too much responsibility by knowing about its implementations.
2. There is no way of redefining the method used to store and access data other than rewriting fields.
3. Finding the right content/model to modify is an implicit policy: Find an adapter for the field's schema and then set the value there.

While implementing some real-world projects, we noticed that this approach is too limiting and we often could not use the form framework when we wanted or had to jump through many hoops to make it work for us. For example, if we want to display a form to collect data that does not correspond to a set of content components, we were forced to not only write a schema for the form, but also implement that schema as a class. but all we wanted was a dictionary. For edit-form like tasks we often also had an initial dictionary, which we just wanted modified.

Data managers abstract the getting and setting of the data. A data manager is responsible for setting one piece of data in a particular context.

```
>>> from z3c.form import datamanager
```

3.2.1 Attribute Field Manager

The most common case, of course, is the management of class attributes through fields. In this case, the data manager needs to know about the context and the field it is managing the data for.

```
>>> import zope.interface
>>> import zope.schema
>>> class IPerson(zope.interface.Interface):
...     name = zope.schema.TextLine(
...         title='Name',
...         default='<no name>')
...     phone = zope.schema.TextLine(
...         title='Phone')
```

```
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...     name = ''
...     def __init__(self, name):
...         self.name = name
```

```
>>> stephan = Person('Stephan Richter')
```

We can now instantiate the data manager for Stephan's name:

```
>>> nameDm = datamanager.AttributeField(stephan, IPerson['name'])
```

The data manager consists of a few simple methods to accomplish its purpose. Getting the value is done using the `get()` or `query()` method:

```
>>> nameDm.get()
'Stephan Richter'
```

```
>>> nameDm.query()
'Stephan Richter'
```

The value can be set using `set()`:

```
>>> nameDm.set('Stephan "Caveman" Richter')
```

```
>>> nameDm.get()
'Stephan "Caveman" Richter'
```

(continues on next page)

(continued from previous page)

```
>>> stephan.name
'Stephan "Caveman" Richter'
```

If an attribute is not available, `get()` fails and `query()` returns a default value:

```
>>> phoneDm = datamanager.AttributeField(stephan, IPerson['phone'])
```

```
>>> phoneDm.get()
Traceback (most recent call last):
...
AttributeError: 'Person' object has no attribute 'phone'
```

```
>>> phoneDm.query()
<NO_VALUE>
>>> phoneDm.query('nothing')
'nothing'
```

A final feature that is supported by the data manager is the check whether a value can be accessed and written. When the context is not security proxied, both, accessing and writing, is allowed:

```
>>> nameDm.canAccess()
True
>>> nameDm.canWrite()
True
```

To demonstrate the behavior for a security-proxied component, we first have to provide security declarations for our person:

```
>>> from zope.security.management import endInteraction
>>> from zope.security.management import newInteraction
>>> from zope.security.management import setSecurityPolicy
>>> import z3c.form.testing
>>> endInteraction()
>>> newPolicy = z3c.form.testing.SimpleSecurityPolicy()
>>> newPolicy.allowedPermissions = ('View', 'Edit')
>>> oldpolicy = setSecurityPolicy(newPolicy)
>>> newInteraction()
```

```
>>> from zope.security.checker import Checker
>>> from zope.security.checker import defineChecker
>>> personChecker = Checker({'name':'View', 'name':'Edit'})
>>> defineChecker(Person, personChecker)
```

We now need to wrap stephan into a proxy:

```
>>> protectedStephan = zope.security.checker.ProxyFactory(stephan)
```

Since we are not logged in as anyone, we cannot acces or write the value:

```
>>> nameDm = datamanager.AttributeField(protectedStephan, IPerson['name'])
```

```
>>> nameDm.canAccess()
False
>>> nameDm.canWrite()
False
```

Clearly, this also means that `get()` and `set()` are also shut off:

```
>>> nameDm.get()
Traceback (most recent call last):
...
Unauthorized: (<Person object at ...>, 'name', 'Edit')
```

```
>>> nameDm.set('Stephan')
Traceback (most recent call last):
...
ForbiddenAttribute: ('name', <Person object at ...>)
```

Now we have to setup the security system and "log in" as a user:

```
>>> newPolicy.allowedPermissions = ('View', 'Edit')
>>> newPolicy.loggedIn = True
```

The created principal, with which we are logged in now, can only access the attribute:

```
>>> nameDm.canAccess()
True
>>> nameDm.canWrite()
False
```

Thus only the `get()` method is allowed:

```
>>> nameDm.get()
'Stephan "Caveman" Richter'
```

```
>>> nameDm.set('Stephan')
Traceback (most recent call last):
...
ForbiddenAttribute: ('name', <Person object at ...>)
```

If field's schema is not directly provided by the context, the datamanager will attempt to find an adapter. Let's give the person an address for example:

```
>>> class IAddress(zope.interface.Interface):
...     city = zope.schema.TextLine(title='City')
```

```
>>> @zope.interface.implementer(IAddress)
... class Address(object):
...     zope.component.adapts(IPerson)
...     def __init__(self, person):
...         self.person = person
...     @property
...     def city(self):
...         return getattr(self.person, '_city', None)
```

(continues on next page)

(continued from previous page)

```
...     @city.setter
...     def city(self, value):
...         self.person._city = value
```

```
>>> zope.component.provideAdapter(Address)
```

Now we can create a data manager for the city attribute:

```
>>> cityDm = datamanager.AttributeField(stephan, IAddress['city'])
```

We can access and write to the city attribute:

```
>>> cityDm.canAccess()
True
>>> cityDm.canWrite()
True
```

Initially there is no value, but of course we can create one:

```
>>> cityDm.get()
```

```
>>> cityDm.set('Maynard')
>>> cityDm.get()
'Maynard'
```

The value can be accessed through the adapter itself as well:

```
>>> IAddress(stephan).city
'Maynard'
```

While we think that implicitly looking up an adapter is not the cleanest solution, it allows us to mimic the behavior of `zope.formlib`. We think that we will eventually provide alternative ways to accomplish the same in a more explicit way.

If we try to set a value that is read-only, a type error is raised:

```
>>> readOnlyName = zope.schema.TextLine(
...     __name__='name',
...     readonly=True)
```

```
>>> nameDm = datamanager.AttributeField(stephan, readOnlyName)
>>> nameDm.set('Stephan')
Traceback (most recent call last):
...
TypeError: Can't set values on read-only fields
           (name=name, class=__builtin__.Person)
```

Finally, we instantiate the data manager with a `zope.schema` field. And we can access the different methods like before.

```
>>> nameDm = datamanager.AttributeField(
...    stephan, zope.schema.TextLine(__name__ = 'name'))
>>> nameDm.canAccess()
```

(continues on next page)

(continued from previous page)

```
True
>>> nameDm.canWrite()
True
```

```
>>> nameDm.get()
'Stephan "Caveman" Richter'
>>> nameDm.query()
'Stephan "Caveman" Richter'
```

```
>>> nameDm.set('Stephan Richter')
>>> nameDm.get()
'Stephan Richter'
```

3.2.2 Dictionary Field Manager

Another implementation of the data manager interface is provided by the dictionary field manager, which does not expect an instance with attributes as its context, but a dictionary. It still uses a field to determine the key to modify.

```
>>> personDict = {}
>>> nameDm = datamanager.DictionaryField(personDict, IPerson['name'])
```

The datamanager can really only deal with dictionaries and mapping types:

```
>>> import zope.interface.common.mapping
>>> import persistent.mapping
>>> import persistent.dict
>>> @zope.interface.implementer(zope.interface.common.mapping.IMapping)
... class MyMapping(object):
...     pass
>>> datamanager.DictionaryField(MyMapping(), IPerson['name'])
<z3c.form.datamanager.DictionaryField object at ...>
>>> datamanager.DictionaryField(persistent.mapping.PersistentMapping(),
...     IPerson['name'])
<z3c.form.datamanager.DictionaryField object at ...>
>>> datamanager.DictionaryField(persistent.dict.PersistentDict(),
...     IPerson['name'])
<z3c.form.datamanager.DictionaryField object at ...>
```

```
>>> datamanager.DictionaryField([], IPerson['name'])
Traceback (most recent call last):
...
ValueError: Data are not a dictionary: <type 'list'>
```

Let's now access the name:

```
>>> nameDm.get()
Traceback (most recent call last):
...
AttributeError
```

```
>>> nameDm.query()
<NO_VALUE>
```

Initially we get the default value (as specified in the field), since the person dictionariy has no entry. If no default value has been specified in the field, the missing value is returned.

Now we set a value and it should be available:

```
>>> nameDm.set('Roger Ineichen')
```

```
>>> nameDm.get()
'Roger Ineichen'
>>> personDict
{'name': 'Roger Ineichen'}
```

Since this dictionary is not security proxied, any field can be accessed and written to:

```
>>> nameDm.canAccess()
True
>>> nameDm.canWrite()
True
```

As with the attribute data manager, readonly fields cannot be set:

```
>>> nameDm = datamanager.DictionaryField(personDict, readOnlyName)
>>> nameDm.set('Stephan')
Traceback (most recent call last):
...
TypeError: Can't set values on read-only fields name=name
```

3.2.3 Cleanup

We clean up the changes we made in these examples:

```
>>> endInteraction()
>>> ignore = setSecurityPolicy(oldpolicy)
```

3.3 Data Converter

The data converter is the component that converts an internal data value as described by a field to an external value as required by a widget and vice versa. The goal of the converter is to avoid field and widget proliferation solely to handle different types of values. The purpose of fields is to describe internal data types and structures and that of widgets to provide one particular mean of input.

The only two discriminators for the converter are the field and the widget.

Let's look at the `Int` field to `TextWidget` converter as an example:

```
>>> import zope.schema
>>> age = zope.schema.Int(
...     __name__='age',
```

(continues on next page)

(continued from previous page)

```
...     title='Age',
...     min=0)
```

```
>>> from z3c.form.testing import TestRequest
>>> from z3c.form import widget
>>> text = widget.Widget(TestRequest())
```

```
>>> from z3c.form import converter
>>> conv = converter.FieldDataConverter(age, text)
```

The field data converter is a generic data converter that can be used for all fields that implement `IFromUnicode`. If, for example, a `Date` field – which does not provide `IFromUnicode` – is passed in, then a type error is raised:

```
>>> converter.FieldDataConverter(zope.schema.Date(), text)
Traceback (most recent call last):
...
TypeError: Field of type ``Date`` must provide ``IFromUnicode``.
```

A named field will tell it's name:

```
>>> converter.FieldDataConverter(zope.schema.Date(__name__="foobar"), text)
Traceback (most recent call last):
...
TypeError: Field ``foobar`` of type ``Date`` must provide ``IFromUnicode``.
```

However, the `FieldDataConverter` is registered for `IField`, since many fields (like `Decimal`) for which we want to create custom converters provide `IFromUnicode` more specifically than their characterizing interface (like `IDecimal`).

The converter can now convert any integer to a the value the test widget deals with, which is an ASCII string:

```
>>> conv.toWidgetValue(34)
'34'
```

When the missing value is passed in, an empty string should be returned:

```
>>> conv.toWidgetValue(age.missing_value)
''
```

Of course, values can also be converted from a widget value to field value:

```
>>> conv.toFieldValue('34')
34
```

An empty string means simply that the value is missing and the missing value of the field is returned:

```
>>> age.missing_value = -1
>>> conv.toFieldValue('')
-1
```

Of course, trying to convert a non-integer string representation fails in a conversion error:

```
>>> conv.toFieldValue('3.4')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
InvalidIntLiteral: invalid literal for int(): 3.4
```

Also, the conversion to the field value also validates the data; in this case negative values are not allowed:

```
>>> conv.toFieldValue('-34')
Traceback (most recent call last):
...
TooSmall: (-34, 0)
```

That's pretty much the entire API. When dealing with converters within the component architecture, everything is a little bit simpler. So let's register the converter:

```
>>> import zope.component
>>> zope.component.provideAdapter(converter.FieldDataConverter)
```

Once we ensure that our widget is a text widget, we can lookup the adapter:

```
>>> import zope.interface
>>> from z3c.form import interfaces
>>> zope.interface.alsoProvides(text, interfaces.ITextWidget)
```

```
>>> zope.component.getMultiAdapter((age, text), interfaces.IDataConverter)
<FieldDataConverter converts from Int to Widget>
```

For field-widgets there is a helper adapter that makes the lookup even simpler:

```
>>> zope.component.provideAdapter(converter.FieldWidgetDataConverter)
```

After converting our simple widget to a field widget,

```
>>> fieldtext = widget.FieldWidget(age, text)
```

we can now lookup the data converter adapter just by the field widget itself:

```
>>> interfaces.IDataConverter(fieldtext)
<FieldDataConverter converts from Int to Widget>
```

3.3.1 Number Data Converters

As hinted on above, the package provides a specific data converter for each of the three main numerical types: `int`, `float`, `Decimal`. Specifically, those data converters support full localization of the number formatting.

```
>>> age = zope.schema.Int()
>>> intdc = converter.IntegerDataConverter(age, text)
>>> intdc
<IntegerDataConverter converts from Int to Widget>
```

Since the age is so small, the formatting is trivial:

```
>>> intdc.toWidgetValue(34)
'34'
```

But if we increase the number, the grouping seprator will be used:

```
>>> intdc.toWidgetValue(3400)
'3,400'
```

An empty string is returned, if the missing value is passed in:

```
>>> intdc.toWidgetValue(None)
''
```

Of course, parsing these outputs again, works as well:

```
>>> intdc.toFieldValue('34')
34
```

But if we increase the number, the grouping seprator will be used:

```
>>> intdc.toFieldValue('3,400')
3400
```

Luckily our parser is somewhat forgiving, and even allows for missing group characters:

```
>>> intdc.toFieldValue('3400')
3400
```

If an empty string is passed in, the missing value of the field is returned:

```
>>> intdc.toFieldValue('')
```

Finally, if the input does not match at all, then a validation error is returned:

```
>>> intdc.toFieldValue('fff')
Traceback (most recent call last):
...
FormatterValidationError:
    ('The entered value is not a valid integer literal.', 'fff')
```

The formatter validation error derives from the regular validation error, but allows you to specify the message that is output when asked for the documentation:

```
>>> err = converter.FormatterValidationError('Something went wrong.', None)
>>> err.doc()
'Something went wrong.'
```

Let's now look at the float data converter.

```
>>> rating = zope.schema.Float()
>>> floatdc = converter.FloatDataConverter(rating, text)
>>> floatdc
<FloatDataConverter converts from Float to Widget>
```

Again, you can format and parse values:

```
>>> floatdc.toWidgetValue(7.43)
'7.43'
```

(continues on next page)

(continued from previous page)

```
>>> floatdc.toWidgetValue(10239.43)
'10,239.43'
```

```
>>> floatdc.toFieldValue('7.43') == 7.43
True
>>> type(floatdc.toFieldValue('7.43'))
<class 'float'>
>>> floatdc.toFieldValue('10,239.43')
10239.43
```

The error message, however, is customized to the floating point:

```
>>> floatdc.toFieldValue('fff')
Traceback (most recent call last):
...
FormatterValidationError:
    ('The entered value is not a valid decimal literal.', 'fff')
```

The decimal converter works like the other two before.

```
>>> money = zope.schema.Decimal()
>>> decimaldc = converter.DecimalDataConverter(money, text)
>>> decimaldc
<DecimalDataConverter converts from Decimal to Widget>
```

Formatting and parsing should work just fine:

```
>>> import decimal
```

```
>>> decimaldc.toWidgetValue(decimal.Decimal('7.43'))
'7.43'
>>> decimaldc.toWidgetValue(decimal.Decimal('10239.43'))
'10,239.43'
```

```
>>> decimaldc.toFieldValue('7.43')
Decimal("7.43")
>>> decimaldc.toFieldValue('10,239.43')
Decimal("10239.43")
```

Again, the error message, is customized to the floating point:

```
>>> floatdc.toFieldValue('fff')
Traceback (most recent call last):
...
FormatterValidationError:
    ('The entered value is not a valid decimal literal.', 'fff')
```

3.3.2 Bool Data Converter

```
>>> yesno = zope.schema.Bool()
>>> yesnowidget = widget.Widget(TestRequest())
>>> conv = converter.FieldDataConverter(yesno, yesnowidget)
>>> conv.toWidgetValue(True)
'True'
```

```
>>> conv.toWidgetValue(False)
'False'
```

3.3.3 Text Data Converters

Users often add empty spaces by mistake, for example when copy-pasting content into the form.

```
>>> name = zope.schema.TextLine()
>>> namewidget = widget.Widget(TestRequest())
>>> conv = converter.FieldDataConverter(name, namewidget)
>>> conv.toFieldValue('Einstein ')
'Einstein'
```

3.3.4 Date Data Converter

Since the `Date` field does not provide `IFromUnicode`, we have to provide a custom data converter. This default one is not very sophisticated and is inteded for use with the text widget:

```
>>> date = zope.schema.Date()
```

```
>>> ddc = converter.DateDataConverter(date, text)
>>> ddc
<DateDataConverter converts from Date to Widget>
```

Dates are simply converted to ISO format:

```
>>> import datetime
>>> bday = datetime.date(1980, 1, 25)
```

```
>>> ddc.toWidgetValue(bday)
'80/01/25'
```

If the date is the missing value, an empty string is returned:

```
>>> ddc.toWidgetValue(None)
''
```

The converter only knows how to convert this particular format back to a datetime value:

```
>>> ddc.toFieldValue('80/01/25')
datetime.date(1980, 1, 25)
```

By default the converter converts missing input to missin_input value:

```
>>> ddc.toFieldValue('') is None
True
```

If the passed in string cannot be parsed, a formatter validation error is raised:

```
>>> ddc.toFieldValue('8.6.07')
Traceback (most recent call last):
...
FormatterValidationError: ("The datetime string did not match the pattern
                            'yy/MM/dd'.", '8.6.07')
```

3.3.5 Time Data Converter

Since the `Time` field does not provide `IFromUnicode`, we have to provide a custom data converter. This default one is not very sophisticated and is inteded for use with the text widget:

```
>>> time = zope.schema.Time()
```

```
>>> tdc = converter.TimeDataConverter(time, text)
>>> tdc
<TimeDataConverter converts from Time to Widget>
```

Dates are simply converted to ISO format:

```
>>> noon = datetime.time(12, 0, 0)
```

```
>>> tdc.toWidgetValue(noon)
'12:00'
```

The converter only knows how to convert this particular format back to a datetime value:

```
>>> tdc.toFieldValue('12:00')
datetime.time(12, 0)
```

By default the converter converts missing input to missin_input value:

```
>>> tdc.toFieldValue('') is None
True
```

3.3.6 Datetime Data Converter

Since the `Datetime` field does not provide `IFromUnicode`, we have to provide a custom data converter. This default one is not very sophisticated and is inteded for use with the text widget:

```
>>> dtField = zope.schema.Datetime()
```

```
>>> dtdc = converter.DatetimeDataConverter(dtField, text)
>>> dtdc
<DatetimeDataConverter converts from Datetime to Widget>
```

Dates are simply converted to ISO format:

```
>>> bdayNoon = datetime.datetime(1980, 1, 25, 12, 0, 0)
```

```
>>> dtdc.toWidgetValue(bdayNoon)
'80/01/25 12:00'
```

The converter only knows how to convert this particular format back to a datetime value:

```
>>> dtdc.toFieldValue('80/01/25 12:00')
datetime.datetime(1980, 1, 25, 12, 0)
```

By default the converter converts missing input to missin_input value:

```
>>> dtdc.toFieldValue('') is None
True
```

3.3.7 Timedelta Data Converter

Since the `Timedelta` field does not provide `IFromUnicode`, we have to provide a custom data converter. This default one is not very sophisticated and is inteded for use with the text widget:

```
>>> timedelta = zope.schema.Timedelta()
```

```
>>> tddc = converter.TimedeltaDataConverter(timedelta, text)
>>> tddc
<TimedeltaDataConverter converts from Timedelta to Widget>
```

Dates are simply converted to ISO format:

```
>>> allOnes = datetime.timedelta(1, 3600+60+1)
```

```
>>> tddc.toWidgetValue(allOnes)
'1 day, 1:01:01'
```

The converter only knows how to convert this particular format back to a datetime value:

```
>>> fv = tddc.toFieldValue('1 day, 1:01:01')
>>> (fv.days, fv.seconds)
(1, 3661)
```

If no day is available, the following short form is used:

```
>>> noDay = datetime.timedelta(0, 3600+60+1)
>>> tddc.toWidgetValue(noDay)
'1:01:01'
```

And now back to the field value:

```
>>> fv = tddc.toFieldValue('1:01:01')
>>> (fv.days, fv.seconds)
(0, 3661)
```

By default the converter converts missing input to missin_input value:

```
>>> tddc.toFieldValue('') is None
True
```

3.3.8 File Upload Data Converter

Since the `Bytes` field can contain a `FileUpload` object, we have to make sure we can convert `FileUpload` objects to bytes too.

```
>>> import z3c.form.browser.file
>>> fileWidget = z3c.form.browser.file.FileWidget(TestRequest())
>>> bytes = zope.schema.Bytes()
```

```
>>> fudc = converter.FileUploadDataConverter(bytes, fileWidget)
>>> fudc
<FileUploadDataConverter converts from Bytes to FileWidget>
```

The file upload widget usually provides a file object. But sometimes is also provides a string:

```
>>> simple = 'foobar'
>>> fudc.toFieldValue(simple)
b'foobar'
```

The converter can also convert `FileUpload` objects. So we need to setup a fields storage stub ...

```
>>> class FieldStorageStub:
...     def __init__(self, file):
...         self.file = file
...         self.headers = {}
...         self.filename = 'foo.bar'
```

and a `FileUpload` component:

```
>>> from io import BytesIO
>>> from zope.publisher.browser import FileUpload
>>> myfile = BytesIO(b'File upload contents.')
>>> aFieldStorage = FieldStorageStub(myfile)
>>> myUpload = FileUpload(aFieldStorage)
```

Let's try to convert the input now:

```
>>> fudc.toFieldValue(myUpload)
b'File upload contents.'
```

By default the converter converts missing input to the `NOT_CHANGED` value:

```
>>> fudc.toFieldValue('')
<NOT_CHANGED>
```

This allows machinery later to ignore the field without sending all the data around.

If we get an empty filename in a `FileUpload` obejct, we also get the `missing_value`. But this really means that there was an error somewhere in the upload, since you are normaly not able to upload a file without a filename:

```
>>> class EmptyFilenameFieldStorageStub:
...     def __init__(self, file):
...         self.file = file
...         self.headers = {}
...         self.filename = ''
>>> myfile = BytesIO(b'')
>>> aFieldStorage = EmptyFilenameFieldStorageStub(myfile)
>>> myUpload = FileUpload(aFieldStorage)
>>> bytes = zope.schema.Bytes()
>>> fudc = converter.FileUploadDataConverter(bytes, fileWidget)
>>> fudc.toFieldValue(myUpload) is None
True
```

There is also a `ValueError` if we don't get a seekable file from the `FieldStorage` during the upload:

```
>>> myfile = ''
>>> aFieldStorage = FieldStorageStub(myfile)
>>> myUpload = FileUpload(aFieldStorage)
>>> bytes = zope.schema.Bytes()
>>> fudc = converter.FileUploadDataConverter(bytes, fileWidget)
>>> fudc.toFieldValue(myUpload) is None
Traceback (most recent call last):
...
ValueError: ('Bytes data are not a file object', ...AttributeError...)
```

When converting to the widget value, not conversion should be done, since bytes are not convertable in that sense.

```
>>> fudc.toWidgetValue(b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x04')
```

When the file upload widget is not used and a text-based widget is desired, then the regular field data converter will be chosen. Using a text widget, however, must be setup manually in the form with code like this:

```
fields['bytesField'].widgetFactory = TextWidget
```

3.3.9 Sequence Data Converter

For widgets and fields that work with choices of a sequence, a special data converter is required that works with terms. A prime example is a choice field. Before we can use the converter, we have to register some adapters:

```
>>> from z3c.form import term
>>> import zc.sourcefactory.browser.source
>>> import zc.sourcefactory.browser.token
>>> zope.component.provideAdapter(term.ChoiceTermsVocabulary)
>>> zope.component.provideAdapter(term.ChoiceTermsSource)
>>> zope.component.provideAdapter(term.ChoiceTerms)
>>> zope.component.provideAdapter(
...     zc.sourcefactory.browser.source.FactoredTerms)
>>> zope.component.provideAdapter(
...     zc.sourcefactory.browser.token.fromInteger)
```

The choice fields can be used together with vocabularies and sources.

Using vocabulary

Let's now create a choice field (using a vocabulary) and a widget:

```
>>> from zope.schema.vocabulary import SimpleVocabulary
```

```
>>> gender = zope.schema.Choice(
...     vocabulary = SimpleVocabulary([
...             SimpleVocabulary.createTerm(0, 'm', 'male'),
...             SimpleVocabulary.createTerm(1, 'f', 'female'),
...             ]) )
```

```
>>> from z3c.form import widget
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = gender
```

We now use the field and widget to instantiate the converter:

```
>>> sdv = converter.SequenceDataConverter(gender, seqWidget)
```

We can now convert a real value to a widget value, which will be the term's token:

```
>>> sdv.toWidgetValue(0)
['m']
```

The result is always a sequence, since sequence widgets only deal collections of values. Of course, we can convert the widget value back to an internal value:

```
>>> sdv.toFieldValue(['m'])
0
```

Sometimes a field is not required. In those cases, the internal value is the missing value of the field. The converter interprets that as no value being selected:

```
>>> gender.missing_value = 'missing'
```

```
>>> sdv.toWidgetValue(gender.missing_value)
[]
```

If the internal value is not a valid item in the terms, it is treated as missing:

```
>>> sdv.toWidgetValue(object())
[]
```

If "no value" has been specified in the widget, the missing value of the field is returned:

```
>>> sdv.toFieldValue(['--NOVALUE--'])
'missing'
```

An empty list will also cause the missing value to be returned:

```
>>> sdv.toFieldValue([])
'missing'
```

Using source

Let's now create a choice field (using a source) and a widget:

```
>>> from zc.sourcefactory.basic import BasicSourceFactory
>>> class GenderSourceFactory(BasicSourceFactory):
...     _mapping = {0: 'male', 1: 'female'}
...     def getValues(self):
...         return self._mapping.keys()
...     def getTitle(self, value):
...         return self._mapping[value]
>>> gender_source = zope.schema.Choice(
...     source = GenderSourceFactory())
```

```
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = gender_source
```

We now use the field and widget to instantiate the converter:

```
>>> sdv = converter.SequenceDataConverter(gender, seqWidget)
```

We can now convert a real value to a widget value, which will be the term's token:

```
>>> sdv.toWidgetValue(0)
['0']
```

The result is always a sequence, since sequence widgets only deal collections of values. Of course, we can convert the widget value back to an internal value:

```
>>> sdv.toFieldValue(['0'])
0
```

Sometimes a field is not required. In those cases, the internalvalue is the missing value of the field. The converter interprets that as no value being selected:

```
>>> gender.missing_value = 'missing'
```

```
>>> sdv.toWidgetValue(gender.missing_value)
[]
```

If "no value" has been specified in the widget, the missing value of the field is returned:

```
>>> sdv.toFieldValue(['--NOVALUE--'])
'missing'
```

An empty list will also cause the missing value to be returned:

```
>>> sdv.toFieldValue([])
'missing'
```

3.3.10 Collection Sequence Data Converter

For widgets and fields that work with a sequence of choices, another data converter is required that works with terms. A prime example is a list field. Before we can use the converter, we have to register the terms adapters:

```
>>> from z3c.form import term
>>> zope.component.provideAdapter(term.CollectionTerms)
>>> zope.component.provideAdapter(term.CollectionTermsVocabulary)
>>> zope.component.provideAdapter(term.CollectionTermsSource)
```

Collections can also use either vocabularies or sources.

Using vocabulary

Let's now create a list field (using the previously defined field using a vocabulary) and a widget:

```
>>> genders = zope.schema.List(value_type=gender)
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders
```

We now use the field and widget to instantiate the converter:

```
>>> csdv = converter.CollectionSequenceDataConverter(genders, seqWidget)
```

We can now convert a real value to a widget value, which will be the term's token:

```
>>> csdv.toWidgetValue([0])
['m']
```

The result is always a sequence, since sequence widgets only deal collections of values. Of course, we can convert the widget value back to an internal value:

```
>>> csdv.toFieldValue(['m'])
[0]
```

Of course, a collection field can also have multiple values:

```
>>> csdv.toWidgetValue([0, 1])
['m', 'f']
```

```
>>> csdv.toFieldValue(['m', 'f'])
[0, 1]
```

If any of the values are not a valid choice, they are simply ignored:

```
>>> csdv.toWidgetValue([0, 3])
['m']
```

Sometimes a field is not required. In those cases, the internal value is the missing value of the field. The converter interprets that as no values being given:

```
>>> genders.missing_value is None
True
```

(continues on next page)

(continued from previous page)

```
>>> csdv.toWidgetValue(genders.missing_value)
[]
```

For some field, like the `Set`, the collection type is a tuple. Sigh. In these cases we use the last entry in the tuple as the type to use:

```
>>> genders = zope.schema.Set(value_type=gender)
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders
```

```
>>> csdv = converter.CollectionSequenceDataConverter(genders, seqWidget)
```

```
>>> csdv.toWidgetValue({0})
['m']
```

```
>>> csdv.toFieldValue(['m'])
{0}
```

Getting Terms

As an optimization of this converter, the converter actually does not look up the terms itself but uses the widget's `terms` attribute. If the terms are not yet retrieved, the converter will ask the widget to do so when in need.

So let's see how this works when getting the widget value:

```
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders
```

```
>>> seqWidget.terms
```

```
>>> csdv = converter.CollectionSequenceDataConverter(genders, seqWidget)
>>> csdv.toWidgetValue([0])
['m']
```

```
>>> seqWidget.terms
<z3c.form.term.CollectionTermsVocabulary object ...>
```

The same is true when getting the field value:

```
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders
```

```
>>> seqWidget.terms
```

```
>>> csdv = converter.CollectionSequenceDataConverter(genders, seqWidget)
>>> csdv.toFieldValue(['m'])
{0}
```

```
>>> seqWidget.terms
<z3c.form.term.CollectionTermsVocabulary object ...>
```

Corner case: Just in case the field has a sequence as _type:

```
>>> class myField(zope.schema.List):
...     _type = (list, tuple)
```

```
>>> genders = myField(value_type=gender)
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders
```

We now use the field and widget to instantiate the converter:

```
>>> csdv = converter.CollectionSequenceDataConverter(genders, seqWidget)
```

The converter uses the last type (tuple in this case) to convert:

```
>>> csdv.toFieldValue(['m'])
(0,)
```

Using source

Let's now create a list field (using the previously defined field using a source) and a widget:

```
>>> genders_source = zope.schema.List(value_type=gender_source)
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders_source
```

We now use the field and widget to instantiate the converter:

```
>>> csdv = converter.CollectionSequenceDataConverter(
...     genders_source, seqWidget)
```

We can now convert a real value to a widget value, which will be the term's token:

```
>>> csdv.toWidgetValue([0])
['0']
```

The result is always a sequence, since sequence widgets only deal collections of values. Of course, we can convert the widget value back to an internal value:

```
>>> csdv.toFieldValue(['0'])
[0]
```

For some field, like the Set, the collection type is a tuple. Sigh. In these cases we use the last entry in the tuple as the type to use:

```
>>> genders_source = zope.schema.Set(value_type=gender_source)
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders_source
```

```
>>> csdv = converter.CollectionSequenceDataConverter(
...     genders_source, seqWidget)
```

```
>>> csdv.toWidgetValue({0})
['0']
```

```
>>> csdv.toFieldValue(['0'])
{0}
```

Getting Terms

As an optimization of this converter, the converter actually does not look up the terms itself but uses the widget's `terms` attribute. If the terms are not yet retrieved, the converter will ask the widget to do so when in need.

So let's see how this works when getting the widget value:

```
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders_source
```

```
>>> seqWidget.terms
```

```
>>> csdv = converter.CollectionSequenceDataConverter(
...     genders_source, seqWidget)
>>> csdv.toWidgetValue([0])
['0']
```

```
>>> seqWidget.terms
<z3c.form.term.CollectionTermsSource object ...>
```

The same is true when getting the field value:

```
>>> seqWidget = widget.SequenceWidget(TestRequest())
>>> seqWidget.field = genders_source
```

```
>>> seqWidget.terms
```

```
>>> csdv = converter.CollectionSequenceDataConverter(
...     genders_source, seqWidget)
>>> csdv.toFieldValue(['0'])
{0}
```

```
>>> seqWidget.terms
<z3c.form.term.CollectionTermsSource object ...>
```

3.3.11 Boolean to Single Checkbox Data Converter

The conversion from any field to the single checkbox widget value is a special case, because it has to be defined what selecting the value means. In the case of the boolean field, "selected" means `True` and if unselected, `False` is returned:

```
>>> boolField = zope.schema.Bool()
```

```
>>> bscbx = converter.BoolSingleCheckboxDataConverter(boolField, seqWidget)
>>> bscbx
<BoolSingleCheckboxDataConverter converts from Bool to SequenceWidget>
```

Let's now convert boolean field to widget values:

```
>>> bscbx.toWidgetValue(True)
['selected']
>>> bscbx.toWidgetValue(False)
[]
```

Converting back is equally simple:

```
>>> bscbx.toFieldValue(['selected'])
True
>>> bscbx.toFieldValue([])
False
```

Note that this widget has no concept of missing value, since it can only represent two states by desgin.

3.3.12 Text Lines Data Converter

For sequence widgets and fields that work with a sequence of *TextLine* value fields, a simple data converter is required. Let's create a list of text lines field and a widget:

```
>>> languages = zope.schema.List(
...     value_type=zope.schema.TextLine(),
...     default=[],
...     missing_value=None,
...     )
```

```
>>> from z3c.form.browser import textlines
>>> tlWidget = textlines.TextLinesWidget(TestRequest())
>>> tlWidget.field = languages
```

We now use the field and widget to instantiate the converter:

```
>>> tlc = converter.TextLinesConverter(languages, tlWidget)
```

We can now convert a real value to a widget value:

```
>>> tlc.toWidgetValue(['de', 'fr', 'en'])
'de\nfr\nen'
```

Empty entries are significant:

```
>>> tlc.toWidgetValue(['de', 'fr', 'en', ''])
'de\nfr\nen\n'
```

The result is always a string, since text lines widgets only deal with textarea as input field. Of course, we can convert the widget value back to an internal value:

```
>>> tlc.toFieldValue('de\nfr\nen')
['de', 'fr', 'en']
```

Each line should be one item:

```
>>> tlc.toFieldValue('this morning\ntomorrow evening\nyesterday')
['this morning', 'tomorrow evening', 'yesterday']
```

Empty lines are significant:

```
>>> tlc.toFieldValue('de\n\nfr\nen')
['de', '', 'fr', 'en']
```

Empty lines are also significant at the end:

```
>>> tlc.toFieldValue('de\nfr\nen\n')
['de', 'fr', 'en', '']
```

An empty string will also cause the missing value to be returned:

```
>>> tlc.toFieldValue('') is None
True
```

It also should work for schema fields that define their type as tuple, in former times zope.schema.Int declared its type as (int, long).

```
>>> class MyField(zope.schema.Int):
...     _type = (int, float)
>>> ids = zope.schema.List(
...     value_type=MyField(),
...     )
```

Let's illustrate the problem:

```
>>> MyField._type == (int, float)
True
```

The converter will use the first one.

```
>>> tlWidget.field = ids
>>> tlc = converter.TextLinesConverter(ids, tlWidget)
```

Of course, it still can convert to the widget value:

```
>>> tlc.toWidgetValue([1,2,3])
'1\n2\n3'
```

And back:

```
>>> tlc.toFieldValue('1\n2\n3')
[1, 2, 3]
```

An empty string will also cause the missing value to be returned:

```
>>> tlc.toFieldValue('') is None
True
```

Converting Missing value to Widget value returns ‘’:

```
>>> tlc.toWidgetValue(tlc.field.missing_value)
''
```

Just in case the field has sequence as its _type:

```
>>> class myField(zope.schema.List):
...     _type = (list, tuple)
```

```
>>> ids = myField(
...     value_type=zope.schema.Int(),
...     )
```

The converter will use the last one, tuple in this case.

```
>>> tlWidget.field = ids
>>> tlc = converter.TextLinesConverter(ids, tlWidget)
```

Of course, it still can convert to the widget value:

```
>>> tlc.toWidgetValue([1,2,3])
'1\n2\n3'
```

And back:

```
>>> tlc.toFieldValue('1\n2\n3')
(1, 2, 3)
```

What if we have a wrong number:

```
>>> tlc.toFieldValue('1\n2\n3\nfoo')
Traceback (most recent call last):
...
FormatterValidationError: ("invalid literal for int() with base 10: 'foo'", 'foo')
```

3.3.13 Multi Data Converter

For multi widgets and fields that work with a sequence of other basic types, a separate data converter is required. Let’s create a list of integers field and a widget:

```
>>> numbers = zope.schema.List(
...     value_type=zope.schema.Int(),
...     default=[],
```

(continues on next page)

(continued from previous page)

```
...     missing_value=None,
...     )
```

```
>>> from z3c.form.browser import multi
>>> multiWidget = multi.MultiWidget(TestRequest())
>>> multiWidget.field = numbers
```

Before we can convert, we have to regsiter a widget for the integer field:

```
>>> from z3c.form.browser import text
>>> zope.component.provideAdapter(
...     text.TextFieldWidget,
...     (zope.schema.Int, TestRequest))
```

We now use the field and widget to instantiate the converter:

```
>>> conv = converter.MultiConverter(numbers, multiWidget)
```

We can now convert a list of integers to the multi-widget internal representation:

```
>>> conv.toWidgetValue([1, 2, 3])
['1', '2', '3']
```

If the value is the missing value, an empty list is returned:

```
>>> conv.toWidgetValue(None)
[]
```

Now, let's look at the reverse:

```
>>> conv.toFieldValue(['1', '2', '3'])
[1, 2, 3]
```

If the list is empty, the missing value is returned:

```
>>> conv.toFieldValue([]) is None
True
```

Just in case the field has sequence as its _type:

```
>>> @zope.interface.implementer(zope.schema.interfaces.IList)
... class MySequence(zope.schema._field.AbstractCollection):
...     _type = (list, tuple)
```

```
>>> numbers = MySequence(
...     value_type=zope.schema.Int(),
...     default=[],
...     missing_value=None,
...     )
```

```
>>> from z3c.form.browser import multi
>>> multiWidget = multi.MultiWidget(TestRequest())
>>> multiWidget.field = numbers
```

We now use the field and widget to instantiate the converter:

```
>>> conv = converter.MultiConverter(numbers, multiWidget)
```

We can now convert a list or tuple of integers to the multi-widget internal representation:

```
>>> conv.toWidgetValue([1, 2, 3, 4])
['1', '2', '3', '4']
```

```
>>> conv.toWidgetValue((1, 2, 3, 4))
['1', '2', '3', '4']
```

Now, let's look at the reverse. We get a tuple because that's the last type in `_type`:

```
>>> conv.toFieldValue(['1', '2', '3', '4'])
(1, 2, 3, 4)
```

3.3.14 Dict Multi Data Converter

For multi widgets and fields that work with a dictionary of other basic types, a separate data converter is required. Let's create a dict of integers field and a widget:

```
>>> numbers = zope.schema.Dict(
...     value_type=zope.schema.Int(),
...     key_type=zope.schema.Int(),
...     default={},
...     missing_value=None,
...     )
```

```
>>> from z3c.form.browser import multi
>>> multiWidget = multi.MultiWidget(TestRequest())
>>> multiWidget.field = numbers
```

Before we can convert, we have to regsiter a widget for the integer field:

```
>>> from z3c.form.browser import text
>>> zope.component.provideAdapter(
...     text.TextFieldWidget,
...     (zope.schema.Int, TestRequest))
```

We now use the field and widget to instantiate the converter:

```
>>> conv = converter.DictMultiConverter(numbers, multiWidget)
```

We can now convert a dict of integers to the multi-widget internal representation:

```
>>> sorted(conv.toWidgetValue({1:1, 2:4, 3:9}))
[('1', '1'), ('2', '4'), ('3', '9')]
```

If the value is the missing value, an empty dict is returned:

```
>>> conv.toWidgetValue(None)
[]
```

Now, let's look at the reverse:

```
>>> conv.toFieldValue([('1','1'), ('2','4'), ('3','9')])
{1: 1, 2: 4, 3: 9}
```

If the list is empty, the missing value is returned:

```
>>> conv.toFieldValue([]) is None
True
```

3.4 Terms

Terms are used to provide choices for sequence widgets or any other construct needing them. Since Zope 3 already has sources and vocabularies, the base terms class simply builds on them.

3.4.1 Vocabularies

Thus, let's create a vocabulary first:

```
>>> from zope.schema import vocabulary
>>> ratings = vocabulary.SimpleVocabulary([
...     vocabulary.SimpleVocabulary.createTerm(0, '0', 'bad'),
...     vocabulary.SimpleVocabulary.createTerm(1, '1', 'okay'),
...     vocabulary.SimpleVocabulary.createTerm(2, '2', 'good')
...     ])
```

Terms

Now we can create the terms object:

```
>>> from z3c.form import term
>>> terms = term.Terms()
>>> terms.terms = ratings
```

Getting a term from a given value is simple:

```
>>> terms.getTerm(0).title
'bad'
>>> terms.getTerm(3)
Traceback (most recent call last):
...
LookupError: 3
```

When converting values from their Web representation back to the internal representation, we have to be able to look up a term by its token:

```
>>> terms.getTermByToken('0').title
'bad'
>>> terms.getTerm('3')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
LookupError: 3
```

However, often we just want the value so asking for the value that is represented by a token saves usually one line of code:

```
>>> terms.getValue('0')
0
>>> terms.getValue('3')
Traceback (most recent call last):
...
LookupError: 3
```

You can also iterate through all terms:

```
>>> [entry.title for entry in terms]
['bad', 'okay', 'good']
```

Or ask how many terms you have in the first place:

```
>>> len(terms)
3
```

Finally the API allows you to check whether a particular value is available in the terms:

```
>>> 0 in terms
True
>>> 3 in terms
False
```

Now, there are several terms implementations that were designed for particular fields. Within the framework, terms are used as adapters with the follwoing discriminators: context, request, form, field, vocabulary/source and widget.

Choice field

The first terms implementation is for `Choice` fields. Choice fields unfortunately can have a vocabulary and a source which behave differently. Let's have a look a the vocabulary first:

```
>>> import zope.component
>>> zope.component.provideAdapter(term.ChoiceTermsVocabulary)
>>> import z3c.form.testing
>>> request = z3c.form.testing.TestRequest()
>>> import z3c.form.widget
>>> widget = z3c.form.widget.Widget(request)
```

```
>>> import zope.schema
```

```
>>> ratingField = zope.schema.Choice(
...     title='Rating',
...     vocabulary=ratings)
```

```
>>> terms = term.ChoiceTerms(
...     None, request, None, ratingField, widget)
>>> [entry.title for entry in terms]
['bad', 'okay', 'good']
```

Sometimes choice fields only specify a vocabulary name and the actual vocabulary is looked up at run time.

```
>>> ratingField2 = zope.schema.Choice(
...     title='Rating',
...     vocabulary='Ratings')
```

Initially we get an error because the "Ratings" vocabulary is not defined:

```
>>> terms = term.ChoiceTerms(
...     None, request, None, ratingField2, widget)
Traceback (most recent call last):
...
MissingVocabularyError: Can't validate value without vocabulary named 'Ratings'
```

Let's now register the vocabulary under this name:

```
>>> def RatingsVocabulary(obj):
...     return ratings
```

```
>>> from zope.schema import vocabulary
>>> vr = vocabulary.getVocabularyRegistry()
>>> vr.register('Ratings', RatingsVocabulary)
```

We should now be able to get all terms as before:

```
>>> terms = term.ChoiceTerms(
...     None, request, None, ratingField, widget)
>>> [entry.title for entry in terms]
['bad', 'okay', 'good']
```

Missing terms

Sometimes it happens that a term goes away from the vocabulary, but our stored objects still reference that term.

```
>>> zope.component.provideAdapter(term.MissingChoiceTermsVocabulary)
```

```
>>> terms = term.ChoiceTerms(
...     None, request, None, ratingField, widget)
>>> term = terms.getTermByToken('42')
Traceback (most recent call last):
...
LookupError: 42
```

The same goes with looking up a term by value:

```
>>> term = terms.getTerm('42')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
LookupError: 42
```

Ooops, well this works only if the context has the right value for us. This is because we don't want to accept any crap that's coming from HTML.

```
>>> class IPerson(zope.interface.Interface):
...     gender = zope.schema.Choice(title='Gender', vocabulary='Genders')
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...     gender = None
>>> gendersVocabulary = vocabulary.SimpleVocabulary([
...     vocabulary.SimpleVocabulary.createTerm(1, 'male', 'Male'),
...     vocabulary.SimpleVocabulary.createTerm(2, 'female', 'Female'),
...     ])
>>> def GendersVocabulary(obj):
...     return ratings
>>> vr.register('Genders', GendersVocabulary)
```

```
>>> ctx = Person()
>>> ctx.gender = 42
```

```
>>> genderWidget = z3c.form.widget.Widget(request)
>>> genderWidget.context = ctx
>>> from z3c.form import interfaces
>>> zope.interface.alsoProvides(genderWidget, interfaces.IContextAware)
>>> from z3c.form.datamanager import AttributeField
>>> zope.component.provideAdapter(AttributeField)
```

```
>>> terms = term.ChoiceTerms(
...     ctx, request, None, IPerson['gender'], genderWidget)
```

Here we go:

```
>>> missingTerm = terms.getTermByToken('42')
```

We get the term, we passed the token, the value is coming from the context.

```
>>> missingTerm.token
'42'
>>> missingTerm.value
42
```

We cannot figure the title, so we construct one. Override `makeMissingTerm` if you want your own.

```
>>> missingTerm.title
'Missing: ${value}'
```

Still we raise LookupError if the token does not fit the context's value:

```
>>> missingTerm = terms.getTermByToken('99')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
LookupError: 99
```

The same goes with looking up a term by value. We get the term if the context's value fits:

```
>>> missingTerm = terms.getTerm(42)
>>> missingTerm.token
'42'
```

And an exception if it does not:

```
>>> missingTerm = terms.getTerm(99)
Traceback (most recent call last):
...
LookupError: 99
```

Bool fields

A similar terms implementation exists for a `Bool` field:

```
>>> truthField = zope.schema.Bool()
```

```
>>> terms = term.BoolTerms(None, None, None, truthField, None)
>>> [entry.title for entry in terms]
['yes', 'no']
```

In case you don't like the choice of 'yes' and 'no' for the labels, we can subclass the `BoolTerms` class to control the display labels.

```
>>> class MyBoolTerms(term.BoolTerms):
...   trueLabel = 'True'
...   falseLabel = 'False'
```

```
>>> terms = MyBoolTerms(None, None, None, truthField, None)
>>> [entry.title for entry in terms]
['True', 'False']
```

Collections

Finally, there are a terms adapters for all collections. But we have to register some adapters before using it:

```
>>> from z3c.form import term
>>> zope.component.provideAdapter(term.CollectionTerms)
>>> zope.component.provideAdapter(term.CollectionTermsVocabulary)
>>> zope.component.provideAdapter(term.CollectionTermsSource)
```

```
>>> ratingsField = zope.schema.List(
...     title='Ratings',
...     value_type=ratingField)
```

```
>>> terms = term.CollectionTerms(
...     None, request, None, ratingsField, widget)
>>> [entry.title for entry in terms]
['bad', 'okay', 'good']
```

3.4.2 Sources

Basic sources

Basic sources need no context to compute their value. Let's create a source first:

```
>>> from zc.sourcefactory.basic import BasicSourceFactory
>>> class RatingSourceFactory(BasicSourceFactory):
...     _mapping = {10: 'ugly', 20: 'nice', 30: 'great'}
...     def getValues(self):
...         return self._mapping.keys()
...     def getTitle(self, value):
...         return self._mapping[value]
```

As we did not include the configure.zcml of zc.sourcefactory we have to register some required adapters manually. We also need the ChoiceTermsSource adapter:

```
>>> import zope.component
>>> import zc.sourcefactory.browser.source
>>> import zc.sourcefactory.browser.token
>>> zope.component.provideAdapter(
...     zc.sourcefactory.browser.source.FactoredTerms)
>>> zope.component.provideAdapter(
...     zc.sourcefactory.browser.token.fromInteger)
>>> zope.component.provideAdapter(term.ChoiceTermsSource)
```

Choice fields

Sources can be used with `Choice` fields like vocabularies. First we create a field based on the source:

```
>>> sourceRatingField = zope.schema.Choice(
...     title='Sourced Rating',
...     source=RatingSourceFactory())
```

We connect the field to a widget to see the ITerms adapter for sources at work:

```
>>> terms = term.ChoiceTerms(
...     None, request, None, sourceRatingField, widget)
```

Iterating over the terms adapter returnes the term objects:

```
>>> [entry for entry in terms]
[<zc.sourcefactory.browser.source.FactoredTerm object at 0x...>,
 <zc.sourcefactory.browser.source.FactoredTerm object at 0x...>,
 <zc.sourcefactory.browser.source.FactoredTerm object at 0x...>]
>>> len(terms)
```

(continues on next page)

(continued from previous page)

```
3
>>> [entry.token for entry in terms]
['10', '20', '30']
>>> [entry.title for entry in terms]
['ugly', 'nice', 'great']
```

Using a token it is possible to look up the term and the value:

```
>>> terms.getTermByToken('20').title
'nice'
>>> terms.getValue('30')
30
```

With can test if a value is in the source:

```
>>> 30 in terms
True
>>> 25 in terms
False
```

Missing terms

Sometimes it happens that a value goes away from the source, but our stored objects still has this value.

```
>>> zope.component.provideAdapter(term.MissingChoiceTermsSource)
```

```
>>> terms = term.ChoiceTerms(
...     None, request, None, sourceRatingField, widget)
>>> terms.getTermByToken('42')
Traceback (most recent call last):
...
LookupError: 42
```

The same goes with looking up a term by value:

```
>>> terms.getTerm(42)
Traceback (most recent call last):
...
LookupError: 42
```

Ooops, well this works only if the context has the right value for us. This is because we don't want to accept any crap that's coming from HTML.

```
>>> class IRating(zope.interface.Interface):
...     rating = zope.schema.Choice(title='Sourced Rating',
...                                 source=RatingSourceFactory())
>>> @zope.interface.implementer(IRating)
... class Rating(object):
...     rating = None
```

```
>>> ctx = Rating()
>>> ctx.rating = 42
```

```
>>> ratingWidget = z3c.form.widget.Widget(request)
>>> ratingWidget.context = ctx
>>> from z3c.form import interfaces
>>> zope.interface.alsoProvides(ratingWidget, interfaces.IContextAware)
>>> from z3c.form.datamanager import AttributeField
>>> zope.component.provideAdapter(AttributeField)
```

```
>>> terms = term.ChoiceTerms(
...     ctx, request, None, IRating['rating'], ratingWidget)
```

Here we go:

```
>>> missingTerm = terms.getTermByToken('42')
```

We get the term, we passed the token, the value is coming from the context.

```
>>> missingTerm.token
'42'
>>> missingTerm.value
42
```

We cannot figure the title, so we construct one. Override `makeMissingTerm` if you want your own.

```
>>> missingTerm.title
'Missing: ${value}'
```

Still we raise LookupError if the token does not fit the context's value:

```
>>> missingTerm = terms.getTermByToken('99')
Traceback (most recent call last):
...
LookupError: 99
```

The same goes with looking up a term by value. We get the term if the context's value fits:

```
>>> missingTerm = terms.getTerm(42)
>>> missingTerm.token
'42'
```

And an exception if it does not:

```
>>> missingTerm = terms.getTerm(99)
Traceback (most recent call last):
...
LookupError: 99
```

Collections

Finally, there are terms adapters for all collections:

```
>>> sourceRatingsField = zope.schema.List(
...     title='Sourced Ratings',
...     value_type=sourceRatingField)
```

```
>>> terms = term.CollectionTerms(
...     None, request, None, sourceRatingsField, widget)
>>> [entry.title for entry in terms]
['ugly', 'nice', 'great']
```

Contextual sources

Contextual sources depend on the context they are called on. Let's create a context and a contextual source:

```
>>> from zc.sourcefactory.contextual import BasicContextualSourceFactory
>>> class RatingContext(object):
...     base_value = 10
>>> class ContextualRatingSourceFactory(BasicContextualSourceFactory):
...     _mapping = {10: 'ugly', 20: 'nice', 30: 'great'}
...     def getValues(self, context):
...         return [context.base_value + x for x in self._mapping.keys()]
...     def getTitle(self, context, value):
...         return self._mapping[value - context.base_value]
```

As we did not include the configure.zcml of zc.sourcefactory we have to register some required adapters manually. We also need the ChoiceTermsSource adapter:

```
>>> import zope.component
>>> import zc.sourcefactory.browser.source
>>> import zc.sourcefactory.browser.token
>>> zope.component.provideAdapter(
...     zc.sourcefactory.browser.source.FactoredContextualTerms)
>>> zope.component.provideAdapter(
...     zc.sourcefactory.browser.token.fromInteger)
>>> zope.component.provideAdapter(term.ChoiceTermsSource)
```

Choice fields

Contextual sources can be used with `Choice` fields like vocabularies. First we create a field based on the source:

```
>>> contextualSourceRatingField = zope.schema.Choice(
...     title='Context Sourced Rating',
...     source=ContextualRatingSourceFactory())
```

We create an context object and connect the field to a widget to see the ITerms adapter for sources at work:

```
>>> rating_context = RatingContext()
>>> rating_context.base_value = 100
```

(continues on next page)

(continued from previous page)

```
>>> terms = term.ChoiceTerms(
...     rating_context, request, None, contextualSourceRatingField, widget)
```

Iterating over the terms adapter returnes the term objects:

```
>>> [entry for entry in terms]
[<zc.sourcefactory.browser.source.FactoredTerm object at 0x...>,
 <zc.sourcefactory.browser.source.FactoredTerm object at 0x...>,
 <zc.sourcefactory.browser.source.FactoredTerm object at 0x...>]
>>> len(terms)
3
>>> [entry.token for entry in terms]
['110', '120', '130']
>>> [entry.title for entry in terms]
['ugly', 'nice', 'great']
```

Using a token, it is possible to look up the term and the value:

```
>>> terms.getTermByToken('120').title
'nice'
>>> terms.getValue('130')
130
```

With can test if a value is in the source:

```
>>> 130 in terms
True
>>> 125 in terms
False
```

Collections

Finally, there are terms adapters for all collections:

```
>>> contextualSourceRatingsField = zope.schema.List(
...     title='Contextual Sourced Ratings',
...     value_type=contextualSourceRatingField)
```

```
>>> terms = term.CollectionTerms(
...     rating_context, request, None, contextualSourceRatingsField, widget)
>>> [entry.title for entry in terms]
['ugly', 'nice', 'great']
```

Missing terms in collections

Sometimes it happens that a value goes away from the source, but our stored collection still has this value.

```
>>> zope.component.provideAdapter(term.MissingCollectionTermsSource)
```

```
>>> terms = term.CollectionTerms(
...     RatingContext(), request, None, contextualSourceRatingsField, widget)
>>> terms
<z3c.form.term.MissingCollectionTermsSource object at 0x...>
>>> terms.getTermByToken('42')
Traceback (most recent call last):
...
LookupError: 42
```

The same goes with looking up a term by value:

```
>>> terms.getTerm(42)
Traceback (most recent call last):
...
LookupError: 42
```

The same goes with looking up a value by the token:

```
>>> terms.getValue('42')
Traceback (most recent call last):
...
LookupError: 42
```

Ooops, well this works only if the context has the right value for us. This is because we don't want to accept any crap that's coming from HTML.

```
>>> class IRatings(zope.interface.Interface):
...     ratings = zope.schema.List(
...         title='Contextual Sourced Ratings',
...         value_type=contextualSourceRatingField)
>>> @zope.interface.implementer(IRatings)
... class Ratings(object):
...     ratings = None
...     base_value = 10
```

```
>>> ctx = Ratings()
>>> ctx.ratings = [42, 10]
```

```
>>> ratingsWidget = z3c.form.widget.Widget(request)
>>> ratingsWidget.context = ctx
>>> from z3c.form import interfaces
>>> zope.interface.alsoProvides(ratingsWidget, interfaces.IContextAware)
>>> from z3c.form.datamanager import AttributeField
>>> zope.component.provideAdapter(AttributeField)
```

```
>>> terms = term.CollectionTerms(
...     ctx, request, None, IRatings['ratings'], ratingsWidget)
```

Here we go:

```
>>> term = terms.getTerm(42)
>>> missingTerm = terms.getTermByToken('42')
```

We get the term, we passed the token, the value is coming from the context.

```
>>> missingTerm.token
'42'
>>> missingTerm.value
42
```

We cannot figure the title, so we construct one. Override `makeMissingTerm` if you want your own.

```
>>> missingTerm.title
'Missing: ${value}'
```

We can get the value for a missing term:

```
>>> terms.getValue('42')
42
```

Still we raise LookupError if the token does not fit the context's value:

```
>>> missingTerm = terms.getTermByToken('99')
Traceback (most recent call last):
...
LookupError: 99
```

The same goes with looking up a term by value. We get the term if the context's value fits:

```
>>> missingTerm = terms.getTerm(42)
>>> missingTerm.token
'42'
```

And an exception if it does not:

```
>>> missingTerm = terms.getTerm(99)
Traceback (most recent call last):
...
LookupError: 99
```

3.5 Utility Functions and Classes

This file documents the utility functions and classes that are otherwise not tested.

```
>>> from z3c.form import util
```

3.5.1 createId(name) Function

This function converts an arbitrary unicode string into a valid Python identifier. If the name is a valid identifier, then it is just returned, but all upper case letters are lowered:

```
>>> util.createId('Change')
'change'
```

```
>>> util.createId('Change_2')
'change_2'
```

If a name is not a valid identifier, a hex code of the string is created:

```
>>> util.createId('Change 3')
'4368616e67652033'
```

The function can also handle non-ASCII characters:

```
>>> id = util.createId('Ändern')
```

Since the output depends on how Python is compiled (UCS-2 or 4), we only check that we have a valid id:

```
>>> util._identifier.match(id) is not None
True
```

3.5.2 createCSSId(name) Function

This function takes any unicode name and coverts it into an id that can be easily referenced by CSS selectors. Characters that are in the ascii alphabet, are numbers, or are '-' or '_' will be left the same. All other characters will be converted to ordinal numbers:

```
>>> util.createCSSId('NormalId')
'NormalId'
>>> id = util.createCSSId('')
>>> util._identifier.match(id) is not None
True
>>> util.createCSSId('This has spaces')
'This20has20spaces'
```

```
>>> util.createCSSId(str([(1, 'x'), ('foobar', 42)]))
'5b2812c2027x27292c202827foobar272c2042295d'
```

3.5.3 getWidgetById(form, id) Function

Given a form and a widget id, this function extracts the widget for you. First we need to create a properly developed form:

```
>>> import zope.interface
>>> import zope.schema
```

```
>>> class IPerson(zope.interface.Interface):
...     name = zope.schema.TextLine(title='Name')
```

```
>>> from z3c.form import form, field
>>> class AddPerson(form.AddForm):
...     fields = field.Fields(IPerson)
```

```
>>> from z3c.form import testing
>>> testing.setupFormDefaults()
```

```
>>> addPerson = AddPerson(None, testing.TestRequest())
>>> addPerson.update()
```

We can now ask for the widget:

```
>>> util.getWidgetById(addPerson, 'form-widgets-name')
<TextWidget 'form.widgets.name'>
```

The widget id can be split into a prefix and a widget name. The id must always start with the correct prefix, otherwise a value error is raised:

```
>>> util.getWidgetById(addPerson, 'myform-widgets-name')
Traceback (most recent call last):
...
ValueError: Name 'myform.widgets.name' must start with prefix 'form.widgets.'
```

If the widget is not found but the prefix is correct, None is returned:

```
>>> util.getWidgetById(addPerson, 'form-widgets-myname') is None
True
```

3.5.4 extractFileName(form, id, cleanup=True, allowEmptyPostfix=False) Function

Test the filename extraction method:

```
>>> class IDocument(zope.interface.Interface):
...     data = zope.schema.Bytes(title='Data')
```

Define a widgets stub and a upload widget stub class and setup them as a faked form:

```
>>> class FileUploadWidgetStub(object):
...     def __init__(self):
...         self.filename = None
```

```
>>> class WidgetsStub(object):
...     def __init__(self):
...         self.data = FileUploadWidgetStub()
...         self.prefix = 'widgets.'
...     def get(self, name, default):
...         return self.data
```

```
>>> class FileUploadFormStub(form.AddForm):
...     def __init__(self):
...         self.widgets = WidgetsStub()
...
...     def setFakeFileName(self, filename):
...         self.widgets.data.filename = filename
```

Now we can setup the stub form. Note this form is just a fake it's not a real implementation. We just provide a form like class which simulates the FileUpload object in the a widget. See *z3c/form/browser/file.rst* for a real file upload test uscase:

```
>>> uploadForm = FileUploadFormStub()
>>> uploadForm.setFakeFileName('foo.txt')
```

And extract the filename

```
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
'foo.txt'
```

Test a unicode filename:

```
>>> uploadForm.setFakeFileName('foo.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
'foo.txt'
```

Test a windows IE uploaded filename:

```
>>> uploadForm.setFakeFileName('D:\\some\\folder\\foo.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
'foo.txt'
```

Test another filename:

```
>>> uploadForm.setFakeFileName('D:/some/folder/foo.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
'foo.txt'
```

Test another filename:

```
>>> uploadForm.setFakeFileName('/tmp/folder/foo.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
'foo.txt'
```

Test special characters in filename, e.g. dots:

```
>>> uploadForm.setFakeFileName('/tmp/foo.bar.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
'foo.bar.txt'
```

Test some other special characters in filename:

```
>>> uploadForm.setFakeFileName('/tmp/foo-bar.v.0.1.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
'foo-bar.v.0.1.txt'
```

Test special characters in file path of filename:

```
>>> uploadForm.setFakeFileName('/tmp-v.1.0/foo-bar.v.0.1.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
'foo-bar.v.0.1.txt'
```

Test optional keyword arguments. But remember it's hard for Zope to guess the content type for filenames without extensions:

```
>>> uploadForm.setFakeFileName('minimal')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True,
...     allowEmptyPostfix=True)
'minimal'
```

```
>>> uploadForm.setFakeFileName('/tmp/minimal')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True,
...     allowEmptyPostfix=True)
'minimal'
```

```
>>> uploadForm.setFakeFileName('D:\\some\\folder\\minimal')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True,
...     allowEmptyPostfix=True)
'minimal'
```

There will be a ValueError if we get a empty filename by default:

```
>>> uploadForm.setFakeFileName('/tmp/minimal')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=True)
Traceback (most recent call last):
...
ValueError: Missing filename extension.
```

We also can skip removing a path from a upload. Note only IE will upload a path in a upload <input type="file" ...> field:

```
>>> uploadForm.setFakeFileName('/tmp/foo.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=False)
'/tmp/foo.txt'
```

```
>>> uploadForm.setFakeFileName('/tmp-v.1.0/foo-bar.v.0.1.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=False)
'/tmp-v.1.0/foo-bar.v.0.1.txt'
```

```
>>> uploadForm.setFakeFileName('D:\\some\\folder\\foo.txt')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=False)
'D:\\some\\folder\\foo.txt'
```

And missing filename extensions are also not allowed by deafault if we skip the filename:

```
>>> uploadForm.setFakeFileName('/tmp/minimal')
>>> util.extractFileName(uploadForm, 'form.widgets.data', cleanup=False)
Traceback (most recent call last):
...
ValueError: Missing filename extension.
```

3.5.5 extractContentType(form, id) Function

There is also a method which is able to extract the content type for a given file upload. We can use the stub form from the previous test.

Not sure if this an error but on my windows system this test returns image/pjpeg (progressive jpeg) for foo.jpg and image/x-png for foo.png. So let's allow this too since this depends on guess_content_type and is not really a part of z3c.form.

```
>>> uploadForm = FileUploadFormStub()
>>> uploadForm.setFakeFileName('foo.txt')
>>> util.extractContentType(uploadForm, 'form.widgets.data')
'text/plain'
```

```
>>> uploadForm.setFakeFileName('foo.gif')
>>> util.extractContentType(uploadForm, 'form.widgets.data')
'image/gif'
```

```
>>> uploadForm.setFakeFileName('foo.jpg')
>>> util.extractContentType(uploadForm, 'form.widgets.data')
'image/...jpeg'
```

```
>>> uploadForm.setFakeFileName('foo.png')
>>> util.extractContentType(uploadForm, 'form.widgets.data')
'image/...png'
```

```
>>> uploadForm.setFakeFileName('foo.tif')
>>> util.extractContentType(uploadForm, 'form.widgets.data')
'image/tiff'
```

```
>>> uploadForm.setFakeFileName('foo.doc')
>>> util.extractContentType(uploadForm, 'form.widgets.data')
'application/msword'
```

```
>>> uploadForm.setFakeFileName('foo.zip')
>>> (util.extractContentType(uploadForm, 'form.widgets.data')
...     in ('application/zip', 'application/x-zip-compressed'))
True
```

```
>>> uploadForm.setFakeFileName('foo.unknown')
>>> util.extractContentType(uploadForm, 'form.widgets.data')
'text/x-unknown-content-type'
```

3.5.6 *Manager* object

The manager object is a base class of a mapping object that keeps track of the key order as they are added.

```
>>> manager = util.Manager()
```

Initially the manager is empty:

```
>>> len(manager)
0
```

Since this base class mainly defines a read-interface, we have to add the values manually:

```
>>> manager['b'] = 2
>>> manager['a'] = 1
```

Let's iterate through the manager:

```
>>> tuple(iter(manager))
('b', 'a')
>>> list(manager.keys())
['b', 'a']
>>> list(manager.values())
[2, 1]
>>> list(manager.items())
[('b', 2), ('a', 1)]
```

Let's ow look at item access:

```
>>> 'b' in manager
True
>>> manager.get('b')
2
>>> manager.get('c', 'None')
'None'
```

It also supports deletion:

```
>>> del manager['b']
>>> list(manager.items())
[('a', 1)]
```

3.5.7 *SelectionManager* object

The selection manager is an extension to the manager and provides a few more API functions. Unfortunately, this base class is totally useless without a sensible constructor:

```
>>> import zope.interface
```

```
>>> class MySelectionManager(util.SelectionManager):
...     managerInterface = zope.interface.Interface
...
...     def __init__(self, *args):
```

(continues on next page)

(continued from previous page)

```
...         super(MySelectionManager, self).__init__()
...         args = list(args)
...         for arg in args:
...             if isinstance(arg, MySelectionManager):
...                 args += arg.values()
...                 continue
...             self[str(arg)] = arg
```

Let's now create two managers:

```
>>> manager1 = MySelectionManager(1, 2)
>>> manager2 = MySelectionManager(3, 4)
```

You can add two managers:

```
>>> manager = manager1 + manager2
>>> list(manager.values())
[1, 2, 3, 4]
```

Next, you can select only certain names:

```
>>> list(manager.select('1', '2', '3').values())
[1, 2, 3]
```

Or simply omit a value.

```
>>> list(manager.omit('2').values())
[1, 3, 4]
```

You can also easily copy a manager:

```
>>> manager.copy() is not manager
True
```

That's all.

3.5.8 *getSpecification()* function

This function is capable of returning an *ISpecification* for any object, including instances.

For an interface, it simply returns the interface:

```
>>> import zope.interface
>>> class IFoo(zope.interface.Interface):
...     pass
```

```
>>> util.getSpecification(IFoo) == IFoo
True
```

Ditto for a class:

```
>>> class Bar(object):
...     pass
```

```
>>> util.getSpecification(Bar) == Bar
True
```

For an instance, it will create a marker interface on the fly if necessary:

```
>>> bar = Bar()
>>> util.getSpecification(bar)
<InterfaceClass z3c.form.util.IGeneratedForObject_...>
```

The ellipsis represents a hash of the object.

If the function is called twice on the same object, it will not create a new marker each time:

```
>>> baz = Bar()
>>> barMarker = util.getSpecification(bar)
>>> bazMarker1 = util.getSpecification(baz)
>>> bazMarker2 = util.getSpecification(baz)
```

```
>>> barMarker is bazMarker1
False
```

```
>>> bazMarker1 == bazMarker2
True
>>> bazMarker1 is bazMarker2
True
```

3.5.9 *changedField()* function

Decide whether a field was changed/modified.

```
>>> class IPerson(zope.interface.Interface):
...     login = zope.schema.TextLine(
...         title='Login')
...     address = zope.schema.Object(
...         schema=zope.interface.Interface)
```

```
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...     login = 'johndoe'
>>> person = Person()
```

field.context is None and no context passed:

```
>>> util.changedField(IPerson['login'], 'foo')
True
```

IObject field:

```
>>> util.changedField(IPerson['address'], object(), context = person)
True
```

field.context or context passed:

```
>>> import z3c.form.datamanager
>>> zope.component.provideAdapter(z3c.form.datamanager.AttributeField)
```

```
>>> util.changedField(IPerson['login'], 'foo', context = person)
True
>>> util.changedField(IPerson['login'], 'johndoe', context = person)
False
```

```
>>> fld = IPerson['login'].bind(person)
>>> util.changedField(fld, 'foo')
True
>>> util.changedField(fld, 'johndoe')
False
```

No access:

```
>>> save = z3c.form.datamanager.AttributeField.canAccess
>>> z3c.form.datamanager.AttributeField.canAccess = lambda self: False
```

```
>>> util.changedField(IPerson['login'], 'foo', context = person)
True
>>> util.changedField(IPerson['login'], 'johndoe', context = person)
True
```

```
>>> z3c.form.datamanager.AttributeField.canAccess = save
```

3.5.10 *changedWidget()* function

Decide whether a widget value was changed/modified.

```
>>> import z3c.form.testing
>>> request = z3c.form.testing.TestRequest()
>>> import z3c.form.widget
>>> widget = z3c.form.widget.Widget(request)
```

If the widget is not IContextAware, there's nothing to check:

```
>>> from z3c.form import interfaces
>>> interfaces.IContextAware.providedBy(widget)
False
```

```
>>> util.changedWidget(widget, 'foo')
True
```

Make it IContextAware:

```
>>> widget.context = person
>>> zope.interface.alsoProvides(widget, interfaces.IContextAware)
```

```
>>> widget.field = IPerson['login']
```

>> util.changedWidget(widget, 'foo') True

```
>>> util.changedWidget(widget, 'johndoe')
False
```

Field and context is also overridable:

```
>>> widget.field = None
>>> util.changedWidget(widget, 'johndoe', field=IPerson['login'])
False
```

```
>>> p2 = Person()
>>> p2.login = 'foo'
```

```
>>> util.changedWidget(widget, 'foo', field=IPerson['login'], context=p2)
False
```

3.5.11 *sortedNone()* function

```
>>> util.sortedNone([None, 'a', 'b'])
[None, 'a', 'b']
```

```
>>> util.sortedNone([None, 1, 2])
[None, 1, 2]
```

```
>>> util.sortedNone([None, True, False])
[None, False, True]
```

```
>>> util.sortedNone([['true'], [], ['false']])
[[], ['false'], ['true']]
```

```
>>> util.sortedNone([('false',), ('true',), ()])
[(), ('false',), ('true',)]
```

3.6 Add Forms for `IAdding`

While using `IAdding`-based add forms is strongly discouraged by this package due to performance and code complexity concerns, there is still the need for add forms based on IAdding, especially when one wants to extend the default ZMI and use the add menu.

Before we get started, we need to register a bunch of form-related components:

```
>>> from z3c.form import testing
>>> testing.setupFormDefaults()
```

Let's first create a content component:

```
>>> import zope.interface
>>> import zope.schema
>>> class IPerson(zope.interface.Interface):
...
...     name = zope.schema.TextLine(
...         title=u'Name',
...         required=True)
```

```
>>> from zope.schema.fieldproperty import FieldProperty
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...     name = FieldProperty(IPerson['name'])
...
...     def __init__(self, name):
...         self.name = name
...
...     def __repr__(self):
...         return '<%s %r>' %(self.__class__.__name__, self.name)
```

Next we need a container to which we wish to add the person:

```
>>> from zope.container.btree import BTreeContainer
>>> people = BTreeContainer()
```

When creating and adding a new object using the `IAdding` API, the container is adapted to `IAdding` view:

```
>>> request = testing.TestRequest()
```

```
>>> from zope.app.container.browser.adding import Adding
>>> adding = Adding(people, request)
>>> adding
<zope.app.container.browser.adding.Adding object at ...>
```

To be able to create a person using `IAdding`, we need to create an add form for it now:

```
>>> import os
>>> from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile
>>> from z3c.form import tests, field
>>> from z3c.form.adding import AddForm
```

```
>>> class AddPersonForm(AddForm):
...     template = ViewPageTemplateFile(
...         'simple_edit.pt', os.path.dirname(tests.__file__))
...
...     fields = field.Fields(IPerson)
...
...     def create(self, data):
...         return Person(**data)
```

Besides the usual template and field declarations, one must also implement the `create()` method. Note that the `add()` and `nextURL()` methods are implemented for you already in comparison to the default add form. After instantiating the form, ...

```
>>> add = AddPersonForm(adding, request)
```

... we can now view the form:

```
>>> print(add())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-name">Name</label>
        <input type="text" id="form-widgets-name" name="form.widgets.name"
               class="text-widget required textline-field" value="" />
      </div>
      <div class="action">
        <input type="submit" id="form-buttons-add"
               name="form.buttons.add" class="submit-widget button-field"
               value="Add" />
      </div>
    </form>
  </body>
</html>
```

Once the form is filled out and the add button is clicked, ...

```
>>> request = testing.TestRequest(
...      form={'form.widgets.name': u'Stephan', 'form.buttons.add': 1})
```

```
>>> adding = Adding(people, request)
>>> add = AddPersonForm(adding, request)
>>> add.update()
```

... the person is added to the container:

```
>>> sorted(people.keys())
[u'Person']
>>> people['Person']
<Person u'Stephan'>
```

When the add form is rendered, nothing is returned and only the redirect header is set to the next URL. For this to work, we need to setup the location root correctly:

```
>>> from zope.traversing.interfaces import IContainmentRoot
>>> zope.interface.alsoProvides(people, IContainmentRoot)
```

```
>>> add.render()
''
```

```
>>> request.response.getHeader('Location')
'http://127.0.0.1/@@contents.html'
```

3.7 Testing support

3.7.1 Data Converter for Testing

Sometimes, we want to upload binary files. Particulary in Selenium tests, it is nearly impossible to correctly input binary data - so we allow the user to specify *base64* encoded data to be uploaded. This is accomplished by using a hidden input field that holds the value of the encoding desired.

```
>>> import zope.schema
>>> from z3c.form import widget
>>> from z3c.form import testing
```

As in converter.rst, we want to test a file upload widget.

```
>>> filedata = zope.schema.Text(
...     __name__='data',
...     title=u'Some data to upload',)
```

Lets try passing a simple string, and not specify any encoding.

```
>>> dataWidget = widget.Widget(testing.TestRequest(
...    form={'data.testing': 'haha'}))
>>> dataWidget.name = 'data'
```

```
>>> conv = testing.TestingFileUploadDataConverter(filedata, dataWidget)
>>> conv.toFieldValue('')
b'haha'
```

And now, specify a encoded string

```
>>> import base64
>>> encStr = base64.b64encode(b'hoohoo')
>>> dataWidget = widget.Widget(testing.TestRequest(
...    form={'data.testing': encStr, 'data.encoding': 'base64'}))
>>> dataWidget.name = 'data'
```

```
>>> conv = testing.TestingFileUploadDataConverter(filedata, dataWidget)
>>> conv.toFieldValue('')
b'hoohoo'
```

3.8 ObjectWidget caveat

ObjectWidget itself seems to be fine, but we discovered a fundamental problem in z3c.form.

The meat is that widget value * validation * extraction * applying values need to be separated and made recursive-aware.

3.8.1 Currently

- There is a loop that extracts and validates each widgets value. Then it moves on to the next widget in the same loop.
- The problem is that the ObjectWidget MUST keep it's values in the object itself, not in any dict or helper structure. That means in case of a validation failure later in the loop the ObjectWidget's values are already applied and cannot be reverted.
- Also on a single level of widgets this loop might be OK, because the loop is just flat.

3.8.2 We need

- To do a loop to validate ALL widget values.
- To do a loop to extract ALL values. (maybe apply too, let's think about it...)
- Then in a different loop apply those extracted (and validated) values.
- Problem is that the current API does not support separate methods for that.
- One more point is to take into account that with the ObjectWidget forms and widgets can be _recursive_, that means there can be a form-widgets-subform-widgets-subform-widgets level of widgets.

An example:

> The situation is the following: > - schema is like this: > class IMySubObject(zope.interface.Interface): > foofield = zope.schema.Int(> title=u"My foo field", > default=1111, > max=9999) > barfield = zope.schema.Int(> title=u"My dear bar", > default=2222, > required=False) > class IMyObject(zope.interface.Interface): > subobject = zope.schema.Object(title=u'my object', > schema=IMySubObject) > name = zope.schema.TextLine(title=u'name') > > - on object editing > - we need to keep the (**old**) (IMySubObject) object in place > – do not create a new one > - value setting is done in the editform handleApply > - extractData, extract needs to extract recursively > - return assignable values > - it has no idea about subobjects > - let's say the IMySubObject data is validated OK, but there's an > error in IMyObject (with name) > - now the problem is: > - IMyObject.subobject extract gets called first > it sets the values on the existing object (and fires > ObjectModifiedEvent) > - IMyObject.name detects the error > it does not set the value > BUT IMyObject.subobject sticks to the extracted value that should be > discarded, because the whole form did not validate?!?!?!

CHAPTER

FOUR

WIDGETS

Each documentation file comprehensively explains the widget and how it is ensured to work properly.

4.1 Checkbox Widget

Note: the checkbox widget isn't registered for a field by default. You can use the `widgetFactory` argument of a `IField` object if you construct fields or set the custom widget factory on selected fields later.

The `CheckBoxWidget` widget renders a checkbox input type field e.g. <input type="checkbox" />

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import checkbox
```

The `CheckboxWidget` is a widget:

```
>>> verifyClass(interfaces.IWidget, checkbox.CheckBoxWidget)
True
```

The widget can render a input field only by adapting a request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = checkbox.CheckBoxWidget(request)
```

Set a name and id for the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

Such a field provides IWidget:

```
>>> interfaces.IWidget.providedBy(widget)
True
```

We also need to register the template for at least the widget and request:

```
>>> import os.path
>>> import zope.interface
>>> from zope.publisher.interfaces.browser import IDefaultBrowserLayer
>>> from zope.pagetemplate.interfaces import IPageTemplate
```

(continues on next page)

(continued from previous page)

```
>>> import z3c.form.browser
>>> import z3c.form.widget
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'checkbox_input.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='input')
```

If we render the widget we only get the empty marker:

```
>>> print(widget.render())
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

Let's provide some values for this widget. We can do this by defining a vocabulary providing `ITerms`. This vocabulary uses descriminators wich will fit for our setup.

```
>>> import zope.schema.interfaces
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> import z3c.form.term
>>> class MyTerms(z3c.form.term.ChoiceTermsVocabulary):
...     def __init__(self, context, request, form, field, widget):
...         self.terms = SimpleVocabulary.fromValues(['yes', 'no'])
>>> zope.component.provideAdapter(z3c.form.term.BoolTerms,
...     adapts=(zope.interface.Interface,
...             interfaces.IFormLayer, zope.interface.Interface,
...             zope.interface.Interface, interfaces.ICheckBoxWidget))
```

Now let's try if we get widget values:

```
>>> widget.update()
>>> print(widget.render())
<span id="widget-id">
 <span class="option">
  <input type="checkbox" id="widget-id-0" name="widget.name:list"
         class="checkbox-widget" value="true" />
  <label for="widget-id-0">
    <span class="label">yes</span>
  </label>
    </span><span class="option">
  <input type="checkbox" id="widget-id-1" name="widget.name:list"
         class="checkbox-widget" value="false" />
  <label for="widget-id-1">
    <span class="label">no</span>
  </label>
 </span>
</span>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

The checkbox json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
```

(continues on next page)

(continued from previous page)

```
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'options': [{'checked': False,
              'id': 'widget-id-0',
              'label': 'yes',
              'name': 'widget.name:list',
              'value': 'true'},
             {'checked': False,
              'id': 'widget-id-1',
              'label': 'no',
              'name': 'widget.name:list',
              'value': 'false'}],
 'required': False,
 'type': 'check',
 'value': ()}
```

If we set the value for the widget to `yes`, we can se that the checkbox field get rendered with a checked flag:

```
>>> widget.value = 'true'
>>> widget.update()
>>> print(widget.render())
<span id="widget-id">
 <span class="option">
  <input type="checkbox" id="widget-id-0" name="widget.name:list"
         class="checkbox-widget" value="true"
         checked="checked" />
  <label for="widget-id-0">
    <span class="label">yes</span>
  </label>
</span><span class="option">
  <input type="checkbox" id="widget-id-1" name="widget.name:list"
         class="checkbox-widget" value="false" />
  <label for="widget-id-1">
    <span class="label">no</span>
  </label>
 </span>
</span>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

The checkbox json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'options': [{'checked': True,
```

(continues on next page)

(continued from previous page)

```
              'id': 'widget-id-0',
              'label': 'yes',
              'name': 'widget.name:list',
              'value': 'true'},
             {'checked': False,
              'id': 'widget-id-1',
              'label': 'no',
              'name': 'widget.name:list',
              'value': 'false'}],
 'required': False,
 'type': 'check',
 'value': 'true'}
```

Check HIDDEN_MODE:

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'checkbox_hidden.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='hidden')
```

```
>>> widget.value = 'true'
>>> widget.mode = interfaces.HIDDEN_MODE
>>> print(widget.render())
<span class="option">
  <input type="hidden" id="widget-id-0" name="widget.name:list"
         class="checkbox-widget" value="true" />
</span>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

The checkbox json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'hidden',
 'name': 'widget.name',
 'options': [{'checked': True,
              'id': 'widget-id-0',
              'label': 'yes',
              'name': 'widget.name:list',
              'value': 'true'},
             {'checked': False,
              'id': 'widget-id-1',
              'label': 'no',
              'name': 'widget.name:list',
              'value': 'false'}],
 'required': False,
 'type': 'check',
```

(continues on next page)

(continued from previous page)

```
 'value': 'true'}
```

Make sure that we produce a proper label when we have no title for a term and the value (which is used as a backup label) contains non-ASCII characters:

```
>>> terms = SimpleVocabulary.fromValues([b'yes\012', b'no\243'])
>>> widget.terms = terms
>>> widget.update()
>>> pprint(list(widget.items))
[{'checked': False,
  'id': 'widget-id-0',
  'label': 'yes\n',
  'name': 'widget.name:list',
  'value': 'yes\n'},
 {'checked': False,
  'id': 'widget-id-1',
  'label': 'no',
  'name': 'widget.name:list',
  'value': 'no...'}]
```

Note: The "234" character is interpreted differently in Pytohn 2 and 3 here. (This is mostly due to changes int he SimpleVocabulary code.)

4.1.1 Single Checkbox Widget

Instead of using the checkbox widget as an UI component to allow multiple selection from a list of choices, it can be also used by itself to toggle a selection, effectively making it a binary selector. So in this case it lends itself well as a boolean UI input component.

The `SingleCheckboxWidget` is a widget:

```
>>> verifyClass(interfaces.IWidget, checkbox.SingleCheckBoxWidget)
True
```

The widget can render a input field only by adapting a request:

```
>>> request = TestRequest()
>>> widget = checkbox.SingleCheckBoxWidget(request)
```

Set a name and id for the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

Such a widget provides the `IWidget` interface:

```
>>> interfaces.IWidget.providedBy(widget)
True
```

For there to be a sensible output, we need to give the widget a label:

```
>>> widget.label = u'Do you want that?'
```

```
>>> widget.update()
>>> print(widget.render())
<span class="option" id="widget-id">
  <input type="checkbox" id="widget-id-0"
         name="widget.name:list"
         class="single-checkbox-widget" value="selected" />
  <label for="widget-id-0">
    <span class="label">Do you want that?</span>
  </label>
</span>
<input name="widget.name-empty-marker" type="hidden"
       value="1" />
```

The checkbox json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': 'Do you want that?',
 'mode': 'input',
 'name': 'widget.name',
 'options': [{'checked': False,
              'id': 'widget-id-0',
              'label': 'Do you want that?',
              'name': 'widget.name:list',
              'value': 'selected'}],
 'required': False,
 'type': 'check',
 'value': ()}
```

Initially, the box is not checked. Changing the widget value to the selection value, ...

```
>>> widget.value = ['selected']
```

will make the box checked:

```
>>> widget.update()
>>> print(widget.render())
<span class="option" id="widget-id">
  <input type="checkbox" id="widget-id-0"
         name="widget.name:list"
         class="single-checkbox-widget" value="selected"
         checked="checked" />
  <label for="widget-id-0">
    <span class="label">Do you want that?</span>
  </label>
</span>
<input name="widget.name-empty-marker" type="hidden"
       value="1" />
```

If you do not specify the label on the widget directly, it is taken from the field

```
>>> from zope.schema import Bool
>>> widget = checkbox.SingleCheckBoxWidget(request)
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
>>> widget.field = Bool(title=u"Do you REALLY want that?")
>>> widget.update()
>>> print(widget.render())
<span class="option" id="widget-id">
  <input type="checkbox" id="widget-id-0"
         name="widget.name:list"
         class="single-checkbox-widget" value="selected" />
  <label for="widget-id-0">
    <span class="label">Do you REALLY want that?</span>
  </label>
</span>
<input name="widget.name-empty-marker" type="hidden"
       value="1" />
```

Check HIDDEN_MODE:

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'checkbox_hidden.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='hidden')
```

```
>>> widget.value = 'selected'
>>> widget.mode = interfaces.HIDDEN_MODE
>>> print(widget.render())
<span class="option">
  <input type="hidden" id="widget-id-0"
         name="widget.name:list"
         class="single-checkbox-widget" value="selected" />
</span>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

4.1.2 Term with non ascii __str__

Check if a term which __str__ returns non ascii string does not crash the update method

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import checkbox
>>> from z3c.form.testing import TestRequest
```

```
>>> request = TestRequest()
```

```
>>> widget = checkbox.CheckBoxWidget(request)
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

```
>>> import zope.schema.interfaces
>>> from zope.schema.vocabulary import SimpleVocabulary,SimpleTerm
>>> import z3c.form.term
>>> class ObjWithNonAscii__str__:
...     def __str__(self):
...         return 'héhé!'
>>> class MyTerms(z3c.form.term.ChoiceTermsVocabulary):
...     def __init__(self, context, request, form, field, widget):
...         self.terms = SimpleVocabulary([
...             SimpleTerm(ObjWithNonAscii__str__(), 'one', 'One'),
...             SimpleTerm(ObjWithNonAscii__str__(), 'two', 'Two'),
...         ])
>>> zope.component.provideAdapter(MyTerms,
...     adapts=(zope.interface.Interface,
...             interfaces.IFormLayer, zope.interface.Interface,
...             zope.interface.Interface, interfaces.ICheckBoxWidget))
>>> widget.update()
>>> print(widget.render())
<html>
  <body>
    <span id="widget-id">
     <span class="option">
      <input class="checkbox-widget" id="widget-id-0" name="widget.name:list" type=
↪"checkbox" value="one">
      <label for="widget-id-0">
        <span class="label">One</span>
      </label>
    </span>
    <span class="option">
      <input class="checkbox-widget" id="widget-id-1" name="widget.name:list" type=
↪"checkbox" value="two">
      <label for="widget-id-1">
        <span class="label">Two</span>
      </label>
     </span>
    </span>
    <input name="widget.name-empty-marker" type="hidden" value="1">
  </body>
</html>
```

4.2 Radio Widget

The RadioWidget renders a radio input type field e.g. <input type="radio" />

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import radio
```

The RadioWidget is a widget:

```
>>> verifyClass(interfaces.IWidget, radio.RadioWidget)
 True
```

The widget can render a input field only by adapting a request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = radio.RadioWidget(request)
```

Set a name and id for the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

Such a field provides IWidget:

```
>>> interfaces.IWidget.providedBy(widget)
 True
```

We also need to register the template for at least the widget and request:

```
>>> import os.path
>>> import zope.interface
>>> from zope.publisher.interfaces.browser import IDefaultBrowserLayer
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> import z3c.form.browser
>>> import z3c.form.widget
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'radio_input.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='input')
>>> template_single = os.path.join(
...     os.path.dirname(z3c.form.browser.__file__),
...     'radio_input_single.pt')
>>> zope.component.provideAdapter(
...     z3c.form.widget.WidgetTemplateFactory(template_single),
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='input_single')
```

If we render the widget we only get the empty marker:

```
>>> print(widget.render())
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

Let's provide some values for this widget. We can do this by defining a source providing ITerms. This source uses discriminators which will fit for our setup.

```
>>> import zope.schema.interfaces
>>> import z3c.form.term
>>> from zc.sourcefactory.basic import BasicSourceFactory
>>> class YesNoSourceFactory(BasicSourceFactory):
...     def getValues(self):
```

(continues on next page)

(continued from previous page)

```
...         return ['yes', 'no']
>>> class MyTerms(z3c.form.term.ChoiceTermsSource):
...     def __init__(self, context, request, form, field, widget):
...         self.terms = YesNoSourceFactory()
>>> zope.component.provideAdapter(z3c.form.term.BoolTerms,
...     adapts=(zope.interface.Interface,
...             interfaces.IFormLayer, zope.interface.Interface,
...             zope.interface.Interface, interfaces.IRadioWidget))
```

Now let's try if we get widget values:

```
>>> widget.update()
>>> print(widget.render())
<span class="option">
  <label for="widget-id-0">
    <input type="radio" id="widget-id-0" name="widget.name"
           class="radio-widget" value="true" />
    <span class="label">yes</span>
  </label>
</span><span class="option">
  <label for="widget-id-1">
    <input type="radio" id="widget-id-1" name="widget.name"
           class="radio-widget" value="false" />
    <span class="label">no</span>
  </label>
</span>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

The radio json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'options': [{'checked': False,
              'id': 'widget-id-0',
              'label': 'yes',
              'name': 'widget.name',
              'value': 'true'},
             {'checked': False,
              'id': 'widget-id-1',
              'label': 'no',
              'name': 'widget.name',
              'value': 'false'}],
 'required': False,
 'type': 'radio',
 'value': ()}
```

If we set the value for the widget to `yes`, we can se that the radio field get rendered with a checked flag:

```
>>> widget.value = 'true'
>>> widget.update()
>>> print(widget.render())
<span class="option">
  <label for="widget-id-0">
    <input type="radio" id="widget-id-0" name="widget.name"
           class="radio-widget" value="true" checked="checked" />
    <span class="label">yes</span>
  </label>
</span><span class="option">
  <label for="widget-id-1">
    <input type="radio" id="widget-id-1" name="widget.name"
           class="radio-widget" value="false" />
    <span class="label">no</span>
  </label>
</span>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

The radio json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'options': [{'checked': True,
              'id': 'widget-id-0',
              'label': 'yes',
              'name': 'widget.name',
              'value': 'true'},
             {'checked': False,
              'id': 'widget-id-1',
              'label': 'no',
              'name': 'widget.name',
              'value': 'false'}],
 'required': False,
 'type': 'radio',
 'value': 'true'}
```

We can also render the input elements for each value separately:

```
>>> print(widget.renderForValue('true'))
<input id="widget-id-0" name="widget.name" class="radio-widget"
       value="true" checked="checked" type="radio" />
```

```
>>> print(widget.renderForValue('false'))
<input id="widget-id-1" name="widget.name" class="radio-widget"
       value="false" type="radio" />
```

Additionally we can render the “no value” input element used for non-required fields:

```
>>> from z3c.form.widget import SequenceWidget
>>> print(SequenceWidget.noValueToken)
--NOVALUE--
>>> print(widget.renderForValue(SequenceWidget.noValueToken))
<input id="widget-id-novalue" name="widget.name" class="radio-widget"
        value="--NOVALUE--" type="radio" />
```

Check HIDDEN_MODE:

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'radio_hidden.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='hidden')
```

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'radio_hidden_single.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='hidden_single')
```

```
>>> widget.value = ['true']
>>> widget.mode = interfaces.HIDDEN_MODE
>>> print(widget.render())
<input id="widget-id-0" name="widget.name" value="true"
       class="hidden-widget" type="hidden" />
```

And independently:

```
>>> print(widget.renderForValue('true'))
<input id="widget-id-0" name="widget.name" value="true"
       class="hidden-widget" type="hidden" />
```

The unchecked values do not need a hidden field, hence they are empty:

```
>>> print(widget.renderForValue('false'))
```

Check DISPLAY_MODE:

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'radio_display.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='display')
```

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'radio_display_single.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
```

(continues on next page)

(continued from previous page)

```
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='display_single')
```

```
>>> widget.value = ['true']
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="widget-id" class="radio-widget">
  <span class="selected-option">yes</span>
</span>
```

And independently:

```
>>> print(widget.renderForValue('true'))
<span id="widget-id" class="radio-widget"><span class="selected-option">yes</span></span>
```

Again, unchecked values are not displayed:

```
>>> print(widget.renderForValue('false'))
```

Make sure that we produce a proper label when we have no title for a term and the value (which is used as a backup label) contains non-ASCII characters:

```
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> terms = SimpleVocabulary.fromValues([b'yes\012', b'no\243'])
>>> widget.terms = terms
>>> widget.update()
>>> pprint(list(widget.items))
[{'checked': False,
  'id': 'widget-id-0',
  'label': 'yes\n',
  'name': 'widget.name',
  'value': 'yes\n'},
 {'checked': False,
  'id': 'widget-id-1',
  'label': 'no',
  'name': 'widget.name',
  'value': 'no...'}]
```

Note: The “234” character is interpreted differently in Pytohn 2 and 3 here. (This is mostly due to changes int he SimpleVocabulary code.)

4.2.1 Term with non ascii __str__

Check if a term which __str__ returns non ascii string does not crash the update method

```
>>> request = TestRequest()
>>> widget = radio.RadioWidget(request)
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'radio_input.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
```

(continues on next page)

(continued from previous page)

```
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='input')
```

```
>>> import zope.schema.interfaces
>>> from zope.schema.vocabulary import SimpleVocabulary,SimpleTerm
>>> import z3c.form.term
>>> class ObjWithNonAscii__str__:
...     def __str__(self):
...         return 'héhé!'
>>> class MyTerms(z3c.form.term.ChoiceTermsVocabulary):
...     def __init__(self, context, request, form, field, widget):
...         self.terms = SimpleVocabulary([
...             SimpleTerm(ObjWithNonAscii__str__(), 'one', 'One'),
...             SimpleTerm(ObjWithNonAscii__str__(), 'two', 'Two'),
...         ])
>>> zope.component.provideAdapter(MyTerms,
...     adapts=(zope.interface.Interface,
...             interfaces.IFormLayer, zope.interface.Interface,
...             zope.interface.Interface, interfaces.IRadioWidget))
>>> widget.update()
>>> print(widget.render())
<html>
  <body>
    <span class="option">
      <label for="widget-id-0">
        <input class="radio-widget" id="widget-id-0" name="widget.name" type="radio"␣
↪value="one">
        <span class="label">One</span>
      </label>
    </span>
    <span class="option">
      <label for="widget-id-1">
        <input class="radio-widget" id="widget-id-1" name="widget.name" type="radio"␣
↪value="two">
        <span class="label">Two</span>
      </label>
    </span>
    <input name="widget.name-empty-marker" type="hidden" value="1">
  </body>
</html>
```

4.3 Text Widget

The widget can render a input field for a text line:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import text
```

The TextWidget is a widget:

```
>>> verifyClass(interfaces.IWidget, text.TextWidget)
 True
```

The widget can render a input field only by adapting a request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = text.TextWidget(request)
```

Such a field provides IWidget:

```
>>> interfaces.IWidget.providedBy(widget)
 True
```

We also need to register the template for at least the widget and request:

```
>>> import os.path
>>> import zope.interface
>>> from zope.publisher.interfaces.browser import IDefaultBrowserLayer
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> import z3c.form.browser
>>> import z3c.form.widget
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'text_input.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='input')
```

If we render the widget we get the HTML:

```
>>> print(widget.render())
<input type="text" class="text-widget" value="" />
```

Adding some more attributes to the widget will make it display more:

```
>>> widget.id = 'id'
>>> widget.name = 'name'
>>> widget.value = u'value'
>>> widget.style = u'color: blue'
>>> widget.placeholder = u'Email address'
>>> widget.autocapitalize = u'off'
```

```
>>> print(widget.render())
<input type="text" id="id" name="name" class="text-widget"
       placeholder="Email address" autocapitalize="off"
       style="color: blue" value="value" />
```

Check DISPLAY_MODE:

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'text_display.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='display')
```

```
>>> widget.value = u'foobar'
>>> widget.style = None
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="id" class="text-widget">foobar</span>
```

Check HIDDEN_MODE:

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'text_hidden.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='hidden')
```

```
>>> widget.value = u'foobar'
>>> widget.mode = interfaces.HIDDEN_MODE
>>> print(widget.render())
<input id="id" name="name" value="foobar" class="hidden-widget" type="hidden" />
```

4.4 TextArea Widget

The widget can render a text area field for a text:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import textarea
```

The `TextAreaWidget` is a widget:

```
>>> verifyClass(interfaces.IWidget, textarea.TextAreaWidget)
True
```

The widget can render a input field only by adapting a request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = textarea.TextAreaWidget(request)
```

Such a field provides IWidget:

```
>>> interfaces.IWidget.providedBy(widget)
True
```

We also need to register the template for at least the widget and request:

```
>>> import os.path
>>> import zope.interface
>>> from zope.publisher.interfaces.browser import IDefaultBrowserLayer
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> import z3c.form.browser
>>> import z3c.form.widget
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'textarea_input.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='input')
```

If we render the widget we get the HTML:

```
>>> print(widget.render())
<textarea class="textarea-widget"></textarea>
```

Adding some more attributes to the widget will make it display more:

```
>>> widget.id = 'id'
>>> widget.name = 'name'
>>> widget.value = u'value'
```

```
>>> print(widget.render())
<textarea id="id" name="name" class="textarea-widget">value</textarea>
```

The json data representing the textarea widget:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'id',
 'label': '',
 'mode': 'input',
 'name': 'name',
 'required': False,
 'type': 'textarea',
 'value': 'value'}
```

Check DISPLAY_MODE:

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'textarea_display.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='display')
```

```
>>> widget.value = u'foobar'
>>> widget.mode = interfaces.DISPLAY_MODE
>>> print(widget.render())
<span id="id" class="textarea-widget">foobar</span>
```

Check HIDDEN_MODE:

```
>>> template = os.path.join(os.path.dirname(z3c.form.browser.__file__),
...     'textarea_hidden.pt')
>>> factory = z3c.form.widget.WidgetTemplateFactory(template)
>>> zope.component.provideAdapter(factory,
...     (zope.interface.Interface, IDefaultBrowserLayer, None, None, None),
...     IPageTemplate, name='hidden')
```

```
>>> widget.value = u'foobar'
>>> widget.mode = interfaces.HIDDEN_MODE
>>> print(widget.render())
<input class="hidden-widget" id="id" name="name"
       type="hidden" value="foobar">
```

4.5 TextLines Widget

The text lines widget allows you to store a sequence of textline. This sequence is stored as a list or tuple. This depends on what you are using as sequence type.

As for all widgets, the text lines widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import textlines
```

```
>>> verifyClass(interfaces.IWidget, textlines.TextLinesWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = textlines.TextLinesWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'id'
>>> widget.name = 'name'
```

We also need to register the template for at least the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('textlines_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ITextLinesWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get an empty textarea widget:

```
>>> print(widget.render())
<textarea id="id" name="name" class="textarea-widget"></textarea>
```

Adding some more attributes to the widget will make it display more:

```
>>> widget.id = 'id'
>>> widget.name = 'name'
>>> widget.value = u'foo\nbar'
```

```
>>> print(widget.render())
<textarea id="id" name="name" class="textarea-widget">foo
bar</textarea>
```

4.5.1 TextLinesFieldWidget

The field widget needs a field:

```
>>> import zope.schema
>>> text = zope.schema.List(
...     title=u'List',
...      value_type=zope.schema.TextLine())
```

```
>>> widget = textlines.TextLinesFieldWidget(text, request)
>>> widget
<TextLinesWidget ''>
```

```
>>> widget.id = 'id'
>>> widget.name = 'name'
>>> widget.value = u'foo\nbar'
```

```
>>> print(widget.render())
<textarea id="id" name="name" class="textarea-widget">foo
bar</textarea>
```

4.5.2 TextLinesFieldWidgetFactory

```
>>> widget = textlines.TextLinesFieldWidgetFactory(text, text.value_type,
...     request)
>>> widget
<TextLinesWidget ''>
```

```
>>> widget.id = 'id'
>>> widget.name = 'name'
>>> widget.value = u'foo\nbar'
```

```
>>> print(widget.render())
<textarea id="id" name="name" class="textarea-widget">foo
bar</textarea>
```

4.6 Password Widget

The password widget allows you to upload a new password to the server. The "password" type of the "INPUT" element is described here:

http://www.w3.org/TR/1999/REC-html401-19991224/interact/forms.html#edef-INPUT

As for all widgets, the password widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import password
```

```
>>> verifyClass(interfaces.IWidget, password.PasswordWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = password.PasswordWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget.id'
>>> widget.name = 'widget.name'
```

We also need to register the template for the widget:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('password_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IPasswordWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get a simple input element:

```
>>> print(widget.render())
<input type="password" id="widget.id" name="widget.name"
       class="password-widget" />
```

Even when we set a value on the widget, it is not displayed for security reasons:

```
>>> widget.value = 'password'
>>> print(widget.render())
<input type="password" id="widget.id" name="widget.name"
       class="password-widget" />
```

Adding some more attributes to the widget will make it display more:

```
>>> widget.style = u'color: blue'
>>> widget.placeholder = u'Confirm password'
>>> widget.autocapitalize = u'off'
```

```
>>> print(widget.render())
<input type="password" id="widget.id" name="widget.name"
       placeholder="Confirm password" autocapitalize="off"
       style="color: blue" class="password-widget" />
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> widget.request = TestRequest(form={'widget.name': 'password'})
>>> widget.update()
>>> widget.extract()
'password'
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = TestRequest()
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

4.7 Select Widget

The select widget allows you to select one or more values from a set of given options.

4.7.1 Select Widget

The select widget allows you to select one or more values from a set of given options. The "SELECT" and "OPTION" elements are described here:

http://www.w3.org/TR/1999/REC-html401-19991224/interact/forms.html#edef-SELECT

As for all widgets, the select widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import select
```

```
>>> verifyClass(interfaces.IWidget, select.SelectWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = select.SelectWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

We also need to register the template for at least the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('select_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISelectWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get an empty widget:

```
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget" size="1">
</select>
<input name="widget.name-empty-marker" type="hidden"
       value="1" />
```

Let's provide some values for this widget. We can do this by defining a source providing `ITerms`. This source uses descriminators which will fit our setup.

```
>>> import zope.schema.interfaces
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> import z3c.form.term
```

```
>>> class SelectionTerms(z3c.form.term.Terms):
...     def __init__(self, context, request, form, field, widget):
...         self.terms = SimpleVocabulary.fromValues(['a', 'b', 'c'])
```

```
>>> zope.component.provideAdapter(SelectionTerms,
...     (None, interfaces.IFormLayer, None, None, interfaces.ISelectWidget) )
```

Now let's try if we get widget values:

```
>>> widget.update()
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget" size="1">
<option id="widget-id-novalue" selected="selected" value="--NOVALUE--">No value</option>
<option id="widget-id-0" value="a">a</option>
<option id="widget-id-1" value="b">b</option>
<option id="widget-id-2" value="c">c</option>
</select>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

Select json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'options': [{'content': 'No value',
              'id': 'widget-id-novalue',
              'selected': True,
              'value': '--NOVALUE--'},
             {'content': 'a',
              'id': 'widget-id-0',
              'selected': False,
              'value': 'a'},
             {'content': 'b',
              'id': 'widget-id-1',
              'selected': False,
              'value': 'b'},
             {'content': 'c',
              'id': 'widget-id-2',
              'selected': False,
              'value': 'c'}],
 'required': False,
 'type': 'select',
 'value': ()}
```

If we select item "b", then it should be selected:

```
>>> widget.value = ['b']
>>> widget.update()
```

(continues on next page)

(continued from previous page)

```
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget" size="1">
<option id="widget-id-novalue" value="--NOVALUE--">No value</option>
<option id="widget-id-0" value="a">a</option>
<option id="widget-id-1" value="b" selected="selected">b</option>
<option id="widget-id-2" value="c">c</option>
</select>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

Select json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'options': [{'content': 'No value',
              'id': 'widget-id-novalue',
              'selected': False,
              'value': '--NOVALUE--'},
             {'content': 'a',
              'id': 'widget-id-0',
              'selected': False,
              'value': 'a'},
             {'content': 'b',
              'id': 'widget-id-1',
              'selected': True,
              'value': 'b'},
             {'content': 'c',
              'id': 'widget-id-2',
              'selected': False,
              'value': 'c'}],
 'required': False,
 'type': 'select',
 'value': ['b']}
```

Let's see what happens if we have values that are not in the vocabulary:

```
>>> widget.value = ['x', 'y']
>>> widget.update()
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget" size="1">
<option id="widget-id-novalue" value="--NOVALUE--">No value</option>
<option id="widget-id-0" value="a">a</option>
<option id="widget-id-1" value="b">b</option>
<option id="widget-id-2" value="c">c</option>
</select>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> widget.request = TestRequest(form={'widget.name': ['c']})
>>> widget.update()
>>> widget.extract()
('c',)
```

When "No value" is selected, then no verification against the terms is done:

```
>>> widget.request = TestRequest(form={'widget.name': ['--NOVALUE--']})
>>> widget.update()
>>> widget.extract(default=1)
('--NOVALUE--',)
```

Unfortunately, when nothing is selected, we do not get an empty list sent into the request, but simply no entry at all. For this we have the empty marker, so that:

```
>>> widget.request = TestRequest(form={'widget.name-empty-marker': '1'})
>>> widget.update()
>>> widget.extract()
()
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = TestRequest()
>>> widget.update()
>>> widget.extract(default=1)
1
```

Let's now make sure that a bogus value causes extract to return the default as described by the interface:

```
>>> widget.request = TestRequest(form={'widget.name': ['x']})
>>> widget.update()
>>> widget.extract(default=1)
1
```

Custom No Value Messages

Additionally to the standard dynamic attribute values, the select widget also allows dynamic values for the "No value message". Initially, we have the default message:

```
>>> widget.noValueMessage
'No value'
```

Let's now register an attribute value:

```
>>> from z3c.form.widget import StaticWidgetAttribute
>>> NoValueMessage = StaticWidgetAttribute('- nothing -')
```

```
>>> import zope.component
>>> zope.component.provideAdapter(NoValueMessage, name='noValueMessage')
```

After updating the widget, the no value message changed to the value provided by the adapter:

```
>>> widget.update()
>>> widget.noValueMessage
'- nothing -'
```

Select json_data representation:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'options': [{'content': '- nothing -',
              'id': 'widget-id-novalue',
              'selected': True,
              'value': '--NOVALUE--'},
             {'content': 'a',
              'id': 'widget-id-0',
              'selected': False,
              'value': 'a'},
             {'content': 'b',
              'id': 'widget-id-1',
              'selected': False,
              'value': 'b'},
             {'content': 'c',
              'id': 'widget-id-2',
              'selected': False,
              'value': 'c'}],
 'required': False,
 'type': 'select',
 'value': ()}
```

Explicit Selection Prompt

In certain scenarios it is desirable to ask the user to select a value and display it as the first choice, such as "please select a value". In those cases you just have to set the `prompt` attribute to `True`:

```
>>> widget.prompt = True
>>> widget.update()
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget" size="1">
<option id="widget-id-novalue" value="--NOVALUE--"
        selected="selected">Select a value ...</option>
<option id="widget-id-0" value="a">a</option>
<option id="widget-id-1" value="b">b</option>
<option id="widget-id-2" value="c">c</option>
</select>
<input name="widget.name-empty-marker" type="hidden"
       value="1" />
```

As you can see, even though the field is not required, only the explicit prompt is shown. However, the prompt will also be shown if the field is required:

```
>>> widget.required = True
>>> widget.update()
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget required" size="1">
<option id="widget-id-novalue" value="--NOVALUE--"
        selected="selected">Select a value ...</option>
<option id="widget-id-0" value="a">a</option>
<option id="widget-id-1" value="b">b</option>
<option id="widget-id-2" value="c">c</option>
</select>
<input name="widget.name-empty-marker" type="hidden"
       value="1" />
```

Since the prompy uses the "No value" as the value for the selection, all behavior is identical to selecting "No value". As for the no-value message, the prompt message, which is available under

```
>>> widget.promptMessage
'Select a value ...'
```

can also be changed using an attribute value adapter:

```
>>> PromptMessage = StaticWidgetAttribute('Please select a value')
>>> zope.component.provideAdapter(PromptMessage, name='promptMessage')
```

So after updating the widget you have the custom value:

```
>>> widget.update()
>>> widget.promptMessage
'Please select a value'
```

Additionally, the select widget also allows dynamic value for the `prompt` attribute . Initially, value is `False`:

```
>>> widget.prompt = False
>>> widget.prompt
False
```

Let's now register an attribute value:

```
>>> from z3c.form.widget import StaticWidgetAttribute
>>> AllowPrompt = StaticWidgetAttribute(True)
```

```
>>> import zope.component
>>> zope.component.provideAdapter(AllowPrompt, name='prompt')
```

After updating the widget, the value for the prompt attribute changed to the value provided by the adapter:

```
>>> widget.update()
>>> widget.prompt
True
```

Display Widget

The select widget comes with a template for DISPLAY_MODE. Let's register it first:

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('select_display.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISelectWidget),
...     IPageTemplate, name=interfaces.DISPLAY_MODE)
```

```
>>> widget.mode = interfaces.DISPLAY_MODE
>>> widget.value = ['b', 'c']
>>> widget.update()
>>> print(widget.render())
<span id="widget-id" class="select-widget required">
  <span class="selected-option">b</span>,
  <span class="selected-option">c</span>
</span>
```

Let's see what happens if we have values that are not in the vocabulary:

```
>>> widget.value = ['x', 'y']
>>> widget.update()
>>> print(widget.render())
<span id="widget-id" class="select-widget required"></span>
```

Hidden Widget

The select widget comes with a template for HIDDEN_MODE. Let's register it first:

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('select_hidden.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISelectWidget),
...     IPageTemplate, name=interfaces.HIDDEN_MODE)
```

We can now set our widget's mode to hidden and render it:

```
>>> widget.mode = interfaces.HIDDEN_MODE
>>> widget.value = ['b']
>>> widget.update()
>>> print(widget.render())
<input type="hidden" name="widget.name:list"
       class="hidden-widget" value="b" id="widget-id-1" />
<input name="widget.name-empty-marker" type="hidden"
       value="1" />
```

Let's see what happens if we have values that are not in the vocabulary:

```
>>> widget.value = ['x', 'y']
>>> widget.update()
>>> print(widget.render())
<input name="widget.name-empty-marker" type="hidden"
       value="1" />
```

4.7.2 Select Widget, missing terms

The select widget allows you to select one or more values from a set of given options. The "SELECT" and "OPTION" elements are described here:

http://www.w3.org/TR/1999/REC-html401-19991224/interact/forms.html#edef-SELECT

As for all widgets, the select widget must provide the new `IWidget` interface:

```
>>> from z3c.form import interfaces
>>> from z3c.form.browser import select
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = select.SelectWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

We also need to register the template for at least the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('select_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISelectWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

We need some context:

```
>>> class IPerson(zope.interface.Interface):
...     rating = zope.schema.Choice(
...     vocabulary='Ratings')
```

```
>>> @zope.interface.implementer(IPerson)
... class Person(object):
...     pass
>>> person = Person()
```

Let's provide some values for this widget. We can do this by defining a source providing `ITerms`. This source uses descriminators which will fit our setup.

```
>>> import zope.schema.interfaces
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> from zope.schema.vocabulary import SimpleTerm
>>> import z3c.form.term
```

```
>>> from zope.schema import vocabulary
>>> ratings = vocabulary.SimpleVocabulary([
...     vocabulary.SimpleVocabulary.createTerm(0, '0', u'bad'),
...     vocabulary.SimpleVocabulary.createTerm(1, '1', u'okay'),
...     vocabulary.SimpleVocabulary.createTerm(2, '2', u'good')
...     ])
```

```
>>> def RatingsVocabulary(obj):
...     return ratings
```

```
>>> vr = vocabulary.getVocabularyRegistry()
>>> vr.register('Ratings', RatingsVocabulary)
```

```
>>> class SelectionTerms(z3c.form.term.MissingChoiceTermsVocabulary):
...     def __init__(self, context, request, form, field, widget):
...         self.context = context
...         self.field = field
...         self.terms = ratings
...         self.widget = widget
...
...     def _makeMissingTerm(self, token):
...         if token == 'x':
...             return super(SelectionTerms, self)._makeMissingTerm(token)
...         else:
...             raise LookupError
```

```
>>> zope.component.provideAdapter(SelectionTerms,
...     (None, interfaces.IFormLayer, None, None, interfaces.ISelectWidget) )
```

```
>>> import z3c.form.datamanager
>>> zope.component.provideAdapter(z3c.form.datamanager.AttributeField)
```

Now let's try if we get widget values:

```
>>> widget.update()
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget" size="1">
<option id="widget-id-novalue" selected="selected" value="--NOVALUE--">No value</option>
<option id="widget-id-0" value="0">bad</option>
<option id="widget-id-1" value="1">okay</option>
<option id="widget-id-2" value="2">good</option>
</select>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

If we set the widget value to "x", then it should be present and selected:

```
>>> widget.value = ('x',)
>>> widget.context = person
>>> widget.field = IPerson['rating']
>>> zope.interface.alsoProvides(widget, interfaces.IContextAware)
```

(continues on next page)

(continued from previous page)

```
>>> person.rating = 'x'
>>> widget.terms = None
```

```
>>> widget.update()
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget" size="1">
<option id="widget-id-novalue" value="--NOVALUE--">No value</option>
<option id="widget-id-0" value="0">bad</option>
<option id="widget-id-1" value="1">okay</option>
<option id="widget-id-2" value="2">good</option>
<option id="widget-id-missing-0" selected="selected" value="x">Missing: x</option>
</select>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

If we set the widget value to "y", then it should NOT be around:

```
>>> widget.value = ['y']
>>> widget.update()
>>> print(widget.render())
<select id="widget-id" name="widget.name:list"
        class="select-widget" size="1">
<option id="widget-id-novalue" value="--NOVALUE--">No value</option>
<option id="widget-id-0" value="0">bad</option>
<option id="widget-id-1" value="1">okay</option>
<option id="widget-id-2" value="2">good</option>
</select>
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> widget.request = TestRequest(form={'widget.name': ['c']})
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

Well, only of it matches the context's current value:

```
>>> widget.request = TestRequest(form={'widget.name': ['x']})
>>> widget.update()
>>> widget.extract()
('x',)
```

When "No value" is selected, then no verification against the terms is done:

```
>>> widget.request = TestRequest(form={'widget.name': ['--NOVALUE--']})
>>> widget.update()
>>> widget.extract(default=1)
('--NOVALUE--',)
```

Let's now make sure that we can extract user entered missing data from a widget:

```
>>> widget.request = TestRequest(form={'widget.name': ['x']})
>>> widget.update()
>>> widget.extract()
('x',)
```

```
>>> widget.request = TestRequest(form={'widget.name': ['y']})
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

Unfortunately, when nothing is selected, we do not get an empty list sent into the request, but simply no entry at all. For this we have the empty marker, so that:

```
>>> widget.request = TestRequest(form={'widget.name-empty-marker': '1'})
>>> widget.update()
>>> widget.extract()
()
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = TestRequest()
>>> widget.update()
>>> widget.extract(default=1)
1
```

Let's now make sure that a bogus value causes extract to return the default as described by the interface:

```
>>> widget.request = TestRequest(form={'widget.name': ['y']})
>>> widget.update()
>>> widget.extract(default=1)
1
```

Display Widget

The select widget comes with a template for DISPLAY_MODE. Let's register it first:

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('select_display.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISelectWidget),
...     IPageTemplate, name=interfaces.DISPLAY_MODE)
```

Let's see what happens if we have values that are not in the vocabulary:

```
>>> widget.required = True
>>> widget.mode = interfaces.DISPLAY_MODE
>>> widget.value = ['0', '1', 'x']
>>> widget.update()
>>> print(widget.render())
<span id="widget-id" class="select-widget">
  <span class="selected-option">bad</span>,
  <span class="selected-option">okay</span>,
  <span class="selected-option">Missing: x</span>
</span>
```

Hidden Widget

The select widget comes with a template for HIDDEN_MODE. Let's register it first:

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('select_hidden.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISelectWidget),
...     IPageTemplate, name=interfaces.HIDDEN_MODE)
```

Let's see what happens if we have values that are not in the vocabulary:

```
>>> widget.mode = interfaces.HIDDEN_MODE
>>> widget.value = ['0', 'x']
>>> widget.update()
>>> print(widget.render())
<input id="widget-id-0" name="widget.name:list" value="0" class="hidden-widget" type=
↪"hidden" />
<input id="widget-id-missing-0" name="widget.name:list" value="x" class="hidden-widget"␣
↪type="hidden" />
<input name="widget.name-empty-marker" type="hidden" value="1" />
```

4.7.3 Customizing widget lookup for IChoice

Widgets for fields implementing IChoice are looked up not only according to the field, but also according to the source used by the field.

```
>>> import z3c.form.testing
>>> import zope.interface
>>> import zope.component
>>> from z3c.form import interfaces
>>> from z3c.form.testing import TestRequest
```

```
>>> z3c.form.testing.setupFormDefaults()
>>> def setupWidget(field):
...     request = TestRequest()
...     widget = zope.component.getMultiAdapter((field, request),
...         interfaces.IFieldWidget)
...     widget.id = 'foo'
...     widget.name = 'bar'
...     return widget
```

We define a sample field and source:

```
>>> from zope.schema import vocabulary
>>> terms = [vocabulary.SimpleTerm(*value) for value in
...          [(True, 'yes', 'Yes'), (False, 'no', 'No')]]
>>> vocabulary = vocabulary.SimpleVocabulary(terms)
>>> field = zope.schema.Choice(default=True, vocabulary=vocabulary)
```

The default widget is the SelectWidget:

```
>>> widget = setupWidget(field)
>>> type(widget)
<class 'z3c.form.browser.select.SelectWidget'>
```

But now we define a marker interface and have our source provide it:

```
>>> from z3c.form.widget import FieldWidget
>>> class ISampleSource(zope.interface.Interface):
...     pass
>>> zope.interface.alsoProvides(vocabulary, ISampleSource)
```

We can then create and register a special widget for fields using sources with the ISampleSource marker:

```
>>> class SampleSelectWidget(z3c.form.browser.select.SelectWidget):
...     pass
>>> def SampleSelectFieldWidget(field, source, request):
...     return FieldWidget(field, SampleSelectWidget(request))
>>> zope.component.provideAdapter(
...   SampleSelectFieldWidget,
...   (zope.schema.interfaces.IChoice, ISampleSource, interfaces.IFormLayer),
...   interfaces.IFieldWidget)
```

If we now look up the widget for the field, we get the specialized widget:

```
>>> widget = setupWidget(field)
>>> type(widget)
<class 'SampleSelectWidget'>
```

Backwards compatibility

To maintain backwards compatibility, SelectFieldWidget() still can be called without passing a source:

```
>>> import z3c.form.browser.select
>>> request = TestRequest()
>>> widget = z3c.form.browser.select.SelectFieldWidget(field, request)
>>> type(widget)
<class 'z3c.form.browser.select.SelectWidget'>
```

4.8 Ordered-Select Widget

The ordered select widget allows you to select one or more values from a set of given options and sort those options. Unfortunately, HTML does not provide such a widget as part of its specification, so that the system has to use a combnation of "option" elements, buttons and Javascript.

As for all widgets, the select widget must provide the new `IWidget` interface:

```
>>> from zope.interface import verify
>>> from z3c.form import interfaces
>>> from z3c.form.browser import orderedselect
```

```
>>> verify.verifyClass(interfaces.IWidget, orderedselect.OrderedSelectWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form import testing
>>> request = testing.TestRequest()
```

```
>>> widget = orderedselect.OrderedSelectWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

We also need to register the template for at least the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('orderedselect_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IOrderedSelectWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get an empty widget:

```
>>> print(widget.render())
<script type="text/javascript" src="++resource++orderedselect_input.js" />
<table border="0" class="ordered-selection-field" id="widget-id">
  <tr>
    <td>
      <select id="widget-id-from" name="widget.name.from"
              size="5" multiple="multiple">
      </select>
    </td>
    <td>
      <button name="from2toButton" type="button"
              value="&rarr;"
              onclick="javascript:from2to('widget-id')">&rarr;</button>
      <br />
      <button name="to2fromButton" type="button"
              value="&larr;"
              onclick="javascript:to2from('widget-id')">&larr;</button>
    </td>
    <td>
      <select id="widget-id-to" name="widget.name.to"
              size="5" multiple="multiple">
      </select>
      <input name="widget.name-empty-marker" type="hidden" />
      <span id="widget-id-toDataContainer" style="display: none">
```

(continues on next page)

(continued from previous page)

```
        <script type="text/javascript">
          copyDataForSubmit('widget-id');</script>
      </span>
    </td>
    <td>
      <button name="upButton" type="button" value="&uarr;"
              onclick="javascript:moveUp('widget-id')">&uarr;</button>
      <br />
      <button name="downButton" type="button" value="&darr;"
              onclick="javascript:moveDown('widget-id')">&darr;</button>
    </td>
  </tr>
</table>
```

The json data representing the oredered select widget:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'notSelected': (),
 'options': (),
 'required': False,
 'selected': (),
 'type': 'multiSelect',
 'value': ()}
```

Let's provide some values for this widget. We can do this by defining a source providing `ITerms`. This source uses descriminators wich will fit our setup.

```
>>> import zope.schema.interfaces
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> import z3c.form.term
```

```
>>> class SelectionTerms(z3c.form.term.Terms):
...     def __init__(self, context, request, form, field, widget):
...         self.terms = SimpleVocabulary([
...              SimpleVocabulary.createTerm(1, 'a', u'A'),
...              SimpleVocabulary.createTerm(2, 'b', u'B'),
...              SimpleVocabulary.createTerm(3, 'c', u'C'),
...              SimpleVocabulary.createTerm(4, 'd', u'A'),
...              ])
```

```
>>> zope.component.provideAdapter(SelectionTerms,
...     (None, interfaces.IFormLayer, None, None,
...      interfaces.IOrderedSelectWidget) )
```

Now let's try if we get widget values:

```
>>> widget.update()
>>> print(testing.render(widget, './/table//td[1]'))
<td>
  <select id="widget-id-from" name="widget.name.from"
          size="5" multiple="multiple">
    <option value="a">A</option>
    <option value="b">B</option>
    <option value="c">C</option>
    <option value="d">A</option>
  </select>
</td>
```

If we select item "b", then it should be selected:

```
>>> widget.value = ['b']
>>> widget.update()
>>> print(testing.render(widget, './/table//select[@id="widget-id-from"]/..'))
<td>
  <select id="widget-id-from" name="widget.name.from"
          size="5" multiple="multiple">
    <option value="a">A</option>
    <option value="c">C</option>
    <option value="d">A</option>
  </select>
</td>
```

```
>>> print(testing.render(widget, './/table//select[@id="widget-id-to"]'))
<select id="widget-id-to" name="widget.name.to"
        size="5" multiple="multiple">
  <option value="b">B</option>
</select>
```

The json data representing the oredered select widget:

```
>>> from pprint import pprint
>>> pprint(widget.json_data())
{'error': '',
 'id': 'widget-id',
 'label': '',
 'mode': 'input',
 'name': 'widget.name',
 'notSelected': [{'content': 'A', 'id': 'widget-id-0', 'value': 'a'},
                 {'content': 'C', 'id': 'widget-id-2', 'value': 'c'},
                 {'content': 'A', 'id': 'widget-id-3', 'value': 'd'}],
 'options': [{'content': 'A', 'id': 'widget-id-0', 'value': 'a'},
             {'content': 'B', 'id': 'widget-id-1', 'value': 'b'},
             {'content': 'C', 'id': 'widget-id-2', 'value': 'c'},
             {'content': 'A', 'id': 'widget-id-3', 'value': 'd'}],
 'required': False,
 'selected': [{'content': 'B', 'id': 'widget-id-0', 'value': 'b'}],
 'type': 'multiSelect',
 'value': ['b']}
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> widget.request = testing.TestRequest(form={'widget.name': ['c']})
>>> widget.update()
>>> widget.extract()
('c',)
```

Unfortunately, when nothing is selected, we do not get an empty list sent into the request, but simply no entry at all. For this we have the empty marker, so that:

```
>>> widget.request = testing.TestRequest(form={'widget.name-empty-marker': '1'})
>>> widget.update()
>>> widget.extract()
()
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = testing.TestRequest()
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

Let's now make sure that a bogus value causes extract to return the default as described by the interface:

```
>>> widget.request = testing.TestRequest(form={'widget.name': ['x']})
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

Finally, let's check correctness of widget rendering in one rare case when we got selection terms with callable values and without titles. For example, you can get those terms when you using the "Content Types" vocabulary from zope.app.content.

```
>>> class CallableValue(object):
...     def __init__(self, value):
...         self.value = value
...     def __call__(self):
...         pass
...     def __str__(self):
...        return 'Callable Value %s' % self.value
```

```
>>> class SelectionTermsWithCallableValues(z3c.form.term.Terms):
...     def __init__(self, context, request, form, field, widget):
...         self.terms = SimpleVocabulary([
...              SimpleVocabulary.createTerm(CallableValue(1), 'a'),
...              SimpleVocabulary.createTerm(CallableValue(2), 'b'),
...              SimpleVocabulary.createTerm(CallableValue(3), 'c')
...              ])
```

```
>>> widget.terms = SelectionTermsWithCallableValues(
...     None, testing.TestRequest(), None, None, widget)
>>> widget.update()
>>> print(testing.render(widget, './/table//select[@id="widget-id-from"]'))
<select id="widget-id-from" name="widget.name.from"
        size="5" multiple="multiple">
```

(continues on next page)

(continued from previous page)

```
  <option value="a">Callable Value 1</option>
  <option value="b">Callable Value 2</option>
  <option value="c">Callable Value 3</option>
</select>
```

4.9 File Widget

The file widget allows you to upload a new file to the server. The "file" type of the "INPUT" element is described here:

http://www.w3.org/TR/1999/REC-html401-19991224/interact/forms.html#edef-INPUT

As for all widgets, the file widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import file
```

```
>>> verifyClass(interfaces.IWidget, file.FileWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = file.FileWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget.id'
>>> widget.name = 'widget.name'
```

We also need to register the template for the widget:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('file_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IFileWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get a simple input element:

```
>>> print(widget.render())
<input type="file" id="widget.id" name="widget.name"
       class="file-widget" />
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> from io import BytesIO
>>> myfile = BytesIO(b'My file contents.')
```

```
>>> widget.request = TestRequest(form={'widget.name': myfile})
>>> widget.update()
>>> isinstance(widget.extract(), BytesIO)
True
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = TestRequest()
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

Make also sure that we can handle FileUpload objects given form a file upload.

```
>>> from zope.publisher.browser import FileUpload
```

Let's define a FieldStorage stub:

```
>>> class FieldStorageStub:
...     def __init__(self, file):
...         self.file = file
...         self.headers = {}
...         self.filename = 'foo.bar'
```

Now build a FileUpload:

```
>>> myfile = BytesIO(b'File upload contents.')
>>> aFieldStorage = FieldStorageStub(myfile)
>>> myUpload = FileUpload(aFieldStorage)
```

```
>>> widget.request = TestRequest(form={'widget.name': myUpload})
>>> widget.update()
>>> widget.extract()
<zope.publisher.browser.FileUpload object at ...>
```

If we render them, we get a regular file upload widget:

```
>>> print(widget.render())
<input type="file" id="widget.id" name="widget.name"
       class="file-widget" />
```

4.10 File Testing Widget

The File Testing widget is just like the file widget except it has another hidden field where the contents of a would be file can be uploaded in a textarea. As for all widgets, the file widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import file
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = file.FileWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget.id'
>>> widget.name = 'widget.name'
```

We also need to register the template for the widget:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('file_testing_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IFileWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get a text area element instead of a simple input element, but also with a text area:

```
>>> print(widget.render())
<input type="file" id="widget.id" name="widget.name"
       class="file-widget" />
<input name="widget.name.encoding" type="hidden" value="plain">
<textarea name="widget.name.testing" style="display: none;"><!--
   nothing here --></textarea>
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> from io import BytesIO
>>> myfile = BytesIO(b'My file contents.')
```

```
>>> widget.request = TestRequest(form={'widget.name': myfile})
>>> widget.update()
>>> isinstance(widget.extract(), BytesIO)
True
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = TestRequest()
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

Make also sure that we can handle FileUpload objects given from a file upload.

```
>>> from zope.publisher.browser import FileUpload
```

Let's define a FieldStorage stub:

```
>>> class FieldStorageStub:
...     def __init__(self, file):
...         self.file = file
...         self.headers = {}
...         self.filename = 'foo.bar'
```

Now build a FileUpload:

```
>>> myfile = BytesIO(b'File upload contents.')
>>> aFieldStorage = FieldStorageStub(myfile)
>>> myUpload = FileUpload(aFieldStorage)
```

```
>>> widget.request = TestRequest(form={'widget.name': myUpload})
>>> widget.update()
>>> widget.extract()
<zope.publisher.browser.FileUpload object at ...>
```

If we render them, we get a regular file upload widget:

```
>>> print(widget.render())
<input type="file" id="widget.id" name="widget.name"
       class="file-widget" />
<input name="widget.name.encoding" type="hidden" value="plain">
<textarea name="widget.name.testing" style="display: none;"><!--
   nothing here --></textarea>
```

Alternatively, we can also pass in the file upload content via the testing text area:

```
>>> widget.request = TestRequest(
...     form={'widget.name.testing': 'File upload contents.'})
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

The extract method uses the request directly, but we can get the value using the data converter.

```
>>> from z3c.form import testing
>>> import zope.schema
>>> conv = testing.TestingFileUploadDataConverter(
...     zope.schema.Bytes(), widget)
>>> conv
<TestingFileUploadDataConverter converts from Bytes to FileWidget>
```

(continues on next page)

(continued from previous page)

```
>>> conv.toFieldValue("")
b'File upload contents.'
```

4.11 Image Widget

The image widget allows you to submit a form to the server by clicking on an image. The "image" type of the "INPUT" element is described here:

http://www.w3.org/TR/1999/REC-html401-19991224/interact/forms.html#edef-INPUT

As for all widgets, the image widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import image
```

```
>>> verifyClass(interfaces.IWidget, image.ImageWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = image.ImageWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget.id'
>>> widget.name = 'widget.name'
```

We also need to register the template for the widget:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('image_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IImageWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get a simple input element:

```
>>> print(widget.render())
<input type="image" id="widget.id" name="widget.name"
       class="image-widget" />
```

Setting an image source for the widget effectively changes the "src" attribute:

```
>>> widget.src = u'widget.png'
>>> print(widget.render())
<input type="image" id="widget.id" name="widget.name"
       class="image-widget" src="widget.png" />
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> widget.request = TestRequest(
...     form={'widget.name.x': '10',
...           'widget.name.y': '20',
...           'widget.name': 'value'})
>>> widget.update()
>>> sorted(widget.extract().items())
[('value', 'value'), ('x', 10), ('y', 20)]
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = TestRequest()
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

4.12 Multi Widget

The multi widget allows you to add and edit one or more values.

4.12.1 Multi Widget

The multi widget allows you to add and edit one or more values.

As for all widgets, the multi widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import multi
```

```
>>> verifyClass(interfaces.IWidget, multi.MultiWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = multi.MultiWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

We also need to register the template for at least the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('multi_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IMultiWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

For the next test, we need to setup our button handler adapters.

```
>>> from z3c.form import button
>>> zope.component.provideAdapter(button.ButtonActions)
>>> zope.component.provideAdapter(button.ButtonActionHandler)
>>> zope.component.provideAdapter(button.ButtonAction,
...     provides=interfaces.IButtonAction)
```

Our submit buttons will need a template as well:

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('submit_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISubmitWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

We can now render the widget:

```
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="0" />
```

As you can see the widget is empty and doesn't provide values. This is because the widget does not know what sub-widgets to display. So let's register a *IFieldWidget* adapter and a template for our *IInt* field:

```
>>> import z3c.form.interfaces
>>> from z3c.form.browser.text import TextFieldWidget
>>> zope.component.provideAdapter(TextFieldWidget,
...     (zope.schema.interfaces.IInt, z3c.form.interfaces.IFormLayer))
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('text_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ITextWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

Let's now update the widget and check it again.

```
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="0" />
```

It's still the same. Since the widget doesn't provide a field nothing useful gets rendered. Now let's define a field for this widget and check it again:

```
>>> field = zope.schema.List(
...     __name__='foo',
...     value_type=zope.schema.Int(title='Number'),
...     )
>>> widget.field = field
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="0" />
```

As you can see, there is still no input value. Let's provide some values for this widget. Before we can do that, we will need to register a data converter for our multi widget and the data converter dispatcher adapter:

```
>>> from z3c.form.converter import IntegerDataConverter
>>> from z3c.form.converter import FieldWidgetDataConverter
>>> from z3c.form.validator import SimpleFieldValidator
>>> zope.component.provideAdapter(IntegerDataConverter)
>>> zope.component.provideAdapter(FieldWidgetDataConverter)
>>> zope.component.provideAdapter(SimpleFieldValidator)
```

```
>>> widget.value = ['42', '43']
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
```

(continues on next page)

(continued from previous page)

```
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-0-remove"
                   name="widget.name.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-0" name="widget.name.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
    <div id="widget-id-1-row" class="row">
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-1-remove"
                   name="widget.name.1.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-1" name="widget.name.1"
               class="text-widget required int-field" value="43" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="2" />
```

If we now click on the Add button, we will get a new input field for enter a new value:

```
>>> widget.request = TestRequest(form={'widget.name.count':'2',
...                                    'widget.name.0':'42',
...                                    'widget.name.1':'43',
...                                    'widget.name.buttons.add':'Add'})
```

(continues on next page)

(continued from previous page)

```
>>> widget.update()
```

```
>>> widget.extract()
['42', '43']
```

```
>>> print(widget.render())
<div class="multi-widget">
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-0-remove"
                   name="widget.name.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-0" name="widget.name.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
    <div id="widget-id-1-row" class="row">
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-1-remove"
                   name="widget.name.1.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-1" name="widget.name.1"
               class="text-widget required int-field" value="43" />
        </div>
      </div>
    </div>
    <div id="widget-id-2-row" class="row">
        <div class="label">
          <label for="widget-id-2">
            <span>Number</span>
```

(continues on next page)

(continued from previous page)

```
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-2-remove"
                   name="widget.name.2.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-2" name="widget.name.2"
               class="text-widget required int-field" value="" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="3" />
```

Now let's store the new value:

```
>>> widget.request = TestRequest(form={'widget.name.count':'3',
...                                    'widget.name.0':'42',
...                                    'widget.name.1':'43',
...                                    'widget.name.2':'44'})
>>> widget.update()
```

```
>>> widget.extract()
['42', '43', '44']
```

```
>>> print(widget.render())
<div class="multi-widget">
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-0-remove"
```

(continues on next page)

(continued from previous page)

```
                  name="widget.name.0.remove" />
        </div>
        <div class="multi-widget-input"><input
             type="text" id="widget-id-0" name="widget.name.0"
             class="text-widget required int-field" value="42" />
      </div>
    </div>
  </div>
  <div id="widget-id-1-row" class="row">
      <div class="label">
        <label for="widget-id-1">
          <span>Number</span>
          <span class="required">*</span>
        </label>
      </div>
      <div class="widget">
        <div class="multi-widget-checkbox">
          <input type="checkbox" value="1"
                 class="multi-widget-checkbox checkbox-widget"
                 id="widget-id-1-remove"
                 name="widget.name.1.remove" />
        </div>
        <div class="multi-widget-input"><input
             type="text" id="widget-id-1" name="widget.name.1"
             class="text-widget required int-field" value="43" />
      </div>
    </div>
  </div>
  <div id="widget-id-2-row" class="row">
      <div class="label">
        <label for="widget-id-2">
          <span>Number</span>
          <span class="required">*</span>
        </label>
      </div>
      <div class="widget">
        <div class="multi-widget-checkbox">
          <input type="checkbox" value="1"
                 class="multi-widget-checkbox checkbox-widget"
                 id="widget-id-2-remove"
                 name="widget.name.2.remove" />
        </div>
        <div class="multi-widget-input"><input
             type="text" id="widget-id-2" name="widget.name.2"
             class="text-widget required int-field" value="44" />
        </div>
      </div>
  </div>
<div class="buttons">
  <input type="submit" id="widget-name-buttons-add"
     name="widget.name.buttons.add"
     class="submit-widget button-field" value="Add" />
```

(continues on next page)

(continued from previous page)

```
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="3" />
```

As you can see in the above sample, the new stored value get rendered as a real value and the new adding value input field is gone. Now let's try to remove an existing value:

```
>>> widget.request = TestRequest(form={'widget.name.count':'3',
...                                    'widget.name.0':'42',
...                                    'widget.name.1':'43',
...                                    'widget.name.2':'44',
...                                    'widget.name.1.remove':'1',
...                                    'widget.name.buttons.remove':'Remove selected'})
>>> widget.update()
```

This is good so, because the Remove selected is an widget-internal submit action

```
>>> widget.extract()
['42', '43', '44']
```

```
>>> print(widget.render())
<div class="multi-widget">
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-0-remove"
                   name="widget.name.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-0" name="widget.name.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
    <div id="widget-id-1-row" class="row">
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
```

(continues on next page)

(continued from previous page)

```
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-1-remove"
                   name="widget.name.1.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-1" name="widget.name.1"
               class="text-widget required int-field" value="44" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="2" />
```

Change again a value after delete:

```
>>> widget.request = TestRequest(form={'widget.name.count':'2',
...                                    'widget.name.0':'42',
...                                    'widget.name.1':'45'})
>>> widget.update()
```

```
>>> print(widget.render())
<div class="multi-widget">
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input id="widget-id-0-remove" name="widget.name.0.remove"
            class="multi-widget-checkbox checkbox-widget"
            type="checkbox" value="1" />
          </div>
          <div class="multi-widget-input">
              <input id="widget-id-0" name="widget.name.0"
              class="text-widget required int-field" value="42" type="text" />
          </div>
        </div>
    </div>
```

(continues on next page)

(continued from previous page)

```
    <div id="widget-id-1-row" class="row">
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input id="widget-id-1-remove" name="widget.name.1.remove"
            class="multi-widget-checkbox checkbox-widget"
            type="checkbox" value="1" />
          </div>
          <div class="multi-widget-input">
              <input id="widget-id-1" name="widget.name.1"
              class="text-widget required int-field" value="45" type="text" />
          </div>
        </div>
    </div>
  <div class="buttons">
    <input id="widget-name-buttons-add" name="widget.name.buttons.add"
    class="submit-widget button-field" value="Add" type="submit" />
    <input id="widget-name-buttons-remove" name="widget.name.buttons.remove"
    class="submit-widget button-field" value="Remove selected" type="submit" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="2" />
```

Error handling is next. Let's use the value "bad" (an invalid integer literal) as input for our internal (sub) widget.

```
>>> from z3c.form.error import ErrorViewSnippet
>>> from z3c.form.error import StandardErrorViewTemplate
>>> zope.component.provideAdapter(ErrorViewSnippet)
>>> zope.component.provideAdapter(StandardErrorViewTemplate)
```

```
>>> widget.request = TestRequest(form={'widget.name.count':'2',
...                                    'widget.name.0':'42',
...                                    'widget.name.1':'bad'})
>>> widget.update()
```

```
>>> widget.extract()
['42', 'bad']
```

```
>>> print(widget.render())
<div class="multi-widget">
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
```

(continues on next page)

(continued from previous page)

```
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-0-remove"
                   name="widget.name.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-0" name="widget.name.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
    <div id="widget-id-1-row" class="row">
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="error">The entered value is not a valid integer
                           literal.</div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-1-remove"
                   name="widget.name.1.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-1" name="widget.name.1"
               class="text-widget required int-field" value="bad" />
          </div>
        </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="2" />
```

The widget filters out the add and remove buttons depending on the current value and the field constraints. You already saw that there's no remove button for empty value. Now, let's check rendering with minimum and maximum lengths defined in the field constraints.

```
>>> field = zope.schema.List(
```

(continues on next page)

(continued from previous page)

```
...     __name__='foo',
...     value_type=zope.schema.Int(title='Number'),
...     min_length=1,
...     max_length=3
...     )
>>> widget.field = field
>>> widget.widgets = []
>>> widget.value = []
```

Let's test with minimum sequence, there should be no remove button:

```
>>> widget.request = TestRequest(form={'widget.name.count':'1',
...                                    'widget.name.0':'42'})
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-0-remove"
                   name="widget.name.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-0" name="widget.name.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="1" />
```

Now, with middle-length sequence. All buttons should be there.

```
>>> widget.request = TestRequest(form={'widget.name.count':'2',
...                                    'widget.name.0':'42',
...                                    'widget.name.1':'43'})
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
```

(continues on next page)

(continued from previous page)

```
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-0-remove"
                   name="widget.name.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-0" name="widget.name.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
    <div id="widget-id-1-row" class="row">
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-1-remove"
                   name="widget.name.1.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-1" name="widget.name.1"
               class="text-widget required int-field" value="43" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="2" />
```

Okay, now let's check the maximum-length sequence. There should be no add button:

```
>>> widget.request = TestRequest(form={'widget.name.count':'3',
...                                    'widget.name.0':'42',
...                                    'widget.name.1':'43',
...                                    'widget.name.2':'44'})
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
    <div id="widget-id-0-row" class="row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-0-remove"
                   name="widget.name.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-0" name="widget.name.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
    <div id="widget-id-1-row" class="row">
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-1-remove"
                   name="widget.name.1.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-1" name="widget.name.1"
               class="text-widget required int-field" value="43" />
        </div>
      </div>
    </div>
    <div id="widget-id-2-row" class="row">
        <div class="label">
          <label for="widget-id-2">
            <span>Number</span>
            <span class="required">*</span>
          </label>
```

(continues on next page)

(continued from previous page)

```
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="widget-id-2-remove"
                   name="widget.name.2.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="widget-id-2" name="widget.name.2"
               class="text-widget required int-field" value="44" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="3" />
```

Dictionaries

The multi widget also supports IDict schemas.

```
>>> field = zope.schema.Dict(
...     __name__='foo',
...     key_type=zope.schema.Int(title='Number'),
...     value_type=zope.schema.Int(title='Number'),
...     )
>>> widget.field = field
>>> widget.widgets = []
>>> widget.value = [('1','42')]
>>> print(widget.render())
  <div class="multi-widget">
      <div id="widget-id-0-row" class="row">
            <div class="label">
              <label for="widget-id-key-0">
                <span>Number</span>
                <span class="required">*</span>
              </label>
            </div>
            <div class="widget">
                <div class="multi-widget-input-key">
      <input id="widget-id-key-0" name="widget.name.key.0" class="text-widget required␣
↪int-field" value="1" type="text" />
  </div>
            </div>
          <div class="label">
            <label for="widget-id-0">
```

(continues on next page)

(continued from previous page)

```
              <span>Number</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <div class="multi-widget-checkbox">
              <input id="widget-id-0-remove" name="widget.name.0.remove" class="multi-
↪widget-checkbox checkbox-widget" type="checkbox" value="1" />
            </div>
            <div class="multi-widget-input">
      <input id="widget-id-0" name="widget.name.0" class="text-widget required int-field
↪" value="42" type="text" />
  </div>
          </div>
      </div>
   <div class="buttons">
  <input id="widget-name-buttons-remove" name="widget.name.buttons.remove" class="submit-
↪widget button-field" value="Remove selected" type="submit" />
     </div>
  </div>
  <input type="hidden" name="widget.name.count" value="1" />
```

If we now click on the Add button, we will get a new input field for entering a new value:

```
>>> widget.request = TestRequest(form={'widget.name.count':'1',
...                                    'widget.name.key.0':'1',
...                                    'widget.name.0':'42',
...                                    'widget.name.buttons.add':'Add'})
>>> widget.update()
```

```
>>> widget.extract()
[('1', '42')]
```

```
>>> print(widget.render())
<html>
  <body>
    <div class="multi-widget">
      <div class="row" id="widget-id-0-row">
        <div class="label">
          <label for="widget-id-key-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-input-key">
            <input class="text-widget required int-field" id="widget-id-key-0" name=
↪"widget.name.key.0" type="text" value="1">
          </div>
        </div>
        <div class="label">
          <label for="widget-id-0">
```

(continues on next page)

(continued from previous page)

```
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input class="multi-widget-checkbox checkbox-widget" id="widget-id-0-remove"␣
↪name="widget.name.0.remove" type="checkbox" value="1">
          </div>
          <div class="multi-widget-input">
            <input class="text-widget required int-field" id="widget-id-0" name="widget.
↪name.0" type="text" value="42">
          </div>
        </div>
      </div>
      <div class="row" id="widget-id-1-row">
        <div class="label">
          <label for="widget-id-key-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-input-key">
            <input class="text-widget required int-field" id="widget-id-key-1" name=
↪"widget.name.key.1" type="text" value="">
          </div>
        </div>
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input class="multi-widget-checkbox checkbox-widget" id="widget-id-1-remove"␣
↪name="widget.name.1.remove" type="checkbox" value="1">
          </div>
          <div class="multi-widget-input">
            <input class="text-widget required int-field" id="widget-id-1" name="widget.
↪name.1" type="text" value="">
          </div>
        </div>
      </div>
      <div class="buttons">
        <input class="submit-widget button-field" id="widget-name-buttons-add" name=
↪"widget.name.buttons.add" type="submit" value="Add">
        <input class="submit-widget button-field" id="widget-name-buttons-remove" name=
↪"widget.name.buttons.remove" type="submit" value="Remove selected">
      </div>
    </div>
```

(continues on next page)

(continued from previous page)

```
    <input name="widget.name.count" type="hidden" value="2">
  </body>
</html>
```

Now let's store the new value:

```
>>> widget.request = TestRequest(form={'widget.name.count':'2',
...                                    'widget.name.key.0':'1',
...                                    'widget.name.0':'42',
...                                    'widget.name.key.1':'2',
...                                    'widget.name.1':'43'})
>>> widget.update()
```

```
>>> widget.extract()
[('1', '42'), ('2', '43')]
```

We will get an error if we try and set the same key twice

```
>>> from z3c.form.error import InvalidErrorViewSnippet
>>> zope.component.provideAdapter(InvalidErrorViewSnippet)
```

```
>>> widget.request = TestRequest(form={'widget.name.count':'2',
...                                    'widget.name.key.0':'1',
...                                    'widget.name.0':'42',
...                                    'widget.name.key.1':'1',
...                                    'widget.name.1':'43'})
>>> widget.update()
```

```
>>> widget.extract()
[('1', '42'), ('1', '43')]
```

```
>>> print(widget.render())
  <div class="multi-widget">
      <div id="widget-id-0-row" class="row">
            <div class="label">
              <label for="widget-id-key-0">
                <span>Number</span>
                <span class="required">*</span>
              </label>
            </div>
            <div class="widget">
                <div class="multi-widget-input-key">
      <input id="widget-id-key-0" name="widget.name.key.0" class="text-widget required
↪int-field" value="1" type="text" />
  </div>
            </div>
          <div class="label">
            <label for="widget-id-0">
              <span>Number</span>
              <span class="required">*</span>
            </label>
```

(continues on next page)

(continued from previous page)

```
          </div>
          <div class="widget">
            <div class="multi-widget-checkbox">
              <input id="widget-id-0-remove" name="widget.name.0.remove" class="multi-
↪widget-checkbox checkbox-widget" type="checkbox" value="1" />
            </div>
            <div class="multi-widget-input">
      <input id="widget-id-0" name="widget.name.0" class="text-widget required int-field
↪" value="42" type="text" />
  </div>
          </div>
      </div>
      <div id="widget-id-1-row" class="row">
            <div class="label">
              <label for="widget-id-key-1">
                <span>Number</span>
                <span class="required">*</span>
              </label>
            </div>
    <div class="error">Duplicate key</div>
            <div class="widget">
                <div class="multi-widget-input-key">
      <input id="widget-id-key-1" name="widget.name.key.1" class="text-widget required␣
↪int-field" value="1" type="text" />
  </div>
            </div>
          <div class="label">
            <label for="widget-id-1">
              <span>Number</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <div class="multi-widget-checkbox">
              <input id="widget-id-1-remove" name="widget.name.1.remove" class="multi-
↪widget-checkbox checkbox-widget" type="checkbox" value="1" />
            </div>
            <div class="multi-widget-input">
      <input id="widget-id-1" name="widget.name.1" class="text-widget required int-field
↪" value="43" type="text" />
  </div>
          </div>
      </div>
    <div class="buttons">
  <input id="widget-name-buttons-add" name="widget.name.buttons.add" class="submit-
↪widget button-field" value="Add" type="submit" />
  <input id="widget-name-buttons-remove" name="widget.name.buttons.remove" class="submit-
↪widget button-field" value="Remove selected" type="submit" />
    </div>
  </div>
  <input type="hidden" name="widget.name.count" value="2" />
```

Displaying

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = multi.MultiWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

Set the mode to DISPLAY_MODE:

```
>>> widget.mode = interfaces.DISPLAY_MODE
```

We also need to register the template for at least the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('multi_display.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IMultiWidget),
...     IPageTemplate, name=interfaces.DISPLAY_MODE)
```

We can now render the widget:

```
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget" id="widget-id"></div>
```

As you can see the widget is empty and doesn't provide values. This is because the widget does not know what sub-widgets to display. So let's register a *IFieldWidget* adapter and a template for our *IInt* field:

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('text_display.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ITextWidget),
...     IPageTemplate, name=interfaces.DISPLAY_MODE)
```

Let's now update the widget and check it again.

```
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget" id="widget-id"></div>
```

It's still the same. Since the widget doesn't provide a field nothing useful gets rendered. Now let's define a field for this widget and check it again:

```
>>> field = zope.schema.List(
...     __name__='foo',
```

(continues on next page)

(continued from previous page)

```
...     value_type=zope.schema.Int(title='Number'),
...     )
>>> widget.field = field
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget" id="widget-id"></div>
```

As you can see, there is still no input value. Let's provide some values for this widget. Before we can do that, we will need to register a data converter for our multi widget and the data converter dispatcher adapter:

```
>>> widget.update()
>>> widget.value = ['42', '43']
>>> print(widget.render())
<div class="multi-widget" id="widget-id">
  <div class="row" id="widget-id-0-row">
    <div class="label">
      <label for="widget-id-0">
        <span>Number</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-display">
        <span class="text-widget int-field" id="widget-id-0">42</span>
      </div>
    </div>
  </div>
  <div class="row" id="widget-id-1-row">
    <div class="label">
      <label for="widget-id-1">
        <span>Number</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-display">
        <span class="text-widget int-field" id="widget-id-1">43</span>
      </div>
    </div>
  </div>
</div>
```

We can also use the multi widget with dictionaries

```
>>> field = zope.schema.Dict(
...     __name__='foo',
...     key_type=zope.schema.Int(title='Number'),
...     value_type=zope.schema.Int(title='Number'),
...     )
>>> widget.field = field
>>> widget.value = [('1', '42'), ('2', '43')]
>>> print(widget.render())
```

(continues on next page)

(continued from previous page)

```
<div class="multi-widget" id="widget-id">
  <div class="row" id="widget-id-0-row">
    <div class="label">
      <label for="widget-id-key-0">
        <span>Number</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-display">
        <span class="text-widget int-field" id="widget-id-key-0">1</span>
      </div>
    </div>
    <div class="label">
      <label for="widget-id-0">
        <span>Number</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-display">
        <span class="text-widget int-field" id="widget-id-0">42</span>
      </div>
    </div>
  </div>
  <div class="row" id="widget-id-1-row">
    <div class="label">
      <label for="widget-id-key-1">
        <span>Number</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-display">
        <span class="text-widget int-field" id="widget-id-key-1">2</span>
      </div>
    </div>
    <div class="label">
      <label for="widget-id-1">
        <span>Number</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-display">
        <span class="text-widget int-field" id="widget-id-1">43</span>
      </div>
    </div>
  </div>
</div>
```

Hidden mode

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = multi.MultiWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

Set the mode to HIDDEN_MODE:

```
>>> widget.mode = interfaces.HIDDEN_MODE
```

We also need to register the template for at least the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('multi_hidden.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IMultiWidget),
...     IPageTemplate, name=interfaces.HIDDEN_MODE)
```

We can now render the widget:

```
>>> widget.update()
>>> print(widget.render())
<input name="widget.name.count" type="hidden" value="0">
```

As you can see the widget is empty and doesn't provide values. This is because the widget does not know what sub-widgets to display. So let's register a *IFieldWidget* adapter and a template for our *IInt* field:

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('text_hidden.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ITextWidget),
...     IPageTemplate, name=interfaces.HIDDEN_MODE)
```

Let's now update the widget and check it again.

```
>>> widget.update()
>>> print(widget.render())
<input name="widget.name.count" type="hidden" value="0">
```

It's still the same. Since the widget doesn't provide a field nothing useful gets rendered. Now let's define a field for this widget and check it again:

```
>>> field = zope.schema.List(
...     __name__='foo',
```

(continues on next page)

(continued from previous page)

```
...     value_type=zope.schema.Int(title='Number'),
...     )
>>> widget.field = field
>>> widget.update()
>>> print(widget.render())
<input name="widget.name.count" type="hidden" value="0">
```

As you can see, there is still no input value. Let's provide some values for this widget. Before we can do that, we will need to register a data converter for our multi widget and the data converter dispatcher adapter:

```
>>> widget.update()
>>> widget.value = ['42', '43']
>>> print(widget.render())
<input class="hidden-widget"
       id="widget-id-0" name="widget.name.0" type="hidden" value="42">
<input class="hidden-widget"
       id="widget-id-1" name="widget.name.1" type="hidden" value="43">
<input name="widget.name.count" type="hidden" value="2">
```

We can also use the multi widget with dictionaries

```
>>> field = zope.schema.Dict(
...     __name__='foo',
...     key_type=zope.schema.Int(title='Number'),
...     value_type=zope.schema.Int(title='Number'),
...     )
>>> widget.field = field
>>> widget.value = [('1', '42'), ('2', '43')]
>>> print(widget.render())
<input class="hidden-widget"
       id="widget-id-key-0" name="widget.name.key.0" type="hidden" value="1">
<input class="hidden-widget"
       id="widget-id-0" name="widget.name.0" type="hidden" value="42">
<input class="hidden-widget"
       id="widget-id-key-1" name="widget.name.key.1" type="hidden" value="2">
<input class="hidden-widget"
       id="widget-id-1" name="widget.name.1" type="hidden" value="43">
<input name="widget.name.count" type="hidden" value="2">
```

Label

There is an option which allows to disable the label for the subwidgets. You can set the *showLabel* option to *False* which will skip rendering the labels. Alternatively you can also register your own template for your layer if you like to skip the label rendering for all widgets. One more way is to register an attribute adapter for specific field/widget/layer/etc. See below for an example.

```
>>> field = zope.schema.List(
...     __name__='foo',
...     value_type=zope.schema.Int(
...         title='Ignored'),
...     )
```

(continues on next page)

(continued from previous page)

```
>>> request = TestRequest()
>>> widget = multi.MultiWidget(request)
>>> widget.field = field
>>> widget.value = ['42', '43']
>>> widget.showLabel = False
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
    <div id="None-0-row" class="row">
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="None-0-remove" name="None.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="None-0" name="None.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
    <div id="None-1-row" class="row">
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="None-1-remove" name="None.1.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="None-1" name="None.1"
               class="text-widget required int-field" value="43" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-buttons-add"
       name="widget.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-buttons-remove"
       name="widget.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="None.count" value="2" />
```

We can also override the showLabel attribute value with an attribute adapter. We set it to False for our widget before, but the update method sets adapted attributes, so if we provide an attribute, it will be used to set the `showLabel`. Let's see.

```
>>> from z3c.form.widget import StaticWidgetAttribute
```

```
>>> doShowLabel = StaticWidgetAttribute(True, widget=widget)
```

(continues on next page)

(continued from previous page)

```
>>> zope.component.provideAdapter(doShowLabel, name="showLabel")
```

```
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
    <div id="None-0-row" class="row">
        <div class="label">
          <label for="None-0">
            <span>Ignored</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="None-0-remove" name="None.0.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="None-0" name="None.0"
               class="text-widget required int-field" value="42" />
        </div>
      </div>
    </div>
    <div id="None-1-row" class="row">
        <div class="label">
          <label for="None-1">
            <span>Ignored</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input type="checkbox" value="1"
                   class="multi-widget-checkbox checkbox-widget"
                   id="None-1-remove" name="None.1.remove" />
          </div>
          <div class="multi-widget-input"><input
               type="text" id="None-1" name="None.1"
               class="text-widget required int-field" value="43" />
        </div>
      </div>
    </div>
  <div class="buttons">
    <input type="submit" id="widget-buttons-add"
       name="widget.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-buttons-remove"
       name="widget.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
```

(continues on next page)

(continued from previous page)

```
<input type="hidden" name="None.count" value="2" />
```

Coverage happiness

```
>>> field = zope.schema.List(
...     __name__='foo',
...     value_type=zope.schema.Int(title='Number'),
...     )
>>> request = TestRequest()
>>> widget = multi.MultiWidget(request)
>>> widget.field = field
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
>>> widget.widgets = []
>>> widget.value = []
```

```
>>> widget.request = TestRequest()
>>> widget.update()
```

```
>>> widget.value = ['42', '43', '44']
>>> widget.value = ['99']
```

```
>>> print(widget.render())
  <html>
  <body>
    <div class="multi-widget">
      <div class="row" id="widget-id-0-row">
        <div class="label">
          <label for="widget-id-0">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input class="multi-widget-checkbox checkbox-widget" id="widget-id-0-remove"␣
↪name="widget.name.0.remove" type="checkbox" value="1">
          </div>
          <div class="multi-widget-input">
            <input class="text-widget required int-field" id="widget-id-0" name="widget.
↪name.0" type="text" value="99">
          </div>
        </div>
      </div>
      <div class="row" id="widget-id-1-row">
        <div class="label">
          <label for="widget-id-1">
            <span>Number</span>
            <span class="required">*</span>
          </label>
```

(continues on next page)

(continued from previous page)

```
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input class="multi-widget-checkbox checkbox-widget" id="widget-id-1-remove"␣
↪name="widget.name.1.remove" type="checkbox" value="1">
          </div>
          <div class="multi-widget-input">
            <input class="text-widget required int-field" id="widget-id-1" name="widget.
↪name.1" type="text" value="">
          </div>
        </div>
      </div>
      <div class="row" id="widget-id-2-row">
        <div class="label">
          <label for="widget-id-2">
            <span>Number</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input class="multi-widget-checkbox checkbox-widget" id="widget-id-2-remove"␣
↪name="widget.name.2.remove" type="checkbox" value="1">
          </div>
          <div class="multi-widget-input">
            <input class="text-widget required int-field" id="widget-id-2" name="widget.
↪name.2" type="text" value="">
          </div>
        </div>
      </div>
      <div class="buttons">
        <input class="submit-widget button-field" id="widget-name-buttons-add" name=
↪"widget.name.buttons.add" type="submit" value="Add">
      </div>
    </div>
    <input name="widget.name.count" type="hidden" value="3">
  </body>
</html>
```

4.12.2 MultiWidget Dict integration tests

Checking components on the highest possible level.

```
>>> from datetime import date
>>> from z3c.form import form
>>> from z3c.form import field
>>> from z3c.form import testing
```

```
>>> request = testing.TestRequest()
```

```
>>> class EForm(form.EditForm):
...     form.extends(form.EditForm)
...     fields = field.Fields(
...         testing.IMultiWidgetDictIntegration).omit('dictOfObject')
```

Our single content object:

```
>>> obj = testing.MultiWidgetDictIntegration()
```

We recreate the form each time, to stay as close as possible. In real life the form gets instantiated and destroyed with each request.

```
>>> def getForm(request, fname=None):
...     frm = EForm(obj, request)
...     testing.addTemplate(frm, 'integration_edit.pt')
...     frm.update()
...     content = frm.render()
...     if fname is not None:
...         testing.saveHtml(content, fname)
...     return content
```

Empty

All blank and empty values:

```
>>> content = getForm(request, 'MultiWidget_dict_edit_empty.html')
```

```
>>> print(testing.plainText(content))
DictOfInt label

[Add]
DictOfBool label

[Add]
DictOfChoice label

[Add]
DictOfTextLine label

[Add]
DictOfDate label

[Add]
[Apply]
```

Some valid default values

```
>>> obj.dictOfInt = {-101: -100, -1:1, 101:100}
>>> obj.dictOfBool = {True: False, False: True}
>>> obj.dictOfChoice = {'key1': 'three', 'key3': 'two'}
>>> obj.dictOfTextLine = {'textkey1': 'some text one',
...     'textkey2': 'some txt two'}
>>> obj.dictOfDate = {
...     date(2011, 1, 15): date(2014, 6, 20),
...     date(2012, 2, 20): date(2013, 5, 19)}
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: True, True: False}
  dictOfChoice: {'key1': 'three', 'key3': 'two'}
  dictOfDate: {datetime.date(2011, 1, 15): datetime.date(2014, 6, 20),
 datetime.date(2012, 2, 20): datetime.date(2013, 5, 19)}
  dictOfInt: {-101: -100, -1: 1, 101: 100}
  dictOfTextLine: {'textkey1': 'some text one', 'textkey2': 'some txt two'}>
```

```
>>> content = getForm(request, 'MultiWidget_dict_edit_simple.html')
```

```
>>> print(testing.plainText(content))
DictOfInt label

Int key *
[-1]
Int label *
[ ]
[1]
Int key *
[-101]
Int label *
[ ]
[-100]
Int key *
[101]
Int label *
[ ]
[100]
[Add]
[Remove selected]
DictOfBool label

Bool key *
( ) yes (O) no
Bool label *
[ ]
(O) yes ( ) no
Bool key *
(O) yes ( ) no
Bool label *
```

(continues on next page)

(continued from previous page)

```
[ ]
( ) yes (O) no
[Add]
[Remove selected]
DictOfChoice label

Choice key *
[key1]
Choice label *
[ ]
[three]
Choice key *
[key3]
Choice label *
[ ]
[two]
[Add]
[Remove selected]
DictOfTextLine label

TextLine key *
[textkey1]
TextLine label *
[ ]
[some text one]
TextLine key *
[textkey2]
TextLine label *
[ ]
[some txt two]
[Add]
[Remove selected]
DictOfDate label

Date key *
[11/01/15]
Date label *
[ ]
[14/06/20]
Date key *
[12/02/20]
Date label *
[ ]
[13/05/19]
[Add]
[Remove selected]
[Apply]
```

dictOfInt

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfInt.key.2'] = 'foobar'
>>> submit['form.widgets.dictOfInt.2'] = 'foobar'
```

```
>>> submit['form.widgets.dictOfInt.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get "The entered value is not a valid integer literal." for "foobar" and a new input.

```
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_int.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfInt"]'))
DictOfInt label

Int key *

[-1]

Int label *

[ ]
[1]
Int key *

[-101]

Int label *

[ ]
[-100]
Int key *

The entered value is not a valid integer literal.
[foobar]

Int label *

The entered value is not a valid integer literal.
[ ]
[foobar]
Int key *

[]

Int label *

[ ]
[]
```

(continues on next page)

(continued from previous page)

```
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_int2.html')
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-dictOfInt"]//div[@class="error"]'))
Required input is missing.
Required input is missing.
The entered value is not a valid integer literal.
The entered value is not a valid integer literal.
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfInt.1.remove'] = '1'
>>> submit['form.widgets.dictOfInt.3.remove'] = '1'
>>> submit['form.widgets.dictOfInt.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_remove_int.html')
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-dictOfInt"]'))
DictOfInt label

Int key *

Required input is missing.
[]

Int label *

Required input is missing.
[ ]
[]
Int key *

[-101]

Int label *

[ ]
[-100]
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: True, True: False}
  dictOfChoice: {'key1': 'three', 'key3': 'two'}
```

(continues on next page)

(continued from previous page)

```
 dictOfDate: {datetime.date(2011, 1, 15): datetime.date(2014, 6, 20),
datetime.date(2012, 2, 20): datetime.date(2013, 5, 19)}
 dictOfInt: {-101: -100, -1: 1, 101: 100}
 dictOfTextLine: {'textkey1': 'some text one', 'textkey2': 'some txt two'}>
```

dictOfBool

Add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfBool.buttons.add'] = 'Add'
>>> request = testing.TestRequest(form=submit)
```

Important is that we get a new input.

```
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_bool.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfBool"]'))
DictOfBool label

Bool key *

( ) yes (O) no

Bool label *

[ ]
(O) yes ( ) no
Bool key *

(O) yes ( ) no

Bool label *

[ ]
( ) yes (O) no
Bool key *

( ) yes ( ) no

Bool label *

[ ]
( ) yes ( ) no
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
```

(continues on next page)

(continued from previous page)

```
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_bool2.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfBool"]//div[@class="error"]'))
Required input is missing.
Required input is missing.
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfBool.1.remove'] = '1'
>>> submit['form.widgets.dictOfBool.2.remove'] = '1'
>>> submit['form.widgets.dictOfBool.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_remove_bool.html')
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-dictOfBool"]'))
DictOfBool label

Bool key *

Required input is missing.
( ) yes ( ) no

Bool label *

Required input is missing.
[ ]
( ) yes ( ) no
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: True, True: False}
  dictOfChoice: {'key1': 'three', 'key3': 'two'}
  dictOfDate: {datetime.date(2011, 1, 15): datetime.date(2014, 6, 20),
 datetime.date(2012, 2, 20): datetime.date(2013, 5, 19)}
  dictOfInt: {-101: -100, -1: 1, 101: 100}
  dictOfTextLine: {'textkey1': 'some text one', 'textkey2': 'some txt two'}>
```

dictOfChoice

Add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfChoice.buttons.add'] = 'Add'
>>> request = testing.TestRequest(form=submit)
```

Important is that we get a new input.

```
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_choice.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfChoice"]'))
DictOfChoice label

Choice key *

[key1]

Choice label *

[ ]
[three]
Choice key *

[key3]

Choice label *

[ ]
[two]
Choice key *

[[    ]]

Choice label *

[ ]
[[    ]]
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_choice2.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfChoice"]//div[@class="error"]'))
Duplicate key
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfChoice.0.remove'] = '1'
>>> submit['form.widgets.dictOfChoice.1.remove'] = '1'
>>> submit['form.widgets.dictOfChoice.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_remove_choice.html')
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-dictOfChoice"]'))
DictOfChoice label
```

(continues on next page)

(continued from previous page)

```
Choice key *

[key3]

Choice label *

[ ]
[two]
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: True, True: False}
  dictOfChoice: {'key1': 'three', 'key3': 'two'}
  dictOfDate: {datetime.date(2011, 1, 15): datetime.date(2014, 6, 20),
 datetime.date(2012, 2, 20): datetime.date(2013, 5, 19)}
  dictOfInt: {-101: -100, -1: 1, 101: 100}
  dictOfTextLine: {'textkey1': 'some text one', 'textkey2': 'some txt two'}>
```

dictOfTextLine

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfTextLine.key.0'] = 'foo\nbar'
>>> submit['form.widgets.dictOfTextLine.0'] = 'foo\nbar'
```

```
>>> submit['form.widgets.dictOfTextLine.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get “Constraint not satisfied” for “foonbar” and a new input.

```
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_textline.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfTextLine"]'))
DictOfTextLine label

TextLine key *

Constraint not satisfied
[foo
bar]

TextLine label *

Constraint not satisfied
[ ]
[foo
```

(continues on next page)

(continued from previous page)

```
bar]
TextLine key *

[textkey2]

TextLine label *

[ ]
[some txt two]
TextLine key *

[]

TextLine label *

[ ]
[]
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_textline2.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfTextLine"]//div[@class="error"]'))
Required input is missing.
Required input is missing.
Constraint not satisfied
Constraint not satisfied
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfTextLine.2.remove'] = '1'
>>> submit['form.widgets.dictOfTextLine.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_remove_textline.html')
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-dictOfTextLine"]'))
DictOfTextLine label

TextLine key *

Required input is missing.
[]

TextLine label *

Required input is missing.
[ ]
```

(continues on next page)

(continued from previous page)

```
[]
TextLine key *

Constraint not satisfied
[foo
bar]

TextLine label *

Constraint not satisfied
[ ]
[foo
bar]
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: True, True: False}
  dictOfChoice: {'key1': 'three', 'key3': 'two'}
  dictOfDate: {datetime.date(2011, 1, 15): datetime.date(2014, 6, 20),
 datetime.date(2012, 2, 20): datetime.date(2013, 5, 19)}
  dictOfInt: {-101: -100, -1: 1, 101: 100}
  dictOfTextLine: {'textkey1': 'some text one', 'textkey2': 'some txt two'}>
```

dictOfDate

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfDate.key.0'] = 'foobar'
>>> submit['form.widgets.dictOfDate.0'] = 'foobar'
```

```
>>> submit['form.widgets.dictOfDate.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get “The entered value is not a valid integer literal.” for “foobar” and a new input.

```
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_date.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfDate"]'))
DictOfDate label

Date key *

[12/02/20]

Date label *
```

(continues on next page)

(continued from previous page)

```
[ ]
[13/05/19]
Date key *

The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]

Date label *

The datetime string did not match the pattern 'yy/MM/dd'.
[ ]
[foobar]
Date key *

[]

Date label *

[ ]
[]
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_date2.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfDate"]//div[@class="error"]'))
Required input is missing.
Required input is missing.
The datetime string did not match the pattern 'yy/MM/dd'.
The datetime string did not match the pattern 'yy/MM/dd'.
```

And fill in a valid value:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfDate.key.0'] = '14/05/12'
>>> submit['form.widgets.dictOfDate.0'] = '14/06/21'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_submit_date3.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-dictOfDate"]'))
DictOfDate label

Date key *

[12/02/20]

Date label *
```

(continues on next page)

(continued from previous page)

```
[ ]
[13/05/19]
Date key *

[14/05/12]

Date label *

[ ]
[14/06/21]
Date key *

The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]

Date label *

The datetime string did not match the pattern 'yy/MM/dd'.
[ ]
[foobar]
[Add]
[Remove selected]
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfDate.1.remove'] = '1'
>>> submit['form.widgets.dictOfDate.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_remove_date.html')
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-dictOfDate"]'))
DictOfDate label

Date key *

[12/02/20]

Date label *

[ ]
[13/05/19]
Date key *

The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]

Date label *

The datetime string did not match the pattern 'yy/MM/dd'.
[ ]
[foobar]
```

(continues on next page)

(continued from previous page)

```
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: True, True: False}
  dictOfChoice: {'key1': 'three', 'key3': 'two'}
  dictOfDate: {datetime.date(2011, 1, 15): datetime.date(2014, 6, 20),
 datetime.date(2012, 2, 20): datetime.date(2013, 5, 19)}
  dictOfInt: {-101: -100, -1: 1, 101: 100}
  dictOfTextLine: {'textkey1': 'some text one', 'textkey2': 'some txt two'}>
```

And apply

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content))
There were some errors.
* DictOfInt label: Wrong contained type
* DictOfBool label: Wrong contained type
* DictOfTextLine label: Constraint not satisfied
* DictOfDate label: The datetime string did not match the pattern 'yy/MM/dd'.
...
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: True, True: False}
  dictOfChoice: {'key1': 'three', 'key3': 'two'}
  dictOfDate: {datetime.date(2011, 1, 15): datetime.date(2014, 6, 20),
 datetime.date(2012, 2, 20): datetime.date(2013, 5, 19)}
  dictOfInt: {-101: -100, -1: 1, 101: 100}
  dictOfTextLine: {'textkey1': 'some text one', 'textkey2': 'some txt two'}>
```

Let's fix the values

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfInt.key.1'] = '42'
>>> submit['form.widgets.dictOfInt.1'] = '43'
>>> submit['form.widgets.dictOfTextLine.0.remove'] = '1'
>>> submit['form.widgets.dictOfTextLine.buttons.remove'] = 'Remove selected'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
```

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfTextLine.key.0'] = 'lorem ipsum'
>>> submit['form.widgets.dictOfTextLine.0'] = 'ipsum lorem'
>>> submit['form.widgets.dictOfDate.key.1'] = '14/06/25'
```

(continues on next page)

(continued from previous page)

```
>>> submit['form.widgets.dictOfDate.1'] = '14/07/28'
>>> submit['form.widgets.dictOfInt.key.0'] = '-101'
>>> submit['form.widgets.dictOfInt.0'] = '-100'
>>> submit['form.widgets.dictOfBool.key.0'] = 'false'
>>> submit['form.widgets.dictOfBool.0'] = 'true'
```

```
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_dict_edit_fixit.html')
>>> print(testing.plainText(content))
Data successfully updated.
...
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: True}
  dictOfChoice: {'key3': 'two'}
  dictOfDate: {datetime.date(2012, 2, 20): datetime.date(2013, 5, 19),
 datetime.date(2014, 6, 25): datetime.date(2014, 7, 28)}
  dictOfInt: {-101: -100, 42: 43}
  dictOfTextLine: {'lorem ipsum': 'ipsum lorem'}>
```

Twisting some keys

Change key values, item values must stick to the new values.

```
>>> obj.dictOfInt = {-101: -100, -1:1, 101:100}
>>> obj.dictOfBool = {True: False, False: True}
>>> obj.dictOfChoice = {'key1': 'three', 'key3': 'two'}
>>> obj.dictOfTextLine = {'textkey1': 'some text one',
...     'textkey2': 'some txt two'}
>>> obj.dictOfDate = {
...     date(2011, 1, 15): date(2014, 6, 20),
...     date(2012, 2, 20): date(2013, 5, 19)}
```

```
>>> request = testing.TestRequest()
>>> content = getForm(request, 'MultiWidget_dict_edit_twist.html')
```

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfInt.key.2'] = '42'  # was 101:100
>>> submit['form.widgets.dictOfBool.key.0'] = 'true'  # was False:True
>>> submit['form.widgets.dictOfBool.key.1'] = 'false'  # was True:False
>>> submit['form.widgets.dictOfChoice.key.1:list'] = 'key2'  # was key3: two
>>> submit['form.widgets.dictOfChoice.key.0:list'] = 'key3'  # was key1: three
>>> submit['form.widgets.dictOfTextLine.key.1'] = 'lorem'  # was textkey2: some txt two
>>> submit['form.widgets.dictOfTextLine.1'] = 'ipsum'  # was textkey2: some txt two
>>> submit['form.widgets.dictOfTextLine.key.0'] = 'foobar'  # was textkey1: some txt one
>>> submit['form.widgets.dictOfDate.key.0'] = '14/06/25'  # 11/01/15: 14/06/20
```

```
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

```
>>> content = getForm(request, 'MultiWidget_dict_edit_twist2.html')
```

```
>>> submit = testing.getSubmitValues(content)
```

```
>>> pprint(obj)
<MultiWidgetDictIntegration
  dictOfBool: {False: False, True: True}
  dictOfChoice: {'key2': 'two', 'key3': 'three'}
  dictOfDate: {datetime.date(2012, 2, 20): datetime.date(2013, 5, 19),
 datetime.date(2014, 6, 25): datetime.date(2014, 6, 20)}
  dictOfInt: {-101: -100, -1: 1, 42: 100}
  dictOfTextLine: {'foobar': 'some text one', 'lorem': 'ipsum'}>
```

4.12.3 MultiWidget List integration tests

Checking components on the highest possible level.

```
>>> from datetime import date
>>> from z3c.form import form
>>> from z3c.form import field
>>> from z3c.form import testing
```

```
>>> request = testing.TestRequest()
```

```
>>> class EForm(form.EditForm):
...     form.extends(form.EditForm)
...     fields = field.Fields(
...         testing.IMultiWidgetListIntegration).omit(
...             'listOfChoice', 'listOfObject')
```

Our single content object:

```
>>> obj = testing.MultiWidgetListIntegration()
```

We recreate the form each time, to stay as close as possible. In real life the form gets instantiated and destroyed with each request.

```
>>> def getForm(request, fname=None):
...     frm = EForm(obj, request)
...     testing.addTemplate(frm, 'integration_edit.pt')
...     frm.update()
...     content = frm.render()
...     if fname is not None:
...         testing.saveHtml(content, fname)
...     return content
```

Empty

All blank and empty values:

```
>>> content = getForm(request, 'MultiWidget_list_edit_empty.html')
```

```
>>> print(testing.plainText(content))
ListOfInt label

[Add]
ListOfBool label

[Add]
ListOfTextLine label

[Add]
ListOfDate label

[Add]
[Apply]
```

Some valid default values

```
>>> obj.listOfInt = [-100, 1, 100]
>>> obj.listOfBool = [True, False, True]
>>> obj.listOfChoice = ['two', 'three']
>>> obj.listOfTextLine = ['some text one', 'some txt two']
>>> obj.listOfDate = [date(2014, 6, 20)]
```

```
>>> pprint(obj)
<MultiWidgetListIntegration
  listOfBool: [True, False, True]
  listOfChoice: ['two', 'three']
  listOfDate: [datetime.date(2014, 6, 20)]
  listOfInt: [-100, 1, 100]
  listOfTextLine: ['some text one', 'some txt two']>
```

```
>>> content = getForm(request, 'MultiWidget_list_edit_simple.html')
```

```
>>> print(testing.plainText(content))
ListOfInt label

Int label *
[ ]
[-100]
Int label *
[ ]
[1]
Int label *
[ ]
[100]
```

(continues on next page)

(continued from previous page)

```
[Add]
[Remove selected]
ListOfBool label

Bool label *
[ ]
(O) yes ( ) no
Bool label *
[ ]
( ) yes (O) no
Bool label *
[ ]
(O) yes ( ) no
[Add]
[Remove selected]
ListOfTextLine label

TextLine label *
[ ]
[some text one]
TextLine label *
[ ]
[some txt two]
[Add]
[Remove selected]
ListOfDate label

Date label *
[ ]
[14/06/20]
[Add]
[Remove selected]
[Apply]
```

```
>>> pprint(obj)
<MultiWidgetListIntegration
  listOfBool: [True, False, True]
  listOfChoice: ['two', 'three']
  listOfDate: [datetime.date(2014, 6, 20)]
  listOfInt: [-100, 1, 100]
  listOfTextLine: ['some text one', 'some txt two']>
```

listOfInt

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfInt.1'] = 'foobar'
```

```
>>> submit['form.widgets.listOfInt.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get "The entered value is not a valid integer literal." for "foobar" and a new input.

```
>>> content = getForm(request, 'MultiWidget_list_edit_submit_int.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-listOfInt"]'))
ListOfInt label

Int label *
[ ]
[-100]
Int label *
The entered value is not a valid integer literal.
[ ]
[foobar]
Int label *
[ ]
[100]
Int label *
[ ]
[]
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-listOfInt"]//div[@class="error"]'))
The entered value is not a valid integer literal.
Required input is missing.
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfInt.1.remove'] = '1'
>>> submit['form.widgets.listOfInt.2.remove'] = '1'
>>> submit['form.widgets.listOfInt.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_list_edit_remove_int.html')
>>> print(testing.plainText(content,
```

(continues on next page)

(continued from previous page)

```
...     './/div[@id="row-form-widgets-listOfInt"]'))
ListOfInt label

Int label *

[ ]
[-100]
Int label *

Required input is missing.
[ ]
[]
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetListIntegration
  listOfBool: [True, False, True]
  listOfChoice: ['two', 'three']
  listOfDate: [datetime.date(2014, 6, 20)]
  listOfInt: [-100, 1, 100]
  listOfTextLine: ['some text one', 'some txt two']>
```

listOfBool

Add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfBool.buttons.add'] = 'Add'
>>> request = testing.TestRequest(form=submit)
```

Important is that we get a new input.

```
>>> content = getForm(request, 'MultiWidget_list_edit_submit_bool.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-listOfBool"]'))
ListOfBool label

Bool label *

[ ]
(O) yes ( ) no
Bool label *

[ ]
( ) yes (O) no
Bool label *

[ ]
(O) yes ( ) no
Bool label *
```

(continues on next page)

(continued from previous page)

```
[ ]
( ) yes ( ) no
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-listOfBool"]//div[@class="error"]'))
Required input is missing.
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfBool.1.remove'] = '1'
>>> submit['form.widgets.listOfBool.2.remove'] = '1'
>>> submit['form.widgets.listOfBool.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_list_edit_remove_bool.html')
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-listOfBool"]'))
ListOfBool label

Bool label *

[ ]
(O) yes ( ) no
Bool label *

Required input is missing.
[ ]
( ) yes ( ) no
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetListIntegration
  listOfBool: [True, False, True]
  listOfChoice: ['two', 'three']
  listOfDate: [datetime.date(2014, 6, 20)]
  listOfInt: [-100, 1, 100]
  listOfTextLine: ['some text one', 'some txt two']>
```

listOfTextLine

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfTextLine.1'] = 'foo\nbar'
```

```
>>> submit['form.widgets.listOfTextLine.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get "Constraint not satisfied" for "foonbar" and a new input.

```
>>> content = getForm(request, 'MultiWidget_list_edit_submit_textline.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-listOfTextLine"]'))
ListOfTextLine label

TextLine label *

[ ]
[some text one]
TextLine label *

Constraint not satisfied
[ ]
[foo
bar]
TextLine label *

[ ]
[]
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-listOfTextLine"]//div[@class="error"]'))
Constraint not satisfied
Required input is missing.
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfTextLine.0.remove'] = '1'
>>> submit['form.widgets.listOfTextLine.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_list_edit_remove_textline.html')
>>> print(testing.plainText(content,
```

(continues on next page)

(continued from previous page)

```
...     './/div[@id="row-form-widgets-listOfTextLine"]'))
ListOfTextLine label

TextLine label *

Constraint not satisfied
[ ]
[foo
bar]
TextLine label *

Required input is missing.
[ ]
[]
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetListIntegration
  listOfBool: [True, False, True]
  listOfChoice: ['two', 'three']
  listOfDate: [datetime.date(2014, 6, 20)]
  listOfInt: [-100, 1, 100]
  listOfTextLine: ['some text one', 'some txt two']>
```

listOfDate

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfDate.0'] = 'foobar'
```

```
>>> submit['form.widgets.listOfDate.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get "The datetime string did not match the pattern" for "foobar" and a new input.

```
>>> content = getForm(request, 'MultiWidget_list_edit_submit_date.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-listOfDate"]'))
ListOfDate label

Date label *

The datetime string did not match the pattern 'yy/MM/dd'.
[ ]
[foobar]
Date label *
```

(continues on next page)

(continued from previous page)

```
[ ]
[]
[Add]
[Remove selected]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-listOfDate"]//div[@class="error"]'))
The datetime string did not match the pattern 'yy/MM/dd'.
Required input is missing.
```

Add one more field:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfDate.buttons.add'] = 'Add'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
```

And fill in a valid value:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfDate.2'] = '14/06/21'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_list_edit_submit_date2.html')
>>> print(testing.plainText(content,
...     './/form/div[@id="row-form-widgets-listOfDate"]'))
ListOfDate label

Date label *

The datetime string did not match the pattern 'yy/MM/dd'.
[ ]
[foobar]
Date label *

Required input is missing.
[ ]
[]
Date label *

[ ]
[14/06/21]
[Add]
[Remove selected]
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfDate.2.remove'] = '1'
```

(continues on next page)

(continued from previous page)

```
>>> submit['form.widgets.listOfDate.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'MultiWidget_list_edit_remove_date.html')
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-listOfDate"]'))
ListOfDate label

Date label *

The datetime string did not match the pattern 'yy/MM/dd'.
[ ]
[foobar]
Date label *

Required input is missing.
[ ]
[]
[Add]
[Remove selected]
```

```
>>> pprint(obj)
<MultiWidgetListIntegration
  listOfBool: [True, False, True]
  listOfChoice: ['two', 'three']
  listOfDate: [datetime.date(2014, 6, 20)]
  listOfInt: [-100, 1, 100]
  listOfTextLine: ['some text one', 'some txt two']>
```

And apply

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content))
There were some errors.
* ListOfInt label: Wrong contained type
* ListOfBool label: Wrong contained type
* ListOfTextLine label: Constraint not satisfied
* ListOfDate label: The datetime string did not match the pattern 'yy/MM/dd'.
...
```

```
>>> pprint(obj)
<MultiWidgetListIntegration
  listOfBool: [True, False, True]
  listOfChoice: ['two', 'three']
  listOfDate: [datetime.date(2014, 6, 20)]
  listOfInt: [-100, 1, 100]
  listOfTextLine: ['some text one', 'some txt two']>
```

Let's fix the values

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfInt.1'] = '42'
>>> submit['form.widgets.listOfBool.1'] = 'false'
>>> submit['form.widgets.listOfTextLine.0'] = 'ipsum lorem'
>>> submit['form.widgets.listOfTextLine.1'] = 'lorem ipsum'
>>> submit['form.widgets.listOfDate.0'] = '14/06/25'
>>> submit['form.widgets.listOfDate.1'] = '14/06/24'
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content))
Data successfully updated.
...
```

```
>>> pprint(obj)
<MultiWidgetListIntegration
  listOfBool: [True, False]
  listOfChoice: ['two', 'three']
  listOfDate: [datetime.date(2014, 6, 25), datetime.date(2014, 6, 24)]
  listOfInt: [-100, 42]
  listOfTextLine: ['ipsum lorem', 'lorem ipsum']>
```

4.13 Object Widget

The *ObjectWidget* widget is about rendering a schema's fields as a single widget.

4.13.1 ObjectWidget

The *ObjectWidget* widget is about rendering a schema's fields as a single widget.

There are way too many preconditions to exercise the *ObjectWidget* as it relies heavily on the from and widget framework. It renders the sub-widgets in a sub-form.

In order to not overwhelm you with our set of well-chosen defaults, all the default component registrations have been made prior to doing those examples:

```
>>> from z3c.form import testing
>>> testing.setupFormDefaults()
```

```
>>> import zope.component
```

```
>>> from z3c.form import datamanager
>>> zope.component.provideAdapter(datamanager.DictionaryField)
```

```
>>> from z3c.form.testing import IMySubObject
>>> from z3c.form.testing import IMySecond
>>> from z3c.form.testing import MySubObject
>>> from z3c.form.testing import MySecond
```

```
>>> import z3c.form.object
>>> zope.component.provideAdapter(z3c.form.object.ObjectConverter)
```

```
>>> from z3c.form.error import MultipleErrorViewSnippet
>>> zope.component.provideAdapter(MultipleErrorViewSnippet)
```

As for all widgets, the objectwidget must provide the new *IWidget* interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> import z3c.form.browser.object
```

```
>>> verifyClass(interfaces.IWidget, z3c.form.browser.object.ObjectWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = z3c.form.browser.object.ObjectWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

Also, stand-alone widgets need to ignore the context:

```
>>> widget.ignoreContext = True
```

There is no life for ObjectWidget without a schema to render:

```
>>> widget.update()
Traceback (most recent call last):
...
ValueError: <ObjectWidget 'widget.name'> .field is None,
            that's a blocking point
```

This schema is specified by the field:

```
>>> from z3c.form.widget import FieldWidget
>>> from z3c.form.testing import IMySubObject
>>> import zope.schema
>>> field = zope.schema.Object(
...     __name__='subobject',
...     title='my object widget',
...     schema=IMySubObject)
```

Apply the field nicely with the helper:

```
>>> widget = FieldWidget(field, widget)
```

We also need to register the templates for the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('object_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IObjectWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('object_display.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IObjectWidget),
...     IPageTemplate, name=interfaces.DISPLAY_MODE)
```

We can now render the widget:

```
>>> widget.update()
>>> print(widget.render())
<html>
  <body>
    <div class="object-widget required">
      <div class="label">
        <label for="subobject-widgets-foofield">
          <span>My foo field</span>
          <span class="required">*</span>
        </label>
      </div>
      <div class="widget">
        <input class="text-widget required int-field"
        id="subobject-widgets-foofield"
        name="subobject.widgets.foofield"
        type="text" value="1,111">
      </div>
      <div class="label">
        <label for="subobject-widgets-barfield">
          <span>My dear bar</span>
        </label>
      </div>
      <div class="widget">
        <input class="text-widget int-field"
        id="subobject-widgets-barfield"
        name="subobject.widgets.barfield"
        type="text" value="2,222">
      </div>
      <input name="subobject-empty-marker" type="hidden" value="1">
    </div>
  </body>
</html>
```

As you see all sort of default values are rendered.

Let's provide a more meaningful value:

```
>>> from z3c.form.testing import MySubObject
>>> v = MySubObject()
>>> v.foofield = 42
>>> v.barfield = 666
>>> v.__marker__ = "ThisMustStayTheSame"
```

```
>>> widget.ignoreContext = False
>>> widget.value = dict(foofield='42', barfield='666')
```

```
>>> widget.update()
```

```
>>> print(widget.render())
<html>
  <body>
    <div class="object-widget required">
      <div class="label">
        <label for="subobject-widgets-foofield">
          <span>My foo field</span>
          <span class="required">*</span>
        </label>
      </div>
      <div class="widget">
        <input class="text-widget required int-field"
        id="subobject-widgets-foofield"
        name="subobject.widgets.foofield"
        type="text" value="42">
      </div>
      <div class="label">
        <label for="subobject-widgets-barfield">
          <span>My dear bar</span>
        </label>
      </div>
      <div class="widget">
        <input class="text-widget int-field"
        id="subobject-widgets-barfield"
        name="subobject.widgets.barfield"
        type="text" value="666">
      </div>
      <input name="subobject-empty-marker" type="hidden" value="1">
    </div>
  </body>
</html>
```

The widget's value is NO_VALUE until it gets a request:

```
>>> widget.value
<NO_VALUE>
```

Let's fill in some values via the request:

```
>>> widget.request = TestRequest(form={'subobject.widgets.foofield':'2',
...                                    'subobject.widgets.barfield':'999',
```

(continues on next page)

(continued from previous page)

```
...                                     'subobject-empty-marker':'1'})
>>> widget.update()
>>> print(widget.render())
<html>
  <body>
    <div class="object-widget required">
      <div class="label">
        <label for="subobject-widgets-foofield">
          <span>My foo field</span>
          <span class="required">*</span>
        </label>
      </div>
      <div class="widget">
        <input class="text-widget required int-field"
        id="subobject-widgets-foofield"
        name="subobject.widgets.foofield"
        type="text" value="2">
      </div>
      <div class="label">
        <label for="subobject-widgets-barfield">
          <span>My dear bar</span>
        </label>
      </div>
      <div class="widget">
        <input class="text-widget int-field"
        id="subobject-widgets-barfield"
        name="subobject.widgets.barfield"
        type="text" value="999">
      </div>
      <input name="subobject-empty-marker" type="hidden" value="1">
    </div>
  </body>
</html>
```

Widget value comes from the request:

```
>>> wv = widget.value
>>> pprint(wv)
{'barfield': '999', 'foofield': '2'}
```

But our object will not be modified, since there was no “apply”-like control.

```
>>> v
<z3c.form.testing.MySubObject object at ...>
>>> v.foofield
42
>>> v.barfield
666
```

The marker must stay (we have to modify the same object):

```
>>> v.__marker__
'ThisMustStayTheSame'
```

```
>>> converter = interfaces.IDataConverter(widget)
```

```
>>> value = converter.toFieldValue(wv)
Traceback (most recent call last):
...
RuntimeError: No IObjectFactory adapter registered for z3c.form.testing.IMySubObject
```

We have to register object factory adapters to allow the objectwidget to create objects:

```
>>> from z3c.form.object import registerFactoryAdapter
>>> registerFactoryAdapter(IMySubObject, MySubObject)
>>> registerFactoryAdapter(IMySecond, MySecond)
```

```
>>> value = converter.toFieldValue(wv)
>>> value
<z3c.form.testing.MySubObject object at ...>
>>> value.foofield
2
>>> value.barfield
999
```

This is a different object:

```
>>> value.__marker__
Traceback (most recent call last):
...
AttributeError: 'MySubObject' object has no attribute '__marker__'
```

Setting missing values on the widget works too:

```
>>> widget.value = converter.toWidgetValue(field.missing_value)
```

```
>>> widget.update()
```

Default values get rendered:

```
>>> print(widget.render())
<div class="object-widget required">
  <div class="label">
    <label for="subobject-widgets-foofield">
      <span>My foo field</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <input id="subobject-widgets-foofield"
           name="subobject.widgets.foofield"
           class="text-widget required int-field" value="2" type="text" />
  </div>
  <div class="label">
    <label for="subobject-widgets-barfield">
      <span>My dear bar</span>
    </label>
```

(continues on next page)

(continued from previous page)

```
  </div>
  <div class="widget">
    <input id="subobject-widgets-barfield"
           name="subobject.widgets.barfield"
           class="text-widget int-field" value="999" type="text" />
  </div>
  <input name="subobject-empty-marker" type="hidden" value="1" />
</div>
```

But on the return we get default values back:

```
>>> pprint(widget.value)
{'barfield': '999', 'foofield': '2'}
```

```
>>> value = converter.toFieldValue(widget.value)
>>> value
<z3c.form.testing.MySubObject object at ...>
```

HMMMM.... do we have to test error handling here? I'm tempted to leave it out as no widgets seem to do this.

#Error handling is next. Let's use the value "bad" (an invalid integer literal) #as input for our internal (sub) widget. # # >>> widget.request = TestRequest(form={'subobject.widgets.foofield':'55', # ... 'subobject.widgets.barfield':'bad', # ... 'subobject-empty-marker':'1'}) # # # >>> widget.update() # >>> print(widget.render()) # <html> # <body> # # # <label for="subobject-widgets-foofield"> # My foo field # * # </label> # # # <input class="text-widget required int-field" # id="subobject-widgets-foofield" # name="subobject.widgets.foofield" # type="text" value="55"> # # # <label for="subobject-widgets-barfield"> # My dear bar # </label> # # The entered value is not a valid integer literal. # # <input class="text-widget int-field" # id="subobject-widgets-barfield" # name="subobject.widgets.barfield" # type="text" value="bad"> # # <input name="subobject-empty-marker" type="hidden" value="1"> # # </body> # </html> # #Getting the widget value raises the widget errors: # # >>> widget.value # Traceback (most recent call last): # ... # MultipleErrors # #Our object still must retain the old values: # # >>> v # <z3c.form.testing.MySubObject object at ...> # >>> v.foofield # 42 # >>> v.barfield # 666 # >>> v.__marker__ # 'ThisMustStayTheSame'

In forms

Do all that fun in add and edit forms too:

We have to provide an adapter first:

```
>>> zope.component.provideAdapter(z3c.form.browser.object.ObjectFieldWidget)
```

Forms and our objectwidget fire events on add and edit, setup a subscriber for those:

```
>>> eventlog = []
>>> import zope.lifecycleevent
>>> @zope.component.adapter(zope.lifecycleevent.ObjectModifiedEvent)
... def logEvent(event):
...     eventlog.append(event)
>>> zope.component.provideHandler(logEvent)
>>> @zope.component.adapter(zope.lifecycleevent.ObjectCreatedEvent)
... def logEvent2(event):
```

(continues on next page)

(continued from previous page)

```
...     eventlog.append(event)
>>> zope.component.provideHandler(logEvent2)
```

```
>>> def printEvents():
...     for event in eventlog:
...         print(str(event))
...         if isinstance(event, zope.lifecycleevent.ObjectModifiedEvent):
...             for attr in event.descriptions:
...                 print(attr.interface)
...                 print(sorted(attr.attributes))
```

We define an interface containing a subobject, and an addform for it:

```
>>> from z3c.form import form, field
>>> from z3c.form.testing import MyObject, IMyObject
```

Note, that creating an object will print(some information about it:)

```
>>> class MyAddForm(form.AddForm):
...     fields = field.Fields(IMyObject)
...     def create(self, data):
...         print("MyAddForm.create")
...         pprint(data)
...         return MyObject(**data)
...     def add(self, obj):
...         self.context[obj.name] = obj
...     def nextURL(self):
...         pass
```

We create the form and try to update it:

```
>>> request = TestRequest()
>>> myaddform =  MyAddForm(root, request)
```

```
>>> myaddform.update()
```

As usual, the form contains a widget manager with the expected widget

```
>>> list(myaddform.widgets.keys())
['subobject', 'name']
>>> list(myaddform.widgets.values())
[<ObjectWidget 'form.widgets.subobject'>, <TextWidget 'form.widgets.name'>]
```

The widget has sub-widgets:

```
>>> list(myaddform.widgets['subobject'].widgets.keys())
['foofield', 'barfield']
```

If we want to render the addform, we must give it a template:

```
>>> import os
>>> from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile
>>> from zope.browserpage.viewpagetemplatefile import BoundPageTemplate
```

(continues on next page)

(continued from previous page)

```
>>> from z3c.form import tests
>>> def addTemplate(form):
...     form.template = BoundPageTemplate(
...         ViewPageTemplateFile(
...             'simple_edit.pt', os.path.dirname(tests.__file__)), form)
>>> addTemplate(myaddform)
```

Now rendering the addform renders the subform as well:

```
>>> print(myaddform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-subobject">my object</label>
        <div class="object-widget required">
          <div class="label">
            <label for="form-widgets-subobject-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field"
            id="form-widgets-subobject-widgets-foofield"
            name="form.widgets.subobject.widgets.foofield"
            type="text" value="1,111">
          </div>
          <div class="label">
            <label for="form-widgets-subobject-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field"
            id="form-widgets-subobject-widgets-barfield"
            name="form.widgets.subobject.widgets.barfield"
            type="text" value="2,222">
          </div>
          <input name="form.widgets.subobject-empty-marker" type="hidden"
          value="1">
        </div>
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field" id="form-widgets-name" name=
↪"form.widgets.name" type="text" value="">
      </div>
      <div class="action">
        <input class="submit-widget button-field" id="form-buttons-add"
        name="form.buttons.add" type="submit" value="Add">
      </div>
```

(continues on next page)

(continued from previous page)

```
    </form>
  </body>
</html>
```

We don't have the object (yet) in the root:

```
>>> root['first']
Traceback (most recent call last):
...
KeyError: 'first'
```

Let's try to add an object:

```
>>> request = TestRequest(form={
...     'form.widgets.subobject.widgets.foofield':'66',
...     'form.widgets.subobject.widgets.barfield':'99',
...     'form.widgets.name':'first',
...     'form.widgets.subobject-empty-marker':'1',
...     'form.buttons.add':'Add'})
>>> myaddform.request = request
```

```
>>> myaddform.update()
MyAddForm.create
{'name': 'first',
 'subobject': <z3c.form.testing.MySubObject object at ...>}
```

Wow, it got added:

```
>>> root['first']
<z3c.form.testing.MyObject object at ...>
```

```
>>> root['first'].subobject
<z3c.form.testing.MySubObject object at ...>
```

Field values need to be right:

```
>>> root['first'].subobject.foofield
66
>>> root['first'].subobject.barfield
99
```

Let's see our event log:

```
>>> len(eventlog)
4
```

```
>>> printEvents()
<zope...ObjectCreatedEvent object at ...>
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMySubObject
['barfield', 'foofield']
<zope...ObjectCreatedEvent object at ...>
<zope...contained.ContainerModifiedEvent object at ...>
```

```
>>> eventlog=[]
```

Let's try to edit that newly added object:

```
>>> class MyEditForm(form.EditForm):
...     fields = field.Fields(IMyObject)
```

```
>>> editform = MyEditForm(root['first'], TestRequest())
>>> addTemplate(editform)
>>> editform.update()
```

Watch for the widget values in the HTML:

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-subobject">my object</label>
        <div class="object-widget required">
          <div class="label">
            <label for="form-widgets-subobject-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field"
            id="form-widgets-subobject-widgets-foofield"
            name="form.widgets.subobject.widgets.foofield"
            type="text" value="66">
          </div>
          <div class="label">
            <label for="form-widgets-subobject-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field"
            id="form-widgets-subobject-widgets-barfield"
            name="form.widgets.subobject.widgets.barfield"
            type="text" value="99">
          </div>
          <input name="form.widgets.subobject-empty-marker" type="hidden"
          value="1">
        </div>
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field" id="form-widgets-name"
        name="form.widgets.name" type="text" value="first">
      </div>
      <div class="action">
```

(continues on next page)

(continued from previous page)

```
        <input class="submit-widget button-field" id="form-buttons-apply"
        name="form.buttons.apply" type="submit" value="Apply">
      </div>
    </form>
  </body>
</html>
```

Let's modify the values:

```
>>> request = TestRequest(form={
...     'form.widgets.subobject.widgets.foofield':'43',
...     'form.widgets.subobject.widgets.barfield':'55',
...     'form.widgets.name':'first',
...     'form.widgets.subobject-empty-marker':'1',
...     'form.buttons.apply':'Apply'})
```

They are still the same:

```
>>> root['first'].subobject.foofield
66
>>> root['first'].subobject.barfield
99
```

```
>>> editform.request = request
>>> editform.update()
```

Until we have updated the form:

```
>>> root['first'].subobject.foofield
43
>>> root['first'].subobject.barfield
55
```

Let's see our event log:

```
>>> len(eventlog)
2
```

```
>>> printEvents()
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMySubObject
['barfield', 'foofield']
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMyObject
['subobject']
```

```
>>> eventlog=[]
```

After the update the form says that the values got updated and renders the new values:

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
```

(continues on next page)

(continued from previous page)

```
  <body>
    <i>Data successfully updated.</i>
    <form action=".">
      <div class="row">
        <label for="form-widgets-subobject">my object</label>
        <div class="object-widget required">
          <div class="label">
            <label for="form-widgets-subobject-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field"
            id="form-widgets-subobject-widgets-foofield"
            name="form.widgets.subobject.widgets.foofield"
            type="text" value="43">
          </div>
          <div class="label">
            <label for="form-widgets-subobject-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field"
            id="form-widgets-subobject-widgets-barfield"
            name="form.widgets.subobject.widgets.barfield"
            type="text" value="55">
          </div>
          <input name="form.widgets.subobject-empty-marker" type="hidden"
          value="1">
        </div>
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field" id="form-widgets-name"
        name="form.widgets.name" type="text" value="first">
      </div>
      <div class="action">
        <input class="submit-widget button-field" id="form-buttons-apply"
        name="form.buttons.apply" type="submit" value="Apply">
      </div>
    </form>
  </body>
</html>
```

Let's see if the widget keeps the old object on editing:

We add a special property to keep track of the object:

```
>>> root['first'].__marker__ = "ThisMustStayTheSame"
```

```
>>> root['first'].subobject.foofield
43
>>> root['first'].subobject.barfield
55
```

Let's modify the values:

```
>>> request = TestRequest(form={
...     'form.widgets.subobject.widgets.foofield':'666',
...     'form.widgets.subobject.widgets.barfield':'999',
...     'form.widgets.name':'first',
...     'form.widgets.subobject-empty-marker':'1',
...     'form.buttons.apply':'Apply'})
```

```
>>> editform.request = request
```

```
>>> editform.update()
```

Let's check what are ther esults of the update:

```
>>> root['first'].subobject.foofield
666
>>> root['first'].subobject.barfield
999
>>> root['first'].__marker__
'ThisMustStayTheSame'
```

Let's make a nasty error, by typing 'bad' instead of an integer:

```
>>> request = TestRequest(form={
...     'form.widgets.subobject.widgets.foofield':'99',
...     'form.widgets.subobject.widgets.barfield':'bad',
...     'form.widgets.name':'first',
...     'form.widgets.subobject-empty-marker':'1',
...     'form.buttons.apply':'Apply'})
```

```
>>> editform.request = request
>>> eventlog=[]
>>> editform.update()
```

Eventlog must be clean:

```
>>> len(eventlog)
0
```

Watch for the error message in the HTML: it has to appear at the field itself and at the top of the form:

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <i>There were some errors.</i>
    <ul>
      <li>
```

(continues on next page)

(continued from previous page)

```
      my object:
        <div class="error">The entered value is not a valid integer literal.</div>
      </li>
    </ul>
    <form action=".">
      <div class="row">
        <b>
          <div class="error">The entered value is not a valid integer literal.</div>
        </b>
        <label for="form-widgets-subobject">my object</label>
        <div class="object-widget required">
          <div class="label">
            <label for="form-widgets-subobject-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field"
            id="form-widgets-subobject-widgets-foofield"
            name="form.widgets.subobject.widgets.foofield"
            type="text" value="99">
          </div>
          <div class="label">
            <label for="form-widgets-subobject-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="error">The entered value is not a valid integer literal.</div>
          <div class="widget">
            <input class="text-widget int-field"
            id="form-widgets-subobject-widgets-barfield"
            name="form.widgets.subobject.widgets.barfield"
            type="text" value="bad">
          </div>
          <input name="form.widgets.subobject-empty-marker" type="hidden"
          value="1">
        </div>
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field" id="form-widgets-name"
        name="form.widgets.name" type="text" value="first">
      </div>
      <div class="action">
        <input class="submit-widget button-field" id="form-buttons-apply"
        name="form.buttons.apply" type="submit" value="Apply">
      </div>
    </form>
  </body>
</html>
```

The object values must stay at the old ones:

```
>>> root['first'].subobject.foofield
666
>>> root['first'].subobject.barfield
999
```

Let's make more errors: Now we enter 'bad' and '999999', where '999999' hits the upper limit of the field.

```
>>> request = TestRequest(form={
...     'form.widgets.subobject.widgets.foofield':'999999',
...     'form.widgets.subobject.widgets.barfield':'bad',
...     'form.widgets.name':'first',
...     'form.widgets.subobject-empty-marker':'1',
...     'form.buttons.apply':'Apply'})
```

```
>>> editform.request = request
>>> editform.update()
```

Both errors must appear at the top of the form:

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <i>There were some errors.</i>
    <ul>
      <li>
      my object:
        <div class="error">Value is too big</div>
        <div class="error">The entered value is not a valid integer literal.</div>
      </li>
    </ul>
    <form action=".">
      <div class="row">
        <b>
          <div class="error">Value is too big</div>
          <div class="error">The entered value is not a valid integer literal.</div>
        </b>
        <label for="form-widgets-subobject">my object</label>
        <div class="object-widget required">
          <div class="label">
            <label for="form-widgets-subobject-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="error">Value is too big</div>
          <div class="widget">
            <input class="text-widget required int-field"
            id="form-widgets-subobject-widgets-foofield"
            name="form.widgets.subobject.widgets.foofield"
            type="text" value="999999">
          </div>
          <div class="label">
            <label for="form-widgets-subobject-widgets-barfield">
```

(continues on next page)

(continued from previous page)

```
            <span>My dear bar</span>
          </label>
        </div>
        <div class="error">The entered value is not a valid integer literal.</div>
        <div class="widget">
          <input class="text-widget int-field"
          id="form-widgets-subobject-widgets-barfield"
          name="form.widgets.subobject.widgets.barfield"
          type="text" value="bad">
        </div>
        <input name="form.widgets.subobject-empty-marker" type="hidden"
        value="1">
      </div>
    </div>
    <div class="row">
      <label for="form-widgets-name">name</label>
      <input class="text-widget required textline-field" id="form-widgets-name"
      name="form.widgets.name" type="text" value="first">
    </div>
    <div class="action">
      <input class="submit-widget button-field" id="form-buttons-apply"
      name="form.buttons.apply" type="submit" value="Apply">
    </div>
  </form>
 </body>
</html>
```

And of course, the object values do not get modified:

```
>>> root['first'].subobject.foofield
666
>>> root['first'].subobject.barfield
999
```

Simple but often used use-case is the display form:

```
>>> editform = MyEditForm(root['first'], TestRequest())
>>> addTemplate(editform)
>>> editform.mode = interfaces.DISPLAY_MODE
>>> editform.update()
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-subobject">my object</label>
        <div class="object-widget">
          <div class="label">
            <label for="form-widgets-subobject-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
```

(continues on next page)

(continued from previous page)

```
          </div>
          <div class="widget">
            <span class="text-widget int-field"
            id="form-widgets-subobject-widgets-foofield">666</span>
          </div>
          <div class="label">
            <label for="form-widgets-subobject-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <span class="text-widget int-field"
            id="form-widgets-subobject-widgets-barfield">999</span>
          </div>
        </div>
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <span class="text-widget textline-field"
        id="form-widgets-name">first</span>
      </div>
      <div class="action">
        <input class="submit-widget button-field"
        id="form-buttons-apply" name="form.buttons.apply" type="submit" value="Apply">
      </div>
    </form>
  </body>
</html>
```

Let's see what happens in HIDDEN_MODE: (not quite sane thing, but we want to see the objectwidget rendered in hidden mode)

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('object_hidden.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IObjectWidget),
...     IPageTemplate, name=interfaces.HIDDEN_MODE)
```

```
>>> editform = MyEditForm(root['first'], TestRequest())
>>> addTemplate(editform)
>>> editform.mode = interfaces.HIDDEN_MODE
>>> editform.update()
```

Note, that the labels and the button is there because the form template for testing does/should not care about the form being hidden. What matters is that the objectwidget is rendered hidden.

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-subobject">my object</label>
        <input id="form-widgets-subobject-widgets-foofield"
```

(continues on next page)

(continued from previous page)

```
               name="form.widgets.subobject.widgets.foofield"
               value="666" class="hidden-widget" type="hidden" />
        <input id="form-widgets-subobject-widgets-barfield"
               name="form.widgets.subobject.widgets.barfield"
               value="999" class="hidden-widget" type="hidden" />
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input id="form-widgets-name" name="form.widgets.name"
               value="first" class="hidden-widget" type="hidden" />
      </div>
      <div class="action">
       <input id="form-buttons-apply" name="form.buttons.apply"
              class="submit-widget button-field" value="Apply" type="submit" />
      </div>
    </form>
  </body>
</html>
```

Editforms might use dicts as context:

```
>>> newsub = MySubObject()
>>> newsub.foofield = 78
>>> newsub.barfield = 87
```

```
>>> class MyEditFormDict(form.EditForm):
...     fields = field.Fields(IMyObject)
...     def getContent(self):
...         return {'subobject': newsub, 'name': 'blooki'}
```

```
>>> editform = MyEditFormDict(None, TestRequest())
>>> addTemplate(editform)
>>> editform.update()
```

Watch for the widget values in the HTML:

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-subobject">my object</label>
        <div class="object-widget required">
          <div class="label">
            <label for="form-widgets-subobject-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field"
            id="form-widgets-subobject-widgets-foofield"
```

(continues on next page)

(continued from previous page)

```
            name="form.widgets.subobject.widgets.foofield"
            type="text" value="78">
          </div>
          <div class="label">
            <label for="form-widgets-subobject-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field"
            id="form-widgets-subobject-widgets-barfield"
            name="form.widgets.subobject.widgets.barfield"
            type="text" value="87">
          </div>
          <input name="form.widgets.subobject-empty-marker" type="hidden"
          value="1">
        </div>
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field" id="form-widgets-name"
        name="form.widgets.name" type="text" value="blooki">
      </div>
      <div class="action">
        <input class="submit-widget button-field" id="form-buttons-apply"
        name="form.buttons.apply" type="submit" value="Apply">
      </div>
    </form>
  </body>
</html>
```

Let's modify the values:

```
>>> request = TestRequest(form={
...      'form.widgets.subobject.widgets.foofield':'43',
...      'form.widgets.subobject.widgets.barfield':'55',
...      'form.widgets.name':'first',
...      'form.widgets.subobject-empty-marker':'1',
...      'form.buttons.apply':'Apply'})
```

They are still the same:

```
>>> newsub.foofield
78
>>> newsub.barfield
87
```

Until updating the form:

```
>>> editform.request = request
>>> eventlog=[]
```

```
>>> editform.update()
```

```
>>> newsub.foofield
43
>>> newsub.barfield
55
```

```
>>> len(eventlog)
2
>>> printEvents()
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMySubObject
['barfield', 'foofield']
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMyObject
['name', 'subobject']
```

Object in an Object situation

We define an interface containing a subobject, and an addform for it:

```
>>> from z3c.form import form, field
>>> from z3c.form.testing import IMyComplexObject
```

Note, that creating an object will print some information about it:

```
>>> class MyAddForm(form.AddForm):
...     fields = field.Fields(IMyComplexObject)
...     def create(self, data):
...         print("MyAddForm.create", str(data))
...         return MyObject(**data)
...     def add(self, obj):
...         self.context[obj.name] = obj
...     def nextURL(self):
...         pass
```

We create the form and try to update it:

```
>>> request = TestRequest()
```

```
>>> myaddform =  MyAddForm(root, request)
```

```
>>> myaddform.update()
```

As usual, the form contains a widget manager with the expected widget

```
>>> list(myaddform.widgets.keys())
['subobject', 'name']
>>> list(myaddform.widgets.values())
[<ObjectWidget 'form.widgets.subobject'>, <TextWidget 'form.widgets.name'>]
```

The addform has our ObjectWidget which in turn contains the sub-widgets:

```
>>> list(myaddform.widgets['subobject'].widgets.keys())
['subfield', 'moofield']
```

```
>>> list(myaddform.widgets['subobject'].widgets['subfield'].widgets.keys())
['foofield', 'barfield']
```

If we want to render the addform, we must give it a template:

```
>>> addTemplate(myaddform)
```

Now rendering the addform renders the subform as well:

```
>>> print(myaddform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-subobject">my object</label>
        <div class="object-widget required">
          <div class="label">
            <label for="form-widgets-subobject-widgets-subfield">
              <span>Second-subobject</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <div class="object-widget required">
              <div class="label">
                <label for="form-widgets-subobject-widgets-subfield-widgets-foofield">
                  <span>My foo field</span>
                  <span class="required">*</span>
                </label>
              </div>
              <div class="widget">
                <input class="text-widget required int-field"
                id="form-widgets-subobject-widgets-subfield-widgets-foofield"
                name="form.widgets.subobject.widgets.subfield.widgets.foofield"
                type="text" value="1,111">
              </div>
              <div class="label">
                <label for="form-widgets-subobject-widgets-subfield-widgets-barfield">
                  <span>My dear bar</span>
                </label>
              </div>
              <div class="widget">
                <input class="text-widget int-field"
                id="form-widgets-subobject-widgets-subfield-widgets-barfield"
                name="form.widgets.subobject.widgets.subfield.widgets.barfield"
                type="text" value="2,222">
              </div>
              <input name="form.widgets.subobject.widgets.subfield-empty-marker" type=
↪"hidden" value="1">
            </div>
```

(continues on next page)

(continued from previous page)

```
          </div>
          <div class="label">
            <label for="form-widgets-subobject-widgets-moofield">
              <span>Something</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required textline-field"
            id="form-widgets-subobject-widgets-moofield"
            name="form.widgets.subobject.widgets.moofield" type="text" value="">
          </div>
          <input name="form.widgets.subobject-empty-marker" type="hidden" value="1">
        </div>
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field"
        id="form-widgets-name" name="form.widgets.name" type="text" value="">
      </div>
      <div class="action">
        <input class="submit-widget button-field" id="form-buttons-add" name="form.
↪buttons.add" type="submit" value="Add">
      </div>
    </form>
  </body>
</html>
```

Coverage happiness

Converting interfaces.NO_VALUE holds None:

```
>>> converter.toFieldValue(interfaces.NO_VALUE) is None
True
```

This is a complicated case. Happens when the context is a dict, and the dict misses the field. (Note, we're making `sub__object` instead of `subobject`)

```
>>> context = dict(sub__object=None, foo=123, bar=456)
```

All the story the create a widget:

```
>>> field = zope.schema.Object(
...     __name__='subobject',
...     title='my object widget',
...     schema=IMySubObject)
```

```
>>> wv = z3c.form.object.ObjectWidgetValue(
...     {'foofield': '2', 'barfield': '999'})
```

```
>>> request = TestRequest()
>>> widget = z3c.form.browser.object.ObjectWidget(request)
>>> widget = FieldWidget(field, widget)
>>> widget.context = context
>>> widget.value = wv
>>> widget.update()
>>> converter = interfaces.IDataConverter(widget)
```

And still we get a MySubObject, no failure:

```
>>> value = converter.toFieldValue(wv)
>>> value
<z3c.form.testing.MySubObject object at ...>
>>> value.foofield
2
>>> value.barfield
999
```

Easy (after the previous). In case the previous value on the context is None (or missing). We need to create a new object to be able to set properties on.

```
>>> context['subobject'] = None
>>> value = converter.toFieldValue(wv)
>>> value
<z3c.form.testing.MySubObject object at ...>
>>> value.foofield
2
>>> value.barfield
999
```

In case there is something that cannot be adapted to the right interface, it just burps: (might be an idea to create in this case also a new blank object)

```
>>> context['subobject'] = 'brutal'
>>> converter.toFieldValue(wv)
Traceback (most recent call last):
...
TypeError: ('Could not adapt', 'brutal',
z3c.form.testing.IMySubObject
```

```
>>> context['subobject'] = None
```

One more. Value to convert misses a field. Should never happen actually:

```
>>> wv = z3c.form.object.ObjectWidgetValue(
...     {'foofield': '2'})
>>> value = converter.toFieldValue(wv)
```

Known property is set:

```
>>> value.foofield
2
```

Unknown sticks ti it's default value:

```
>>> value.barfield
2222
```

4.13.2 ObjectWidget single widgets integration tests

Checking components on the highest possible level.

```
>>> from datetime import date
>>> from z3c.form import form
>>> from z3c.form import field
>>> from z3c.form import testing
```

```
>>> from z3c.form.object import registerFactoryAdapter
>>> registerFactoryAdapter(testing.IObjectWidgetSingleSubIntegration,
...     testing.ObjectWidgetSingleSubIntegration)
```

```
>>> request = testing.TestRequest()
```

>> from z3c.form.object import registerFactoryAdapter >> registerFactoryAdapter(testing.IObjectWidgetSingleSubIntegration, .. testing.ObjectWidgetSingleSubIntegration)

```
>>> class EForm(form.EditForm):
...     form.extends(form.EditForm)
...     fields = field.Fields(testing.IObjectWidgetSingleIntegration)
```

Our single content object:

```
>>> obj = testing.ObjectWidgetSingleIntegration()
```

We recreate the form each time, to stay as close as possible. In real life the form gets instantiated and destroyed with each request.

```
>>> def getForm(request, fname=None):
...     frm = EForm(obj, request)
...     testing.addTemplate(frm, 'integration_edit.pt')
...     frm.update()
...     content = frm.render()
...     if fname is not None:
...         testing.saveHtml(content, fname)
...     return content
```

Empty

All blank and empty values:

```
>>> content = getForm(request, 'ObjectWidget_single_edit_empty.html')
```

```
>>> print(testing.plainText(content))
Object label
Int label *
```

(continues on next page)

(continued from previous page)

```
[]
Bool label *
( ) yes ( ) no
Choice label *
[[    ]]
ChoiceOpt label
[No value]
TextLine label *
[]
Date label *
[]
ReadOnly label *
[]
[Apply]
```

Some valid default values

```
>>> obj.subobj = testing.ObjectWidgetSingleSubIntegration(
...     singleInt=-100,
...     singleBool=False,
...     singleChoice='two',
...     singleChoiceOpt='six',
...     singleTextLine='some text one',
...     singleDate=date(2014, 6, 20),
...     singleReadOnly='some R/O text')
```

```
>>> content = getForm(request, 'ObjectWidget_single_edit_simple.html')
```

```
>>> print(testing.plainText(content))
Object label
Int label *
[-100]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[some text one]
Date label *
[14/06/20]
ReadOnly label *
some R/O text
[Apply]
```

Wrong values

Set wrong values:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.subobj.widgets.singleInt'] = 'foobar'
>>> submit['form.widgets.subobj.widgets.singleTextLine'] = 'foo\nbar'
>>> submit['form.widgets.subobj.widgets.singleDate'] = 'foobar'
```

```
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

We should get lots of errors:

```
>>> content = getForm(request, 'ObjectWidget_single_edit_submit_wrong.html')
>>> print(testing.plainText(content,
...     './/ul[@id="form-errors"]'))
* Object label: The entered value is not a valid integer literal.
Constraint not satisfied
The datetime string did not match the pattern 'yy/MM/dd'.
```

```
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-subobj"]/b/div[@class="error"]'))
The entered value is not a valid integer literal.
Constraint not satisfied
The datetime string did not match the pattern 'yy/MM/dd'.
```

```
>>> print(testing.plainText(content,
...     './/div[@id="row-form-widgets-subobj"]'))
The entered value is not a valid integer literal.
Constraint not satisfied
The datetime string did not match the pattern 'yy/MM/dd'. Object label
Int label *

The entered value is not a valid integer literal.
[foobar]

Bool label *

( ) yes (O) no

Choice label *

[two]

ChoiceOpt label

[six]

TextLine label *
```

(continues on next page)

(continued from previous page)

```
Constraint not satisfied
[foo
bar]

Date label *

The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]

ReadOnly label *

some R/O text
```

Let's fix the values:

```
>>> submit = testing.getSubmitValues(content)
```

```
>>> submit['form.widgets.subobj.widgets.singleInt'] = '1042'
>>> submit['form.widgets.subobj.widgets.singleBool'] = 'true'
>>> submit['form.widgets.subobj.widgets.singleChoice:list'] = 'three'
>>> submit['form.widgets.subobj.widgets.singleChoiceOpt:list'] = 'four'
>>> submit['form.widgets.subobj.widgets.singleTextLine'] = 'foobar'
>>> submit['form.widgets.subobj.widgets.singleDate'] = '14/06/21'
```

```
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

```
>>> content = getForm(request, 'ObjectWidget_single_edit_submit_fixed.html')
>>> print(testing.plainText(content))
Data successfully updated.

Object label
Int label *
[1,042]
Bool label *
(O) yes ( ) no
Choice label *
[three]
ChoiceOpt label
[four]
TextLine label *
[foobar]
Date label *
[14/06/21]
ReadOnly label *
some R/O text
[Apply]
```

Bool was misbehaving

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.subobj.widgets.singleBool'] = 'false'
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

```
>>> content = getForm(request, 'ObjectWidget_single_edit_submit_bool1.html')
>>> print(testing.plainText(content))
Data successfully updated.
...
```

```
>>> pprint(obj.subobj)
<ObjectWidgetSingleSubIntegration
  singleBool: False
  singleChoice: 'three'
  singleChoiceOpt: 'four'
  singleDate: datetime.date(2014, 6, 21)
  singleInt: 1042
  singleReadOnly: 'some R/O text'
  singleTextLine: 'foobar'>
```

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.subobj.widgets.singleBool'] = 'true'
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

```
>>> content = getForm(request, 'ObjectWidget_single_edit_submit_bool2.html')
>>> print(testing.plainText(content))
Data successfully updated.
...
```

```
>>> pprint(obj.subobj)
<ObjectWidgetSingleSubIntegration
  singleBool: True
  singleChoice: 'three'
  singleChoiceOpt: 'four'
  singleDate: datetime.date(2014, 6, 21)
  singleInt: 1042
  singleReadOnly: 'some R/O text'
  singleTextLine: 'foobar'>
```

4.13.3 Multi+Object Widget

The multi widget allows you to add and edit one or more values.

In order to not overwhelm you with our set of well-chosen defaults, all the default component registrations have been made prior to doing those examples:

```
>>> from z3c.form import testing
>>> testing.setupFormDefaults()
```

As for all widgets, the multi widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import multi
```

```
>>> verifyClass(interfaces.IWidget, multi.MultiWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
>>> widget = multi.MultiWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget-id'
>>> widget.name = 'widget.name'
```

We also need to register the template for at least the widget and request:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('multi_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IMultiWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('multi_display.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IMultiWidget),
...     IPageTemplate, name=interfaces.DISPLAY_MODE)
```

For the next test, we need to setup our button handler adapters.

```
>>> from z3c.form import button
>>> zope.component.provideAdapter(button.ButtonActions)
>>> zope.component.provideAdapter(button.ButtonActionHandler)
>>> zope.component.provideAdapter(button.ButtonAction,
...     provides=interfaces.IButtonAction)
```

Our submit buttons will need a template as well:

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('submit_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISubmitWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

We can now render the widget:

```
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="0" />
```

As you can see the widget is empty and doesn't provide values. This is because the widget does not know what sub-widgets to display. So let's register a *IFieldWidget* adapter and a template for our *IInt* field:

```
>>> import z3c.form.interfaces
>>> from z3c.form.browser.text import TextFieldWidget
>>> zope.component.provideAdapter(TextFieldWidget,
...     (zope.schema.interfaces.IInt, z3c.form.interfaces.IFormLayer))
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('text_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ITextWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

Let's now update the widget and check it again.

```
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget">
  <div class="buttons">
    <input type="submit" id="widget-name-buttons-add"
       name="widget.name.buttons.add"
       class="submit-widget button-field" value="Add" />
    <input type="submit" id="widget-name-buttons-remove"
       name="widget.name.buttons.remove"
       class="submit-widget button-field" value="Remove selected" />
   </div>
</div>
<input type="hidden" name="widget.name.count" value="0" />
```

It's still the same. Since the widget doesn't provide a field nothing useful gets rendered. Now let's define a field for this widget and check it again:

```
>>> from z3c.form.widget import FieldWidget
```

```
>>> from z3c.form.testing import IMySubObjectMulti
>>> from z3c.form.testing import MySubObjectMulti
```

```
>>> from z3c.form.object import registerFactoryAdapter
>>> registerFactoryAdapter(IMySubObjectMulti, MySubObjectMulti)
```

```
>>> field = zope.schema.List(
...     __name__='foo',
...     value_type=zope.schema.Object(title=u'my object widget',
...                                   schema=IMySubObjectMulti),
...     )
```

```
>>> widget = FieldWidget(field, widget)
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget required">
  <div class="buttons">
    <input type="submit" id="foo-buttons-add"
       name="foo.buttons.add"
       class="submit-widget button-field" value="Add" />
   </div>
</div>
<input type="hidden" name="foo.count" value="0" />
```

As you can see, there is still no input value. Let's provide some values for this widget. Before we can do that, we will need to register a data converter for our multi widget and the data converter dispatcher adapter:

```
>>> from z3c.form.converter import IntegerDataConverter
>>> from z3c.form.converter import FieldWidgetDataConverter
>>> from z3c.form.converter import MultiConverter
>>> from z3c.form.validator import SimpleFieldValidator
>>> zope.component.provideAdapter(IntegerDataConverter)
>>> zope.component.provideAdapter(FieldWidgetDataConverter)
>>> zope.component.provideAdapter(SimpleFieldValidator)
>>> zope.component.provideAdapter(MultiConverter)
```

Bunch of adapters to get objectwidget work:

```
>>> from z3c.form import datamanager
>>> zope.component.provideAdapter(datamanager.DictionaryField)
```

```
>>> import z3c.form.browser.object
>>> zope.component.provideAdapter(z3c.form.browser.object.ObjectFieldWidget)
>>> import z3c.form.object
>>> zope.component.provideAdapter(z3c.form.object.ObjectConverter)
>>> import z3c.form.error
>>> zope.component.provideAdapter(z3c.form.error.ValueErrorViewSnippet)
```

```
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('object_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IObjectWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('object_display.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IObjectWidget),
...     IPageTemplate, name=interfaces.DISPLAY_MODE)
```

```
>>> widget.update()
```

It must not fail if we assign values that do not meet the constraints, just cry about it in the HTML:

```
>>> widget.value = [z3c.form.object.ObjectWidgetValue(
...     {'foofield': u'', 'barfield': '666'})]
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget required">
  <div class="row" id="foo-0-row">
    <div class="label">
      <label for="foo-0">
        <span>my object widget</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="error">An object failed schema or invariant validation.</div>
    <div class="widget">
      <div class="multi-widget-checkbox">
        <input class="multi-widget-checkbox checkbox-widget"
               id="foo-0-remove" name="foo.0.remove" type="checkbox" value="1">
      </div>
      <div class="multi-widget-input">
        <div class="object-widget required">
          <div class="label">
            <label for="foo-0-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="error">Required input is missing.</div>
          <div class="widget">
            <input class="text-widget required int-field"
                   id="foo-0-widgets-foofield" name="foo.0.widgets.foofield"
                   type="text" value="">
          </div>
          <div class="label">
            <label for="foo-0-widgets-barfield">
```

(continues on next page)

(continued from previous page)

```
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field"
                   id="foo-0-widgets-barfield" name="foo.0.widgets.barfield"
                   type="text" value="666">
          </div>
          <input name="foo.0-empty-marker" type="hidden" value="1">
        </div>
      </div>
    </div>
  </div>
  <div class="buttons">
    <input id="foo-buttons-add" name="foo.buttons.add"
           class="submit-widget button-field" value="Add"
           type="submit" />
    <input id="foo-buttons-remove"
           name="foo.buttons.remove"
           class="submit-widget button-field" value="Remove selected"
           type="submit" />
  </div>
</div>
<input name="foo.count" type="hidden" value="1">
```

Let's set acceptable values:

```
>>> widget.value = [
...     z3c.form.object.ObjectWidgetValue(dict(foofield=u'42', barfield=u'666')),
...     z3c.form.object.ObjectWidgetValue(dict(foofield=u'789', barfield=u'321'))]
```

```
>>> print(widget.render())
<div class="multi-widget required">
    <div id="foo-0-row" class="row">
        <div class="label">
          <label for="foo-0">
            <span>my object widget</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <div class="multi-widget-checkbox">
            <input id="foo-0-remove"
                   name="foo.0.remove"
                   class="multi-widget-checkbox checkbox-widget"
                   type="checkbox" value="1" />
          </div>
          <div class="multi-widget-input">
            <div class="object-widget required">
              <div class="label">
                <label for="foo-0-widgets-foofield">
                  <span>My foo field</span>
```

(continues on next page)

(continued from previous page)

```
                <span class="required">*</span>
              </label>
            </div>
            <div class="widget">
                <input id="foo-0-widgets-foofield"
                       name="foo.0.widgets.foofield"
                       class="text-widget required int-field" value="42"
                       type="text" />
            </div>
            <div class="label">
              <label for="foo-0-widgets-barfield">
                <span>My dear bar</span>
              </label>
            </div>
            <div class="widget">
                <input id="foo-0-widgets-barfield"
                       name="foo.0.widgets.barfield"
                       class="text-widget int-field" value="666"
                       type="text" />
            </div>
            <input name="foo.0-empty-marker" type="hidden"
                   value="1" />
          </div>
        </div>
      </div>
  </div>
  <div id="foo-1-row" class="row">
      <div class="label">
        <label for="foo-1">
          <span>my object widget</span>
          <span class="required">*</span>
        </label>
      </div>
      <div class="widget">
        <div class="multi-widget-checkbox">
          <input id="foo-1-remove"
                 name="foo.1.remove"
                 class="multi-widget-checkbox checkbox-widget"
                 type="checkbox" value="1" />
        </div>
        <div class="multi-widget-input">
          <div class="object-widget required">
            <div class="label">
              <label for="foo-1-widgets-foofield">
                <span>My foo field</span>
                <span class="required">*</span>
              </label>
            </div>
            <div class="widget">
              <input id="foo-1-widgets-foofield"
                     name="foo.1.widgets.foofield"
                     class="text-widget required int-field"
```

(continues on next page)

(continued from previous page)

```
                    value="789" type="text" />
            </div>
            <div class="label">
              <label for="foo-1-widgets-barfield">
                <span>My dear bar</span>
              </label>
            </div>
            <div class="widget">
                <input id="foo-1-widgets-barfield"
                       name="foo.1.widgets.barfield"
                       class="text-widget int-field" value="321"
                       type="text" />
            </div>
            <input name="foo.1-empty-marker" type="hidden"
                   value="1" />
          </div>
        </div>
      </div>
    </div>
  <div class="buttons">
    <input id="foo-buttons-add" name="foo.buttons.add"
           class="submit-widget button-field" value="Add"
           type="submit" />
    <input id="foo-buttons-remove"
           name="foo.buttons.remove"
           class="submit-widget button-field" value="Remove selected"
           type="submit" />
  </div>
</div>
<input type="hidden" name="foo.count" value="2" />
```

Let's see what we get on value extraction:

```
>>> widget.extract()
<NO_VALUE>
```

If we now click on the Add button, we will get a new input field for enter a new value:

```
>>> widget.request = TestRequest(form={'foo.count':u'2',
...                                    'foo.0.widgets.foofield':u'42',
...                                    'foo.0.widgets.barfield':u'666',
...                                    'foo.0-empty-marker':u'1',
...                                    'foo.1.widgets.foofield':u'789',
...                                    'foo.1.widgets.barfield':u'321',
...                                    'foo.1-empty-marker':u'1',
...                                    'foo.buttons.add':'Add'})
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget required">
  <div class="row" id="foo-0-row">
    <div class="label">
      <label for="foo-0">
```

(continues on next page)

(continued from previous page)

```
      <span>my object widget</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <div class="multi-widget-checkbox">
      <input class="multi-widget-checkbox checkbox-widget"
             id="foo-0-remove"
             name="foo.0.remove"
             type="checkbox" value="1">
    </div>
    <div class="multi-widget-input">
      <div class="object-widget required">
        <div class="label">
          <label for="foo-0-widgets-foofield">
            <span>My foo field</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget required int-field"
                 id="foo-0-widgets-foofield"
                 name="foo.0.widgets.foofield"
                 type="text" value="42">
        </div>
        <div class="label">
          <label for="foo-0-widgets-barfield">
            <span>My dear bar</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget int-field"
                 id="foo-0-widgets-barfield"
                 name="foo.0.widgets.barfield"
                 type="text" value="666">
        </div>
        <input name="foo.0-empty-marker" type="hidden" value="1">
      </div>
    </div>
  </div>
</div>
<div class="row" id="foo-1-row">
  <div class="label">
    <label for="foo-1">
      <span>my object widget</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <div class="multi-widget-checkbox">
      <input class="multi-widget-checkbox checkbox-widget"
             id="foo-1-remove"
```

(continues on next page)

(continued from previous page)

```
              name="foo.1.remove"
              type="checkbox" value="1">
    </div>
    <div class="multi-widget-input">
      <div class="object-widget required">
        <div class="label">
          <label for="foo-1-widgets-foofield">
            <span>My foo field</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget required int-field"
                 id="foo-1-widgets-foofield"
                 name="foo.1.widgets.foofield"
                 type="text" value="789">
        </div>
        <div class="label">
          <label for="foo-1-widgets-barfield">
            <span>My dear bar</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget int-field"
                 id="foo-1-widgets-barfield"
                 name="foo.1.widgets.barfield"
                 type="text" value="321">
        </div>
        <input name="foo.1-empty-marker" type="hidden" value="1">
      </div>
    </div>
  </div>
</div>
<div class="row" id="foo-2-row">
  <div class="label">
    <label for="foo-2">
      <span>my object widget</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <div class="multi-widget-checkbox">
      <input class="multi-widget-checkbox checkbox-widget"
             id="foo-2-remove"
             name="foo.2.remove"
             type="checkbox" value="1">
    </div>
    <div class="multi-widget-input">
      <div class="object-widget required">
        <div class="label">
          <label for="foo-2-widgets-foofield">
            <span>My foo field</span>
```

(continues on next page)

(continued from previous page)

```
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget required int-field"
                 id="foo-2-widgets-foofield"
                 name="foo.2.widgets.foofield"
                 type="text" value="">
        </div>
        <div class="label">
          <label for="foo-2-widgets-barfield">
            <span>My dear bar</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget int-field"
                 id="foo-2-widgets-barfield"
                 name="foo.2.widgets.barfield"
                 type="text" value="2,222">
        </div>
        <input name="foo.2-empty-marker" type="hidden" value="1">
      </div>
    </div>
  </div>
  <div class="buttons">
    <input id="foo-buttons-add" name="foo.buttons.add"
           class="submit-widget button-field" value="Add"
           type="submit" />
    <input id="foo-buttons-remove"
           name="foo.buttons.remove"
           class="submit-widget button-field" value="Remove selected"
           type="submit" />
  </div>
</div>
<input name="foo.count" type="hidden" value="3">
```

Let's see what we get on value extraction:

```
>>> value = widget.extract()
>>> pprint(value)
[{'barfield': '666', 'foofield': '42'}, {'barfield': '321', 'foofield': '789'}]
>>> converter = interfaces.IDataConverter(widget)
```

```
>>> value = converter.toFieldValue(value)
>>> value
[<z3c.form.testing.MySubObjectMulti object at ...>,
<z3c.form.testing.MySubObjectMulti object at ...>]
```

```
>>> value[0].foofield
42
>>> value[0].barfield
```

(continues on next page)

(continued from previous page)

```
666
```

Now let's store the new value:

```
>>> widget.request = TestRequest(form={'foo.count':u'3',
...                                    'foo.0.widgets.foofield':u'42',
...                                    'foo.0.widgets.barfield':u'666',
...                                    'foo.0-empty-marker':u'1',
...                                    'foo.1.widgets.foofield':u'789',
...                                    'foo.1.widgets.barfield':u'321',
...                                    'foo.1-empty-marker':u'1',
...                                    'foo.2.widgets.foofield':u'46',
...                                    'foo.2.widgets.barfield':u'98',
...                                    'foo.2-empty-marker':u'1',
...                                    })
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget required">
  <div class="row" id="foo-0-row">
    <div class="label">
      <label for="foo-0">
        <span>my object widget</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-checkbox">
        <input class="multi-widget-checkbox checkbox-widget" id="foo-0-remove"
               name="foo.0.remove" type="checkbox" value="1">
      </div>
      <div class="multi-widget-input">
        <div class="object-widget required">
          <div class="label">
            <label for="foo-0-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field"
                   id="foo-0-widgets-foofield" name="foo.0.widgets.foofield"
                   type="text" value="42">
          </div>
          <div class="label">
            <label for="foo-0-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field" id="foo-0-widgets-barfield"
                   name="foo.0.widgets.barfield" type="text" value="666">
          </div>
```

(continues on next page)

(continued from previous page)

```
        <input name="foo.0-empty-marker" type="hidden" value="1">
      </div>
    </div>
  </div>
</div>
<div class="row" id="foo-1-row">
  <div class="label">
    <label for="foo-1">
      <span>my object widget</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <div class="multi-widget-checkbox">
      <input class="multi-widget-checkbox checkbox-widget" id="foo-1-remove"
             name="foo.1.remove" type="checkbox" value="1">
    </div>
    <div class="multi-widget-input">
      <div class="object-widget required">
        <div class="label">
          <label for="foo-1-widgets-foofield">
            <span>My foo field</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget required int-field"
                 id="foo-1-widgets-foofield" name="foo.1.widgets.foofield"
                 type="text" value="789">
        </div>
        <div class="label">
          <label for="foo-1-widgets-barfield">
            <span>My dear bar</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget int-field" id="foo-1-widgets-barfield"
                 name="foo.1.widgets.barfield" type="text" value="321">
        </div>
        <input name="foo.1-empty-marker" type="hidden" value="1">
      </div>
    </div>
  </div>
</div>
<div class="row" id="foo-2-row">
  <div class="label">
    <label for="foo-2">
      <span>my object widget</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
```

(continues on next page)

(continued from previous page)

```
      <div class="multi-widget-checkbox">
        <input class="multi-widget-checkbox checkbox-widget" id="foo-2-remove"
               name="foo.2.remove" type="checkbox" value="1">
      </div>
      <div class="multi-widget-input">
        <div class="object-widget required">
          <div class="label">
            <label for="foo-2-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field"
                   id="foo-2-widgets-foofield" name="foo.2.widgets.foofield"
                   type="text" value="46">
          </div>
          <div class="label">
            <label for="foo-2-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field" id="foo-2-widgets-barfield"
                   name="foo.2.widgets.barfield" type="text" value="98">
          </div>
          <input name="foo.2-empty-marker" type="hidden" value="1">
        </div>
      </div>
    </div>
  </div>
  <div class="buttons">
    <input class="submit-widget button-field" id="foo-buttons-add"
           name="foo.buttons.add" type="submit" value="Add">
    <input class="submit-widget button-field" id="foo-buttons-remove"
           name="foo.buttons.remove" type="submit" value="Remove selected">
  </div>
</div>
<input name="foo.count" type="hidden" value="3">
```

Let's see what we get on value extraction:

```
>>> value = widget.extract()
>>> pprint(value)
[{'barfield': '666', 'foofield': '42'},
 {'barfield': '321', 'foofield': '789'},
 {'barfield': '98', 'foofield': '46'}]
>>> converter = interfaces.IDataConverter(widget)
```

```
>>> value = converter.toFieldValue(value)
>>> value
[<z3c.form.testing.MySubObjectMulti object at ...>,
```

(continues on next page)

(continued from previous page)

```
<z3c.form.testing.MySubObjectMulti object at ...>]
```

```
>>> value[0].foofield
42
>>> value[0].barfield
666
```

As you can see in the above sample, the new stored value gets rendered as a real value and the new adding value input field is gone. Now let's try to remove an existing value:

```
>>> widget.request = TestRequest(form={'foo.count':u'3',
...                                    'foo.0.widgets.foofield':u'42',
...                                    'foo.0.widgets.barfield':u'666',
...                                    'foo.0-empty-marker':u'1',
...                                    'foo.1.widgets.foofield':u'789',
...                                    'foo.1.widgets.barfield':u'321',
...                                    'foo.1-empty-marker':u'1',
...                                    'foo.2.widgets.foofield':u'46',
...                                    'foo.2.widgets.barfield':u'98',
...                                    'foo.2-empty-marker':u'1',
...                                    'foo.1.remove':u'1',
...                                    'foo.buttons.remove':'Remove selected'})
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget required">
  <div class="row" id="foo-0-row">
    <div class="label">
      <label for="foo-0">
        <span>my object widget</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-checkbox">
        <input class="multi-widget-checkbox checkbox-widget" id="foo-0-remove"
               name="foo.0.remove" type="checkbox" value="1">
      </div>
      <div class="multi-widget-input">
        <div class="object-widget required">
          <div class="label">
            <label for="foo-0-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field"
                   id="foo-0-widgets-foofield" name="foo.0.widgets.foofield"
                   type="text" value="42">
          </div>
          <div class="label">
            <label for="foo-0-widgets-barfield">
```

(continues on next page)

(continued from previous page)

```
            <span>My dear bar</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget int-field" id="foo-0-widgets-barfield"
                 name="foo.0.widgets.barfield" type="text" value="666">
        </div>
        <input name="foo.0-empty-marker" type="hidden" value="1">
      </div>
    </div>
  </div>
</div>
<div class="row" id="foo-1-row">
  <div class="label">
    <label for="foo-1">
      <span>my object widget</span>
      <span class="required">*</span>
    </label>
  </div>
  <div class="widget">
    <div class="multi-widget-checkbox">
      <input class="multi-widget-checkbox checkbox-widget" id="foo-1-remove"
             name="foo.1.remove" type="checkbox" value="1">
    </div>
    <div class="multi-widget-input">
      <div class="object-widget required">
        <div class="label">
          <label for="foo-1-widgets-foofield">
            <span>My foo field</span>
            <span class="required">*</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget required int-field"
                 id="foo-1-widgets-foofield" name="foo.1.widgets.foofield"
                 type="text" value="46">
        </div>
        <div class="label">
          <label for="foo-1-widgets-barfield">
            <span>My dear bar</span>
          </label>
        </div>
        <div class="widget">
          <input class="text-widget int-field" id="foo-1-widgets-barfield"
                 name="foo.1.widgets.barfield" type="text" value="98">
        </div>
        <input name="foo.1-empty-marker" type="hidden" value="1">
      </div>
    </div>
  </div>
</div>
<div class="buttons">
```

(continues on next page)

(continued from previous page)

```
    <input class="submit-widget button-field" id="foo-buttons-add"
           name="foo.buttons.add" type="submit" value="Add">
    <input class="submit-widget button-field" id="foo-buttons-remove"
           name="foo.buttons.remove" type="submit" value="Remove selected">
  </div>
</div>
<input name="foo.count" type="hidden" value="2">
```

Let's see what we get on value extraction: (this is good so, because Remove selected is a widget-internal submit)

```
>>> value = widget.extract()
>>> pprint(value)
[{'barfield': '666', 'foofield': '42'},
 {'barfield': '321', 'foofield': '789'},
 {'barfield': '98', 'foofield': '46'}]
>>> converter = interfaces.IDataConverter(widget)
```

```
>>> value = converter.toFieldValue(value)
>>> value
[<z3c.form.testing.MySubObjectMulti object at ...>,
<z3c.form.testing.MySubObjectMulti object at ...>]
```

```
>>> value[0].foofield
42
>>> value[0].barfield
666
```

Error handling is next. Let's use the value "bad" (an invalid integer literal) as input for our internal (sub) widget.

```
>>> from z3c.form.error import ErrorViewSnippet
>>> from z3c.form.error import StandardErrorViewTemplate
>>> zope.component.provideAdapter(ErrorViewSnippet)
>>> zope.component.provideAdapter(StandardErrorViewTemplate)
```

```
>>> widget.request = TestRequest(form={'foo.count':u'2',
...                                    'foo.0.widgets.foofield':u'42',
...                                    'foo.0.widgets.barfield':u'666',
...                                    'foo.0-empty-marker':u'1',
...                                    'foo.1.widgets.foofield':u'bad',
...                                    'foo.1.widgets.barfield':u'98',
...                                    'foo.1-empty-marker':u'1',
...                                    })
```

```
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget required">
  <div class="row" id="foo-0-row">
    <div class="label">
      <label for="foo-0">
        <span>my object widget</span>
        <span class="required">*</span>
```

(continues on next page)

(continued from previous page)

```
      </label>
    </div>
    <div class="widget">
      <div class="multi-widget-checkbox">
        <input class="multi-widget-checkbox checkbox-widget" id="foo-0-remove" name="foo.
↪0.remove" type="checkbox" value="1">
      </div>
      <div class="multi-widget-input">
        <div class="object-widget required">
          <div class="label">
            <label for="foo-0-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field" id="foo-0-widgets-foofield"␣
↪name="foo.0.widgets.foofield" type="text" value="42">
          </div>
          <div class="label">
            <label for="foo-0-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field" id="foo-0-widgets-barfield" name="foo.0.
↪widgets.barfield" type="text" value="666">
          </div>
          <input name="foo.0-empty-marker" type="hidden" value="1">
        </div>
      </div>
    </div>
  </div>
  <div class="row" id="foo-1-row">
    <div class="label">
      <label for="foo-1">
        <span>my object widget</span>
        <span class="required">*</span>
      </label>
    </div>
    <div class="error">The entered value is not a valid integer literal.</div>
    <div class="widget">
      <div class="multi-widget-checkbox">
        <input class="multi-widget-checkbox checkbox-widget" id="foo-1-remove" name="foo.
↪1.remove" type="checkbox" value="1">
      </div>
      <div class="multi-widget-input">
        <div class="object-widget required">
          <div class="label">
            <label for="foo-1-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
```

(continues on next page)

(continued from previous page)

```
            </label>
          </div>
          <div class="error">The entered value is not a valid integer literal.</div>
          <div class="widget">
            <input class="text-widget required int-field" id="foo-1-widgets-foofield"␣
↪name="foo.1.widgets.foofield" type="text" value="bad">
          </div>
          <div class="label">
            <label for="foo-1-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field" id="foo-1-widgets-barfield" name="foo.1.
↪widgets.barfield" type="text" value="98">
          </div>
          <input name="foo.1-empty-marker" type="hidden" value="1">
        </div>
      </div>
    </div>
  </div>
  <div class="buttons">
    <input class="submit-widget button-field" id="foo-buttons-add" name="foo.buttons.add
↪" type="submit" value="Add">
    <input class="submit-widget button-field" id="foo-buttons-remove" name="foo.buttons.
↪remove" type="submit" value="Remove selected">
  </div>
</div>
<input name="foo.count" type="hidden" value="2">
```

Let's see what we get on value extraction:

```
>>> value = widget.extract()
>>> pprint(value)
[{'barfield': '666', 'foofield': '42'},
 {'barfield': '98', 'foofield': 'bad'}]
```

Label

There is an option which allows to disable the label for the (sub) widgets. You can set the *showLabel* option to *False* which will skip rendering the labels. Alternatively you can also register your own template for your layer if you like to skip the label rendering for all widgets.

```
>>> field = zope.schema.List(
...     __name__='foo',
...     value_type=zope.schema.Object(title=u'ignored_title',
...                                   schema=IMySubObjectMulti),
...     )
>>> request = TestRequest()
>>> widget = multi.MultiWidget(request)
>>> widget = FieldWidget(field, widget)
```

(continues on next page)

(continued from previous page)

```
>>> widget.value = [
...     z3c.form.object.ObjectWidgetValue(dict(foofield='42', barfield='666')),
...     z3c.form.object.ObjectWidgetValue(dict(foofield='789', barfield='321'))]
>>> widget.showLabel = False
>>> widget.update()
>>> print(widget.render())
<div class="multi-widget required">
  <div class="row" id="foo-0-row">
    <div class="widget">
      <div class="multi-widget-checkbox">
        <input class="multi-widget-checkbox checkbox-widget" id="foo-0-remove" name="foo.
↪0.remove" type="checkbox" value="1">
      </div>
      <div class="multi-widget-input">
        <div class="object-widget required">
          <div class="label">
            <label for="foo-0-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget required int-field" id="foo-0-widgets-foofield"␣
↪name="foo.0.widgets.foofield" type="text" value="42">
          </div>
          <div class="label">
            <label for="foo-0-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field" id="foo-0-widgets-barfield" name="foo.0.
↪widgets.barfield" type="text" value="666">
          </div>
          <input name="foo.0-empty-marker" type="hidden" value="1">
        </div>
      </div>
    </div>
  </div>
  <div class="row" id="foo-1-row">
    <div class="widget">
      <div class="multi-widget-checkbox">
        <input class="multi-widget-checkbox checkbox-widget" id="foo-1-remove" name="foo.
↪1.remove" type="checkbox" value="1">
      </div>
      <div class="multi-widget-input">
        <div class="object-widget required">
          <div class="label">
            <label for="foo-1-widgets-foofield">
              <span>My foo field</span>
              <span class="required">*</span>
            </label>
```

(continues on next page)

(continued from previous page)

```
          </div>
          <div class="widget">
            <input class="text-widget required int-field" id="foo-1-widgets-foofield"␣
↪name="foo.1.widgets.foofield" type="text" value="789">
          </div>
          <div class="label">
            <label for="foo-1-widgets-barfield">
              <span>My dear bar</span>
            </label>
          </div>
          <div class="widget">
            <input class="text-widget int-field" id="foo-1-widgets-barfield" name="foo.1.
↪widgets.barfield" type="text" value="321">
          </div>
          <input name="foo.1-empty-marker" type="hidden" value="1">
        </div>
      </div>
    </div>
  </div>
  <div class="buttons">
    <input class="submit-widget button-field" id="foo-buttons-add" name="foo.buttons.add
↪" type="submit" value="Add">
    <input class="submit-widget button-field" id="foo-buttons-remove" name="foo.buttons.
↪remove" type="submit" value="Remove selected">
  </div>
</div>
<input name="foo.count" type="hidden" value="2">
```

In a form

Let's try a simple example in a form.

We have to provide an adapter first:

```
>>> import z3c.form.browser.object
>>> zope.component.provideAdapter(z3c.form.browser.object.ObjectFieldWidget)
```

Forms and our objectwidget fire events on add and edit, setup a subscriber for those:

```
>>> eventlog = []
>>> import zope.lifecycleevent
>>> @zope.component.adapter(zope.lifecycleevent.ObjectModifiedEvent)
... def logEvent(event):
...     eventlog.append(event)
>>> zope.component.provideHandler(logEvent)
>>> @zope.component.adapter(zope.lifecycleevent.ObjectCreatedEvent)
... def logEvent2(event):
...     eventlog.append(event)
>>> zope.component.provideHandler(logEvent2)
```

```
>>> def printEvents():
...     for event in eventlog:
...         print(event)
...         if isinstance(event, zope.lifecycleevent.ObjectModifiedEvent):
...             for attr in event.descriptions:
...                 print(attr.interface)
...                 print(sorted(attr.attributes))
```

We need to provide the widgets for the List

```
>>> from z3c.form.browser.multi import multiFieldWidgetFactory
>>> zope.component.provideAdapter(multiFieldWidgetFactory,
...     (zope.schema.interfaces.IList, z3c.form.interfaces.IFormLayer))
>>> zope.component.provideAdapter(multiFieldWidgetFactory,
...     (zope.schema.interfaces.ITuple, z3c.form.interfaces.IFormLayer))
>>> zope.component.provideAdapter(multiFieldWidgetFactory,
...     (zope.schema.interfaces.IDict, z3c.form.interfaces.IFormLayer))
```

We define an interface containing a subobject, and an addform for it:

```
>>> from z3c.form import form, field
>>> from z3c.form.testing import MyMultiObject, IMyMultiObject
```

Note, that creating an object will print some information about it:

```
>>> class MyAddForm(form.AddForm):
...     fields = field.Fields(IMyMultiObject)
...     def create(self, data):
...         print("MyAddForm.create")
...         pprint(data)
...         return MyMultiObject(**data)
...     def add(self, obj):
...         self.context[obj.name] = obj
...     def nextURL(self):
...         pass
```

We create the form and try to update it:

```
>>> request = TestRequest()
>>> myaddform =  MyAddForm(root, request)
```

```
>>> myaddform.update()
```

As usual, the form contains a widget manager with the expected widget

```
>>> list(myaddform.widgets.keys())
['listOfObject', 'name']
>>> list(myaddform.widgets.values())
[<MultiWidget 'form.widgets.listOfObject'>, <TextWidget 'form.widgets.name'>]
```

If we want to render the addform, we must give it a template:

```
>>> import os
>>> from zope.browserpage.viewpagetemplatefile import ViewPageTemplateFile
```

(continues on next page)

(continued from previous page)

```
>>> from zope.browserpage.viewpagetemplatefile import BoundPageTemplate
>>> from z3c.form import tests
>>> def addTemplate(form):
...     form.template = BoundPageTemplate(
...         ViewPageTemplateFile(
...             'simple_edit.pt', os.path.dirname(tests.__file__)), form)
>>> addTemplate(myaddform)
```

Now rendering the addform renders no items yet:

```
>>> print(myaddform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-listOfObject">My list field</label>
        <div class="multi-widget required">
          <div class="buttons">
            <input class="submit-widget button-field"
                   id="form-widgets-listOfObject-buttons-add"
                   name="form.widgets.listOfObject.buttons.add"
                   type="submit" value="Add">
          </div>
        </div>
        <input name="form.widgets.listOfObject.count" type="hidden" value="0">
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field"
               id="form-widgets-name" name="form.widgets.name"
               type="text" value="">
      </div>
      <div class="action">
        <input class="submit-widget button-field"
               id="form-buttons-add" name="form.buttons.add"
               type="submit" value="Add">
      </div>
    </form>
  </body>
</html>
```

We don't have the object (yet) in the root:

```
>>> root['first']
Traceback (most recent call last):
...
KeyError: 'first'
```

Add a row to the multi widget:

```
>>> request = TestRequest(form={
...     'form.widgets.listOfObject.count':u'0',
```

(continues on next page)

(continued from previous page)

```
...     'form.widgets.listOfObject.buttons.add':'Add'})
>>> myaddform.request = request
```

Update with the request:

```
>>> myaddform.update()
```

Render the form:

```
>>> print(myaddform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-listOfObject">My list field</label>
        <div class="multi-widget required">
          <div class="row" id="form-widgets-listOfObject-0-row">
            <div class="label">
              <label for="form-widgets-listOfObject-0">
                <span>my object widget</span>
                <span class="required">*</span>
              </label>
            </div>
            <div class="widget">
              <div class="multi-widget-checkbox">
                <input class="multi-widget-checkbox checkbox-widget"
                       id="form-widgets-listOfObject-0-remove"
                       name="form.widgets.listOfObject.0.remove"
                       type="checkbox" value="1">
              </div>
              <div class="multi-widget-input">
                <div class="object-widget required">
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-foofield">
                      <span>My foo field</span>
                      <span class="required">*</span>
                    </label>
                  </div>
                  <div class="widget">
                    <input class="text-widget required int-field"
                           id="form-widgets-listOfObject-0-widgets-foofield"
                           name="form.widgets.listOfObject.0.widgets.foofield"
                           type="text" value="">
                  </div>
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-barfield">
                      <span>My dear bar</span>
                    </label>
                  </div>
                  <div class="widget">
                    <input class="text-widget int-field"
                           id="form-widgets-listOfObject-0-widgets-barfield"
```

(continues on next page)

(continued from previous page)

```
                          name="form.widgets.listOfObject.0.widgets.barfield"
                          type="text" value="2,222">
                  </div>
                  <input name="form.widgets.listOfObject.0-empty-marker"
                         type="hidden" value="1">
                </div>
              </div>
            </div>
          </div>
          <div class="buttons">
            <input class="submit-widget button-field"
                   id="form-widgets-listOfObject-buttons-add"
                   name="form.widgets.listOfObject.buttons.add"
                   type="submit" value="Add">
            <input class="submit-widget button-field"
                   id="form-widgets-listOfObject-buttons-remove"
                   name="form.widgets.listOfObject.buttons.remove"
                   type="submit" value="Remove selected">
          </div>
        </div>
        <input name="form.widgets.listOfObject.count" type="hidden" value="1">
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field"
               id="form-widgets-name" name="form.widgets.name" type="text" value="">
      </div>
      <div class="action">
        <input class="submit-widget button-field" id="form-buttons-add"
               name="form.buttons.add" type="submit" value="Add">
      </div>
    </form>
  </body>
</html>
```

Now we can fill in some values to the object, and a name to the whole schema:

```
>>> request = TestRequest(form={
...     'form.widgets.listOfObject.count':u'1',
...     'form.widgets.listOfObject.0.widgets.foofield':u'66',
...     'form.widgets.listOfObject.0.widgets.barfield':u'99',
...     'form.widgets.listOfObject.0-empty-marker':u'1',
...     'form.widgets.name':u'first',
...     'form.buttons.add':'Add'})
>>> myaddform.request = request
```

Update the form with the request:

```
>>> myaddform.update()
MyAddForm.create
{'listOfObject': [<z3c.form.testing.MySubObjectMulti ...],
 'name': 'first'}
```

Wow, it got added:

```
>>> root['first']
<z3c.form.testing.MyMultiObject object at ...>
```

```
>>> root['first'].listOfObject
[<z3c.form.testing.MySubObjectMulti object at ...>]
```

Field values need to be right:

```
>>> root['first'].listOfObject[0].foofield
66
>>> root['first'].listOfObject[0].barfield
99
```

Let's see our event log:

```
>>> len(eventlog)
6
```

((why is IMySubObjectMulti created twice???))

```
>>> printEvents()
<zope...ObjectCreatedEvent object at ...>
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMySubObjectMulti
['barfield', 'foofield']
<zope...ObjectCreatedEvent object at ...>
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMySubObjectMulti
['barfield', 'foofield']
<zope...ObjectCreatedEvent object at ...>
<zope...contained.ContainerModifiedEvent object at ...>
```

```
>>> eventlog=[]
```

Let's try to edit that newly added object:

```
>>> class MyEditForm(form.EditForm):
...     fields = field.Fields(IMyMultiObject)
```

```
>>> editform = MyEditForm(root['first'], TestRequest())
>>> addTemplate(editform)
>>> editform.update()
```

Watch for the widget values in the HTML:

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-listOfObject">My list field</label>
```

(continues on next page)

(continued from previous page)

```
        <div class="multi-widget required">
          <div class="row" id="form-widgets-listOfObject-0-row">
            <div class="label">
              <label for="form-widgets-listOfObject-0">
                <span>my object widget</span>
                <span class="required">*</span>
              </label>
            </div>
            <div class="widget">
              <div class="multi-widget-checkbox">
                <input class="multi-widget-checkbox checkbox-widget"
                       id="form-widgets-listOfObject-0-remove"
                       name="form.widgets.listOfObject.0.remove"
                       type="checkbox" value="1">
              </div>
              <div class="multi-widget-input">
                <div class="object-widget required">
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-foofield">
                      <span>My foo field</span>
                      <span class="required">*</span>
                    </label>
                  </div>
                  <div class="widget">
                    <input class="text-widget required int-field"
                           id="form-widgets-listOfObject-0-widgets-foofield"
                           name="form.widgets.listOfObject.0.widgets.foofield"
                           type="text" value="66">
                  </div>
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-barfield">
                      <span>My dear bar</span>
                    </label>
                  </div>
                  <div class="widget">
                    <input class="text-widget int-field"
                           id="form-widgets-listOfObject-0-widgets-barfield"
                           name="form.widgets.listOfObject.0.widgets.barfield"
                           type="text" value="99">
                  </div>
                  <input name="form.widgets.listOfObject.0-empty-marker"
                         type="hidden" value="1">
                </div>
              </div>
            </div>
          </div>
          <div class="buttons">
            <input class="submit-widget button-field"
                   id="form-widgets-listOfObject-buttons-add"
                   name="form.widgets.listOfObject.buttons.add"
                   type="submit" value="Add">
            <input class="submit-widget button-field"
```

(continues on next page)

(continued from previous page)

```
                    id="form-widgets-listOfObject-buttons-remove"
                    name="form.widgets.listOfObject.buttons.remove"
                    type="submit" value="Remove selected">
          </div>
        </div>
        <input name="form.widgets.listOfObject.count" type="hidden" value="1">
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field"
               id="form-widgets-name" name="form.widgets.name"
               type="text" value="first">
      </div>
      <div class="action">
        <input class="submit-widget button-field" id="form-buttons-apply"
               name="form.buttons.apply" type="submit" value="Apply">
      </div>
    </form>
  </body>
</html>
```

Let's modify the values:

```
>>> request = TestRequest(form={
...     'form.widgets.listOfObject.count':u'1',
...     'form.widgets.listOfObject.0.widgets.foofield':u'43',
...     'form.widgets.listOfObject.0.widgets.barfield':u'55',
...     'form.widgets.listOfObject.0-empty-marker':u'1',
...     'form.widgets.name':u'first',
...     'form.buttons.apply':'Apply'})
```

They are still the same:

```
>>> root['first'].listOfObject[0].foofield
66
>>> root['first'].listOfObject[0].barfield
99
```

```
>>> editform.request = request
>>> editform.update()
```

Until we have updated the form:

```
>>> root['first'].listOfObject[0].foofield
43
>>> root['first'].listOfObject[0].barfield
55
```

Let's see our event log:

```
>>> len(eventlog)
5
```

((TODO: now this is real crap here, why is IMySubObjectMulti created 3 times???))

```
>>> printEvents()
<zope...ObjectCreatedEvent object at ...>
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMySubObjectMulti
['barfield', 'foofield']
<zope...ObjectCreatedEvent object at ...>
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMySubObjectMulti
['barfield', 'foofield']
<zope...ObjectModifiedEvent object at ...>
z3c.form.testing.IMyMultiObject
['listOfObject']
```

```
>>> eventlog=[]
```

After the update the form says that the values got updated and renders the new values:

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <i>Data successfully updated.</i>
    <form action=".">
      <div class="row">
        <label for="form-widgets-listOfObject">My list field</label>
        <div class="multi-widget required">
          <div class="row" id="form-widgets-listOfObject-0-row">
            <div class="label">
              <label for="form-widgets-listOfObject-0">
                <span>my object widget</span>
                <span class="required">*</span>
              </label>
            </div>
            <div class="widget">
              <div class="multi-widget-checkbox">
                <input class="multi-widget-checkbox checkbox-widget"
                       id="form-widgets-listOfObject-0-remove"
                       name="form.widgets.listOfObject.0.remove"
                       type="checkbox" value="1">
              </div>
              <div class="multi-widget-input">
                <div class="object-widget required">
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-foofield">
                      <span>My foo field</span>
                      <span class="required">*</span>
                    </label>
                  </div>
                  <div class="widget">
                    <input class="text-widget required int-field"
                           id="form-widgets-listOfObject-0-widgets-foofield"
                           name="form.widgets.listOfObject.0.widgets.foofield"
```

(continues on next page)

(continued from previous page)

```
                        type="text" value="43">
                </div>
                <div class="label">
                  <label for="form-widgets-listOfObject-0-widgets-barfield">
                    <span>My dear bar</span>
                  </label>
                </div>
                <div class="widget">
                  <input class="text-widget int-field"
                         id="form-widgets-listOfObject-0-widgets-barfield"
                         name="form.widgets.listOfObject.0.widgets.barfield"
                         type="text" value="55">
                </div>
                <input name="form.widgets.listOfObject.0-empty-marker"
                       type="hidden" value="1">
              </div>
            </div>
          </div>
        </div>
        <div class="buttons">
          <input class="submit-widget button-field"
                 id="form-widgets-listOfObject-buttons-add"
                 name="form.widgets.listOfObject.buttons.add" type="submit"
                 value="Add">
          <input class="submit-widget button-field"
                 id="form-widgets-listOfObject-buttons-remove"
                 name="form.widgets.listOfObject.buttons.remove"
                 type="submit" value="Remove selected">
        </div>
      </div>
      <input name="form.widgets.listOfObject.count" type="hidden" value="1">
    </div>
    <div class="row">
      <label for="form-widgets-name">name</label>
      <input class="text-widget required textline-field"
             id="form-widgets-name" name="form.widgets.name"
             type="text" value="first">
    </div>
    <div class="action">
      <input class="submit-widget button-field" id="form-buttons-apply"
             name="form.buttons.apply" type="submit" value="Apply">
    </div>
  </form>
 </body>
</html>
```

Let's see if the widget keeps the old object on editing:

We add a special property to keep track of the object:

```
>>> root['first'].listOfObject[0].__marker__ = "ThisMustStayTheSame"
```

```
>>> root['first'].listOfObject[0].foofield
43
>>> root['first'].listOfObject[0].barfield
55
```

Let's modify the values:

```
>>> request = TestRequest(form={
...     'form.widgets.listOfObject.count':u'1',
...     'form.widgets.listOfObject.0.widgets.foofield':u'666',
...     'form.widgets.listOfObject.0.widgets.barfield':u'999',
...     'form.widgets.listOfObject.0-empty-marker':u'1',
...     'form.widgets.name':u'first',
...     'form.buttons.apply':'Apply'})
```

```
>>> editform.request = request
```

```
>>> editform.update()
```

Let's check what are ther esults of the update:

```
>>> root['first'].listOfObject[0].foofield
666
>>> root['first'].listOfObject[0].barfield
999
```

((TODO: bummer... we can't keep the old object))

#>>> root['first'].listOfObject[0].__marker__ #'ThisMustStayTheSame'

Let's make a nasty error, by typing 'bad' instead of an integer:

```
>>> request = TestRequest(form={
...     'form.widgets.listOfObject.count':u'1',
...     'form.widgets.listOfObject.0.widgets.foofield':u'99',
...     'form.widgets.listOfObject.0.widgets.barfield':u'bad',
...     'form.widgets.listOfObject.0-empty-marker':u'1',
...     'form.widgets.name':u'first',
...     'form.buttons.apply':'Apply'})
```

```
>>> editform.request = request
>>> eventlog=[]
>>> editform.update()
```

Eventlog must be clean:

```
>>> len(eventlog)
2
```

((TODO: bummer... who creates those 2 objects???))

```
>>> printEvents()
<zope...ObjectCreatedEvent object at ...>
<zope...ObjectCreatedEvent object at ...>
```

Watch for the error message in the HTML: it has to appear at the field itself and at the top of the form: ((not nice: at the top `Object is of wrong type.` appears))

```
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <i>There were some errors.</i>
    <ul>
      <li>
      My list field:
        <div class="error">The entered value is not a valid integer literal.</div>
      </li>
    </ul>
    <form action=".">
      <div class="row">
        <b>
          <div class="error">The entered value is not a valid integer literal.</div>
        </b>
        <label for="form-widgets-listOfObject">My list field</label>
        <div class="multi-widget required">
          <div class="row" id="form-widgets-listOfObject-0-row">
            <div class="label">
              <label for="form-widgets-listOfObject-0">
                <span>my object widget</span>
                <span class="required">*</span>
              </label>
            </div>
            <div class="error">The entered value is not a valid integer literal.</div>
            <div class="widget">
              <div class="multi-widget-checkbox">
                <input class="multi-widget-checkbox checkbox-widget"
                       id="form-widgets-listOfObject-0-remove"
                       name="form.widgets.listOfObject.0.remove"
                       type="checkbox" value="1">
              </div>
              <div class="multi-widget-input">
                <div class="object-widget required">
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-foofield">
                      <span>My foo field</span>
                      <span class="required">*</span>
                    </label>
                  </div>
                  <div class="widget">
                    <input class="text-widget required int-field"
                           id="form-widgets-listOfObject-0-widgets-foofield"
                           name="form.widgets.listOfObject.0.widgets.foofield"
                           type="text" value="99">
                  </div>
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-barfield">
                      <span>My dear bar</span>
                    </label>
```

(continues on next page)

(continued from previous page)

```
                  </div>
                  <div class="error">The entered value is not a valid integer literal.</
↪div>
                  <div class="widget">
                    <input class="text-widget int-field"
                           id="form-widgets-listOfObject-0-widgets-barfield"
                           name="form.widgets.listOfObject.0.widgets.barfield"
                           type="text" value="bad">
                  </div>
                  <input name="form.widgets.listOfObject.0-empty-marker" type="hidden"␣
↪value="1">
                </div>
              </div>
            </div>
          </div>
          <div class="buttons">
            <input class="submit-widget button-field"
                   id="form-widgets-listOfObject-buttons-add"
                   name="form.widgets.listOfObject.buttons.add"
                   type="submit" value="Add">
            <input class="submit-widget button-field"
                   id="form-widgets-listOfObject-buttons-remove"
                   name="form.widgets.listOfObject.buttons.remove"
                   type="submit" value="Remove selected">
          </div>
        </div>
        <input name="form.widgets.listOfObject.count" type="hidden" value="1">
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <input class="text-widget required textline-field"
               id="form-widgets-name" name="form.widgets.name"
               type="text" value="first">
      </div>
      <div class="action">
        <input class="submit-widget button-field" id="form-buttons-apply"
               name="form.buttons.apply" type="submit" value="Apply">
      </div>
    </form>
  </body>
</html>
```

The object values must stay at the old ones:

```
>>> root['first'].listOfObject[0].foofield
666
>>> root['first'].listOfObject[0].barfield
999
```

Simple but often used use-case is the display form:

```
>>> editform = MyEditForm(root['first'], TestRequest())
```

(continues on next page)

(continued from previous page)

```
>>> addTemplate(editform)
>>> editform.mode = interfaces.DISPLAY_MODE
>>> editform.update()
>>> print(editform.render())
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <form action=".">
      <div class="row">
        <label for="form-widgets-listOfObject">My list field</label>
        <div class="multi-widget" id="form-widgets-listOfObject">
          <div class="row" id="form-widgets-listOfObject-0-row">
            <div class="label">
              <label for="form-widgets-listOfObject-0">
                <span>my object widget</span>
                <span class="required">*</span>
              </label>
            </div>
            <div class="widget">
              <div class="multi-widget-display">
                <div class="object-widget">
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-foofield">
                      <span>My foo field</span>
                      <span class="required">*</span>
                    </label>
                  </div>
                  <div class="widget">
                    <span class="text-widget int-field"
                          id="form-widgets-listOfObject-0-widgets-foofield">666</span>
                  </div>
                  <div class="label">
                    <label for="form-widgets-listOfObject-0-widgets-barfield">
                      <span>My dear bar</span>
                    </label>
                  </div>
                  <div class="widget">
                    <span class="text-widget int-field"
                          id="form-widgets-listOfObject-0-widgets-barfield">999</span>
                  </div>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
      <div class="row">
        <label for="form-widgets-name">name</label>
        <span class="text-widget textline-field"
              id="form-widgets-name">first</span>
      </div>
      <div class="action">
        <input class="submit-widget button-field"
```

(continues on next page)

(continued from previous page)

```
                id="form-buttons-apply" name="form.buttons.apply"
                type="submit" value="Apply">
      </div>
    </form>
  </body>
</html>
```

4.13.4 ObjectWidget integration with MultiWidget of dict

a.k.a. dict of objects widget

```
>>> from datetime import date
>>> from z3c.form import form
>>> from z3c.form import field
>>> from z3c.form import testing
```

```
>>> from z3c.form.object import registerFactoryAdapter
>>> registerFactoryAdapter(testing.IObjectWidgetMultiSubIntegration,
...     testing.ObjectWidgetMultiSubIntegration)
```

```
>>> request = testing.TestRequest()
```

```
>>> class EForm(form.EditForm):
...     form.extends(form.EditForm)
...     fields = field.Fields(
...         testing.IMultiWidgetDictIntegration).select('dictOfObject')
```

Our multi content object:

```
>>> obj = testing.MultiWidgetDictIntegration()
```

We recreate the form each time, to stay as close as possible. In real life the form gets instantiated and destroyed with each request.

```
>>> def getForm(request, fname=None):
...     frm = EForm(obj, request)
...     testing.addTemplate(frm, 'integration_edit.pt')
...     frm.update()
...     content = frm.render()
...     if fname is not None:
...         testing.saveHtml(content, fname)
...     return content
```

Empty

All blank and empty values:

```
>>> content = getForm(request, 'ObjectMulti_dict_edit_empty.html')
```

```
>>> print(testing.plainText(content))
DictOfObject label

[Add]
[Apply]
```

Some valid default values

```
>>> sub1 = testing.ObjectWidgetMultiSubIntegration(
...     multiInt=-100,
...     multiBool=False,
...     multiChoice='two',
...     multiChoiceOpt='six',
...     multiTextLine='some text one',
...     multiDate=date(2014, 6, 20))
```

```
>>> sub2 = testing.ObjectWidgetMultiSubIntegration(
...     multiInt=42,
...     multiBool=True,
...     multiChoice='one',
...     multiChoiceOpt='four',
...     multiTextLine='second txt',
...     multiDate=date(2011, 3, 15))
```

```
>>> obj.dictOfObject = {'subob1': sub1, 'subob2': sub2}
```

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>}
```

```
>>> content = getForm(request, 'ObjectMulti_dict_edit_simple.html')
>>> print(testing.plainText(content))
```

(continues on next page)

(continued from previous page)

```
DictOfObject label

Object key *
[subob1]
Object label *
[ ]
Int label *
[-100]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[some text one]
Date label *
[14/06/20]
Object key *
[subob2]
Object label *
[ ]
Int label *
[42]
Bool label *
(O) yes ( ) no
Choice label *
[one]
ChoiceOpt label
[four]
TextLine label *
[second txt]
Date label *
[11/03/15]
[Add]
[Remove selected]
[Apply]
```

wrong input (Int)

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.widgets.multiInt'] = 'foobar'
```

```
>>> submit['form.widgets.dictOfObject.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get “The entered value is not a valid integer literal.” for “foobar” and a new input.

```
>>> content = getForm(request, 'ObjectMulti_dict_edit_submit_int.html')
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-dictOfObject-0-row"]'))
Object key *

[subob1]

Object label *

The entered value is not a valid integer literal.

[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[some text one]
Date label *
[14/06/20]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_dict_edit_submit_int_again.html')
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-dictOfObject-0-row"]//div[@class="error"]'))
Required input is missing.
An object failed schema or invariant validation.
Required input is missing.
Required input is missing.
Required input is missing.
Required input is missing.
```

```
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-dictOfObject-1-row"]//div[@class="error"]'))
The entered value is not a valid integer literal.
The entered value is not a valid integer literal.
```

```
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-dictOfObject-0-row"]'))
Object key *

Required input is missing.

[]
```

(continues on next page)

(continued from previous page)

```
Object label *

An object failed schema or invariant validation.

[ ]
Int label *
Required input is missing.
[]
Bool label *
Required input is missing.
( ) yes ( ) no
Choice label *
[one]
ChoiceOpt label
[No value]
TextLine label *
Required input is missing.
[]
Date label *
Required input is missing.
[]
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.remove'] = '1'
>>> submit['form.widgets.dictOfObject.2.remove'] = '1'
>>> submit['form.widgets.dictOfObject.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_dict_edit_remove_int.html')
>>> print(testing.plainText(content))
DictOfObject label

Object key *
[subob1]
Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[some text one]
Date label *
[14/06/20]
```

(continues on next page)

(continued from previous page)

```
[Add]
[Remove selected]
[Apply]
```

The object is unchanged:

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>}
```

wrong input (TextLine)

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.widgets.multiTextLine'] = 'foo\nbar'
```

```
>>> submit['form.widgets.dictOfObject.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get “Constraint not satisfied” for “foonbar” and a new input.

```
>>> content = getForm(request, 'ObjectMulti_dict_edit_submit_textline.html')
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-dictOfObject-0-row"]'))
Object key *

[subob1]

Object label *

The entered value is not a valid integer literal.

[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
```

(continues on next page)

(continued from previous page)

```
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
[14/06/20]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_dict_edit_submit_textline_again.html')
```

```
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-dictOfObject-0-row"]//div[@class="error"]'))
Required input is missing.
An object failed schema or invariant validation.
Required input is missing.
Required input is missing.
Required input is missing.
Required input is missing.
```

```
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-dictOfObject-1-row"]//div[@class="error"]'))
The entered value is not a valid integer literal.
The entered value is not a valid integer literal.
Constraint not satisfied
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.remove'] = '1'
>>> submit['form.widgets.dictOfObject.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_dict_edit_remove_textline.html')
>>> print(testing.plainText(content))
DictOfObject label

Object key *
[subob1]
Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
```

(continues on next page)

(continued from previous page)

```
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
[14/06/20]
[Add]
[Remove selected]
[Apply]
```

The object is unchanged:

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>}
```

wrong input (Date)

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.widgets.multiDate'] = 'foobar'
```

```
>>> submit['form.widgets.dictOfObject.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get “The datetime string did not match the pattern” for “foobar” and a new input.

```
>>> content = getForm(request, 'ObjectMulti_dict_edit_submit_date.html')
>>> print(testing.plainText(content))
DictOfObject label

Object key *
```

(continues on next page)

(continued from previous page)

```
[subob1]
Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]
Object key *
[]
Object label *
[ ]
Int label *
[]
Bool label *
( ) yes ( ) no
Choice label *
[[    ]]
ChoiceOpt label
[No value]
TextLine label *
[]
Date label *
[]
[Add]
[Remove selected]
[Apply]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_dict_edit_submit_date_again.html')
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-dictOfObject-1-row"]//div[@class="error"]'))
The entered value is not a valid integer literal.
The entered value is not a valid integer literal.
Constraint not satisfied
The datetime string did not match the pattern 'yy/MM/dd'.
```

Fill in a valid value:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.widgets.multiDate'] = '14/06/21'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_dict_edit_submit_date2.html')
>>> print(testing.plainText(content))
DictOfObject label Object key *
Required input is missing.
[]
Object label *
An object failed schema or invariant validation.
[ ]
Int label *
Required input is missing.
[]
Bool label *
Required input is missing.
( ) yes ( ) no
Choice label *
[one]
ChoiceOpt label
[No value]
TextLine label *
Required input is missing.
[]
Date label *
[14/06/21]
Object key *
[subob1]
Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]
[Add] [Remove selected]
[Apply]
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
```

(continues on next page)

(continued from previous page)

```
>>> submit['form.widgets.dictOfObject.0.remove'] = '1'
>>> submit['form.widgets.dictOfObject.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_dict_edit_remove_date.html')
>>> print(testing.plainText(content))
DictOfObject label

Object key *
[subob1]
Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]
[Add]
[Remove selected]
[Apply]
```

The object is unchanged:

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>}
```

Fix values

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.widgets.multiInt'] = '1042'
>>> submit['form.widgets.dictOfObject.0.widgets.multiTextLine'] = 'moo900'
>>> submit['form.widgets.dictOfObject.0.widgets.multiDate'] = '14/06/23'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_dict_edit_fix_values.html')
>>> print(testing.plainText(content))
DictOfObject label

Object key *
[subob1]
Object label *
[ ]
Int label *
[1,042]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[moo900]
Date label *
[14/06/23]
[Add]
[Remove selected]
[Apply]
```

The object is unchanged:

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>}
```

And apply

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content))
Data successfully updated.

DictOfObject label

Object key *
[subob1]
Object label *
[ ]
Int label *
[1,042]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[moo900]
Date label *
[14/06/23]
[Add]
[Remove selected]
[Apply]
```

Now the object gets updated:

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 23)
  multiInt: 1042
  multiTextLine: 'moo900'>}
```

Twisting some keys

Change key values, item values must stick to the new values.

```
>>> sub1 = testing.ObjectWidgetMultiSubIntegration(
...     multiInt=-100,
...     multiBool=False,
...     multiChoice='two',
...     multiChoiceOpt='six',
...     multiTextLine='some text one',
...     multiDate=date(2014, 6, 20))
```

```
>>> sub2 = testing.ObjectWidgetMultiSubIntegration(
...     multiInt=42,
...     multiBool=True,
...     multiChoice='one',
...     multiChoiceOpt='four',
...     multiTextLine='second txt',
...     multiDate=date(2011, 3, 15))
```

```
>>> obj.dictOfObject = {'subob1': sub1, 'subob2': sub2}
```

```
>>> request = testing.TestRequest()
>>> content = getForm(request, 'ObjectMulti_dict_edit_twist.html')
```

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.key.0'] = 'twisted'  # was subob1
```

```
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

```
>>> content = getForm(request, 'ObjectMulti_dict_edit_twist2.html')
```

```
>>> pprint(obj.dictOfObject)
{'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>,
 'twisted': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>}
```

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.key.1'] = 'subob2'  # was twisted
>>> submit['form.widgets.dictOfObject.key.0'] = 'subob1'  # was subob2
```

```
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

```
>>> content = getForm(request, 'ObjectMulti_dict_edit_twist2.html')
```

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>,
 'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>}
```

Bool was misbehaving

```
>>> sub1 = testing.ObjectWidgetMultiSubIntegration(
...     multiInt=-100,
...     multiBool=False,
...     multiChoice='two',
...     multiChoiceOpt='six',
...     multiTextLine='some text one',
...     multiDate=date(2014, 6, 20))
```

```
>>> sub2 = testing.ObjectWidgetMultiSubIntegration(
...     multiInt=42,
...     multiBool=True,
...     multiChoice='one',
...     multiChoiceOpt='four',
...     multiTextLine='second txt',
...     multiDate=date(2011, 3, 15))
```

```
>>> obj.dictOfObject = {'subob1': sub1, 'subob2': sub2}
```

```
>>> request = testing.TestRequest()
>>> content = getForm(request)
```

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.widgets.multiBool'] = 'true'
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

```
>>> content = getForm(request)
>>> print(testing.plainText(content))
Data successfully updated.
...
```

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>}
```

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.dictOfObject.0.widgets.multiBool'] = 'false'
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
```

```
>>> content = getForm(request)
>>> print(testing.plainText(content))
Data successfully updated.
...
```

```
>>> pprint(obj.dictOfObject)
{'subob1': <ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 'subob2': <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>}
```

4.13.5 ObjectWidget integration with MultiWidget of list

a.k.a. list of objects widget

```
>>> from datetime import date
>>> from z3c.form import form
>>> from z3c.form import field
>>> from z3c.form import testing
```

```
>>> from z3c.form.object import registerFactoryAdapter
>>> registerFactoryAdapter(testing.IObjectWidgetMultiSubIntegration,
...     testing.ObjectWidgetMultiSubIntegration)
```

```
>>> request = testing.TestRequest()
```

```
>>> class EForm(form.EditForm):
...     form.extends(form.EditForm)
...     fields = field.Fields(
...         testing.IMultiWidgetListIntegration).select('listOfObject')
```

Our multi content object:

```
>>> obj = testing.MultiWidgetListIntegration()
```

We recreate the form each time, to stay as close as possible. In real life the form gets instantiated and destroyed with each request.

```
>>> def getForm(request, fname=None):
...     frm = EForm(obj, request)
...     testing.addTemplate(frm, 'integration_edit.pt')
...     frm.update()
...     content = frm.render()
...     if fname is not None:
...         testing.saveHtml(content, fname)
...     return content
```

Empty

All blank and empty values:

```
>>> content = getForm(request, 'ObjectMulti_list_edit_empty.html')
```

```
>>> print(testing.plainText(content))
ListOfObject label

[Add]
[Apply]
```

Some valid default values

```
>>> sub1 = testing.ObjectWidgetMultiSubIntegration(
...     multiInt=-100,
...     multiBool=False,
...     multiChoice='two',
...     multiChoiceOpt='six',
...     multiTextLine='some text one',
...     multiDate=date(2014, 6, 20))
```

```
>>> sub2 = testing.ObjectWidgetMultiSubIntegration(
...     multiInt=42,
...     multiBool=True,
...     multiChoice='one',
...     multiChoiceOpt='four',
...     multiTextLine='second txt',
...     multiDate=date(2011, 3, 15))
```

```
>>> obj.listOfObject = [sub1, sub2]
```

```
>>> pprint(obj.listOfObject)
[<ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>]
```

```
>>> content = getForm(request, 'ObjectMulti_list_edit_simple.html')
>>> print(testing.plainText(content))
ListOfObject label

Object label *
[ ]
Int label *
[-100]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[some text one]
```

(continues on next page)

(continued from previous page)

```
Date label *
[14/06/20]
Object label *
[ ]
Int label *
[42]
Bool label *
(O) yes ( ) no
Choice label *
[one]
ChoiceOpt label
[four]
TextLine label *
[second txt]
Date label *
[11/03/15]
[Add]
[Remove selected]
[Apply]
```

wrong input (Int)

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.0.widgets.multiInt'] = 'foobar'
```

```
>>> submit['form.widgets.listOfObject.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get “The entered value is not a valid integer literal.” for “foobar” and a new input.

```
>>> content = getForm(request, 'ObjectMulti_list_edit_submit_int.html')
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-listOfObject-0-row"]'))
Object label *

The entered value is not a valid integer literal.

[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
```

(continues on next page)

(continued from previous page)

```
[some text one]
Date label *
[14/06/20]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_list_edit_submit_int_again.html')
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-listOfObject-0-row"]//div[@class="error"]'))
The entered value is not a valid integer literal.
The entered value is not a valid integer literal.
```

```
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-listOfObject-1-row"]//div[@class="error"]'))
```

```
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-listOfObject-2-row"]'))
Object label *

An object failed schema or invariant validation.

[ ]
Int label *
Required input is missing.
[]
Bool label *
Required input is missing.
( ) yes ( ) no
Choice label *
[one]
ChoiceOpt label
[No value]
TextLine label *
Required input is missing.
[]
Date label *
Required input is missing.
[]
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.1.remove'] = '1'
>>> submit['form.widgets.listOfObject.2.remove'] = '1'
>>> submit['form.widgets.listOfObject.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_list_edit_remove_int.html')
>>> print(testing.plainText(content))
ListOfObject label
```

(continues on next page)

(continued from previous page)

```
Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[some text one]
Date label *
[14/06/20]
[Add]
[Remove selected]
[Apply]
```

The object is unchanged:

```
>>> pprint(obj.listOfObject)
[<ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>]
```

wrong input (TextLine)

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.0.widgets.multiTextLine'] = 'foo\nbar'
```

```
>>> submit['form.widgets.listOfObject.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get “Constraint not satisfied” for “foonbar” and a new input.

```
>>> content = getForm(request, 'ObjectMulti_list_edit_submit_textline.html')
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-listOfObject-0-row"]'))
Object label *

The entered value is not a valid integer literal.

[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
[14/06/20]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_list_edit_submit_textline_again.html')
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-listOfObject-0-row"]//div[@class="error"]'))
The entered value is not a valid integer literal.
The entered value is not a valid integer literal.
Constraint not satisfied
```

```
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-listOfObject-1-row"]//div[@class="error"]'))
An object failed schema or invariant validation.
Required input is missing.
Required input is missing.
Required input is missing.
Required input is missing.
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.1.remove'] = '1'
>>> submit['form.widgets.listOfObject.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_list_edit_remove_textline.html')
>>> print(testing.plainText(content))
ListOfObject label
```

(continues on next page)

(continued from previous page)

```
Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
[14/06/20]
[Add]
[Remove selected]
[Apply]
```

The object is unchanged:

```
>>> pprint(obj.listOfObject)
[<ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>]
```

wrong input (Date)

Set a wrong value and add a new input:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.0.widgets.multiDate'] = 'foobar'
```

```
>>> submit['form.widgets.listOfObject.buttons.add'] = 'Add'
```

```
>>> request = testing.TestRequest(form=submit)
```

Important is that we get “The datetime string did not match the pattern” for “foobar” and a new input.

```
>>> content = getForm(request, 'ObjectMulti_list_edit_submit_date.html')
>>> print(testing.plainText(content))
ListOfObject label

Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]
Object label *
[ ]
Int label *
[]
Bool label *
( ) yes ( ) no
Choice label *
[[    ]]
ChoiceOpt label
[No value]
TextLine label *
[]
Date label *
[]
[Add]
[Remove selected]
[Apply]
```

Submit again with the empty field:

```
>>> submit = testing.getSubmitValues(content)
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content,
...     './/div[@id="form-widgets-listOfObject-0-row"]//div[@class="error"]'))
The entered value is not a valid integer literal.
The entered value is not a valid integer literal.
Constraint not satisfied
The datetime string did not match the pattern 'yy/MM/dd'.
```

Add one more field:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.buttons.add'] = 'Add'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
```

And fill in a valid value:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.2.widgets.multiDate'] = '14/06/21'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_list_edit_submit_date2.html')
>>> print(testing.plainText(content))
ListOfObject label Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]
Object label *
An object failed schema or invariant validation.
[ ]
Int label *
Required input is missing.
[]
Bool label *
Required input is missing.
( ) yes ( ) no
Choice label *
[one]
ChoiceOpt label
[No value]
TextLine label *
Required input is missing.
[]
Date label *
Required input is missing.
[]
Object label *
An object failed schema or invariant validation.
[ ]
```

(continues on next page)

(continued from previous page)

```
Int label *
Required input is missing.
[]
Bool label *
Required input is missing.
( ) yes ( ) no
Choice label *
[one]
ChoiceOpt label
[No value]
TextLine label *
Required input is missing.
[]
Date label *
[14/06/21]
[Add] [Remove selected]
[Apply]
```

Let's remove some items:

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.2.remove'] = '1'
>>> submit['form.widgets.listOfObject.buttons.remove'] = 'Remove selected'
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_list_edit_remove_date.html')
>>> print(testing.plainText(content))
ListOfObject label

Object label *
The entered value is not a valid integer literal.
[ ]
Int label *
The entered value is not a valid integer literal.
[foobar]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
Constraint not satisfied
[foo
bar]
Date label *
The datetime string did not match the pattern 'yy/MM/dd'.
[foobar]
Object label *
An object failed schema or invariant validation.
[ ]
Int label *
Required input is missing.
```

(continues on next page)

(continued from previous page)

```
[]
Bool label *
Required input is missing.
( ) yes ( ) no
Choice label *
[one]
ChoiceOpt label
[No value]
TextLine label *
Required input is missing.
[]
Date label *
Required input is missing.
[]
[Add] [Remove selected]
[Apply]
```

The object is unchanged:

```
>>> pprint(obj.listOfObject)
[<ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>]
```

Fix values

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.0.widgets.multiInt'] = '1042'
>>> submit['form.widgets.listOfObject.0.widgets.multiTextLine'] = 'moo900'
>>> submit['form.widgets.listOfObject.0.widgets.multiDate'] = '14/06/23'
```

```
>>> submit['form.widgets.listOfObject.1.remove'] = '1'
>>> submit['form.widgets.listOfObject.buttons.remove'] = 'Remove selected'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request, 'ObjectMulti_list_edit_fix_values.html')
>>> print(testing.plainText(content))
ListOfObject label

```

(continues on next page)

(continued from previous page)

```
Object label *
[ ]
Int label *
[1,042]
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[moo900]
Date label *
[14/06/23]
[Add]
[Remove selected]
[Apply]
```

The object is unchanged:

```
>>> pprint(obj.listOfObject)
[<ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 20)
  multiInt: -100
  multiTextLine: 'some text one'>,
 <ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'one'
  multiChoiceOpt: 'four'
  multiDate: datetime.date(2011, 3, 15)
  multiInt: 42
  multiTextLine: 'second txt'>]
```

And apply

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content))
Data successfully updated.

ListOfObject label

Object label *
[ ]
Int label *
[1,042]
```

(continues on next page)

(continued from previous page)

```
Bool label *
( ) yes (O) no
Choice label *
[two]
ChoiceOpt label
[six]
TextLine label *
[moo900]
Date label *
[14/06/23]
[Add]
[Remove selected]
[Apply]
```

Now the object gets updated:

```
>>> pprint(obj.listOfObject)
[<ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 23)
  multiInt: 1042
  multiTextLine: 'moo900'>]
```

Bool was misbehaving

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.0.widgets.multiBool'] = 'true'
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content))
Data successfully updated.
...
```

```
>>> pprint(obj.listOfObject)
[<ObjectWidgetMultiSubIntegration
  multiBool: True
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 23)
  multiInt: 1042
  multiTextLine: 'moo900'>]
```

```
>>> submit = testing.getSubmitValues(content)
>>> submit['form.widgets.listOfObject.0.widgets.multiBool'] = 'false'
>>> submit['form.buttons.apply'] = 'Apply'
```

```
>>> request = testing.TestRequest(form=submit)
>>> content = getForm(request)
>>> print(testing.plainText(content))
Data successfully updated.
...
```

```
>>> pprint(obj.listOfObject)
[<ObjectWidgetMultiSubIntegration
  multiBool: False
  multiChoice: 'two'
  multiChoiceOpt: 'six'
  multiDate: datetime.date(2014, 6, 23)
  multiInt: 1042
  multiTextLine: 'moo900'>]
```

4.14 Button Widget

The button widget allows you to provide buttons whose actions are defined using Javascript scripts. The "button" type of the "INPUT" element is described here:

http://www.w3.org/TR/1999/REC-html401-19991224/interact/forms.html#edef-INPUT

As for all widgets, the button widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import button
```

```
>>> verifyClass(interfaces.IWidget, button.ButtonWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = button.ButtonWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget.id'
>>> widget.name = 'widget.name'
```

We also need to register the template for the widget:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('button_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.IButtonWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get a simple input element:

```
>>> print(widget.render())
<input type="button" id="widget.id" name="widget.name"
       class="button-widget" />
```

Setting a value for the widget effectively changes the button label:

```
>>> widget.value = 'Button'
>>> print(widget.render())
<input type="button" id="widget.id" name="widget.name"
       class="button-widget" value="Button" />
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> widget.request = TestRequest(form={'widget.name': 'button'})
>>> widget.update()
>>> widget.extract()
'button'
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = TestRequest()
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

4.15 Submit Widget

The submit widget allows you to upload a new submit to the server. The "submit" type of the "INPUT" element is described here:

http://www.w3.org/TR/1999/REC-html401-19991224/interact/forms.html#edef-INPUT

As for all widgets, the submit widget must provide the new `IWidget` interface:

```
>>> from zope.interface.verify import verifyClass
>>> from z3c.form import interfaces
>>> from z3c.form.browser import submit
```

```
>>> verifyClass(interfaces.IWidget, submit.SubmitWidget)
True
```

The widget can be instantiated only using the request:

```
>>> from z3c.form.testing import TestRequest
>>> request = TestRequest()
```

```
>>> widget = submit.SubmitWidget(request)
```

Before rendering the widget, one has to set the name and id of the widget:

```
>>> widget.id = 'widget.id'
>>> widget.name = 'widget.name'
```

We also need to register the template for the widget:

```
>>> import zope.component
>>> from zope.pagetemplate.interfaces import IPageTemplate
>>> from z3c.form.testing import getPath
>>> from z3c.form.widget import WidgetTemplateFactory
```

```
>>> zope.component.provideAdapter(
...     WidgetTemplateFactory(getPath('submit_input.pt'), 'text/html'),
...     (None, None, None, None, interfaces.ISubmitWidget),
...     IPageTemplate, name=interfaces.INPUT_MODE)
```

If we render the widget we get a simple input element:

```
>>> print(widget.render())
<input type="submit" id="widget.id" name="widget.name"
       class="submit-widget" />
```

Setting a value for the widget effectively changes the button label:

```
>>> widget.value = 'Submit'
>>> print(widget.render())
<input type="submit" id="widget.id" name="widget.name"
       class="submit-widget" value="Submit" />
```

Let's now make sure that we can extract user entered data from a widget:

```
>>> widget.request = TestRequest(form={'widget.name': 'submit'})
>>> widget.update()
>>> widget.extract()
'submit'
```

If nothing is found in the request, the default is returned:

```
>>> widget.request = TestRequest()
>>> widget.update()
>>> widget.extract()
<NO_VALUE>
```

CHAPTER

FIVE

API DOCUMENTATION

5.1 Interfaces

5.1.1 Model interfaces

Form and Widget Framework Interfaces

5.1.2 Browser interfaces

Browser Widget Framework Interfaces

CHAPTER

SIX

DEVELOPMENT

6.1 Changelog

6.1.1 5.1 (2023-07-19)

- HTMLFormElement.attributes: Allow to extend HTML attributes programmatically.

6.1.2 5.0 (2023-07-17)

- Add support for Python 3.11.
- Drop support for Python 2.7, 3.5, 3.6.
- Drop deprecated support for `python setup.py test`.
- HTMLFormElement.addClass: Improve removal of duplicates. It's now possible to add multiple classes as whitespace separated string and still detect class duplicates.

6.1.3 4.3 (2022-03-24)

- Add support for Python 3.10.
- Update tests to `lxml > 4.7`, thus requiring at least that version. (#107)
- Remove unused `ISubformFactory`. (#110) [petschki]

6.1.4 4.2 (2021-07-29)

- Fix `MultiConverter.toFieldValue` tuple typed field support (when `field._type` is a tuple of types, not a single type).
- Add Python 3.9 compatibility and testing.
- Apply `zopefoundation.meta` config
- Fix tests for the `zope.schema.Bool` required default change.
- Fix tests for the `zope.interface repr()` change.
- Fix compatibility with changed repeat syntax. Fixes issue 94.
- Some fixes in spanish translation. [erral]
- Drop support for Python 3.4.

- Add support for Python 3.8b4.
- Try to fix the buggy hidden mode behaviour of checkbox widgets. Fixes issue 89.

6.1.5 4.1.2 (2019-03-04)

- Fix an edge case when field `missing_value` is not `None` but a custom value that works as `None`. That ended up calling `zope.i18n NumberFormat.format` with `None` what then failed.

6.1.6 4.1.1 (2018-11-26)

- Fix `FieldWidgets.copy()`. It was broken since `SelectionManager` was reimplemented using `OrderedDict`.

6.1.7 4.1.0 (2018-11-15)

- Add support for Python 3.7.
- Deal with items with same name but different values in ordered field widget. [rodfersou]
- Move homegrown Manager implementation to `OrderedDict`. [tomgross]
- Adapt tests to `lxml >= 4.2`, `zope.configuration >= 4.3` and `zope.schema >= 4.7`.

6.1.8 4.0.0 (2017-12-20)

- Upgrade the major version 4 to reflect the breaking changes in 3.3.0. (Version 3.6 will be a re-release of 3.2.x not containing the changes since 3.3.0 besides cherry-picks.) Fixes: https://github.com/zopefoundation/z3c.form/issues/41
- Host documentation at https://z3cform.readthedocs.io

6.1.9 3.5.0 (2017-09-19)

- Add support for Python 3.6.
- Drop support for Python 3.3.
- Avoid duplicated IDs when using a non-required field with `z3c.formwidget.query.widget.QuerySourceRadioWidget`. [pgrunewald]

6.1.10 3.4.0 (2016-11-15)

- Drop support for Python 2.6.
- Support Python 3.5 officially.
- Fix TypeError: object of type 'generator' has no `len()`. Happens with z3c.formwidget.query. [maurits]
- Turned `items` into a property again on all widgets. For the select widget it was a method since 2.9.0. For the radio and checkbox widgets it was a method since 3.2.10. For orderedselect and multi it was always a property. Fixes https://github.com/zopefoundation/z3c.form/issues/44 [maurits]
- Fix handling of missing terms in collections. (See version 2.9 describing this feature.)

- Fix `orderedselect_input.js` resource to be usable on browser layers which do not extend `zope.publisher.interfaces.browser.IDefaultBrowserLayer`.

6.1.11 3.3.0 (2016-03-09)

- *MAJOR* overhaul of ObjectWidget:
  - low level unittests passed, but high level was not tops basic rule is that widgets want RAW values and all conversion must be done in `ObjectConverter`
  - `ObjectSubForm` and `SubformAdapter` is removed, it was causing more problems than good
  - added high level integration tests
- Removed `z3c.coverage` from `test` extra. [gforcada, maurits]

6.1.12 3.2.10 (2016-03-09)

- RadioWidget items are better determined when they are needed [agroszer]
- CheckBoxWidget items are better determined when they are needed [agroszer]
- Bugfix: The `ChoiceTerms` adapter blindly assumed that the passed in field is unbound, which is not necessarily the case in interesting ObjectWidget scenarios. Not it checks for a non-None field context first. [srichter]

6.1.13 3.2.9 (2016-02-01)

- Correctly handled `noValueToken` in RadioWidget. This avoids a `LookupError: --NOVALUE--`. [gaudenz,ale-rt]
- Added `json` method for forms and `json_data` method for widgets. [mmilkin]
- Change javascript for updating ordered select widget hidden structure so it works again on IE11 and doesn't send back an empty list that deletes all selections on save. Fixes https://github.com/zopefoundation/z3c.form/issues/23 [fredvd]
- Started on Dutch translations. [maurits]

6.1.14 3.2.8 (2015-11-09)

- Standardized namespace `__init__`. [agroszer]

6.1.15 3.2.7 (2015-09-20)

- Remove "cannot move farther up/down" messages in ordered select widget. [esteele]
- Updated Traditional Chinese translation. [l34marr]

6.1.16 3.2.6 (2015-09-10)

- Fixed warnings in headers of locales files. Checked with `msgfmt -c`. [maurits]
- Added Finnish translation. [petri]
- Added Traditional Chinese translation. [l34marr]

6.1.17 3.2.5 (2015-09-09)

- Fixed error on Python 3: NameError: global name 'basestring' is not defined. This fixes a bug introduced in version 3.2.1. [maurits]

6.1.18 3.2.4 (2015-07-18)

- Fix ordered select input widget not working. [vangheem]
- ReSt fix. [timo]

6.1.19 3.2.3 (2015-03-21)

- 3.2.2 was a brown bag release. Fix MANIFEST.in to include the js file that has been added in 3.2.2. [timo]

6.1.20 3.2.2 (2015-03-21)

- move js to separate file to prevent escaped entities in Plone 5. [pbauer]

6.1.21 3.2.1 (2014-06-09)

- Add DataExtractedEvent, which is thrown after data and errors are extracted from widgets. Fixes https://github.com/zopefoundation/z3c.form/pull/18
- Remove spaces at start and end of text field values.
- Explicitly hide span in `orderedselect_input.pt`. This only contains hidden inputs, but Internet Explorer 10 was showing them anyway. Fixes https://github.com/zopefoundation/z3c.form/issues/19

6.1.22 3.2.0 (2014-03-18)

- Feature: Added text and password widget HTML5 attributes required by plone.login.

6.1.23 3.1.1 (2014-03-02)

- Feature: Added a consistent id on single checkbox and multi checkbox widgets.

6.1.24 3.1.0 (2013-12-02)

- Feature: Added a consistent id on ordered selection widget.
- Feature: Added a hidden template for the textlines widget.
- Feature: added an API to render each radio button separately.

6.1.25 3.0.5 (2013-10-09)

- Bug: Remove errors for cases where the key field of a dict field uses a sequence widget (most notably choices). The sequence widget always returns lists as widget values, which are not hashable. We convert those lists to tuples now within the dict support.

6.1.26 3.0.4 (2013-10-06)

- Feature: Moved registration of translation directories to a separate ZCML file.
- Bug: Fixed a typo in German translations.

6.1.27 3.0.3 (2013-09-06)

- Feature: Version 2.9 introduced a solution for missing terms in vocabularies. Adapted sources to this solution, too.

6.1.28 3.0.2 (2013-08-14)

- Bug: Fix unicode decode error in weird cases in `checkbox.CheckboxWidget.update()` and `radio.RadioWidget.update()` (eg: when `term.value` is an Plone Archetype ATFile)

6.1.29 3.0.1 (2013-06-25)

- Bug: The alpha slipped out as 3.0.0, removed `ZODB-4.0.0dev.tar.gz` to reduce damage
- Bug: Fixed a bug in `widget.py def wrapCSSClass`

6.1.30 3.0.0 (2013-06-24)

- Feature: Added support for `IDict` field in `MultiWidget`.
- Bug: Only add the 'required' CSS class to widgets when they are in input mode.
- Bug: Catch bug where if a select value was set as from hidden input or through a rest url as a single value, it won't error out when trying to remove from ignored list. Probably not the 100% right fix but it catches core dumps and is sane anyways.

6.1.31 3.0.0a3 (2013-04-08)

- Feature: Updated pt_BR translation.
- Bug: Fixed a bug where file input value was interpeted as UTF-8.

6.1.32 3.0.0a2 (2013-02-26)

- Bug: The 3.0.0a1 release was missing some files (e.g. `locales`) due to an incomplete `MANIFEST.in`.

6.1.33 3.0.0a1 (2013-02-24)

- Feature: Removed several parts to be installed by default, since some packages are not ported yet.
- Feature: Added support for Python 3.3.
- Feature: Replaced deprecated `zope.interface.implements` usage with equivalent `zope.interface.implementer` decorator.
- Feature: Dropped support for Python 2.4 and 2.5.
- Bug: Make sure the call to the method that returns the default value is made with a field which has its context bound.

6.1.34 2.9.1 (2012-11-27)

- Feautre: The `updateWidgets` method has received an argument `prefix` which allows setting the prefix of the field widgets adapter.

  This allows updating the common widgets prefix before the individual widgets are updated, useful for situations where neither a form, nor a widgets prefix is desired.
- Bug: Capitalize the messages 'no value' and 'select a value'. This change has been applied also to the existing translations (where applicable).
- Bug: `TextLinesConverter`: Do not ignore newlines at the end of the inputted string, thus do not eat blank items
- Bug: `TextLinesConverter`: `toFieldValue()`, convert conversion exceptions to `FormatterValidationError`, for cases like got a string instead of int.

6.1.35 2.9.0 (2012-09-17)

- Feature: Missing terms in vocabularies: this was a pain until now. Now it's possible to have the same (missing) value unchanged on the object with an EditForm after save as it was before editing. That brings some changes with it:
  - *MAJOR*: unchanged values/fields do not get validated anymore (unless they are empty or are FileUploads)
  - A temporary `SimpleTerm` gets created for the missing value Title is by default "Missing: ${value}". See MissingTermsMixin.
- Feature: Split `configure.zcml`
- Bug: `SequenceWidget` DISPLAY_MODE: silently ignore missing tokens, because INPUT_MODE and HIDDEN_MODE does that too.

6.1.36 2.8.2 (2012-08-17)

- Feature: Added `IForm.ignoreRequiredOnValidation`, `IWidgets.ignoreRequiredOnValidation`, `IWidget.ignoreRequiredOnValidation`. Those enable `extract` and `extractData` to return without errors in case a required field is not filled. That also means the usual "Missing value" error will not get displayed. But the `required-info` (usually the *) yes. This is handy to store partial state.

6.1.37 2.8.1 (2012-08-06)

- Fixed broken release, my python 2.7 windows setup didn't release the new widget.zcml, widget_layout.pt and widget_layout_hidden.pt files. After enhance the pattern in MANIFEST.in everything seems fine. That's probably because I patched my python version with the *build exclude pattern patch. And yes, the new files where added to the svn repos! After deep into this again, it seems that only previous added *.txt, *.pt files get added to the release. A fresh checkout sdist release only contains the *.py and *.mo files. Anyway the enhanced MANIFEST.in file solved the problem.

6.1.38 2.8.0 (2012-08-06)

- Feature: Implemented widget layout concept similar to z3c.pagelet. The new layout concept allows to register layout templates additional to the widget templates. Such a layout template only get used if a widget get called. This enhacement is optional and compatible with all previous z3c.form versions and doesn't affect existing code and custom implementations except if you implemented a own `__call__` method for widgets which wasn't implemented in previous versions. The new `__call__` method will lookup and return a layout template which supports additional HTML code used as a wrapper for the HTML code returned from the widget render method. This concept allows to define additional HTML construct provided for all widget and render specific CSS classes arround the widget per context, view, request, etc discriminators. Such a HTML constuct was normaly supported in form macros which can't get customized on a per widget, view or context base.

  Summary; the new layout concept allows us to define a wrapper CSS elements for the widget element (label, widget, error) on a per widgte base and skip the generic form macros offered from z3c.formui.

  Note; you only could get into trouble if you define a widget in tal without to prefix them with `nocall:` e.g. tal:define="widget view/widgets/foo" Just add a nocall like tal:define="widget nocall:view/widgets/foo" if your rendering engine calls the __call__method by default. Also note that the following will also call the `__call__` method `tal:define="widget myWidget"`.

- Fixed content type extraction test which returned different values. This probably depends on a newer version of guess_content_type. Just allow image/x-png and image/pjpeg as valid values.

6.1.39 2.7.0 (2012-07-11)

- Remove `zope34` extra, use an older version of z3c.form if you need to support pre-ZTK versions.
- Require at least zope.app.container 3.7 for adding support.
- Avoid dependency on ZODB3.
- Added IField.showDefault and IWidget.showDefault That controls whether the widget should look for field default values to display. This can be really helpful in EditForms, where you don't want to have default values instead of actual (missing) values. By default it is True to provide backwards compatibility.

6.1.40 2.6.1 (2012-01-30)

- Fixed a potential problem where a non-ascii vocabulary/source term value could cause the checkbox and readio widget to crash.
- Fixed a problem with the `datetime.timedelta` converter, which failed to convert back to the field value, when the day part was missing.

6.1.41 2.6.0 (2012-01-30)

- Remove ":list" from radio inputs, since radio buttons can be only one value by definition. See LP580840.
- Changed radio button and checkbox widget labels from token to value (wrapped by a unicode conversion) to make it consistent with the parent `SequenceWidget` class. This way, edit and display views of the widgets show the same label. See LP623210.
- Remove dependency on zope.site.hooks, which was moved to zope.component in 3.8.0 (present in ZTK 1.0 and above).
- Make zope.container dependency more optional (it is only used in tests)
- Properly escape JS code in script tag for the ordered-select widget. See LP829484.
- Cleaned whitespace in page templates.
- Fix `IGroupForm` interface and actually use it in the `GroupForm` class. See LP580839.
- Added Spanish translation.
- Added Hungarian translation.

6.1.42 2.5.1 (2011-11-26)

- Better compatibility with Chameleon 2.x.
- Added *.mo files missing in version 2.5.0.
- Pinned minimum version of test dependency `z3c.template`.

6.1.43 2.5.0 (2011-10-29)

- Fixed coverage report generator script buildout setup.
- Note: z3c.pt and chameleon are not fully compatible right now with TAL. Traversing the repeat wrapper is not done the same way. ZPT uses the following pattern: <tal:block condition="not:repeat/value/end">, </tal:block>

  Chameleon only supports python style traversing: <tal:block condition="not:python:repeat['value'].end">, </tal:block>
- Upgrade to chameleon 2.0 template engine and use the newest z3c.pt and z3c.ptcompat packages adjusted to work with chameleon 2.0.

  See the notes from the z3c.ptcompat package:

  Update z3c.ptcompat implementation to use component-based template engine configuration, plugging directly into the Zope Toolkit framework.

  The z3c.ptcompat package no longer provides template classes, or ZCML directives; you should import directly from the ZTK codebase.

Also, note that the PREFER_Z3C_PT environment option has been rendered obsolete; instead, this is now managed via component configuration.

Attention: You need to include the configure.zcml file from z3c.ptcompat for enable the z3c.pt template engine. The configure.zcml will plugin the template engine. Also remove any custom built hooks which will import z3c.ptcompat in your tests or other places.

You can directly use the BoundPageTemplate and ViewPageTempalteFile from zope.browserpage.viewpagetemplatefile if needed. This templates will implicit use the z3c.pt template engine if the z3c.ptcompat configure.zcml is loaded.

6.1.44 2.4.4 (2011-07-11)

- Remove unneeded dependency on deprecated `zope.app.security`.
- Fixed ButtonActions.update() to correctly remove actions when called again, after the button condition become false.

6.1.45 2.4.3 (2011-05-20)

- Declare TextLinesFieldWidget as an IFieldWidget implementer.
- Clarify MultiWidget.extract(), when there are zero items, this is now [] instead of <NO_VALUE>
- Some typos fixed
- Fixed test failure due to change in floating point representation in Python 2.7.
- Ensure at least min_length widgets are rendered for a MultiWidget in input mode.
- Added Japanese translation.
- Added base of Czech translation.
- Added Portuguese Brazilian translation.

6.1.46 2.4.2 (2011-01-22)

- Adjust test for the contentprovider feature to not depend on the ContentProviderBase class that was introduced in zope.contentprovider 3.5.0. This restores compatibility with Zope 2.10.
- Security issue, removed IBrowserRequest from IFormLayer. This prevents to mixin IBrowserRequest into non IBrowserRequest e.g. IJSONRPCRequest. This should be compatible since a browser request using z3c.form already provides IBrowserRequest and the IFormLayer is only a marker interface used as skin layer.
- Add English translation (generated from translation template using msgen z3c.form.pot > en/LC_MESSAGES/z3c.form.po).
- Added Norwegian translation, thanks to Helge Tesdal and Martijn Pieters.
- Updated German translation.

6.1.47 2.4.1 (2010-07-18)

- Since version 2.3.4 `applyChanges` required that the value exists when the field had a `DictionaryField` data manager otherwise it broke with an `AttributeError`. Restored previous behavior that values need not to be exist before `applyChanges` was called by using `datamanager.query()` instead of `datamanager.get()` to get the previous value.
- Added missing dependency on `zope.contentprovider`.
- No longer using deprecated `zope.testing.doctest` by using python's built-in `doctest` module.

6.1.48 2.4.0 (2010-07-01)

- Feature: mix fields and content providers in forms. This allow to enrich the form by interlacing html snippets produced by content providers. Adding html outside the widgets avoids the systematic need of subclassing or changing the full widget rendering.
- Bug: Radio widget was not treating value as a list in hidden mode.

6.1.49 2.3.4 (2010-05-17)

- Bugfix: applyChanges should not try to compare old and new values if the old value can not be accessed.
- Fix DictionaryField to conform to the IDataManager spec: get() should raise an exception if no value can be found.

6.1.50 2.3.3 (2010-04-20)

- The last discriminator of the 'message' IValue adapter used in the ErrorViewSnippet is called 'content', but it was looked up as the error view itself. It is now looked up on the form's context.
- Don't let util.getSpecification() generate an interface more than once. This causes strange effects when used in value adapters: if two adapters use e.g. ISchema['some_field'] as a "discriminator" for 'field', with one adapter being more specific on a discriminator that comes later in the discriminator list (e.g. 'form' for an ErrorViewMessage), then depending on the order in which these two were set up, the adapter specialisation may differ, giving unexpected results that make it look like the adapter registry is picking the wrong adapter.
- Fix trivial test failures on Python 2.4 stemming from differences in pprint's sorting of dicts.
- Don't invoke render() when publishing the form as a view if the HTTP status code has been set to one in the 3xx range (e.g. a redirect or not-modified response) - the response body will be ignored by the browser anyway.
- Handle Invalid exceptions from constraints and field validators.
- Don't create unnecessary self.items in update() method of SelectWidget in DISPLAY_MODE. Now items is a property.
- Add hidden widget templates for radio buttons and checkboxes.

6.1.51 2.3.2 (2010-01-21)

- Reverted changes made in the previous release as the `getContent` method can return anything it wants to as long as a data manager can map the fields to it. So `context` should be used for group instantiation. In cases where `context` is not wanted, the group can be instantiated in the `update` method of its parent group or form. See also https://mail.zope.org/pipermail/zope-dev/2010-January/039334.html

  (So version 2.3.2 is the same as version 2.3.0.)

6.1.52 2.3.1 (2010-01-18)

- `GroupForm` and `Group` now use `getContent` method when instantiating group classes instead of directly accessing `self.context`.

6.1.53 2.3.0 (2009-12-28)

Refactoring

- Removed deprecated zpkg slug and ZCML slugs.
- Adapted tests to `zope.schema` 3.6.0.
- Avoid to use `zope.testing.doctestunit` as it is now deprecated.

Update

- Updated German translations.

6.1.54 2.2.0 (2009-10-27)

- Feature: Add `z3c.form.error.ComputedErrorViewMessage` factory for easy creation of dynamically computed error messages.
- Bug: was generated twice for MultiWidget and ObjectWidget in input mode.
- Bug: Replace dots with hyphens when generating form id from its name.
- Refactored OutputChecker to its own module to allow using `z3c.form.testing` without needing to depend on `lxml`.
- Refactored: Folded duplicate code in `z3c.form.datamanager.AttributeField` into a single property.

6.1.55 2.1.0 (2009-07-22)

- Feature: The `DictionaryFieldManager` now allows all mappings (`zope.interface.common.mapping.IMapping`), even `persistent.mapping.PersistentMapping` and `persistent.dict.PersistentDict`. By default, however, the field manager is only registered for dict, because it would otherwise get picked up in undesired scenarios.
- Bug: Updated code to pass all tests on the latest package versions.

- Bug: Completed the Zope 3.4 backwards-compatibility. Also created a buidlout configuration file to test the Zope 3.4 compatibility. Note: You *must* use the 'latest' or 'zope34' extra now to get all required packages. Alternatively, you can specify the packages listed in either of those extras explicitely in your product's required packages.

6.1.56 2.0.0 (2009-06-14)

Features

- KGS 3.4 compatibility. This is a real hard thing, because `z3c.form` tests use `lxml` >= 2.1.1 to check test output, but KGS 3.4 has `lxml` 1.3.6. Therefore we agree on that if tests pass with all package versions nailed by KGS 3.4 but `lxml` overridden to 2.1.1 then the `z3c.form` package works with a plain KGS 3.4.
- Removed hard `z3c.ptcompat` and thus `z3c.pt` dependency. If you have `z3c.ptcompat` on the Python path it will be used.
- Added nested group support. Groups are rendered as fieldsets. Nested fieldsets are very useful when designing forms.

  WARNING: If your group did have an `applyChanges()` (or any added(?)) method the new one added by this change might not match the signature.
- Added `labelRequired` and `requiredInfo` form attributes. This is useful for conditional rendering a required info legend in form templates. The `requiredInfo` label depends by default on a given `labelRequired` message id and will only return the label if at least one widget field is required.
- Add support for refreshing actions after their execution. This is useful when button action conditions are changing as a result of action execution. All you need is to set the `refreshActions` flag of the form to `True` in your action handler.
- Added support for using sources. Where it was previosly possible to use a vocabulary it is now also possible to use a source. This works both for basic and contextual sources.

  IMPORTANT: The `ChoiceTerms` and `CollectionTerms` in `z3c.form.term` are now simple functions that query for real `ITerms` adapters for field's `source` or `value_type` respectively. So if your code inherits the old `ChoiceTerms` and `CollectionTerms` classes, you'll need to review and adapt it. See the `z3c.form.term` module and its documentation.
- The new `z3c.form.interfaces.NOT_CHANGED` special value is available to signal that the current value should be left as is. It's currently handled in the `z3c.form.form.applyChanges()` function.
- When no file is specified in the file upload widget, instead of overwriting the value with a missing one, the old data is retained. This is done by returning the new `NOT_CHANGED` special value from the `FileUploadDataConvereter`.
- Preliminary support for widgets for the `schema.IObject` field has been added. However, there is a big caveat, please read the `object-caveat.txt` document inside the package.

  A new `objectWidgetTemplate` ZCML directive is provided to register widget templates for specific object field schemas.
- Implemented the `MultiWidget` widget. This widget allows you to use simple fields like `ITextLine`, `IInt`, `IPassword`, etc. in a `IList` or `ITuple` sequence.
- Implemented `TextLinesWidget` widget. This widget offers a text area element and splits lines in sequence items. This is usfull for power user interfaces. The widget can be used for sequence fields (e.g. `IList`) that specify a simple value type field (e.g. `ITextLine` or `IInt`).

- Added a new flag `ignoreContext` to the form field, so that one can individually select which fields should and which ones should not ignore the context.
- Allow raw request values of sequence widgets to be non-sequence values, which makes integration with Javascript libraries easier.
- Added support in the file upload widget's testing flavor to specify 'base64'-encoded strings in the hidden text area, so that binary data can be uploaded as well.
- Allow overriding the `required` widget attribute using `IValue` adapter just like it's done for `label` and `name` attributes.
- Add the `prompt` attribute of the `SequenceWidget` to the list of adaptable attributes.
- Added benchmarking suite demonstrating performance gain when using `z3c.pt`.
- Added support for `z3c.pt`. Usage is switched on via the "PREFER_Z3C_PT" environment variable or via `z3c.ptcompat.config.[enable/diable]()`.
- The `TypeError` message used when a field does not provide `IFormUnicode` now also contains the type of the field.
- Add support for internationalization of `z3c.form` messages. Added Russian, French, German and Chinese translations.
- Sphinx documentation for the package can now be created using the new `docs` script.
- The widget for fields implementing `IChoice` is now looked up by querying for an adapter for `(field, field.vocabulary, request)` so it can be differentiated according to the type of the source used for the field.
- Move `formErrorsMessage` attribute from `AddForm` and `EditForm` to the `z3c.form.form.Form` base class as it's very common validation status message and can be easily reused (especially when translations are provided).

Refactoring

- Removed compatibility support with Zope 3.3.
- Templates now declare XML namespaces.
- HTML output is now compared using a modified version of the XML-aware output checker provided by `lxml`.
- Remove unused imports, adjust buildout dependencies in `setup.py`.
- Use the `z3c.ptcompat` template engine compatibility layer.

Fixed Bugs

- **IMPORTANT** - The signature of `z3c.form.util.extractFileName` function changed because of spelling mistake fix in argument name. The `allowEmtpyPostFix` is now called `allowEmptyPostfix` (note `Empty` instead of `Emtpy` and `Postfix` instead of `PostFix`).
- **IMPORTANT** - The `z3c.form.interfaces.NOVALUE` special value has been renamed to `z3c.form.interfaces.NO_VALUE` to follow the common naming style. The backward-compatibility `NOVALUE` name is still in place, but the `repr` output of the object has been also changed, thus it may break your doctests.
- When dealing with `Bytes` fields, we should do a null conversion when going to its widget value.
- `FieldWidgets` update method were appending keys and values within each update call. Now the `util.Manager` uses a `UniqueOrderedKeys` implementation which will ensure that we can't add duplicated manager keys. The implementation also ensures that we can't override the `UniqueOrderedKeys` instance with a new list by using a decorator. If this `UniqueOrderedKeys` implementation doesn't fit for all use cases, we should probably use a

customized UserList implementation. Now we can call widgets.update() more then one time without any side effect.

- ButtonActions update where appending keys and values within each update call. Now we can call actions.update() more then one time without any side effect.
- The CollectionSequenceDataConverter no longer throws a TypeError: 'NoneType' object is not iterable when passed the value of a non-required field (which in the case of a List field is None).
- The SequenceDataConverter and CollectionSequenceDataConverter converter classes now ignore values that are not present in the terms when converting to a widget value.
- Use nocall: modifier in orderedselect_input.pt to avoid calling list entry if it is callable.
- SingleCheckBoxFieldWidget doesn't repeat the label twice (once in , and once in the <label> next to the checkbox).
- Don't cause warnings in Python 2.6.
- validator.SimpleFieldValidator is now able to handle interfaces.NOT_CHANGED. This value is set for file uploads when the user does not choose a file for upload.

6.1.57 1.9.0 (2008-08-26)

- Feature: Use the query() method in the widget manager to try extract a value. This ensures that the lookup is never failing, which is particularly helpful for dictionary-based data managers, where dictionaries might not have all keys.
- Feature: Changed the get() method of the data manager to throw an error when the data for the field cannot be found. Added query() method to data manager that returns a default value, if no value can be found.
- Feature: Deletion of widgets from field widget managers is now possible.
- Feature: Groups now produce detailed ObjectModifiedEvent descriptions like regular edit forms do. (Thanks to Carsten Senger for providing a patch.)
- Feature: The widget manager's extract() method now supports an optional setErrors (default value: True) flag that allows one to not set errors on the widgets and widget manager during data extraction. Use case: You want to inspect the entered data and handle errors manually.
- Bug: The ignoreButtons flag of the z3c.form.form.extends() method was not honored. (Thanks to Carsten Senger for providing a patch.)
- Bug: Group classes now implement IGroup. This also helps with the detection of group instantiation. (Thanks to Carsten Senger for providing a patch.)
- Bug: The list of changes in a group were updated incorrectly, since it was assumed that groups would modify mutually exclusive interfaces. Instead of using an overwriting dictionary update() method, a purely additive merge is used now. (Thanks to Carsten Senger for providing a patch.)
- Bug: Added a widget for IDecimal field in testing setup.
- Feature: The z3c.form.util module has a new function, createCSSId() method that generates readable ids for use with css selectors from any unicode string.
- Bug: The applyChanges() method in group forms did not return a changes dictionary, but simply a boolean. This is now fixed and the group form changes are now merged with the main form changes.
- Bug: Display widgets did not set the style attribute if it was available, even though the input widgets did set the style attribute.

6.1.58 1.8.2 (2008-04-24)

- Bug: Display Widgets added spaces (due to code indentation) to the displayed values, which in some cases, like when displaying Python source code, caused the appearance to be incorrect.
- Bug: Prevent to call `__len__` on `ITerms` and use `is None` for check for existence. Because `__len__` is not a part of the ITerms API and `not widget.terms` will end in calling `__len__` on existing terms.

6.1.59 1.8.1 (2008-04-08)

- Bug: Fixed a bug that prohibited groups from having different contents than the parent form. Previously, the groups contents were not being properly updated. Added new documentation on how to use groups to generate object-based sub-forms. Thanks to Paul Carduner for providing the fix and documentation.

6.1.60 1.8.0 (2008-01-23)

- Feature: Implemented `IDisplayForm` interface.
- Feature: Added integration tests for form interfaces. Added default class attribute called `widgets` in form class with default value `None`. This helps to pass the integration tests. Now, the `widgets` attribute can also be used as a indicator for updated forms.
- Feature: Implemented additional `createAndAdd` hook in `AddForm`. This allows you to implement create and add in a single method. It also supports graceful abortion of a create and add process if we do not return the new object. This means it can also be used as a hook for custom error messages for errors happen during create and add.
- Feature: Add a hidden widget template for the `ISelectWidget`.
- Feature: Arrows in the ordered select widget replaced by named entities.
- Feature: Added `CollectionSequenceDataConverter` to `setupFormDefaults`.
- Feature: Templates for the CheckBox widget are now registered in `checkbox.zcml`.
- Feature: If a value cannot be converted from its unicode representation to a field value using the field's `IFromUnicode` interface, the resulting type error now shows the field name, if available.
- Bug: `createId` could not handle arbitrary unicode input. Thanks to Andreas Reuleaux for reporting the bug and a patch for it. (Added descriptive doctests for the function in the process.)
- Bug: Interface invariants where not working when not all fields needed for computing the invariant are in the submitted form.
- Bug: Ordered select didn't submit selected values.
- Bug: Ordered select lists displayed tokens instead of value,
- Bug: `SequenceWidget` displayed tokens instead of value.

6.1.61 1.7.0 (2007-10-09)

- Feature: Implemented `ImageButton`, `ImageAction`, `ImageWidget`, and `ImageFieldWidget` to support imge submit buttons.
- Feature: The `AttributeField` data manager now supports adapting the content to the fields interface when the content doesn't implement this interface.
- Feature: Implemented single checkbox widget that can be used for boolean fields. They are not available by default but can be set using the `widgetFactory` attribute.
- Bug: More lingual issues have been fixed in the documentation. Thanks to Martijn Faassen for doing this.
- Bug: When an error occurred during processing of the request the widget ended up being security proxied and the system started throwing `TraversalError`-'s trying to access the `label` attribute of the widget. Declared that the widgets require the `zope.Public` permission in order to access these attributes.
- Bug: When rendering a widget the `style` attribute was not honored. Thanks to Andreas Reuleaux for reporting.
- Bug: When an error occurred in the sub-form, the status message was not set correctly. Fixed the code and the incorrect test. Thanks to Markus Kemmerling for reporting.
- Bug: Several interfaces had the `self` argument in the method signature. Thanks to Markus Kemmerling for reporting.

6.1.62 1.6.0 (2007-08-24)

- Feature: An event handler for `ActionErrorOccurred` events is registered to merge the action error into the form's error collectors, such as `form.widgets.errors` and `form.widgets['name'].error` (if applicable). It also sets the status of the form. (Thanks to Herman Himmelbauer, who requested the feature, for providing use cases.)
- Feature: Action can now raise `ActionExecutionError` exceptions that will be handled by the framework. These errors wrap the original error. If an error is specific to a widget, then the widget name is passed to a special `WidgetActionExecutionError` error. (Thanks to Herman Himmelbauer, who requested the feature, for providing use cases.)
- Feature: After an action handler has been executed, an action executed event is sent to the system. If the execution was successful, the event is `ActionSuccessfull` event is sent. If an action execution error was raised, the `ActionErrorOccurred` event is raised. (Thanks to Herman Himmelbauer, who requested the feature, for providing use cases.)
- Feature: The `applyChanges()` function now returns a dictionary of changes (grouped by interface) instead of a boolean. This allows us to generate a more detailed object-modified event. If no changes are applied, an empty dictionary is returned. The new behavior is compatible with the old one, so no changes to your code are required. (Thanks to Darryl Cousins for the request and implementation.)
- Feature: A new `InvalidErrorViewSnippet` class provides an error view snippet for `zope.interface.Invalid` exceptions, which are frequently used for invariants.
- Feature: When a widget is required, HTML-based widgets now declare a "required" class.
- Feature: The validation data wrapper now knows about the context of the validation, which provides a hook for invariants to access the environment.
- Feature: The BoolTerms term tokens are now cosntants and stay the same, even if the label has changed. The choice for the token is "true" and "false". By default it used to be "yes" and "no", so you probably have to change some unit tests. Functional tests are still okay, because you select by term title.
- Feature: BoolTerms now expose the labels for the true and false values to the class. This makes it a matter of doing trivial sub-classing to change the labels for boolean terms.

- Feature: Exposed several attributes of the widget manager to the form for convenience. The attributes are: mode, ignoreContext, ignoreRequest, ignoreReadonly.
- Feature: Provide more user-friendly error messages for number formatting.
- Refactoring: The widget specific class name was in camel-case. A converntion that later developed uses always dash-based naming of HTML/CSS related variables. So for example, the class name "textWidget" is now "text-widget". This change will most likely require some changes to your CSS declarations!
- Documentation: The text of `field.txt` has been reviewed linguistically.
- Documentation: While reviewing the `form.txt` with some people, several unclear and incomplete statements were discovered and fixed.
- Bug (IE): In Internet Explorer, when a label for a radio input field is only placed around the text describing the choice, then only the text is surrounded by a dashed box. IE users reported this to be confusing, thus we now place the label around the text and the input element so that both are surrounded by the dashed border. In Firefox and KHTML (Safari) only the radio button is surrounded all the time.
- Bug: When extracting and validating data in the widget manager, invariant errors were not converted to error view snippets.
- Bug: When error view snippets were not widget-specific – in other words, the `widget` attribute was `None` – rendering the template would fail.

6.1.63 1.5.0 (2007-07-18)

- Feature: Added a span around values for widgets in display mode. This allows for easier identification widget values in display mode.
- Feature: Added the concept of widget events and implemented a particular "after widget update" event that is called right after a widget is updated.
- Feature: Restructured the approach to customize button actions, by requiring the adapter to provide a new interface `IButtonAction`. Also, an adapter is now provided by default, still allowing cusotmization using the usual methods though.
- Feature: Added button widget. While it is not very useful without Javascript, it still belongs into this package for completion.
- Feature: All `IFieldWidget` instances that are also HTML element widgets now declare an additional CSS class of the form "<fieldtype.lower()>-field".
- Feature: Added `addClass()` method to HTML element widgets, so that adding a new CSS class is simpler.
- Feature: Renamed "css" attribute of the widget to "klass", because the class of an HTML element is a classification, not a CSS marker.
- Feature: Reviewed all widget attributes. Added all available HTML attributes to the widgets.
- Documentation: Removed mentioning of widget's "hint" attribute, since it does not exist.
- Optimization: The terms for a sequence widget were looked up multiple times among different components. The widget is now the canonical source for the terms and other components, such as the converter uses them. This avoids looking up the terms multiple times, which can be an expensive process for some applications.
- Bug/Feature: Correctly create labels for radio button choices.
- Bug: Buttons did not honor the name given by the schema, if created within one, because we were too anxious to give buttons a name. Now name assignment is delayed until the button is added to the button manager.
- Bug: Button actions were never updated in the actions manager.

- Bug: Added tests for textarea widget.

6.1.64 1.4.0 (2007-06-29)

- Feature: The select widget grew a new `prompt` flag, which allows you to explicitely request a selection prompt as the first option in the selection (even for required fields). When set, the prompt message is shown. Such a prompt as option is common in Web-UIs.
- Feature: Allow "no value message" of select widgets to be dynamically changed using an attribute value adapter.
- Feature: Internationalized data conversion for date, time, date/time, integer, float and decimal. Now the locale data is used to format and parse those data types to provide the bridge to text-based widgets. While those features require the latest zope.i18n package, backward compatibility is provided.
- Feature: All forms now have an optional label that can be used by the UI.
- Feature: Implemented groups within forms. Groups allow you to combine a set of fields/widgets into a logical unit. They were designed with ease of use in mind.
- Feature: Button Actions – in other words, the widget for the button field – can now be specified either as the "actionFactory" on the button field or as an adapter.
- Bug: Recorded all public select-widget attributes in the interface.

6.1.65 1.3.0 (2007-06-22)

- Feature: In an edit form applying the data and generating all necessary messages was all done within the "Apply" button handler. Now the actual task of storing is factored out into a new method called "applyChanges(data)", which returns whether the data has been changed. This is useful for forms not dealing with objects.
- Feature: Added support for `hidden` fields. You can now use the `hidden` mode for widgets which should get rendered as `<input type="hidden" />`.

  Note: Make sure you use the new formui templates which will avoid rendering
  labels for hidden widgets or adjust your custom form macros.
- Feature: Added `missing_value` support to data/time converters
- Feature: Added named vocabulary lookup in `ChoiceTerms` and `CollectionTerms`.
- Feature: Implemented support for `FileUpload` in `FileWidget`.
  - Added helper for handling `FileUpload` widgets:
    * `extractContentType(form, id)`

      Extracts the content type if `IBytes/IFileWidget` was used.
    * `extractFileName(form, id, cleanup=True, allowEmtpyPostFix=False)`

      Extracts a filename if `IBytes/IFileWidget` was used.

      Uploads from win/IE need some cleanup because the filename includes also the path. The option `cleanup=True` will do this for you. The option `allowEmtpyPostFix` allows you to pass a filename without extensions. By default this option is set to `False` and will raise a `ValueError` if a filename doesn't contain an extension.
  - Created afile upload data converter registered for `IBytes/IFileWidget` ensuring that the converter will only be used for fiel widgets. The file widget is now the default for the bytes field. If you need to use a text area widget for `IBytes`, you have to register a custom widget in the form using:

```
fields['foobar'].widgetFactory = TextWidget
```

- Feature: Originally, when an attribute access failed in Unauthorized or ForbiddenAttribute exceptions, they were ignored as if the attribute would have no value. Now those errors are propagated and the system will fail providing the developer with more feedback. The datamanager also grew a new `query()` method that returns always a default and the `get()` method propagates any exceptions.
- Feature: When writing to a field is forbidden due to insufficient priviledges, the resulting widget mode will be set to "display". This behavior can be overridden by explicitely specifying the mode on a field.
- Feature: Added an add form implementation against `IAdding`. While this is not an encouraged method of adding components, many people still use this API to extend the ZMI.
- Feature: The `IFields` class' `select()` and `omit()` method now support two ketword arguments "prefix" and "interface" that allow the selection and omission of prefixed fields and still specify the short name. Thanks to Nikolay Kim for the idea.
- Feature: HTML element ids containing dots are not very good, because then the "element#id" CSS selector does not work and at least in Firefox the attribute selector ("element[attr=value]") does not work for the id either. Converted the codebase to use dashes in ids instead.
- Bug/Feature: The `IWidgets` component is now an adapter of the form content and not the form context. This guarantees that vocabulary factories receive a context that is actually useful.
- Bug: The readonly flag within a field was never honored. When a field is readonly, it is displayed in "display" mode now. This can be overridden by the widget manager's "ignoreReadonly" flag, which is necessary for add forms.
- Bug: The mode selection made during the field layout creation was not honored and the widget manager always overrode the options providing its value. Now the mode specified in the field is more important than the one from the widget manager.
- Bug: It sometimes happens that the sequence widget has the no-value token as one element. This caused `displayValue()` to fail, since it tried to find a term for it. For now we simply ignore the no-value token.
- Bug: Fixed the converter when the incoming value is an empty string. An empty string really means that we have no value and it is thus missing, returning the missing value.
- Bug: Fix a slightly incorrect implementation. It did not cause any harm in real-world forms, but made unit testing much harder, since an API expectation was not met correctly.
- Bug: When required selections where not selected in radio and checkbox widgets, then the conversion did not behave correctly. This also revealed some issues with the converter code that have been fixed now.
- Bug: When fields only had a vocabulary name, the choice terms adaptation would fail, since the field was not bound. This has now been corrected.
- Documentation: Integrated English language and content review improvements by Roy Mathew in `form.txt`.

6.1.66 1.2.0 (2007-05-30)

- Feature: Added ability to change the button action title using an `IValue` adapter.

6.1.67 1.1.0 (2007-05-30)

- Feature: Added compatibility for Zope 3.3 and thus Zope 2.10.

6.1.68 1.0.0 (2007-05-24)

- Initial Release

6.2 Authors

6.2.1 Initial developers

- Stephan Richter (stephan.richter <at> gmail.com)
- Roger Ineichen (roger <at> projekt01.ch)

6.2.2 Contributors

- Adam Groszer
- Axel Muller
- Brian Sutherland
- Carsten Senger
- Christian Zagrodnick
- Christopher Combelles
- Dan Korostelev
- Daniel Nouri
- Darryl Cousins
- David Glick
- Herman Himmelbauer
- Jacob Holm
- Laurent Mignon
- Malthe Borch
- Marius Gedminas
- Martijn Faassen
- Martin Aspeli
- Michael Howitz
- Michael Kerrin
- Paul Carduner
- Dylan Jay
- Chris Calloway

6.3 To-dos and help wanted

6.3.1 Python 3

- Activate parts as packages get ported.
- Support new schema fields, `NativeString` and `NativeStringLine`.

6.3.2 Framework

- There is only hidden widget templates registered for `ITextWidget` and `ISelectWidget`. We have to define more hidden widgets.

6.3.3 Widgets

- The MultiWidget should render a minimum number of elements by default if the field defines min_length. Also, if user tries to remove elements so the number of widgets becomes less than minimum - the widget should add widgets to make their number minimum again. (Did you understand this?:))
- The MultiWidget should provide a way to reorder elements if used for the ISequence field.
- The values defined in "browser" widgets, like `klass` or `onclick` should be overridable by value adapters.
- File upload widget and converter should have a "clear current" option available to clear current value if there's any and the field is not required.

6.3.4 Documentation

- Proofread documentation
- Extend API documentation

6.3.5 Samples

Write some samples to show the power of the new widget and form framework.

- Object name as an additional field in an add form

6.3.6 Improvements

Add explicit difference between error and confirmation message e.g. "There were some errors." and "No changes were applied."

Think about making the framework use NOT_CHANGED on every widget that does not change it's value.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

Z

INDEX

M

Z