Community Experience Distilled

Spring Python 1.1

Create powerful and versatile Spring Python applications using pragmatic libraries and useful abstractions

Greg Lee Turnquist

[PACKT] PUBLISHING open source* community experience distilled

Spring Python 1.1

Create powerful and versatile Spring Python applications using pragmatic libraries and useful abstractions

Greg Lee Turnquist

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Spring Python 1.1

First published: May 2010

Production Reference: 1180510

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849510-66-0

www.packtpub.com

Cover Image by Parag Kadam (paragvkadam@gmail.com)

Credits

Author
Greg Lee Turnquist

Reviewers
Bala Sundarasamy
Russ Miles
Sylvain Hellegouarch

Acquisition Editor
Steven Wilding

Development Editor
Mehul Shetty

Technical Editor
Aditya Belpathak

Indexer
Hemangini Bari

Editorial Team Leader
Aanchal Kumar

Project Team Leader
Lata Basantani

Project Coordinator
Joel Goveya

Proofreader
Lesley Harrison

Graphics
Geetanjali Sawant

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Greg Lee Turnquist has worked in the software industry since 1997. He is an active participant in the open source community, and has contributed patches to several projects including MythTV, Spring Security, MediaWiki, and the TestNG Eclipse plugin. As a test-bitten script junky, he has always sought the right tool for the job. He is a firm believer in agile practices and automated testing. He has developed distributed systems, LAMP-based setups, and supported mission critical systems hosted on various platforms.

After graduating from Auburn University with a Master's in Computer Engineering, Greg started working with Harris Corporation. He worked on many contracts utilizing many types of technology. In 2006, he created the Spring Python project, which became the first official Spring Extension to reach live status. He recently joined SpringSource as part of their international software development team.

I would like to extend thanks to Russ Miles and Sylvain Hellegouarch for taking the time to technically review this book and provide valuable feedback. I also thank my father, Dr. Paul Turnquist, for providing inspiration to write. And most importantly, I thank my wife Sara, for the support, encouragement, and patience.

About the Reviewers

Bala Sundarasamy, graduated from the College of Engineering, Guindy. He has an extensive experience of more than 17 years in designing and building applications using Java and .Net technologies. He has performed various roles in technical functions and project delivery with some of the leading software consulting companies in India.

He has also taught numerous young developers to write good object-oriented code using Java and C#. He has proven expertise in training fresh engineers to adopt industry standard best practices and processes in software writing.

Russ Miles is the author of three bestselling books: *AspectJ Cookbook*, *Learning UML 2.0*, and *Head First Software Development* (all by O' Reilly Media). In particular, writing for the *Head First* series was an incredible learning experience in how to turn dry, technical subjects into great learning experiences; he has successfully applied these skills to the development of several successful courses and workshops.

He is a frequent speaker on many, wide-ranging technical subjects including the fun of being a self-proclaimed 'polyglot programmer'. A certified Scrum Master and, alongside the more specific technical coaching, he works with many companies as they come to terms with the fact that software development is ultimately about people.

He is the principle consultant and founder of OpenCredo, a high level software development consultancy based in London. As part of his daily activities, Russ writes for several publications (when he finds the time!) on topical software development subjects of the day.

I'd like to thank my beautiful wife, Corinne and my 'little monster', Mali for their support, laughs and hair tugs (just Mali).

Sylvain Hellegouarch, is a senior software developer at Ubikod, a company dedicated to innovative mobile solutions. He has a passion for open-source software and has created many Python projects targeting social communication based on the AtomPub and XMPP protocols. He wrote the *CherryPy Essentials* book, which was published by Packt Publishing in 2007, a hands-on book about CherryPy, a popular web framework. His interests are currently pushing him to the rich world of the semantic web, linked data and how people interact with each other.

Table of Contents

Preface

Back in 2004, a new revolution was brewing in the Java industry: the Spring Framework. It challenged the concept of monolithic, one-size-fits-all application servers by offering pragmatic solutions without heavy handed requirements. Contrary to previous frameworks, Spring did not require all-or-nothing adoption. Instead, it offered a practical approach by providing convenient abstractions over commonly used patterns. It also embraced the concept of dependency injection, allowing non-intrusive adoption of powerful options.

The Spring Framework doesn't hinge on the technology that it is coded in. The concepts and methods, known as the *Spring way*, work anywhere. In 2006, I created the Spring Python project. This was to combine the non-intrusive, powerful concepts of Spring with the nimble, dynamic flexibility of Python.

This book will guide you through the building blocks of Spring Python with the aim of giving you the key to build better Python applications.

The first seven chapters show you how to download and get started with Spring Python. Each of these chapters introduces a different module you can use to build applications with. The eighth chapter shows how to use the modules in concert to build a more powerful yet easier to maintain application. The ninth chapter introduces a command-line utility used to rapidly create applications. The final chapter shows how to integrate Python and Java together, quickly and easily.

I have written this book with the intention that developers can discover the scope and beauty of the *Spring way* and how it can enrich development, without complicating it. I often look at software development as more craftsmanship than science, and believe Spring Python can make for a valuable tool in any developer's tool box.

What this book covers

Chapter 1: Getting Started with Spring Python gives you a quick glance at various features of Spring Python, followed by information on how to download and install it.

Chapter 2: The Heart of Spring Python – Inversion of Control introduces you to Spring Python's core container which is reused through the rest of the book to empower testing and non-intrusive features.

Chapter 3: Adding Services to APIs shows how to smoothly add cross cutting services to your code using Spring Python's aspect oriented programming.

Chapter 4: Easily Writing SQL Queries with Spring Python shows you how to rapidly write pure SQL queries without dealing with mind-numbing boilerplate code. It also shows how Spring Python works nicely with ORM-based persistence.

Chapter 5: Adding Integrity to your Data Access with Transactions shows how to seamlessly wrap business code with SQL transactions.

Chapter 6: Securing your Application with Spring Python shows how to wrap web applications with flexible authentication and authorization services, while easily supporting policy changes. It also shows how to code custom extensions to integrate with security systems not currently supported.

Chapter 7: Scaling your Application Across Nodes with Spring Python's Remoting shows how Spring Python nicely integrates with Pyro, the RPC library for Python. This powerful combination will allow you to non-intrusively develop simple apps on your desktop and retool them for multi-node production systems

Chapter 8: Case Study I – Integrating Spring Python with your Web Application invites you to use all of Spring Python's components to build a strong, secured banking application. This shows how the various parts of Spring Python work together in concert to empower you the developer.

Chapter 9: Creating Skeleton Apps with Coily introduces Spring Python's plugin-based Coily tool to rapidly create web applications.

Chapter 10: Case Study II – Integration Spring Python with your Java Application ends the book by showing how easy it is to link Java and Python components together using Jython. This allows polyglot programmers to mix together their favorite services.

What you need for this book

You will need Python 2.4 or above and Spring Python 1.1.0.FINAL (covered in Chapter 1).

Who this book is for

This book is for Python developers who want to take their applications to the next level, by adding/using parts that scale their application up, without adding unnecessary complexity. It is also helpful for Java developers who want to mix in some Python to speed up their coding effort.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We added `closeAccount` to `SpringBankView` to let the customer close an existing account it its balance is zero."

A block of code is set as follows:

```
def get_article(self, article):
    if article in self.cache:
        return self.cache[article]
        else:
        return self.delegate.get_article(article)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
def get_article(self, article):
    if article in self.cache:
        return self.cache[article]
        else:
        return self.delegate.get_article(article)
```

Any command-line input or output is written as follows:

```
$ svn checkout https://src.springframework.org/svn/se-springpython-py/
trunk/springpython springpython
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Let's look at the layout of the **gen-cherrypy-app** plugin as an example."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit `https://www.packtpub.com//sites/default/files/downloads/0660_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or web site name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Spring Python

Spring Python takes the concepts of the Spring Framework and Spring Security, and brings them to the world of Python. It isn't a simple line-by-line port of the code. Instead, it takes some powerful ideas that were discovered in the realm of Java, and pragmatically applies them in the world of Python.

Spring (Java) provides many simple, easy-to-use functional parts to assemble applications instead of a monolithic framework to extend. Spring Python uses this same approach. This means we can use as little or as much Spring Python as we need to get the job done for each Python application.

In this chapter, we will learn:

- About Spring Python's a non-invasive API which makes it easy to use other libraries without having to make major changes to your own code base
- How Spring Python uses inversion of control to decouple object creation from object usage to empower the developer
- How Spring Python provides the means to help professional Python developers by offering a non-invasive API to easily access advanced services
- The ways in which Spring Python offers professional Java developers an easy way to mix Python and Java together through the combination of Python/Jython/Java
- How to install the library from both binary and source code
- How extensible Spring Python is, and also some links to the Spring Python community

Spring Python for Python developers

You have already picked one of the most popular and technically powerful dynamic languages to develop software, Python. Spring Python makes it even easier to solve common problems encountered by Python developers every day.

Exploring Spring Python's non-invasive nature

Spring Python has a non-invasive nature, which means it is easy to adopt the parts that meet your needs, without rewriting huge blocks of code. For example, Pyro (`http://pyro.sourceforge.net`) is a 3rd party library that provides an easy way to make remote procedure calls.

In order to demonstrate the *Spring way* of non-invasiveness, let's code a simple service, publish it as a web service using Pyro's API, and then publish it using Spring Python's wrapper around Pyro. This will show the difference in how Spring Python simplifies API access for us, and how it makes the 3rd party library easier to use without as much rework to our own code.

1. First, let's write a simple service that parses out the parameters from a web request string:

   ```
   class ParamParser(object):
       def parse_web_parms(self, parm):
           return [tuple(p.split("=")) for p in parm.split("&")]
   ```

2. Now we can write a simple, functional piece of code that uses our service in order to have a working version.

   ```
   parser = ParamParser()
   parser.parse_web_parms("pages=5&article=Spring_Python")
   ```

   This is just instantiating the `ParamParser` and accessing the function. To make this a useful internet service, it needs to be instantiated on a central server and should be configured to listen for calls from clients.

3. The next step is to advertise it as a web service using the API of `Pyro`. This will make it reachable by multiple `Pyro` clients. To do this, we define a `daemon` which will host our service on port `9000` and initialize it.

   ```
   import Pyro

   daemon = Pyro.core.Daemon(host="localhost", port="9000")
   Pyro.core.initServer()
   ```

4. Next, we create a `Pyro` object instance to act as proxy to our service as well as an instance of our `ParamParser`. We configure the proxy to delegate all method calls to our service.

```
pyro_proxy = Pyro.core.ObjBase()
parser = ParamParser()
pyro_proxy.delegateTo(parser)
```

5. Finally, we register the `pyro_proxy` object with the `daemon`, and startup a listen-dispatch loop so that it's ready to handle requests:

```
daemon.connect(pyro_proxy, "mywebservice")
daemon.requestLoop(True)
```

When we run this server code, an instance of our `ParamParser` will be created and advertised at `PYROLOC://localhost:9000/mywebservice`.

To make this service complete, we need to create a `Pyro` client that will call into our service. The proxy seamlessly transfers Python objects over the wire using the `Pyro` library, in this case the tuple of request parameters.

```
import Pyro

url_base = "PYROLOC://localhost:9000"
client_proxy = Pyro.core.getProxyForURI(  \
    url_base + "/mywebservice")
print client_proxy.parse_web_parms(  \
    "pages=5&article=Spring_Python")
```

The `Pyro` library is easy to use. One key factor is how our `ParamParser` never gets tightly coupled to the `Pyro` machinery used to serve it to remote clients. However, it's very invasive.

What if we had already developed a simple application on a single machine with lots of methods making use of our utility? In order to convert our application into a client-server application, we would have to rewrite it to use the `Pyro` client proxy pattern everywhere that it was called. If we miss any instances, we will have bugs that need to be cleaned up. If we had written automated tests, they would also have to be rewritten as well. Converting a simple, one-machine application into a multi-node application can quickly generate a lot of work.

That is where Spring Python comes in. It provides a different way of creating objects which makes it easy for us to replace a local object with a remoting mechanism such as `Pyro`. Later on, we will explore the concepts of Spring Python's container in more detail.

Let's utilize Spring Python's container to create our parser and also to serve it up with Pyro.

```
from springpython.config import PythonConfig
from springpython.config import Object
from springpython.remoting.pyro import PyroServiceExporter
from springpython.remoting.pyro import PyroProxyFactory

class WebServiceContainer(PythonConfig):
    def __init__(self):
        super(WebServiceContainer, self).__init__()

    @Object(lazy_init=True)
    def my_web_server(self):
        return PyroServiceExporter(service=ParamParser(),
                                   service_name="mywebservice",
                                   service_port=9000)

    @Object(lazy_init=True)
    def my_web_client(self):
        myService = PyroProxyFactory()
        myService.service_url="PYROLOC://localhost:9000/mywebservice"
        return myService
```

With this container definition, it is easy to write both a server application as well as a client application. To spin up one instance of our Pyro server, we use the following code:

```
from springpython.context import ApplicationContext
container = ApplicationContext(WebServiceContainer())
container.get_object("my_web_server")
```

The client application looks very similar.

```
from springpython.context import ApplicationContext
container = ApplicationContext(WebServiceContainer())
myService = container.get_object("my_web_client")
myService.parse_web_parms("pages=5&article=Spring_Python")
```

The Spring Python container works by *containing all the definitions for creating key objects*. We create an instance of the container, ask it for a specific object, and then use it.

This easily looks like just as much (if not more) code than using the `Pyro API` directly. So why is it considered less invasive?

Looking at the last block of code, we can see that we are no longer creating the parser or the Pyro proxy. Instead, we are relying on the container to create it for us. The Spring Python container decouples the creation of our parser, whether its for a local application, or if it uses `Pyro` to join them remotely. The server application doesn't know that it is being exported as a `Pyro` service, because all that information is stored in the `WebServiceContainer`. Any changes made to the container definition aren't seen by the server application code.

The same can be said for the client. By putting creation of the client inside the container, we don't have to know whether we are getting an instance of our service or a proxy. This means that additional changes can be made inside the definition of the container of Spring Python, without impacting our client and server apps. This makes it easy to split the server and client calls into separate scripts to be run in separate instances of Python or on separate nodes in our enterprise.

This demonstrates how it is possible to mix in remoting to our existing application. By using this pattern of delegating creation of key objects to the container, it is easy to start with simple object creation, and then layer on useful services such as remoting. Later in this book, we will also see how this makes it easy to add other services like transactions and security. Due to Spring Python's open ended design, we can easily create new services and add them on without having to alter the original framework.

Adding in some useful templates

In addition to the non-invasive ability to mix in services, Spring Python has several utilities that ease the usage of low level APIs through a template pattern. The template pattern involves capturing a logical flow of steps. What occurs at each step is customizable by the developer, while still maintaining the same overall sequence.

One example where a template would be useful is for writing a SQL query. Coding SQL queries by hand using Python's database API (`http://www.python.org/dev/peps/pep-0249`) is very tedious. We must properly handle errors and harvest the results. The extra code involved with connecting things together and handling issues is commonly referred to as plumbing code. Let's look at the following code to see how Python's database API functions.

The more plumbing code we have to maintain, the higher the cost. Having an application with dozens or hundreds of queries can become unwieldy, even cost prohibitive to maintain.

1. Using Python's database API, we only have to write the following code once for setup.

```
### One time setup
import MySQLdb
conn = MySQLdb.connection(username="me",
                          password"secret",
                          hostname="localhost",
                          db="springpython")
```

2. Now let's use Python's database API to perform a single query.

```
### Repeated for every query
cursor = conn.cursor()
results = []
try:
  cursor.execute("""select title, air_date, episode_number, writer
                      from tv_shows where name = %s""",
                      ("Monty Python",))
    for row in cursor.fetchall():
        tvShow = TvShow(title=row[0],
                        airDate=row[1],
                        episodeNumber=row[2],
                        writer=row[3])
        results.append(tvShow)
finally:
    try:
        cursor.close()
    except Exception:
        pass
conn.close()
return results
```

   The specialized code we wrote to look up TV shows is contained in the execute statement and also the part that creates an instance of `TvShow`. The rest is just plumbing code needed to handle errors, manage the database cursor, and iterate over the results.

   This may not look like much, but have you ever developed an application with just one SQL query? We could have dozens or even hundreds of queries, and having to repeatedly code these steps can become overwhelming. Spring Python's `DatabaseTemplate` lets us just inject the query string and and a row mapper to reduce the total amount of code that we need to write.

3. We need a slightly different setup than before.

```
"""One time setup"""
from springpython.database.core import *
from springpython.database.factory import *
connectionFactory = MySQLConnectionFactory(username="me",
                                           password="secret",
                                           hostname="localhost",
                                           db="springpython")
```

4. We also need to define a mapping to generate our `TvShow` objects.

```
class TvShowMapper(RowMapper):
    def map_row(self, row, metadata=None):
        return TvShow(title=row[0],
                      airDate=row[1],
                      episodeNumber=row[2],
                      writer=row[3])
```

5. With all this setup, we can now create an instance of `DatabaseTemplate` and use it to execute the same query with a much lower footprint.

```
dt = DatabaseTemplate(connectionFactory)

"""Repeated for each query"""
results = dt.query("""select title, air_date, episode_number,
                      writer from tv_shows where name = %s""",
                   ("Monty Python",), TvShowMapper())
```

This example shows how we can replace 19 lines of code with a single statement using Spring Python's template solution.

Object Relational Mappers (ORMs) have sprung up in response to the low level nature of ANSI SQL's protocol. Many applications have simple object persistence requirements and many of us would prefer working on code, and not database design. By having a tool to help do the schema management work, these ORMs have been a great productivity boost.

But they are not necessarily the answer for every use case. Some queries are very complex and involve looking up information spread between many tables, or involve making complex calculations and involve decoding specific values. Also, many legacy systems are denormalized and don't fit the paradigm that ORMs were originally designed to handle. The complexity of these queries can require working around, or even against, the ORM-based solutions, making them not worth the effort.

To alleviate the frustration of working with SQL, Spring Python's `DatabaseTemplate` greatly simplifies writing SQL, while giving you complete freedom in mapping the results into objects, dictionaries, and tuples. `DatabaseTemplate` can easily augment your application, whether or not you are already using an ORM. That way, simple object persistence can be managed with ORM, while complex queries can be handed over to Spring Python's `DatabaseTemplate`, resulting in a nice blend of productive, functional code.

Other templates, such as `TransactionTemplate`, relieve you of the burden of dealing with the low level idioms needed to code transactions that makes them challenging to incorporate correctly. Later in this book, we will learn how easy it is to add transactions to our code both programmatically and declaratively.

Applying the services you need and abstracting away low level APIs is a key part of the *Spring way* and lets us focus our time and effort on our customer's business requirements instead of our own technical ones.

By using the various components we just looked at, it isn't too hard to develop a simple Pyro service that serves up TV shows from a relational database.

```
from springpython.database.factory import *
from springpython.config import *
from springpython.remoting.pyro import *

class TvShowMapper(RowMapper):
    def map_row(self, row, metadata=None):
        return (title=row[0],
                      airDate=row[1],
                      episodeNumber=row[2],
                      writer=row[3])

class TvShowService(object):
    def __init__(self):
        self.connFactory = MySQLConnectionFactory(username="me",
                                                  password="secret",
                                                hostname="localhost",
                                                  db="springpython")
        self.dt = DatabaseTemplate(connFactory)

    def get_tv_shows(self):
        return dt.query("""select title, air_date, episode_number,
                                  writer
                           from tv_shows where name = %s""",
                           ("Monty Python",), TvShowMapper())
```

```
class TvShowContainer(PythonConfig):
    def __init__(self):
        super(TvShowContainer, self).__init__()

    @Object(lazy_init=True)
    def web_server(self):
        return PyroServiceExporter(service=TvShowService(),
                                   service_name="tvshows",
                                   service_port=9000)

    @Object(lazy_init=True)
    def web_client(self):
        myService = PyroProxyFactory()
        myService.service_url="PYROLOC://localhost:9000/tvshows"
        return myService

if __name__ == "__main__":
    container = ApplicationContext(TvShowContainer())
    container.get_object("web_server")
```

By querying the database for TV shows and serving it up through Pyro, this block of code demonstrates how easy it is to use these powerful modules without mixing them together. It is much easier to maintain software over time when things are kept simple and separated.

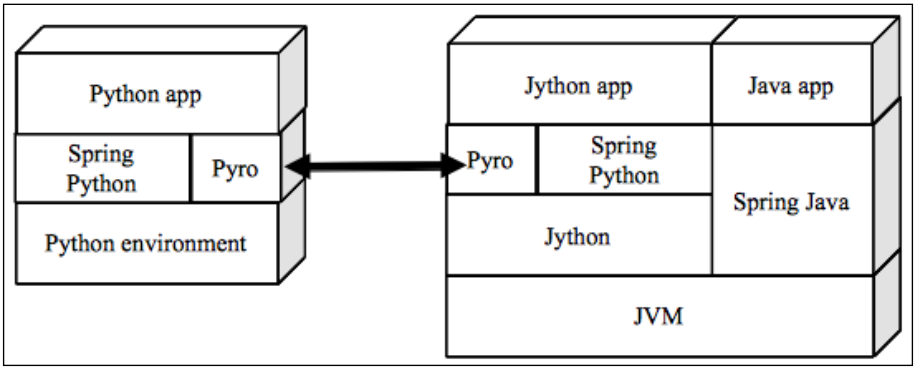
We just took a quick walk through SQL and Pyro and examined their low level APIs. Many low level APIs require a certain sequence of steps to properly utilize them. We just looked at a SQL query. The need for templates also exists with database transactions and LDAP calls. By capturing the flow of the API in a Spring Python template and allowing you to insert your custom code, you can get out of writing plumbing code and instead work on your application's business logic.

Spring Python for Java developers

Java developers often seek ways to increase productivity. With the incredible growth of alternative languages on the Java Virtual Machine (JVM), it is no wonder developers are looking into Jython (`http://www.jython.org`) as an alternative way to write scripts, segments, and subsystems. While it's convenient to use the Java libraries you already know, it can sometimes be rough interacting with the Java idioms of the library.

Spring Python is written in pure Python, which makes it easy to use inside Jython as well. You may find it easier and more appealing to utilize the pythonic APIs of Spring Python rather than the Java idioms of the Spring Framework. The choice ultimately is yours, and it's not an either/or decision; you can use both libraries in both your Java and Python segments.

The following diagram shows a combination of Python, Jython, and pure Java as a stack.



With Spring Python, it's easy to mix Java and Python together. Let's code a simple example, where we write a Jython client that talks to the TV service shown in the previous section. By reusing that code, we can easily code a client. To really show the power of Python/Jython/Java integration, let's expose our Jython client to a pure Java consumer.

1. To begin, we need a pure Java interface that our Jython code can extend.

```
import org.python.core.PyList;

  public interface JavaTvShowService {

    public PyList getTvShows();

}
```

2. Next, we write our Jython class to extend this interface.

```
class JythonTvShowService(JavaTvShowService):
    def __init__(self):
        self.container = ApplicationContext(TvShowContainer())
        self.client = container.get_object("web_client")

    def getTvShows(self):
        return self.client.get_tv_shows()
```

3. For Java clients to call Jython, we need to create a factory that creates an embedded Jython interpreter.

```
import org.python.core.PyObject;
import org.python.util.PythonInterpreter;

public class TvShowServiceFactory {

  private PyObject clazz;

  public TvShowServiceFactory() {
        PythonInterpreter interp  = new PythonInterpreter();
        interp.exec("import JythonTvShowService");
        clazz = interp.get("JythonTvShowService");
}
  public JavaTvShowService createTvShowService() {
    PyObject tvShowService = clazz.__call__();
    return (JavaTvShowService)
    tvShowService.__tojava__(JavaTvShowService.class);
  }
}
```

4. Finally, we can write a main function that invokes the factory first, and then the `JavaTvShowService`.

```
public class Main {

  public static void main(String[] args) {
    TvShowServiceFactory factory =
                    new TvShowServiceFactory();
    JavaTvShowService service =
                  factory.createTvShowService();
    PyList tvShows = service.getTvShows();
    for (int i = 0; i < tvShows.__len__(); i++) {
      PyObject obj = tvShows.__getitem__(i);
      PyTuple tvShow =
             (PyTuple)obj.__tojava__(PyTuple.class);
      System.out.println("Tv show title=" + tvShow.get(0) +
                         " airDate=" + tvShow.get(1) +
                         " episode=" + tvShow.get(2) +
                         " writer=" + tvShow.get(3));
    }
  }
}
```

We defined an interface, providing a good boundary to code both Java consumers and Jython services against. Our Jython class extends the interface and when created, creates an IoC container which requests a `web_client` object. This is our `PyroProxyFactory` that uses Pyro to call our pure Python service over the wire. By implementing the interface, we conveniently expose our remote procedure call to the Java consumers. Our Java app needs a `factory` class to create a Jython interpreter which in turn creates an instance of our Jython class. We use this in our application main to get an instance of our TV service. Using the service, we call the exposed method which makes our Pyro remote call. We iterate over the received list and extract a tuple in order to print out results.

After taking a glance at how to expose a pure Python service to a pure Java consumer, the options from here are endless. We could use Spring Java's `RmiServiceExporter` or `HessianServiceExporter` to expose our Python service to other protocols. This service could be integrated into a Spring Integration workflow or made part of a Spring Batch job. There really is no limit. We will explore this in more detail in *Chapter 10* with our case study.

We have already looked at how easy it is to expose Python services to Java consumers. If you are already familiar with Spring Java, hopefully as you read this book, you will notice that the *Spring way* works not only in Java but in Python as well. Spring Python uses the same concepts as Spring Java, shares some class names with Spring Java, and even has an XML parser that can read Spring Java's format. All of these things increase the ability to rewrite some parts in Python while making small changes in the configuration files. All of these aspects make it easier to incrementally move our application to the sweet spot we need for improved productivity.

Extending Spring Python

Spring Python provides easy access to services, libraries, and APIs through its neatly segmented set of modules. Spring Python's flexibility is due to its modular nature. Providing you with a small set of blocks that are easy to combine is much more reusable than a monolithic set of classes to extend. These techniques open the door to easily integrating with current technologies and future ones as well.

In this chapter, we have already looked at various features, including remoting, `DatabaseTemplate`, and integrating Python with Java. As we explore the other features of Spring Python, I hope you can realize that using the *Spring way* of writing useful abstractions and delegating object creation to the Spring Python container makes it easy to write more modules that aren't part of this project yet.

And as your needs grow, this project will grow as well. Spring Python makes it easy to code your own templates and modules. You can code and inject your custom services just as easily as Spring Python's pre-built ones. If you have ideas for new features that you think belong in Spring Python, you can visit the website at `http://springpython.webfactional.com` to find out how to submit ideas, patches, and perhaps join the team to help grow Spring Python.

Installing Spring Python

Spring Python is easy to install. It comes with two installable tar balls. One is the core library itself. The other is a set of samples you can optionally install to help get a better understanding of Spring Python. As it is written in pure Python, the installed files are the source code as well, so you can see how Spring Python works.

This installation section does NOT show how to install all the parts needed to develop patches for the Spring Python project. For more information on setting up a developer's environment, please join the Spring Python mailing list at `http://lists.springsource.com/listmanager/listinfo/springpython-users` where you can ask about current development requirements.

Setting up an environment for Spring Python

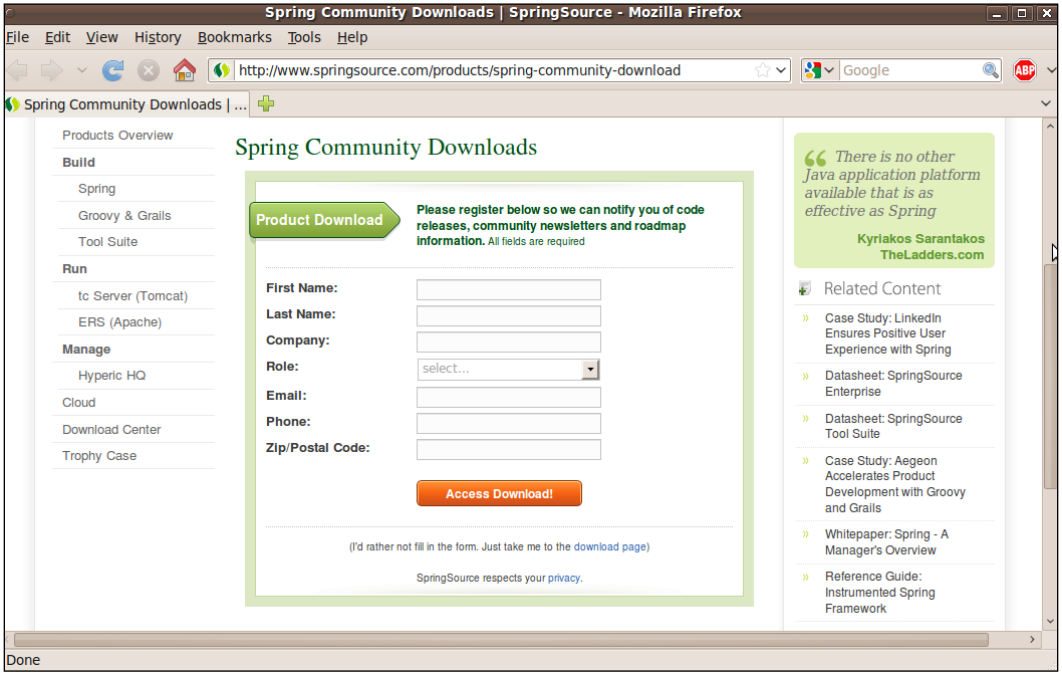
Installing Spring Python currently requires Python 2.4, 2.5, or 2.6. It can also be run on Jython 2.5. At the time of this writing, it hasn't been extended to support Python 3 due to lack of backwards compatibility and immaturity. However, as Python 3 gains acceptance in the user community, Spring Python will move to support it as well.

Spring Python makes it easier to integrate with certain 3rd party libraries. Installation of those 3rd party libraries is not covered in this book. For example, to write queries against an Oracle database, you need to install the **cxora** package.

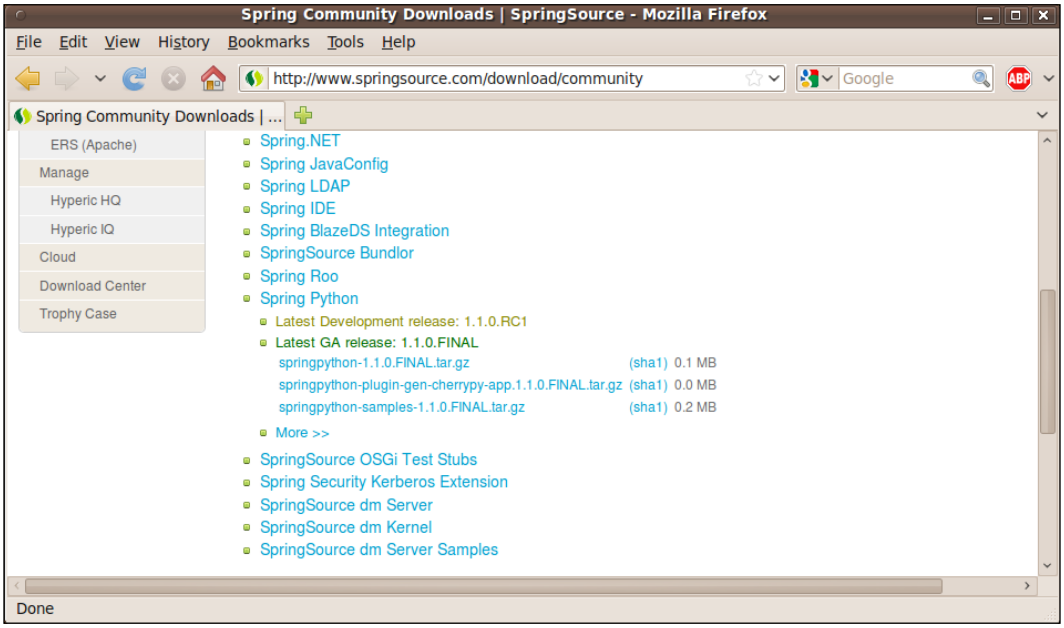
Installing from a pre-built binary download

The binaries are hosted as tar balls by SpringSource, in the community section. They are non-OS specific and should work on any platform.

1. Go to `http://www.springsource.org/download/community`. You can either fill out the registration information, or simply click on the link to take you to the download site



2. Click on **Spring Python** to see the latest version:



3. Download `springpython-<release>.tar.gz` to get the core library
4. Unpack the tarball, and go to the directory containing `setup.py`
5. Type the following command to install Spring Python

Note: This step may require admin privileges!

```
$ python setup.py install
```

6. You should now be able to test the installation.

```
$ python
>>> import springpython
>>> dir()
['__builtins__', '__doc__', '__name__', 'springpython']
```

Installing from source

Spring Python can be installed from source code like many other open source projects. The code is hosted on a subversion repository. The following steps will help you download the code and install it on your machine:

1. Type the following command to checkout the latest trunk repository:

   ```
   $ svn checkout https://src.springframework.org/svn/
   se-springpython-py/trunk/springpython springpython
   ```

   This will create a local directory `springpython` containing the latest changes.

2. Move in to the directory where `build.py` is located. This script serves as the key tool to build, package, and test Spring Python. Type the following command to generate an installable Spring Python package:

   ```
   $ ./build.py -package
   ```

3. Move to `target/artifacts`, the directory containing the newly generated tar balls.

4. Unpack the tar ball `springpython-<release>.tar.gz`, and go to the directory containing `setup.py` and type the following command to install Spring Python:

   Note: This step may require admin privileges.

   ```
   $ python setup.py install
   ```

5. You should now be able to test the installation.

   ```
   $ python
   >>> import springpython
   >>> dir()
   ['__builtins__', '__doc__', '__name__', 'springpython']
   ```

6. Using the subversion repository to fetch source code sets you up to easily install new updates.

7. Type the following command to update your checkout.

   ```
   $ svn update /path/for/springpython
   ```

8. Repeat the steps with `build.py`, `target/artifacts`, and `setup.py`.

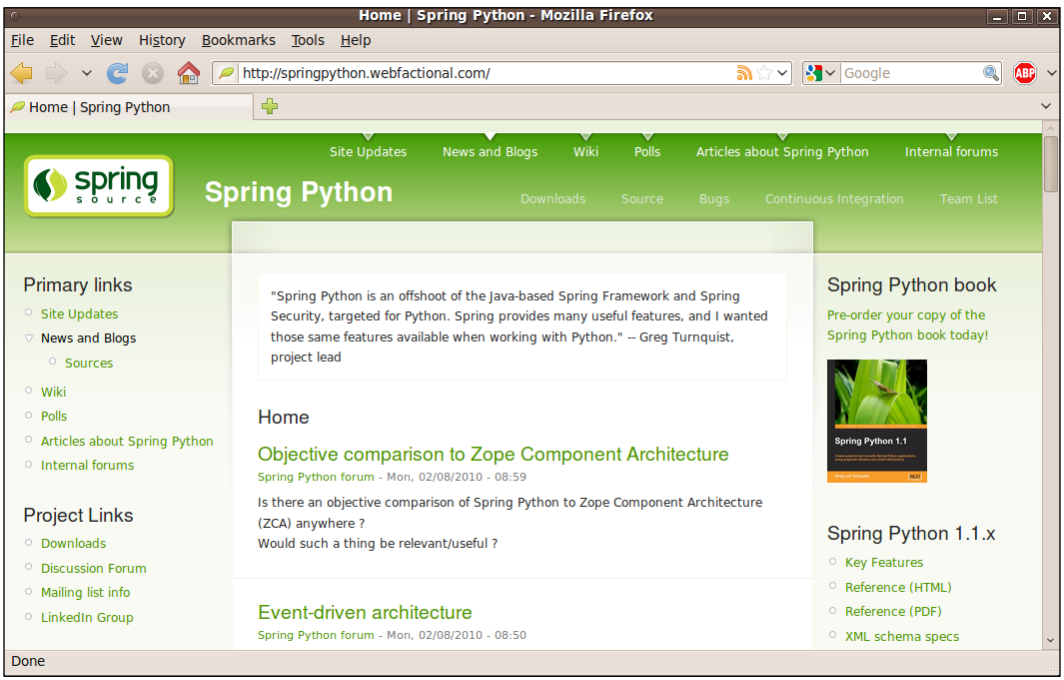
   Official releases are found in the **tags** section of the subversion repository, adjacent to the **trunk**.

You can also visually inspect the code using SpringSource's Fisheye viewer at `https://fisheye.springframework.org/browse/se-springpython-py`. This provides a way to view the code, change sets, and change logs as well.

Spring Python community

Spring Python values its user community, and takes its cue on key features from user requests. Many of the features currently found in Spring Python were added based on user requests, submitted patches, and feedback.

- `http://springpython.webfactional.com`—The project web site, including links to reference documentation



- `http://forum.springsource.org/forumdisplay.php?f=45`—The community discussion forum
- `http://blog.springpython.webfactional.com`—The author's blog site for Spring Python

- `http://lists.springsource.com/archives/springpython-users` — A mailing list for users of Spring Python
- `http://www.linkedin.com/groups?gid=1525237` — A LinkedIn group for Spring Python users

Summary

We have taken a quick tour of the various parts of Spring Python. The examples showed examples of how Spring Python can make things easier for both Python and Java developers.

In this chapter, we learned that:

- Spring Python has a non-invasive API that makes it easy to use other libraries without having to make major changes to your own code base
- Spring Python has a container that decouples object creation from object usage to empower the developer
- Spring Python provides the means to help professional Python developers by offering a non-invasive API to easily access advanced services
- Spring Python offers professional Java developers an easy way to mix Python and Java together through the combination of Python/Jython/Java
- Spring Python can be installed from both binary and source code
- Spring Python is very extensible and easily lets us add new components
- There is a user community where we can post questions, answers, ideas, and patches

In the next chapter, we will go deep into the heart of Spring Python with its `Inversion of Control` container.

2

The Heart of Spring Python—Inversion of Control

Many software teams seek ways to improve productivity without sacrificing quality. This has led to an increase in automated testing and, in turn, sparked an interest in making applications more testable. Many automated test frameworks promote isolating objects by swapping collaborating objects with test doubles to provide canned responses. This generates a need for developers to plugin test doubles with as little change as possible.

On some occasions, managers have trimmed budgets by cutting back on development and integration hardware. This leaves developers with the quandary of having to develop on a smaller footprint than their production environment. Software developers need the ability to switch between different test configurations and deployment scenarios.

Software products sold to customers often need the ability to make flexible adjustments on site, whether its through a company representative or from the customer.

In this chapter, we will explore how Spring Python's **Inversion of Control (IoC)** container meets all these needs by making it possible to move the creation of critical objects into a container definition, allowing flexible adjustment without impacting the core business code.

This chapter will cover:

- How IoC containers make it easy to isolate objects under test in order to work with automated testing.
- A detailed explanation of IoC with high level diagrams. We will learn how Dependency Injection is a type of IoC and uses the Hollywood principle of *Don't call us, we'll call you!*
- Spring Python's IoC container, which handles dependency resolution, setting properties, and lazy initialiation. It also provides extension points for developers to utilize as needed.
- Spring Python's answer to the community debate of IoC in dynamic languages.
- How mixing Python and Java components together is easy, can be done many ways, and provides a good way for developers to choose what best fits their needs.

Swapping production code with test doubles

As developers, we need the ability to exchange production objects with mocks and stubs.

Mocks and stubs are two ways of generating canned responses used to test our code. **Mocks** are used to generate canned method calls. **Stubs** are used to generate canned data. Both of these are useful tools to simulate external components our code must work with. For more details, see `http://martinfowler.com/articles/mocksArentStubs.html`

In this example, we are going to explore how an IoC container makes it easy to exchange production objects with mock instances. We start by developing a simple service along with some automated testing for a wiki engine.

1. First of all, let's define a simple service for our wiki engine. The following `WikiService` has a function that looks into a MySQL database and returns the number of hits for a page as well as the ratio of reads per edit.

```
class WikiService(object):

    def __init_(self):
        self.data_access = MySqlDataAccess()
```

```
    def statistics(self, page_name):
    """Return tuple containg (num hits, hits per edit)"""
    hits = self.data_access.hits(page_name)
    return (hits, hits / len(self.data_access.edits(page_name)))
```

In this situation, `WikiService` directly defines `data_access` to use `MySqlDataAccess`. The `statistics` method calls into `MySqlDataAccess` to fetch some information, and returns a tuple containing the number of hits against the page as well as the ratio of reads per edit.

It is important to point out that our current version of `WikiService` has a strong dependency on MySQL as implied by directly creating an instance of `MySqlDataAccess`.

2. Next, we will write some startup code used to run our wiki web server.

```
if __name__ == "__main__":
    service = WikiService()
    WikiWebApp(service).run()
```

The startup code creates an instance of `WikiService`, which in turn creates an instance of `MySqlDataAccess`. We create an instance of our main web application component, `WikiWebApp`, and start it up, giving it a handle on our service.

We have not defined `WikiWebApp` or `MySqlDataAccess`. Instead, we will be focusing on the functionality `WikiService` provides, and on how to isolate and test it.

Let's look more closely at testing the code we've written. A good test case would involve exercising `statistics`. Considering that it uses `hits` and `edits` from `MySqlDataAccess`, it is directly dependent on connecting to a live MySQL database. One option for testing would be pre-loading the database with fixed content. However, the cost of set up and tear down can quickly scale out of control. If you have multiple developers on your team, you also don't want the contention where one developer is setting up while another is tearing down the same tables.

3. Since we do not want to continuously setup and tear down live database tables, we are going to code a stubbed out version of the data access object instead.

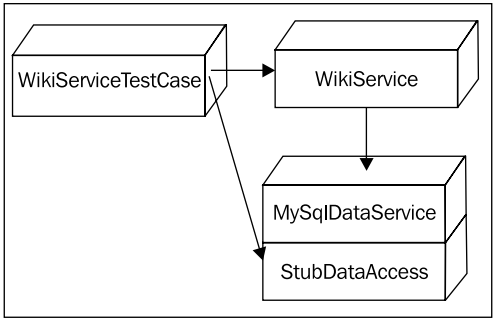
```
class StubDataAccess(object):
    def hits(self):
        return 10.0

    def edits(self, page_name):
        return 2.0
```

Notice how our stub version returns canned values that are easy to test against.

4. Now let's write a test case that exercises our `WikiService`. In order to replace the `data_access` attribute with a test double, the test case must directly override the attribute.

```
class WikiServiceTestCase(unittest.TestCase):
    def testHittingWikiService(self):
        service = WikiService()
        service.data_access = StubDataAccess()

        results = service.statistics("stub page")
        self.assertEquals(10.0, results[0])
        self.assertEquals(5.0, results[1])
```

This solution nicely removes the need to deal with the MySQL database by plugging in a stubbed out version of our data access layer. These dependencies are depicted in the following diagram.



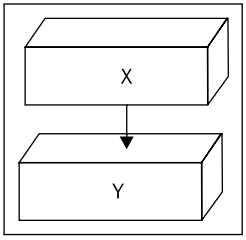
`WikiServiceTestCase` depends on `WikiService` and `StubDataAccess`, as well as an inherited dependency to `MySqlDataAccess`. Any change to any of these dependencies could impact our test code.

If we build a huge test suite involving hundreds of test methods, all using this pattern of instantiating `WikiService` and then overriding `data_access` with a test double, we have set ourselves up with a big risk. For example, `WikiService` or `StubDataAccess` could have some new initializing arguments added. If we later need to change something about this pattern of creating our testable `WikiService`, we may have to update every test method! We would need to make every change right. This is where Spring Python's Inversion of Control can help.

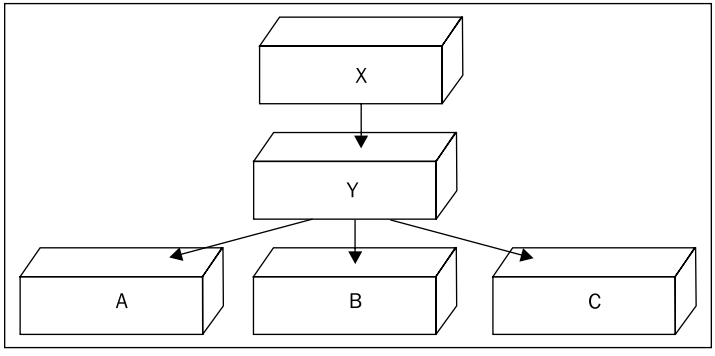
More about Inversion of Control

Before diving into our solution, let's look a little deeper into the meaning of Inversion of Control.

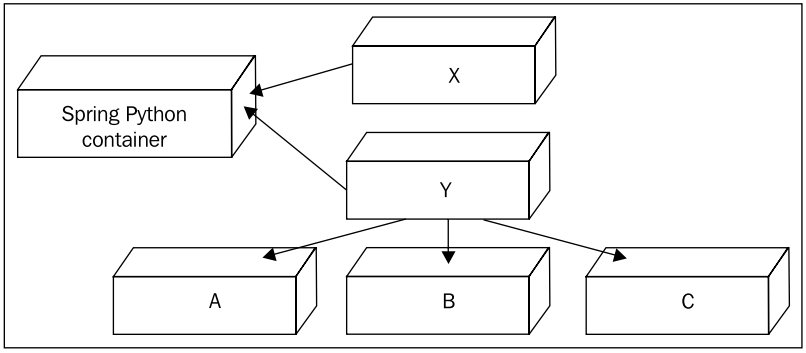
Inversion of Control is a paradigm where we alter the way we create objects. In simple object creation scenarios, if we have modules X and Y, and X needs an instance of Y, we would let X directly create it. This introduces a direct dependency between X and Y, as shown in the following diagram.



It's dependent because any changes to Y may impact X and incur required updates. This dependency isn't just between X and Y. X is now dependent on Y *and all of Y's dependencies* (as shown below). Any updates to Y or any of its dependencies run the risk of impacting X.



With Inversion of Control, we break this potential risk of impacts from Y and its dependencies to X by delegating creation of instances of Y to a separate container, as shown below.



This shifts the dependency between X and Y over to the container, reducing the coupling.

It's important to note that the dependency hasn't been entirely eliminated. Because X is still given an instance of Y, there is still some dependency on its API, but the fact that Y and none of its dependencies have to be imported into X makes X a smaller, more manageable block of code.

We saw in our wiki engine code how `WikiService` was dependent on `MySqlDataAccess`. `MySqlDataAccess` is dependent on MySQLdb, a python library for communicating with MySQL. Using the container and coding against a well defined interface opens up the opportunity to change what version of data access is being injected into `WikiService`.

As we continue our example in the next section, we'll see how IoC can help us make management of test code easier, reducing long term maintenance costs.

Adding Inversion of Control to our application

We already took the first step in reducing maintenance costs by eliminating our dependency on MySQL by using a test stub. However the mechanism we used incurred a great risk due to violation of the **DRY** (**Don't Repeat Yourself**) principle.

For our current problem, we want Spring Python to manage the creation of `WikiService` in a way that allows us to make changes in one place, so we don't have to edit every test method. To do this, we will define an IoC container and let it handle creating the objects for us. The following code shows a simple container definition.

1. First of all, let's create an IoC container using Spring Python and have it create our instance of `WikiService`.

```
from springpython.config import PythonConfig
from springpython.config import Object

class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()

    @Object
    def wiki_service(self):
        return WikiService()
```

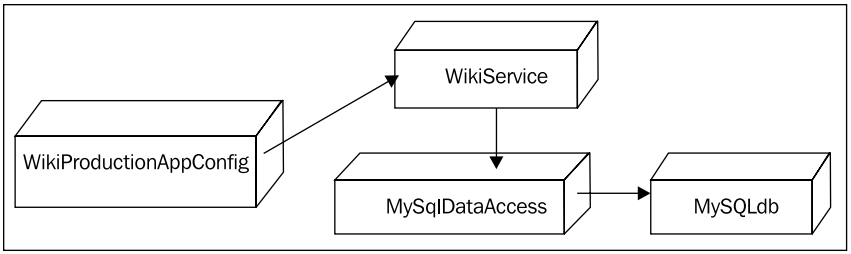
   You can spot the objects defined in the container by noting Spring Python's `@Object` decorator.

2. Now we want to change our startup code so that it uses the container instead of creating `WikiService` object directly.

```
if __name__ == "__main__":
    from springpython.context import ApplicationContext
    container = ApplicationContext(WikiProductionAppConfig())
    service = container.get_object("wiki_service")
    WikiWebApp(service).run()
```

   In this version of our wiki web application, the service object was obtained by asking the container for `wiki_service`. Spring Python dispatches this request to `WikiProductionAppConfig` where it invokes the `wiki_service()` method. It is now up to the container to create an instance of `WikiService` and return it back to us.

The dependencies are shown in the following diagram:



At this intermediate stage, please note that `WikiService` is still dependent on `MySqlDataAccess`. We have only modified how our application gets a copy of `WikiService`. In later steps, we will completely remove this dependency to `MySqlDataAccess`.

3. Let's see what our test case looks like when we use the IoC container to retrieve `WikiService`.

```
from springpython.context import ApplicationContext

class WikiServiceTestCase(unittest.TestCase):
    def testHittingWikiService(self):
        container = ApplicationContext(WikiProductionAppConfig())
        service = container.get_object("wiki_service")
        service.data_access = StubDataAccess()

        results = service.statistics("stub page")
        self.assertEquals(10.0, results[0])
        self.assertEquals(5.0, results[1])
```

With this change, we are now getting `WikiService` from the container, and then overriding it with the `StubDataAccess`. You may wonder "what was the point of that?" The key point is that we *shifted creation of our testable wiki_service object to the container*. To complete the transition, we need to remove all dependency of `MySqlDataAccess` from `WikiService`. Before we do that, let's discuss Dependency Injection.

Dependency Injection a.k.a. the Hollywood principle

So far, we have managed to delegate creation of our `WikiService` object to Spring Python's Inversion of Control container. However, `WikiService` still has a hard-coded dependency to `MySqlDataAccess`.

The nature of IoC is to push object creation into a 3rd party location. Up to this point, we have been using the term *Inversion of Control*. The way that Spring Python implements IoC, is through the mechanism of **Dependency Injection**. Dependency Injection, or DI, is where dependencies are pushed into objects through either initialization code or by letting the container directly assign attributes. This is sometimes described by the Hollywood cliché of *Don't call us, we'll call you*. It means that the object which needs a certain dependency shouldn't make it directly, but instead wait on the external container to provide it when needed.

Because Spring Python only using Dependency Injection, the terms, Inversion of Control, IoC, Dependency Injection, and DI may be used interchangeably throughout this book to communicate the same idea.

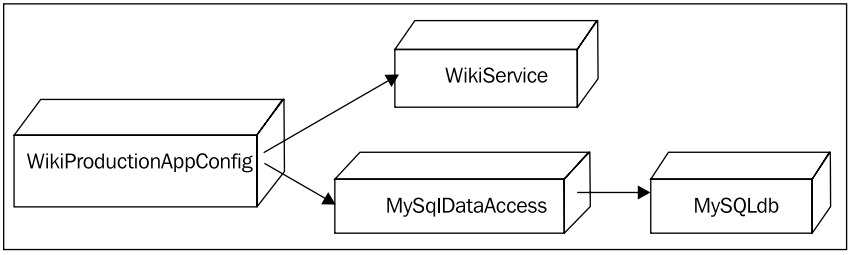
The following version of `WikiService` shows a complete removal of dependence on `MySqlDataAccess`.

```
class WikiService(object):
    def __init__(self, data_access):
        self.data_access = data_access

    def statistics(self, page_name):
        """Return tuple containing (num hits, hits per edit)"""
        hits = self.data_access.hits(page_name)
        return (hits, hits / len(self.data_access.edits(page_name)))
```

With this altered version of `WikiService`, it is now up to `WikiService`'s creator to provide the concrete instance of `data_access`. The corresponding change to make to our IoC container looks like this.

```
from springpython.config import PythonConfig
from springpython.config import Object

class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()

    @Object
    def data_access(self):
        return MySqlDataAccess()

    @Object
    def wiki_service(self):
        return WikiService(self.data_access())
```

We have added a definition for `MySqlDataAccess` so that we can *inject* this into the instance of `WikiService` when it's created. Our dependency diagram now shows a complete break of dependency between `WikiService` and `MySqlDataAccess`.



As described earlier, this is akin to the Hollywood mantra *Don't call us, we'll call you*. It opens up our code to the possibility of having any variation injected, giving us greater flexibility, without having to rewrite other parts of the system.

Adding Inversion of Control to our test

With this adjustment to `WikiService` and `WikiProductionAppConfig`, we can now code an alternative way of setting up our test case.

1. First, we subclass the production container configuration to create a test version. Then we override the `data_access` object so that it returns our stub alternative.

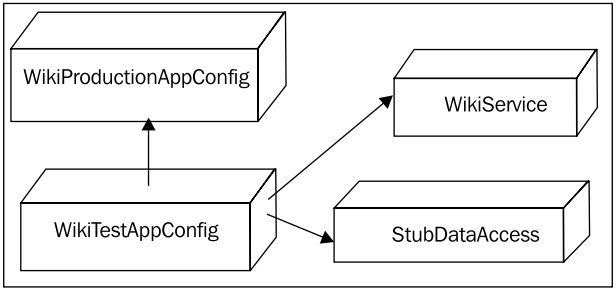
```
class WikiTestAppConfig(WikiProductionAppConfig):
    def __init__(self):
        super(WikiTestAppConfig, self).__init__()

    @Object
    def data_access(self):
        return StubDataAccess()
```

   Using this technique, we inherit all the production definitions. Then we simply override the parts needed for our test situation. In this case, we return a slightly modified version of `WikiService` that is suitable for our testing needs.

2. Now, we can alter our test suite to fetch `WikiService` just like the production main code, except using our alternative container.

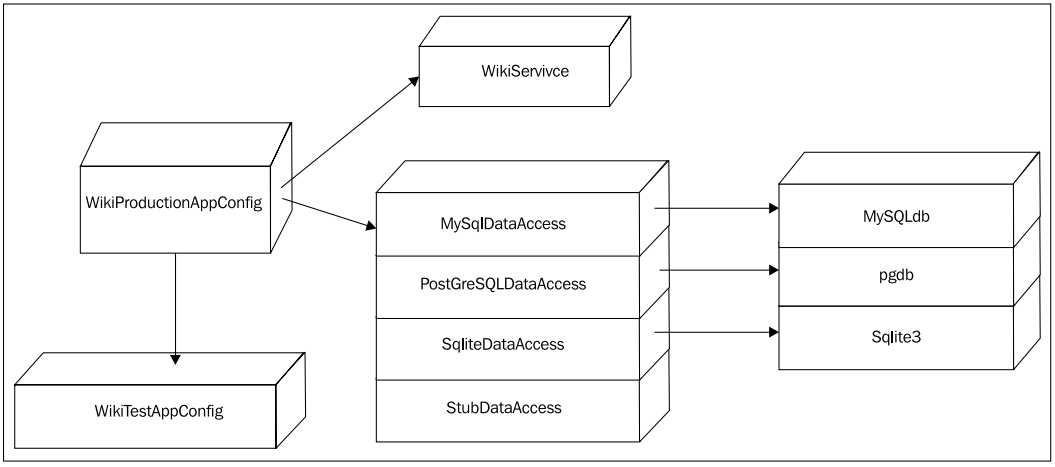
```
from springpython.context import ApplicationContext

class WikiServiceTestCase(unittest.TestCase):
    def testHittingWikiService(self):
        container = ApplicationContext(WikiTestAppConfig())
        service = container.get_object("wiki_service")

        results = service.statistics("stub page")
        self.assertEquals(10.0, results[0])
        self.assertEquals(5.0, results[1])
```

The dependencies (or lack of!) between `WikiService` and `StubDataAccess` are shown in following diagram:



This makes it easy to plug in our alternative `StubDataAccess`. An important thing to note is how our test code is no longer overriding the `data_access` attribute, or doing any other special steps when creating `WikiService`. Instead, this creation logic has been totally turned over to the IoC container. We can now easily write as many tests as we want with no risk of changes to `WikiService` or `data_access`, provided we continue to rely on the IoC container to handle object creation. We just visit the blueprints of `WikiTestAppConfig` to make any future alterations.

In conclusion of this example, the following diagram shows our current objects as well as potential enhancement to our system. In this possible situation, we have defined multiple data access components. To find out which one is being injected into `WikiService`, we simply look at the relevant container's definition, whether its production or a particular test scenario. Our current system may use **MySQL**, but if a new customer wanted to use **PostGreSQL**, we simply create another variation of our container, and inject the alternate data access object. The same can be said for **Sqlite**. And all of this can be done with no impact to `WikiService`.



Container versus Context

You may have noticed in this chapter's example how we keep referring to the code that contains our object definitions as a container, and yet, the class used to create it is called `ApplicationContext`. This chapter will clarify the differences between a **context** and a **container**.

Spring Python has a base class called `ObjectContainer`, which is responsible for managing the object definitions, creating instances of objects based on these definitions, and is largely responsible for the functionality that we have looked at so far. Any instance of this class would be a container in object oriented terms. In fact, we could substitute that class in our previous example everywhere we see `ApplicationContext`, and it would act mostly the same.

`ObjectContainer` has a subclass called `ApplicationContext`. An instance of `ApplicationContext` is a context. However, from an OOP perspective, a context is a container. So generally referring to instances of either one as an IoC container is common practice.

An `ApplicationContext` has some differences in behavior from an `ObjectContainer` as well as extra features involving life cycle management and object management. These differences will first be shown in the following table of features and will be covered in more detail later.

Class	Features
`ObjectContainer`	• Lazily instantiates objects on startup with option to override • Objects are scoped singleton by default
`ApplicationContext`	• Eagerly instantiates objects on startup with option to override • Objects are scoped singleton by default • Supports property driven objects • Supports post processors • Supports context-aware objects

Lazy objects

Earlier I said that most of the behavior in this chapter's example would be the same if we had replaced `ApplicationContext` with `ObjectContainer`. One difference is that `ObjectContainer` doesn't instantiate any objects when the container is first created. Instead, it waits until the object is requested to actually create it. `ApplicationContext` automatically instantiates all objects immediately when the container is started.

In the examples so far, this difference would have little effect, because right after creating a container, we request the principle object. This won't be the case with some of the examples later in this book.

It is possible to override this by setting `lazy_init` to `True` in the object's definition. This will delay object creation until first request.

```
class WikiTestAppConfig(WikiProductionAppConfig):
    def __init__(self):
        super(WikiTestAppConfig, self).__init__()

    @Object(lazy_init=True)
    def data_access(self):
        return StubDataAccess()
```

One reason for using this is if we had some object that was only needed on certain occasions and could incur a lot of overhead when created. Setting `lazy_init` to `True` would defer its creation, making it behave as an on-demand service.

This override works for both `ObjectContainer` and `ApplicationContext`.

Scoped objects

Another key duty of the container is to also manage the scope of objects. This means at what time that objects are created, where the instances are stored, and how long before they are destroyed.

Spring Python currently supports two scopes: **singleton** and **prototype**.
A singleton-scoped object is cached in the container until shutdown. A prototype-scoped object is never stored, thus requiring the object factory to create a new instance every time the object is requested from the container.

When a singleton object includes a prototype object as one of its initializing properties, it is important to realize that the prototype isn't recreated every time the cached singleton is used.

The default policy for the container is to make everything singleton. The scope for each object can be individually overridden as shown below.

```
class WikiTestAppConfig(WikiProductionAppConfig):
    def __init__(self):
        super(WikiTestAppConfig, self).__init__()

    @Object(scope=scope.PROTOTYPE)
    def data_access(self):
        return StubDataAccess()
```

Each request for `data_access` will yield a different instance. This happens whether the request is from outside the container, or from another container object injecting `data_access`.

This override works for both `ObjectContainer` and `ApplicationContext`.

Property driven objects

`ApplicationContext` will invoke `after_properties_set()` on any object that has this method, after it has been created and all container-defined properties have been set. Here are some examples of how this can be useful:

- Including validation logic in class definitions. If some properties are optional and others are not (or certain combinations of properties need to be set), this is our opportunity to define it, and let the container validate that an object was defined correctly.
- Starting up background services. For example, `PyroServiceExporter` launches a daemon thread in the background after all properties are set.

Post processor objects

`ApplicationContext` will search for **post processors**. These are classes that manipulate other objects in the container. They extend Spring Python's `ObjectPostProcessor` class.

```
class ObjectPostProcessor(object):
    def post_process_before_initialization(self, obj, obj_name):
        return obj
    def post_process_after_initialization(self, obj, obj_name):
        return obj
```

When the context starts up, all non-post processors are fed one-at-a-time to each post processor twice: before the initialization and after. By default, the object is passed through with no change. By coding a custom post processor, we can override either stage, and apply any changes we wish. This can include altering the object itself, or substituting a different object.

```
class RemoteService(ObjectPostProcessor):
    def post_process_after_initialization(self, obj, obj_name):
        if obj_name.endswith("Service"):
            return PyroServiceExporter(service=obj,
                                       service_name=obj_name,
                                       service_port=9000)
        else:
            return obj
```

This post processor looks at an object's name, and if its name ends with `Service`, returns a proxy that exports the object as a Pyro service, using the object's name as part of the URL. Otherwise, it simply returns the object.

This example shows how we can easily develop a convention-over-configuration client-server mechanism, simply by using Spring Python's IoC container and some custom post processors.

Context aware objects

We can define classes that get a copy of the `ApplicationContext` injected into the attribute `app_context` by extending `ApplicationContextAware`.

This is *only* recommended for objects that need to manipulate the context in some fashion. It is *not* a good practice to use this hook to simply fetch objects, because it ties our code directly to the container.

Instead, use the principles of Dependency Injection as shown earlier in this chapter.

Debate about IoC in dynamic languages

There is a powerful debate in the technical community about whether or not IoC is of any value in dynamic languages such as Python, Ruby, and others. This section addresses several of the arguments, and shows how IoC definitely has value in any language. If you are already sold on the idea of IoC, you can skip this section and learn about more options to configure IoC definitions provided by Spring Python.

One argument is that IoC was invented as a paradigm to facilitate static languages, particularly Java. In the dynamic world, where passing around objects with handles is much more fluid, it is understandable that IoC may not solve as many problems as it does for static languages such as Java. While it is true that many Java developers that engaged in using IoC frameworks saw considerable leverage in productivity, it is also credible to not leave dynamic languages out of this.

The strongest principle that transcends the dynamic/static divide and is handled by IoC is **DRY** (**Don't Repeat Yourself**). By defining the creation of an object in one place, and being able to use that definition repeatedly is very useful, and makes it easy to adjust how the object is created.

One keen Ruby developer went through several phases of developing a Ruby IoC solution. The first round involved static configuration files very similar to the XML option. Later on, he realized that coding a Ruby DSL was simpler, easier to read, and more of the Ruby way. Finally, he realized that using Ruby directly in his code without the framework was actually easier and more salient. In finishing his blog article, he alludes to the fact that he uses DI every day. He just doesn't need a framework to do it.

In response, I am the first person to say that not every object created in a system needs to be served up by an IoC container; just the critical ones that would benefit from being swapped with test doubles, environmental changes, and wiring up different configurations for different customers. For example, coding small scripts and simple apps may actually not require any IoC at all. However, as our apps grow, the need to manage dependencies and object creation grows, and in those situations, Spring Python's IoC container can meet those needs. IoC also provides easy solutions for situations that need remoting, transactions, security, and other.

Opponents to dynamic IoC solutions have suggested that these solutions are simple port jobs and are really more for people not familiar with the dynamic language environment. In essence, people that like IoC when coding statically with C# or Java, want the same warm comforts in Python. While Spring Python has utilized some of the same class names as Spring Java, it is by no means a simple port job. Spring Python has ported many of the concepts but not the code. One of the best examples of this is Spring Python's security module. The architecture of Spring Security (originally known as Acegi Security) has become a de facto security standard in the Java industry. Their architectural concepts involve using a common set of interfaces to link into countless security systems such as database, X.509, LDAP, CAS, kerberos, OpenID, OpenSSO, and more, without having to get tied down to a specific API. Instead, the concepts of authentication and authorization have been abstracted to the right level, such that it is easy to add security after a system has been developed without major rework. The credit for this strongly resides in the power of IoC and its ability to wrap this critical service around already written business logic. It would be foolish to pass up this powerful security architecture on the basis of simple static versus. dynamic arguments.

Another argument tends to be based on Spring Java and its historical use of XML for configuration files. While XML is commonly used for many Java-based frameworks, the Python community tends to avoid XML unless customer requirements dictate otherwise. Some developers have criticized Spring Python for being an XML-based container. However, this book has already shown pure Python container definitions. Developers don't have to use XML at all to harness the power of Inversion of Control. By using Python directly for container definitions, the following powerful options are available:

- We can code a switchable context where the container reads the hostname and deduces whether it is running in development, test, or production environment
- We can code a clustered context where we vary object configurations based on which side of a cluster we are running
- We can write code that reads extra properties from an external source, like an Apache web server's `httpd.conf` file, allowing us to stay nicely integrated
- We can look up object properties in a database

Frankly, the options are limitless, given the power of Python and the added utilities Spring Python provides

Migrating a Spring Java application to Python

In the previous section, I made the point that Spring Python doesn't require us to use XML. However, there is support for XML along with other formats we haven't looked at yet. Other formats are provided to make it easier for Spring Java developers to migrate to Python using what they already know.

Spring Python has two file parsers that read XML files containing object definitions. `SpringJavaConfig` reads Spring Java XML configuration files, and `XMLConfig` reads a format similar to that, but uniquely defined to offer Python-oriented options, such as support for dictionaries, tuples, sets, and frozen sets.

`SpringJavaConfig` does not support all the features found in Spring Java's XML specifications, including the specialized namespaces, Spring Expression Language, nor properties. At this point in time, it is meant to be convenient tool to demonstrate simple integration between Python and Java.

Let's explore the steps taken if our wiki engine software had originally been written in Java, and we are exploring the option to migrate to Python. To make this as seamless as possible, we will run the Python bits in Jython.

Jython is a Python compiler written in Java and designed to run on the JVM. Jython scripts can access both Python and Java-based libraries. However, Jython can not access Python extensions coded in C. Spring Python 1.1 runs on Jython 2.5.1. Information about download, installation, and other documentation of Jython can be found at `http://jython.org`.

1. Let's assume that the Java code has a well defined interface for data access.

```
public interface DataAccess {

  public int hits(String pageName);

  public int edits(String pageName);
}
```

2. For demonstration purposes, let's create this concrete Java implementation of that interface.

```
public class MySqlDataAccess implements DataAccess {

  private JdbcTemplate jt;

  public MySqlDataAccess(DataSource ds) {
    this.jt = new JdbcTemplate(ds);
  }

  public int hits(String pageName) {
    return jt.queryForInt(
        "select HITS from METRICS " +
        "where PAGE = ?", pageName);
  }

  public int edits(String pageName) {
    return jt.queryForInt(
        "select count(*) from VERSIONS " +
        "where PAGE = ?", pageName);
  }
}
```

This Java code may appear contrived, considering `MySqlDataAccess` is supposed to be dependent on MySQL. It is for demonstration purposes that this over simplified version was created.

3. Finally, let's look at a Java version of `WikiService`.

```
public class WikiService {

  private DataAccess dataAccess;

  public DataAccess getDataAccess() {
    return this.dataAccess;
  }

  public void setDataAccess(DataAccess dataAccess) {
    this.dataAccess = dataAccess;
  }

  public double[] statistics(String pageName) {
    double hits = dataAccess.hits(pageName);
    double ratio = hits / dataAccess.edits(pageName);
    return new double[]{hits, ratio};
  }
}
```

4. A common way this would have been wired with Spring Java can be found in `javaBeans.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans
           http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd">

  <bean id="dataAccess" class="MySqlDataAccess"/>

  <bean id="wikiService" class="WikiService">
    <property name="dataAccess" ref="dataAccess"/>
  </bean>
</beans>
```

Note that this XML format is defined and managed by Spring Java, *not* Spring Python.

5. Now that we can see all the parts of this pure Java application, the next step is to start it up using Spring Python and Jython.

```
if __name__ == "__main__":
    from springpython.context import ApplicationContext
    from springpython.config import SpringJavaConfig
    ctx = ApplicationContext(SpringJavaConfig("javaBeans.xml"))
    service = ctx.get_object("wikiService")
    service.calculateWikiStats()
```

   Thanks to Jython's ability to transparently create both Python and Java objects, our Spring Python container can parse the object definitions in `javaBeans.xml`.

6. As our development cycle continues, we find time to port `MySqlDataAccess` to Python.

7. After doing so, we decide that we want to take advantage of more of Python's features, like tuples, sets, and frozen sets. The Spring Java format doesn't support these. Let's convert `javaBeans.xml` to Spring Python's XML format and save it in `production.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/
schema/objects"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/
springpython/schema/objects
           http://springpython.webfactional.com/schema/context/
spring-python-context-1.0.xsd">

  <object id="data_access" class="Py.MySqlDataAccess"/>
  <object id="wiki_service" class="WikiService">
    <property name="dataAccess" ref="data_access"/>
  </object>
</objects>
```

This is very similar to the Spring Java format, and that's the point. By changing the header and replacing 'bean' with 'object', we can now use `XMLConfig`.

```
if __name__ == "__main__":
    from springpython.context import ApplicationContext
    from springpython.config import XMLConfig
    ctx = ApplicationContext(XMLConfig("production.xml"))
    service = ctx.get_object("wiki_service")
    service.calculateWikiStats()
```

We now have access to more options such as the following:

```
<property name="some_set">
  <set>
    <value>Hello, world!</value>
    <ref object="SingletonString"/>
    <value>Spring Python</value>
  </set>
</property>

<property name="some_frozen_set">
  <frozenset>
    <value>Hello, world!</value>
    <ref object="SingletonString"/>
    <value>Spring Python</value>
  </frozenset>
</property>

<property name="some_tuple">
  <tuple>
    <value>Hello, world!</value>
    <ref object="SingletonString"/>
    <value>Spring Python</value>
  </tuple>
</property>
```

8. Having migrated this much, we decide to start coding some tests. We want to use the style shown at the beginning of this chapter where we swap out `MySqlDataAccess` with `StubDataAccess` using an alternative configuration. To avoid changing `production.xml`, we create an alternate configuration file `test.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/
schema/objects"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/
springpython/schema/objects
                http://springpython.webfactional.com/schema/
context/spring-python-context-1.0.xsd">

        <object id="data_access" class="Py.StubDataAccess"/>
</objects>
```

9. Now let's code a test case using these two files for a specialized container. To do that, we give `ApplicationContext` with a list of configurations.

```
import unittest
from springpython.context import ApplicationContext
from springpython.config import XMLConfig

class WikiServiceTestCase(unittest.TestCase):
    def testHittingWikiService(self):
        ctx = ApplicationContext([XMLConfig("production.xml"),
                                  XMLConfig("test.xml")])
        service = ctx.get_object("wiki_service")
        results = service.statistics("stub page")
        self.assertEquals(10.0, results[0])
        self.assertEquals(2.0, results[1])
```

The order of filenames is important. First, a list of definitions is read from `production.xml`. Next, more definitions are read from `test.xml`. Any new definitions with the same name as previous ones will overwrite the previous definitions. In our case, we overwrite `data_access` with a reference to our stub object.

10. After all the success we have had with the Python platform, we decide to migrate the container definitions to pure Python. First, let's convert `production.xml` to `WikiProductionAppConfig`.

```
from springpython.config import PythonConfig, Object
from Py import MySqlDataAccess
import WikiService

class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def data_access(self):
        return MySqlDataAccess()

    @Object
    def wiki_service(self):
        results = WikiService()
        results.dataAccess = self.data_access()
        return results
```

11. Let's mix this with `test.xml`.

```
import unittest
from springpython.context import ApplicationContext
from springpython.config import XMLConfig

class WikiServiceTestCase(unittest.TestCase):
    def testHittingWikiService(self):
        ctx = ApplicationContext([WikiProductionAppConfig(),
                                  XMLConfig("test.xml")])
        service = ctx.get_object("wiki_service")
        results = service.statistics("stub page")
        self.assertEquals(10.0, results[0])
        self.assertEquals(2.0, results[1])
```

We simply replace `XMLConfig("production.xml")` with `WikiProductionAppConfig()`. `ApplicationContext` comfortably accepts any combination of definitions stored in different formats. Spring Python's flexible container gives us a fine grained control over how we store all our definitions.

We could continue and easily replace `test.xml` with something like `WikiTestAppConfig` written in pure Python.

This demonstrates how easy it is to convert things a piece at a time. We could have stopped earlier based on what we wanted to keep in Java and what we wanted to move to Python. If we move the rest of the Java parts to Python solutions, then we may opt to use either Python or Jython. The point is that XML support makes it easy to move from Java to Python with minimal impact.

It is important to realize that just because we were starting with a Spring Java application doesn't mean this technique is only for Spring-based Java projects. In the previous example, we quickly moved past `javaBeans.xml` and wired the Java components using Spring Python's XML format.

Summary

In this chapter we explored swapping test doubles using Inversion of Control (IoC). This is not the only use case for IoC. Being able to easily reconfigure things with no impact to the core business logic is of incredible value.

IoC allows us to move object creation to a centralized location. This empowers our applications to be easier to inject alternatives. This makes things such as testing, remoting, transactions, and security much easier.

Spring Python supports many formats including pure Python, Spring Python XML, and Spring Java XML. This gives users a wide range of choices based on their needs. By working with Jython, it is much easier to mix together Python and Java code together.

Exchanging one object definition with another constitutes a useful service. Throughout this book, other services will be explored. Just remember, at the heart of those services is Spring Python's IoC container.

In this chapter, we learned:

- IoC containers make it easy to isolate objects under test by swapping production collaborators with test doubles
- Detailed definitions of IoC and Dependency Injection
- IoC containers positively impact our application by separating creation of objects from their usage, and this helps modularize our applications
- The Hollywood principle of Dependency Injection means that our objects don't create their collaborators, but instead wait for the container to create them and inject them into our object

- Spring Python has an `ObjectContainer` with the basic parts of an IoC container. This container handles dependency resolution, setting properties, and lazy initialiation
- Spring Python has an `ApplicationContext`, that subclasses the container, and provides extension points for developers to manipulate creation of objects
- While there may be debate about using IoC in dynamic languages, the benefits of delegating object creation to a separate container is not confined to just static languages
- By using Spring Python's container with Jython, it is easy to mix Python and Java components together. This tactic is even more productive for applications that were already built with Spring Java

In the next chapter, we will look at adding services to existing code using **Aspect Oriented Programming (AOP)**.

3
Adding Services to APIs

In the previous chapter, we took a look at Spring Python's Inversion of Control container, and used it to exchange production objects with test doubles. One of the most prolific paradigms used today is **Object Oriented Programming (OOP)** which uses classes. Classes serve as representations of the domain and business problems that we solve. But this isn't enough!

We also need robust features such as caching, storage and retrieval, transactions, security, and logging. These services are orthogonal to the business problems that we must solve.

Mixing classes with these orthogonal services can quickly become costly to code correctly as well as maintain. **Aspect Oriented Programming (AOP)** solves these orthongal problems through a new type of modularization called an **aspect**.

In this chapter, we will learn:

- AOP from 10,000 feet
- Adding caching to Spring Python objects
- Extending the example by adding performance and security advisors
- How AOP is a paradigm and not just a library
- The distinct features of Spring Python's AOP, compared to other libraries as well as Python language features
- The risks of AOP and how to mitigate them by showing how to test our aspects
- Some advice about applying `Advice`

AOP from 10,000 feet

Throughout this chapter, we will explore examples of AOP and how to use them. To do that, it's important that we learn some basic definitions.

Crosscutting problems impact more than one hierarchy of classes. AOP efficiently solves these problems.

Commonly cited examples of crosscutting problems include security, transactions, tracing, auditing, and logging. What do you think makes these concerns crosscutting?

Crosscutting versus hierarchical

A common cycle of work often starts with a single issue. How would you solve a single, concrete, well defined issue? By coding a single, concrete solution! We usually don't think of reuse at this time. We are more likely to think of reuse when we need to solve a newer problem using an already-written block of code.

When we are solving another problem and trying to reuse a block of code, we start to see how we can make adjustments and abstractions. We don't just abstract things to reduce duplication of code. We are also adjusting our definition of the business solution with a more refined, higher level concept. As new requirements come in, our business solution hierarchy evolves.

As we continue solving business issues, it is not unusual to run into infrastructure issues. For example, the need to wrap operations with transactions may arise even though the customer may not have asked for it.

Infrastructure issues can easily crosscut hierarchies of classes. SQL transactions are the perfect example. Manually coding the transactional pattern in every method that needs it would lead to a lot of code duplication. And then what do we do when the transactional requirements change? Rewrite all impacted methods? Unforeseen changes in requirements could easily accrue a of work as we maintain the code. And to top it off, what are the odds that we will code the transaction pattern correctly in every method?

AOP provides another means to address this issue without using copy-and-paste tactics. We will explore this is in more detail throughout this chapter.

Crosscutting elements

The following terms describe the crosscutting elements that make up aspect oriented programming:

`Join point`	A `Join point` is an identifiable point in the execution of a program. This is the place where crosscutting actions are woven in.
`Pointcut`	A `Pointcut` is a program construct that selects join points and collects context at that point.
`Advice`	`Advice` is code that is to be executed at a join point that has been selected by a pointcut.
`Aspect`	An `Aspect` is to AOP what a class is to OOP. It is the central programming construct where a set of pointcuts and advice are defined.
`Advisor/ interceptor`	An `Advisor` or `interceptor` is Spring Python's mechanism for implementing an aspect.

With these terms defined, we will dig in through the rest of this chapter to explore crosscutting problems and their solutions.

Weaving crosscutting behavior

As stated earlier, aspects contain pointcuts and their associated advice. Applying aspects to our code is known as weaving.

Different systems provide various ways to weave. **AspectJ** is a Java-based AOP solution that has both static and dynamic weaving.

AspectJ is the trail blazer of AOP and targets the Java platform. Spring Python doesn't necessarily integrate with AspectJ, nor attempt to mimic everything AspectJ does. Instead, Spring Python looks at AspectJ as a mature example of implementing AOP.

For more information about AspectJ, you could read *AspectJ in Action* by *Ramnivas Laddad*. While that book focuses on AspectJ and Java, it also has much more detail about the concepts of AOP.

Static weaving is applied at same time that code compilation occurs. Dynamic weaving is applied to already compiled code.

Spring Python provides **dynamic weaving** through the use of **proxies**. Proxies are configured using the IoC container. Calls that are normally targeted at the object are instead routed through the proxy. The proxy evaluates possible pointcuts and applies `Advice`. This is also known as **intercepting**.

Adding caching to Spring Python objects

In this example, we will enhance the wiki engine that we have been developing by writing a service that retrieves wiki text from the database and converts it to HTML. Then, to limit the load on our infrastructure, we will add caching support.

1. First, we need to code our service. The service will call a data access component to retrieve the wiki text stored in our database. Then we will convert it to HTML and hand it back to the caller.

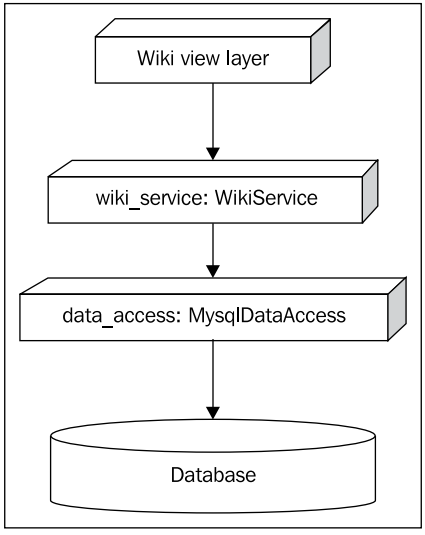
```
class WikiService(object):
    def __init__(self, data_access):
        self.data_access = data_access

    def get_article(self, article):
        return self.data_access.retrieve_wiki_text(article)

    def store_article(self, article):
        self.data_access.store_wiki_text(article)

    def html(self, text):
        pass # return wiki text converted to HTML

    def statistics(self, article):
        hits = self.data_access.hits(article)
        return (hits, hits /
            len(self.data_access.edits(article)))
```

   Here we have a `WikiService` similar to the one we used in our IoC example. In addition to having a `statistics` method, this version is able to retrieve wiki text from the database and format it into HTML. It can also store edited wiki articles in the database. In this example we aren't particularly concerned with the code that transforms wiki text to HTML and so we've omitted it.

2. Let's put our `WikiService` in a Spring Python IoC container, so that it can be used as part of our wiki application.

```
from springpython.config import PythonConfig
from springpython.config import Object

class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()

    @Object
    def data_access(self):
        return MysqlDataAccess()

    @Object
    def wiki_service(self):
        return WikiService(self.data_access())
```

This should look familiar. It is the same IoC container definition we used in the previous chapter. The following block diagram shows the components of our application, with the Wiki's view layer calling into `WikiService`, which in turn calls `data_access`, finally reaching the database.



3. Let's assume that our wiki engine is now being used for a very popular site with lots of articles and users. Multiple requests for the same article between edits are likely going to be common and will tax the database server with a database query every time an article is retrieved. To limit that performance hit, let's code a simple caching solution.

```
class WikiServiceWithCaching(object):
    def __init__(self, data_access):
        self.data_access = data_access
        self.cache = {}

    def get_article(self, article):
        if article not in self.cache:
            self.cache[article] =
                self.data_access.retrieve_wiki_text(article)
        return self.cache[article]

    def store_article(self, article):
        del self.cache[article]
        self.data_access.store_wiki_text(article)

    def html(self, text):
        pass # return wiki text converted to HTML

    def statistics(self, article):
        hits = self.data_access.hits(article)
        return (hits, hits /
            len(self.data_access.edits(article)))
```

To handle multiple requests for the same article, `WikiServiceWithCaching` stores the wiki text in an internal dictionary called `cache`. This cache is then checked for the requested article before falling back to the database if the article is not in the cache. The cache is cleared whenever new edits are made. Finally the `statistics` don't have any caching at all.

To bring in our new caching wiki service, we just need to change the `wiki_service` declaration in our IoC container definition.

```
class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()

    @Object
    def data_access(self):
```

```
        return MysqlDataAccess()

    @Object
    def wiki_service(self):
        return WikiServiceWithCaching(self.data_access())
```

This idea is pretty simple and should lessen the load on our database server. However, by mixing caching with wiki services we have violated the **Single Responsibility Principle (SRP)**: a class should have one (and only one), reason to change. Changes to one of these functions could break the other. It is also harder to isolate these functions for testing and, ultimately, makes the code harder to read and understand.

4. Let's decouple these two concerns, maintaining the wiki and caching, by pulling our caching mechanism into a separate class and then delegating to our original `WikiService`.

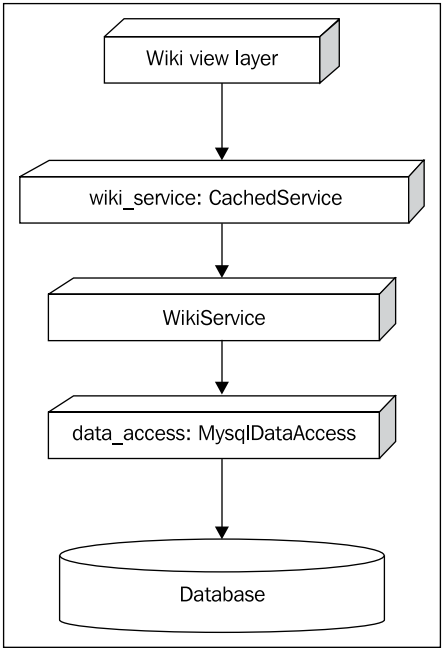
```
class CachedService(object):
    def __init__(self, delegate):
        self.cache = {}
        self.delegate = delegate

    def get_article(self, article):
        if article in self.cache:
            return self.cache[article]
        else:
            return self.delegate.get_article(article)

    def store_article(self, article):
        del self.cache[article]
        self.delegate.store_article(article)

    def html(self, text):
        return self.delegate.html(text)

    def statistics(self, article):
        return self.delegate.statistics(article)
```

Now we need to update our IoC configuration.

```
class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()
```

```
@Object
def data_access(self):
    return MysqlDataAccess()

@Object
def wiki_service(self):
    return CachedService(WikiService(self.data_access()))
```

Our architecture has changed slightly, with the `Wiki view layer` calling into our `CachedService`.



Now the SRP is obeyed as each class has only one responsibility. One class handles the caching by passing off all necessary `WikiService` calls to `delegate`. However, it isn't very practical. While CachedService (Code in Text) separates caching logic from `WikiService`, our usage of an adapter introduces some ugly constraints. The `CachedService` must have the same interface as the `WikiService` in order to handle its requests and we even had to code a passthrough for `statistics`. `CachedService` is also too specialized and wouldn't work for any generalized solution. Every time we add another method to `WikiService`, we have to change `CachedService`, giving us a tightly coupled pair of classes. Let's fix that now.

5. Let's use Spring Python to code an `Interceptor` that not only separates caching from wiki text management, but is also general enough to be reused in other places.

```
from springpython.aop import *

class CachingInterceptor(MethodInterceptor):
    def __init__(self):
        self.cache = {}

    def invoke(self, invocation):
        if invocation.method_name.startswith("get"):
            if invocation.args not in self.cache:
                self.cache[invocation.args] =
                                invocation.proceed()
            return self.cache[invocation.args]

        elif invocation.method_name.startswith("store"):
            del self.cache[invocation.args]
            invocation.proceed()
```

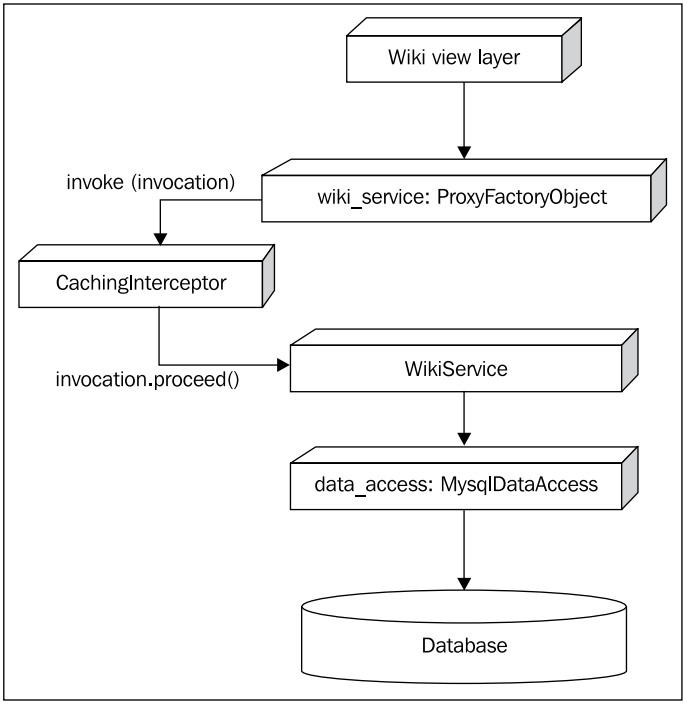
`CachingInterceptor` defines a Spring Python `aspect` that has all the caching functionality and none of the wiki functionality. It contains both `Advice` (the caching functionality) and a `pointcut` (only applies to methods starting with `get`).

6. To use `CachingInterceptor`, we must create a Spring Python AOP proxy with an instance of our aspect as well as an instance of `WikiService`.

```
class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()

    @Object
    def data_access(self):
        return MysqlDataAccess()

    @Object
    def wiki_service(self):
        return ProxyFactoryObject(
                    target=WikiService(self.data_access()),
                    interceptors=CachingInterceptor())
```

By creating a proxy and wiring it with our aspect, we have dynamically woven the aspect into the code. In our situation, we have created an instance of Spring Python's `ProxyFactoryObject` to combine `WikiService` with `CachingInterceptor`.

By moving the Spring Python object `wiki_service` from `WikiService` to the `ProxyFactoryObject`, our call sequence now flows through our new interceptor, as shown in the following diagram.



Now, instead of having a specialized cache handler, we have a proxy that links to our generic caching aspect. Whenever the view layer submits a request to `wiki_service`, the calls get routed to `CachingInterceptor`.

The key difference between `CachingInterceptor` and the earlier `CachedService` is how Spring Python bundles up all the information of the original method call into the `invocation` argument and dispatches it to the `invoke` method of `CachingInterceptor`. This allows us to manage entering and exiting from any method on `WikiService` in one place. This behavior is known as **advising the target**.

In our example, `CachingInterceptor` checks if the target method name starts with `get`. If so, it checks the cache, using the target method's arguments as the key (in our case, the article name). If the arguments are not found in the cache, `CachingInterceptor` calls `WikiService` through `invocation.proceed()` as if we had called it directly. The results are stored in the cache, and then returned to the view layer. If the target method name starts with `store`, the cache entry is deleted followed by `invocation.proceed()`.

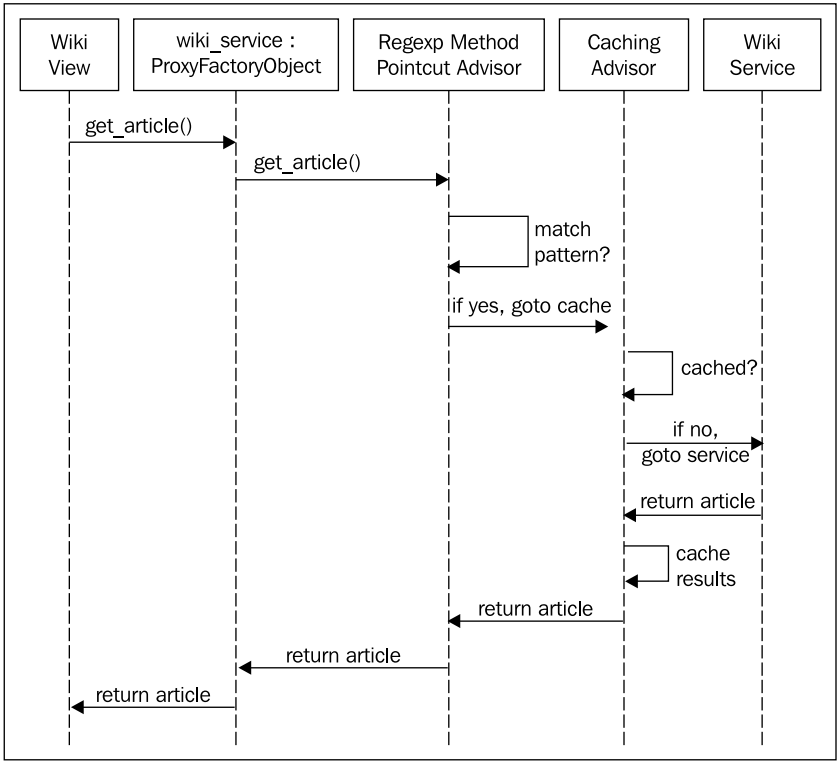
7. `ProxyFactoryObject` routes all method requests through our aspect. However, `CachingInterceptor` can't handle a call to statistics, because it doesn't start with `get` or `store`. To deal with this, let's insert another piece of advice that only sends `get` and `store` calls to `CachingInterceptor`, and all others directly to `WikiService`.

```
class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()

    @Object
    def data_access(self):
        return MysqlDataAccess()

    @Object
    def wiki_service(self):
        advisor = RegexpMethodPointcutAdvisor(
                     advice=[CachingInterceptor()],
                     patterns=[".*get.*", ".*store.*"])
        return ProxyFactoryObject(
                    target=WikiService(self.data_access()),
                    interceptors=advisor)
```

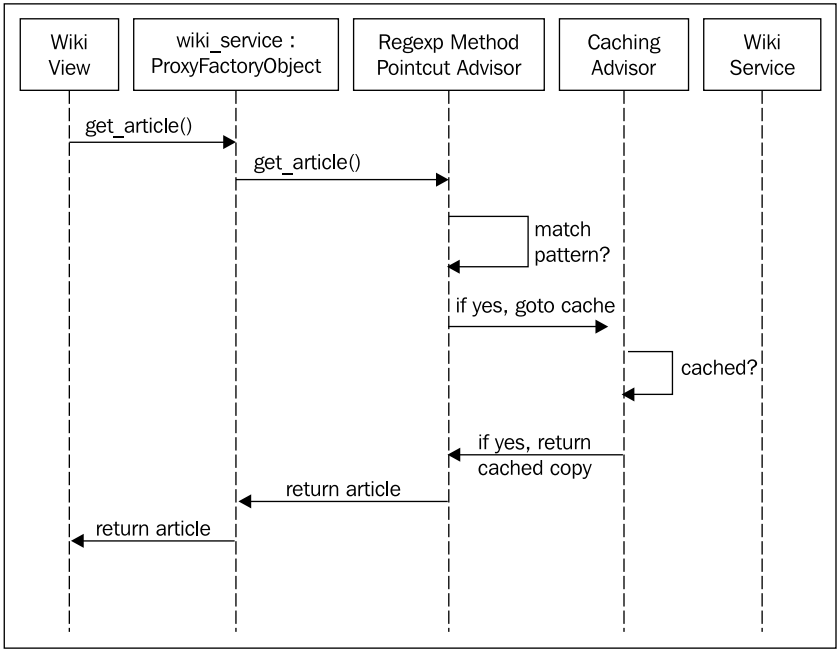
Spring Python's `RegexpMethodPointcutAdvisor` is an out-of-the-box advisor that uses a set of regular expressions to define its pointcuts.

As mentioned earlier, an **advisor** is Spring Python's implementation of an aspect. A `pointcut` is a definition of where to apply an aspect's `Advice`. In this case, the pointcut is defined as a regular expression. But not all pointcuts require regular expressions.

Each pattern is checked against the call stack's complete canonical method name (`<package>.<class>.<method>`) until a match is found (or the patterns are exhausted). If there is a match, the list of advisors is applied. Otherwise, it bypasses the list of advisors and instead directly calls the target. The following diagrams shows how these components are chained together; the first one depicts the sequence of requesting a specific article for the first time:

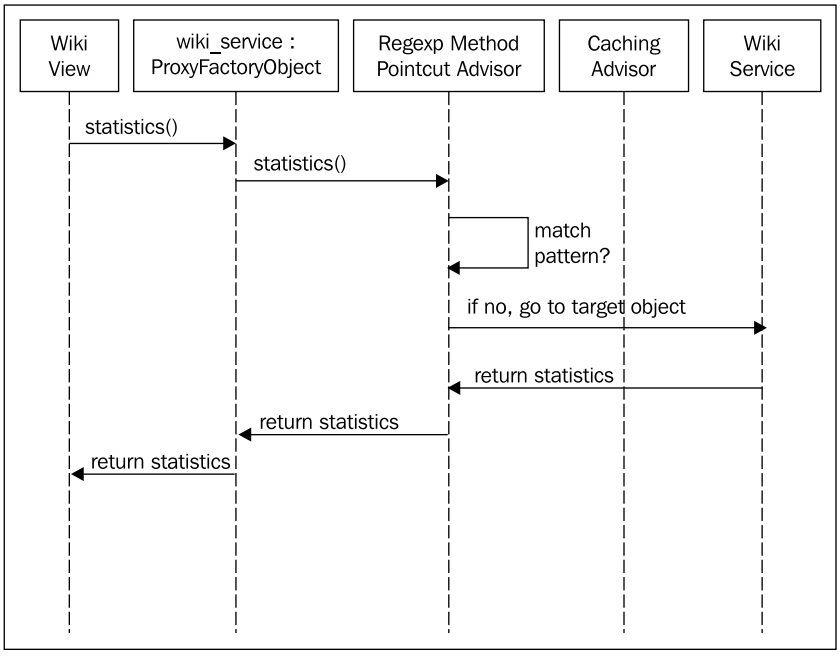


In the previous sequence of steps, the method name is checked to see if it matches the pattern for caching. Assuming that it does, it is forwarded to the caching interceptor. Then the cache is checked. If it hasn't been cached yet, it is fordward to `WikiService`. After reaching `WikiService`, the results are cached and then returned. The next diagram shows the sequence of steps when that article is requested again:



In this case, we again check the same regular expression pattern to see if the method qualifies for caching. Since it does, we next check the cache. Because it's there, we don't have to call `WikiService`, saving us from having to make a database call.

The final sequence shows what happens if we invoke a method that doesn't match our caching pattern.



In this situation, we again exercise the pattern match. But because it doesn't match, `RegexpMethodPointcutAdvisor` falls through to the target object, bypassing the caching advisor. We can go straight to `WikiService` and find up-to-date statistics.

It is important to realize that while the caching interceptor reduces hits to the database, there is still an overhead cost of pattern matching on the method name. This type of overheard cost must be included in any end-to-end performance analysis.

This solves the problem of cleanly applying a caching service to the API of our wiki engine, without breaking the SRP. By using Spring Python's AOP module, we have been able to code a generic, re-usable caching module and plugged it in with no impact to our wiki API.

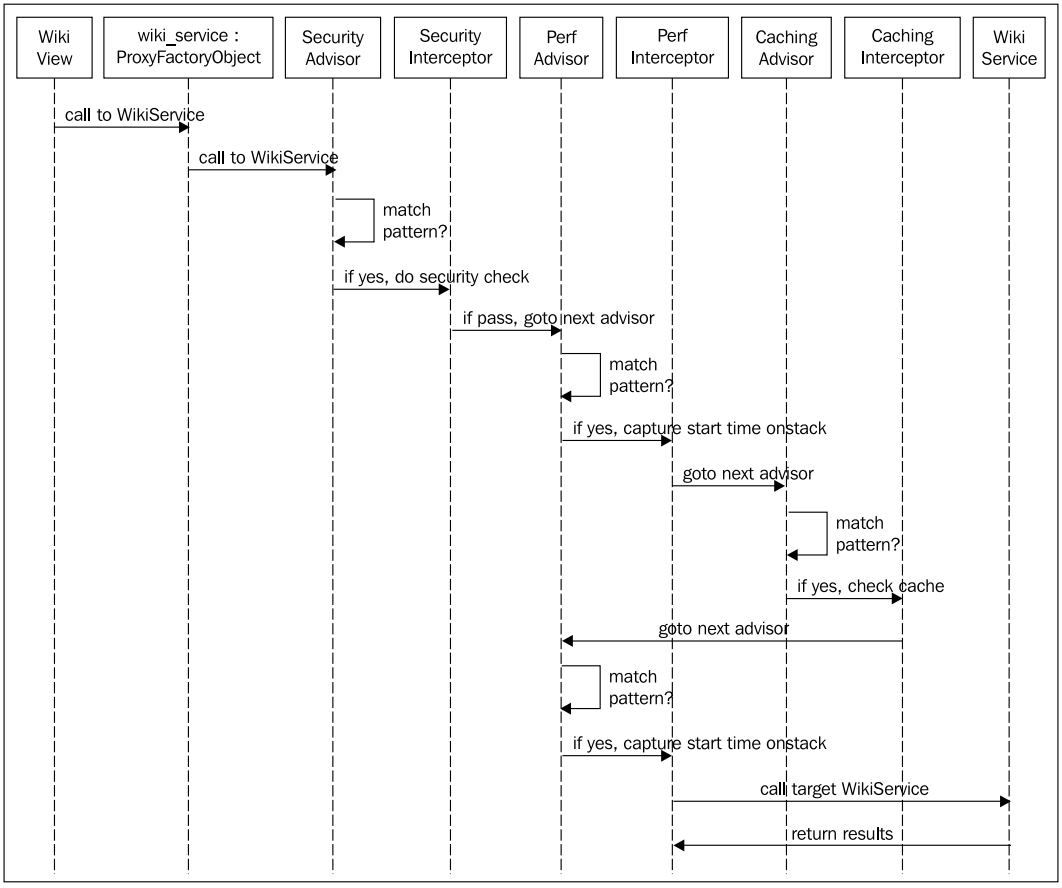
Applying many advisors to a service

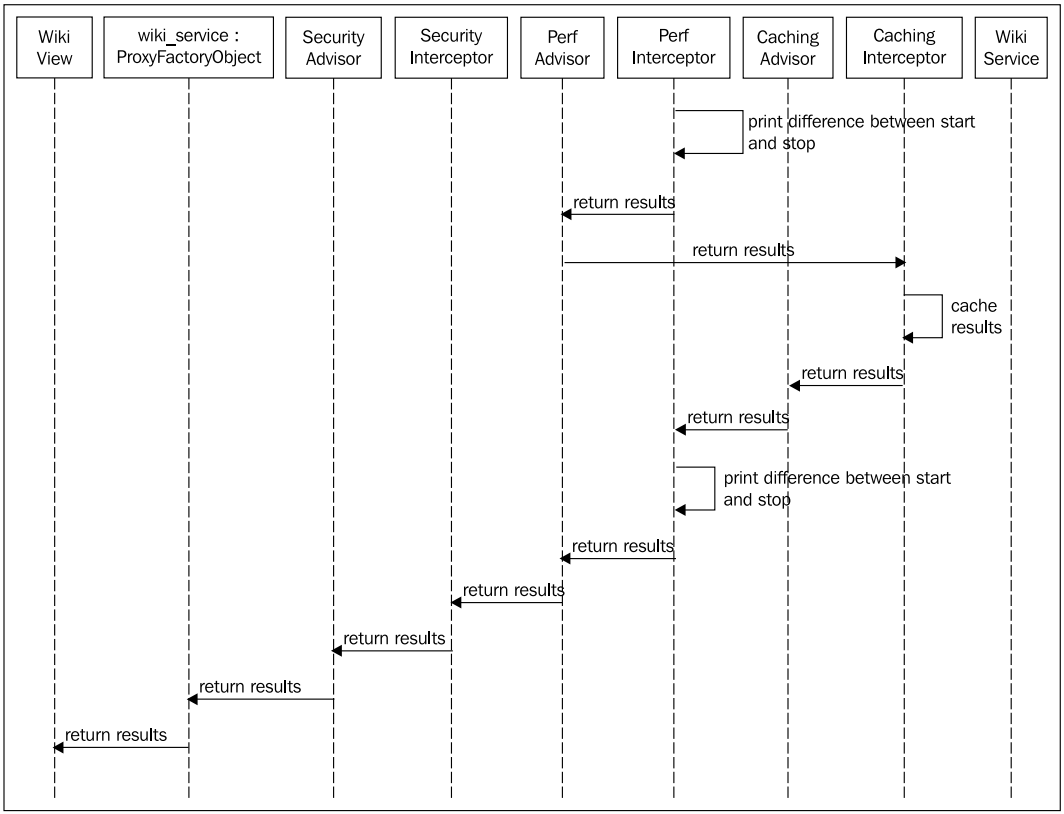
Towards the end of coding our caching solution, we used `RegexpMethodPointcutAdvisor` to conditionally apply a list of advisors based on regular expression patterns. Spring Python's AOP solution supports applying more than one advisor to an object. This makes it easy to mix multiple services together for different objects, such as caching, transactions, security, and logging.

Spring Python makes it easy to add a service on top of an API. There is no limit on how many places a piece of advice can be reapplied. An example of a more complex and realistic configuration is shown below:

```
class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()

    @Object
    def data_access(self):
        return MysqlDataAccess()

    @Object
    def security_advisor(self):
        return RegexpMethodPointcutAdvisor(
            advice=[SecurityInterceptor()],
            patterns=[".*store.*"])

    @Object
    def perf_advisor(self):
        return RegexpMethodPointcutAdvisor(
            advice=[PerformanceInterceptor()],
            patterns=[".*get.*"])

    @Object
    def caching_advisor(self):
        return RegexpMethodPointcutAdvisor(
            advice=[CachingInterceptor()],
            patterns=[".*get.*", ".*store.*"])

    @Object
    def wiki_service(self):
        return ProxyFactoryObject(
            target=WikiService(self.data_access()),
            interceptors=[self.security_advisor(),
                          self.perf_advisor(),
                          self.caching_advisor(),
                          self.perf_advisor()])
```

In this example, `wiki_service` has several advisors: `security_advisor`, `perf_advisor`, and `caching_advisor`. They are chained together in a stack, with `perf_advisor` being used twice. The following diagram shows the call stack of advisors and their interceptors, followed by a detailed explanation of what each advisor does.





`security_advisor` is an instance of `RegexpMethodPointcutAdvisor` that applies `SecurityInterceptor` against any method that begins with `store`.

```
class SecurityInterceptor(MethodInterceptor):
    def invoke(self, invocation):
        if user.has_access(invocation):
            return invocation.proceed()
        else:
            raise SecurityException("Unauthorized Access")
```

`SecurityInterceptor` checks if the user's credentials are adequate to complete this operation. If access is granted, it calls `invocation.proceed()`, which flows on to the first instance of `perf_advisor`. If access is denied, it raises a security exception, breaking out of the entire call stack, leaving the caller to handle the exception. In this example, `user` is assumed to be a global variable containing the current user's profile.

`perf_advisor` is an instance of `RegexpMethodPointcutAdvisor` that applies `PerformanceInterceptor` against any methods that begin with `get`. While there is only one instance of `perf_advisor`, it is used twice in the chain of advisors. This means it will be used twice during the normal flow into `WikiService`.

```
import time
class PerformanceInterceptor(MethodInterceptor):
    def invoke(self, invocation):
        start = time.time()
        results = invocation.proceed()
        stop = time.time()
        print "Method took %2f seconds" % (stop-start)
        return results
```

`PerformanceInterceptor` measures the performance of a method by capturing system time, calling `invocation.proceed()`, and then capturing system time again when the invocation is complete. It then prints out the difference in times on the screen, and finally returns back to the calling advisor. The first time it is used is before `caching_advisor`, and the second time after `caching_advisor`. This allows measuring both cached and un-cached calls, measuring relief provided by the caching.

We've already seen how the `CachingInterceptor` works.

While the entire flow involved in making a call to `WikiService` is much more complicated than our earlier example, it was easy to quickly define extra services and apply them to our `WikiService` API. Each interceptor is neat and clean, and easy to understand. This helps lower maintenance costs. The interceptors are nicely decoupled from each other as well as `WikiService`, making it easier to mix and match aspects to suit our requirements. The exact sequence in which everything is wired together is conveniently kept in one place, our IoC container definition. This is shown by how we used `perf_advisor` twice, measuring performance with and without caching. This collectively demonstrates how easy it is to build new services and apply them to existing APIs.

Performance cost of AOP

AOP isn't free. There is a certain overhead cost involved with wrapping a target object with a Spring Python AOP proxy and performing checks on method calls. This clearly depends on how much advice you are using and what your pointcuts are. Using lots of regular expressions can get expensive, while simply applying an interceptor to every method with no conditional checks has a smaller cost.

There is also a different impact on whether or not the advised target object is inside a tight loop.

It is important to measure costs before optimizing. Premature optimization can result in wasted effort with little benefit. Using a Python profiler (or Java profiler when using Jython) is key to identifying performance bottlenecks.

AOP is a paradigm, not a library

There are many articles about AOP. Developers have discussed whether or not Python needs an AOP library.

Aspect oriented programming is not just a library. It's a paradigm, just like object oriented programming and functional programming.

The key goal that AOP strives to solve is writing crosscutting solutions in places where OOP can't. OOP solves problems where the solution can be inherited. AOP is transversal whilst OOP is vertical. Either way, the goals are the same: "**DRY (Don't Repeat Yourself)**" and "obey the SRP". This also is described by the **1:1 Principle**, which states that one requirement has one and only one manifestation in the implemented code. This is a combination of the DRY and SRP principles.

Developers resort to copy-and-paste style coding when it comes to adding logging, caching, transactions, etc. This code which spans across multiple service classes is repetitive and represents a single strategy. Whenever there is a change the change must be repeated everywhere, and this is risky and costly.

In our example, we used Spring Python to code an interceptor and applied it using pointcuts. This allowed us to avoid repetitive code while still keeping things simple and nicely encapsulated. This has the following benefits.

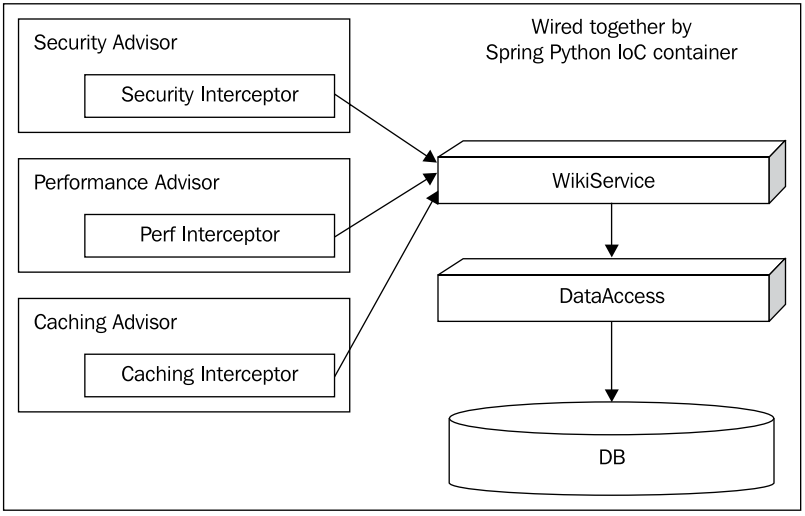
- Any time a new method is added to `WikiService`, we can check the IoC configuration to see if the `CachingInterceptor` applies. There is only one place to check, meaning it is easy to check the pointcuts
- We can easily tune the pointcuts without touching `CachingInterceptor`. This demonstrates how the advice is cleanly decoupled from the pointcuts

- In our more complex configuration we introduced `PerformanceInterceptor`, where we observed which methods are taking the longest and evaluate them for caching. This was relatively easy to add, because it required no interference with the other interceptors or their pointcuts. This suggests that future updates should be relatively easy to make

With advice decoupled from pointcuts, it is easy to add new pointcuts and new advice, thereby updating our crosscutting behavior. And all of this can be done without impacting the core API we are enhancing.

We have shown that AOP supports loosely coupling solutions. We have also looked at how it is easy to easily apply advice with precision, offering high cohesion. These factors will help us by making it easy to respond to changes we know are coming.

The following diagram shows a high-level view of how the components are wired together.



Each interceptor has its code is cleanly wrapped by an advisor with its pointcuts, forming well defined aspects. Each aspect is also separated from the business logic of the `WikiService`. With this separation of concerns, it is easy to make changes with minimal risk. It is also easy to add new advisors and change the order of execution.

Distinct features of Spring Python's AOP module

Spring Python isn't the only means to code an AOP solution. There are other libraries available, such as `Aspyct` and `aspects.py`. Python's rich feature set also provides the ability to code in an AOP-like fashion, using things like meta-class programming and decorators.

It is important to note what makes Spring Python's AOP solution distinct. While I have tried to include some high-level comparison with `Aspyct` and `aspects.py`, the reader is encouraged to visit these projects and compare the features in more detail.

Difference	Description
Advice applicable to instances of objects.	Spring Python gives the developer the option of applying advice to individual instances of objects, without requiring all instances to adopt the behavior. This gives the developer maximum flexibility. The tradeoff is that if this behavior is desired for all instances, then the developer is responsible to apply it to all instances. `Aspyct` allows you to define aspects and apply them to functions. The behavior permanently modifies the function, and this impacts all instances. The ability to wrap functions around methods is offered by `aspects.py`. This would apply to all instances. It also supports wrapping a subset of instances in this fashion.
No metaclass programming.	For certain problems, metaclass programming may be the easiest solution. For other problems, metaclass programming isn't what is needed. It is an alternative paradigm from standard OOP and procedural development practices. In OOP, classes are templates for creating objects. Metaclasses are templates for creating classes. This means that some crosscutting problems may easily fit metaclasses programming, while other problems do not. Spring Python's AOP solution doesn't require learning a complex concept. Instead, it is based on creating classes and using IoC to define pointcuts that apply the advice.

Difference	Description
Don't need access to source code.	Metaclass programming requires access to the source code. So does applying decorators defined in an AOP library. Both of these options make it harder to apply advice to a 3rd party library, since developers are less likely to take on maintaining patches.
	`Aspyct` and `aspects.py` don't require altering the source code. However, `Aspyct's` primary means of application involves using decorators, which would require access to the source code.
	Considering how Spring Python leverages its IoC container to apply advice, it is easy to utilize classes from any library and apply independent advisors.
No monkey patching.	Monkey patching is where an object's functions are added, removed, or replaced at runtime. While some people consider monkey patching a valuable tool, it must be used with care. There is a certain level of associated risk with altering the class definitions of libraries, which may result in unpredictable bugs.
	Spring Python utilizes proxies to merge target objects with interceptors. This makes it easy to add and remove advice, while maintaining a nice separation between services and target APIs. Later on in this chapter, automated testing of aspects will be discussed.

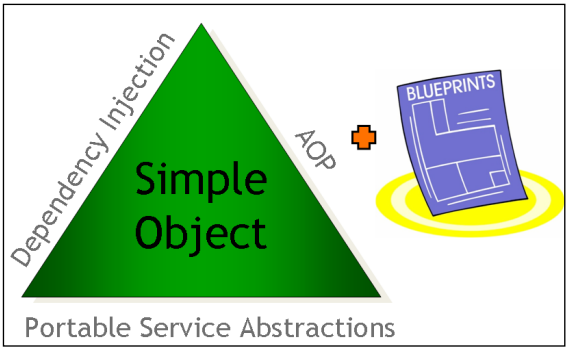
The risks of AOP

AOP carries its own risks. When some of the functionality (in our case, caching), is moved out of our core code and into an interceptor, it may not be apparent when the caching logic is active. We may not even be aware that there is caching in the system. But the same could be said about OOP practices, where commonly used functionality is moved to other classes up or down the inheritance hierarchy. All of these coding styles are better served by the right set of tools, developer training, documentation, automated test suites, and communication amongst the development team. Since AOP is relatively new compared to OOP, the tool sets aren't as mature and developers are not as familiar with the concepts.

But this doesn't mean AOP should be abandoned. Instead, it should be evaluated just like any other technology, language, and tool suite used by the team.

AOP is part of the Spring triangle

The following diagram—known as the Spring triangle—encompasses the key principles used by Spring Python:



Using Spring Python's IoC container we have been able to take a simple `WikiService` object and layer on a `CachingService` through **AOP**. By using **Dependency Injection**, we have kept `WikiService` clear of any Spring Python dependencies. To read the details of this, we just look up the blue prints of our IoC container in `WikiProductionAppConfig`. Spring also utilized **Portable Service Abstractions**, such as `DatabaseTemplate`, to reduce the need to work with low level APIs. Later on in this book, we will revisit the Spring triangle to discuss this in more detail.

Testing our aspects

If you're a professional software developer, you'll be feeling a little nervous at this point. We've written quite a bit of code, in a number of aspects, and things have 'just worked'. We all know this is rarely the case.

Aspects are just like any other piece of code, they need to be tested. To mitigate risks, and to keep to best practices, we should write automated tests for our aspects. There are different types of tests to pursue. For our caching example, it would be useful to isolate the caching functionality to make sure it meets our requirements; commonly referred to as unit testing. Also, ensuring that the right advice is being applied to the right functions is critical to confirming that AOP is working and this can be exercised using an integration test.

Decoupling the service from the advice

In this chapter we have developed an aspect that generalizes caching. We showed a later enhancement with several more aspects. For the rest of this example, we are going to revert to the earlier configuration that has only our `CachingInterceptor` plugged in.

Our aspect happens to be tightly coupled with its caching service. Even though the caching solution we have coded is simple, let's assume that we are planning to replace it with something more sophisticated. To make it easier to code and test enhancements to our caching service, let's go ahead and break it out into a separate module.

1. First, let's rewrite the advice so that it is using a caching service, instead of handling the caching itself.

```
class CachingInterceptor(MethodInterceptor):
    def __init__(self, caching_service=None):
        self.caching_service = caching_service

    def invoke(self, invocation):
        if invocation.method_name.startswith("get"):
            if invocation.args not in self.caching_service.keys():
                self.caching_service.store(invocation.args,
                    invocation.proceed())
            return self.caching_service.get(invocation.args)

        elif invocation.method_name.startswith("store"):
            self.caching_service.del(invocation.args)
            invocation.proceed()
```

2. Next, let's move the caching logic into a separate class.

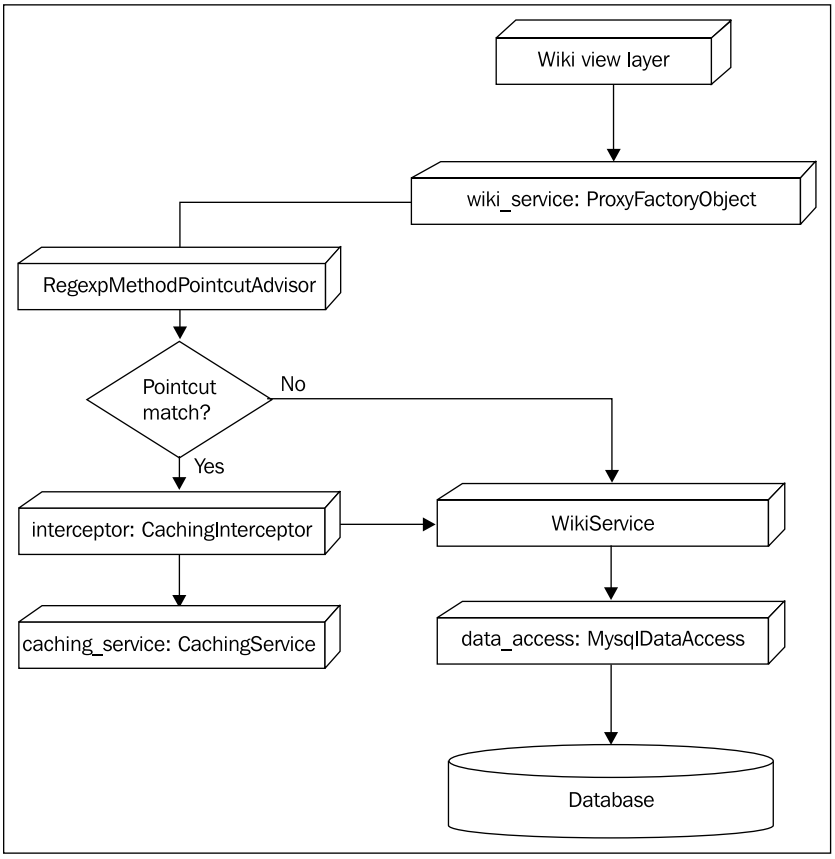
```
class CachingService(object):
    def __init__(self):
        self.cache = {}

    def keys(self):
        return self.cache.keys

    def store(self, key, value):
        self.cache[key] = value

    def get(self, key):
```

```
        return self.cache[key]

    def del(self, key):
        del self.cache[key]
```

3. This requires an update to our IoC blue prints, so that we inject an instance of `CachingService` into the `CachingInterceptor`.

```
class WikiProductionAppConfig(PythonConfig):
    def __init__(self):
        super(WikiProductionAppConfig, self).__init__()

    @Object
    def data_access(self):
        return MysqlDataAccess()

    @Object
    def caching_service(self):
        return CachingService()

    @Object
    def interceptor(self):
        return CachingInterceptor(self.caching_service())

    @Object
    def wiki_service(self):
        advisor = RegexpMethodPointcutAdvisor(
                     advice=[self.interceptor()],
                     patterns=[".*get.*", ".*store.*"])
        return ProxyFactoryObject(
                    target=WikiService(self.data_access()),
                    interceptors=advisor)
```

The following diagram shows how we have pulled `CachingService` into a separate component, and promoted it along with `CachingInterceptor` to fully named Spring Python objects.



Testing our service

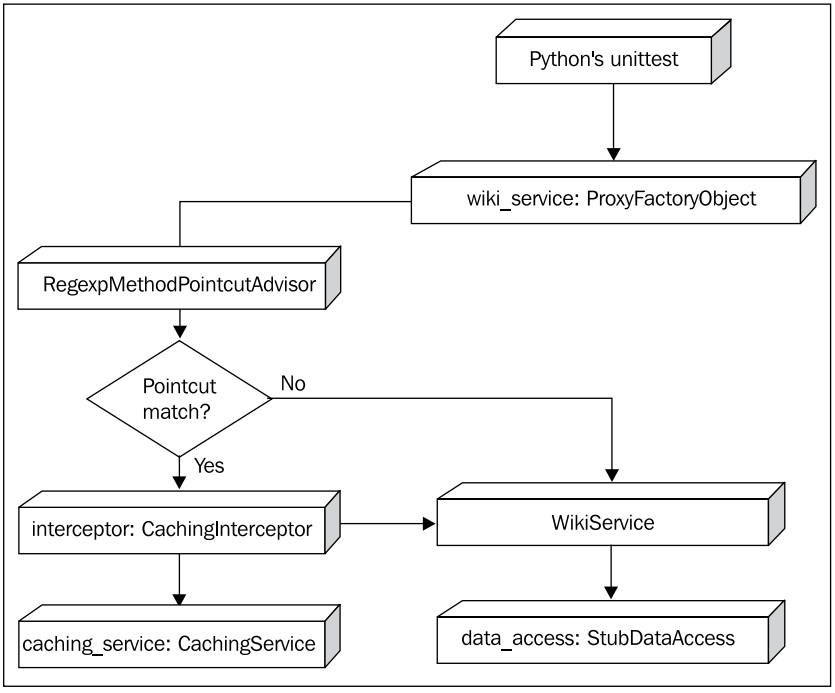
Now that we have pulled our caching service into a separate module, it is easy to write some automated tests.

1. Before we write any tests, let's create a testable version of the IoC container that isolates us from any layers above and below `WikiService` and the `ProxyFactoryObject` that contains our aspect.

```
class WikiTestAppConfig(WikiProductionAppConfig):
    def __init__(self):
        super(WikiTestAppConfig, self).__init__()
```

```
    @Object
    def data_access(self):
        return StubDataAccess()
```

2. This replaces `MysqlDataAccess` with `StubDataAccess` which runs quicker, avoids database contention with other developers, and has pre-formatted responses for each method. With Python's unit test framework taking the place of the view layer as the caller, we have isolated our code base for testing.



3. Let's write a test that verifies the values of the caching service.

```
class CachedWikiTest(unittest.TestCase):
    def testCachingService(self):
        context = ApplicationContext(WikiTestAppConfig())
        caching_service = context.get_object("caching_service")
        self.assertEquals(len(caching_service.keys()), 0)
        caching_service.store("key", "value")
        self.assertEquals(len(caching_service.keys()), 1)
        self.assertEquals(caching_service.get("key"), "value")
        caching_service.del("key")
        self.assertEquals(len(caching_service.keys()), 0)
```

In this test method, we fetch a copy of `caching_service` from our IoC container. Then, we verify it's empty. Next, we store a simple key/value pair, and verify the size and content of the cache. Finally, we exercise `caching_service`'s `del()` method, and verify that the cache has been properly emptied.

I admit that `CachingService` is a bit over engineered, considering it's just a Python dictionary. I normally wouldn't write unit tests for language-level structures like this. However, the purpose of this example is to show that we can move our solution into a separate module, free of any AOP machinery, and then enhance it with more sophisticated features. We could modify it to be a distributed cache that would persist across multiple nodes without impacting either `WikiService` or `CachingInterceptor`.

Testing the caching service is valuable because it keeps bugs from creeping back into the code base. But we also need to know that our aspect is being correctly woven with the Wiki API that we have coded.

Confirming that our service is correctly woven into the API

We have confirmed that the caching service works by isolating it and writing an automated test. The final task we need to complete is verifying that we have wired the caching service into our API correctly.

Let's add another test method to `CachedWikiTest` showing that `WikiService` is being properly advised.

```
    def testWikiServiceWithCaching(self):
        context = ApplicationContext(WikiTestAppConfig())
        caching_service = context.get("caching_service")
        self.assertEquals(len(caching_service.keys()), 0)
        wiki_service = context.get_object("wiki_service")
        wiki_service.statistics("Spring Python")
        self.assertEquals(len(caching_service.keys()), 0)
        html = wiki_service.get_article("Spring Python")
        self.assertEquals(len(caching_service.keys()), 1)
        wiki_service.store_article("Spring Python")
        self.assertEquals(len(caching_service.keys()), 0)
```

In this test, we fetch a copy of `caching_service` from our IoC container and verify that it's empty. Next, we fetch a copy of `wiki_service` from our IoC container. Inside our IoC container, we know that `caching_service` is linked to `wiki_service` through some AOP advice. We call `statistics`, and assert that the cache is still empty, since the advice doesn't apply to that method. Next, we call `get_article`, and verify that the cache has a new entry. Finally, we call `store_article`, and verify that it cleared the cache.

Combining this test with the earlier one, we clearly show that our `CachingInterceptor` advice is working as expected. Having used the IoC container, we have decoupled things nicely, and it is now easy to adjust one class with no impact to the other.

Summary

We have explored how to solve cross cutting problems using aspect oriented programming. These problems include things like caching, security, and performance measuring; problems that all require ugly copy-paste solutions, if we resort to standard OOP practices.

Spring Python's AOP module helps us to uphold the DRY and SRP principles. This reduces maintenance costs and helps us to handle future changes that are always coming.

In this chapter, we learned:

- How to add caching to Spring Python objects
- How to extend the example by adding performance and security advisors
- AOP is a paradigm, and not just a library
- The distinct features of Spring Python's AOP, compared to other libraries as well as Python language features
- The risks of AOP and how to mitigate them by automated testing
- Some tips about applying `Advice`

In the next chapter, we will look at easily writing SQL queries using Spring Python's `DatabaseTemplate`.

4

Easily Writing SQL Queries with Spring Python

Many of our applications contain dynamic data that needs to be pulled from and stored within a relational database. Even though key/value based data stores exist, a huge majority of data stores in production are housed in a SQL-based relational database.

Given this de facto requirement, it improves developer efficiency if we can focus on the SQL queries themselves, and not spend lots of time writing plumbing code and making every query fault tolerant.

In this chapter, we will learn:

- The classic SQL query issue that affects APIs in many modern programming languages
- Using Spring Python's `DatabaseTemplate` to reduce query management code
- Comparing `DatabaseTemplate` with **Object Relational Mappers (ORMs)**
- Combining `DatabaseTemplate` with an ORM to build a robust application
- Testing queries with mocks

The classic SQL issue

SQL is a long existing standard that shares a common paradigm for writing queries with many modern programming languages (including Python). The resulting effect is that coding queries by hand is laborious. Let's explore this dilemma by writing a simple SQL query using Python's database API.

1. First, let's create a database schema for our wiki engine so that we can store and retrieve wiki context.

```
DROP TABLE IF EXISTS article;

CREATE TABLE article (
  id serial PRIMARY KEY,
  title VARCHAR(11),
  wiki_text VARCHAR(10000)
);

INSERT INTO article
(id, title, wiki_text
VALUES
(1,
 'Spring Python Book',
 'Welcome to the [http://springpythonbook.com Spring Python] book,
where you can learn more about [[Spring Python]].');
INSERT INTO article
(id, title, wiki_text
VALUES
(2,
 'Spring Python',
 '\'\'\'Spring Python\'\'\' takes the concepts of Spring and
applies them to world of [http://python.org Python].');
```

2. Now, let's write a SQL statement that counts the number of wiki articles in the system using the database's shell.

```
SELECT COUNT(*) FROM ARTICLE
```

3. Now let's write some Python code that will run the same query on an `sqlite3` database using Python's official database API (`http://www.python.org/dev/peps/pep-0249`).

```
import sqlite3

db = sqlite3.connect("/path/to/sqlite3db")

cursor = db.cursor()
results = None
try:
    try:
        cursor.execute("SELECT COUNT(*) FROM ARTICLE")
        results = cursor.fetchall()
    except Exception, e:
        print "execute: Trapped %s" % e
finally:
    try:
        cursor.close()
    except Exception, e:
        print "close: Trapped %s, and throwing away" % e

return results[0][0]
```

That is a considerable block of code to execute such a simple query. Let's examine it in closer detail.

4. First, we `connect` to the database. For `sqlite3`, all we needed was a path. Other database engines usually require a username and a password.
5. Next, we create a `cursor` in which to hold our result set.
6. Then we `execute` the query. To protect ourselves from any exceptions, we need to wrap this with some exception handlers.
7. After completing the query, we fetch the results.
8. After pulling the results from the result set into a variable, we `close` the cursor.
9. Finally, we can return our response. Python bundles up the results into an array of tuples. Since we only need one row, and the first column, we do a double index lookup.

What is all this code trying to find in the database? The key statement is in a single line.

```
cursor.execute("SELECT COUNT(*) FROM ARTICLE")
```

What if we were writing a script? This would be a lot of work to find one piece of information. Granted, a script that exits quickly could probably skip some of the error handling as well as closing the cursor. But it is still is quite a bit of boiler plate to just get a cursor for running a query.

But what if this is part of a long running application? We need to close the cursors after every query to avoid leaking database resources. Large applications also have a lot of different queries we need to maintain. Coding this pattern over and over can sap a development team of its energy.

Parameterizing the code

This boiler plate block of code is a recurring pattern. Do you think we could parameterize it and make it reusable? We've already identified that the key piece of the SQL statement. Let's try and rewrite it as a function doing just that.

```
import sqlite3

def query(sql_statement):
    db = sqlite3.connect("/path/to/sqlite3db")

    cursor = db.cursor()
    results = None
    try:
        try:
            cursor.execute(sql_statement)
            results = cursor.fetchall()
        except Exception, e:
            print "execute: Trapped %s" % e
    finally:
        try:
            cursor.close()
        except Exception, e:
            print "close: Trapped %s, and throwing away" % e

    return results[0][0]
```

Our first step nicely parameterizes the SQL statement, but that is not enough. The `return` statement is hard coded to return the first entry of the first row. For counting articles, what we have written its fine. But this isn't flexible enough for other queries. We need the ability to plug in our own results handler.

```
import sqlite3

def query(sql_statement, row_handler):
    db = sqlite3.connect("/path/to/sqlite3db")

    cursor = db.cursor()
    results = None
    try:
        try:
            cursor.execute(sql_statement)
            results = cursor.fetchall()
        except Exception, e:
            print "execute: Trapped %s" % e
    finally:
        try:
            cursor.close()
        except Exception, e:
            print "close: Trapped %s, and throwing away" % e

    return row_handler(results)
```

We can now code a custom handler.

```
def count_handler(results):
    return results[0][0]

query("select COUNT(*) from ARTICLES", count_handler)
```

With this custom results handler, we can now invoke our `query` function, and feed it both the query and the handler. The only thing left is to handle creating a connection to the database. It is left as an exercise for the reader to wrap the `sqlite3` connection code with a factory solution.

What we have coded here is essentially the core functionality of `DatabaseTemplate`. This method of taking an algorithm and parameterizing it for reuse is known as the **template pattern**. There are some extra checks done to protect the query from SQL injection attacks.

Replacing multiple lines of query code with one line of Spring Python

Spring Python has a convenient utility class called `DatabaseTemplate` that greatly simplifies this problem.

1. Let's replace the two lines of import and connect code from the earlier example with some Spring Python setup code.

   ```
   from springpython.database.factory import Sqlite3ConnectionFactory
   from springpython.database.core import DatabaseTemplate

   conn_factory = Sqlite3ConnectionFactory("/path/to/sqlite3db")
   dt = DatabaseTemplate(conn_factory)
   ```

   At first glance, we appear to be taking a step back. We just replaced two lines of earlier code with four lines. However, the next block should improve things significantly.

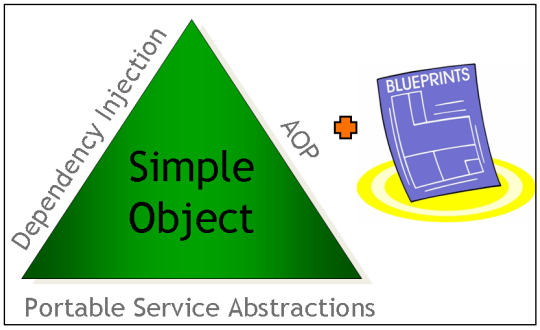
2. Let's replace the earlier coded query with a call using our instance of `DatabaseTemplate`.

   ```
   return dt.query_for_object("SELECT COUNT(*) FROM ARTICLE")
   ```

Now we have managed to reduce a complex 14-line block of code into one line of Spring Python code. This makes our Python code appear as simple as the original SQL statement we typed in the database's shell. And it also reduces the noise.

The Spring triangle—Portable Service Abstractions

We saw this diagram earlier in the book, as an illustration of the key principles behind Spring Python.



The `DatabaseTemplate` represents a **Portable Service Abstraction** because:

- It is **portable** because it uses Python's standardized API, not tying us to any database vendor. Instead, in our example, we injected in an instance of `Sqlite3ConnectionFactory`
- It provides the useful **service** of easily accessing information stored in a relational database, but letting us focus on the query, not the plumbing code
- It offers a nice **abstraction** over Python's low level database API with reduced code noise. This allows us to avoid the cost and risk of writing code to manage cursors and exception handling

`DatabaseTemplate` handles exceptions by catching and holding them, then properly closing the cursor. It then raises it wrapped inside a Spring Python `DataAccessException`. This way, database resources are properly disposed of without losing the exception stack trace.

Using DatabaseTemplate to retrieve objects

Our first example showed how we can easily reduce our code volume. But it was really only for a simple case. A really useful operation would be to execute a query, and transform the results into a list of objects.

1. First, let's define a simple object we want to populate with the information retrieved from the database. As shown on the Spring triangle diagram, using simple objects is a core facet to the 'Spring way'.

```
class Article(object):
    def __init__(self, id=None, title=None, wiki_text=None):
        self.id = id
        self.title = title
        self.wiki_text = wiki_text
```

2. If we wanted to code this using Python's standard API, our code would be relatively verbose like this:

```
cursor = db.cursor()
results = []
try:
    try:
        cursor.execute("SELECT id, title, wiki_text FROM ARTICLE")
        temp = cursor.fetchall()
```

```
            for row in temp:
                results.append(
                    Article(id=temp[0],
                            title=temp[1],
                            wiki_text=temp[2]))
        except Exception, e:
            print "execute: Trapped %s" % e
    finally:
        try:
            cursor.close()
        except Exception, e:
            print "close: Trapped %s, and throwing away" % e

    return results
```

This isn't that different from the earlier example. The key difference is that instead of assigning `fetchall` directly to `results`, we instead iterate over it, generating a list of `Article` objects.

3. Instead, let's use `DatabaseTemplate` to cut down on the volume of code.

```
return dt.query("SELECT id, title, wiki_text FROM ARTICLE",
                ArticleMapper())
```

4. We aren't done yet. We have to code `ArticleMapper`, the object class used to iterate over our result set.

```
from springpython.database.core import RowMapper


class ArticleMapper(RowMapper):
    def map_row(self, row, metadata=None):
        return Article(id=row[0], title=row[1], wiki_text=row[2])
```

`RowMapper` defines a single method: `map_row`. This method is called for each row of data, and includes not only the information, but also the metadata provided by the database. `ArticleMapper` can be re-used for every query that performs the same mapping.

This is slightly different from the parameterized example shown earlier where we defined a row-handling function. Here we define a class that contains the `map_row` function. But the concept is the same: inject a row-handler to convert the data.

Mapping queries by convention over configuration

Our class definition happens to have the same property names as the columns in our database. Spring Python offers `SimpleRowMapper` as a convenient out-of-the-box mapper that takes advantage of this.

Instead of writing the specialized `ArticleMapper`, let's use Spring Python's `SimpleRowMapper` instead.

```
return dt.query("SELECT id, title, wiki_text FROM ARTICLE",
                SimpleRowMapper(Article))
```

`SimpleRowMapper` requires that the class has a default constructor, and also that the class's properties match the query's.

It's important to remember that column-to-property matching is based on the query, not the table. This means we can use SQL aliasing to link up table columns with objects.

Mapping queries into dictionaries

Spring Python also offers the `DictionaryRowMapper`, which conveniently maps the query into a Python dictionary.

Instead of using the `SimpleRowMapper`, let's use Spring Python's `DictionaryRowMapper` instead.

```
return dt.query("SELECT id, title, wiki_text FROM ARTICLE",
                DictionaryRowMapper())
```

This last step breaks out of our original requirement to return a list of `Article` objects. But it is a convenient way of providing a simple 'window on data' scenario and may perfectly match our needs.

DatabaseTemplate and ORMs

- `DatabaseTemplate` focuses on accessing the database without writing lots of boiler plate code
- ORMs focus on mapping tables to objects

`DatabaseTemplate` does not contest with ORM. The choice we must make is between using SQL and processing result sets or using an ORM.

Before going into detail about ORMs and `DatabaseTemplate`, it may be useful to look at a quick example of a popular Python ORM: **SQLAlchemy** (`http://www.sqlalchemy.org`). We could have picked any number of ORMs for this demonstration.

```
from sqlalchemy import *

engine = create_engine("sqlite:/tmp/springpython.db", echo=True)
metadata = BoundMetaData(engine)
article_table = Table('Article', metadata,
                      Column('id', Integer, primary_key=True),
                      Column('title', String()),
                      Column('wiki_text', String()))

article_mapper = mapper(Article, article_table)

session = create_session(bind_to=engine)
articles = session.query(Article)
```

This demonstrates how we would use SQLAlchemy to define the mapping between the `ARTICLE` table and the `Article` class. ORMs also offer many other query options, including filters and, critieria. The key purpose of ORMs is to map databases to objects. There is boiler plate with using ORMs just as there is with raw SQL.

Solutions provided by DatabaseTemplate

If we choose `DatabaseTemplate` for our data needs, we would write our updates, inserts, deletes, and queries using pure SQL. If our team was comprised of database designers and software developers who are all familiar with SQL, this would be of huge benefit—being a more natural fit to their skills. The whole team could contribute to the effort of designing tables, queries, and data management by speaking the common language of SQL.

In this scenario `DatabaseTemplate` would definitely make things easier, as shown earlier. This would allow our team to spend its effort on designing and managing our application's data.

The set of operations provided by `DatabaseTemplate` is provided in the following table.

Operation	Description
`execute(sql_statement, args=None)`	Execute any statement, return number of rows affected
`query(sql_query, args=None, rowhandler=None)`	Query, return list converted by `rowhandler`
`query_for_list(sql_query, args=None)`	Query, return list of Python tuples
`query_for_int(sql_query, args=None)`	Run query for a single column of a single row, and return an integer, throws an exception otherwise
`query_for_long(sql_query, args=None)`	Query for a single column of a single row, and return a long, throws an exception otherwise
`query_for_object(sql_query, args=None, required_type=None)`	Query for a single column of a single row, and return the object with an optional type check
`update(sql_statement, args=None)`	Update the database, return number of rows affected

This may not appear like a lot of operations, but the purpose of `DatabaseTemplate` is to provide easy access to writing SQL. This API provides the power to code inserts, updates, deletes, and queries, while also being able to call stored procedures.

`DatabaseTemplate` also works nicely with Spring Python transactions. This cross cutting feature will be explored in detail in the next chapter.

How DatabaseTemplate and ORMs can work together

Often, while building our application we tend to start with one paradigm, and discover it has its limits. Building an enterprise grade application that supports many users with lots of complex functions from either a pure SQL perspective or from an ORM perspective may exceed the capacity of both. This is when it may be time to use both `DatabaseTemplate` and an ORM in the same application.

It would be a practical solution to use an ORM to code and manage the simple entities and straightforward relationships. We could quickly build persistence into our application and move onto real business solutions.

But the queries needed to generate complex reports, detailed structures, and stored procedures may be better managed using `DatabaseTemplate`.

If we can free up our team from coding custom SQL for the simple objects, they could focus on writing specialized SQL for the hard queries.

Using the *right tool for the right job* should be a key element of our software development process, and having both `DatabaseTemplate` and an ORM in our toolbox is the pragmatic thing to do.

Testing our data access layer with mocks

In previous chapters, we have written automated tests that involved stubbing out the data access layer. Now that we are in the heart of writing the data access layer, we need to look at ways to test our queries. To be specific, we need to make sure we are using `DatabaseTemplate` correctly, along with any custom row mapper we write.

Mocks are used to primarily record what functions are called, and provide options of returning certain values. The idea is to create a set of expected actions, and then call the actual API and measure if this is what happened. This is compared to stubs, which you code yourself and provide canned answers and don't necessarily care what methods were called. Both of these tools are useful for automated testing.

Spring Python uses **pmock**, a Python library inspired by the fluent API of **jMock** (`http://www.jmock.org/`), to do some of its automated testing. You don't have to use this particular mocking library. There are lots of other candidates around. For our purposes we are going to use it here to show the general idea of mocking your data access layer for testing.

Due to lack of updates from the original developers of `pmock`, the source code of this library was added to Spring Python's set of managed code, and has some custom updates. See `http://springpython.webfactional.com` for details on downloading.

1. First, let's code a simple `DataAccess` class that uses `DatabaseTemplate` to fetch the number of articles we have.

```
class DataAccess(object):
    def __init__(self, conn_factory):
        self.dt = DatabaseTemplate(conn_factory)

    def count_wiki_articles(self):
        return self.dt.query_for_object("SELECT COUNT(*) FROM
ARTICLE")
```

   This simple data access layer has one method: `count_wiki_articles`. It utilizes the code we wrote earlier involving the `DatabaseTemplate`. In this example, `DataAccess` expects to be initialized with a connection factory.

   Now, to test this out, we need `DatabaseTemplate` to do its job, but we want to catch it at the right point in order to inject some pre-built values. The piece of code that does the heavy lifting is Spring Python's cursor object, which is supplied by a connection. This means we need to code a special stubbed out connection factory that will hold a mocked cursor.

2. Let's write a specialized class to act as a mocked connection.

```
class MockedConnection(object):
    def __init__(self):
        self.mockCursor = None
    def cursor(self):
        return self.mockCursor
```

   This connection will supply `DatabaseTemplate` with a special type of mocked cursor. Further down, we will see how `mockCursor` gets populated.

3. In order to use this, let's code a connection factory for `DatabaseTemplate` that produces this type of connection.

```
class MockingDBFactory(ConnectionFactory):
    def __init__(self):
        ConnectionFactory.__init__(self, [types.TupleType])
        self.mockedConnection = MockedConnection()
    def connect(self):
        return self.mockedConnection
```

   When this connection factory is asked to connect to the database, it will return a `MockedConnection`.

4. To tie this rigging together, we need to setup a `MockTestCase`. This is a special type of unit test that provides extra hooks to library calls of `pmock`.

```
from pmock import *

class DataAccessMockTestCase(MockTestCase):
    def setUp(self):
        # Create a mock instance to record events
        self.mock = self.mock()
        conn_factory = MockingDBFactory()
        conn_factory.mockedConnection.mockCursor = self.mock
        self.data_access = DataAccess(conn_factory)
```

   Here in the `setUp` method for our test, we grab an instance of `mock()`. This object has APIs to record expectations. The object is meant to be injected so that function calls are then made against it, and at the end of a `MockTestCase` test method, the results are compared with the expectations.

   In this situation, the `mockCursor` is the key holder of the mock. There is also a local copy, so that the `MockTestCase` has a handle to check out the results.

5. After setting all this up, let's define a mocked test method.

```
    def testCountingArticles(self):
        self.mock.expects(once()).method("execute")
        self.mock.expects(once()).method("fetchall")
                                    .will(return_value([(2,)]))

        count = self.data_access.count_wiki_articles()
        self.assertEquals(count, 2)
```

The first two steps of this test use the mock object to define expectations. The `mock` is expected to receive an `execute` method call once, and also a `fetchall` method call. The `fetchall` will return a value of `[(2,)]`.

Not only do we get to check the assertions, but `pmock` will verify that each of these methods was invoked by `DatabaseTemplate`.

How much testing is enough?

For this test scenario, we dug down deep. This type of testing can't get much closer to the hardware of the database server without directly talking to a live database.

It could be viewed that we were really testing the core of `DatabaseTemplate`. In many development situations, this isn't needed. Testing a 3rd party library would probably be out of scope.

This example test scenario is largely based on automated tests used to confirm Spring Python's functionality. Adequate testing for your business needs may involve stubbing or mocking out the data access layer as was shown in earlier parts of this book.

This should introduce you to the concept of mocking, where you measure method calls and answers. This type of testing may perfectly fit other test needs. If the SQL queries have been well isolated in this layer, then it may be safe to say that the only testing needed would be the queries themselves against a live database and in turn, not require any mocking or stubbing at all.

Summary

This chapter has shown a way to write pure SQL without having to deal with the low level cursor and connection management. This removes a lot of boiler plate and error handling that is perfect for a framework to handle.

We have also explored how `DatabaseTemplate` and ORMs can work together to make persistence management easier for developers.

We took a look at mocking, and how we were able to get inside the querying process to verify that the right method calls were being made. Then we stepped back and considered how much testing is enough.

In this chapter we learned:

- The classic SQL query issue that affects APIs in many modern programming languages extends into Python as well
- Using Spring Python's `DatabaseTemplate` let's us get rid of query boiler plate code that gets in the way of solving use cases
- `DatabaseTemplate` is useful for writing SQL code without dealing with cursor management
- `DatabaseTemplate` combined with an ORM can help us build robust applications
- Mocking is a valuable tool in automated testing, but we must choose the right tool for the right situation

In the next chapter, we will look at augmenting our data access layer with non-invasive transactions.

5

Adding Integrity to your Data Access with Transactions

SQL operations are used in lots of applications in order to supply data. A key ingredient used to grow applications to enterprise scale are transactions. They add integrity to data management by defining an atomic unit of work.

An atomic unit of work means that the whole sequence of steps when completed must appear like a single step. If there is any failure in the chain of steps, everything must rollback to the state before the transaction started. For SQL transactions, this means that the state of the database must update or rollback atomically.

This chapter will inspect the pattern of coding and using SQL transactions, and how Spring Python makes it easy to code an otherwise monotonous pattern.

In this chapter, you will learn:

- The classic pattern of SQL transactions and the issues imposed when coding transactions by hand
- That it is easy to add transactions to a banking application using Spring Python's `@transactional` decorator
- You can choose between coding transactions programmatically or by decorator
- You can choose between using and not using the IoC container
- Spring Python provides the means to non-intrusively mix in transactions to non-transactional code without editing the source

Classic transaction issues

Transactions have a simple pattern of execution as shown in this block of pseudo-Python.

```
# Start transaction
try:
    #****************************************
    # Execute business logic that
    # that contains database operations.
    #****************************************

    # Commit transaction
except:
    # Rollback transaction
```

Let's look at the steps involved in defining a transaction:

1. First, the transaction must be *started*.
2. Then, a block of business code is executed, which contains several database operations, including `select`, `update`, and `insert`.
3. Finally, the transaction is *committed*. If any type of error occurs, an exception handler *rolls back* the transaction, undoing all the changes that were made.

The problem with writing transactions manually is similar to the problem with writing SQL operations manually. There is a lot of boilerplate code that must be written. The boilerplate has a tangling effect because it must be written before and after the business code in order for the transaction to be handled correctly. This pattern must then be copied into every location where some business logic needs to be wrapped in a transaction.

This violates the **DRY** (**Don't Repeat Yourself**) principle. It also tangles our code with not only business logic, but transactional logic as well. And it breaks the **Single Responsibility Principle** (**SRP**) by having our code depend on both business and integrity requirements.

Violating these principles puts us at risk. We might introduce bugs when changes to the transactional logic are needed, but we fail to to repeat them in all the right places.

This is a crosscutting problem because it extends beyond our class hierarchy. Steps contained within the unit of work can quickly cut across several classes. This is a perfect use case for aspect oriented programming that was discussed earlier in this book. As we delve into this in more detail in this chapter, we will see how to solve it with ease using Spring Python.

Creating a banking application

Until now, the various chapters have shown sample code that involved designing a wiki engine. For this chapter, a banking application will be used as the basis for code examples. Banks must be able to move money around and not lose a single penny, maintaining a high integrity. This provides a simple problem space to demonstrate writing transactional code.

1. First, let's write a simple transfer operation that transfers a certain amount of money from one account to another without any concept of a transaction.

```
def transfer(transfer_amt, source_act, target_act):
    cursor = conn.cursor()
    cursor.execute("""
        update ACCOUNT
        set BALANCE = BALANCE - %s
        where ACCOUNT_NUM = %s""",
        (transfer_amt, source_act))
    cursor.execute("""
        update ACCOUNT
        set BALANCE = BALANCE + %s
        where ACCOUNT_NUM = %s""",
        (transfer_amt, target_act))
    cursor.close()
```

   Let's assume we are transferring $10,000 from the *SAVINGS* account to the *CHECKING* account. In our business logic, we accomplish this by first withdrawing $10,000 from the *SAVINGS* account and then depositing $10,000 into the *CHECKING* account.

   Imagine if there was some system error that happened after we had withdrawn from the *SAVINGS* account that caused the system to restart. Having never deposited the transfer amount into the *CHECKING* account, the bank would have leaked $10,000 to nowhere. How long would you keep your money in a bank like that? Probably not for long!

2. Let's combine this operation with the transaction pattern we looked at earlier, to give our bank some integrity.

```
def transfer(transfer_amt, source_act, target_act):
    conn.commit()
    try:
        cursor = conn.cursor()
        cursor.execute("""
            update ACCOUNT
            set BALANCE = BALANCE - %s
            where ACCOUNT_NUM = %s""",
```

```
            (transfer_amt, source_act))
        cursor.execute("""
            update ACCOUNT
            set BALANCE = BALANCE + %s
            where ACCOUNT_NUM = %s""",
            (transfer_amt, target_act))
        cursor.close()
        conn.commit()
    except:
        conn.rollback()
```

In our updated example, we now start with `conn.commit()` in order to start a new transaction.

If you use a Python DB API compatible database module, then transactions are available when you connect to the database. `connection.commit()` and `connection.rollback()` finishes an existing transaction and implicitly starts a new one (`http://www.pubbs.net/python/200901/18953/`). This is the reason this block of code starts with a `commit` statement.

We enter the `try/except` block and start executing the same business code that we wrote earlier. In the end, we execute `conn.commit()`, to commit our results to the database.

In the event of some system error, the `commit` would never be executed. This means the withdrawal from the *SAVINGS* account would not be written into the database. If this was really a hard system failure, relational databases would revert to the state before the transaction started.

For softer failures where some sort of application level exception raised would result in `conn.rollback()` being called, reverting all changes back to when the transaction started.

This solution solves the problem such that our bank doesn't leak money due to system faults and errors. But in order to re-use this transactional pattern in other parts of our banking application, we must repeat the coding pattern over and over, increasing the risk of bugs.

Transactions and their properties

Transactions carry four distinct properties:

- **Atomicity**—We need to guarantee that all steps complete, or no steps complete
- **Consistent**—This is concept of ensuring our data maintains a consistent state
- **Isolated**—When we withdraw money from the bank, it would be bad if our business partner was withdrawing at the same time, and we ended up with a negative balance
- **Durable**—We expect completed transactions to survive hardware failures

Consistency and durability tend to be related to resources, such as the database and the server it runs on, and typically doesn't affect the way we write code. Atomicity and Isolation however are commonly handled by developer code.

To guarantee atomicity, we must start a transaction, and then commit or rollback. Throughout this chapter we will explore how Spring Python makes it easy to define atomic transactions.

Python's DB API specification (`http://www.python.org/dev/peps/pep-0249/`) doesn't define Isolation levels. Instead, each vendor implements this differently. To alter Isolation levels, we must investigate the database engine we are using, and then access either the connection or the cursor provided with our factory in order to alter this setting from vendor defaults.

Transactions can either be supported locally or be distributed across multiple databases. Local transactions typically involved a single database schema. In order to extend to other systems, the Python specification utilizes a 'two-phase commit' mechanism.

At the time of writing, Spring Python only supports local transactions, but is open to the possibility of expanding its capabilities in the future.

Another factor in transaction definition is **propagation**. When we have multiple operations that are defining a transaction, it is important to combine them together in the right fashion. This involves dealing with circumstances where a new transaction is encountered while one is already in progress. We will look at this in more detail later in this chapter.

Getting transactions right is hard

The transactional pattern shown earlier above is very simplistic and incomplete. There are many issues that can occur which requires even more detail.

- The most important issue is whether or not the underlying database engine supports transactions, and if it's been properly configured to avoid auto commits. If this isn't set up appropriately, the code won't function as expected. Auto commits work by committing every SQL statement as executed, instead of waiting for `conn.commit()`, hence not offering any option to rollback.
- If a transaction was already in progress when the `transfer` is called, there may be special handling required. In our example, we abruptly commit previous work and then start a new transaction with the initial `commit`. With a transaction in progress, this may not be the right step. It's hard to tell considering we don't have any surrounding business context.
- Exception handling isn't the only type of rollback scenario to handle. While our transfer doesn't have any alternate return paths, more complex business logic can easily code guard clauses and other return statements that bypass both the `commit` and the `rollback`. Leaving the transaction hanging and not committed to the database would be very sloppy and risky.

The result is that the pattern shown above and utilized for our simple example isn't comprehensive enough to handle the simplest risks. A more complex pattern has to be coded. Given that it must be repeated for every transactional point in our application, it makes our integrity problem even harder to solve.

There is a side effect of efficiency when performing transactions. Because the same connection is used to conduct the transaction, the cost of opening and closing connections is avoided.

Simplify by using @transactional

Spring Python solves these problems with its `TransactionTemplate`. This utility class makes it easy to wrap business methods with transactional functionality that solves all of the problems listed earlier. Spring Python makes it easy to wrap our existing business functions with the `TransactionTemplate` using its `@transactional` decorator.

1. First, let's take our simple `transfer` function, and put into a `Bank` class.

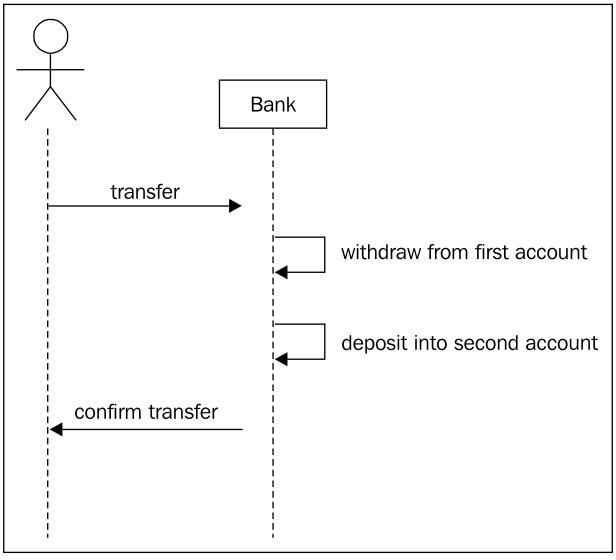
```
from springpython.database.core import *
from springpython.database.factory import *
```

```
class Bank(object):
    def __init__(self, connectionFactory):
        self.factory = connectionFactory
        self.dt = DatabaseTemplate(self.factory)

    def transfer(self, transfer_amt, source_act, target_act):
        self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE - %s
            where ACCOUNT_NUM = %s""",
            (transfer_amt, source_act))
        self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE + %s
            where ACCOUNT_NUM = %s""",
            (transfer_amt, target_act))
```

In this situation, we have stripped out all the hand-coded transaction code. Instead, we have the simple, concise business logic that defines a `transfer` operation.

Please note, this version of the Bank application is *NOT* yet safe.

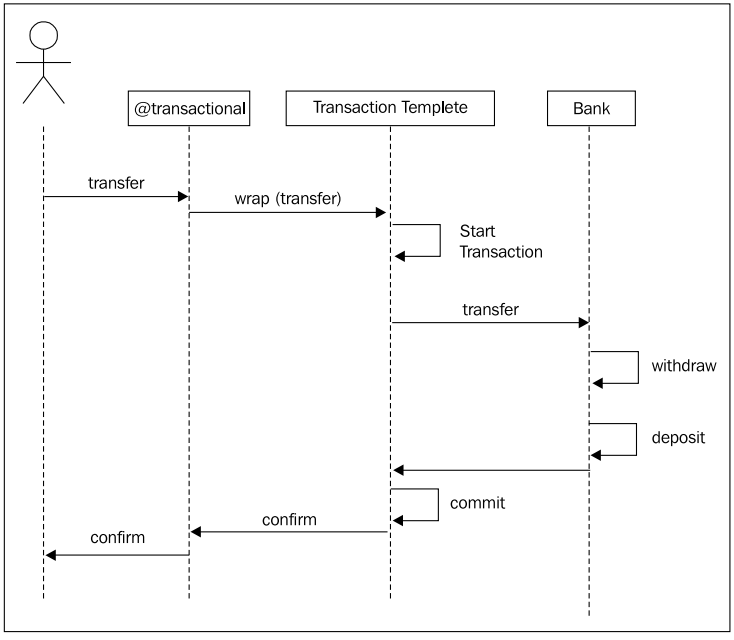
The steps are easily shown with the following sequence diagram:



2. Now let's wrap our `transfer` method with transaction protection by using Spring Python's `@transactional` decorator.

```
from springpython.database.core import *
from springpython.database.factory import *
from springpython.database.transaction import *

class Bank(object):
    def __init__(self, connectionFactory):
        self.factory = connectionFactory
        self.dt = DatabaseTemplate(self.factory)

    @transactional
    def transfer(self, transfer_amt, source_act, target_act):
        self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE - %s
            where ACCOUNT_NUM = %s""",
            (transfer_amt, source_act))
        self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE + %s
            where ACCOUNT_NUM = %s""",
            (transfer_amt, target_act))
```

`@transactional` is a decorator that uses a hidden instance of `TransactionTemplate` to execute our transfer function inside a very robust version of the transaction pattern. Our interactions are better shown in the following diagram.



- Because `@transactional` wraps our `transfer` function, when the user invokes `transfer`, they are hitting the decorator first
- `@transactional` passes all the context of the requested method call to its private instance of `TransactionTemplate`
- `TransactionTemplate` starts a transaction
- `TransactionTemplate` then calls the original `Bank` `transfer` function
- `Bank` carries out its business, totally unaware it is inside a transaction
- When complete, `Bank` hands control back to the `TransactionTemplate`, which issues a `commit`
- `TransactionTemplate` hands control back to `@transactional`, and finally back to the caller

This configuration totally decouples transactional logic from our transfer operation.

- Since `TransactionTemplate` is the caller of `transfer`, it can easily handle any number of return statements. If an exception is raised anywhere inside `transfer`, `TransactionTemplate` will `rollback` instead of `commit`.

With this clean separation of concerns, we can work on the business code without having to worry about getting transactions right.

More about TransactionTemplate

Our first version of the transaction pattern was simple and naïve. We then examined the list of issues with that pattern. `TransactionTemplate` has a much more sophisticated pattern that handles these extra situations:

- It handles the simple case of catching any exception thrown by catching it, issuing a rollback, and then re-throwing the exception
- It handles all return statements by catching the return value of the method, issuing the necessary commit, and then finally returning the results
- `@transactional` is coded by default to start new transactions if one isn't currently in progress, and to join a transaction if one already exists. We will look at the other transactional options later in this chapter

Another integrity gap exists in our `transfer` code. It lies somewhere between the transaction pattern and our business logic. Can you spot it?

There are no checks to make sure that we even have $10,000 to transfer! Also, there is no type of security check ensuring that we own either of these two accounts. We will address this deficiency later on, when filling in the transactional details.

We're not done yet. In order to have `@transactional` do its job, we need to link it with a **Transaction Manager** through an `AutoTransactionalObject`. The Transaction Manager provides `@transactional` with a handle into the database to issue necessary commits and rollbacks. It also tracks the context of existing transactions and make appropriate decisions about when to start new transactions.

1. To inject everything, let's define a pure Python IoC container that links `ConnectionFactoryTransactionManager` to `@transaction` through an `AutoTransactionalObject`.

```
from springpython.database.transaction import *
class BankAppConfig(PythonConfig):
    def __init__(self, factory):
        PythonConfig.__init__(self)
        self.factory = factory

    @Object
    def transactionalObject(self):
        return AutoTransactionalObject(self.tx_mgr())

    @Object
    def tx_mgr(self):
        return ConnectionFactoryTransactionManager(self.factory)

    @Object
    def bank(self):
        return Bank(self.factory)
```

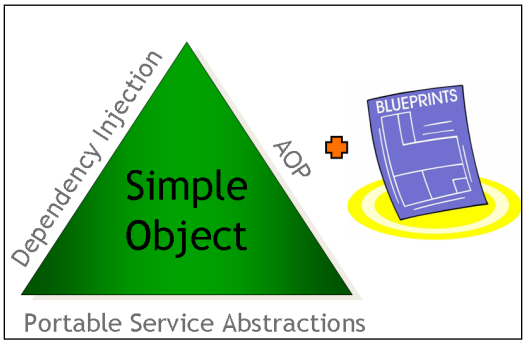
   - `tx_mgr` defines our Transaction Manager, which uses an injected factory in order to perform the SQL transaction APIs. This is the same type of factory used by `DatabaseTemplate`. `tx_mgr` tracks when transactions begin and end, providing the necessary services for `TransactionTemplate` and `@transactional`.
   - `transactionalObject` defines an instance of `AutoTransactionalObject`, an IoC *post processor*. Its job is to find all instances of `@transactional` and link them with the `tx_mgr`. This is what empowers `@transactional` to do its job of ensuring data integrity through SQL transactions.
   - The `bank` class is our business class.

2. Now, let's write some startup code to perform the `transfer`.

```
if __name__ == "__main__":
    from springpython.context import ApplicationContext
    ctx = ApplicationContext(BankAppConfig(
        Sqlite3ConnectionFactory("/path/to/sqlite3db")))
    service = ctx.get_object("bank")
    bank.transfer(10000.0, "SAVINGS", "CHECKING")
```

The Spring Triangle—Portable Service Abstractions

We saw this diagram earlier in the book, as an illustration of the key principles behind Spring Python.



`TransactionTemplate` represents a **Portable Service Abstraction**.

- It is **portable** because it uses Python's standardized API for SQL transactions, not tying us to any database vendor or custom database connection library
- It provides the useful **service** of letting us easily wrap methods with a sophisticated and powerful transaction pattern
- It offers a nice **abstraction** to writing transactional code, without requiring us to handle the SQL transaction APIs directly

Programmatic transactions

Our banking example has shown how to decorate some business logic with Spring Python's `@transactional` in order to make the operation transactional. Throughout the example, we have repeatedly mentioned `TransactionTemplate`.

In this section, we will use `TransactionTemplate` directly instead of `@transactional`. We will also explore how to do this with and without IoC configuration.

Configuring with the IoC container

1. First, let's rewrite `Bank`, replacing `@transactional` with `TransactionTemplate`.

```
class Bank(object):
    def __init__(self, connectionFactory):
        self.factory = connectionFactory:
        self.dt = DatabaseTemplate(self.factory)
        self.tx_mgr = ConnectionFactoryTransactionManager(self.
factory)
        self.tx_template = TransactionTemplate(self.tx_mgr)

    def transfer(self, transfer_amt, source_act, target_act):
        class TxDefinition(TransactionCallbackWithoutResult):
            def doInTransactionWithoutResult(s, status):
                self.dt.execute("""
                    update ACCOUNT
                    set BALANCE = BALANCE - %s
                    where ACCOUNT_NUM = %s""",
                    (transfer_amt, source_act))
                self.dt.execute("""
                    update ACCOUNT
                    set BALANCE = BALANCE + %s
                    where ACCOUNT_NUM = %s""",
                    (transfer_amt, target_act))
        self.tx_template.execute(TxDefinition())
```

   This version of our `Bank` shows two more attributes: `tx_mgr` and `tx_template`. These could be injected into our class, but we chose to inject the connection factory only.

Inside the `transfer` function, we have defined class `TxDefinition`. We then instantiate it when calling `tx_template.execute()`.

- This inner class defines one method: `doInTransactionWithoutResult`, which contains our business logic.
- To avoid confusion between `self` passed into `transfer` and `self` passed into `TxDefinition`, `doInTransactionWithoutResult` names its first argument `s`.
- Because transfer is not expected to return anything, it uses `TransactionCallbackWithoutResult` as a base class. For return values, use `TransactionCallback/doInTransaction` instead.

2. Next, let's adjust the IoC configuration to handle these changes.

```
from springpython.database.transaction import *
class BankAppConfig(PythonConfig):
    def __init__(self, factory):
        PythonConfig.__init__(self)
        self.factory = factory

    @Object
    def bank(self):
        return Bank(self.factory)
```

3. This should have no effect on the code used to run our app and execute the same transfer.

```
if __name__ == "__main__":
    from springpython.context import ApplicationContext
    ctx = ApplicationContext(BankAppConfig(
        Sqlite3ConnectionFactory("/path/to/sqlite3db")))
    service = ctx.get_object("bank")
    bank.transfer(10000.0, "SAVINGS", "CHECKING")
```

Configuring without the IoC container

In this situation, our IoC configuration is pretty simple. We could code the application without it. Just remember: IoC provides useful assistance in things like testing, mocking, and being able to swap out key objects.

We begin by rewriting the startup script, so that it doesn't need any IoC container.

```
if __name__ == "__main__":
    service = Bank(Sqlite3ConnectionFactory("/path/to/sqlite3db"))
    bank.transfer(10000.0, "SAVINGS", "CHECKING")
```

Because our `Bank` was written using simple constructor injection, there was no need to alter it in order to run it without a container. Since we are programmatically using `TransactionTemplate`, there is no requirement to use the IoC container. This offers developers an opportunity to evaluate Spring Python purely for the transactional features without having to try out the IoC container at the same time.

But it is important to remember that multiple examples of the value of IoC have already been shown in this book, and many more are coming.

@transactional versus programmatic

The Spring way includes giving developers options. In order to choose the right approach, here are some pros and cons:

`@transactional`	• Pros:
	  ◦ Defines transactions with a single line.
	  ◦ Clear, concise, easy-to-read.
	• Cons:
	  ◦ Requires IoC to wire `AutoTransactionalObject`.
	  ◦ Requires editing existing code.

programmatic	• Pros:
	  ◦ Explicitly shows the transactional behavior where it occurs.
	  ◦ Doesn't require IoC.
	• Cons:
	  ◦ Re-introduces code tangling.
	  ◦ Requires editing existing code.

If the cons for either of these solutions are not acceptable, there is a third choice: declaring transactions from inside the IoC container. This allows easy wrapping of business code with transactions *without code tangling* and *without editing already existing code*. We will demonstrate this later in the chapter. But first let's look at adding new functions.

Making new functions play nice with existing transactions

So far, we have managed to build a bank that does one thing: transfer money. As with any software project, we typically have to grow the functionality. As we proceed with modifications to our `Bank`, we want to add new transactions without risking existing ones. Spring Python's transaction management makes this very simple.

In this section, we will go back to our `@transactional` Bank and add some new functionality.

1. Let's extract a `withdraw` function and `deposit` function from the transfer function.

```
class Bank(object):
    def __init__(self, connectionFactory):
        self.factory = connectionFactory):
        self.dt = DatabaseTemplate(self.factory)

    def withdraw(self, amt, act):
        self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE - %s
            where ACCOUNT_NUM = %s""",
            (amt, act))

    def deposit(self, amt, act):
        self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE + %s
            where ACCOUNT_NUM = %s""",
            (amt, act))

    @transactional
    def transfer(self, transfer_amt, source_act, target_act):
        self.withdraw(transfer_amt, source_act)
        self.deposit(transfer_amt, target_act)
```

   By moving the two SQL statements into separate functions, we have nicely defined `transfer` as `withdraw` followed by `deposit`. We are now ready to offer these two new functions to our clients. Do you notice anything wrong with this? Are the functions safe transaction-wise by themselves? What if another banking operation tried to reuse these primitives?

2. Secure the `withdraw` and `deposit` methods with `@transactional`.

```
class Bank(object):
    def __init__(self, connectionFactory):
        self.factory = connectionFactory):
        self.dt = DatabaseTemplate(self.factory)

    @transactional
    def withdraw(self, amt, act):
        self.dt.execute("""
```

```
            update ACCOUNT
            set BALANCE = BALANCE - %s
            where ACCOUNT_NUM = %s""",
            (amt, act))

    @transactional
    def deposit(self, amt, act):
        self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE + %s
            where ACCOUNT_NUM = %s""",
            (amt, act))

    @transactional
    def transfer(self, transfer_amt, source_act, target_act):
        self.withdraw(transfer_amt, source_act)
        self.deposit(transfer_amt, target_act)
```

The only difference in our code is marking `withdraw` and `deposit` with `@transactional`.

3. Now let's add an extra layer of integrity to our `Bank` by doing some checks before and after executing the SQL.

```
class Bank(object):
    def __init__(self, connectionFactory):
        self.factory = connectionFactory):
        self.dt = DatabaseTemplate(self.factory)

    @transactional(["PROPAGATION_SUPPORTS"])
    def balance(self, act):
        return self.dt.queryForObject("""
            SELECT BALANCE
            FROM ACCOUNT
            WHERE ACCOUNT_NUM = ?""",
            (act,), types.FloatType)

    @transactional
    def withdraw(self, amt, act):
        if (self.balance(act) > amt):
            rows = self.dt.execute("""
                update ACCOUNT
                set BALANCE = BALANCE - %s
                where ACCOUNT_NUM = %s""",
                (amt, act))
            if (rows == 0):
```

```
                raise Exception("Account %s does not exist." % act)
            else:
                raise Exception("Account %s has insufficient funds." %
act)

    @transactional
    def deposit(self, amt, act):
        rows = self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE + %s
            where ACCOUNT_NUM = %s""",
            (amt, act))
        if (rows == 0) {
            raise Exception("Account %s does not exist." % act)

    @transactional
    def transfer(self, transfer_amt, source_act, target_act):
        self.withdraw(transfer_amt, source_act)
        self.deposit(transfer_amt, target_act)
```

We have made several changes that start to make our application look like a real bank. They are as follows:

- We created a `balance` function that allows us to look up the balance for an account
- The `withdraw` function now checks the balance to make sure there is enough to withdraw
- The `withdraw` function verifies that a row of data was updated, confirming the withdrawn account is real
- The `deposit` function verifies that a row of data was updated, confirming that the deposited account is real

How Spring Python lets us define a transaction's ACID properties

As discussed earlier, the ACID properties of transactions are as follows:

- **Atomicity**—We need to guarantee that all steps complete, or no steps complete.
- **Consistent**—This is the concept of ensuring our data maintains a consistent state.

- **Isolated**—When we withdraw money from the bank, it would be bad if our business partner was withdrawing at the same time, and we ended up with a negative balance.
- **Durable**—We expect completed transactions to survive hardware failures

Regarding atomicity, we have already practiced defined the beginning and end points for transactions by using `@transactional` and `TransactionTemplate`.

Spring Python supports *propagation*. Earlier, we stated that the default policy of `@transactional` is to start a new transaction (if none existed) and join an existing transaction (if one was already in progress). Spring Python conveniently lets us take the safe, atomic operations of `withdraw` and `deposit`, and combine them together into `transfer`, without having to interact with the SQL transaction APIs at all.

We also created another function, `balance`, to lookup the current balance of accounts. Since `balance` performs no updates, it doesn't require a transaction when run by itself. However, when called upon by an existing transaction, we want it to join in as if it was part of the transaction. This is accomplished by providing `@transactional` with a propagation override:

```
@transactional(["PROPAGATION_SUPPORTS"])
def balance(self, act):
    return self.dt.queryForObject("""
        SELECT BALANCE
        FROM ACCOUNT
        WHERE ACCOUNT_NUM = ?""",
        (act,), types.FloatType)
```

`@transactional` scans the list of transaction definitions. Currently, Spring Python supports the following definitions:

Property	Description
`PROPAGATION_REQUIRED`	A transaction is required. If a current one exists, join it. Otherwise, start a new one. This is the default for `@transactional`.
`PROPAGATION_SUPPORTS`	A transaction is not required. This code can run inside or outside a transaction.
`PROPAGATION_MANDATORY`	A transaction is required. If a current one exists, join it. Otherwise, raise an exception.
`PROPAGATION_NEVER`	A transaction is not allowed. If a current one exists, raise an exception. Otherwise, run the code.

Spring Python provides incredibly useful transaction context management, transaction API handling, and allows us clean demarcation of transactions.

As better definitions are added to Python's database specification for things like isolation, Spring Python will add more options to support it. This will increase our ability to cleanly declare the exact type of transaction needed to wrap our code.

Applying transactions to non-transactional code

An important aspect of Spring Python is its non-invasive nature. This was demonstrated in great detail in the chapter that introduced aspect oriented programming. Spring Python provides a convenient, non-intrusive method interceptor that allows the demarcation of existing code.

This solves the problem mentioned earlier, where neither editing existing source code nor tangling our business logic with transaction management are acceptable.

1. Let's start with an alternative version of `Bank` class that has no transaction demarcation.

```
class Bank(object):
    def __init__(self, connectionFactory):
        self.factory = connectionFactory:
        self.dt = DatabaseTemplate(self.factory)

    def balance(self, act):
        return self.dt.queryForObject("""
            SELECT BALANCE
            FROM ACCOUNT
            WHERE ACCOUNT_NUM = ?""",
            (act,), types.FloatType)

    def withdraw(self, amt, act):
        if (self.balance(act) > amt):
            rows = self.dt.execute("""
                update ACCOUNT
                set BALANCE = BALANCE - %s
                where ACCOUNT_NUM = %s""",
                (amt, act))
            if (rows == 0):
               raise Exception("Account %s does not exist." % act)
        else:
```

```
            raise Exception("Account %s has insufficient funds." % act)

    def deposit(self, amt, act):
        rows = self.dt.execute("""
            update ACCOUNT
            set BALANCE = BALANCE + %s
            where ACCOUNT_NUM = %s""",
            (amt, act))
        if (rows == 0) {
            raise Exception("Account %s does not exist." % act)

    def transfer(self, transfer_amt, source_act, target_act):
        self.withdraw(transfer_amt, source_act)
        self.deposit(transfer_amt, target_act)
```

This `Bank` class is identical to the previous one, except for the fact that there are no `@transactional` decorators.

2. Let's define an application context that has equivalent transaction demarcation points.

```
class BankAppConfig(PythonConfig):
    def __init__(self, factory):
        PythonConfig.__init__(self)
        self.factory = factory

    @Object
    def bank_target(self):
        return Bank(self.factory)

    @Object
    def tx_mgr(self):
        return ConnectionFactoryTransactionManager(self.factory)

    @Object
    def bank(self):
        tx_attrs = []
        tx_attrs.append((".*transfer", ["PROPAGATION_REQUIRED"]))
        tx_attrs.append((".*withdraw", ["PROPAGATION_REQUIRED"]))
        tx_attrs.append((".*deposit", ["PROPAGATION_REQUIRED"]))
        tx_attrs.append((".*balance", ["PROPAGATION_SUPPORTS"]))
        return TransactionProxyFactoryObject(self.tx_mgr(),
                                             self.bank_target(),
                                             tx_attrs)
```

With this alternative configuration, we use `TransactionProxyFactoryObject`. This is an out-of-the-box AOP interceptor that Spring Python offers to automatically wrap certain functions with `TransactionTemplate`. It requires a transaction manager as well as the target object, our `Bank`. It also needs a list of tuples, with each tuple defining a regular expression for method matching as well as a list of transaction properties just like we plugged into `@transactional` earlier in this chapter.

What is the right choice: `@transactional` or `TransactionProxyFactoryObject`?

- `@transactional` is clear, concise, and easy-to-read. Its biggest drawback is the requirement to edit existing code. If we own the code, then this shouldn't be a problem.
- If we are trying to wrap transactions around a 3rd party library that we don't directly control, `TransactionProxyFactoryObject` is the best choice.

Testing your transactions

Transactions are intrinsically tied to databases. Attempting to mock or stub this out would require an extreme amount of effort, and probably not be worth the effort. This is one area where I generally agree with testing against an actual database.

It is possible to use lightweight databases such as `sqlite` for this effort, but it may be risky if this isn't the target platform for production. In fact, the best testing effort would be a properly setup test bed using the same version of database engine as production. The important point is that it is easy to create lightweight tests against something small such as `sqlite`. This can confirm to the developers that things are working as expected. More extensive integration testing can be done with a production grade test-bed.

Easily swapping out different database configurations is one of the major advantages of using Spring Python.

Summary

In this chapter we utilized Spring Python's convenient transactional features in order to turn some simple banking SQL into a resilient application. Being able to pick between the easy-to-read `@transactional` decorator and an AOP-based `TransactionProxyFactoryObject` gives us flexible choices.

In this chapter we have learned that:

- The classic transaction issue makes coding transactions by hand difficult
- Spring Python lets us easily add transactions to a banking application using the `@transactional` decorator
- We can easily code transactions programmatically with and without the IoC container, giving us the maximum in choices
- With the power of AOP, we can non-intrusively mix in transactions to non-transactional code without editing the source

In the next chapter, we will explore ways to secure, web applications using Spring Python security module.

6
Securing your Application with Spring Python

With the rise of the web over the last ten years, many companies have adopted e-commerce solutions to support their business models. Retail, sales, banking, financial, and other industries have adopted the web as a key means to generate revenue. This has triggered a security crisis, since early web applications had little to no security, and the ways to exploit systems were vast. This doesn't involve a small corner of the market, but potentially compromises a huge segment of the market. It is no wonder that companies hire security consultants to come up with the means to protect their already built e-commerce sites.

Software development teams are starting to realize that security needs to be coded into their applications sooner rather than later. However, competitive deadlines and getting products to market sooner rather than later can cause security requirements to get pushed to the back of the line. Many web applications that hit the market have security implemented as more of an after thought. Part of this is due to the fact that coding effective security protocols is hard. **Acegi Security**, a Java framework based on the Spring Framework, was initially released in 2003. Its pluggable architecture and non-intrusive nature took the Java world by storm. By providing support for many security protocols including database, LDAP, OpenID, X.509, **Central Authentication Service (CAS)**, Kerberos, **Java Authentication and Authorization Service (JAAS)**, along with many others, it has become widely used in many industries, and in both the private and public sector.

Spring Python's pythonic implementation of this powerful architecture provides the same mechanisms to secure applications of all types simply and effectively to the Python community. Spring Python Security currently supports web application security. There are future plans to support method-level security just like Spring Security.

In this chapter, we will learn:

- The security problems software developers have to deal with and the challenge faced in effectively coding security
- The requirements for an effective security solution, and the ability of Spring Python Security to meet them
- Wrapping an unsecured web application with a simple solution that cleanly protects by delegating to a security handler
- The concept of authenticating who the user is, and determining what they are authorized to do
- Testing the security of our application
- Configuring a SQL-based security system, including adapting to a custom user/role schema
- Configuring an LDAP-based security system
- Making your application support multiple user communities or migrating from one security system to another with no downtime
- Coding our own security extension for systems not yet supported out-of-the-box by Spring Python Security

Problems with coding security by hand

Securing an application is hard. When coded by hand using simple tactics, security becomes very invasive. For applications to have true access to security settings, the following must be available:

- Securing URLs based on primitive rules is a start, but is rarely adequate as business rules and requirements are revised and updated over time.
- Relying on container security tends to be inflexible and prone to lock-in. For example, using Apache web server `.htaccess` files may work for simple situations. But complex rules are hard to get right, difficult to test automatically, and also discourage relocating to another type of container. Making security a part of the application, and not dependent on another container frees the application from container lock-in.
- To support specialized situations, any method in the code must be able to lookup who the current user is, and what permissions he or she has. Usually this is only needed in a few places, but altering all the necessary APIs to get this information passed from the logon screen to the code logic can have too wide an impact over such an isolated need.

- Security code has a repeating pattern. Developers sometimes abstract this behavior into a base class, with subclasses handling business logic. This may appear to support the **DRY** (**Don't Repeat Yourself**) principle, but it violates the **"is-a"** concept of OOP. It is important to realize that security is a crosscutting behavior that is independent of and orthogonal to the business logic, and must be solved using aspect-oriented concepts.
- When coded by hand, the solution is often hard-wired into the application. Upgrading from a simple username/password management system to something more sophisticated like OpenID, LDAP, or two-factor authentication systems can become impossible due to the ripple effect of changes. This type of switch is often needed when moving users from one security solution to another, or when supporting multiple user communities. Most production systems would prefer to stay where they are rather than spend the money to re-write the security layer of their application.

These issues shine a light on what it takes to implement a reliable security solution:

- The security solution must be orthogonal to the class hierarchy.
- Credential data and other security APIs must be available non-intrusively, to avoid requiring applications to re-write existing APIs.
- Usage of security by the application must be decoupled from the actual securing resource. For example, if the system uses LDAP to store username and password information, the application shouldn't have to make LDAP calls.
- Multiple security providers must be allowed, in order to support users from different communities as well as the ability to transition from one system to another while new credentials are issued to the user community.
- Security policies must be flexible and easy to fine tune, in order to support up-and-coming business requirements.
- Even though there are many standard conventions, users must be able to quickly write custom security extensions to support legacy security solutions.

The Spring Python Security module meets all these requirements. Throughout this chapter, as we develop our example web application and then secure it, we will point out how these requirements are being met.

Building web applications ignoring security

We need a simple application. For this example, we will use CherryPy (`http://cherrypy.org`), a pythonic web framework that conveniently maps URLs into Python methods.

If you want to know more about the CherryPy framework, I highly recommend reading *CherryPy Essentials* by *Sylvain Hellegouarch*.

After building our web application, we will review some of the options people take in securing things. Then, we will plug in Spring Python Security, showing how easy it is to lock down an application.

First, let's build a simple web application that serves wiki pages, and allows us to edit them.

```
import cherrypy

def forward(url):
    return '<META HTTP-EQUIV="Refresh" CONTENT="0; URL=' + url + '">'

class Springwiki(object):
    def __init__(self, controller = None):
        self.controller = controller

    @cherrypy.expose
    def index(self, article="Main Page"):
        page = self.controller.getPage(article)
        return page.html()

    @cherrypy.expose
    def default(self, article="Main Page"):
        return self.index(article)

    @cherrypy.expose
    def submit(self, article, wpTextbox=None, wpSummary=None):
        self.controller.updatePage(article, wpTextbox,
                                   wpSummary)
        return forward("/" + article)

    @cherrypy.expose
    def edit(self, article):
        page = self.controller.getEditPage(article)
        return page.html()
```

This CherryPy application nicely maps URLs onto Python methods that are marked with the `@cherrypy.expose` decorator. HTTP GET/POST parameters are translated into named arguments. Method `index` maps to the web path `/`, `submit` maps to `/submit`, and `edit` maps to `/edit`. As a CherryPy application, we aren't done yet. After filling in some more details, we'll show to how to host this application inside a CherryPy web container.

In this case, our application has an injected controller which is responsible for handling updates to the wiki as well as returning components that generate the HTML content sent back to the browser.

Let's code a simple controller that processes these web requests. For now, let's use a simple Python dictionary as the place to store the information. Later on, we can migrate it to a database server.

```
import time
from model import EditPage
from model import NoPage
from model import Page

wiki_db = {
    "Main Page":["""
Welcome to the Spring Python book's wiki!
""", [("Original", "Initial entry", "13:22, 24 November 2009")]]
    }

class SpringWikiController(object):
    def exists(self, article):
        return article in wiki_db

    def getPage(self, article):
        if self.exists(article):
            return Page(article=article,
                        wikitext=wiki_db[article][0],
                        controller=self)
        else:
            return NoPage(article=article,
                          controller=self)

    def getEditPage(self, article):
        if self.exists(article):
            return EditPage(article=article,
                            wikitext=wiki_db[article][0],
                            controller=self)
        else:
```

```
            return EditPage(article=article,
                            wikitext="",
                            controller=self)

    def create_edit_tuple(self, text, summary):
        return (text,
                summary,
                time.strftime("%H:%M:%S %d %b %Y", time.localtime()))

    def updatePage(self, article, wikitext, summary):
        if self.exists(article):
            wiki_db[article][1].append(
                self.create_edit_tuple(wiki_db[article][0],summary))
            wiki_db[article][0] = wikitext
        else:
            wiki_db[article] = [None, None]
            wiki_db[article][1] = [
                self.create_edit_tuple(wikitext,summary)]
            wiki_db[article][0] = wikitext
```

There is a method to retrieve a page. It checks whether or not the article exists. If not, it returns a specialized page that will support creating a new one. Otherwise, it returns a normal page with the retrieved wiki text.

There is another method to return an edit page, used to edit existing article entries.

Finally, there is a function to update a current page with new wiki text. If you'll notice, when the wiki text is updated, an entry record is created in the form of a tuple, and stored in the tail end list of the article's entry.

Let's define some simple model objects to pass around our application. First, we need a basic representation of a page with some basic wiki formatting rules.

```
import re

intrawikiR = re.compile("\[\[(?P<link>.*?)(\|(?P<desc>.*?))?\]\]")
externalLinkR = re.compile("\[(?P<link>.*?)\s(?P<description>.*)\]")

class Page(object):
    def __init__(self, article, wikitext, controller):
        self.article = article
        self.wikitext = wikitext
        self.controller = controller

    def link_substitution(self, match):
        g = match.groupdict()
```

```
            if self.controller.exists(g["link"]):
                str = '<a href="' + g["link"] + '">'
            else:
                str = '<a href="edit/' + g["link"] + '">'

            if g["desc"]:
                str += g["desc"]
            else:
                str += g["link"]

            str += "</a>"
            return str

        def header(self):
            """Standard header used for all pages"""
            return """
                <html>
                <head>
                <title>Spring Python book demo</title>
                </head>

                <body>
                    <h1>""" + self.article + """</h1>
                """

        def footer(self):
            """Standard footer used for all pages."""
            footer = """
                <ul>
                  <li><a href="/edit/""" + self.article + """">Edit</a></
li>
                </ul>
                <a href="http://springpythonbook.com">Spring Python book</
a>
             </body>
                """
            return footer

        def wiki_to_html(self):
            htmlText = self.wikitext
            try:
                htmlText = intrawikiR.sub('<a href="\g<link>">\g<desc></
a>', htmlText)
            except:
```

```
            htmlText = intrawikiR.sub('<a href="\g<link>">\g<link></
a>', htmlText)
        htmlText = externalLinkR.sub('<a href="\g<link>">\
g<description></a>', htmlText)
        return htmlText

    def html(self):
        results = self.header()
        results += """
            <!-- BEGIN main content -->
        """
        results += self.wiki_to_html()
        results += """
            <!-- END main content -->
        """
        results += self.footer()
        return results
```

`Page` holds the title of the article, the wiki text associated with it, and a handle on the controller. Most of the code in this class is used to support `html()`, a function used to render this page in an HTML format.

Most web frameworks encourage usage of templates to avoid embedding HTML in the application. This helps decouple the information shown on the pages from the format it is displayed in. Since this chapter's focus is on security and not clean **Model-View-Controller** (**MVC**) tactics, the HTML is embedded directly into the application.

Our wiki page basically generates three parts: a header, some wiki text converted into HTML, followed by a `footer()`. The wiki text rules are simple for our example:

- It supports the free linking style of Wikipedia, where `[[article to link to]]` maps to local web path `/article_to_link_to`. You can also insert a pipe followed by an alternate block of text to be displayed `[[article to link|my alt text]]`.
- External links `[http://springpythonbook.com Spring Python book site]` map to the full URL, with the text after the first space being displayed on the page

While it would be easy to implement more wiki format rules, we want to stick with demonstrating security for our web application.

Next we need the representation of an edit page.

```
class EditPage(Page):
    def __init__(self, article, wikitext, controller):
        Page.__init__(self, article, wikitext, controller)

    def header(self):
        """Standard header used for all pages"""
        return """
            <html>
            <head>
            <title>Spring Python book demo</title>
            </head>

            <body>
                <h1>Editing """ + self.article + """</h1>
            """

    def wiki_to_html(self):
        htmlText = """
            <form method="post"
                    action="/submit?article=""" + \
                    self.article + """ enctype="multipart/form-data">
                <textarea name="wpTextbox"
                    rows='25' cols='80'>""" + \
                    self.wikitext + """</textarea>
                <br/>
                Summary: <input type='text' value=""
                    name="wpSummary" maxlength='200' size='60'/>
                <br/>
                <input type='submit' value="Save page"
                    title="Save your changes"/>
                <em><a href="/""" + self.article + """ title='""" + \
                        self.article + """'>Cancel</a></em>
            </form>
            """
        return htmlText
```

An `EditPage` is really a `Page` with some special rendering. The `header()` block inserts **Editing** in front of the article name. The main block of HTML shows a form with the wiki text, summary input box, and a **Save** button.

Let's create the final object modeling a non-existent page.

```
class NoPage(Page):
    def __init__(self, article, controller):
        Page.__init__(self, article,
                      "This page does not yet exist.", controller)
```

`NoPage` is a `Page` with some hard-coded wiki text on it. It inherits the same format rules as `Page`, and also has an edit button, so the user can replace the non-existent page with a real one.

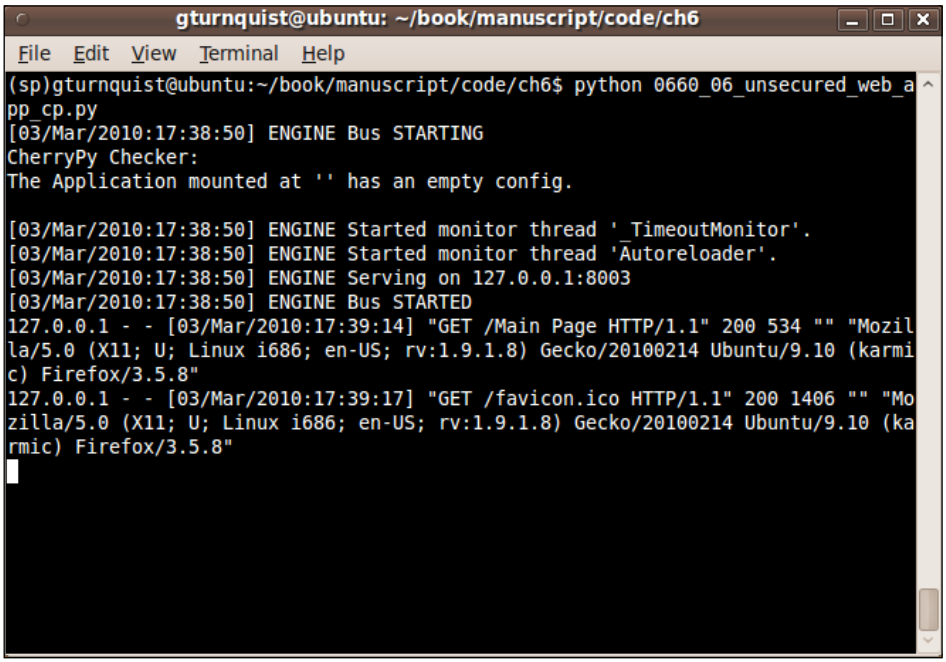
Let's create a pure Python application context to wire these components together.

```
import controller
import view
from springpython.config import PythonConfig
from springpython.config import Object

class SpringWikiAppContext(PythonConfig):
    def __init__(self):
        super(SpringWikiAppContext, self).__init__()

    @Object
    def view(self):
        return view.Springwiki(self.controller())

    @Object
    def controller(self):
        return controller.SpringWikiController()
```

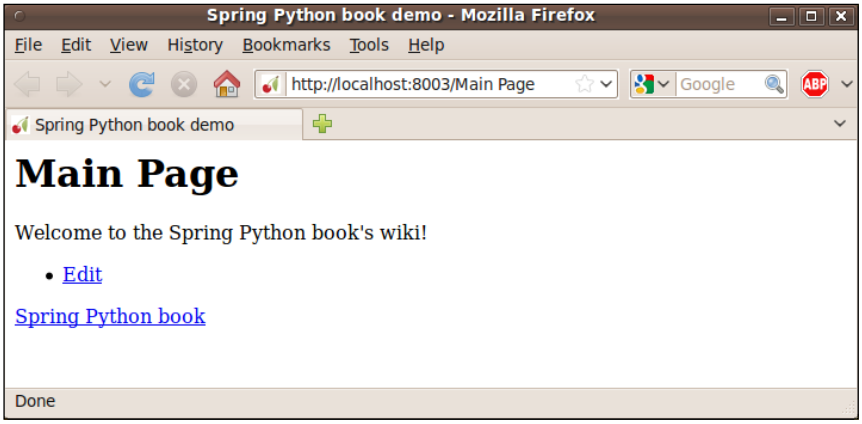
With all the parts coded up, let's write a CherryPy server application to host our web application.

```
import cherrypy
import os
import noxml
from springpython.context import ApplicationContext

if __name__ == '__main__':
    cherrypy.config.update({'server.socket_port': 8003})

    applicationContext = ApplicationContext(noxml.
SpringWikiAppContext())
    cherrypy.tree.mount(
        applicationContext.get_object("view"),
        '/',
        config=None)

    cherrypy.engine.start()
    cherrypy.engine.block()
```

Assuming our controller code is in `controller.py`, the view code is in `view.py`, and the application context is in `noxml.py`, we should be able to fire up our CherryPy application.

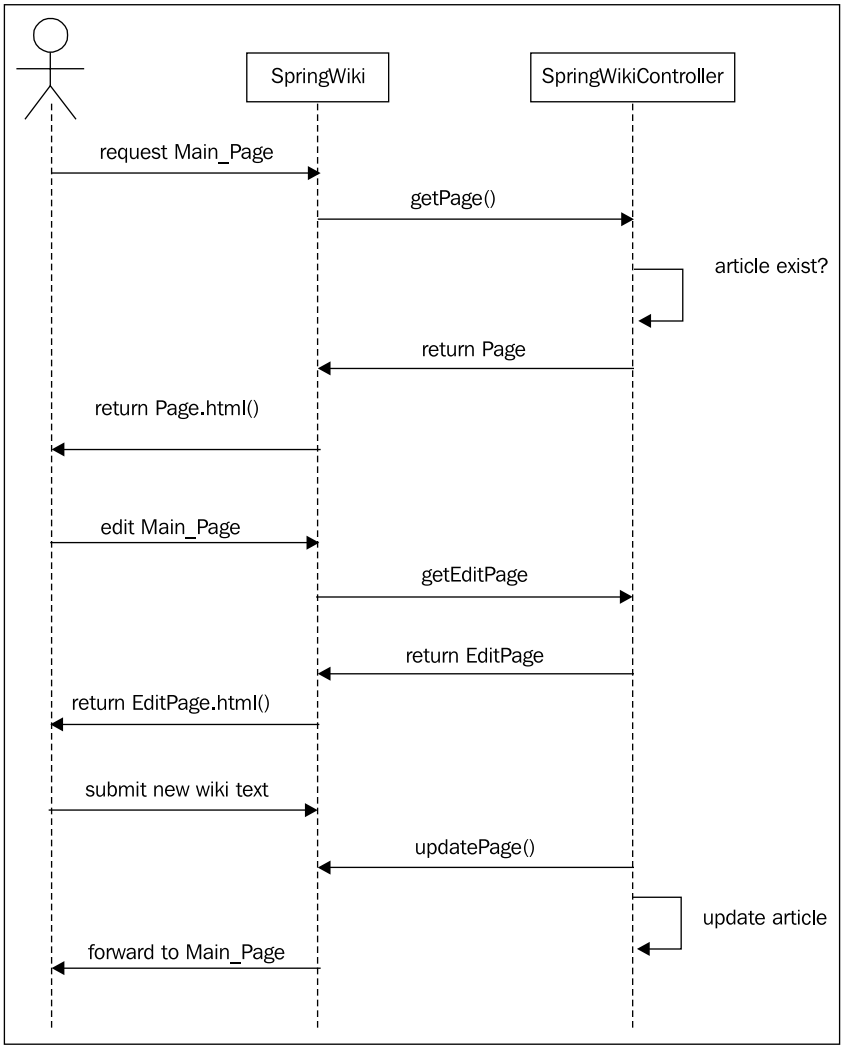


CherryPy runs its own web server, and in this case we have configured it to run on port `8003`. It then creates an instance of our application context. Next, it grabs the `view` object, and mounts it with the web path `/`. Finally, it starts up the CherryPy engine and blocks for any web requests. Open up a browser and point at `http://localhost:8003`, and you will see the results.



Looking at our web application from 10,000 feet

The following diagram shows a high level view of our web application as a user looks up an article, clicks on the **Edit** button, makes changes, and then submits an update. By clicking on the **Save** button, the new wiki text is posted to Spring Wiki.



In our example application, we have coded a simplistic database: a Python dictionary. However, that could easily be replaced by a relational database hosted on a separate server. The concept is the same: save the changes submitted by the user.

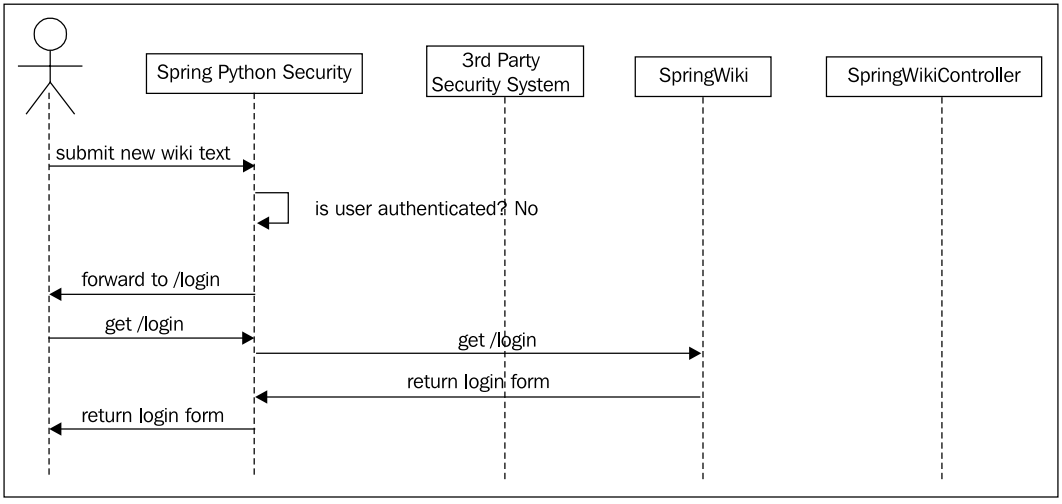
If we started out developing this application on the premise that anyone can read *and* edit any article, this solution works. It is simple, easy to understand, and probably the only security needed would be a firewall to protect our servers from external attack.

Handling new security requirements

We have doubtlessly learned that new requirements are always coming, and we have to be ready to implement them quickly. We started our application by implementing the core functionality. Now we need to support multiple users and need to control who edits what pages.

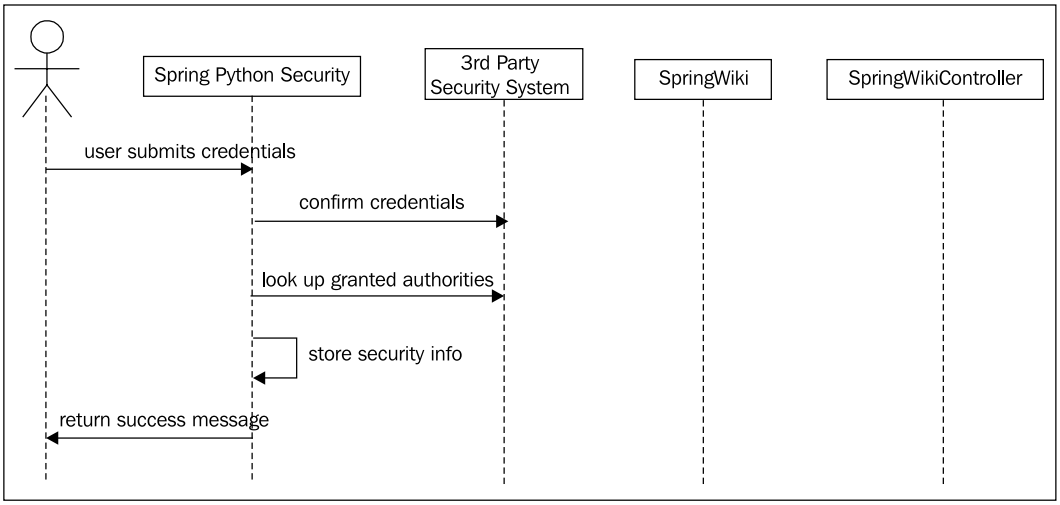
Authentication confirms "who you are"

The first step towards securing an application is **authenticating the user**. This identifies **who the user is**, and is usually accomplished by the user providing a username and a password. The following diagram shows an update to our application's design. It includes a new component not found in the previous diagram: a **Spring Python Security agent** that polices every web request combined with a 3rd party security resource. Before allowing the user to actually touch our wiki application or the database, our security agent checks if the user has been authenticated. If not, Spring Python redirects the user to a login page. In the diagram, we pick up at the point where the user submits new wiki text.



The security agent meets one of our earlier requirements: application of security must be decoupled from the application. The agent is a separate piece of code charged with the responsibility of managing security between the user and our application. Its configuration its not part of our wiki application, but instead configured using the IoC container. We will see how to configure this agent shortly.

The following diagram shows the user submitting credentials. The security agent checks the credentials against the 3rd party security resource to confirm the user's identity. The most common implementation is a simple database table containing user data.

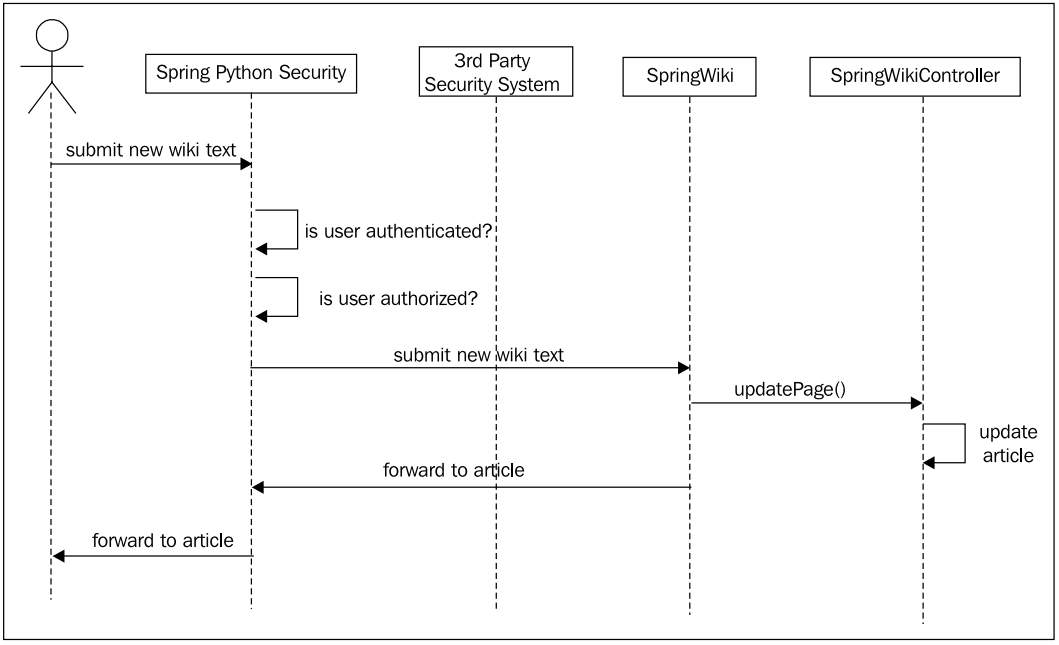


Authorization confirms "what you can do"

It is important to point out that our security agent is not done. After confirming the user's identity, Spring Python next looks up what authorities the user is granted. One user may have the authority to read articles, while another has the authority to read articles and also edit them. The last few steps of the previous diagram shows this as authorities are looked up and stored in Spring Python Security's context holder.

The standard convention is to allow a user to have zero or more roles.
Having collected this information, our security agent stores the user's identity, granted authorities, and authenticated status in a globally accessible `SecurityContextHolder`. This is important, because it provides a way for our application to look up information about the user's security status without modifying the application's APIs. This meets another one of our requirements: credential data must be available non-intrusively. After securing our application, we will see how this is available.

After authenticating and authorizing the user, the web request is allowed through to the web application. The following diagram shows what happens when we again submit our new wiki text, assuming we are authenticated and authorized to do so.



It is important to point out that this requires no changes in either SpringWiki or `SpringWikiController`. We will quickly demonstrate that applying these changes involves only some configuration changes.

Time to add security to our application

Now it's time to add these security features to our application.

First we need to define the Spring Python Security agent. To do this, we need to add the following code to our `SpringWikiAppContext`.

```
@Object
def filterChainProxy(self):
    return CP3FilterChainProxy(filterInvocationDefinitionSource =
        [
        ("/login.*",  ["httpSessionContextIntegrationFilter"]),
        ("/.*",       ["httpSessionContextIntegrationFilter",
                       "exception_translation_filter",
                       "auth_processing_filter",
                       "filter_security_interceptor"])
        ])
```

`springpython.security.cherrypy3.CP3FilterChainProxy` uses CherryPy APIs to insert itself into the chain of handlers that are called before our CherryPy application is called. This allows the filter chain proxy to apply the security policies we are configuring.

The filter chain proxy is configured with a list of URL patterns. On every web request, the proxy will iterate over this list until it finds a match.

When a match is found, it applies the defined chain of filters before allowing access to the web application itself. Each filter is configured as a string used to reflectively look up the actual filter in the application context.

If a match is not found, the entire security stack is bypassed. This is not the recommended solution. Instead, it is best to cover all the possible patterns by having a catch-all pattern (`/.*`) to clearly show what filters are applied where.

Let's add the first filter needed to the application context: `httpSessionContextIntegrationFilter`.

```
@Object
def httpSessionContextIntegrationFilter(self):
    filter = HttpSessionContextIntegrationFilter()
    filter.sessionStrategy = self.session_strategy()
    return filter

@Object
def session_strategy(self):
    return CP3SessionStrategy()
```

First, note that the method name matches the filter's string name found in `filterChainProxy`.

`springpython.security.web.HttpSessionContextIntegrationFilter` is used to transfer security credentials between the user's HTTP session and Spring Python's `SecurityContextHolder`. The rest of the Spring Python Security components use `SecurityContextHolder` to access the user's credential information.

In our application's configuration, this filter is used when accessing the login page (which we haven't coded yet), so that the login page can store credential info. The rest of the application has several more filters.

In order for Spring Python to support different web frameworks, the mechanism to interact with HTTP session data is encapsulated inside `cherrypySessionStrategy()`. In our case, we use `CP3SessionStrategy`.

Let's add the `exception_translation_filter` to manage any security exceptions thrown by other filters or the web application itself.

```
@Object
def exception_translation_filter(self):
    filter = ExceptionTranslationFilter()
    filter.authenticationEntryPoint = self.auth_filter_entry_pt()
    filter.accessDeniedHandler = self.accessDeniedHandler()
    return filter

@Object
def auth_filter_entry_pt(self):
    filter = AuthenticationProcessingFilterEntryPoint()
    filter.loginFormUrl = "/login"
    filter.redirectStrategy = self.redirectStrategy()
    return filter

@Object
def accessDeniedHandler(self):
    handler = SimpleAccessDeniedHandler()
    handler.errorPage = "/accessDenied"
    handler.redirectStrategy = self.redirectStrategy()
    return handler

@Object
def redirectStrategy(self):
    return CP3RedirectStrategy()
```

This filter is responsible for redirecting the user to the right page in the event of a security exception. If the user has not been authenticated yet, he is redirected to the `login` page. If he is trying to access a page without the appropriate authorization, he is redirected to the `accessDenied` page.

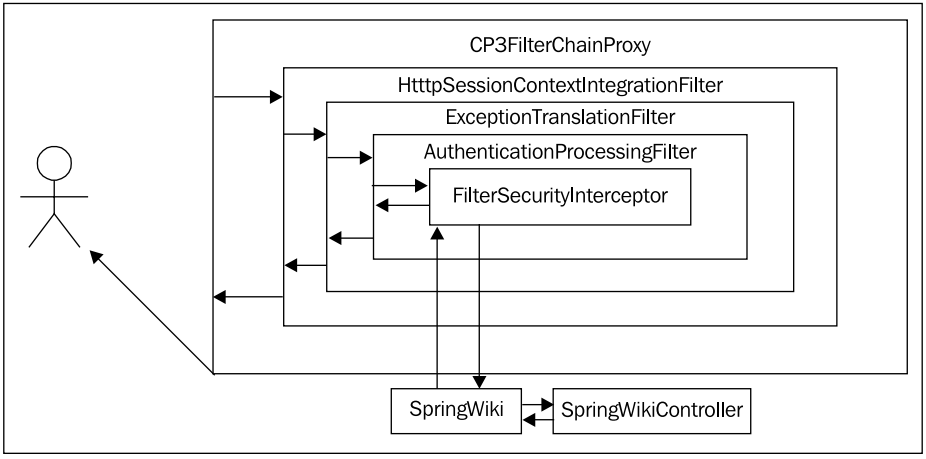
In order for Spring Python to support different web frameworks, the mechanism to issue a web redirect is encapsulated inside `redirectStrategy()`. In our case, we uses `CP3RedirectStrategy`.

Let's add the `authenticationProcessingFilter` to confirm a user is authenticated before proceeding.

```
@Object
def auth_processing_filter(self):
    filter = AuthenticationProcessingFilter()
    filter.auth_manager = self.auth_manager()
    filter.alwaysReauthenticate = False
    return filter
```

This filter's job is to confirm the user is authenticated. If not, a security exception is thrown, and the `exception_translation_filter` handles the outcome. At this stage, it is possible to configure the system to re-authenticate on every web request, or to avoid consuming as many resources by caching the authentication status.

The following diagram shows how the filters are nested together. A call into this stack of filters gives each filter a chance to perform security functions on entry and/or exit. It also shows how the entire Spring Python Security stack is neatly staged between the caller and the application, without having to intertwine itself into the application itself through either class hierarchy or meddling with the application's API.



Let's define the `AuthenticationManager` that manages the lookup of user credentials. For our situation, we will define a fixed list of users for test purposes.

```
@Object
def auth_manager(self):
    auth_manager = AuthenticationManager()
    auth_manager.auth_providers = [self.auth_provider()]
    return auth_manager

@Object
def auth_provider(self):
    provider = DaoAuthenticationProvider()
    provider.user_details_service = self.user_details_service()
    provider.password_encoder = PlaintextPasswordEncoder()
    return provider

@Object
```

```
    def user_details_service(self):
        user_details_service = InMemoryUserDetailsService()
        user_details_service.user_dict = {
            "alice": ("alicespassword",["ROLE_READ", "ROLE_EDIT"],
True),
            "bob": ("bobspassword", ["ROLE_READ"], True)
            }
        return user_details_service
```

`auth_manager` references a list of authentication providers that are used to check credentials. When an authentication request is submitted, `AuthenticationManager` goes down this list, asking each one to attempt authentication until one of them succeeds. If no provider succeeds, `AuthenticationManager` throw a security exception, denying access.

Right now, we have only one provider defined, but it is possible to have more. This meets another one of our key requirements: multiple security providers must be allowed.

Our provider, `DaoAuthenticationProvider`, taps its `user_details_service` in order to lookup the user. It then runs the user-supplied password through its `password_encoder` and compares it to the stored password. In our situation, we are using an `InMemoryUserDetailsService` instead of an actual database, and the password is stored in the clear instead of hashed.

It is highly recommended to *not* store passwords in the clear on any production system.

As you can see, we currently have two user accounts defined:

- `alice` has two defined roles: `ROLE_READ` and `ROLE_EDIT`. `alice` is an enabled account
- `bob` has one defined role: `ROLE_READ`. `bob` is an enabled account

Using the `InMemoryUserDetailsService` makes it easy to get our application up and running from a security perspective, since we don't have to worry about integrating with any external systems.

With all the components nicely separated, we can later swap `InMemoryUserDetailsService` with `DatabaseUserDetailsService` when we are ready to go online. We can also replace `PlaintextPasswordEncoder` with either `ShaPasswordEncoder` or `Md5PasswordEncoder`, to easily support different hashing algorithms.

Let's define the last filter needed in our chain: `filter_security_interceptor`.

```
@Object
def filter_security_interceptor(self):
    filter = FilterSecurityInterceptor()
    filter.auth_manager = self.auth_manager()
    filter.access_decision_mgr = self.access_decision_mgr()
    filter.sessionStrategy = self.session_strategy()
    filter.obj_def_source = [
        ("edit.*", ["ROLE_EDIT"]),
        ("/.*", ["ROLE_READ"])
        ]
    return filter

@Object
def access_decision_mgr(self):
    access_decision_mgr = AffirmativeBased()
    access_decision_mgr.allow_if_all_abstain = False
    access_decision_mgr.access_decision_voters = [RoleVoter()]
    return access_decision_mgr
```

`filter_security_interceptor` is the last security check in our app. Its job is to make sure the user is authorized to complete the request. It looks at the URL of the web request, and then goes down its list of URL patterns until it finds a match. When it does, it sees a list of roles. The roles along with the user's credentials are given to the `access_decision_mgr`, who submits a request for a vote from each of its `access_decision_voters`. In our configuration, we are using `RoleVoter`, which votes on whether or not the user has the role supplied in the list.

`access_decision_mgr` tallies up the votes, and decides if access is granted based on the policy. In our case, we have an `AffirmativeBased` policy, meaning that we need only one up vote, i.e. we only need to have one of the roles in the list. If we switched to a `UnanimousBased` policy, we would be required to have all the roles. A `ConsensusBased` policy would require that we have a majority of the roles.

This policy indicates that a user can only access `/edit*` URLs if he has `ROLE_EDIT`. For any other web path, the user must have `ROLE_READ`. Note: don't forget that `/login*` is excluded from this filter.

- Let's add a login page to our view layer

```
    @cherrypy.expose
    def login(self, fromPage="/", login="", password="",
errorMsg=""):
        return self.controller.getLoginPage(
            fromPage, login, password, errorMgr)
```

- Let's add the code to create and process a login page in our controller

```
    def getLoginPage(self,fromPage="/
",login="",password="",errorMsg=""):
        if login != "" and password != "":
            try:
                self.authenticate(login, password)
                return [self.redirector.redirect(fromPage)]
            except AuthenticationException, e:
                return [self.redirector.redirect(
                  "?login=%s&errorMsg=Username/password failure" %
                    login)]

        results = """
<html>
    <head>
      <title>Spring Python book demo</title>
    </head>

    <body>
      <form method="POST" action="">
       Login: <input type="text" name="login" value="%s"
size="10"/><br/>
       Password: <input type="password" name="password" size="10"/
><br/>
       <input type="hidden" name="fromPage" value="%s"/><br/>
       <input type="submit"/>
      </form>
      <a href="http://springpythonbook.com">Spring Python book</a>
     </body>
</html>
        """ % (login, fromPage)
```

```
            return [results]

        def authenticate(self, username, password):
            token = UsernamePasswordAuthenticationToken(username,
password)
            SecurityContextHolder.getContext().authentication =
                      self.auth_manager.authenticate(token)
            self.httpContextFilter.saveContext()
```

- We also need to initialize the controller with some of the security components

```
    def __init__(self):
        self.httpContextFilter = None
        self.auth_manager = None
        self.redirector = None
```

- These controller attributes must be injected from the IoC container

```
    @Object
    def controller(self):
        ctrl = controller.SpringWikiController()
        ctrl.httpContextFilter =
            self.httpSessionContextIntegrationFilter()
        ctrl.auth_manager = self.auth_manager()
        ctrl.redirector = self.redirectStrategy()
        return ctrl
```

With these parts injected into the controller, it now has the ability to process a login request, store it into the context, and have the user's authenticated credentials stored in their HTTP session data.

With all these modifications in place, Spring Python has clearly made it possible to wrap an existing application with a layer of security. The IoC definitions show fine grained control over what URLs require which filters, and also which authorization roles. This meets the important requirement: 'the security solution must be orthogonal to the class hierarchy'. At no time were we required to extend any security-based classes. Instead, we used a series of filters and security classes, all defined outside the hierarchy of our business model.

By keeping the policies in the centralized location of the application context, it is easy to adjust them as necessary. For example, if we needed a `/admin.*` pattern linked with `ROLE_ADMIN`, it is easy to adjust the `access_decision_manager` to easily support this. We can also put various URL patterns underneath different policies, all from within the application context. This supports the key requirement: 'security policies must be flexible and easy to fine tune'.

Accessing security data from within the app

We briefly mentioned how Spring Python Security stores user credentials in the `SecurityContextHolder`. This provides an easy way to lookup important security information from within our application without having to alter our APIs.

So far, we have managed to develop a relatively simple web application, and then wrap it with a layer of security that protects URLs based on roles. There may be situations where that isn't enough. While protection of URLs is nice, it may be useful to disable or hide some links based on the user's role power. This is a more fine grained option, and is easy to implement.

In the definition of the Page object that is used to render html, the footer definition contains this:

```
def footer(self):
    """Standard footer used for all pages."""
    footer = """
        <ul>
          <li><a href="/edit/""" + self.article + """">Edit
              </a></li>
        </ul>
        <a href="http://springpythonbook.com">Spring Python book
        </a>
     </body>
        """
    return footer
```

In this situation, when viewing an article, we present the user with option to click on the edit link. However, if the user doesn't have `ROLE_EDIT`, the request will redirect him to an access denied page.

The preferred solution would be to hide this link so they don't click on it in the first place. Reducing the opportunities for users to wander into access denied pages not only reduces the demand on the security layer, but also improves the user experience. But our current security solution we have put in doesn't deal with conditionally altering HTML. Writing a filter to manage this isn't pragmatic. Instead, it is best to put some conditional checks right here to check user credentials, and optionally render the hyperlink.

```
def footer(self):
    """Standard footer used for all pages."""
    footer = ""
```

```
        if "ROLE_EDIT" in SecurityContextHolder.getContext().
                                            authentication.granted_auths:
            footer += """
            <ul>
              <li><a href="/edit/""" + self.article + """">Edit
                  </a></li>
            </ul>
            """

        footer += """
            <a href="http://springpythonbook.com">Spring Python book
            </a>
         </body>
            """
        return footer
```

`SecurityContextHolder` is a globally accessible object. It contains a context, which specifies where the current authentication credentials are being stored.

- If `SecurityContextHolder` is configured with mode `"GLOBAL"`, then there is a single context for the entire Python VM, meaning all threads will see the same security credentials
- If `SecurityContextHolder` is configured with mode `"THREADLOCAL"`, then the context is stored in `threading.local()`, meaning there is a separate context for each thread. This is useful for multi-threaded server apps where a separate thread exists for each user

By the time this code is actually executed, `SecurityContextHolder` will have been populated with user data, allowing us to do a quick check and only offer this link of the user has the necessary role, without altering our application API. This satisfies the requirement: 'credential data and other security APIs must be available non-intrusively'.

Testing application security

Testing security has traditionally been a challenge, because there are many facets to cover. Integrating with a 3rd party authentication system is one part that needs to be checked out. Making sure the application is using the security system is also needed. And confirming that both good and bad accounts are handled properly is important. Often, we tend to code successful scenarios. Its important to know that our software responds to failing scenarios as well. While this is important for general coding, it is especially critical to test failing security scenarios. If the system doesn't properly handle invalid or expired credentials, what is the point in securing the system?

By using the `InMemoryUserDetailsService` in an alternative application context, it is easy to create a whole host of user accounts that have all the permutations of roles, privileges, good and bad passwords needed to checkout the services. CherryPy is especially easy to test, because it doesn't require running the web container. Since CherryPy conveniently maps URLs to methods, it is easy enough to call the methods directly from a test harness rather than verify the web handling machinery of CherryPy. And swapping the data access layer with either stubs or mocks makes it easy to isolate business logic.

All these things help to build a high confidence automated test suite. With the core of our application easily subjected to automated testing, it now becomes easy to decide whether or not to use an automated web container test kit for the web layer. It is also an option to test the 3rd party security system through automated integration testing.

By removing the need to code security APIs by hand inside our business logic and also decoupling the web layer, it becomes much easier to automate testing, leading to a more powerful system that meets expectations.

Configuring SQL-based security

We configured our wiki application with startup security by using `InMemoryUserDetailsService`. It's easy to upgrade to database-managed security by simply swapping that component out with `DatabaseUserDetailsService`.

In the `SpringWikiAppContext`, we just need to change `user_details_service` to:

```
@Object
def user_details_service(self):
    service = DatabaseUserDetailsService()
    service.dataSource = self.factory()
    return service
```

In this example, we have left out the coding of `factory()`. This is simply a connection factory used to create a `DatabaseTemplate` to talk to the database, as covered in the earlier SQL chapter.

`DatabaseUserDetailsService` defines the following queries to look up user data and granted authorities:

- To lookup users: `SELECT username,password,enabled FROM users WHERE username = ?`
- To lookup authorities: `SELECT username,authority FROM authorities WHERE username = ?`

Even though these are the default settings, it is easy to support another schema. Just override the variables as shown below.

```
@Object
def user_details_service(self):
    service = DatabaseUserDetailsService()
    service.dataSource = self.factory()
    service.DEF_USERS_BY_USERNAME_QUERY = "<alt user qry>"
    service.DEF_AUTHORITIES_BY_USERNAME_QUERY = "<alt auth qry>"
    return service
```

Configuring LDAP-based security

LDAP is a popular protocol used to host user accounts and group associations as well. Spring Python provides a convenient `LdapAuthenticationProvider` with flexible options to support conventional schemas as well as custom ones.

At the time of writing, Spring Python LDAP only works on CPython and not on Jython.

`LdapAuthenticationProvider` has two components:

- A `BindAuthenticator` and a `PasswordComparisonAuthenticator` to perform authentication
- A `DefaultLdapAuthoritiesPopulator` to lookup granted authorities based on conventional LDAP group associations

LDAP supports binding, where the server confirms the user's password. `BindAuthenticator` utilizes this feature to confirm the user's password.

It is also possible to fetch the password from the directory server, and do a password comparison after properly hashing the user's password. `PasswordComparisonAuthenticator` is used to perform this operation.

After confirming a user's credentials, the second step of `LdapAuthenticationProvider` is to search the directory for groups the user is associated with.

The following code shows an application context where `LdapAuthenticationProvider` is set up with a `BindAuthenticator` and an authorities populator.

```
from springpython.config import PythonConfig
from springpython.config import Object

from springpython.security.providers.Ldap import *

class LdapAppContext(PythonConfig):
    def __init__(self):
        super(LdapAppContext, self).__init__()

    @Object
    def auth_manager(self):
        auth_manager = AuthenticationManager()
        auth_manager.auth_providers = [self.auth_provider()]
        return auth_manager

    @Object
    def auth_provider(self):
        provider = LdapAuthenticationProvider()
        provider.ldap_authenticator = self.authenticator()
        provider.ldap_authorities_populator =
            self.authorities_populator()
        return provider

    @Object
    def context(self):
        return DefaultSpringSecurityContextSource(
            url="ldap://localhost:53389/dc=springpythonbook,dc=com")


    @Object
    def authenticator(self):
        return BindAuthenticator(
            context_source=self.context(),
            user_dn_patterns="uid={0},ou=people")

    @Object
    def authorities_populator(self):
        return DefaultLdapAuthoritiesPopulator(
            context_source=self.context(),
            group_search_base="ou=groups")
```

- All the LDAP components are pointed at one directory server, with the format `ldap://<server>:<port>/<baseDN>`
- `BindAuthenticator` is configured to search recursively from the `<baseDN>` with the pattern `uid={0},ou=people`, where `{0}` is where the supplied username will be substituted. When the dn is found, a bind is performed using the supplied password
- `DefaultLdapAuthoritiesPopulator` will search one level below baseDN at `ou=groups`, looking for any entries that have a `member` attribute pointed at the user's `dn`. It is possible to override `group_search_filter` with something else like `uniqueMember={0}`, if the directory schema is different. By default, the name of the group fetched will be the group's `cn`. Overriding `group_role_attr` allows picking another attribute. By default, the group name will be converted to upper case, and prefixed with `ROLE_`. These can be overridden with `role_prefix` and `convert_to_upper`

  To do a password check instead, replace `BindAuthenticator` with `PasswordComparisonAuthenticator`
- By default it supports both plain text passwords as well as SHA hashed ones that are base64 encoded. This can be overridden by supplying an alternative password encoder to `encoder` attribute
- It has the same constructor arguments as `BindAuthenticator`, meaning you can specify `user_dn_patterns`
- By default, it looks for the password stored in the `userPassword` attribute. You can override this by setting the `password_attr_name` attribute

If you have a non-conventional usage of either password authentication, or group management, you can plug in your own authenticator or authorities populator. I ran into this problem myself when a legacy system stored the roles in a custom attribute inside the user's record instead of a separate group.

With all the options to override the defaults, it should be pretty easy to get LDAP security management underway.

Using multiple security providers is easy

Supporting multiple security providers is a valuable feature. There are several common use cases where this is quite beneficial.

- Migrating from an old security solution to a new one—This is where started with security provider, but need to migrate to another one. For example, we started with user accounts stored in the database, but want to move to a two-factor authentication system.

- Supporting multiple user communities—This is where two different sites running on two different servers are merged to run on a single server. Instead of maintaining two different applications, we want to consolidate the data but maintain existing roles and accounts.
- Providing redundant security access—This is when we need to have multiple security providers to handle failures and maintenance windows.

Let's explore these scenarios in more detail.

Migrating from an old security solution to a new one

One very handy use case is where you want to migrate users from an old login system to a new one. For example, your project may have several tools that have all been using their own custom database solution. However, you finally want to centralize everyone's details in a common LDAP server. With both systems keyed up, as new users are issued new credentials, they can log in with no interruption of service. Once everyone has been migrated, you can remove the old security provider.

It also makes it easy to have redundant security providers, such as having more than one LDAP URL to point at.

Earlier, we saw the configuration for an `AuthenticationManager` that looked like this:

```
@Object
def auth_manager(self):
    auth_manager = AuthenticationManager()
    auth_manager.auth_providers = [self.auth_provider()]
    return auth_manager
```

`AuthenticationManager` supports a list of providers. It will iterate over each one until a successful match is made. To support the migration use case, it can be altered to look like this:

```
@Object
def auth_manager(self):
    auth_manager = AuthenticationManager()
    auth_manager.auth_providers = [
        self.old_auth_provider(),
        self.new_auth_provider()]
    return auth_manager
```

With this setup, our system will first try to use the old authentication provider. However, if it fails, Spring Python Security will try the new authentication provider.

Supporting multiple user communities

It is possible our application may have to support multiple users. Imagine we had developed our application and had complete Spring Python Security configuration as well, handling users based on a `DatabaseUserDetailsService`. Some time after that, our company acquires another company, bringing on board many new employees that need access to the system. It might be too much effort to migrate them into the new system. Or at least, it might be too expensive. Instead, it would be much easier to configure another authentication provider to point at their current system. The only other feature needed would be to add the extra roles from the new users to the access decision manager.

The solution would look much like the previous use case. The company can now easily decide whether to continue in this fashion, or start a migration of the new users into the already existing system as shown in the previous use case.

Providing redundant security access

We just recently walked through configuring security access to an LDAP server. In the current version of Spring Python, it only supports one URL. To handle two LDAP servers in a redundant solution, all you have to do is configure two instances of each and then plug them in to the `AuthenticationManager`.

```
from springpython.config import PythonConfig
from springpython.config import Object

from springpython.security.providers.Ldap import *

class LdapAppContext(PythonConfig):
    def __init__(self):
        super(LdapAppContext, self).__init__()

    @Object
    def auth_manager(self):
        auth_manager = AuthenticationManager()
        auth_manager.auth_providers = [
            self.auth_provider1(),
            self.auth_provider2()]
        return auth_manager

    @Object
```

```
    def auth_provider1(self):
        provider = LdapAuthenticationProvider()
        provider.ldap_authenticator = self.authenticator1()
        provider.ldap_authorities_populator =
            self.authorities_populator1()
        return provider

     @Object
    def auth_provider2(self):
        provider = LdapAuthenticationProvider()
        provider.ldap_authenticator = self.authenticator2()
        provider.ldap_authorities_populator =
            self.authorities_populator2()
        return provider

   @Object
    def context1(self):
        return DefaultSpringSecurityContextSource(
            url="ldap://server1:53389/dc=springpythonbook,dc=com")
                @Object
    def context2(self):
        return DefaultSpringSecurityContextSource(
            url="ldap://server2:53389/dc=springpythonbook,dc=com")

    @Object
    def authenticator1(self):
        return BindAuthenticator(
            context_source=self.context1(),
            user_dn_patterns="uid={0},ou=people")

    @Object
    def authenticator2(self):
        return BindAuthenticator(
            context_source=self.context2(),
            user_dn_patterns="uid={0},ou=people")

    @Object
    def authorities_populator1(self):
        return DefaultLdapAuthoritiesPopulator(
            context_source=self.context1(),
            group_search_base="ou=groups")

    @Object
    def authorities_populator2(self):
```

```
        return DefaultLdapAuthoritiesPopulator(
            context_source=self.context2(),
            group_search_base="ou=groups")
```

While it is possible Spring Python will eventually support multiple URLs, the ability to combine existing components gives you an already working solution right now.

Coding our own security extension

While all the features shown in this chapter so far demonstrate a powerful security framework, arguably the most important feature is the ability to write a custom security extension that plugs into this architecture.

Earlier in the LDAP section, it was mentioned that it is possible to adjust where searches are conducted to find groups related to the users. In the SQL section, it was shown that custom queries can be injected. Both of these features allow flexible adjustments, however, they can only bend so far. If our security system is highly specialized, it may be easier to simply code our own extension, and plug it in.

This is also necessary if another security system is not yet supported by Spring Python, such as two-factor tokens, OpenID, or X.509 certificates. While support for these may appear in the future, our needs may not be able to wait.

Coding a custom authentication provider

For our example, let's write a custom authentication provider that is based on logging in to a database instead of doing a password comparison.

This example matches a real situation I had to deal with. I inherited an application that was based on logging the user into the database and then looking up roles. This confined passwords to whatever the database supported, and in turn not allowing us to have a unified policy for acceptable passwords. The first step to migrate that application off of this limited solution required a custom authentication provider just like this example. It allowed me to utilize this framework, which provided the ability to plug in another security provider pointed at our new user security data.

In order to code our custom security provider, we need to write an `authenticate` method with the following signature:

```
def authenticate(self, authentication)
```

Authentication providers are expected to either succeed or fail with the following behavior:

- If authentication is *successful*, return an authentication object. It is recommended to create a new instance of `UsernamePasswordAuthenticationToken`, copy in the `username`, `password`, and `granted_auths`, and then return it. This is the same class used by all of Spring Python Security's providers. We do *not* have to call `setAuthenticated(true)` on the token. `AuthenticationManager` handles this for us
- If authentication *fails*, raise a `BadCredentialsException`

For this provider, we can use `DatabaseTemplate` to do most of our work. To do that, we need an injected SQL connection factory. If we set the username and password with the values supplied by our user, we can create a `DatabaseTemplate` and attempt a query to retrieve the user's roles. Since our instance of `DatabaseTemplate` will try to login to the database before doing the query, this should serve as our authentication check. If successful, we will receive the granted authorities in a result set. If it fails, we will get a SQL exception, which we can catch and replace with a `BadCredentialsException`.

```
from springpython.security.providers import *
from springpython.database.core import *

class MySqlLoginAuthenticationProvider(AuthenticationProvider):

    def __init__(self, factory=None):
        self.factory = factory

    def authenticate(self, authentication):
        self.factory.username = authentication.username
        self.factory.password = authentication.password
        dt = DatabaseTemplate(self.factory)

        try:
            authorities =
             dt.query("select authority from user where username = ?",
                           (authentication.username,),
                           rowhandler=DictionaryRowMapper())

            return UsernamePasswordAuthenticationToken(
                authentication.username,
                authentication.getCredentials(),
                [auth["authority"] for auth in authorities])
        except:
            raise BadCredentialsException("Invalid credentials")
```

Because each connection factory is slightly different, we will assume this is a `MySqlConnectionFactory` being passed in. Our new provider is ready for immediate use. It's that easy, meeting our requirement: 'users must be able to quickly write custom security extensions to handle legacy security solutions'.

One way to make this example more sophisticated would be to make it handle the current set of connection factories. But for now, it works as a suitable demonstration of how easy it is to authenticate against an existing system in a way not currently covered by Spring Python Security.

Some of the challenges with Spring Python Security

Spring Python Security—as demonstrated throughout this chapter—is extremely flexible, highly extensible, and non-invasive. However, there is a cost. Everything must be configured by hand (and also configured correctly).

This was one of the major criticisms levied against Acegi Security. Since then, Acegi Security (now branded as Spring Security) has made several updated releases, reducing the amount of manual configuration required. They now provide short-cuts to many industry standard security options, while still allowing us to add customization options, making it much easier to wire up the security framework for our needs.

As Spring Python Security develops, it will be making similar changes while also increasing the protocols it supports. Until then, it is good practice to use as much automated testing as possible to make sure the configuration is performing as expected. And for issues and documentation, be sure to make use of:

- Online reference manual (`http://springpython.webfactional.com`)
- Support forum (`http://forum.springsource.org/forumdisplay.php?f=45`)
- Mailing list (`http://lists.springsource.com/archives/springpython-users`) to get up-to-date answers

Summary

In this chapter, we have gone into the details of the complex, yet powerful Spring Python Security architecture, and seen how flexible and configurable it is. We looked at the challenges of coding security into an application and came up with a list of requirements for a useful security solution. Throughout the chapter, we saw how Spring Python Security met those requirements.

We developed a simple application, and then applied simple, hard-coded testable security. Later on, we saw how it was possible to easily swap this out with a SQL-driven solution thanks to Spring Python's IoC container. We also explored configuring an LDAP solution.

We finished be seeing how to easily add our own custom security extensions that easily plug in to Spring Python Security.

We also frankly observed that using Spring Python Security takes some careful configuration. Hopefully this book has lowered the bar to make it more accessible.

In this chapter, we covered:

- Security problems software developers have to deal with and effectively coding solutions is very challenging
- There are many requirements involved with building a security framework, and Spring Python meets them all
- We wrapped an unsecured application with a simple solution that cleanly protects the app by delegating to a security controller
- The concept of authenticating who the user is, and determining what they are authorized to do
- Testing the security of our application is possible, practical, and necessary
- We configured a SQL-based security system, including adapting to a custom user/role schema
- We configured an LDAP-based security system
- We explored making our application support multiple user communities or migrating from one security system to another with no downtime
- We looked at how to code our own security extension for systems not yet supported out-of-the-box by Spring Python Security

In the next chapter, we will explore ways to connect systems together over remote links using the Spring way.

7

Scaling your Application Across Nodes with Spring Python's Remoting

With the explosion of the Internet into e-commerce in recent years, companies are under pressure to support lots of simultaneous customers. With users wanting richer interfaces that perform more business functions, this constantly leads to a need for more computing power than ever before, regardless of being web-based or thick-client. Seeing the slowdown of growth in total CPU horsepower, people are looking to multi-core CPUs, 64-bit chips, or at adding more servers to their enterprise in order to meet their growing needs.

Developers face the challenge of designing applications in the simple environment of a desktop scaled back for cost savings. Then they must be able to deploy into multi-core, multi-server environments in order to meet their companies business demands.

Different technologies have been developed in order to support this. Different protocols have been drafted to help communicate between nodes. The debate rages on as to whether talking across the network should be visible in the API or abstracted away. Different technologies to support remotely connecting client process with server processes is under constant development.

Spring Python offers a clean cut way to take simple applications and split them up between multiple machines using remoting techniques that can be seamlessly injected without causing code rewrite headaches. Spring Python makes it easy to utilize existing technologies, while also being prepared to support ones not yet designed.

In this chapter we will learn how:

- Pyro provides a nice Python-to-Python remoting capability to easily create client-server applications
- Spring Python seamlessly integrates with Pyro so that your application doesn't have to learn the API
- You can convert a simple application into a distributed one, all on the same machine
- It takes little effort to rewire an application by splitting it up into parts, plugging in a round-robin queue manager, and running multiple copies of the server with no impact to our business logic

Introduction to Pyro (Python Remote Objects)

Pyro is an open source project (`http://pyro.sourceforge.net`) that provides an object oriented form of RPC. As stated on the project's site, it resembles Java's **Remote Method Invocation (RMI)**. It is less similar to **CORBA** (`http://ww.corba.org`), a technology-neutral wire protocol used to link multiple processes together, because it doesn't require an interface definition language, nor is oriented towards linking different languages together. Pyro supports Python-to-Python communications. Thanks to the power of Jython, it is easy to link Java-to-Python, and vice versa.

Python Remote Objects is not to be confused with the Python Robotics open source project (also named Pyro).

Pyro is very easy to use out of the box with existing Python applications. The ability to publish services isn't hard to add to existing applications. Pyro uses its own protocol for RPC communication.

Fundamentally, a Pyro-based application involves launching a Pyro daemon thread and then registering your server component with this thread. From that point on, the thread along with your server code is in stand-by mode, waiting to process client calls. The next step involves creating a Pyro client proxy that is configured to find the daemon thread, and then forward client calls to the server. From a high level perspective, this is similar to what Java RMI and CORBA offer. However, thanks to the dynamic nature of Python, the configuration steps are much easier, and there are no requirements to extend any classes or implement any interfaces.

As simple as it is to use Pyro, there is still the requirement to write some minimal code to instantiate your objects and then register them. You must also code up the clients, making them aware of Pyro as well. Since the intent of this chapter is to dive into using Spring Python, we will skip writing a pure Pyro application. Instead, let's see how to use Spring Python's out-of-the-box Pyro-based components, eliminating the need to write any Pyro glue code. This lets us delegate everything to our IoC container so that it can do all the integration steps by itself. This reduces the cost of making our application distributed to zero.

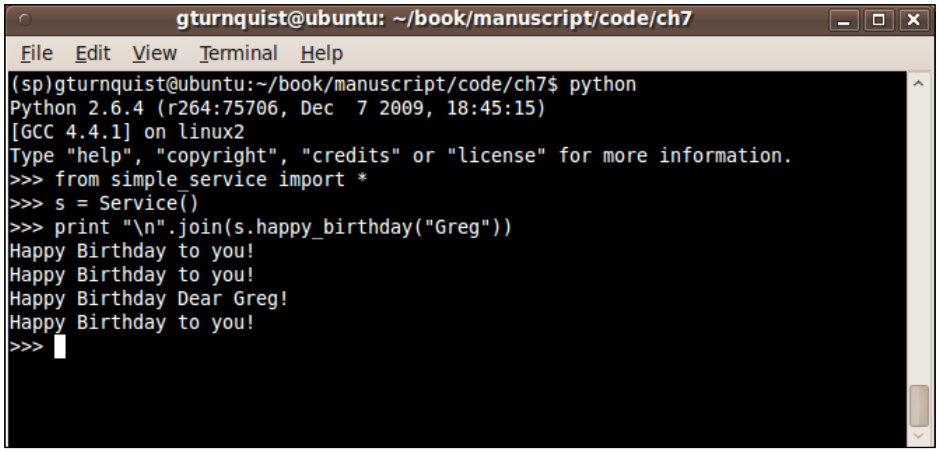
Converting a simple application into a distributed one on the same machine

For this example, let's develop a simple service that processes some data and produces a response. Then, we'll convert it to a distributed service.

First, let's create a simple service. For this example, let's create one that returns us an array of strings representing the Happy Birthday song with someone's name embedded in it.

```
class Service(object):
    def happy_birthday(self, name):
        results = []
        for i in range(4):
          if i == 2:
              results.append("Happy Birthday Dear %s!" % name)
          else:
              results.append("Happy Birthday to you!")
              return results
```

Our service isn't too elaborate. Instead of printing the data directly to screen, it collects it together and returns it to the caller. This allows us the caller to print it, test it, store it, or do whatever it wants with the result. In the following screen text, we see a simple client taking the results and printing them with a little formatting inside the Python shell.



As we can see, we have defined a simple service, and can call it directly. In our case, we are simply joining the list together with a newline character, and printing it to the screen.

Fetching the service from an IoC container

Let's define a simple IoC container that will create an instance of our service.

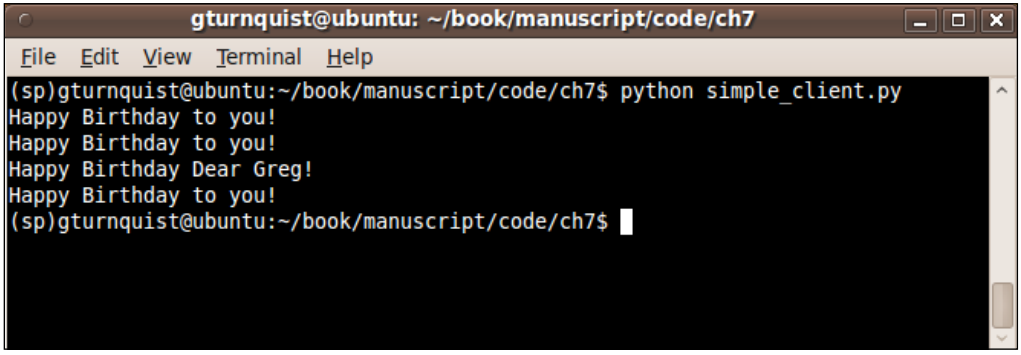
```
from springpython.config import *

from simple_service import *

class HappyBirthdayContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def service(self):
        return Service()
```

Creating a client to call the service

Now let's write a client script that will create an instance of this IoC container, fetch the service, and use it.

```
from springpython.context import *

from simple_service_ctx import *

if __name__ == "__main__":
    ctx = ApplicationContext(HappyBirthdayContext())
    s = ctx.get_object("service")
    print "\n".join(s.happy_birthday("Greg"))
```

Running this client script neatly creates an instance of our IoC container, fetches the service, and calls it with the same arguments shown earlier.



Making our application distributed without changing the client

To make these changes, we are going to split up the application context into two different classes: one with the parts for the server and one with parts for the client.

While there is no change to the API, we do have to slightly modify the original client script, so that it imports our altered context file.

First, let's publish our service using Pyro by making some small changes to the IoC configuration.

```
from springpython.config import *
from springpython.remoting.pyro import *

from simple_service import *


class HappyBirthdayContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def target_service(self):
        return Service()

    @Object()
    def service_exporter(self):
        exporter = PyroServiceExporter()
        exporter.service_name = "service"
        exporter.service = self.target_service()
        return exporter
```

We have renamed the `service` method to `target_service`, to indicate it is the target of our export proxy.

We then created a `PyroServiceExporter`. This is Spring Python's out-of-the-box solution for advertising any Python service using Pyro. It handles all the details of starting up Pyro daemon threads and registering our service.

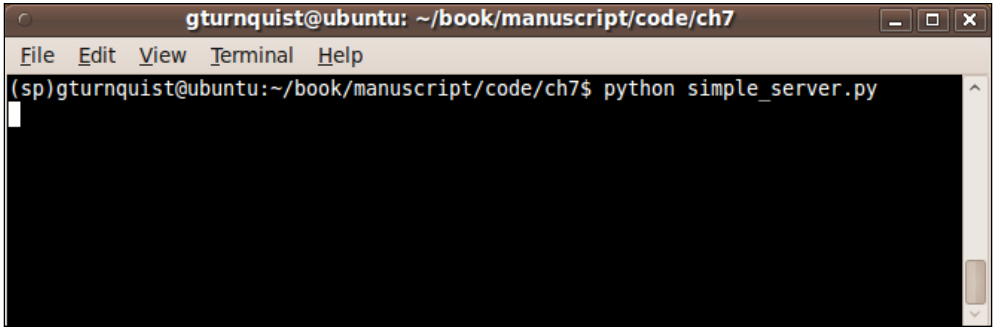
Pyro requires services to be registered with a distinct name, and this is configured by setting the exporter's `service_attribute` attribute to `"service"`. We also need to give it a handle on the actual service object with the export's `service` attribute. We do this by plugging in `target_service`.

- By default, Pyro advertises on IP address `127.0.0.1`, but this can be overridden by setting the exporter's `service_host` attribute
- By default, Pyro advertises on port `7766`, but this can be overridden by setting the exporter's `service_port` attribute

1. To finish the server side of things, let's write a server script to startup and advertise our service.

```
from springpython.context import *

from simple_service_server_ctx import *

if __name__ == "__main__":
    ctx = ApplicationContext(HappyBirthdayContext())
    ctx.get_object("service_exporter")
```

Now let's start up our server process.



2. As you can see, it is standing by, waiting to be called.
3. Next, we need a client-based application context.

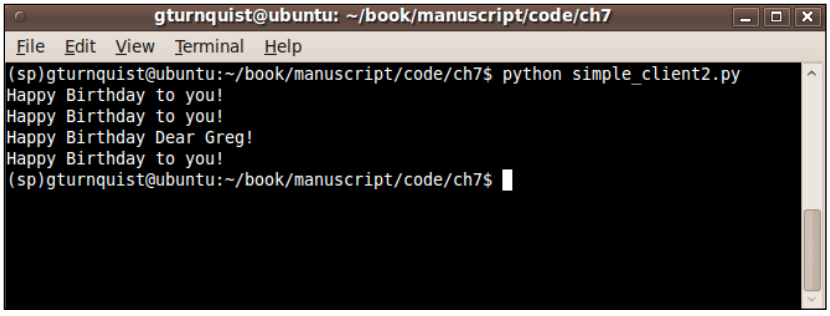
```
from springpython.config import *
from springpython.remoting.pyro import *

class HappyBirthdayContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def service(self):
      proxy = PyroProxyFactory()
      proxy.service_url="PYROLOC://127.0.0.1:7766/service"
      return proxy
```

We define Pyro client using Spring Python's `PyroProxyFactory`, an easy to configure Spring Python proxy factory that handles the task of using Pyro's APIs to find our remote service. And by using the IoC container's original name of service `service` embedded in the URL (`PYROLOC://127.0.0.1:7766/service`), our client script won't require any changes.

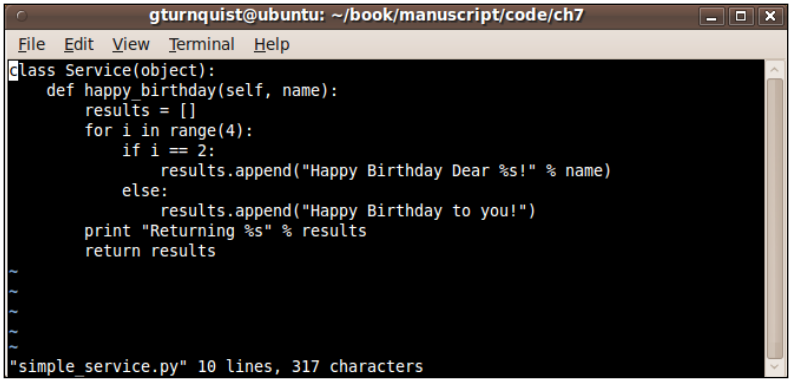
1. Let's create a client-side script to use this context.

```
from springpython.context import *

from simple_service_client_ctx import *

if __name__ == "__main__":
    ctx = ApplicationContext(HappyBirthdayContext())
    s = ctx.get_object("service")
    print "\n".join(s.happy_birthday("Greg"))
```

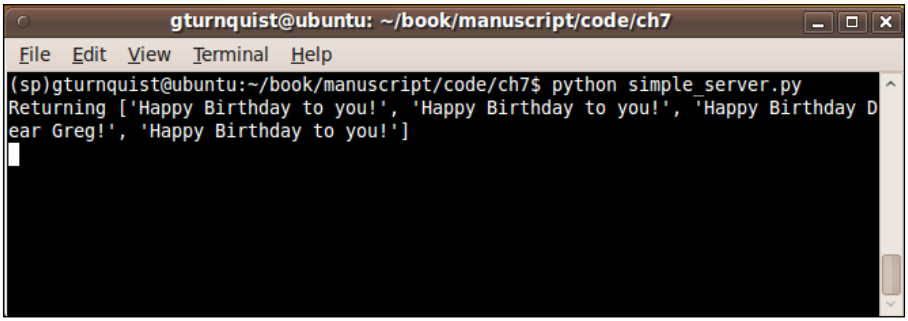
2. Now let's run our client.



Unfortunately, nothing is printed on the screen from the server side. That is because we don't have any print statements or logging. However, if we go and alter our original service at `simple_service.py`, we can introduce a print statement to verify our service is being called on the server side.



Let's restart it and call the client script again.



This shows that our server code is being called from the client. And it's nicely decoupled from the machinery of Pyro.

Is our example contrived?

Does this example appear contrived? Sure it does. However, the basic concept of accessing a service from a client is not.

Instead of our Happy Birthday service, this could be the data access layer of an application, an interface into an airline flight reservation system, or access to trouble ticket data offered by an operations center.

Our example still has the same fundamental concepts of input arguments, output results, and client-side processing after the fact.

Spring Python is non-invasive

We took a very simple service, and by serving it up through an IoC container, it was easy to wrap it with a Pyro exporter without our application realizing it. The IoC container, as demonstrated throughout this book, opens the door to many options. Now, we see how it lends itself to exposing our code as a Pyro service. This concept doesn't stop with Pyro. This same pattern can be applied to export services for other remoting mechanisms. By being able to separate the configuration from the actual business logic, we can yet again apply a useful and practical service without having to rewrite the code to work with the service.

Scaling our application

At the beginning of this chapter, we talked about the need to scale applications. So far, we have shown how to take a very simple client-server application and split it between two instances of Python.

However, scaling applications typically implies running multiple copies of an application in order to handle larger loads. Usually, some type of load balancer is needed. Let's explore how we can use Spring Python's remoting services mixed with some simple Python to create a multi-node version of our service.

Converting the single-node backend into multiple instances

The first step is to create another application context. We are going to spin up a different configuration of components, that is a different blue print, and plan to re-use our existing business code without making any changes.

1. Create a two-node configuration, which creates two instances of Service, advertised on different ports.

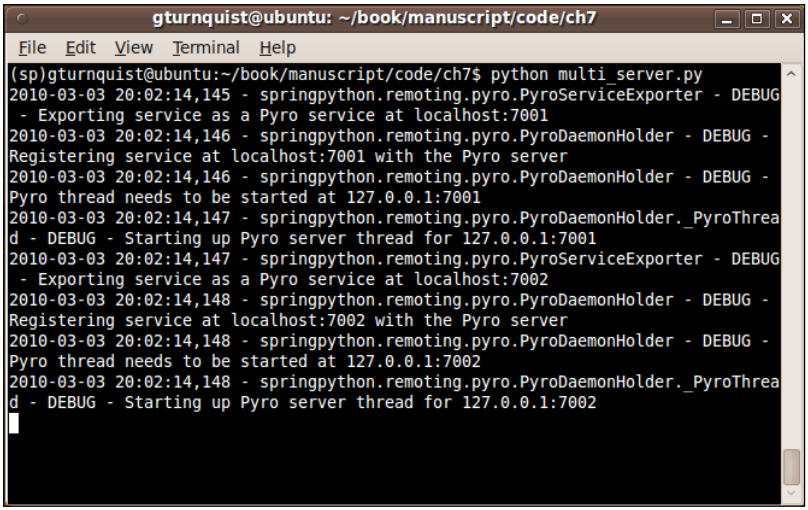
```
from springpython.config import *
from springpython.context import scope
from springpython.remoting.pyro import *

from simple_service import *

class HappyBirthdayContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object(scope.PROTOTYPE)
    def target_service(self):
        return Service()

    @Object()
    def service_exporter1(self):
        exporter = PyroServiceExporter()
        exporter.service_name = "service"
        exporter.service = self.target_service()
        exporter.service_port = 7001
        return exporter
```

```
    @Object()
    def service_exporter2(self):
        exporter = PyroServiceExporter()
        exporter.service_name = "service"
        exporter.service = self.target_service()
        exporter.service_port = 7002
        return exporter
```

2. In this context, we have two different `PyroServiceExporter`s, each with a different port number. While it appears they are both using the `target_service`, notice how we have changed the scope to `PROTOTYPE`. This means each exporter gets its own instance. To run this new context, we need a different startup script.

```
import logging
from springpython.context import *

from multi_server_ctx import *

if __name__ == "__main__":
    logger = logging.getLogger("springpython.remoting")
    loggingLevel = logging.DEBUG
    logger.setLevel(loggingLevel)
    ch = logging.StreamHandler()
    ch.setLevel(loggingLevel)
    formatter = logging.Formatter("%(asctime)s - %(name)s -
%(levelname)s - %(message)s")
    ch.setFormatter(formatter)
    logger.addHandler(ch)

    ctx = ApplicationContext(HappyBirthdayContext())
```

3. Now let's launch the script:



In this script, we turned on some of Spring Python's built in logging, so we could see some of parts involved with starting up remoting. We can see two different Pyro daemon threads being launched, one for each of the ports. And then we see two instances of our `Service` code being registered, one with each thread.

We now have two copies of our service ready and waiting to be called.

Creating a round-robin dispatcher

There are many different ways to code a dispatcher. For our example, let's pick a simple round robin algorithm where we cycle through a fixed list.

Let's create an application context that includes a round robin dispatcher that acts like our service, fed with a list of `PyroProxyFactory` elements.

```
from springpython.config import *
from springpython.remoting.pyro import *

class RoundRobinDispatcher(object):
    def __init__(self, proxies):
        self.proxies = proxies
        self.counter = 0

    def happy_birthday(self, parm):
        self.counter += 1
        proxy = self.proxies[self.counter % 2]
        print "Calling %s" % proxy
```

```
        return proxy.happy_birthday(parm)

class HappyBirthdayContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def service_proxies(self):
        proxies = []
        for port in ["7001", "7002"]:
            proxy = PyroProxyFactory()
            proxy.service_url ="PYROLOC://127.0.0.1:%s/service" % port
            proxies.append(proxy)
        return proxies

    @Object
    def service(self):
        return RoundRobinDispatcher(self.service_proxies())
```

We configured `service_proxies` as an array of `PyroProxyFactory`s. This lets us define a static list of connections.

The `RoundRobinDispatcher` class is injected with the proxies, and mimics the API of our Happy Birthday service. Every call into the dispatcher increments the counter. It then picks one of the proxies to make the actual call.

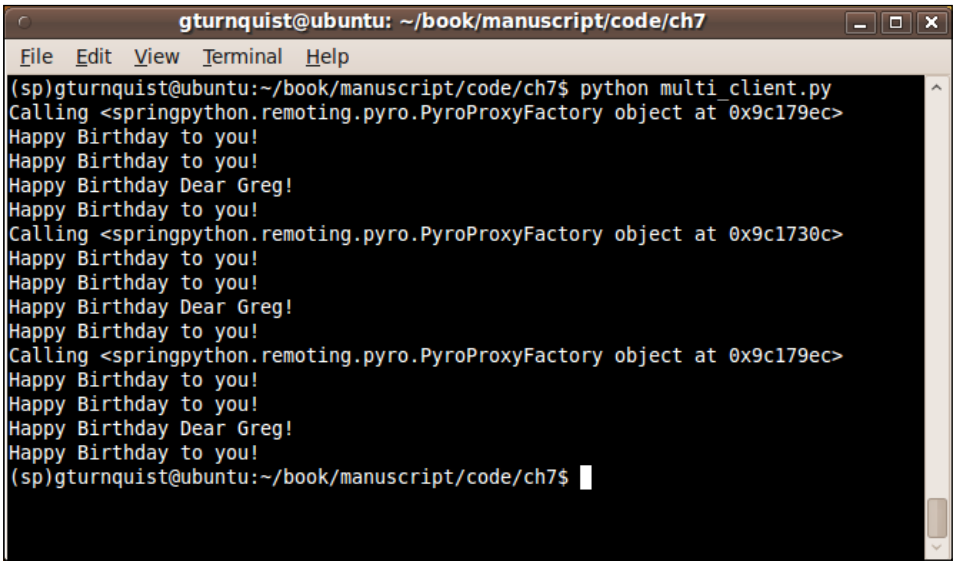
Adjusting client configuration without client code knowing its talking to multiple node backend

Let's write the client script to use our round robin dispatcher.

```
from springpython.context import *

from multi_client_ctx import *

if __name__ == "__main__":
    ctx = ApplicationContext(HappyBirthdayContext())
    s = ctx.get_object("service")

    print "\n".join(s.happy_birthday("Greg"))
    print "\n".join(s.happy_birthday("Greg"))
    print "\n".join(s.happy_birthday("Greg"))
```

There is hardly an impact to the client. The only difference is the import statement, and the fact that we are calling the service multiple times. If the client was actually a GUI application with this tied to a button, there would be no difference.

Now let's run the client script.



As can be seen, each invocation shows which `ProyProxyFactory` was called, and they clearly show switching back and forth between the two proxies registered. With each call smoothly integrated with a corresponding `PyroServiceExporter`, we have scaled our application into a two-node version.

Summary

This example again demonstrates how utilizing an IoC container allows sophisticated wiring of components while not requiring either the client or the server to know about it.

The latest example has totally rewired the original concept of the application simply by coding a different set of blue prints, i.e. IoC configuration. It also demonstrates how things are more powerful and sophisticated when using the pure Python IoC configuration. We were able to mix in code as well as parameters, which is much smoother than writing XML files.

Our application started off as a simple client calling a service. By rewiring it to use IoC, it allowed us to easily reconfigure things to run multiple copies of our service on multiple nodes, without having to change the core logic.

Our round robin dispatcher is admittedly very simplistic. A real production solution would require:

- More sophisticated error handling to deal with things like remote access exceptions.
- We would also need a way to grow new nodes and have them added to the list.
- Perhaps by letting the dispatcher export itself, other services can latch on and find out what services are available.

Spring Python doesn't provide dispatchers, routers, and other types of components used to integrate systems together. Spring Integration (`http://www.springsource.org/spring-integration`) is a separate part of the Spring portfolio that brings these features to Java. Spring Python Remoting is a building block that in the future can be used to build a Spring Python Integration module.

We could spend countless hours refining our dispatcher. We can also focus on adding more features to our core business logic.

The important factor in all this as that we have demonstrated how Spring Python provides a non-invasive way to scale our original application far beyond what it was originally designed for.

In this chapter we have learned:

- How Pyro provides a nice Python-to-Python remoting capability to easily create client-server applications
- How Spring Python seamlessly integrates with Pyro so that your application doesn't have to learn the API
- That it is easy to convert a simple application into a distributed one, all on the same machine
- We rewired our application by splitting it up into parts, plugging in a round-robin queue manager, and running multiple copies of the server with no impact to our business logic

In the next chapter, we will use all of Spring Python's components to build a simple, scalable, and secure application with strong integrity.

8

Case Study I—Integrating Spring Python with your Web Application

Throughout the earlier chapters, we have covered the building blocks of Spring Python: dependency injection, aspect oriented programming, database template, transaction management, security, and remoting. All of these pieces are like the bricks used to build a house. We looked at each brick by itself, and saw how to utilize it. In this chapter we will explore using all of them together to build a comprehensive banking application.

In this chapter we will learn how to:

- Put together a simple banking application with a nicely decoupled view and controller layer
- Apply simple authentication mechanisms to grant access to different types of users
- Apply role-based authorization, distinguishing between different groups of users
- Create custom authorization to prevent customers from seeing each other's data
- Export data over a trusted network in a raw, machine-readable format
- Export data to external users, going through established security protocols to only provide this data to authenticated and authorized clients
- Seamlessly audit banking operations
- Mark up multi-step operations as atomic transactions

Requirements for a good bank

Before we can embark on building our application, we need to establish the stories that we will implement in our coding sprint. What do we need to do to implement a good banking application?

- A customer can open a new account with a balance of $0.00
- A customer can close an account that has $0.00 balance
- Opening and closing accounts written into a log visible to the owning customer and any manager
- A customer can withdraw any amount up to the total balance of the account
- A customer can deposit any amount into an existing account they own
- A customer can transfer from one account they own to another account they own, up to the total balance of the source account
- All withdrawals, deposits, and transfers are written into a log visible to the owning customer and any manager
- Logs will be available through a secure, machine-to-machine format, requiring valid credentials
- The action of a manager viewing a log will be logged separately. This log will be visible by a supervisor

This isn't everything we would want from a bank, but it's a nice start. Our sprint is focused on building some basic deposit/withdraw functionality, while logging these transactions. This demands integrity to avoid leaking money.

We also need supervision over the managers to monitor when they inspect transaction logs. Access for customers, managers, and supervisors will require some fine grained security controls to be put in our banking application.

Finally, the ability to read logs through a remote connection will nicely support integrating with other banks that our customers may work with.

Let's get started!

Building a skeleton web application

To get underway, we need a shell of a web application where we can start filling in the details. We previously used CherryPy (`http://cherrypy.org`) to build a simple web app, so let's use that to kick things off.

1. First of all, let's code a main application that will bootstrap CherryPy as well as our application context.

```
import cherrypy
import os
import ctx
from springpython.context import ApplicationContext

if __name__ == '__main__':
    cherrypy.config.update({'server.socket_port': 8009})

    applicationContext =
        ApplicationContext(ctx.SpringBankAppContext())

    cherrypy.tree.mount(
        applicationContext.get_object("view"),
        '/',
        config=None)

    cherrypy.engine.start()
    cherrypy.engine.block()
```

   You may notice that, it is looking inside package `ctx` for `SpringBankAppContext` to find the view object.

2. Let's write a simple, pure Python IoC container that will create the CherryPy view object we need.

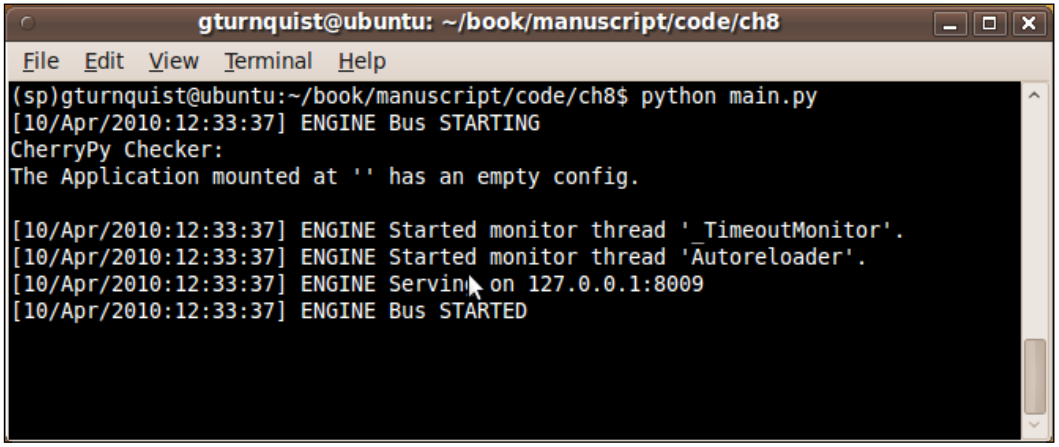
```
from springpython.config import PythonConfig, Object

from app import *

class SpringBankAppContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def view(self):
        return SpringBankView()
```

This is the simplest container possible. We basically have one object, `view`, which returns an instance of `SpringBankView`. Later on, if we need other injected objects or features, we can easily update this configuration.

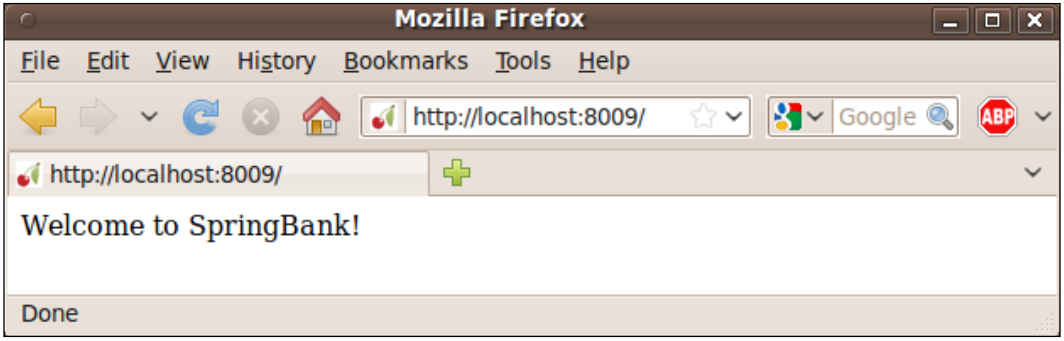
3. Now let's write a simple view layer with a **Hello, world** style opening page.

```
import cherrypy

class SpringBankView(object):

    @cherrypy.expose
    def index(self, args=None):
        return """
            Welcome to SpringBank!
        """
```

4. Let's start our application and see what it looks like.



5. Now let's visit the advertised location of `http://127.0.0.1:8009`.



This is our starting position of a web application. From here, we will spend the rest of the chapter tackling the features we just described.

Securing the application

Before we get going, let's go ahead and plug-in some security. In the security chapter, we discussed how security can be conveniently added after the fact. But it's even better if we start with it first. We will do so by adding a simple login page, and hard wiring three accounts: a customer, a bank manager, and a bank supervisor.

1. Let's revise the boot strap main with some extra features to turn on security.

```
import cherrypy
import os
from springpython.context import ApplicationContext
from springpython.security.context import *

from ctx2 import *

if __name__ == '__main__':
    cherrypy.config.update({'server.socket_port': 8009})

    ctx = ApplicationContext(SpringBankAppContext())

    SecurityContextHolder.setStrategy(SecurityContextHolder.MODE_
GLOBAL)
    SecurityContextHolder.getContext()

    conf = {"/": {"tools.sessions.on":True,
                  "tools.filterChainProxy.on":True}}

    cherrypy.tree.mount(
        ctx.get_object("view"),
        '/',
        config=conf)

    cherrypy.engine.start()
    cherrypy.engine.block()
```

First of all, we need to initialize the security context settings stored in `SecurityContextHolder`. Another thing required by CherryPy is turning on HTTP sessions. We also must activate the `filterChainProxy`. Spring Python's `FilterChainProxy`, setup in our application context, automatically registers itself as a CherryPy tool.

2. Let's add the security components to our IoC configuration.

```
from springpython.config import PythonConfig, Object
from springpython.security.providers import *
from springpython.security.providers.dao import *
from springpython.security.userdetails import *
from springpython.security.vote import *
from springpython.security.web import *
from springpython.security.cherrypy3 import *

from app2 import *

class SpringBankAppContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def view(self):
        view = SpringBankView()
        view.auth_provider = self.auth_provider()
        view.filter = self.auth_processing_filter()
        view.http_context_filter = self.
httpSessionContextIntegrationFilter()
        return view

    @Object
    def filterChainProxy(self):
     return CP3FilterChainProxy(filterInvocationDefinitionSource =
            [
            ("/login.*",  ["httpSessionContextIntegrationFilter"]),
            ("/.*",       ["httpSessionContextIntegrationFilter",
                           "exception_translation_filter",
                           "auth_processing_filter",
                           "filter_security_interceptor"])
            ])
```

```
    @Object
    def httpSessionContextIntegrationFilter(self):
        filter = HttpSessionContextIntegrationFilter()
        filter.sessionStrategy = self.session_strategy()
        return filter

    @Object
    def session_strategy(self):
        return CP3SessionStrategy()

    @Object
    def exception_translation_filter(self):
        filter = ExceptionTranslationFilter()
        filter.authenticationEntryPoint = self.auth_filter_entry_
pt()
        filter.accessDeniedHandler = self.accessDeniedHandler()
        return filter

    @Object
    def auth_filter_entry_pt(self):
        filter = AuthenticationProcessingFilterEntryPoint()
        filter.loginFormUrl = "/login"
        filter.redirectStrategy = self.redirectStrategy()
        return filter

    @Object
    def accessDeniedHandler(self):
        handler = SimpleAccessDeniedHandler()
        handler.errorPage = "/accessDenied"
        handler.redirectStrategy = self.redirectStrategy()
        return handler

    @Object
    def redirectStrategy(self):
        return CP3RedirectStrategy()

    @Object
    def auth_processing_filter(self):
        filter = AuthenticationProcessingFilter()
        filter.auth_manager = self.auth_manager()
        filter.alwaysReauthenticate = False
```

```
        return filter

    @Object
    def auth_manager(self):
        auth_manager = AuthenticationManager()
        auth_manager.auth_providers = [self.auth_provider()]
        return auth_manager

    @Object
    def auth_provider(self):
        provider = DaoAuthenticationProvider()
        provider.user_details_service = self.user_details_service()
        provider.password_encoder = PlaintextPasswordEncoder()
        return provider

    @Object
    def user_details_service(self):
        user_details_service = InMemoryUserDetailsService()
        user_details_service.user_dict = {
            "alice": ("alicespassword",["ROLE_CUSTOMER"], True),
            "bob": ("bobspassword", ["ROLE_MGR"], True),
            "carol": ("carolspassword", ["ROLE_SUPERVISOR"], True)
            }
        return user_details_service

    @Object
    def filter_security_interceptor(self):
        filter = FilterSecurityInterceptor()
        filter.auth_manager = self.auth_manager()
        filter.access_decision_mgr = self.access_decision_mgr()
        filter.sessionStrategy = self.session_strategy()
        filter.obj_def_source = [
          ("/.*", ["ROLE_CUSTOMER", "ROLE_MGR", "ROLE_SUPERVISOR"])
            ]
        return filter

    @Object
    def access_decision_mgr(self):
        access_decision_mgr = AffirmativeBased()
        access_decision_mgr.allow_if_all_abstain = False
        access_decision_mgr.access_decision_voters = [RoleVoter()]
        return access_decision_mgr
```

This is admittedly a lot of code to add to our application context. Spring Python requires a lot of steps to add the security components, a fate shared by the Spring Security project. They spent a considerable amount of effort reducing the amount of code needed to configure security for typical configurations. Hopefully in the future Spring Python can be improved in a similar fashion.

In this configuration, we are using the `InMemoryUserDetailsService` in order to hard code three users: alice (customer), bob (bank manager), and carol (bank supervisor). This saves us from having to configure a database while we develop our application.

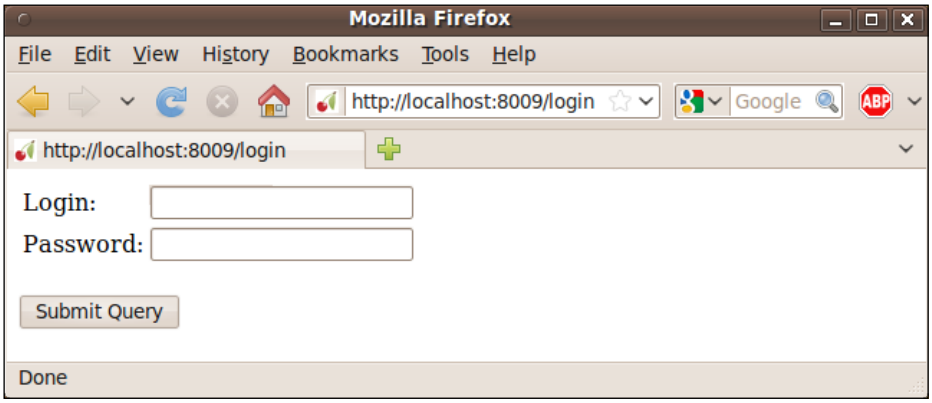
3. Let's change the core app, so that it has a login page and can handle logging in and logging out.

```
import cherrypy

from springpython.security import *
from springpython.security.providers import *
from springpython.security.context import *

class SpringBankView(object):

    def __init__(self):
        self.filter = None
        self.auth_provider = None
        self.http_context_filter = None

    @cherrypy.expose
    def index(self):
        return """
            Welcome to SpringBank!
            <p>
            <p>
            <a href="logout">Logout</a href>
        """

    @cherrypy.expose
    def login(self, from_page="/", login="", password="", error_
msg=""):
        if login != "" and password != "":
            try:
                self.attempt_auth(login, password)
                raise cherrypy.HTTPRedirect(from_page)
            except AuthenticationException, e:
```

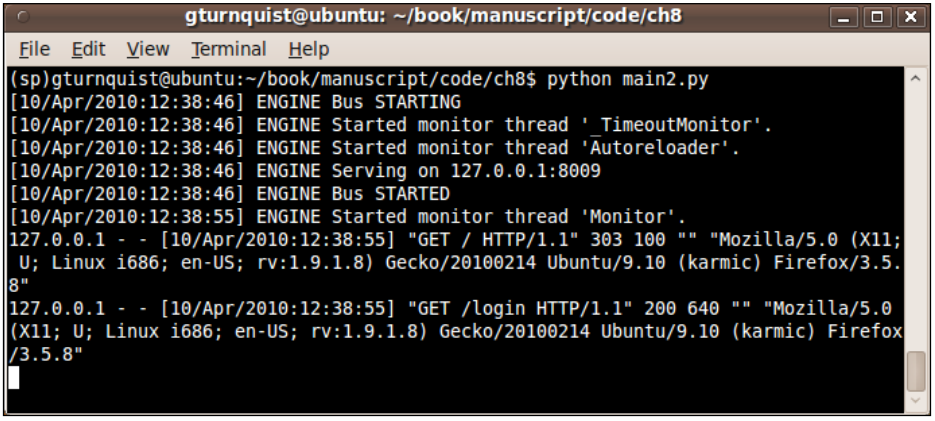
```
                raise cherrypy.HTTPRedirect(
                   "?login=%s&error_msg=Username/password failure"
                    % login)

        return """
                %s<p>
                <form method="POST" action="">
                <table>
                    <tr>
                        <td>Login:</td>
                        <td><input type="text" name="login"
                                                value="%s"/></td>
                    </tr>
                    <tr>
                        <td>Password:</td>
                        <td>
                        <input type="password" name="password"/>
                        </td>
                    </tr>
                </table>
                <input type="hidden" name="from_page"
                       value="%s"/><br/>
                <input type="submit"/>
                </form>
        """ % (error_msg, login, from_page)

    def attempt_auth(self, username, password):
        token = UsernamePasswordAuthenticationToken(username,
                password)
        SecurityContextHolder.getContext().authentication = \
                            self.auth_provider.authenticate(token)
        self.http_context_filter.saveContext()

    @cherrypy.expose
    def logout(self):
        self.filter.logout()

        self.http_context_filter.saveContext()
        raise cherrypy.HTTPRedirect("/")
```

Here we have slightly altered index, so that it prints our 'welcome' but also offers a hyperlink to logout. We have also added a `/login` link that either attempts to log the user in, or displays an HTML form so the user can attempt to login.

4. With these changes in place, let's restart the app, and open up the browser.

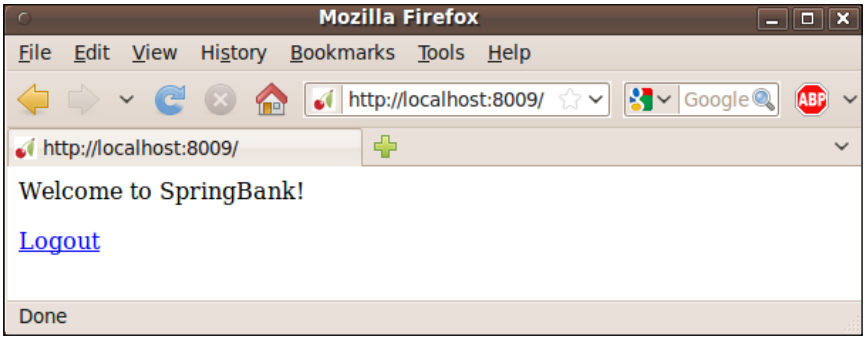


As you can see, we are now looking at a very simple login screen. If we look at the shell in which our updated application is running, we can see what has happened.



We pointed our browser at the root URL, but Spring Python's security filter noticed we had no credentials, so it redirected us to `/login`.

5. Let's login as Bob the bank manager.



Now we have logged in to our SpringBank web site. With the security in place, we should easily be able to start adding more features and control who can access what.

Building some basic customer functions

To get our bank site going, we need to start lettings customers open bank accounts. This is a feature only visible to customers.

1. First, we need to update the main page, so when Alice the customer logs in, she has the option to open a bank account. We can do this by altering the `index` function in `SpringBankView` to offer options driven by the user's credentials.

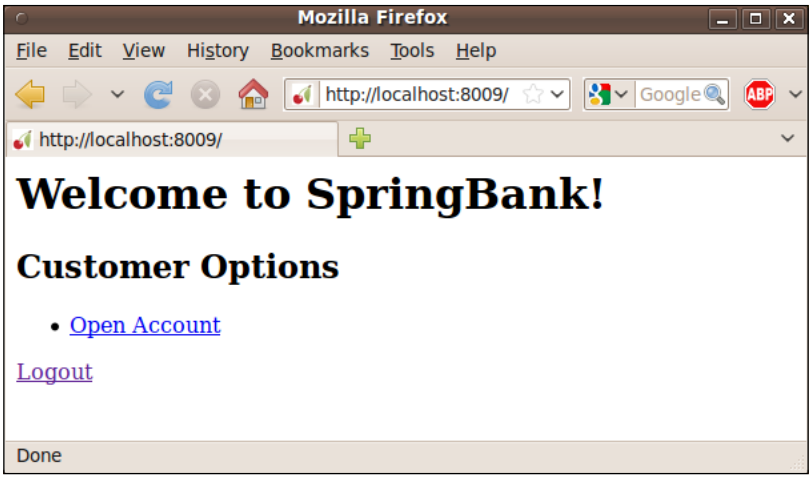
```
@cherrypy.expose
def index(self, message=""):
    results = """
        <h1>Welcome to SpringBank!</h1>
        <p>
        %s
        <p>
    """ % message

    if "ROLE_CUSTOMER" in  \
      SCH.getContext().authentication.granted_auths:
        results += """
            <h2>Customer Options</h2>
            <ul>
               <li><a href="openAccount">Open Account</a></li>
            </ul>
        """
```

```
        results += """
            <a href="logout">Logout</a href>
        """
        return results

    @cherrypy.expose
    def openAccount(self, account="", desc=""):
        if account != "" and desc != "":
            self.controller.open_account(account, desc)
            raise cherrypy.HTTPRedirect("/?message=Account created
successfully")

        return """
           <form method="POST" action="">
           <table>
               <tr>
                   <td>Account Name</td>
                   <td><input type="text" name="account"/><td>
               </tr>
               <tr>
                   <td>Description</td>
                   <td><input type="text" name="desc"/><td>
               </tr>
           </table>
           <input type="submit"/>
           </form>
           <a href="/">Cancel</a>
        """
```

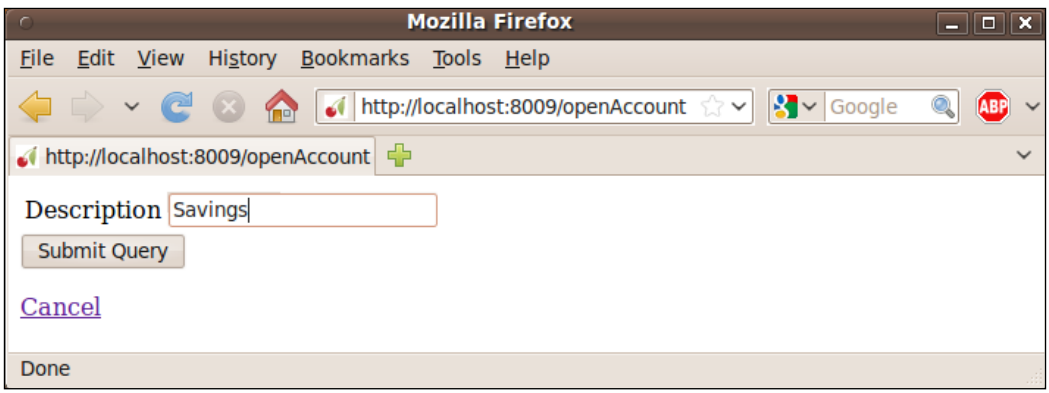
To make this work, we needed to alter one of our `import` statements:

```
from springpython.security.context import SecurityContextHolder
as SCH
```

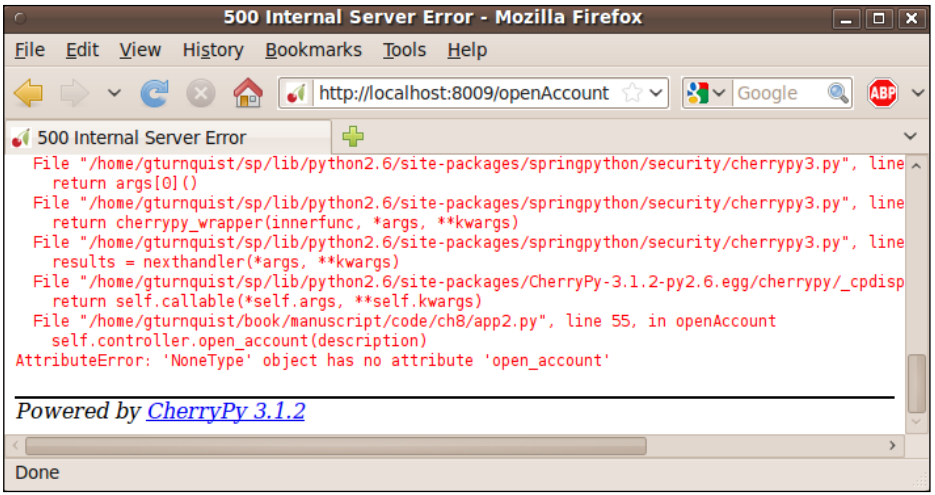
Basically, the index function now checks the user's `granted_auths`, and if they have `ROLE_CUSTOMER`, they are shown the customer options. The first option provided is a link that invokes the `openAccount` function. As seen below, if Alice logs in, she now sees this extra option.



Clicking on the link takes the user to a page where they can fill out the details required to request a new account.



When the page is submitted, the information gets fed to the controller. There is one big problem with this piece of code: there is no controller! If you click the button, it will generate an error, indicating we need to fill in the supporting functionality.



2. Now let's create a controller that will process requests such as opening new bank accounts.

```
class SpringBankController(object):
    def __init__(self, factory):
        self.factory = factory
        self.dt = DatabaseTemplate(self.factory)

    def open_account(self, account, desc):
        self.dt.execute("""
            INSERT INTO account
            (account, description, balance)
            VALUES
            (?, ?, 0.0)""", (account, desc))
```

This controller has one business function: `open_account`. It utilizes a `DatabaseTemplate` to manage this. As you can see, we have coded it to initialize new accounts with a balance of $0.00. It is also designed to depend on *dependency injection* to supply it the necessary connection factory.

3. Let's wire up the controller from our IoC configuration by injecting it with a connection factory.

```
@Object
def controller(self):
    return SpringBankController(self.factory())

@Object
def view(self):
    view = SpringBankView()
    view.auth_provider = self.auth_provider()
    view.filter = self.auth_processing_filter()
    view.http_context_filter =
                    self.httpSessionContextIntegrationFilter()
    view.controller = self.controller()
    return view

@Object
def factory(self):
    factory = MySQLConnectionFactory()
    factory.username = "springbank"
    factory.password = "springbank"
    factory.hostname = "localhost"
    factory.db = "springbank"
    return factory
```

Here we have the new `controller` object as well as the `factory` object, which allows our `controller` to access the database. We have slightly revised the `view` object so that it gets injected with a copy of the `controller`.

4. Adjust `SpringBankView` so that a controller can be injected.

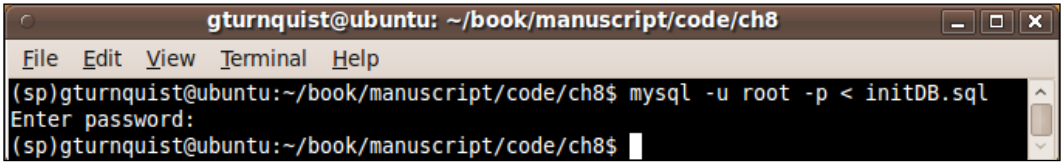
```
class SpringBankView(object):

    def __init__(self):
        self.filter = None
        self.auth_provider = None
        self.http_context_filter = None
        self.controller = None
```

Now our `SpringBankView` has a handle on the controller, arming it to make database calls as needed. However, one piece is missing. We need to set up a database.

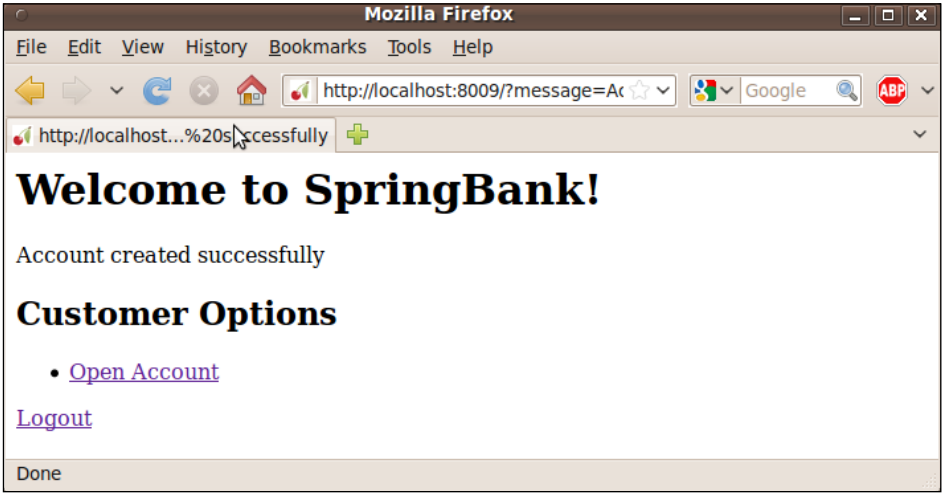
5. We can initialize a MySQL database with the following script.

```
DROP DATABASE IF EXISTS springbank;

CREATE DATABASE springbank;

GRANT ALL ON springbank.* TO springbank@localhost IDENTIFIED BY
'springbank';

USE springbank;

CREATE TABLE account (
  id INT(4) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  description VARCHAR(30),
  balance DECIMAL(10,2)
);
```

If we run that using MySQL's root account, it should set up our account table.

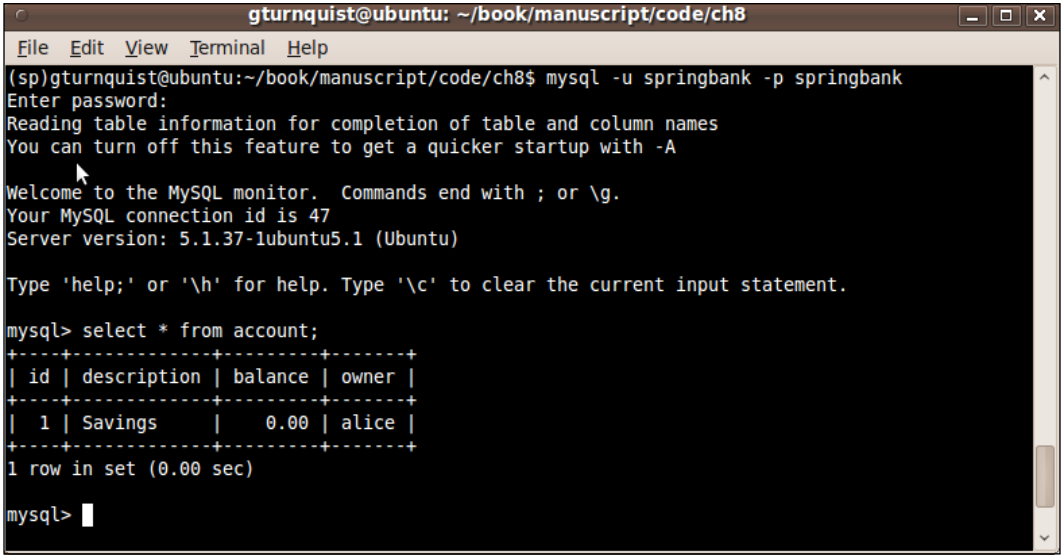


6. Now let's try to create that bank account.



Success! If you look closely at the revised version of `SpringBankView`'s `index` method, you will notice an optional `message` parameter. This lets us push through a status message while redirecting the user to the front page after completing any operation. In our situation, we have created a bank account with $0.00 balance.

We don't yet see existing bank accounts listed, but you can still examine the database from the command line to verify our operation worked.



Coding more features

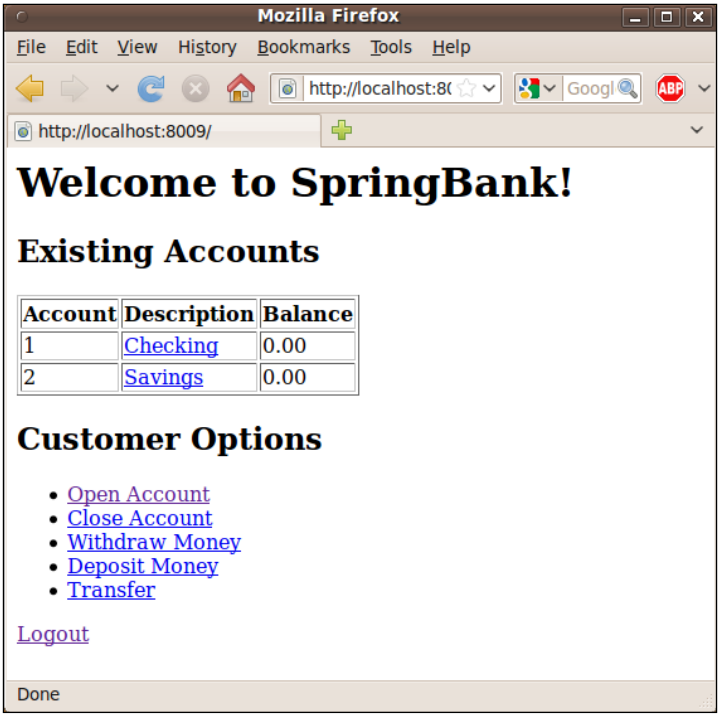
So far, we have coded a few of our requirements from front to back. We built the view layer using CherryPy. Next, we configured security to protect the site. Then we fully implemented one of the customer features so that it stored new accounts in the database.

Coding more features follows a similar pattern. The order the steps are executed are a matter of taste. It is possible to code the entire back end using automated testing, followed by skinning it with a web layer. In our case, we used the more demonstrative style of building the interface first, and then fleshing out the backend later. Either way, it is easy to iterate through and see how Spring Python's IoC container kept the security, view, and controller logic nicely decoupled.

Now let's fast forward development to the point where all the customer features are implemented.

Updating the main page with more features

We updated the index page in `SpringBankView`, so that it lists the customer's current accounts as well as listing the other functions available.



In this example, Alice already has two accounts created. In the previous version, they weren't listed, but now they are.

```
@cherrypy.expose
def index(self, message=""):
    results = """
        <h1>Welcome to SpringBank!</h1>
        <p>
        %s
        <p>
    """ % message

    if "ROLE_CUSTOMER" in  \
      SCH.getContext().authentication.granted_auths:
        results += """
            <h2>Existing Accounts</h2>
```

```
                <table border="1">
                   <tr><th>Account</th><th>Description
                       </th><th>Balance</th></tr>
            """

            for account in self.controller.get_accounts(
                           SCH.getContext().authentication.username):
                results += '''
                    <tr>
                    <td>%s</td><td><a href="history?id=%s">%s</a>
                    </td><td>%s</td>
                    </tr>
                ''' % (account["id"], account["id"],
                       account["description"], account["balance"])

            results += """
                </table>
                <h2>Customer Options</h2>
                <ul>
                    <li><a href="openAccount">Open Account</a></li>
                    <li><a href="closeAccount">Close Account</a></li>
                    <li><a href="withdraw">Withdraw Money</a></li>
                    <li><a href="deposit">Deposit Money</a></li>
                    <li><a href="transfer">Transfer</a></li>
                </ul>
            """

        results += """
            <a href="logout">Logout</a href>
        """
        return results
```

As you can see, wedged into the middle is a for-loop that generates rows in an HTML table, listing the accounts the user owns. `get_accounts` is delegated to the controller.

```
    def get_accounts(self, username):
        return self.dt.query("""
            SELECT id, description, balance
            FROM account
            WHERE owner = ?
            AND status = 'OPEN'""", (username,),
            DictionaryRowMapper())
```

Each row includes a link to view the history for that account, and at the bottom of the screen are links to the other new functions.

Refining the ability to open an account

We altered the back end of `openAccount` by marking new accounts with a state of `OPEN`.

```
        def open_account(self, description):
            self.dt.execute("""
                INSERT INTO account
                (description, balance, owner, status)
                VALUES
                (?, 0.0, ?, 'OPEN')""", (description,\
                     SCH.getContext().authentication.username))

            self.log("TX", self.get_latest_account(), "Opened account %s"
    % description)
```

We also added the extra step of writing a transaction log entry. To support writing the log entry, the controller must retrieve the latest account created for this customer.

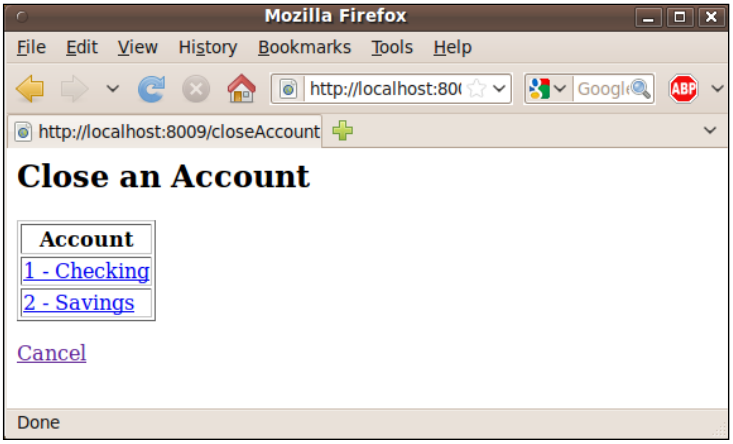
```
        def get_latest_account(self):
            return self.dt.query_for_long("""
                SELECT max(id)
                FROM account
                WHERE owner = ?
                AND status = 'OPEN'""", \
                   (SCH.getContext().authentication.username,))
```

`open_account`, along with several of the other controller operations uses the following code to write log entries.

```
        def log(self, type, account, message):
            self.dt.execute("""
                INSERT INTO log
                (type, account, message)
                values
                (?, ?, ?)""", \
                      (type, account, message))
```

Adding the ability to close an account

We added `closeAccount` to `SpringBankView` to let the customer close an existing account if its balance is zero.




```
@cherrypy.expose
def closeAccount(self, id=""):
    if id != "":
        self.controller.close_account(id)
        raise cherrypy.HTTPRedirect("/?message=
                                    Account successfully closed")

    results = """
        <h2>Close an Account</h2>
        <form method="POST" action="">
        <table border="1">
           <tr><th>Account</th></tr>
    """

    for account in self.controller.closeable_accounts(
                        SCH.getContext().authentication.username):
        results += """
            <tr><td><a href="closeAccount?id=%s">%s - %s</a>
                </td></tr>
        """ % (account["id"], account["id"],
                                            account["description"])

    results += """
        </table>
        </form>
        <a href="/">Cancel</a>
    """
    return results
```

Looking up `closeable_accounts` as well as completing the `close_account` operation is delegated to the controller.

```
def closeable_accounts(self, username):
    return self.dt.query("""
        SELECT id, description
        FROM account
        WHERE owner = ?
        AND balance = 0.0
        AND status = 'OPEN'""", (username,),
        DictionaryRowMapper())

def close_account(self, id):
    self.dt.execute("""
        UPDATE account
        SET status = 'CLOSED'
        WHERE id = ?""", (id,))
    self.log("TX", id, "Closed account")
```

Opening a new account causes a row to be inserted into the ACCOUNT table. However, closing an account does not mean deleting the same row. Instead, its status is updated to CLOSED.

Adding the ability to withdraw money

We added `withdraw` to `SpringBankView` to let the customer withdraw money from an existing account.

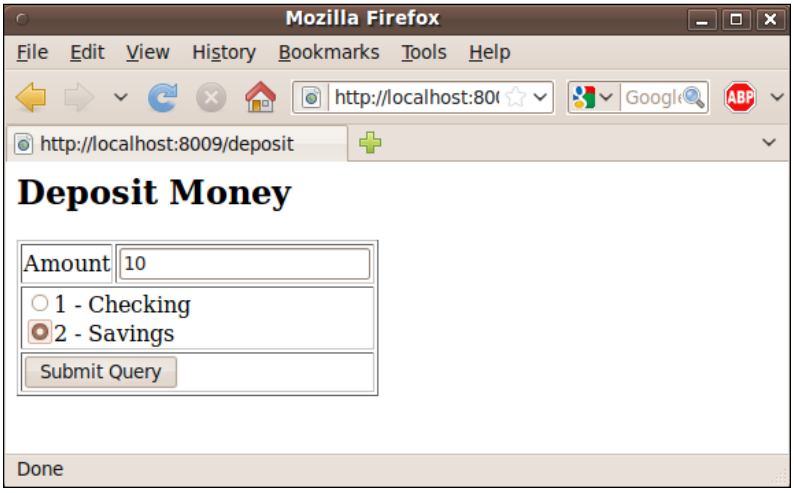
```
    @cherrypy.expose
    def withdraw(self, id="", amount=""):
        if id != "" and amount != "":
            self.controller.withdraw(id, float(amount))
            raise cherrypy.HTTPRedirect("/?message=Successfully
withdrew %s" % amount)

        results = """
            <h2>Withdraw Money</h2>
            <form method="POST" action="">
            <table border="1">
                <tr>
                    <td>Amount</td>
                    <td><input type="text" name="amount"/></td>
                </tr>
                <tr><td colspan="2">
        """

        for account in self.controller.get_accounts(SCH.getContext().
authentication.username):
            results += """
                <input type="radio" name="id" value="%s">%s - %s</
input><br/>
            """ % (account["id"], account["id"],
account["description"])

        results += """
                </td></tr>
                <tr><td colspan="2"><input type="submit"/></td></tr>
            </form>
        """

        return results
```

When the user clicks submit, `withdraw` is delegated to the controller.

```
    def withdraw(self, id, amount):
        self.dt.execute("""
            UPDATE account
            SET balance = balance - ?
            WHERE id = ?""", (amount, id))
        self.log("TX", id, "Withdrew %s" % amount)
```

Adding the ability to deposit money

We added `deposit` to `SpringBankView` to let the customer deposit money into an existing account.




```
    @cherrypy.expose
    def deposit(self, id="", amount=""):
        if id != "" and amount != "":
            self.controller.deposit(id, float(amount))
            raise cherrypy.HTTPRedirect("/?message=Successfully
deposited %s" % amount)

        results = """
            <h2>Deposit Money</h2>
            <form method="POST" action="">
            <table border="1">
                <tr>
                    <td>Amount</td>
                    <td><input type="text" name="amount"/></td>
                </tr>
                <tr><td colspan="2">
        """

        for account in self.controller.get_accounts(SCH.getContext().
authentication.username):
            results += """
                <input type="radio" name="id" value="%s">%s - %s</
input><br/>
```

```
                """ % (account["id"], account["id"],
    account["description"])

            results += """
                    </td></tr>
                    <tr><td colspan="2"><input type="submit"/></td></tr>
                </form>
            """

            return results
```

When the user clicks submit, `deposit` is delegated to the controller.

```
    def deposit(self, id, amount):
        self.dt.execute("""
            UPDATE account
            SET balance = balance + ?
            WHERE id = ?""", (amount, id))
        self.log("TX", id, "Deposited %s" % amount)
```

Adding the ability to transfer money

We added `transfer` to `SpringBankView` to let the customer transfer money from one account to the other.

```
    @cherrypy.expose
    def transfer(self, amount="", source="", target=""):
        if amount != "" and source != "" and target != "":
            self.controller.transfer(amount, source, target)
            raise cherrypy.HTTPRedirect("/?message=Successful
transfer")

        results = """
            <h2>Transfer Money</h2>
            <form method="POST" action="">
            <table border="1">
                <tr>
                    <td>Amount</td>
                    <td><input type="text" name="amount"/></td>
                </tr>
        """

        accounts = self.controller.get_accounts(SCH.getContext().
authentication.username)

        for param in ["source", "target"]:
            results += '<tr><td>%s</td><td>' % (param[0].upper() +
param[1:])
            for account in accounts:
                results += """
                    <input type="radio" name="%s" value="%s">%s - %s</
input><br/>
                """ % (param, account["id"], account["id"],
account["description"])
            results += "</td></tr>"

        results += """
                <tr><td colspan="2"><input type="submit"/></td></tr>
            </form>
        """

        return results
```

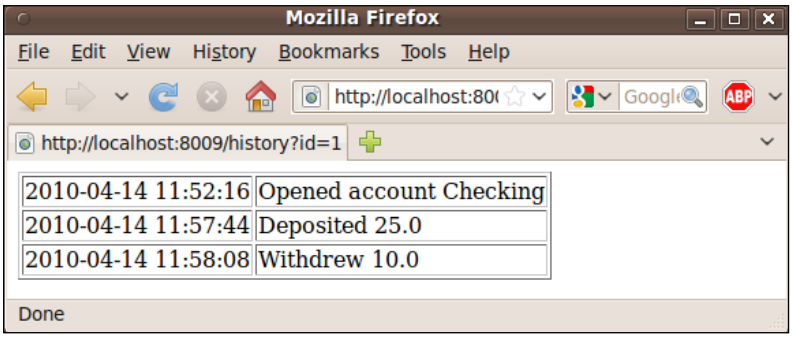
When the user clicks submit, `transfer` is delegated to the controller.

```
    def transfer(self, amount, source, target):
        self.withdraw(source, amount)
        self.deposit(target, amount)
```

`transfer` conveniently re-uses `withdraw` and `deposit`.

Showing account history

The last customer facing feature added was the ability to view an account's history.




```
@cherrypy.expose
def history(self, id):
    results = """
        <table border="1">
    """

    for entry in self.controller.history(id):
        results += """
            <tr><td>%s</td><td>%s</td></tr>
        """ % (entry["date"], entry["message"])

    results += """
        <table>
    """

    return results
```

When the user clicks submit, `history` is delegated to the controller.

```
def history(self, id):
    return self.dt.query("""
        SELECT type, date, message
        FROM log
        WHERE log.account = ?
        AND log.type = 'TX'""", (id,),
        DictionaryRowMapper())
```

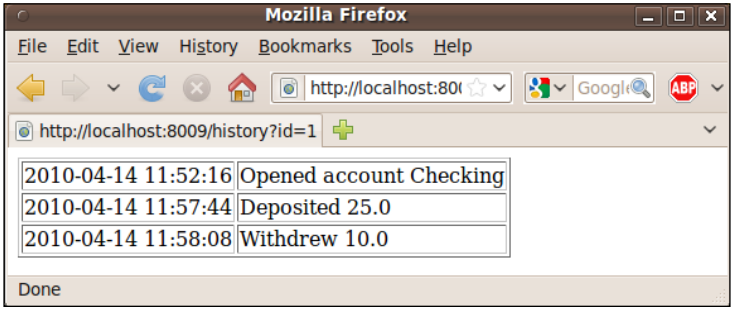
Issues with customer features

With this version of the features, we have roughed in some useful banking features. However, there are some things that still need to be handled.

- There is not enough security preventing another customer from accessing Alice's accounts and transferring money away.
- Withdrawing money from an account has no overdraft protection. This also impacts transfers, because they reuse the withdraw function. A simple solution would be to fail if the requested amount exceeds the balance.
- Transferring money should be transactional to avoid leaking money.

Securing Alice's accounts

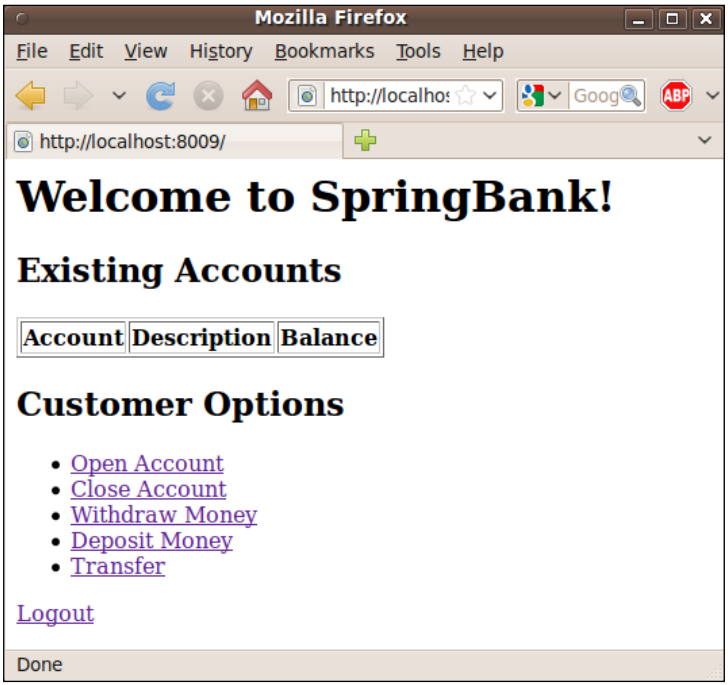
If we log in as Alice, we can view the history of the Checking account.



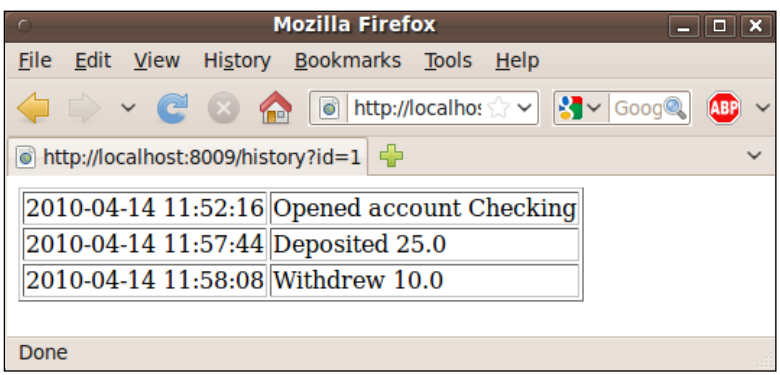
That is alright. However, if there is another customer, then things are not as secured as you might think. Let's adjust the context so that another customer, "Dave" exists.

```
@Object
def user_details_service(self):
    user_details_service = InMemoryUserDetailsService()
    user_details_service.user_dict = {
        "alice": ("alicespassword",["ROLE_CUSTOMER"], True),
        "bob": ("bobspassword", ["ROLE_MGR"], True),
        "carol": ("carolspassword", ["ROLE_SUPERVISOR"], True),
        "dave": ("davespassword",["ROLE_CUSTOMER"], True)
        }
    return user_details_service
```

After looking at Alice's account history, we can easily copy the URL from the browser to clipboard, logout, and then log back in as Dave.



Dave has no accounts yet. However, Dave can paste in the URL and easily view Alice's account history.



This is because both of these users have `ROLE_CUSTOMER`. This means we need to write a customized `AccessDecisionVoter` that will decide whether or not a certain record can be viewed by the current user.

The security chapter showed us how to code a custom authenticator. In this situation, the users are already authenticated. What we need is proper handling of authorization. Our problem requires confirming that the currently logged in user has permission to look at the current account.

1. Let's code our own `AccessDecisionVoter`.

```
class OwnerVoter(AccessDecisionVoter):
    def __init__(self, controller=None):
        self.controller = controller

    def supports(self, attr):
        """This voter will support a list."""
        if isinstance(attr, list) or \
                    (attr is not None and attr == "OWNER"):
            return True
        else:
            return False

    def vote(self, authentication, invocation, config):
        """Grant access if any of the granted
        authorities matches any of the required roles.
        """
        results = self.ACCESS_ABSTAIN
        for attribute in config:
            if self.supports(attribute):
                results = self.ACCESS_DENIED
                id = cgi.parse_qs(
                      invocation.environ["QUERY_STRING"])["id"][0]
                if self.controller.get_username(id) == \
                                authentication.username:
                    return self.ACCESS_GRANTED

        return results
```

   - `OwnerVoter` needs a `supports` method to determine if it is going to vote. In this case, it will if given security configuration in the form of a list, and also if one of the list values is `OWNER`.
   - In order to vote, the voter is provided with a copy of the user's authentication credentials, and a handle on the invocation object which includes all the web request parameters, and what the security role configuration is.

- The voter initializes its `results` to `ACCESS_ABSTAIN`. It iterates over the list of roles, and checks if any of them match `OWNER`. If so, it then changes the `results` to `ACCESS_DENIED`. Then, it checks if the there are any `id` parameters, and if so, retrieves the value. It requests that the `controller` lookup the user associated with the account `id` and if it matches the current user's `username`, it updates the `results` to `ACCESS_GRANTED`.

2. Add a method to the controller to look up the username of an account.

```
def get_username(self, id):
    return self.dt.query_for_object("""
        SELECT owner
        FROM account
        WHERE id = ?
        AND status = 'OPEN'""", (id,), str)
```

3. Now we need to plug the `OwnerVoter` into the context's `AccessDecisionManager` configuration.

```
@Object
def access_decision_mgr(self):
    access_decision_mgr = AffirmativeBased()
    access_decision_mgr.allow_if_all_abstain = False
    access_decision_mgr.access_decision_voters = [RoleVoter(),
                                  OwnerVoter(self.controller())]
    return access_decision_mgr
```

   As you can see, we just added an instance of `OwnerVoter` to the `access_decision_voters` list. `OwnerVoter` needs a handle on the `controller` so it can query for the username linked to the account.

4. Update the `filter_security_interceptor` so that history requests are routed through the `OwnerVoter`.

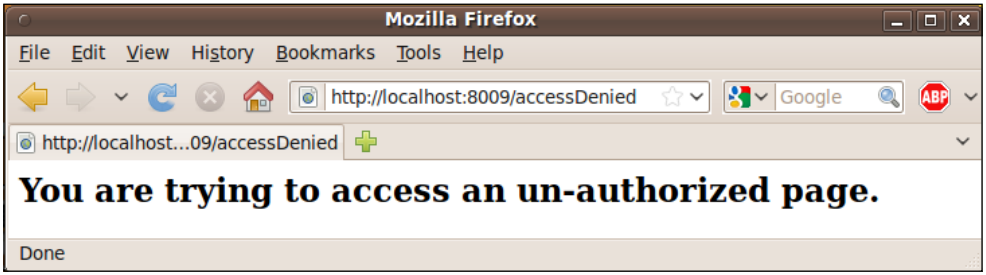
```
@Object
def filter_security_interceptor(self):
    filter = FilterSecurityInterceptor()
    filter.auth_manager = self.auth_manager()
    filter.access_decision_mgr = self.access_decision_mgr()
    filter.sessionStrategy = self.session_strategy()
    filter.obj_def_source = [
        ("/history.*", ["OWNER"]),
        ("/.*", ["ROLE_CUSTOMER", "ROLE_MGR", "ROLE_SUPERVISOR"])
        ]
    return filter
```

Notice how we added a rule for `/history.*`. Because `OWNER` doesn't start with `ROLE_`, the `RoleVoter` will not vote on it. Only the `OwnerVoter` will examine this rule to determine if the current user is authorized to access.

Because the interceptor starts with the first rule and iterates until it finds a match, it is important to put specialized rules towards the top, with more general ones at the bottom.

5. Add an `accessDenied` page at the view level, so that when Dave tries to access the history of Alice's account, he is redirected due to lack of ownership.

```
@cherrypy.expose
def accessDenied(self):
    return """
        <h2>You are trying to access
        an un-authorized page.</h2>
    """
```



We have now secured pages based on ownership. This way, only the account holder can view his or her own history. It is left as an exercise to secure other pages that are id-based using the same `OwnerVoter`.

Adding overdraft protection to withdrawals

One of the simplest business rules is to not allow people to draw more from an account than exists.

1. Modify the `withdraw` operation to first retrieve the account's balance and check it against the amount, throwing an exception if there are insufficient funds.

```
def withdraw(self, id, amount):
    balance = self.dt.query("""
        SELECT balance
        FROM account
        WHERE id = ?""", (id,),
```

```
            DictionaryRowMapper())[0]["balance"]
        if float(balance) < amount:
           raise BankException("Insufficient balance in acct %s" % id)
        self.dt.execute("""
            UPDATE account
            SET balance = balance - ?
            WHERE id = ?""", (amount, id))
        self.log("TX", id, "Withdrew %s" % amount)
```

An important point here is that there is no redirect operation happening. This method isn't aware of the web layer above it. While it would be easy to raise a `cherrypy.HTTPRedirect` message, it is best left to the view layer. This also impacts transfer operations by causing an exception to be thrown if there are insufficient funds.

2. Define a simple `BankException` class.

```
class BankException(Exception):
     pass
```

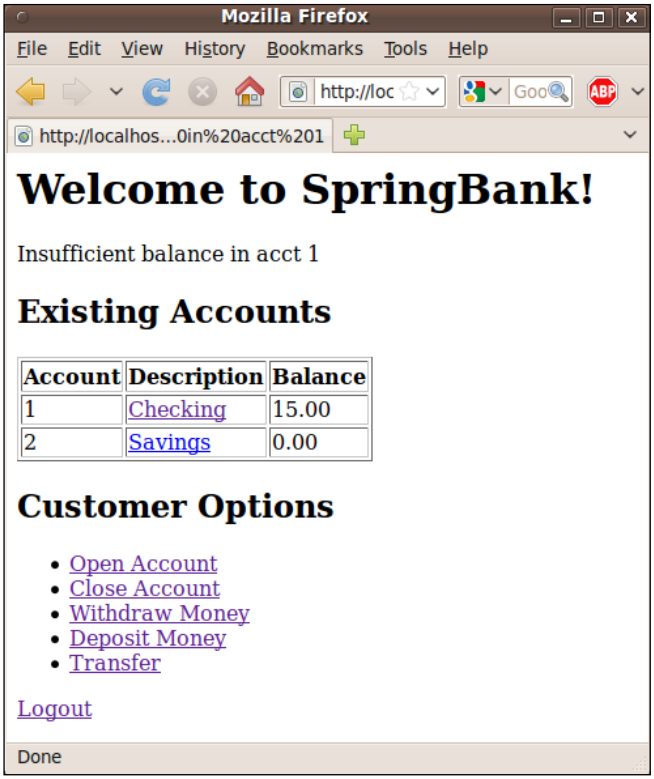
3. Modify the beginning of the `withdraw` web method by adding a try-except block that either redirects with a success message or an error message.

```
@cherrypy.expose
def withdraw(self, id="", amount=""):
    if id != "" and amount != "":
        try:
            self.controller.withdraw(id, float(amount))
            raise cherrypy.HTTPRedirect(\
               "/?message=Successfully withdrew %s" % amount)
        except BankException, e:
            raise cherrypy.HTTPRedirect("/?message=%s" % e)
```

The rest of this view-level method is unchanged. This provides a clear behavior pattern based on any exception thrown down in the `controller`'s `withdraw` operation.

4. Modify the beginning of the `transfer` web method by adding a `try-except` block that either redirects with a success message or an error message.

```
@cherrypy.expose
def transfer(self, amount="", source="", target=""):
    if amount != "" and source != "" and target != "":
        try:
            self.controller.transfer(amount, source, target)
            raise cherrypy.HTTPRedirect(\
                              "/?message=Successful transfer")
        except BankException, e:
            raise cherrypy.HTTPRedirect("/?message=%s" % e)
```



Withdrawals and transfers are now protected from overdrafts.

Making transfers transactional

The final key thing that needs to be implemented is increasing the integrity of transfers. It is vital for any banking operation that they be performed with atomic consistency. Otherwise, the bank will leak money.

1. Markup the controller's transfer operation using the `@transactional` decorator.

```
@transactional
def transfer(self, amount, source, target):
    self.withdraw(source, amount)
    self.deposit(target, amount)
```

2. In order to have the `@transactional` decorator available, the following import statement is required.

```
from springpython.database.transaction import *
```

3. In the application context, define a `TransactionManager` and an `AutoTransactionalObject`, in order to activate the `@transactional` decorator.

```
@Object
def tx_mgr(self):
    return ConnectionFactoryTransactionManager(self.factory())

@Object
def transactionalObject(self):
    return AutoTransactionalObject(self.tx_mgr())
```

4. This change to the application context also requires the same import statement shown above.
5. This is all that is needed. The transfer method now utilizes transactions to avoid leaking money.

Remotely accessing logs

This is a common feature found in online banking, where the user wants to download the transaction history. It allows the importing data into another financial tool for analysis, auditing, or double bookkeeping.

For our case study, we will export using `PyroServiceExporter`. However, we only want to expose the logs and none of the other functions. You may recall that `PyroServiceExporter` exposes all the methods its target object. In order to do this, we need a separate service that can delegate to our controller, and instead expose it.

1. Create a public-facing service, which will contain exposes services.

```
class SpringBankPublic(object):
    def __init__(self, controller):
        self.controller = controller

    def history(self, id):
        return self.controller.history(id)
```

   This class is pretty simple. The constructor call requires a copy of controller. It has only one method, the one intended for public exposure.

2. Export the public-facing service in the application context using Pyro.

```
@Object
def public_service(self):
    return SpringBankPublic(self.controller())

@Object
def spring_bank_exporter(self):
    exporter = PyroServiceExporter()
    exporter.service_name = "springbank"
    exporter.service = self.public_service()
    return exporter
```

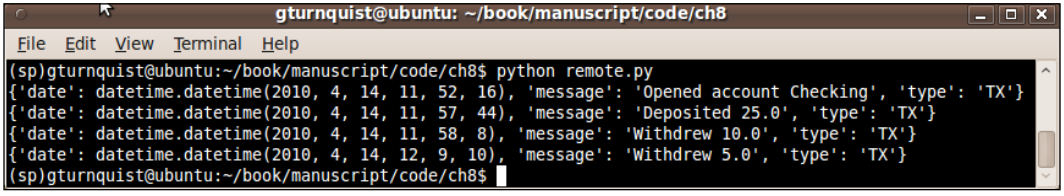
   In order to run this code, the following import statement must be added:

```
from springpython.remoting.pyro import *
```

3. Write a simple client script that will remotely connect to the public service, and retrieve one of the accounts.

```
from springpython.config import *
from springpython.remoting.pyro import *

class RemoteClient(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def client(self):
        service = PyroProxyFactory()
        service.service_url = "PYROLOC://localhost:7766/
springbank"
        return service

if __name__ == "__main__":
    from springpython.context import ApplicationContext
    ctx = ApplicationContext(RemoteClient())
    service = ctx.get_object("client")
    for row in service.history(1):
        print row
```

   This client script connects through Pyro module. It requests the history of **Account 1**, and then prints it out in raw form on the screen.



This easily fetches the raw data we requested. It is a perfect example of how to pipe raw data to other machines.

This form of raw data access has no security restrictions and would be ill advised for exposure to an untrusted network. This type of data would also be more secure if Pyro was configured with SSL. However, at this point in time, Spring Python does not support configuring Pyro with SSL.

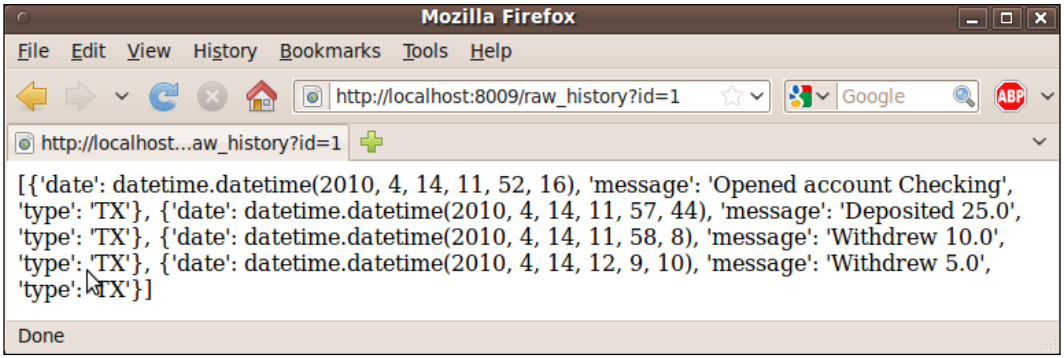
4. Let's create a raw version visible through the web layer.

```
@cherrypy.expose
def raw_history(self, id):
    return str(self.controller.history(id))
```

This basically takes the results of the `history` call, converts them to a string, and returns it for display with no extra HTML formatting.

5. Let's secure `raw_history` with the same protections as `history`.

```
@Object
def filter_security_interceptor(self):
    filter = FilterSecurityInterceptor()
    filter.auth_manager = self.auth_manager()
    filter.access_decision_mgr = self.access_decision_mgr()
    filter.sessionStrategy = self.session_strategy()
    filter.obj_def_source = [
        ("/raw_history.*", ["OWNER"]),
        ("/history.*", ["OWNER"]),
        ("/.*", ["ROLE_CUSTOMER", "ROLE_MGR",
                                            "ROLE_SUPERVISOR"])
        ]
    return filter
```



Now it's possible to write a web-based script that would get redirected to the login page. It must supply login credentials. After that, it can issue the URL necessary to retrieve Python's literal array of dictionaries. It could then convert it to a Python data structure, and harvest the data.

Creating audit logs

The logs that our application is currently writing are banking transactions. Another level of logging may be needed to analyze operations the controller is performing. This can service either performance analysis, post-mortem analysis, and for generating audit reports.

To do this, let's code an *aspect* that logs all of controllers (except for the log operation itself).

1. Code a simple aspect that logs anything except the `log` method using a copy of the controller.

```
class AuditInterceptor(MethodInterceptor):
    def __init__(self, controller):
        self.controller = controller

    def invoke(self, invocation):
        results = invocation.proceed()
        if invocation.method_name != "log":
            self.controller.log("AUDIT", -1,
                                "Method: %s Args: %s" % (\
                invocation.method_name, invocation.args))
        return results
```

   For this code to work, add the following import statement:

```
from springpython.aop import *
```

   This interceptor requires a copy of the `controller` to write its log message. All the other logging done earlier in this chapter was of type `TX`. These log entries are of type `AUDIT`, meaning they won't appear on any user's history or through the remote log access. Also, the account number is hard-coded **-1** since not all operations have an account.

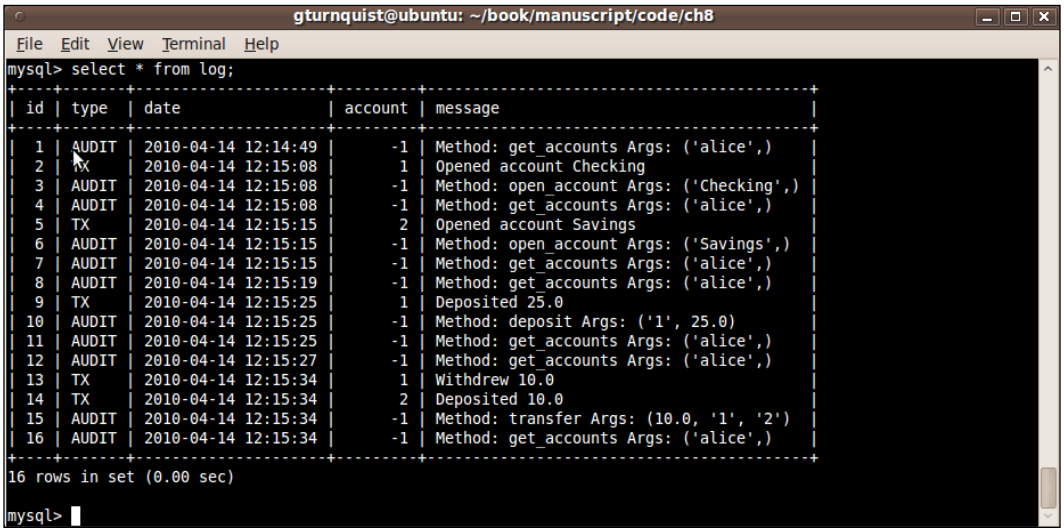
2. Update the definition of the controller in the application context in order to embed this interceptor.

```
@Object
def controller(self):
    target = SpringBankController(self.factory())
    return ProxyFactoryObject(
             target=target,
             interceptors=AuditInterceptor(target))
```

   This update to the application context smoothly replaces the `SpringBankController` with a `ProxyFactoryObject`. This proxy points at the real `controller` object, while also plugging in an instance of the `AuditInterceptor` we just coded.

   Also notice that the `AuditInterceptor` needs a copy of the controller. However, using `self.controller()` like all the other methods would generate a recursive stack error.

3. Let's use MySQL's command-line interface to look at the log table after several operations.



This provides a nice view of what happened on a transaction level, and also on a lower level from an auditing perspective.

It is left as an exercise for the reader to change the advice, so that only when a manager is viewing an account history does it write an audit log entry. It is also an exercise to build a view for the supervisor to view these audit trails.

Summary

With this case study, we have examined the various building blocks of Spring Python and utilized them to develop a simple, yet sophisticated banking site. While it may not have all the visual appeal, that task can easily be handed over to a web interface expert for improvement.

What is important is that key functional concepts have been implemented such as security and integrity. Customer data cannot be seen by other customers. It also has some simple banking protocols implemented such as overdraft prevention. Transactions are also implemented with little impact to the code.

The use cases involving managers and supervisors were not implemented with the intent that the reader could implement them as exercises.

With a functional banking application, the door is opened to further enhancements and implementation or more use cases needed to meet the customers' and the bank's needs.

In this chapter we learned how to:

- Wire together the components of our banking application using dependency injection
- Easily setup user authentication
- Restrict what pages a user can visit based on their role and whether or not they own the data being viewed
- Export raw data using Pyro to another trusted machine inside the enterprise
- Export raw data for the customer, to be used by some external tool
- Write an aspect that audits the database operations performed by the controller
- Easily mark up atomic transfer operations with the `@transactional` decorator

In the next chapter, we will take a look at Spring Python's command-line application, **coily**, which can help us build Spring Python applications quicker. We will also see how to write our own plugins to add functionality as we need.

9
Creating Skeleton Apps with Coily

Spring Python has many useful building blocks. In the last chapter we used these features in concert to build a simple banking application. This illustrated the bottom line task for software developers: delivering runnable applications.

To speed up the process for building apps, Spring Python provides the Python script `coily`. This script is built to support extensible plugins. The first plugin provided by the Spring Python team is **gen-cherrypy-app**, which is based on creating a skeleton CherryPy application using Spring Python IoC and security.

In this chapter, we will learn:

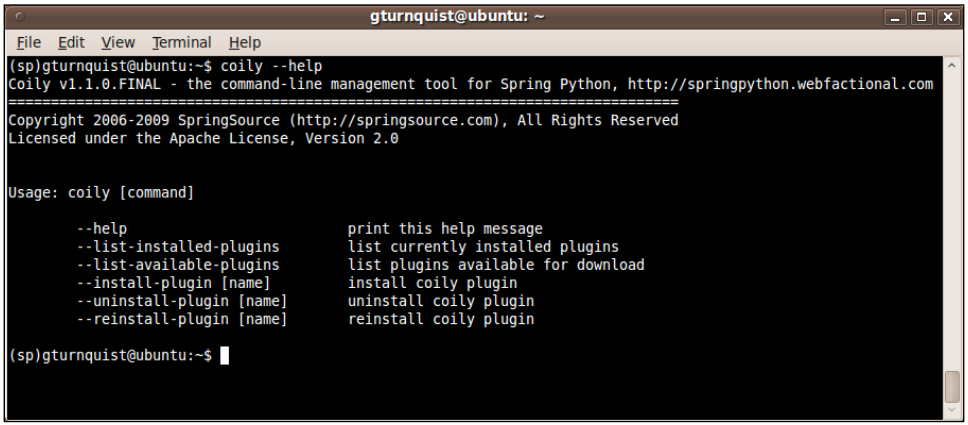
- The plugin driven approach of `coily`, which allows us to utilize plugins written by other developers or to write our own
- The easy-to-code requirements of creating a plugin
- Building a CherryPy application from scratch, with fully configured security, using the template-based `gen-cherrypy-app` plugin

Plugin approach of Coily

`coily` is a Python script designed from the beginning to provide a plugin based platform for building Spring Python apps. Another important feature is version control of the plugins. Developers should not have to worry about installing an out-of-date plugin that was designed for an older version of Spring Python. `coily` allows different users on a system to have different sets of plugins installed. It also requires no administrative privileges to install a plugin.

Key functions of coily

`coily` is included in the standard installation of Spring Python, as documented earlier in this book. To see the available commands, just ask for help.



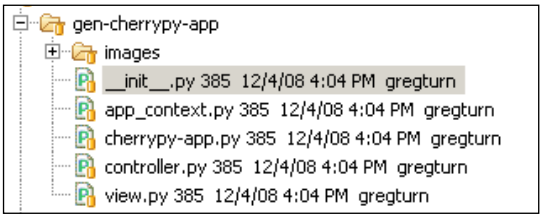
The following table elaborates these commands.

`--help`	Prints out the help menu. It is worth noting that when plugins are installed, they will also be listed as well.
`--list-installed-plugins`	Lists the plug-ins already installed *in this account*. Each installed plugin exists in a sub-folder in `HOME/.springpython`.
`--list-available-plugins`	Lists the plugins available for installation. `coily` is currently configured to search SpringSource's S3 site where Spring Python downloads are found for officially supported plugins. It also looks in the current directory where `coily` is being run, so that you can develop plugins locally.
`--install-plugin`	Installs a plugin by copying its files into `HOME/.springpython`.
`--uninstall-plugin`	Uninstalls the plugin by deleting its directory from `HOME/.springpython`.
`--reinstall-plugin`	Shortcut command that uninstalls a plugin and then installs it again. This is very useful when developing a new plugin in an iterative fashion.

Required parts of a plugin

A `coily` plugin closely resembles a Python package with some slight tweaks. This doesn't mean that a plugin is meant to be installed as a Python package. It is only a description of the folder structure.

Let's look at the layout of the `gen-cherrypy-app` plugin as an example.



Some parts of this layout are required, and other parts are not. The top folder is the name of the plugin.

- A plugin requires a `__init__.py` file inside the top directory.
- `__init__.py` must include a `__description__` variable. This description is shown when we run the `coily --help` command.
- `__init__.py` must include a command function, which is either a `create` or `apply` function. `create` is used when the plugin needs one argument from the user. `apply` is used when no argument is needed from the user.

Let's look at how `gen-cherrypy-app` meets each of these requirements.

1. We can already see from the diagram that the top level folder has the same name as our plugin.
2. Inside `__init__.py`, we can see the following help message defined.

   ```
   __description__ = "plugin to create skeleton CherryPy applications"
   ```

3. `gen-cherrypy-app` is used to create a skeleton application. It needs the user to supply the name of the application it will create. Again, looking inside `__init__.py`, the following method signature can be found.

   ```
   def create(plugin_path, name)
   ```

4. `plugin_path` is an argument provided to `gen-cherrypy-app` by `coily`, which points at the base directory of `gen-cherrypy-app`. This argument is also provided for plug-ins that use the `apply` command function.
5. `name` is the name of the application provided by the user.

It is important to recognize that `create` allows one command-line argument, but receives another one, `plugin_path` from `coily` itself to give the plug-in enough information to do its work. `apply` allows no command-line arguments, but still receives `plugin_path` from `coily`.

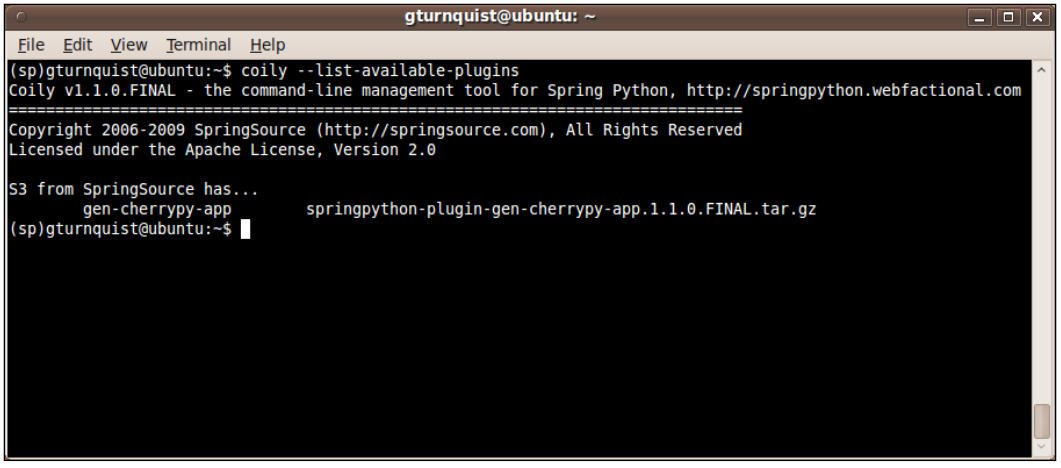
The rest of the files are not plug-in requirements, but instead are utilized by `gen-cherrypy-app` as shown in the next section.

Creating a skeleton CherryPy app

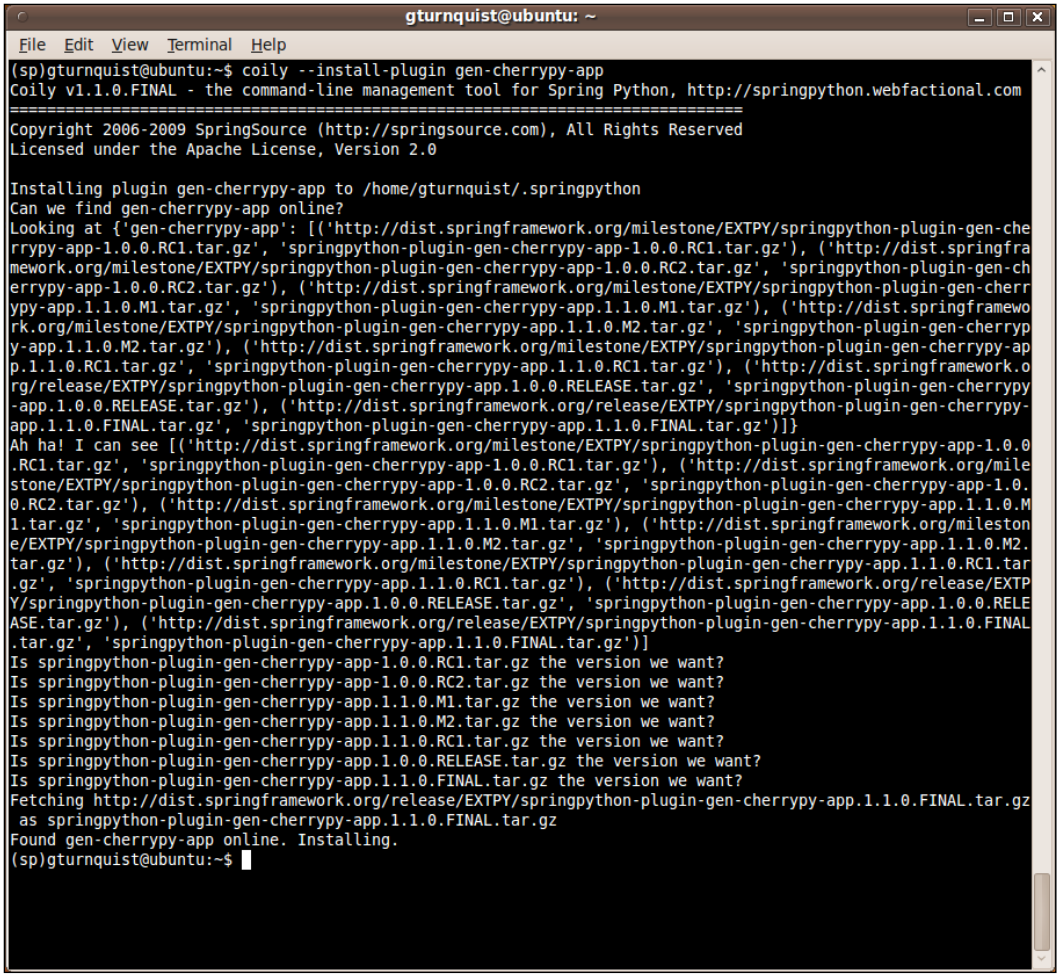
The rest of the files listed on the diagram above form a template of an application.

`app_context.py`	The Spring Python application context used to wire the generated application using decorator-driven `PythonConfig`
`cherrypy-app.py`	The main portion of the application that is runnable
`controller.py`	Contains the business logic of the application
`view.py`	Contains some the CherryPy rendering parts of the application
`images`	A subdirectory containing images used by the view layer of the application

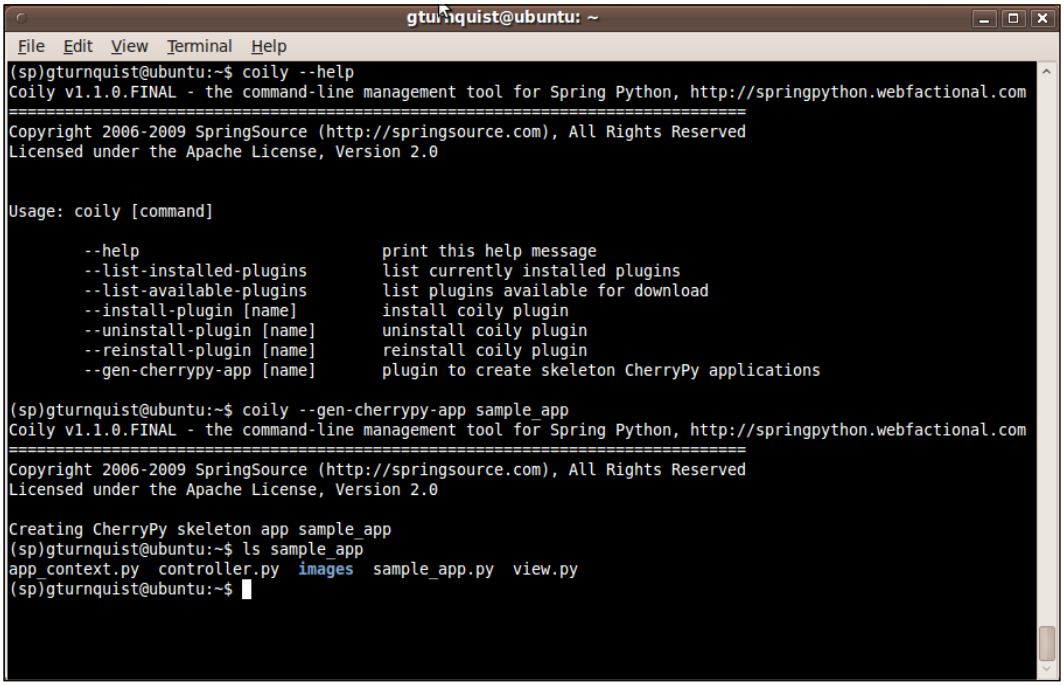
1. Before we create our CherryPy application, we are missing something. In the earlier screenshot, this plugin wasn't listed. We need to find it and install it. First, let's see how we can look up existing plugins.



2. Seeing `gen-cherrypy-app` listed, let's install it using `coily` without touching a browser.



3. Let's use the plugin to create a CherryPy application called `sample_app`. Since we just installed it, the plugin now shows up on `coily`'s help menu. With that in place, we can then run the command to create `sample_app`.

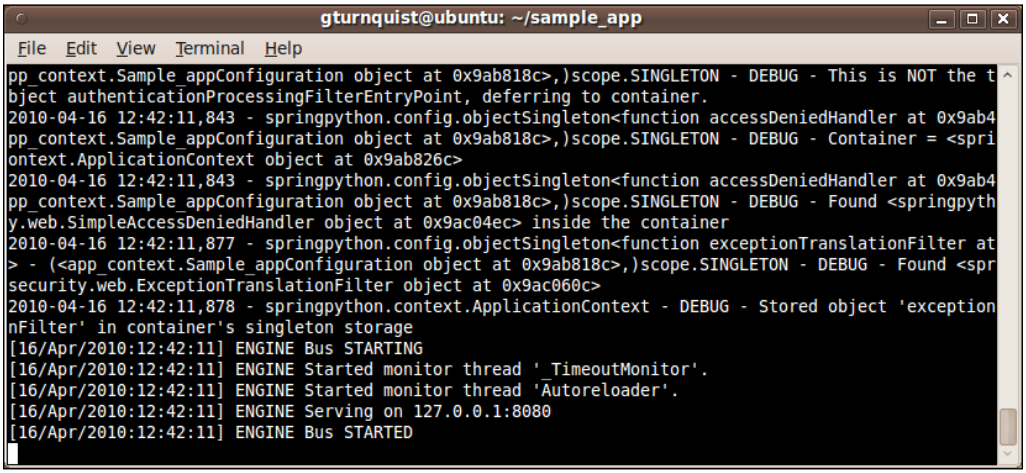


`gen-cherrypy-app` creates a directory named `sample_app` and copies the files listed above into it. It also does some transformations of the files, based on the argument `sample_app`.

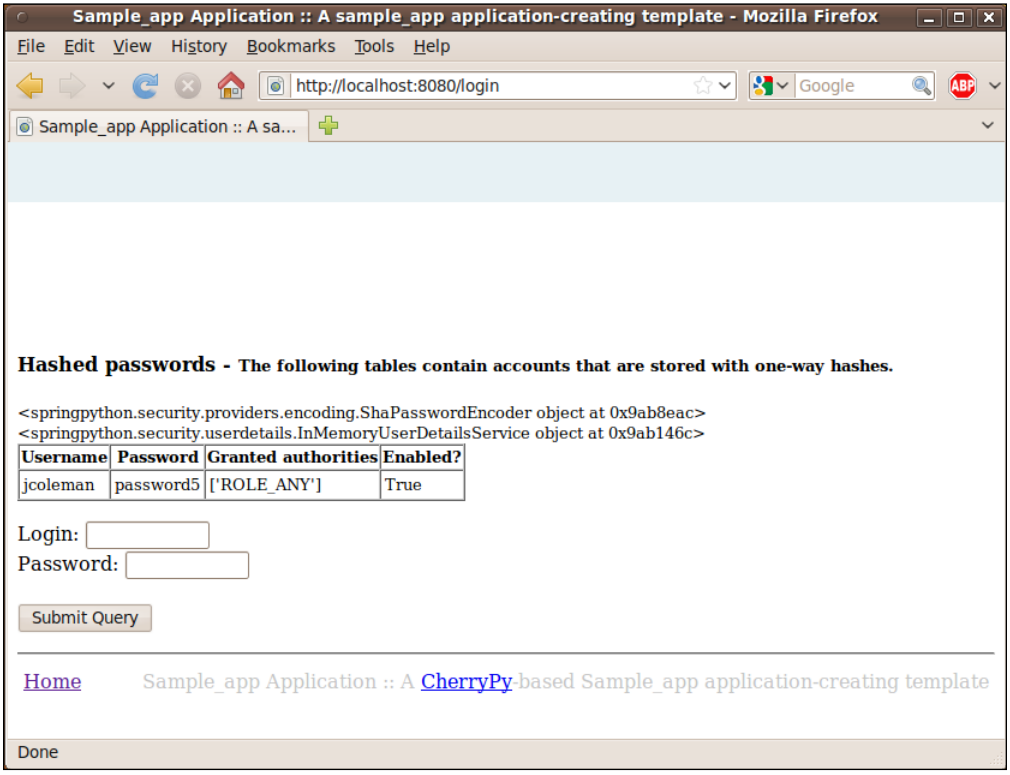
- It replaces all instances of `${name}` in each file with `sample_app`
- It replaces all instances of `${properName}` in each file with `Sample_app`
- It renames `cherrypy-app.py` as `sample_app.py`

4. To run the app, switch to the `sample_app` directory, and run the main script.

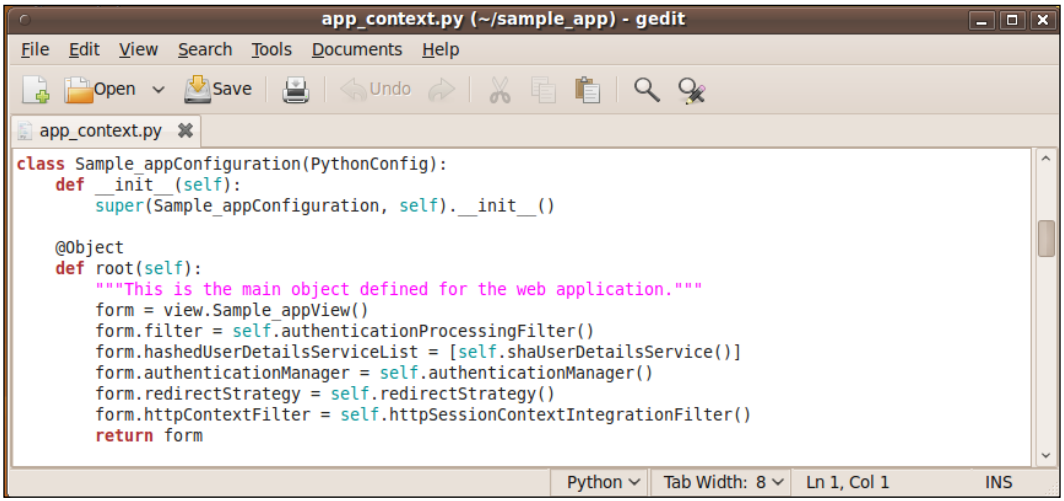
```
./sample_app.py
```



5. Now we can visit our running application at `http://127.0.0.1:8080`.



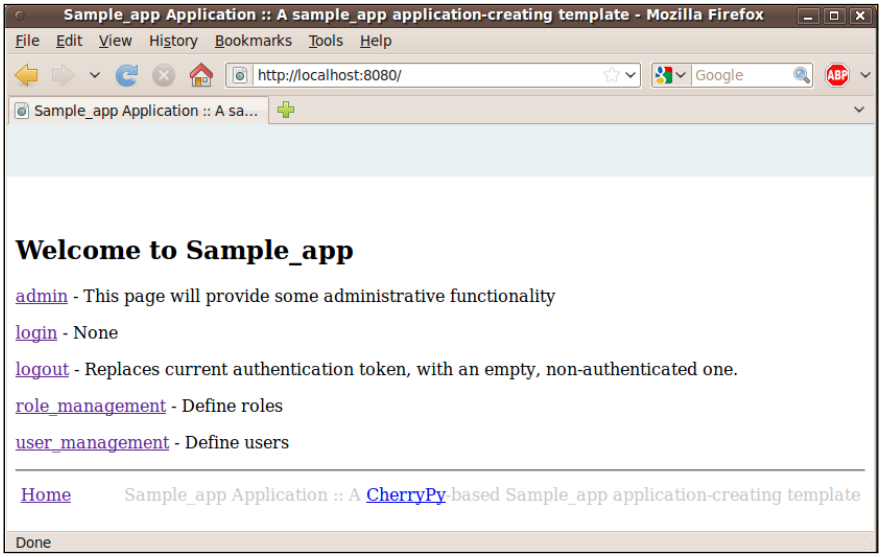
6. Let's inspect `app_context.py` and see some of the key features wired by `gen-cherrypy-app`.



This is the root object being wired. The rest of the configuration (not shown here) is mostly security configuration steps.

In *Chapter 6, Securing your Application with Spring Python*, it was pointed out that security configuration with Spring Python requires many steps. `gen-cherrypy-app` helps out by pre-wiring most of the security parts, allowing the user to modify as needed rather than build from scratch.

From here, we can log in with the hard-wired credentials to view our expandable application.



Currently, there are no controller objects. However, it would take little effort to add such a layer, as demonstrated in the previous chapter's case study.

This plugin is simple enough that most of the work spent in improving this plugin can be focused on the template files. This tactic is very useful to build other templates for other types of application.

Summary

`coily` was built to download, install, and utilize plugins. `gen-cherrypy-app` nicely creates a Spring-ified CherryPy application using a set of templates. This pattern is easy to replicate for other types of applications.

In this chapter, we have learned:

- The basic `coily` commands used to install and uninstall plugins
- The structure of a plug-in, including defining the description shown on `coily`'s help screen and the functions needed to process a command
- That using template files and pattern substitution makes it easy for `gen-cherrypy-app` to generate CherryPy applications

In the next chapter, we will look at how to integrate Spring Python with a Java application.

10
Case Study II—Integrating Spring Python with your Java Application

In this book, we have explored the many facets of Spring Python while examining code samples. So far, all the code we explored was created using CPython, the original implementation of the Python language.

In this day and age, polyglot programming has attracted an ever increasing interest in the developer community. Java developers are looking at other options to increase developer productivity while still retaining access to the Java ecosystem. Python developers are looking at how to run systems on scalable assets such as the Google App Engine.

Spring Python is coded in pure Python, making it easy to run inside Jython. This opens the door to integration, allowing developers to easily mix Java and Python components with the lightweight power of Spring Python.

In this chapter, we will learn how to:

- Build a flight reservation system front end using the pythonic CherryPy web application framework
- Build a flight reservation system back end using Java
- Connect the two together using Spring Python, Jython, and Pyro

Building a flight reservation system

For this chapter, we will explore integrating Java and Python together while building a flight reservation system. The first step towards building a system like this is to show the customer existing flights and allow the customer to filter the data. While many more features are needed to implement a usable system, this feature will be the focus of this demonstration.

We have already explored using the CherryPy web application framework and will use it again to create the front end for this system. For the back end, we will develop a persistence layer using Java components, while wiring everything together using Spring Python's IoC container and run everything from inside Jython. The other key feature that we will explore is how to integrate pure Java with pure Python components using Spring Python and Pyro.

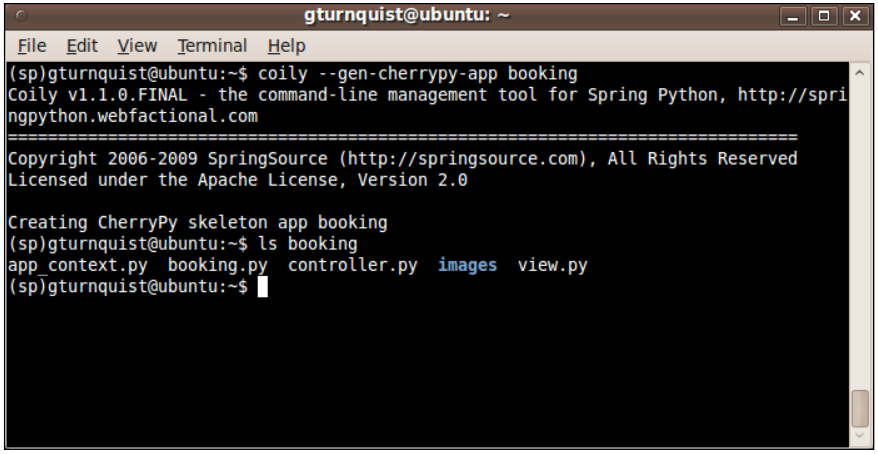
Building a web app the fastest way

How would you look at building our Python frontend? One popular approach is to build small, testable blocks that are easily composable. From this perspective, the user interface would be the last step.

Another tactic is to focus on the customer's view and develop the screens, followed by filling in the back end. Since our goal is to test drive the features of Spring Python, let's take this more visual approach.

To speed up our application, let's use Spring Python's command-line coily tool, which we explored earlier.

1. Using `coily`, let's create a booking application



This command generates a set of files:

File name	Description
`booking.py`	Runable CherryPy application.
`app_context.py`	Spring Python application context, that contains the wiring for our components.
`view.py`	Contains all the web parts, including HTML.
`controller.py`	Contains functions that the view accesses to generate dynamic content for the screens.

This auto-generated application contains pre-wired components, including security, some basic links, and the bootstrapping needed to launch the CherryPy web server engine. It just needs some minor modifications before we start coding our features.

2. Remove any licensing comments embedded in the app. While the template files themselves are marked with a license, this application may not be released under the same license
3. Ease security restrictions defined in `app_context.py`, allowing anonymous users access the main parts of site, while subjecting any link underneath `/customer/*` to full security and requiring `ROLE_CUSTOMER`

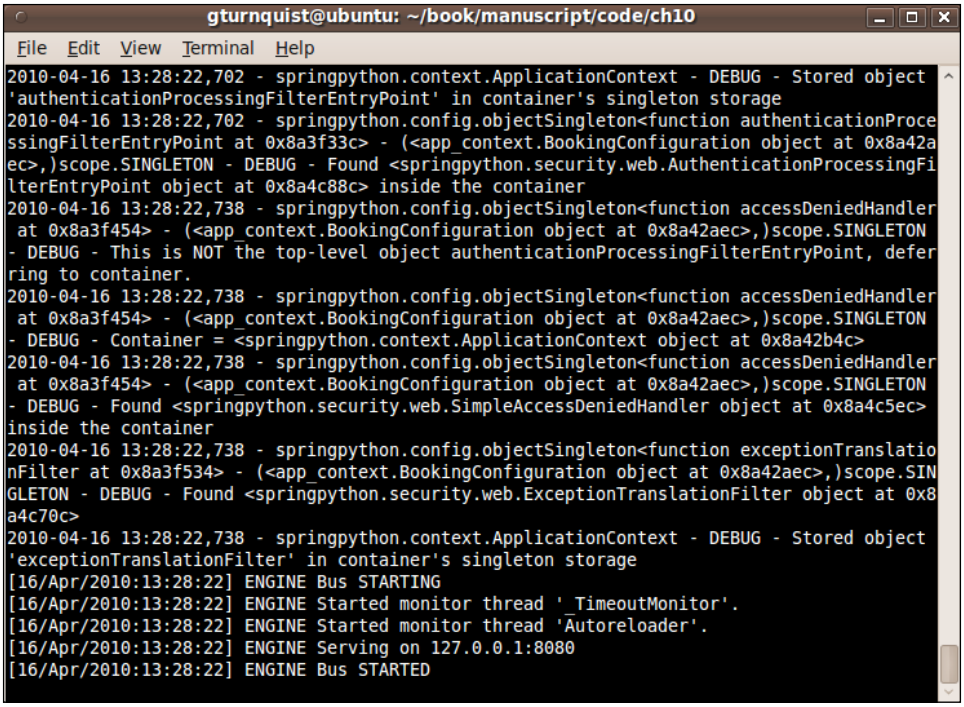
```
@Object
def filterChainProxy(self):
 return CP3FilterChainProxy(filterInvocationDefinitionSource =
       [
       ("/images.*", []),
       ("/html.*",   []),
       ("/accessDenied.*", []),
       ("/login.*",  ["httpSessionContextIntegrationFilter"]),
       ("/customer/.*", ["httpSessionContextIntegrationFilter",
                         "exceptionTranslationFilter",
                         "authenticationProcessingFilter",
                         "filterSecurityInterceptor"])
        ])

@Object
def filterSecurityInterceptor(self):
    filter = FilterSecurityInterceptor()
    filter.auth_manager = self.authenticationManager()
    filter.access_decision_mgr = self.accessDecisionManager()
```

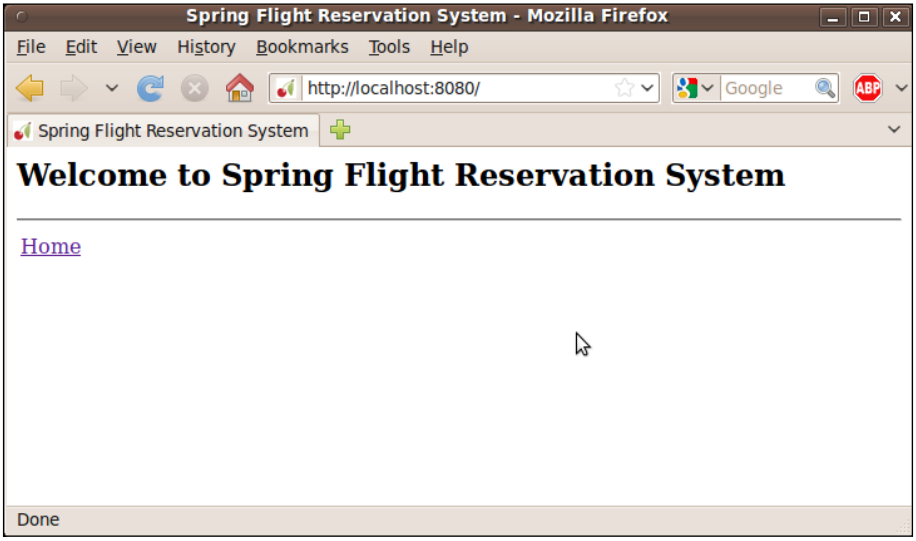
```
        filter.sessionStrategy = self.cherrypySessionStrategy()
        filter.obj_def_source = [
                                ("/customer/.*", ["ROLE_CUSTOMER"])
                                 ]
        return filter
```

4. Remove the images and CSS code and replace them with a simpler header and footer:

```
def header():
    return """
        <html>
        <head>
        <title>Spring Flight Reservation System</title>
        </head>

        <body>
            """

def footer():
    return """
        <hr>
        <table style="width:100%"><tr>
                <td><A href="/">Home</A></td>
        </tr></table>
        </body>
        """
```

5. Remove all methods from the view layers that were exposed using `@cherrypy.expose` (except `index`, `login`, and `logout`)
6. Let's launch this web app using Python and then visit it by opening a browser at `http://localhost:8080`





We now have a clean slate from which we can start fleshing out the features of our system.

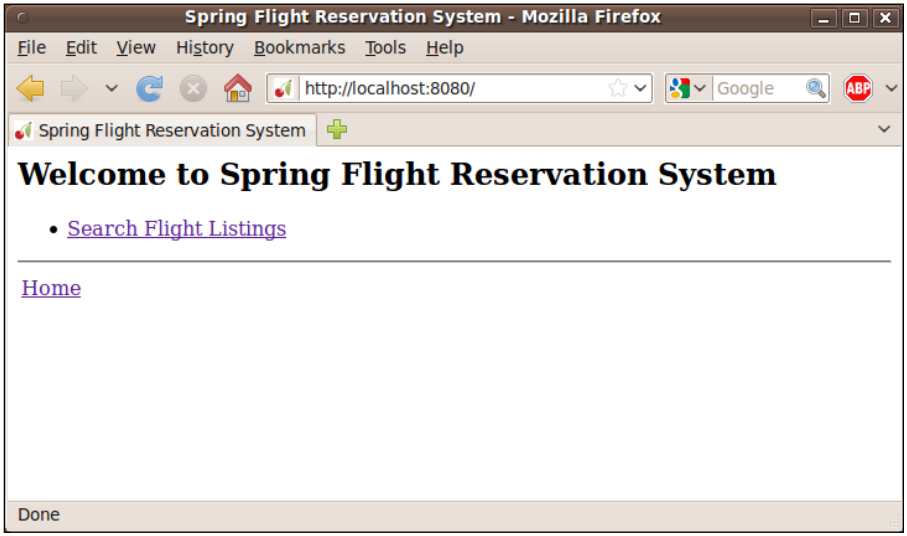
Looking up existing flights

What is the first piece of data that we need for a flight reservation system? Flight listings! Since customers usually need to search based on a date or location, let's fetch some flight data and then build some search features.

1. Let's add a link on the main page to take us to a search page.

```
    @cherrypy.expose
    def index(self):
        """This is the root page for your booking app. Its default
includes links to all other exposed links automatically."""

        return header() + """
            <H2>Welcome to Spring Flight Reservation System</H2>
            <P>
            <ul>
                <li><a href="flight_listings">Search Flight
                    Listings</a></li>
            </ul>
            <P>""" + footer()
```

   We have now created an HTML bulleted list with one entry to **Search Flight Listings**.



2. Before we try to display any flight information, let's define a simple Python class to contain this information:

```
class Flight(object):
    def __init__(self, flight=None, departure=None, arrival=None):
        self.flight = flight
        self.departure = departure
        self.arrival = arrival
```

3. Now that we've created a simple model for our flight data, let's display it on the web page. To do that, let's add a `flight_listings` function and expose it using `@cherrypy.expose`:

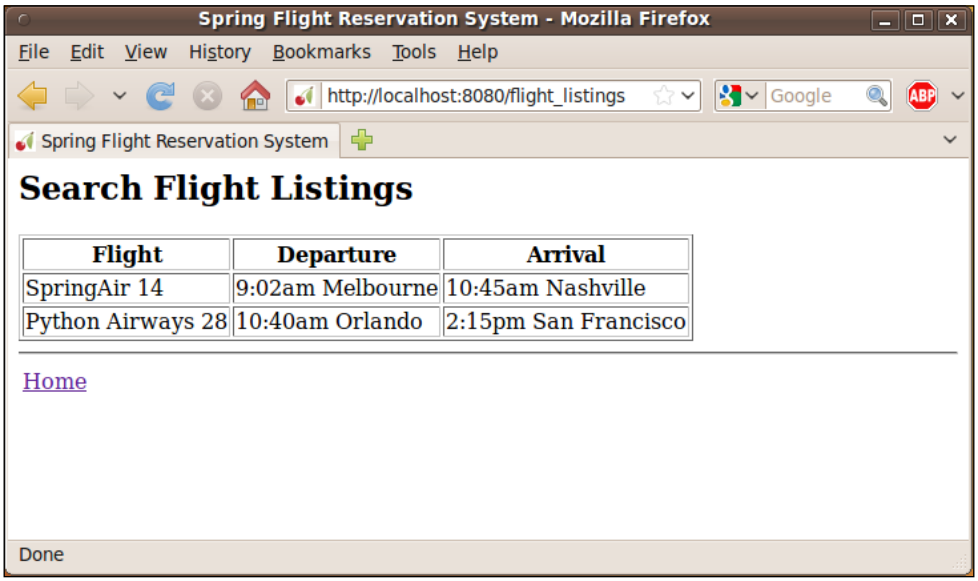
```
    @cherrypy.expose
    def flight_listings(self):
        page = header() + """
            <H2>Search Flight Listings</H2>
            <table border="1">
                <tr><th>Flight</th><th>Departure</th>
                                <th>Arrival</th></tr>
            """

        for f in self.controller.flights():
          page += "<tr><td>%s</td><td>%s</td><td>%s</td></tr>" % \
                (f.flight, f.departure, f.arrival)

        page += "</table>" + footer()

        return page
```

We haven't created a search box yet. Before we do, let's ask the controller to give us all flight data.

4. We can't click on the link yet because there is no controller, so let's build one and have it supply some sample data

```
from model import *

class BookingController(object):
    def __init__(self):
        self.data = []
        self.data.append(Flight("SpringAir 14", "9:02am
Melbourne", "10:45am Nashville"))
        self.data.append(Flight("Python Airways 28", "10:40am
Orlando", "2:15pm San Francisco"))

    def flights(self):
        return self.data
```

There isn't much there, its just a simple Python array containing one `Flight` instance. But we want to build this up from small, simple parts that we can easily visualize. With all these parts in place, let's look at the web page again.



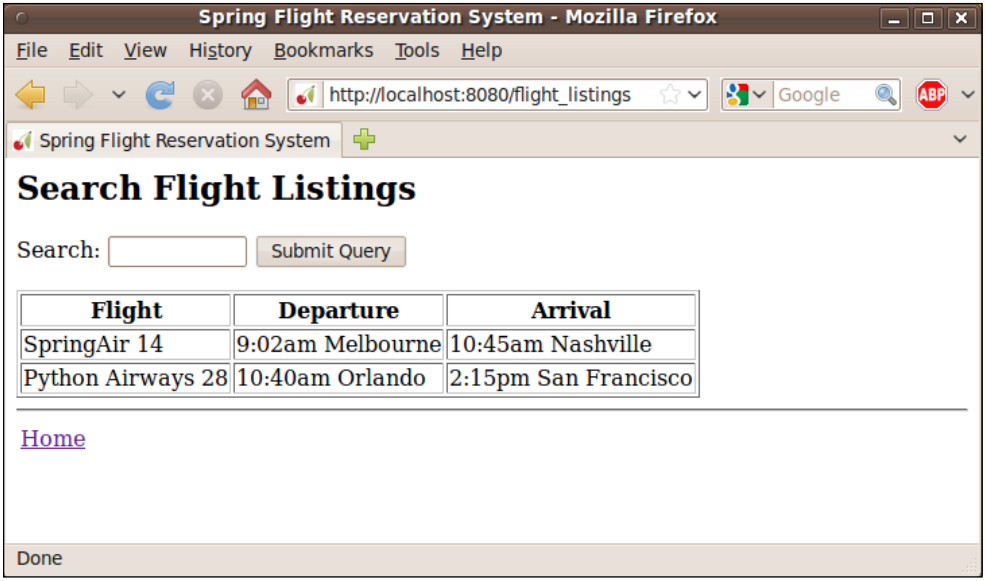
Now we can see this small bit of sample flight data.

5. Let's add a search box to provide some minimal filtering capability

```
@cherrypy.expose
def flight_listings(self, criteria=""):
    page = header() + """
        <H2>Search Flight Listings</H2>
        <form method="POST" action="/flight_listings">
            Search: <input type="text"
                           name="criteria"
                           value="%s"
                           size="10"/>
            <input type="submit"/>
        </form>
        <table border="1">
            <tr>
                <th>Flight</th>
                <th>Departure</th>
                <th>Arrival</th>
            </tr>
        """ % criteria
```

```
        for f in self.controller.flights(criteria):
          page += "<tr><td>%s</td><td>%s</td><td>%s</td></tr>" % \
                (f.flight, f.departure, f.arrival)

        page += "</table>" + footer()

        return page
```

We added a text input with a button to the top of the screen. The search criterion is now passed into the controller object to do any filtering.



6. How would you utilize the text entered by the user to filter in the controller? Should we be case sensitive or perhaps relax that a bit? Which fields would you want to compare against if you were using it as a customer? Here is one way you could use it:

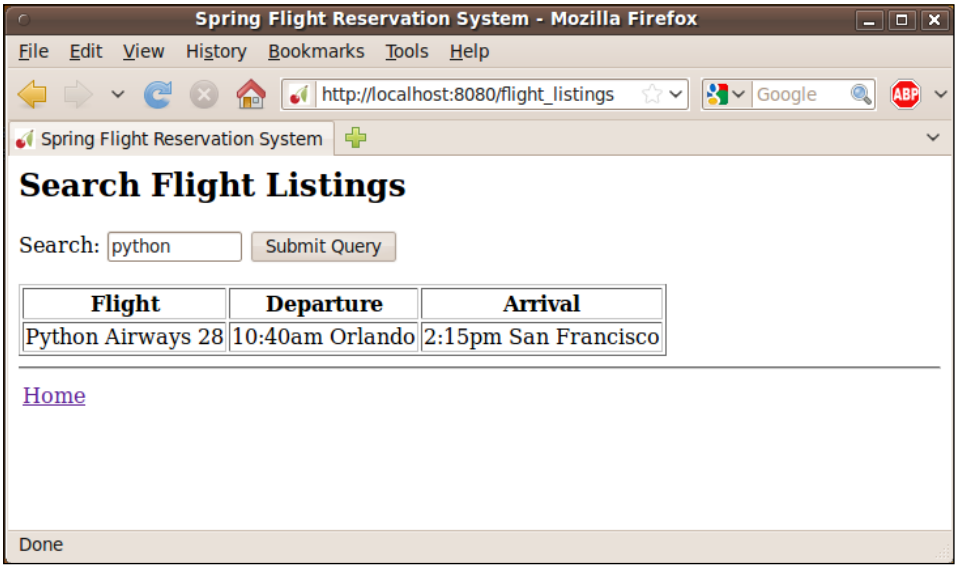
```
    def flights(self, criteria):
        return [f for f in self.data \
                if criteria.lower() in str(f).lower()]
```

This grabs the sample data and applies a filtering list comprehension. The filter applies `lower()` to both the criteria as well as the stringified version of the flight record.

7. One way to let the customers search is against all the data in the flight record

```
def __str__(self):
    return "%s %s %s" % \
           (self.flight, self.departure, self.arrival)
```

With all these changes, we can now do a simple flight search.



What are some other search options you would consider to make this more sophisticated?

Moving from sample Python data to real Java data

So far, we are looking at a couple rows of sample data. A more realistic scenario would be tapping some type of a live system, listing real-time flights. What would you do if the data was stored in a backend database and a service was already coded in Java?

- Would you attempt to bypass the Java system and talk directly to the database? What risks are involved with this approach?
- Would you ditch the Python front end we just built and instead embark on a Java-based portal? What if this had been several months of effort?

Let's look at how Jython can integrate these two worlds together. Does Jython support all the Python components that we have used so far?

Jython in late 2009 caught up to Python 2.5 and has many useful features. But many 3rd party Python libraries still can't run on Jython despite this significant improvement. Many libraries utilize Python's C-extension capability, but this isn't supported in Jython. There are also platform assumptions involved with things like threads that defer in CPython compared to the JVM which Jython runs on.

At the time of writing, CherryPy has issues running on Jython without some small tweaks. Sqlite support also isn't available yet. It's possible to tap Java components from Jython, but it's not realistic to assume all the Python components we plan to use will run.

Spring Python makes it easy to bridge this gap through its integration with the Pyro remoting library.

1. To move our data access controller over to Java, we will also need to move the `Flight` record as well.

```
public class Flight {

  private long id;
  private String flight;
  private String departure;
  private String arrival;

  public long getId() {
    return this.id;
  }

  public void setId(long id) {
    this.id = id;
  }

  public String getFlight() {
    return this.flight;
  }

  public void setFlight(String flight) {
    this.flight = flight;
  }

  public String getDeparture() {
```

```
    return this.departure;
  }

  public void setDeparture(String departure) {
    this.departure = departure;
  }

  public String getArrival() {
  return this.arrival;
  }

  public void setArrival(String arrival) {
    this.arrival = arrival;
  }

  public String toString() {
    return flight + " " + departure + " " + arrival;
  }
}
```

2. Next, let's rewrite our controller in Java. To make things easier, we will use the original Spring Framework to code our query

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

public class FlightDataSystem {

  private JdbcTemplate jdbcTemplate;

  public FlightDataSystem(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
  }

  public List<Flight flights(String criteria) {
    return jdbcTemplate.query(
        "SELECT id, flight, departure, arrival " +
        "FROM flights",
        new RowMapper<FlightData>() {
```

```
            @Override
            public FlightData mapRow(final ResultSet rs, int row)
                throws SQLException {
              return new Flight() {{
                setId(rs.getLong("id"));
                setFlight(rs.getString("flight"));
                setDeparture(rs.getString("departure"));
                setArrival(rs.getString("arrival"));
              }};
            }
          });
    }
}
```

Notice how this Java code has double braces? This is an anonymous subclass of `Flight`, with an initializing block of code. It's a convenient way to remove some of the boilerplate of populating a **POJO (Plain Old Java Object)**.

3. In order to have our controller talk to a MySQL database as well as utilize the Spring Framework, we will need some extra `jar` files. This includes:
    - `org.springframework.beans`
    - `org.springframework.core`
    - `org.springframework.jdbc`
    - `org.springframework.transaction`
    - `commons-dbcp`
    - `commons-logging`
    - `commons-pool`
    - `mysql-connector-java`

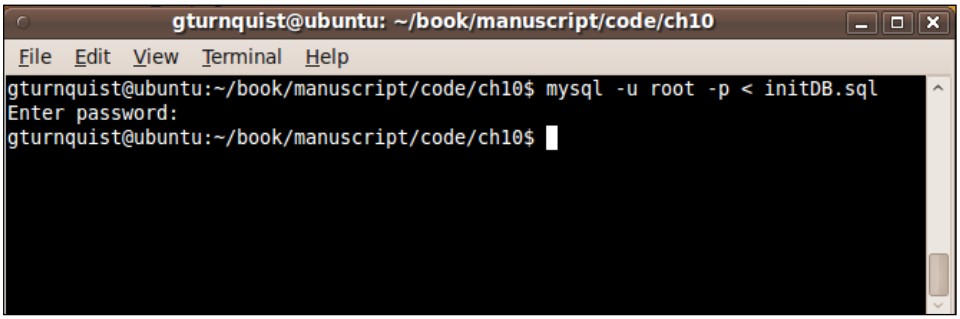
    It is an exercise for the reader to write a script to re-build the Java parts, considering the jar files.

4. Before we can test this code, we need to setup our database:

```
DROP DATABASE IF EXISTS booking;

CREATE DATABASE booking;

GRANT ALL ON booking.* TO booking@localhost IDENTIFIED BY
'booking';
```

```
USE booking;

CREATE TABLE flights (
  id INT(4) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  flight VARCHAR(30),
  departure VARCHAR(30),
  arrival VARCHAR(30)
);

INSERT INTO flights (flight, departure, arrival) values
('SpringAir 14', '9:02am Melbourne', '10:45am Nashville');
INSERT INTO flights (flight, departure, arrival) values ('Python
Airways 28', '10:40am Orlando', '2:15pm San Francisco');
```

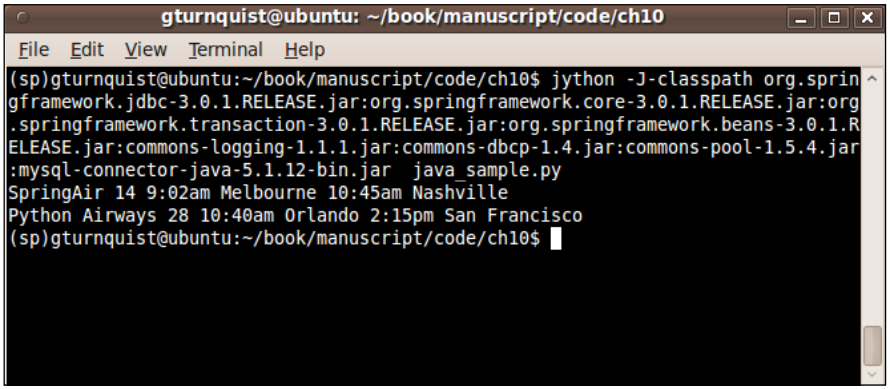
The following screenshot shows this script being run to setup and load data:



5. Now that we've migrated the controller code to Java and set up the schema, let's write a simple Jython script to print out the flight data

```
import FlightDataSystem
from org.apache.commons.dbcp import BasicDataSource

source = BasicDataSource()
source.driverClassName = "com.mysql.jdbc.Driver"
source.url = "jdbc:mysql://localhost/booking"
source.username = "booking"
source.password = "booking"

system = FlightDataSystem(source)

for f in system.getFlightData():
    print f
```

To run it, we need to include several `jar` files.



You can see the command typed, and the output of printing the flight data to the console.

There is one other key factor to deal with before we can wire things together: *serialization of the data*. When our Java code returns an array of Java objects, it has to be serialized by Pyro, sent over the wire, and then deserialized by CPython. For the scalars, this is not a problem. Jython handles it easily. But moving class objects requires class definitions on both sides of the wire. CPython doesn't have access to our Java `Flight` class and the Java code doesn't have access to the CPython class. We need a thin piece of code in the middle that can call our Java service and convert the list of Java objects into Python objects. A Jython wrapper would be perfect.

```
from model import *

class JythonFlightDataSystem(object):
    def __init__(self, java_system):
        self.java_system = java_system

    def flights(self, criteria):
        return [Flight(f.flight, f.departure, f.arrival) \
                for f in self.java_system.flights(criteria)]
```

This wrapper class has the same signature. Running it in Jython will give it access to the Java classes as well as our Python classes.

6. Let's wire up our service using a Spring Python IoC configuration and then expose it using `PyroServiceExporter`

```
import logging
from springpython.config import *
from springpython.context import *
from springpython.remoting.pyro import *

import FlightDataSystem
from org.apache.commons.dbcp import BasicDataSource
from java_wrapper import *

class JavaSystemConfiguration(PythonConfig):
    def __init__(self):
        super(JavaSystemConfiguration, self).__init__()

    @Object
    def data_source(self):
        source = BasicDataSource()
        source.driverClassName = "com.mysql.jdbc.Driver"
        source.url = "jdbc:mysql://localhost/booking"
        source.username = "booking"
        source.password = "booking"
        return source

    @Object
    def exported_controller(self):
        exporter = PyroServiceExporter()
        exporter.service_name = "JavaFlightDataSystem"
        exporter.service = self.controller()
        return exporter

    @Object
    def controller(self):
        java_system = FlightDataSystem(self.data_source())
        wrapper = JythonFlightDataSystem(java_system)
        return wrapper
```

7. With this configuration, we are using the Apache Commons `BasicDataSource` to create a connection pool to access the MySQL database

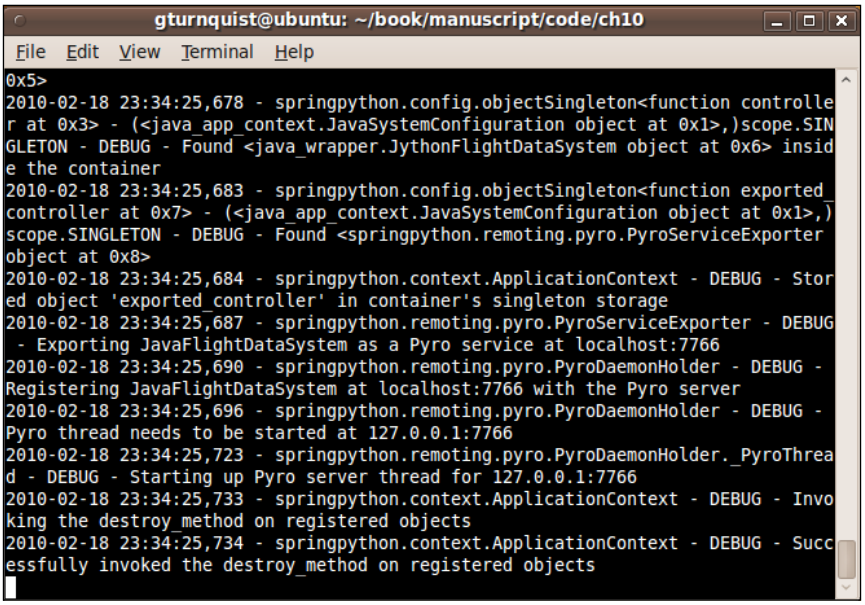
8. We create an instance of our Java-based `FlightDataSystem`. But instead of returning that, we inject it into our `JythonFlightDataSystem`, so that it can translate between Java and Python records
9. We also create a `PyroServiceExporter`, and have it target our `controller()`. It is advertised under the name `JavaFlightDataSystem`
10. We need a boot script to launch our Jython service

```
import logging
import os
import java_app_context
from springpython.context import ApplicationContext

if __name__ == '__main__':
    logger = logging.getLogger("springpython")
    loggingLevel = logging.DEBUG
    logger.setLevel(loggingLevel)
    ch = logging.StreamHandler()
    ch.setLevel(loggingLevel)
    formatter = logging.Formatter("%(asctime)s - %(name)s -
%(levelname)s - %(message)s")
    ch.setFormatter(formatter)
    logger.addHandler(ch)

    applicationContext = ApplicationContext(\
        java_app_context.JavaSystemConfiguration())
    controller = applicationContext.get_object("controller")
```

11. Running Jython scripts with 3rd party libraries can be a little tricky. To make sure that the JVM can load classes from `jar` files, the `jar` files need to be passed in to the JVM as classpath entries

```
jython -J-classpath org.springframework.jdbc-3.0.1.RELEASE.jar:
org.springframework.core-3.0.1.RELEASE.jar:org.springframework.
transaction-3.0.1.RELEASE.jar:org.springframework.beans-
3.0.1.RELEASE.jar:commons-logging-1.1.1.jar:commons-dbcp-1.4.jar:
commons-pool-1.5.4.jar:mysql-connector-java-5.1.12-bin.jar java_
system.py
```

12. The `-J` argument feeds parameters to the JVM. This is necessary so that the class loaders can pull classes from the `jar` files



We can see that Pyro has been started and is ready to serve data to any clients.

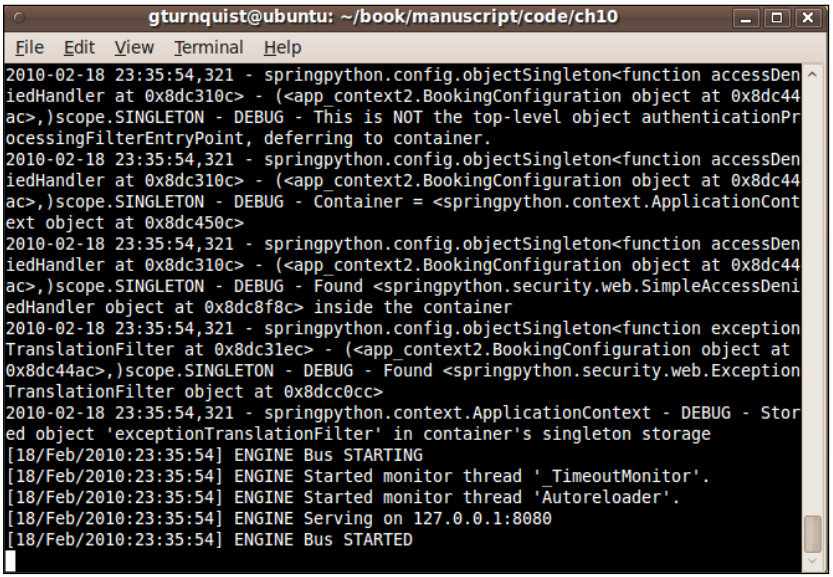
13. With our Jython data feed running, we can now replace the original controller definition inside our application context with a `PyroProxyFactory`

```
@Object
def controller(self):
    java_system = PyroProxyFactory()
    java_system.service_url = \
            "PYROLOC://localhost:7766/JavaFlightDataSystem"
    return java_system
```

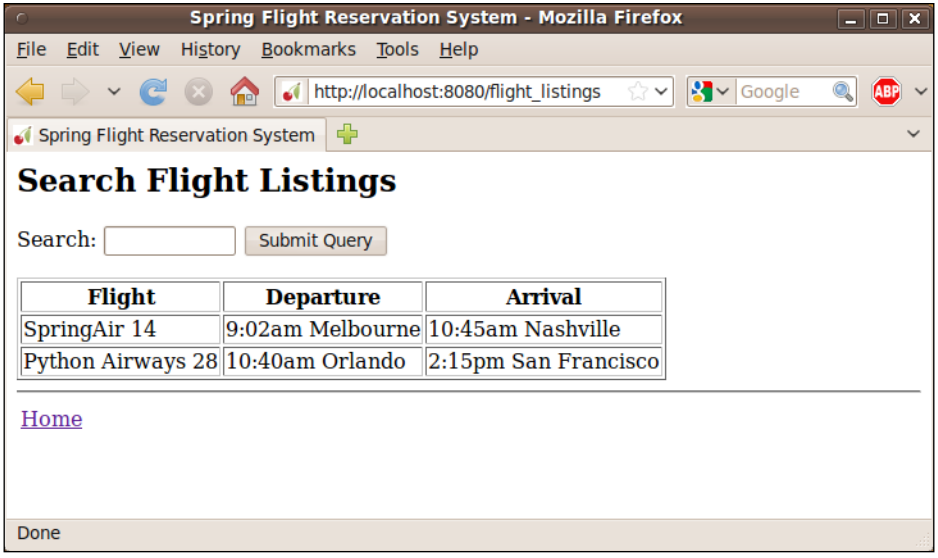
14. By adding the following import statement, we are now able to re-launch our booking application in one shell, and the Jython service in another

```
from springpython.remoting.pyro import *
```

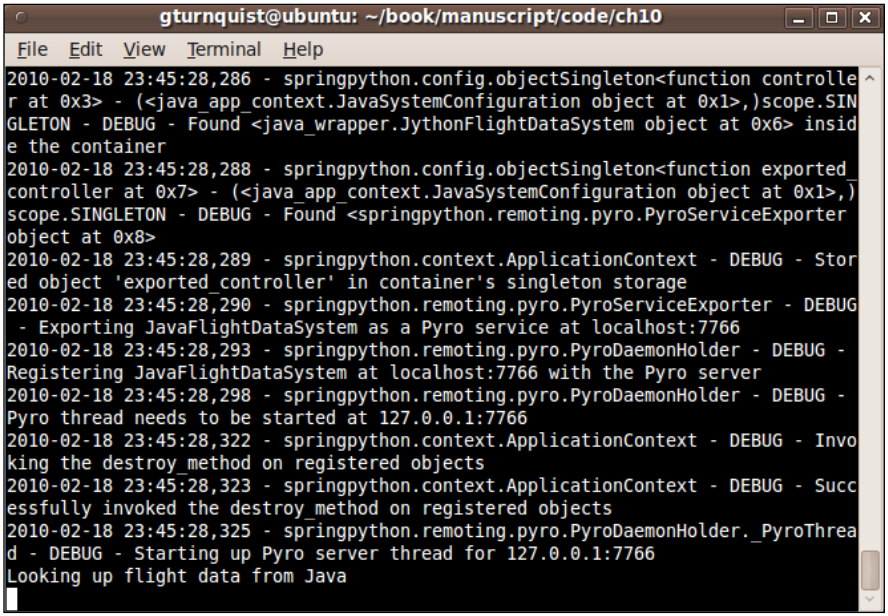
15. Now, let's re-launch our `booking.py` application using CPython, and look at the screens



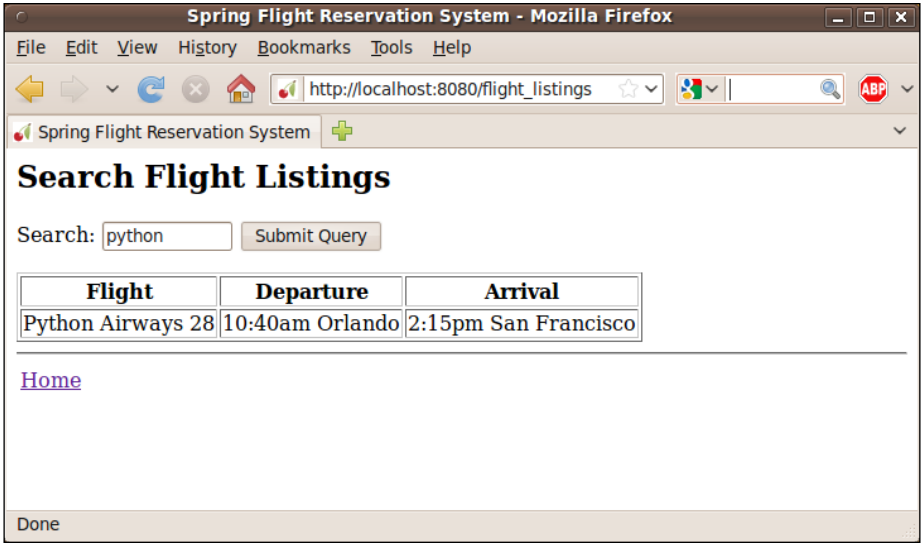
If we revisit the flight data page, it looks the same:



However, looking closer at the console output from the Jython script, we see a message indicating our Java code is being called.



The filtering logic has also been moved to the Java service, so we get the response.



Issues with wrapping Java code

One part of this solution that wasn't very elegant was the wrapper code that transformed between CPython and Java flight data. It is a necessary step to convert between these types when linking these systems together. While a wrapper code solved our immediate problem, the maintenance of that code can grow.

Since it is very likely that other classes require the same attention, whether they are input arguments or results, a solution that is easier to maintain is needed. The behavior involves wrapping the Java service and checking inputs and outputs. This is exactly the type of use case that aspect oriented programming is well suited for.

It is left as an exercise for the reader to code an aspect that can scan inputs and outputs and apply some sort of mapping between the CPython types and Java types, and wire it up to replace the java wrapper.

Summary

We are now running the frontend web server in Python and the backend on the JVM. While this example isn't flashy or necessarily dynamic, it shows the capability of integrating these two systems together.

Since Jython is all about making it possible to run compiled class files, it becomes possible to replace the Java-based database code with something more sophisticated like a Groovy-based feed reader using its terse `XmlSlurper` module or a Scala-based system that monitors RSS feeds from multiple airlines.

At the same time, we can keep coding using our favorite Python tactics in CherryPy, or perhaps another web framework of choice. We can also write useful command-line scripts that access the same back end information.

A key point is that we didn't have to do the wiring ourselves. Instead of managing the low level API of Pyro, Spring Python did the work for us.

Spring Python offers many valuable features that we have visited throughout this book. There are many useful abstractions. But the abstractions aren't one-to-one with the ones provided by Spring Java. Instead, it focuses on Python libraries. For example, Spring Python integrates with Pyro, while Spring Java integrates with RMI. And this makes sense, because Python developers aren't as likely to integrate with an RMI-based system as they are to link together Python components.

This book also introduces the powerful concept of dependency injection. Spring Python and Spring Java share this philosophy. Throughout the various examples in this book, we saw how to decouple components and wiring them together in a container. This made it easy to test, introduce services with little impact, and allow rewiring as needs change. This gives developers the flexibility they need to adapt to changing requirements.

In this chapter we have learned how to:

- Build a front end in CPython using CherryPy and then link it to a back end written in Java
- Create the necessary adapter code to handle serialization between CPython and Java
- Replace a Python-to-Python call with a Python-to-Java call with no impact to the front end or the back end
- Compile Java code and run it from Jython using 3rd party libraries

I hope you have enjoyed this book, and that it will inspire you in new ways to design, code, and maintain more powerful systems.

Index

Symbols

A

B

C

D

E

O

P

R

S

T

U

V

W

Thank you for buying
Spring Python 1.1

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

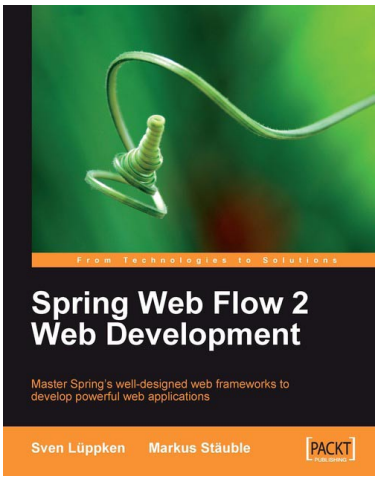
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



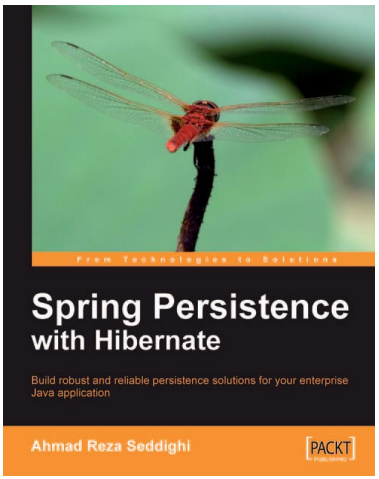
Spring Web Flow 2 Web Development

ISBN: 978-1-847195-42-5 Paperback: 200 pages

Master Spring's well-designed web frameworks to develop powerful web applications

1. Design, develop, and test your web applications using the Spring Web Flow 2 framework
2. Enhance your web applications with progressive AJAX, Spring security integration, and Spring Faces
3. Stay up-to-date with the latest version of Spring Web Flow
4. Walk through the creation of a bug tracker web application with clear explanations

---



Spring Persistence with Hibernate

ISBN: 978-1-849510-56-1 Paperback: 460 pages

Build robust and reliable persistence solutions for your enterprise Java application

1. Get to grips with Hibernate and its configuration manager, mappings, types, session APIs, queries, and much more
2. Integrate Hibernate and Spring as part of your enterprise Java stack development
3. Work with Spring IoC (Inversion of Control), Spring AOP, transaction management, web development, and unit testing considerations and features
4. Covers advanced and useful features of Hibernate in a practical way