



Grok 1.0 Web Development

Carlos de la Guardia



Chapter No. 5 "Forms"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5 "Forms"

A synopsis of the book's content

Information on where to buy this book

About the Author

Carlos de la Guardia has been doing web consulting and development since 1994, when selling any kind of project required two meetings just to explain what the Internet was, in the first place. He was the co-founder of Aldea Systems—a web consulting company—where he spent ten years working on all kinds of web projects, using a diverse range of languages and tools. In 2005, he became an independent developer and consultant, specializing in Zope and Plone projects. He frequently blogs about Plone and other Zope-related subjects.

For More Information: www.PacktPub.com/grok-1-0-web-development/book

Grok 1.0 Web Development

There are many options these days for web development frameworks. Grok is one of the many that are written in the Python programming language, but it may be one of the least known. This book is a step towards getting the word out that Grok can be a very good fit for many kinds of web development projects.

For one thing, Grok is based on a body of software known as the Zope Toolkit (ZTK), which is a huge set of libraries that are used for web development in Python. The ZTK is itself the result of more than ten years of work that started with the Zope web framework, which was one of the very first Python web frameworks.

Grok is also a modern web framework, which means, it has also learned from the innovations of popular web frameworks such as Django or Ruby on Rails. All in all, as we go through the book, you will find Grok as a framework that is agile enough for small applications, yet powerful enough for really complex projects.

What This Book Covers

Chapter 1, *Getting to Know Grok*, goes into what makes Grok an attractive option for Python web development. You'll learn how Grok makes use of the Zope Toolkit and why this makes Grok powerful and flexible. Then some of Grok's most important concepts will be introduced. Finally, you'll briefly see how Grok compares to other web development frameworks.

Chapter 2, *Getting Started with Grok*, shows you how to install Python on different platforms, in case you are new to it. You'll learn what the Python Package Index (PyPI) is and how to work with EasyInstall to quickly install packages from it over the network. Next, you'll create and run your first project using a tool called grokproject.

Chapter 3, *Views*, explains what views are and where in the Grok application code should they be defined. For templating, Grok uses the ZPT templating language, so you'll learn how to use it and see examples of the most common statements in action. To test this knowledge, we'll see how to write a full Grok application using only views. Other topics covered include how to get form parameters from a web request, how to add static resources to a Grok application, and how to create and work with additional views.

Chapter 4, *Models*, introduces the concept of models and what relationship they have with views. Among other key topics, you'll learn how to persist model data on the ZODB and how to structure your code to maintain the separation of display logic from application logic. Next, we'll see what a container is and how to use one. Finally, we'll explain how to use multiple models and associate specific views to each.

For More Information: www.PacktPub.com/grok-1-0-web-development/book

Chapter 5, *Forms*, will start with a quick demonstration of automatic forms. We'll briefly touch the concepts of interface and schema and show how they are used to generate forms automatically. Among other things, you'll learn how to filter fields and prevent them from appearing in a form, and how to change form templates and presentation.

Chapter 6, *The Catalog: An Object-Oriented Search Engine*, will discuss how to search for specific objects in the database using a tool called the catalog. We'll cover what a catalog is and how it works, what indexes are and how they work, and how to store data in the catalog. Next, we'll learn how to perform simple queries on the catalog and how to use that knowledge to create a search interface for our application.

Chapter 7, *Security*, will cover authentication and authorization. Grok security is based on the concepts of principals (users), permissions, and roles. It has a default security policy, which you'll learn to modify. Grok has a pluggable authentication system, which allows us to set up custom security policies. We'll create one and learn how to manage users as well.

Chapter 8, *Application Presentation and Page Layout*, deals with Grok's layout and presentation machinery that is based on the concept of viewlets. We'll learn what viewlet managers and viewlets are and how to define a layout using them. Then we'll cover the concepts of layers and skins, which allow Grok applications to be delivered with alternative presentations and styles. Finally, we'll define an alternative skin for the application.

Chapter 9, *Grok and the ZODB*, tells us more about the ZODB, including how to take advantage of its other features, such as blob handling. We'll also learn a bit about ZODB maintenance and the need to pack the database frequently. Finally, we'll try our hand at using the ZODB as a regular Python library, outside Grok.

Chapter 10, *Grok and Relational Databases*, will help us find out what facilities Grok has for relational database access. Here are some specific things that we will cover—why is it important that Grok allows developers to use relational databases easily, what an Object Relational Mapper is, how to use SQLAlchemy with Grok, and how to change our authentication mechanism to use a relational database instead of the ZODB.

For More Information: www.PacktPub.com/grok-1-0-web-development/book

Chapter 11, *Key Concepts Behind Grok*, goes into a little more depth with the main concepts behind the Zope Component Architecture, a pillar of Grok. We'll start by explaining the main ZCA concepts, such as interfaces, adapters, and utilities, using the code from the last ten chapters for illustration. We'll learn about one of the main benefits of the ZCA, by using some of its patterns to extend our application. Most importantly, we'll cover how to extend a package without touching its code.

Chapter 12, *Grokkers, Martian, and Agile Configuration*, will introduce grokkers. A grokker is a piece of code that allows developers to use framework functionality by making declarations in the code instead of using configuration files. Grok uses a library called Martian to create its own grokkers and we'll see how to create our own as well.

Chapter 13, *Testing and Debugging*, comments briefly on the importance of testing and then explains how to do testing in Grok. We'll start with extending the functional test suite provided by grokproject and then go on to other kinds of testing. Finally, we'll cover some debugging tools, including live web debuggers.

Chapter 14, *Deployment*, discusses how to deploy our application by using the standard paster server. Then, we'll find out how to run the application behind Apache, first by using a simple proxy configuration, and then under mod_wsgi. Finally, we'll explore how ZEO provides horizontal scalability for our application, and will briefly discuss how to make a site support high traffic loads by adding caching and load balancing into the mix.

For More Information: www.PacktPub.com/grok-1-0-web-development/book

5

Forms

We have seen how easy it is to create a small application, such as the to-do list manager that we have developed over the past couple of chapters. We will now take a look at one of the many ways that Grok can help us develop more complex applications.

Until now, we have been working with simple one-or two-field forms. When we changed our model in the previous chapter, we had to go back and edit the HTML for the forms as well. With a couple of fields this requires little work, but when we have complex models with perhaps a dozen fields or more, it would be great if we didn't have to modify two files whenever we make a change.

Fortunately, Grok has a mechanism for automating the creation and processing of forms. We'll see how it works in this chapter, along with a few other form-related subjects:

- What is an interface
- What is a schema
- How interfaces and schemas are used to generate forms automatically, using Grok's form components
- How to create, add, and edit forms
- How to filter fields and prevent them from appearing in a form
- How to change form templates and presentation

For More Information: www.PacktPub.com/grok-1-0-web-development/book

A quick demonstration of automatic forms

Let's start by showing how this works, before getting into the details. To do that, we'll add a project model to our application. A project can have any number of lists associated with it, so that related to-do lists can be grouped together. For now, let's consider the project model by itself. Add the following lines to the `app.py` file, just after the `Todo` application class definition. We'll worry later about how this fits into the application as a whole.

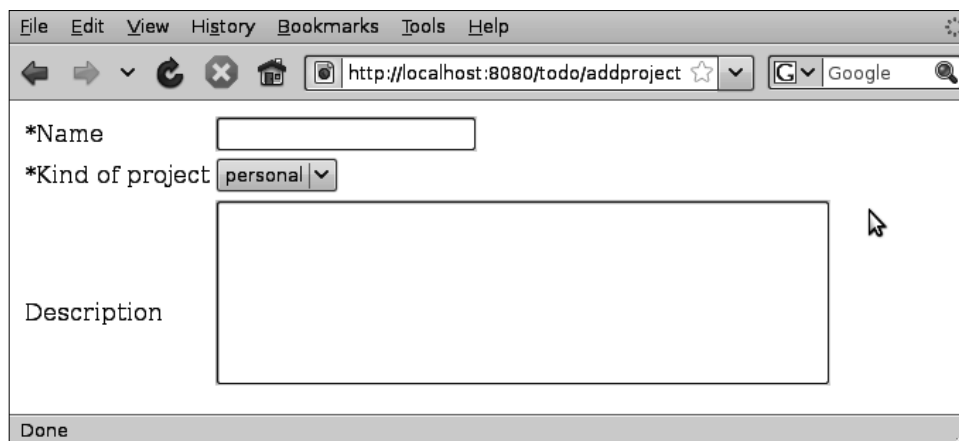
```
class IProject(interface.Interface):
    name = schema.TextLine(title=u'Name', required=True)
    kind = schema.Choice(title=u'Kind of project',
                        values=['personal', 'business'])
    description = schema.Text(title=u'Description')

class AddProject(grok.Form):
    grok.context(Todo)
    form_fields = grok.AutoFields(IProject)
```

We'll also need to add a couple of imports at the top of the file:

```
from zope import interface
from zope import schema
```

Save the file, restart the server, and go to the URL `http://localhost:8080/todo/addproject`. The result should be similar to the following screenshot:



The screenshot shows a web browser window with the address bar containing `http://localhost:8080/todo/addproject`. The page displays a form with the following elements:

- A label `*Name` followed by a text input field.
- A label `*Kind of project` followed by a dropdown menu showing `personal`.
- A label `Description` followed by a large text area.

The browser's status bar at the bottom shows the word `Done`.

OK, where did the HTML for the form come from? We know that `AddProject` is some sort of a view, because we used the `grok.context` class annotation to set its context and name. Also, the name of the class, but in lowercase, was used in the URL, like in previous view examples.

The important new thing is how the form fields were created and used. First, a class named `IProject` was defined. The interface defines the fields on the form, and the `grok.AutoFields` method assigns them to the `Form` view class. That's how the view knows which HTML form controls to generate when the form is rendered.

We have three fields: `name`, `description`, and `kind`. Later in the code, the `grok.AutoFields` line takes this `IProject` class and turns these fields into form fields.

That's it. There's no need for a template or a render method. The `grok.Form` view takes care of generating the HTML required to present the form, taking the information from the value of the `form_fields` attribute that the `grok.AutoFields` call generated.

Interfaces

The `I` in the class name stands for Interface. We imported the `zope.interface` package at the top of the file, and the `Interface` class that we have used as a base class for `IProject` comes from this package.

Example of an interface

An **interface** is an object that is used to specify and describe the external behavior of objects. In a sense, the interface is like a contract. A class is said to implement an interface when it includes all of the methods and attributes defined in an interface class. Let's see a simple example:

```
from zope import interface
class ICaveman(interface.Interface):
    weapon = interface.Attribute('weapon')

    def hunt(animal):
        """Hunt an animal to get food"""
    def eat(animal):
        """Eat hunted animal"""
    def sleep():
        """Rest before getting up to hunt again"""
```

Here, we are describing how cavemen behave. A caveman will have a weapon, and he can hunt, eat, and sleep. Notice that the `weapon` is an *attribute* – something that belongs to the object, whereas `hunt`, `eat`, and `sleep` are *methods*.

Once the interface is defined, we can create classes that implement it. These classes are committed to include all of the attributes and methods of their interface class. Thus, if we say:

```
class Caveman(object):
    interface.implements(ICaveman)
```

Then we are promising that the `Caveman` class will implement the methods and attributes described in the `ICaveman` interface:

```
weapon = 'ax'
def hunt(animal):
    find(animal)
    hit(animal, self.weapon)
def eat(animal):
    cut(animal)
    bite()
def sleep():
    snore()
    rest()
```

Note that though our example class implements all of the interface methods, there is no enforcement of any kind made by the Python interpreter. We could define a class that does not include any of the methods or attributes defined, and it would still work.

Interfaces in Grok

In Grok, a model can implement an interface by using the `grok.implements` method. For example, if we decided to add a project model, it could implement the `IProject` interface as follows:

```
class Project(grok.Container):
    grok.implements(IProject)
```

Due to their descriptive nature, interfaces can be used for documentation. They can also be used for enabling component architectures, but we'll see about that later on. What is of more interest to us right now is that they can be used for generating forms automatically.

Schemas

The way to define the form fields is to use the `zope.schema` package. This package includes many kinds of field definitions that can be used to populate a form.

Basically, a schema permits detailed descriptions of class attributes that are using fields. In terms of a form – which is what is of interest to us here – a schema represents the data that will be passed to the server when the user submits the form. Each field in the form corresponds to a field in the schema.

Let's take a closer look at the schema we defined in the last section:

```
class IProject(interface.Interface):
    name = schema.TextLine(title=u'Name', required=True)
    kind = schema.Choice(title=u'Kind of project',
                        required=False,
                        values=['personal', 'business'])
    description = schema.Text(title=u'Description',
                             required=False)
```

The schema that we are defining for `IProject` has three fields. There are several kinds of fields, which are listed in the following table. In our example, we have defined a `name` field, which will be a required field, and will have the label `Name` beside it. We also have a `kind` field, which is a list of options from which the user must pick one. Note that the default value for `required` is `True`, but it's usually best to specify it explicitly, to avoid confusion. You can see how the list of possible values is passed statically by using the `values` parameter. Finally, `description` is a text field, which means it will have multiple lines of text.

Available schema attributes and field types

In addition to `title`, `values`, and `required`, each schema field can have a number of properties, as detailed in the following table:

Attribute	Description
<code>title</code>	A short summary or label.
<code>description</code>	A description of the field.
<code>required</code>	Indicates whether a field requires a value to exist.
<code>readonly</code>	If <code>True</code> , the field's value cannot be changed.
<code>default</code>	The field's default value may be <code>None</code> , or a valid field value.
<code>missing_value</code>	If input for this field is missing, and that's OK, then this is the value to use.
<code>order</code>	The <code>order</code> attribute can be used to determine the order in which fields in a schema are defined. If one field is created after another (in the same thread), its order will be greater.

In addition to the field attributes described in the preceding table, some field types provide additional attributes. In the previous example, we saw that there are various field types, such as `Text`, `TextLine`, and `Choice`. There are several other field types available, as shown in the following table. We can create very sophisticated forms just by defining a schema in this way, and letting Grok generate them.

Field type	Description	Parameters
<code>Bool</code>	Boolean field.	
<code>Bytes</code>	Field containing a byte string (such as the python <code>str</code>). The value might be constrained to be within length limits.	
<code>ASCII</code>	Field containing a 7-bit ASCII string. No characters <code>> DEL (chr(127))</code> are allowed. The value might be constrained to be within length limits.	
<code>BytesLine</code>	Field containing a byte string without new lines.	
<code>ASCIILine</code>	Field containing a 7-bit ASCII string without new lines.	
<code>Text</code>	Field containing a Unicode string.	
<code>SourceText</code>	Field for the source text of an object.	
<code>TextLine</code>	Field containing a Unicode string without new lines.	
<code>Password</code>	Field containing a Unicode string without new lines, which is set as the password.	
<code>Int</code>	Field containing an Integer value.	
<code>Float</code>	Field containing a Float.	
<code>Decimal</code>	Field containing a Decimal.	
<code>DateTime</code>	Field containing a DateTime.	
<code>Date</code>	Field containing a date.	
<code>Timedelta</code>	Field containing a timedelta.	
<code>Time</code>	Field containing time.	
<code>URI</code>	A field containing an absolute URI.	
<code>Id</code>	A field containing a unique identifier. A unique identifier is either an absolute URI or a dotted name. If it's a dotted name, it should have a module or package name as a prefix.	

Field type	Description	Parameters
Choice	Field whose value is contained in a predefined set.	<p>values: A list of text choices for the field.</p> <p>vocabulary: A Vocabulary object that will dynamically produce the choices.</p> <p>source: A different, newer way to produce dynamic choices.</p> <p>Note: only one of the three should be provided. More information about sources and vocabularies is provided later in this book.</p>
Tuple	Field containing a value that implements the API of a conventional Python tuple.	<p>value_type: Field value items must conform to the given type, expressed via a field.</p> <p>Unique. Specifies whether the members of the collection must be unique.</p>
List	Field containing a value that implements the API of a conventional Python list.	<p>value_type: Field value items must conform to the given type, expressed via a field.</p> <p>Unique. Specifies whether the members of the collection must be unique.</p>
Set	Field containing a value that implements the API of a conventional Python standard library <code>sets.Set</code> or a Python 2.4+ set.	<p>value_type: Field value items must conform to the given type, expressed via a field.</p>
Frozenset	Field containing a value that implements the API of a conventional Python 2.4+ frozenset.	<p>value_type: Field value items must conform to the given type, expressed via a field.</p>
Object	Field containing an object value.	<p>Schema: The interface that defines the fields comprising the object.</p>
Dict	Field containing a conventional dictionary. The <code>key_type</code> and <code>value_type</code> fields allow specification of restrictions for keys and values contained in the dictionary.	<p>key_type: Field keys must conform to the given type, expressed via a field.</p> <p>value_type: Field value items must conform to the given type, expressed via a field.</p>

Form fields and widgets

Schema fields are perfect for defining data structures, but when dealing with forms sometimes they are not enough. In fact, once you generate a form using a schema as a base, Grok turns the schema fields into form fields. A **form field** is like a schema field but has an extended set of methods and attributes. It also has a default associated widget that is responsible for the appearance of the field inside the form.

Rendering forms requires more than the fields and their types. A form field needs to have a user interface, and that is what a widget provides. A `Choice` field, for example, could be rendered as a `<select>` box on the form, but it could also use a collection of checkboxes, or perhaps radio buttons. Sometimes, a field may not need to be displayed on a form, or a writable field may need to be displayed as text instead of allowing users to set the field's value.

Form components

Grok offers four different components that automatically generate forms. We have already worked with the first one of these, `grok.Form`. The other three are specializations of this one:

- `grok.AddForm` is used to add new model instances.
- `grok.EditForm` is used for editing an already existing instance.
- `grok.DisplayForm` simply displays the values of the fields.

A Grok form is itself a specialization of a `grok.View`, which means that it gets the same methods as those that are available to a view. It also means that a model does not actually need a view assignment if it already has a form. In fact, simple applications can get away by using a form as a view for their objects. Of course, there are times when a more complex view template is needed, or even when fields from multiple forms need to be shown in the same view. Grok can handle these cases as well, which we will see later on.

Adding a project container at the root of the site

To get to know Grok's form components, let's properly integrate our project model into our to-do list application. We'll have to restructure the code a little bit, as currently the to-do list container is the root object of the application. We need to have a project container as the root object, and then add a to-do list container to it.

Luckily, we already structured the code properly in the last chapter, so we won't need to make many changes now. To begin, let's modify the top of `app.py`, immediately before the `ToDoList` class definition, to look like this:

```
import grok
from zope import interface, schema

class ToDo(grok.Application, grok.Container):
    def __init__(self):
        super(ToDo, self).__init__()
        self.title = 'To-Do list manager'
        self.next_id = 0

    def deleteProject(self, project):
        del self[project]
```

First, we import `zope.interface` and `zope.schema`. Notice how we keep the `ToDo` class as the root application class, but now it can contain projects instead of lists. We also omitted the `addProject` method, because the `grok.AddForm` instance is going to take care of that. Other than that, the `ToDo` class is almost the same.

```
class IProject(interface.Interface):
    title = schema.TextLine(title=u'Title', required=True)
    kind = schema.Choice(title=u'Kind of project', values=['personal',
        'business'])
    description = schema.Text(title=u'Description', required=False)
    next_id = schema.Int(title=u'Next id', default=0)
```

We then have the interface definition for `IProject`, where we add the `title`, `kind`, `description`, and `next_id` fields. These were the fields that we previously added during the call to the `__init__` method at the time of product initialization.

```
class Project(grok.Container):
    grok.implements(IProject)

    def addList(self, title, description):
        id = str(self.next_id)
        self.next_id = self.next_id+1
        self[id] = ToDoList(title, description)

    def deleteList(self, list):
        del self[list]
```

The key thing to notice in the `Project` class definition is that we use the `grok.implements` class declaration to see that this class will implement the schema that we have just defined.

```
class AddProjectForm(grok.AddForm):
    grok.context(Todo)
    grok.name('index')
    form_fields = grok.AutoFields(Project)
    label = "To begin, add a new project"

    @grok.action('Add project')
    def add(self, **data):
        project = Project()
        self.applyData(project, **data)
        id = str(self.context.next_id)
        self.context.next_id = self.context.next_id+1
        self.context[id] = project
        return self.redirect(self.url(self.context[id]))
```

The actual form view is defined after that, by using `grok.AddForm` as a base class. We assign this view to the main `Todo` container by using the `grok.context` annotation. The name `index` is used for now, so that the default page for the application will be the 'add form' itself.

Next, we create the form fields by calling the `grok.AutoFields` method. Notice that this time the argument to this method call is the `Project` class directly, rather than the interface. This is possible because the `Project` class was associated with the correct interface when we previously used `grok.implements`.

After we have assigned the fields, we set the `label` attribute of the form to the text: **To begin, add a new project**. This is the title that will be shown on the form.

In addition to this new code, all occurrences of `grok.context(Todo)` in the rest of the file need to be changed to `grok.context(Project)`, as the to-do lists and their views will now belong to a project and not to the main `Todo` application. For details, take a look at the source code of this book for Chapter 5.

Form actions

If you carefully look at the screenshot shown in the *A quick demonstration of automatic forms* section, you will see that the form has no submit buttons. In Grok, every form can have one or more actions, and for each action the form will have a submit button. The Grok `action` decorator is used to mark the methods of the form class that will be used as actions. In this case, the `add` method is decorated with it and the value of the text parameter, in this case `Add project`, will be used as the text on the button. To change the text on the button, simply modify the string passed to the decorator:

```
@grok.action('Add project')
def add(self, **data):
    project = Project()
    self.applyData(project, **data)
    id = str(self.context.next_id)
    self.context.next_id = self.context.next_id+1
    self.context[id] = project
    return self.redirect(self.url(self.context[id]))
```

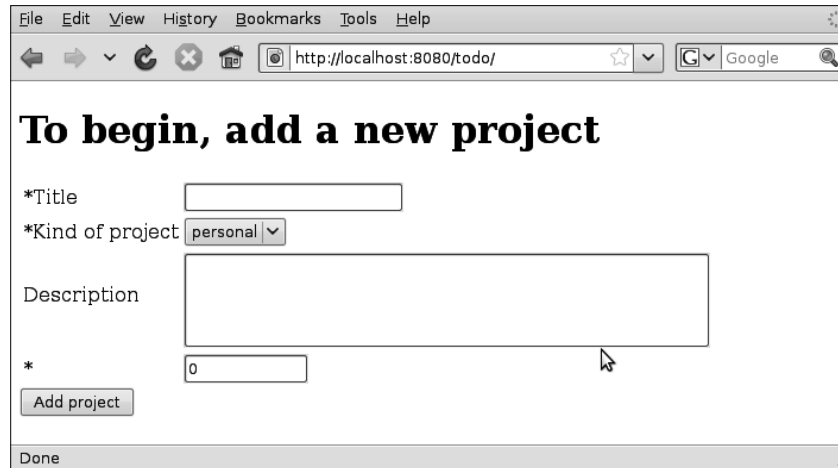
The `add` method receives all of the filled form fields in the `data` parameter, and then creates a `Project` instance. Next, it sets the attributes of `project` to the values in the form by calling the `applyData` method of the form. Finally, it adds the new project to the `Todo` instance and redirects the user to the project page.

Trying out the application

When you try out the application, there are two things to notice. First, when the `add project` form is displayed, the `next_id` field, which is used to name the projects, is shown. We could even edit it if we like. Obviously, we don't want this behavior.

Second, once the project has been created and we get redirected to the project page, everything works as before, even though we didn't touch the templates. With the approach that we tried in Chapter 3, using hidden values and list indexes, we would have had to modify the `index.pt` template in lots of places. Now that we have a structure based on models, the views and methods that were registered for them don't need to change, even though the containment hierarchy is different.

Another important thing to notice is that the add project form has no design or styling at all. This is because the form building mechanism uses a common template, which we haven't styled yet.



Filtering fields

Remember that currently we have the `next_id` field of the project shown on the form. We don't want it there, so how do we remove it? Fortunately for us, the list of fields generated by the `grok.AutoFields` method can easily be filtered.

We can either select precisely the fields we need, using the `select` method:

```
form_fields = grok.AutoFields(Project).select('title', 'kind',
                                             'description')
```

Or we can omit specific fields by using the `omit` method:

```
form_fields = grok.AutoFields(Project).omit('next_id')
```

In both cases, we pass the IDs of the fields as strings to the selected method. Now the `next_id` field is not there anymore, as you can see in the next screenshot.

Filtering fields can be useful not just for removing unwanted fields such as `next_id`. We can also have specialized forms for editing only a part of a schema, or for showing specific fields, depending on user information or input.

Using grok.EditForm

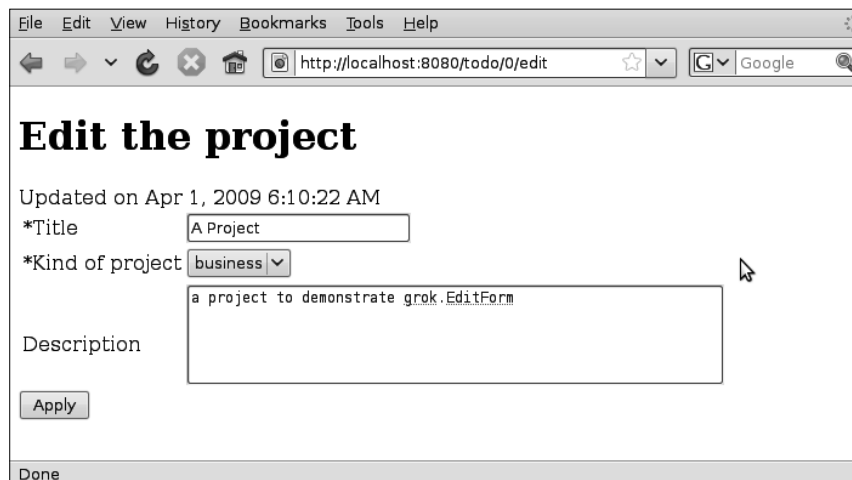
The form that we just made is intended to add a new project to the application, but what if we need to edit an existing project? In this case, we need a form that knows how to get the existing values of all of the fields on the form, and display them when editing. Another difference is that the add project form was assigned to the main `Todo` application, but an edit form would use as a context the actual project that it would be modifying.

That's why Grok has another kind of form component for editing. Using `grok.EditForm` is even easier than `grok.AddForm`. Here's all the code that we need to add to our application in order to be able to edit projects:

```
class EditProjectForm(grok.EditForm):
    grok.context(Project)
    grok.name('edit')
    form_fields = grok.AutoFields(Project).omit('next_id')
    label = "Edit the project"
```

As mentioned earlier, the context for this form is the `Project` class, which we set by using the `grok.context` class annotation. We give this form the name `edit`, so that it will be possible to just append that word to a project URL to get its edit view. As we discussed in the previous section, it is a good idea to eliminate the display of the `next_id` field from the form, so we use the `omit` method to do that. Finally, we set a label for the form and we are then ready to test it.

Start the application. If you haven't created a project already, please do so. Then, go to the URL: `http://localhost:8080/todo/0/edit`. Edit the project fields and then click on the **Apply** button. You should see a screen similar to the one shown in following screenshot:



Notice how we didn't include a redirect after rendering the form, so that when we click on the **Apply** button we go back to the same form, but with a message telling us that the object was updated along with the date of modification. If we wanted, we could add an 'edit' action, by using the `action` decorator and a redirect, just like we did for the add form.

Modifying individual form fields

Having the add and edit forms created automatically by Grok is neat, but there are cases where we will need to make small modifications in how a form is rendered. Grok allows us to modify specific field attributes easily to make that happen.

Remember that each form field will be rendered by a widget, which could be thought of as views for specific fields. These views usually accept a number of parameters to allow the user to customize the appearance of the form in one way or another.

Just before being rendered, Grok's form components always call a method named `setUpWidgets`, which we can override in order to make modifications to the fields and their attributes.

In the add and edit project forms, the title of the project, which is of type `TextLine`, has a widget that displays the `<input>` tag used to capture its value with a length of 20 characters. Many project names could be longer than that, so we want to extend the length to 50 characters. Also, the text area for the description is too long for a project summary, so we'll cut it to five rows instead. Let's use the `setUpWidgets` method for this. Add the following lines to both the `AddProjectForm` and `EditProjectForm` classes:

```
def setUpWidgets(self, ignore_request=False):
    super(EditProjectForm, self).setUpWidgets(ignore_request)
    self.widgets['title'].displayWidth = 50
    self.widgets['description'].height = 5
```

Take care to substitute `EditProjectForm` for `AddProjectForm` on the super call when adding the method to its appropriate class. The `setUpWidgets` method is fairly simple. We first call the super class to make sure that we get the correct properties in the form before trying to modify them. Next, we modify any properties that we want for the field. In this case, we access the `widgets` property to get at the widgets for the fields that we defined, and change the values that we want.

Another thing that requires explanation is the `ignore_request` parameter that is passed to the `setUpWidgets` method. If this is set to `False`, as we have defined it, then this means that any field values present in the `HTTP` request will be applied to the corresponding fields. A value of `True` means that no values should be changed during this call.

Restart the application and you will see that the edit and add forms now present the widgets using the properties that we modified.

Form validation

Now we have working forms for adding and editing projects. In fact, the forms can do more than we have shown so far. For example, if we go to the add form and try to submit it without filling in the required `title` field, we'll get the form back, instead of being redirected. No project will be created, and an error message will be visible on the screen, warning us that the field can't be empty.

This validation happens automatically, but we can also add our own constraints by using the `constraint` parameter, when defining a field in the schema. For example, suppose that we absolutely need to have more than two words in the title. We can do that very easily. Just add the following lines before the interface definition:

```
def check_title(value):
    return len(value.split())>2
```

Next, modify the title field definition to look like this:

```
title = schema.TextLine(title=u'Title', required=True,
    constraint=check_title)
```

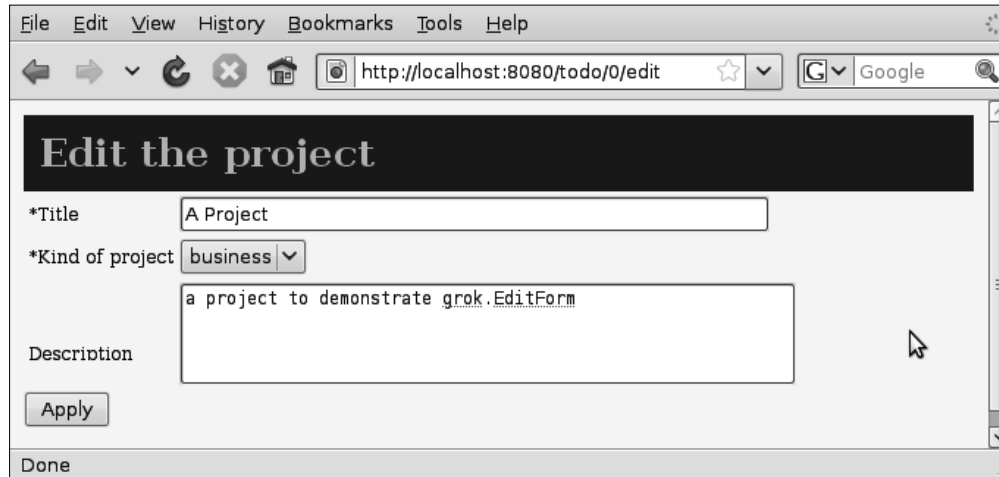
We first defined a function that will receive the field value as a parameter, and must return `True` if the value is valid, or `False` otherwise. We then assign this function to the `constraint` parameter, when defining the field in the interface. This is all very simple, but it allows us to add validations for whatever conditions we need to meet in our form data.

There are cases where a simple constraint is not enough to validate a field. For instance, imagine that what we need is that whenever the kind of the project is 'business', the description can't be empty. In this case, a constraint will not do, as whether or not the description field is valid depends on the value of another field.

A constraint that involves more than one field is known as an **invariant** in Grok. To define one, the `@interface.invariant` decorator is used. For the hypothetical case described earlier, we can use the following definition, which we'll add inside the Interface definition:

```
@interface.invariant
def businessNeedsDescription(project):
    if project.kind=='business' and not project.description:
        raise interface.Invalid(
            "Business projects require a description")
```

Now, when we try to add a project of kind 'business', Grok will complain if the description is empty. Look at the next screenshot for reference:



Customizing the form template

Earlier, we commented on the fact that our new project forms have no styling or design applied, and therefore they look markedly different from the rest of our application. It's time to change that.

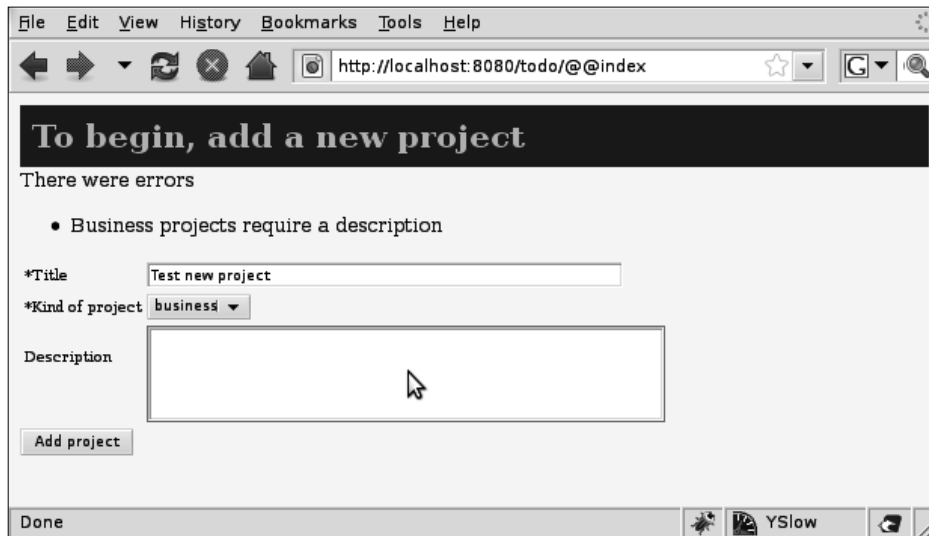
The disadvantage of using automatic form generation is that the default template must be fairly generic to be useful in multiple applications. However, Grok allows us to set a custom template for editing the form. All we need to do is set the `template` attribute in the form:

```
template = grok.PageTemplateFile('custom_edit_form.pt')
```

Of course, for this to work we also have to provide the named template inside our application directory (not inside `app_templates`). For now, let's just add a stylesheet and a class to the generic edit template that comes with Grok. There is nothing special here, so we will just take the default edit form template and add the same stylesheet that we defined previously. Please look at the source code of this book, if you want to see the rest of the template.

```
<html>
<head>
  <title tal:content="context/title">To-Do list manager</title>
  <link rel="stylesheet" type="text/css"
        tal:attributes="href static/styles.css" />
</head>
```

That's all that's needed in order to have our custom template for editing the form. Take a look at the next screenshot to see how it looks. Of course, we would have to further modify it to get it to look just like we want. We could even leave only the fields that we wanted, placed in an irregular arrangement, but the reason that we used the original template and modified it just a little is so that you can look at it and be careful with the sections where the validation messages are shown and the actions are generated. We'll have more to say about this in future chapters.



Summary

We have seen how to automatically generate forms by using schemas, and how it's possible to customize their rendering. In addition, we learned a little bit about some Zope Framework libraries, such as `zope.schema` and `zope.interface`.

Where to buy this book

You can buy Grok 1.0 Web Development from the Packt Publishing website:
<http://www.packtpub.com/grok-1-0-web-development/book>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information: www.PacktPub.com/grok-1-0-web-development/book