## CONTENTS

1 Welcome to the Grok tutorial! 3

2 Getting started with Grok 5
   2.1 Setting up grokproject 5
   2.2 Creating a new Grok project 6
   2.3 Starting up the web server 7
   2.4 An empty Grok project 8

3 Showing pages 11
   3.1 Publishing a simple web page 11
   3.2 A second view 12
   3.3 Making our page dynamic 13
   3.4 Static resources for our web page 14
   3.5 Using view methods 15
   3.6 Generating HTML from Python 17
   3.7 Completely Python-driven views 18
   3.8 Doing some calculation before viewing a page 19
   3.9 Reading URL parameters 19
   3.10 Simple forms 21

4 Models 23
   4.1 A view for a model 23
   4.2 Storing data 25
   4.3 Redirection 26
   4.4 Showing the value in the form 27
   4.5 The rules of persistence 28
   4.6 Explicitly associating a view with a model 31
   4.7 A second model 32
   4.8 Containers 34
Contents

- Grok tutorial
  - Welcome to the Grok tutorial!
  - Getting started with Grok
    * Setting up grokproject
    * Creating a new Grok project
    * Starting up the web server
    * An empty Grok project
  - Showing pages
    * Publishing a simple web page
    * A second view
    * Making our page dynamic
    * Static resources for our web page
    * Using view methods
    * Generating HTML from Python
    * Completely Python-driven views
    * Doing some calculation before viewing a page
    * Reading URL parameters
    * Simple forms
  - Models
    * A view for a model
    * Storing data
    * Redirection
    * Showing the value in the form
    * The rules of persistence
    * Explicitly associating a view with a model
    * A second model
    * Containers
WELCOME TO THE GROK TUTORIAL!

Getting started with Zope Page Templates

You can find introductions and more information about Zope Page Templates (ZPT, sometimes also called TAL) in various places:

http://plone.org/documentation/tutorial/zpt
http://wiki.zope.org/ZPT/FrontPage

Note that some of the information in these introductions may refer to concepts not available in Grok or the Zope Toolkit, in particular variables like `here` or `template`. The basic principles will work with Grok however; try reading `context` or `view` instead.

Grok is a powerful and flexible web application framework for Python developers. In this tutorial we will show you the various things you can do with Grok, and how Grok can help you build your web application. We’ll start out simple, and will slowly go to more complex usage patterns.

All you’re expected to know is the Python programming language and an understanding of basic web programming (HTML, forms, URLs). It also helps if you are familiar with Zope Page Templates, though most of the examples should be fairly obvious if you are already familiar with another templating language.

We recommend that beginners follow the tutorial from top to bottom. The tutorial is designed to explain important concepts in order and slowly builds up from there.

If you are more experienced, or just curious, you may want to skip around instead and read the pieces which interest you most. If something is unclear, you can always backtrack to previous sections.

Grok is based on the Zope Toolkit. You do not need to know about the Zope Toolkit at all to follow this tutorial. Grok builds on existing Zope Toolkit technology but exposes it in a special way to the developer. We believe Grok makes developing with Zope Toolkit technology easier and more fun for beginners and experienced developers alike.
CHAPTER TWO

GETTING STARTED WITH GROK

This chapter will help you get up and running with Grok, using the grokproject tool. We create a new project with grokproject and tell you how to get that project running so you can access it with a web browser.

2.1 Setting up grokproject

Installing easy_install

If you don’t already have easy_install available, you can find the script to set it up on the PEAK EasyInstall page.

You need to download ez_setup.py. Then, you run it like this to install easy_install into your system Python:

$ sudo python2.6 ez_setup.py

This will make easy_install available to you.

Note: Sometimes you have easy_install installed but you need a newer version of the underlying setuptools infrastructure to make Grok work. You can upgrade setuptools with:

$ sudo easy_install -U setuptools

Setting up grok on a Unix-like (Linux, Mac OS X) environment is easy. Most of these instructions should work in a Windows environment as well.

Let’s go through the prerequisites first. You need a computer connected to the internet, as Grok installs itself over the network. You also need Python 2.6 (or Python 2.5) installed.

Because Grok uses a source distribution of the Zope Toolkit libraries, you may need to install your operating system’s Python “dev” package (python-dev on Debian and Ubuntu, python2.6-dev for Python 2.6). You also need a working C compiler (typically gcc) installed, as we compile bits of the Zope Toolkit during setup (build-essential on Debian and Ubuntu).

On Windows such a build environment is not necessary, as grokproject will download and automatically install precompiled libraries for Windows. Finally, you also need easy_install installed so it becomes
easy to install Python packages.
Once you are done with the prerequisites, you can install grokproject itself:

$ easy_install grokproject

If you are on a Unixy environment and you are not working from a virtualenv or custom compiled python installation, you will need to request admin rights using sudo, as this will install new libraries into your system Python.

We’re ready to create our first grok project now!

2.2 Creating a new Grok project

Using paster

For those who know paster: grokproject is just a wrapper around a paster template. So instead of running the grokproject command, you can also run:

$ paster create -t grok Sample

Let’s create a first Grok project. A Grok project is a working environment for a developer of an web application based using Grok. In essence, a directory with a lot of files and subdirectories in it. Let’s create a Grok project called Sample:

$ grokproject Sample

This tells grokproject to create a new subdirectory called Sample and set up the project in there. grokproject will automatically download and install the Zope Toolkit libraries as well as the Grok Toolkit into the project area.

Grok asks you for an initial username and password for the server. We’ll use grok for both in this example:

Enter user (Name of an initial administrator user): grok
Enter passwd (Password for the initial administrator user): grok

Now you have to wait while grokproject downloads and installs the various tools and libraries that are needed in a Grok project. The second time you create a Grok project it will be faster as it can reuse the previously installed libraries. After all that your Grok project is ready to go.
Common problems installing Grok

No Python development environment

Grok includes dependencies that need to be compiled against Python. This happens automatically in the installation process, but a Python development environment does need to be installed, unless you are on Windows, where we supply binary versions of the required libraries.

On Debian and Ubuntu this is the python-dev package (python2.6-dev for Python 2.6). You also need `build-essential`.

2.3 Starting up the web server

You can go into the Sample project directory now and start up the web server for our project:

```
$ cd Sample
$ bin/paster serve parts/etc/deploy.ini
```

This will make Grok available on port 8080. You can log in with username grok and password grok.
Assuming you’ve started up the web server on your development computer, you can go to it here:

`http://localhost:8080`

This first pops up a login dialog (username: grok and password: grok). It will then show a simple Grok admin interface. This user interface allows you to install new Grok applications.

Our sample application (sample.app.Sample) will be available for adding. Let’s try this out. Go to the Grok admin page:

`http://localhost:8080`

and add a Sample application. Give it the name test.

You can now go to the installed application if you click on its link. This will bring you to the following URL:

`http://localhost:8080/test`

You should see a simple web page with the following text on it:

Congratulations!

Your Grok application is up and running. Edit sample/app_templates/index.pt to change this page.

You can shut down the server at any time by hitting CTRL-c. Shut it down now. We will be shutting down and starting up the server often in this tutorial.
Practice restarting the server now, as you’ll end up doing it a lot during this tutorial. It’s just stopping and starting it again: CTRL-c and then bin/paster serve parts/etc/deploy.ini from your Sample project directory.

Alternatively, you can use the –reload flag to start up paster with a monitor that scans your code base (python files only) for changes and automatically restarts the server every time you make a change:

$ bin/paster serve --reload parts/etc/deploy.ini

### 2.4 An empty Grok project

**What about the other directories and files in our project?**

What about the other files and subdirectories in our Sample project directory? Grokproject sets up the project using a system called zc.buildout. The eggs, develop-eggs, bin and parts directories are all set up and maintained by zc.buildout. See its documentation for more information about how to use it. The configuration of the project and its dependency is in buildout.cfg. For the moment however, you can ignore these details however.

Let’s take a closer look at what’s been created in the Sample project directory.

One of the things grokproject created was a setup.py file. This file contains information about your project. This information is used by buildout to download your project’s dependencies and to install it. You can also use the setup.py file to upload your project to the Python Package Index (PyPI).

We have already seen the bin directory. It contains the startup script for the web server (bin/paster) as well as the executable for the buildout system (bin/buildout) which can be used to re-build your project (to update it or to install a new dependency).

The parts’ directory contain configuration and data created and managed by ‘‘buildout, such as the Zope object database (ZODB) storage, and the .ini files to be used with paster.

The actual code of the project will all be inside the src directory. In it is a Python package directory called sample with the app.py file that grokproject said it would create. Let’s look at this file:

```python
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    pass  # see app_templates/index.pt
```

Not very much yet, but enough to make an installable Grok application and display its welcome page. We’ll go into the details of what this means later.

Besides this, there is an empty __init__.py file to make this directory a Python package.
There is also a directory called `app_templates`. It contains a single template called `index.pt`:

```html
<html>
<head>
</head>
<body>
  <h1>Congratulations!</h1>
  <p>Your Grok application is up and running. Edit <code>sample/app_templates/index.pt</code> to change this page.</p>
</body>
</html>
```

This is the template for your project’s welcome page.

There is also a `configure.zcml` file. This file will normally only contain a few lines that load dependencies and then register this application with Grok. This means we can typically completely ignore it, but we’ll show it here once for good measure:

```xml
<configure xmlns="http://namespaces.zope.org/zope"
xmlns:grok="http://namespaces.zope.org/grok">
  <include package="grok" />
  <includeDependencies package="." />
  <grok:grok package="." />
</configure>
```

In non-Grok applications that use the Zope Toolkit (such as something created with Zope 2 or Zope 3), the ZCML file usually plays a much bigger role. It contains directives which registers particular Python objects (typically classes, such as views) with the component architecture that is central to the Zope Toolkit. Grok however automates this registration by putting more information into the Python code directly, so that the ZCML file can remain small.

There is also a `static` directory. This contains static files that can be used in the web application, such as images, css files and javascript files.

Besides these files, there is an `app.txt`, `ftesting.zcml` and `tests.py`. These all have to do with the automatic testing environment and can be ignored for now.
SHOWING PAGES

Showing web pages is what puts the web in “web applications”. Typically HTML templates are used for this, but Grok doesn’t stop at templates. Most web pages in a real-world web application will contain complex presentation logic that is better handled by separate Python code in conjunction with templates. This becomes especially important in more complex interactions with the user, such as form handling. After reading this chapter, you should already be able to write simple web applications with Grok.

3.1 Publishing a simple web page

Let’s publish a simple static web page. Grok is geared towards web applications and therefore is not really meant for publishing a large number of static (pregenerated) web pages. For that you’re better off to use a specialized system such as Apache. Nonetheless, in order to develop any web application we need to know how to put some simple HTML on the web.

As you saw previously, our Sample application has a stock front page, generated by grokproject. Let’s change that.

To do this, go to the app_templates directory in src/sample/. This directory contains the templates used for anything defined in the app module. Grok knows to associate the directory to the module by its name (<module_name>_templates).

In this directory we will edit the index template for our Sample application object. To do this, open the index.pt file in a text editor. The .pt extension indicates that this file is a Zope Page Template (ZPT). We’re just going to put HTML in it now, but this allows us to make page dynamic later on.

Change the index.pt file to contain the following (very simplistic) HTML:

```html
<html>
<body>
<p>Hello world!</p>
</body>
</html>
```

Then reload the page:

```
http://localhost:8080/test
```

You should now see the following text:
Hello world!

Note that you can change templates and see the effects instantly: there is no need to restart the web server to see the effect. This is not true for changes on the Python level, for instance when you add a template. We show an example of this next.

### 3.2 A second view

Our view is named `index`. This in fact means something slightly special: it’s the default view of our application object. We can also access it explicitly by naming the view:

http://localhost:8080/test/index

If you view that URL in your browser, you should see the same result as before. This is the way all other, non-index views are accessed.

Often, your application needs more than one view. A document for instance may have an `index` view that displays it, but another `edit` view to change its contents. To create a second view, create another template called `bye.pt` in `app_templates`. Make it have the following content:

```
<html>
<body>
<p>Bye world!</p>
</body>
</html>
```

Now we need to tell Grok to actually use this template. To do this, modify `src/sample/app.py` so that it reads like this:

```python
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    pass

class Bye(grok.View):
    pass
```

As you can see, all we did was add a class called `Bye` that subclasses from `grok.View`. This indicates to Grok that we want a view named `bye` for the application, just like the `Index` class that was already created for us indicates that we want a view named `index`. A `view` is a way to view some model, in this case installations of our `Sample` application. Note that the view name in the URL is always going to be lowercase, while the class name normally starts with an uppercase letter.

The empty class definition above is enough for Grok to go look in the `app_templates` directory for `bye.pt`. The rule is that a the template should have the same name as the class, but lowercased and with the `.pt` postfix.
Other templating languages

You can also install extensions to allow the use of other templating languages in Grok, see for instance megrok.genshi.

Restart the web server (Ctrl + C, then bin/paster serve parts/etc/deploy.ini). You can now go to a new web page called bye:

http://localhost:8080/test/bye

When you load this web page in a browser, you should see the following text:

Bye world!

3.3 Making our page dynamic

Static web pages are not very helpful if we want to make a dynamic web application. Let’s make a page that shows the result of a very simple calculation: $1 + 1$.

We will use a Zope Page Templates (ZPT) directive to do this calculation inside index.pt template. Change the index.pt to read like this:

```html
<html>
<body>
<p tal:content="python: 1 + 1">this is replaced</p>
</body>
</html>
```

We’ve used the tal:content page template directive to replace the content between the `<p>` and `</p>` tags with something else, in this case the result of the Python expression $1 + 1$.

Since restarting the server is not necessary for changes that are limited to the page templates, you can just reload the web page:

http://localhost:8080/test

You should see the following result:

2

Looking at the source of the web page shows us this:

```html
<html>
<body>
<p>2</p>
</body>
</html>
```
As you can see, the content of the `<p>` tag was indeed replaced with the result of the expression 1 + 1.

### 3.4 Static resources for our web page

In real-world web pages, we almost never publish a web page that just contains bare-bones HTML. We also want to refer to other resources, such as images, CSS files or Javascript files. As an example, let’s add some style to our web page.

To do this, create a new directory called `static` in the `sample` package (so, `src/sample/static`). In it, place a file called `style.css` and put in the following content:

```plaintext
body {
    background-color: #FF0000;
}
```

In order to use it, we also need to refer to it from our `index.pt`. Change the content of `index.pt` to read like this:

```plaintext
<html>
<head>
<link rel="stylesheet" type="text/css"
    tal:attributes="href static/style.css" />
</head>
<body>
<p>Hello world!</p>
</body>
</html>
```

Now restart the server and reload the page:

```
http://localhost:8080/test
```

The web page should now show up with a red background.

You will have noticed we used the `tal:attributes` directive in our `index.pt` now. This uses Zope Page Templates to dynamically generate the link to our file `style.css`.

Let’s take a look at the source code of the generated web page:

```plaintext
<html>
<link rel="stylesheet" type="text/css"
    href="http://localhost:8080/test/@@/sample/style.css" />
<body>
<p>Hello world!</p>
</body>
</html>
```

As you can see, the `tal:attributes` directive is gone and has been replaced with the following URL to the actual stylesheet:

```
http://localhost:8080/test/@@/sample/style.css
```
We will not go into the details of the structure of the URL here, but we will note that because it’s generated this way, the link to style.css will continue to work no matter where you install your application (i.e. in a virtual hosting setup).

Pulling in images or javascript is very similar. Just place your image files and .js files in the static directory, and create the URL to them using static/<filename> in your page template.

3.5 Using view methods

Unassociated templates

If you have followed the tutorial so far, you will have an extra template called bye.pt in your app_templates directory. Since in the given app.py we we have no more class using it, the bye.pt template will have become unassociated. When you try to restart the server, Grok will give you a warning like this:

UserWarning: Found the following unassociated template(s) when grokking 'sample.app': bye. Define view classes inheriting from grok.View to enable the template(s).

To get rid of this warning, simply remove bye.pt from your app_templates directory.

ZPT is deliberately limited in what it allows you to do with Python. It is good application design practice to use ZPT for fairly simple templating purposes only, and to do anything a bit more complicated in Python code. Using ZPT with arbitrary Python code is easy: you just add methods to your view class and use them from your template.

Let’s see how this is done by making a web page that displays the current date and time. We will use our Python interpreter to find out what works:

$ python
Python 2.6.2
Type "help", "copyright", "credits" or "license" for more information.
>>> We will need Python’s datetime class, so let’s import it:

>>> from datetime import datetime

Note that this statement brings us beyond the capabilities of simple ZPT use; it is not allowed to import arbitrary Python modules from within a ZPT template; only Python expressions (with a result) are allowed, not statements such as from .. import ...

Let’s get the current date and time:

>>> now = datetime.now()
This gives us a date time object; something like this:

```python
>>> now
datetime.datetime(2007, 2, 27, 17, 14, 40, 958809)
```

Not very nice to display on a web page, so let’s turn it into a prettier string using the formatting capabilities of the `datetime` object:

```python
>>> now.strftime('%Y-%m-%d %H:%M')
'2007-02-27 17:14'
```

That looks better.

So far nothing new; just Python. We will integrate this code into our Grok project now. Go to `app.py` and change it to read like this:

```python
import grok
from datetime import datetime

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def current_datetime(self):
        now = datetime.now()
        return now.strftime('%Y-%m-%d %H:%M')
```

We’ve simply added a method to our view that returns a string representing the current date and time. Now to get this string in our page template. Change `index.pt` to read like this:

```html
<html>
<body>
<p tal:content="python:view.current_datetime()">Hello world!</p>
</body>
</html>
```

Restart the server. This is needed as we changed the content of a Python file (`app.py`). Now reload our index page to see whether it worked:

```
http://localhost:8080/test
```

You should see a web page with a date and time like this on your screen now:

```
2007-02-27 17:21
```

What happened here? When viewing a page, the view class (in this case `Index` is instantiated by the framework. The name `view` in the template is always made available and is associated with this instance. We then simply call the method on it in our template.

There is another way to write the template that is slightly shorter and may be easier to read in some cases, using a ZPT path expression:
Generating HTML from Python

While usually you will be using templates to generate HTML, sometimes you want to generate complicated HTML in Python and then include it in an existing web page. For reasons of security against cross-site scripting attacks, TAL will automatically escape any HTML into &gt; and &lt;. With the structure directive, you can tell TAL explicitly to not escape HTML this way, so it is passed literally into the template.

Let’s see how this is done. Change app.pt to read like this:

```python
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def some_html(self):
        return '<b>ME GROK BOLD</b>'
```

and then change index.pt to read like the following:

```html
<html>
<body>
<p tal:content="structure python:view.some_html()"></p>
</body>
</html>
```

Let’s take another look at our web page:

```
http://localhost:8080/test
```

You should see the following text (in bold):

ME GROK BOLD

This means the HTML we generated from the some_html method was indeed successfully integrated in our web page. Without the the structure directive, you would’ve seen the following instead:

```
<b>ME GROK BOLD</b>
```
3.7 Completely Python-driven views

Setting the content-type

When generating the complete content of a page yourself, it’s often useful to change the content-type of the page to something else than text/plain. Let’s change our code to return simple XML and set the content type to text/xml:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def render(self):
        self.response.setHeader('Content-Type', 'text/xml; charset=UTF-8')
        return "<doc>Some XML</doc>"
```

All views in Grok have a response property that you can use to manipulate response headers.

Sometimes it is inconvenient to have to use a template at all. Perhaps we are not returning a HTML page at all, for instance. In this case, we can use the special render method on a view.

Modify app.py so it reads like this:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def render(self):
        return "ME GROK NO TEMPLATE"
```

If you were to start up the server with an index.pt template still inside app_templates you would get a an error:

GrokError: Multiple possible ways to render view <class 'sample.app.Index'>. It has both a ‘render’ method as well as an associated template.

In the face of ambiguity Grok, like Python, refuses to guess. To resolve this error, remove index.pt from the app_templates directory.

Now take another look at our test application:

```
http://localhost:8080/test
```
You should see the following:

```
ME GROK NO TEMPLATE
```

You should see this even when you view the source of the page. When looking at the content type of this page, you will see that it is `text/plain`.

### 3.8 Doing some calculation before viewing a page

Instead of calculating some values in a method call from the template, it is often more useful to calculate just before the web page’s template is calculated. This way you are sure that a value is only calculated once per view, even if you use it multiple times.

You can do this by defining an `update` method on the view class. Modify `app.py` to read like this:

```python
import grok
class Sample(grok.Application, grok.Container):
    pass
class Index(grok.View):
    def update(self):
        self.alpha = 2 ** 8
```

This sets a name `alpha` on the view just before the template is being displayed, so we can use it from the template. You can set as many names on `self` as you like.

Now we need a template `index.pt` that uses `alpha`:

```html
<html>
<body>
<p tal:content="python:view.alpha">result</p>
</body>
</html>
```

Restart the server and then let’s take another look at our application:

```
http://localhost:8080/test
```

You should see 256, which is indeed 2 raised to the power 8.

### 3.9 Reading URL parameters

When developing a web application, you don’t just want to output data, but also want to use input. One of the simplest ways for a web application to receive input is by retrieving information as a URL parameter. Let’s devise a web application that can do sums for us. In this application, if you enter the following URL into that application:
http://localhost:8080/test?value1=3&value2=5

you should see the sum (8) as the result on the page.
Modify app.py to read like this:

```python
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def update(self, value1, value2):
        self.sum = int(value1) + int(value2)
```

We need an index.pt that uses sum:

```html
<html>
<body>
<p tal:content="python:view.sum">sum</p>
</body>
</html>
```

Restart the server. Now going to the following URL should display 8:

http://localhost:8080/test?value1=3&value2=5

Other sums work too, of course:

http://localhost:8080/test?value1=50&value2=50

What if we don’t supply the needed parameters (value1 and value2) to the request? We get an error:

http://localhost:8080/test

You can look at the window where you started up the server to see the error traceback. This is the relevant complaint:

```
TypeError: Missing argument to update(): value1
```

We can modify our code so it works even without input for either parameter:

```python
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def update(self, value1=0, value2=0):
        self.sum = int(value1) + int(value2)
```

Restart the server, and see it can now deal with missing parameters (they default to 0).
3.10 Simple forms

**Automatic forms**

Creating forms and converting and validating user input by hand, as shown in this section, can be rather cumbersome. With Grok, you can use the Zope Toolkit’s *schema* and *formlib* systems to automate this and more. This will be discussed in a later section. TDB

Entering the parameters through URLs is not very pretty. Let’s use a form for this instead. Change *index.pt* to contain a form, like this:

```html
<html>
<body>
<form tal:attributes="action python:view.url()" method="GET">
  Value 1: <input type="text" name="value1" value="" /><br />
  Value 2: <input type="text" name="value2" value="" /><br />
  <input type="submit" value="Sum!" />
</form>
<p>The sum is: <span tal:replace="python:view.sum">sum</span></p>
</body>
</html>
```

One thing to note here is that we dynamically generate the form’s *action*. We make the form submit to itself, basically. Grok views have a special method called *url* that you can use to retrieve the URL of the view itself (and other URLs which we’ll go into later).

Leave the *app.py* as in the previous section, for now. You can now go to the web page:

```
http://localhost:8080/test
```

You can submit the form with some values, and see the result displayed below.

We still have a few bugs to deal with however. For one, if we don’t fill in any parameters and submit the form, we get an error like this:

```
File ".//app.py", line 8, in update
    self.sum = int(value1) + int(value2)
ValueError: invalid literal for int():
```

This is because the parameters were empty strings, which cannot be converted to integers. Another thing that is not really pretty is that it displays a sum (0) even if we did not enter any data. Let’s change *app.py* to take both cases into account:

```python
import grok

class Sample(grok.Application, grok.Container):  
  pass
```

3.10. Simple forms
class *Index*(*grok.View*):
    def *update*(self, *value1=None*, *value2=None*):
        try:
            *value1* = *int*(value1)
            *value2* = *int*(value2)
        except (*TypeError*, *ValueError*):
            self.sum = "No sum"
        return
        self.sum = *value1* + *value2"

We catch any *TypeError* and *ValueError* here so that wrong or missing data does not result in a failure. Instead we display the text “No sum”. If we don’t get any error, the conversion to integer was fine, and we can display the sum.

Restart the server and go to the form again to try it out:

    http://localhost:8080/test
Now we know how to show web pages, we need to go into what we are actually showing: the models. The models contain the display-independent logic of your application. In this chapter we will discuss a number of issues surrounding models: how your views connect to models, and how you can make sure the data in your models is stored safely. As the complexity of our sample applications grows, we will also go into a few more issues surrounding form handling.

### 4.1 A view for a model

So far, we have only seen views that do the work all by themselves. In typical applications this is not the case however - views display information that is stored elsewhere. In Grok applications, views work for models: subclasses of grok.Model or grok.Container. For the purposes of this discussion, we can treat a grok.Container as another kind of grok.Model (more about what makes grok.Container special later XXX).

Our Sample class is a grok.Container, so let’s use Sample to demonstrate the basic principle. Let’s modify app.py so that Sample actually makes some data available:

```python
import grok

class Sample(grok.Application, grok.Container):
    def information(self):
        return "This is important information!"

class Index(grok.View):
    pass
```

In this case, the information ("This is important information!") is just hardcoded, but you can imagine information is retrieved from somewhere else, such as a relational database or the filesystem.

We now want to display this information in our template index.pt:

```xml
<html>
<body>
<p tal:content="python:context.information()">replaced</p>
</body>
</html>
```
Restart the server. When you view the page:

    http://localhost:8080/test

You should now see the following:

This is important information!

Previously we have seen that you can access methods and attributes on the view using the special view name in a template. Similarly, the name context is also available in each template. context allows us to access information on the context object the view is displaying. In this case this is an instance of Sample, our application object.

Separating the model from the view that displays it is an important concept in structuring applications. The view, along with the template, is responsible for displaying the information and its user interface. The model represents the actual information (or content) the application is about, such as documents, blog entries or wiki pages. The model should not know anything about the way it is displayed.

This way of structuring your applications allows you to change the way your model is displayed without modifying the model itself, just the way it is viewed.

Let’s do that by making the view do something to the information. Change app.py again:

```python
import grok

class Sample(grok.Application, grok.Container):
    def information(self):
        return "This is important information!"

class Index(grok.View):
    def reversed_information(self):
        return ".join(reversed(self.context.information()))
```

You can see that it is possible to access the context object (an instance of Sample) from within the view class, by accessing the context attribute. This gets the same object as when we used the context name in our template before.

What we do here is reverse the string returned from the information() method. You can try it on the Python prompt:

```python
>>> ".join(reversed('foo'))
'oof'
```

Now let’s modify the index.pt template so that it uses the reversed_information method:

```html
<html>
<body>
<p>The information:
    <span tal:content="python:context.information()">info</span>
</p>
<p>The information, reversed:
    <span tal:replace="python:view.reversed_information()">info</span>
</body>
</html>
```
Restart the server. When you view the page:

```
http://localhost:8080/test
```

You should now see the following:

- The information: This is important information!
- The information, reversed: !noitamrofni tnatropmi si sihT

### 4.2 Storing data

So far we have only displayed either hardcoded data, or calculations based on end-user input. What if we actually want to store some information, such as something the user entered? The easiest way to do this with Grok is to use the Zope Object Database (ZODB).

The ZODB is a database of Python objects. You can store any Python object in it, though you do need to follow a few simple rules (the “rules of persistence”, which we will go into later). Our Sample application object is stored in the object database, so we can store some information on it.

Let’s create an application that stores a bit of text for us. We will use one view to view the text (index) and another to edit it (edit).

Modify `app.py` to read like this:

```python
import grok

class Sample(grok.Application, grok.Container):
    text = 'default text'

class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return
        self.context.text = text
```

The Sample class gained a class attribute with some default text. In the `update` method of the Edit view you can see we actually set the `text` attribute on the context, if at least a `text` value was supplied by a form. This will set the `text` attribute on the instance of the Sample object in the object database, and thus will override the default `text` class attribute.

Change the `index.pt` template to read like this:
This is a very simple template that just displays the text attribute of the context object (our Sample instance).

Create an edit.pt template with the following content:

```
<html>
<body>
<form tal:attributes="action view/url" method="POST">
Text to store: <input type="text" name="text" value="" />
<input type="submit" value="Store" />
</form>
</body>
</html>
```

This template display a form asking for a bit of text. It submits to itself.

Restart the server. Let’s first view the index page:

```
http://localhost:8080/test
```

You should see default text.

Now let’s modify the text by doing to the edit page of the application:

```
http://localhost:8080/test/edit
```

Type in some text and press the “Store” button. Since it submits to itself, we will see the form again, so go to the index page manually:

```
http://localhost:8080/test
```

You should now see the text you just entered on the page. This means that your text was successfully stored in the object database!

You can even restart the server and go back to the index page, and your text should still be there.

### 4.3 Redirection

Let’s make our application a bit easier to use. First, let’s change index.pt so it includes a link to the edit page. To do this, we will use the url method on the view:
Giving url a single string argument will generate a URL to the view named that way on the same object (test), so in this case test/edit.

Now let’s change the edit form so that it redirects back to the index page after you press the submit button:

```python
import grok

class Sample(grok.Application, grok.Container):
    text = 'default text'

class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return
        self.context.text = text
        self.redirect(self.url('index'))
```

The last line is the new one. We use the url method on the view to construct a URL to the index page. Since we’re in the template, we can simply call url on self. Then, we pass this to another special method available on all grok.View subclasses, redirect. We tell the system to redirect to the index page.

### 4.4 Showing the value in the form

Let’s change our application so it displays what we stored the edit form as well, not just on the index page.

To make this work, change edit.pt so it reads like this:

```html
<html>
<body>
<form tal:attributes="action view/url" method="POST">
Text to store: <input type="text" name="text" tal:attributes="value python:context.text" value="" />
<input type="submit" value="Store" />
</form>
</body>
</html>
```

The only change is that we have used tal:attributes to include the value of the text attribute of the context object in the form.
4.5 The rules of persistence

These are the “rules of persistence”:

- You should subclass classes that want to store data from `persistent.Persistent` so that it’s easy to store them in the ZODB. The simplest way to do this with Grok is to subclass from `grok.Model` or `grok.Container`.

- Instances that you want to store should be connected to other persistent classes that are already stored. The simplest way to do this with Grok is to attach them somehow to the `grok.Application` object, directly or indirectly. This can be done by setting them as an attribute, or by putting them in a container (if you made your application subclass `grok.Container`).

- To make sure that the ZODB knows you changed a mutable attribute (such as a simple Python list or dictionary) in your instance, set the special `_p_changed` attribute on that instance to `True`. This is only necessary if that attribute is not `Persistent` itself. It is also not necessary when you create or overwrite an attribute directly using `=`.

If you construct your application’s content out of `grok.Model` and `grok.Container` subclasses you mostly follow the rules already. Just remember to set `_p_changed` in your methods if you find yourself modifying a Python list (with `append`, for instance) or dictionary (by storing a value in it).

The code in the section Storing data is a simple example. We in fact have to do nothing special at all to obey the rules of persistence in that case.

If we use a mutable object such as a list or dictionary to store data instead, we do need to take special action. Let’s change our example code (based on the last section) to use a mutable object (a list):

```python
import grok

class Sample(grok.Application, grok.Container):
    def __init__(self):
        super(Sample, self).__init__()
        self.list = []

class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return
            # this code has a BUG!
        self.context.list.append(text)
        self.redirect(self.url('index'))
```

We have now changed the `Sample` class to do something new: it has an `__init__` method. Whenever you create the `Sample` application object now, it will be created with an attribute called `list`, which will contain an empty Python list.

We also make sure that the `__init__` method of the superclass still gets executed, by using the regular Python `super` idiom. If we didn’t do that, our container would not be fully initialized.
You will also notice a small change to the `update` method of the `Edit` class. Instead of just storing the text as an attribute of our `Sample` model, we add each text we enter to the new `list` attribute on.

Note that this code has a subtle bug in it, which is why we’ve added the comment. We will see what bug this is in a little bit. First, though, let’s change our templates.

We change `index.pt` so that it displays the list:

```html
<html>
<body>
We store the following texts:
<ul>
  <li tal:repeat="text python:context.list" tal:content="text"></li>
</ul>
<a tal:attributes="href python:view.url('edit')">Add a text</a>
</body>
</html>
```

We’ve also changed the text of the link to the `edit` page to reflect the new adding behavior of our application.

We need to undo the change to the `edit.pt` template that we made in the last section, as each time we edit a text we now `add` a new text, instead of changing the original. There is therefore no text to show in as the input value anymore:

```html
<html>
<body>
<form tal:attributes="action view/url" method="POST">
Text to store: <input type="text" name="text" value="" /><br />
<input type="submit" value="Store" />
</form>
</body>
</html>
```

### evolution

What to do when you change an object’s storage structure while your application is already in production? In a later section, we will introduce Zope Toolkit’s object evolution mechanism that allows you to update objects in an existing object database. TDB

Let’s restart the server. If you have followed the tutorial from the last section, you will now see an error when you look at the front page of the application:

A system error occurred.

Look at the output we got when we tried to load our page:

`AttributeError: 'Sample' object has no attribute 'list'`

4.5. The rules of persistence
But we just changed our object to have an attribute `list`, right? Yes we did, but only for new instances of the Sample object. What we are looking at is the sample object from before, still stored in the object database. It has no such attribute. This isn’t a bug by the way (for our actual bug, see later in this section): it is just a database problem.

What to do now? The simplest action to take during development is to simply remove our previously installed application, and create a new one that *does* have this attribute. Go to the Grok admin screen:

```
http://localhost:8080
```

Select the application object (test) and delete it. Now install it again, as test. Now go to its edit screen and add a text:

```
http://localhost:8080/test/edit
```

Click on add a text and add another text. You will see the new texts appear on the index page.

Everything is just fine now, right? In fact, not so! Now we will get to our bug. Restart the server and look at the index page again:

```
http://localhost:8080/test
```

None of the texts we added were saved! What happened? We broke the third rule of persistence as described above: we modified a mutable attribute and did not notify the database that we made this change. This means that the object database was not aware of our change to the object in memory, and thus never saved it to disk.

We can easily amend this by adding one line to the code:

```python
import grok

class Sample(grok.Application, grok.Container):
    def __init__(self):
        super(Sample, self).__init__()
        self.list = []

class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return
        self.context.list.append(text)
        self.context._p_changed = True
        self.redirect(self.url('index'))
```

We’ve now told the server that the context object has changed (because we modified a mutable sub-object), by adding the line:
If you now add some texts and then restart the server, you will notice the data is still there: it has successfully been stored in the object database.

The code shown so far is a bit ugly in the sense that typically we would want to manage our state in the model code (the `Sample` object in this case), and not in the view. Let’s make one final change to show what that would look like:

```python
import grok

class Sample(grok.Application, grok.Container):
    def __init__(self):
        super(Sample, self).__init__()
        self.list = []

    def addText(self, text):
        self.list.append(text)
        self._p_changed = True

class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return self.context.addText(text)
        self.redirect(self.url('index'))
```

As you can see, we have created a method `addText` to the model that takes care of amending the list and informing the ZODB about it. This way, any view code can safely use the API of `Sample` without having to worry about the rules of persistence itself, as that is the model’s responsibility.

## 4.6 Explicitly associating a view with a model

How does Grok know that a view belongs to a model? In the previous examples, Grok has made this association automatically. Grok could do this because there was only a single model defined in the module (`Sample`). In this case, Grok is clever enough to automatically associate all views defined elsewhere in the same module to the only model. Behind the scenes Grok made the model the `context` of the views.

Everything that Grok does implicitly you can also tell Grok to do explicitly. This will come in handy later, as you may sometimes need (or want) to tell Grok what to do, overriding its default behavior. To associate a view with a model automatically, you use the `grok.context` class annotation.

What is a class annotation? A class annotation is a declarative way to tell grok something about a Python class. Let’s look at an example. We will change `app.py` in the example from a second view to demonstrate the use of `grok.context`:
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    grok.context(Sample)

class Bye(grok.View):
    grok.context(Sample)

This code behaves in exactly the same way as the previous example in A second view, but has the relationship between the model and the view made explicit, using the grok.context class annotation.

grok.context is just one class annotation out of many. We will see another one (grok.name) in the next section.

4.7 A second model

How to combine models into a single application?
Curious now about how to combine models into a single application? Can’t wait? Look at the section Containers coming up next, or Traversal later on. TDB

We will now extend our application with a second model. Since we haven’t explained yet how to combine models together into a single application, we will just create a second application next to our first one. Normally we probably wouldn’t want to define two applications in the same module, but we are trying to illustrate a few points, so please bear with us. Change app.py so it looks like this:

import grok

class Sample(grok.Application, grok.Container):
    pass

class Another(grok.Application, grok.Model):
    pass

class SampleIndex(grok.View):
    grok.context(Sample)
    grok.name('index')

class AnotherIndex(grok.View):
    grok.context(Another)
    grok.name('index')

You can see we now defined a second application class, Another. It subclasses from grok.Application to make it an installable application.
It also subclasses from grok.Model. There is a difference between grok.Model and grok.Container, but for the purpose of the discussion we can ignore it for now. We just figured we should use grok.Model for some variety, though we could have indeed subclassed from grok.Container instead.

We also define two templates, one called sampleindex.pt:

```html
<html>
<body>
<p>Sample index</p>
</body>
</html>
```

And one called anotherindex.pt:

```html
<html>
<body>
<p>Another index</p>
</body>
</html>
```

We have named the templates the name as the lowercased class names as the views, so that they get associated with them.

You will have noticed we have used grok.context to associate the views with models. We actually have to do this here, as Grok refuses to guess in the face of ambiguity. Without the use of grok.context, we would have seen an error like this when we start up:

```
GrokError: Multiple possible contexts for <class 'sample.app.AnotherIndex'>, please use grok.context.
```

So, we use grok.context to explicitly associate SampleIndex with the Sample application, and again to associate AnotherIndex with the Another application.

We have another problem: the intent is for these views to be index views. This cannot be deduced automatically from the name of the view classes however, and left to its own devices, Grok would have called the views sampleindex and anotherindex.

Luckily we have another class annotation that can help us here: grok.name. We can use it on both view classes (grok.name('index')) to explicitly explain to Grok what we want.

You can now try to restart the server and create both applications in the Grok Admin interface. They should display the correct index pages when you look at them.

We can see that the introduction of a second model has complicated our code a bit, though you will hopefully agree with us that it is still quite readable. We could have avoided the whole problem by simply placing Another and its views in another module such as another.py. Its associated templates would then need to be placed in a directory another_templates. Often you will find it possible to structure your application so you can use Grok’s default conventions.
4.8 Containers

A container is a special kind of model object that can contain other objects. Our Sample application is already a container, as it subclasses grok.Container. What we will do in this section is build an application that actually puts something into that container.

Grok applications are typically composed of containers and models. Containers are objects that can contain models. This includes other containers, as a container is just a special kind of model.

From the perspective of Python, you can think of containers as dictionaries. They allow item access (container[‘key’]) to get at its contents. They also define methods such as keys() and values(). Containers do a lot more than Python dictionaries though: they are persistent, and when you modify them, you don’t have to use _p_changed anywhere to notice you changed them. They also send out special events that you can listen to when items are placed in them or removed from them. For more on that, see the section on events (TDB).

Our application object will have a single index page that displays the list of items in the container. You can click an item in the list to view that item. Below the list, it will display a form that allows you to create new items.

Here is the app.py of our new application:

```python
import grok

class Sample(grok.Application, grok.Container):
    pass

class Entry(grok.Model):
    def __init__(self, text):
        self.text = text

class SampleIndex(grok.View):
    grok.context(Sample)
    grok.name(‘index’)

    def update(self, name=None, text=None):
        if name is None or text is None:
            return
        self.context[name] = Entry(text)

class EntryIndex(grok.View):
    grok.context(Entry)
    grok.name(‘index’)
```

As you can see, Sample is unchanged. We have also created our first non-application object, Entry. It is just a grok.Model. It needs to be created with an argument text and this text is stored in it. We intend to place instances of Entry in our Sample container.

Next are the views. We have an index page for the Sample container. When its update() is triggered with two values, name and text, it will create a new Entry instance with the given text, and place it under the container under the name name. We use the dictionary-like interface of our Sample container to put our new Entry in the container.
Here is the associated template for SampleIndex, sampleindex.pt:

```html
<html>
<head>
</head>
<body>
<h2>Existing entries</h2>
<ul>
<li tal:repeat="key python:context.keys()">
<a tal:attributes="href python:view.url(key)"
   tal:content="python:key"></a>
</li>
</ul>
<h2>Add a new entry</h2>
<form tal:attributes="action python:view.url()" method="POST">
  Name: <input type="text" name="name" value="" /><br />
  Text: <input type="text" name="text" value="" /><br />
  <input type="submit" value="Add entry" />
</form>
</body>
</html>
```

The first section in the template (`<h2>Existing entries</h2>`) displays a list of the items in the container. We again use dictionary-like access using `keys()` to get a list of all the names of the items in the container. We create a link to these items using `view.url()`.

The next section (`<h2>Add a new entry</h2>`) displays a simple form that submits to the index page itself. It has two fields, `name` and `text`, which we already have seen handled by `update()`.

Finally, we have an index page for `Entry`. It just has a template to display the `text` attribute:

```html
<html>
<head>
</head>
<body>
<h2>Entry <span tal:replace="python:context.__name__"></span></h2>
<p tal:content="python:context.text"></p>
</body>
</html>
```

Restart the server and try this application. Call your application `test`. Pay special attention to the URLs.

First, we have the index page of our application:

```
http://localhost:8080/test
```

When we create an entry called `hello` in the form, and then click on it in the list, you see an URL that looks like this:

```
http://localhost:8080/test/hello
```
We are now looking at the index page of the instance of `Entry` called `hello`.

What kind of extensions to this application can we think of? We could create an `edit` form that allows you to edit the text of entries. We could modify our application so that you can not just add instances of `Entry`, but also other containers. If you made those modifications, you would be on your way to building your own content management system with Grok.