Spring Python - Reference Documentation

Version 1.1.1.BUILD-20101109171232

Table of Contents

Preface

1. Overview

1.1. Key Features ... 1

1.2. What Spring Python is NOT ... 2

1.3. Support ... 2

1.3.1. Forums and Email ... 2

1.3.2. IRC ... 2

1.4. Downloads / Source Code ... 2

1.5. Licensing ... 3

1.6. Spring Python's team ... 3

1.6.1. How to become a team member ... 3

1.7. Deprecated Code ... 4

2. The IoC container

2.1. External dependencies ... 6

2.2. Container ... 6

2.2.1. ObjectContainer vs. ApplicationContext ... 7

2.2.2. Scope of Objects / Lazy Initialization ... 7

2.3. Configuration ... 8

2.3.1. XMLConfig - Spring Python's native XML format ... 9

2.3.2. YamlConfig - Spring Python's YAML format ... 16

2.3.3. PythonConfig and @Object - decorator-driven configuration ... 23

2.3.4. PyContainerConfig - Spring Python's original XML format ... 25

2.3.5. SpringJavaConfig ... 26

2.3.6. Mixing Configuration Modes ... 27

2.4. Object Factories ... 28

2.5. Testable Code ... 28

2.6. Querying and modifying the ApplicationContext in runtime ... 29

3. Aspect Oriented Programming

3.1. External dependencies ... 33

3.2. Interceptors ... 33

3.3. Proxy Factory Objects ... 35

3.4. Pointcuts ... 36

3.5. Interceptor Chain ... 36

3.6. Coding AOP with Pure Python ... 36

4. Data Access

4.1. DatabaseTemplate ... 39

4.1.1. External dependencies ... 39

4.1.2. Traditional Database Query ... 39

4.1.3. DatabaseTemplate ... 40

4.1.4. Mapping rows into objects using convention over configuration ... 40

4.1.5. Mapping rows into dictionaries ... 41

4.1.6. What is a Connection Factory? ... 41

4.1.7. Creating/altering tables, databases, and other DDL ... 41

4.1.8. SQL Injection Attacks ... 41

4.1.9. Have you used Spring Framework's JdbcTemplate? ... 42

4.1.10. Notes on using SQLServerConnectionFactory ... 42

5. Transaction Management

5.1. External dependencies ... 44

5.2. Solutions requiring transactions 44
5.3. TransactionTemplate 45
5.4. @transactional 46
5.4.1. @transactional(["PROPAGATION_REQUIRED"])... 47
6. Security ....
6.1. External dependencies 49
6.2. Shared Objects 49
6.3. Authentication 49
6.3.1. AuthenticationProviders 49
6.3.2. AuthenticationManager 53
6.4. Authorization 54
7. Remoting ....
7.1. External dependencies 57
7.2. Remoting with PYRO (Python Remote Objects) 57
7.2.1. Decoupling a simple service, to setup for remoting 57
7.2.2. Exporting a Spring Service Using Inversion Of Control 58
7.2.3. Do I have to use XML? 59
7.2.4. Splitting up the client and the server 60
7.3. Remoting with Hessian 62
7.4. High-Availability/Clustering Solutions 62
8. JMS Messaging ....
8.1. Introduction 64
8.2. Dependencies 65
8.3. Quickstart 65
8.3.1. Sending 65
8.3.2. Receiving 66
8.4. Connection factories 68
8.4.1. springpython.jms.factory.WebSphereMQConnectionFactory 68
8.5. springpython.jms.core.JmsTemplate 71
8.5.1. Sending 73
8.5.2. Receiving 75
8.5.3. Dynamic queues 76
8.5.4. Message converters 77
8.6. springpython.jms.listener.SimpleMessageListenerContainer and background JMS listeners 78
8.7. springpython.jms.core.TextMessage 80
8.8. Exceptions 84
8.9. Logging and troubleshooting 85
9. Spring Python's plugin system ....
9.1. Introduction 87
9.2. Coily - Spring Python's command-line tool 87
9.2.1. Commands 87
9.3. Officially Supported Plugins 88
9.3.1. gen-cherrypy-app 89
9.4. Writing your own plugin 89
9.4.1. Architecture of a plugin 89
9.4.2. Case Study - gen-cherrypy-app plugin 90
10. Samples ....
10.1. PetClinic 93
10.1.1. How to run 93
10.2. Spring Wiki 94
10.3. Spring Bot 94
10.3.1. Why write a bot? 94

10.3.2. IRC Library 95
10.3.3. What I built 95
10.3.4. External Links 101

Preface

Spring Python is an extension of the Java-based Spring Framework and Spring Security Framework, targeted for Python. It is not a straight port, but instead an extension of the same concepts that need solutions applied in Python.

This document provides a reference guide to Spring's features. Since this document is still to be considered very much work-in-progress, if you have any requests or comments, please post them on the user mailing list or on the Spring Python support forums.

What we mean by "Spring Java"

Throughout this documentation, the term *Spring Java* is used on occasion as shorthand for *The Spring Framework*, referring to the original, java-based framework.

Before we go on, a few words of gratitude are due to the SpringSource team for putting together a framework for writing this reference documentation.

Chapter 1. Overview

"Spring Python is an offshoot of the Java-based Spring Framework and Spring Security Framework, targeted for Python. Spring provides many useful features, and I wanted those same features available when working with Python."
--Greg Turnquist, Spring Python project lead

Spring Python intends to take the concepts that were developed, tested, and proven with the Spring Framework, and carry them over to the language of Python. If anyone has developed a solution using multiple technologies including Java, C#/.NET, and Python, they will realize that certain issues exist in all these platforms.

This is not a direct port of existing source code, but rather, a port of proven solutions, while still remaining faithful to the style, idioms, and overall user community of Python.

1.1. Key Features

The following features have been implemented:

- Inversion Of Control - The idea is to decouple two classes at the interface level. This lets you build many reusable parts in your software, and your whole application becomes more pluggable. You can use either the `PyContainerConfig` or the `PythonConfig` to plugin your object definition to an `ApplicationContext`.
- Aspect Oriented Programming - Spring Python provides great ways to wrap advice around objects. It is utilized for remoting. Another use is for debug tracers and performance tracing.
- DatabaseTemplate - Reading from the database requires a monotonous cycle of opening cursors, reading rows, and closing cursors, along with exception handlers. With this template class, all you need is the SQL query and row-handling function. Spring Python does the rest.
- Database Transactions - Wrapping multiple database calls with transactions can make your code hard to read. This module provides multiple ways to define transactions without making things complicated.
- Security - Plugin security interceptors to lock down access to your methods, utilizing both authentication and domain authorization.
- Remoting - It is easy to convert your local application into a distributed one. If you have already built your client and server pieces using the IoC container, then going from local to distributed is just a configuration change.
- JMS Messaging - Connect to Java or Python applications using queueing middleware. Spring Python can act as a standalone client of a JMS provider with no Java EE infrastructure needed on Python side.
- Plug-ins/command-line tool - Use the plugin system designed to help you rapidly develop applications.
- Samples - to help demonstrate various features of Spring Python, some sample applications have been created:
    - PetClinic - Everybody's favorite Spring sample application has been rebuilt from the ground up using various web containers including: CherryPy. Go check it out for an example of how to use this

framework.

- Spring Wiki - Wikis are powerful ways to store and manage content, so we created a simple one as a demo!
- Spring Bot - Use Spring Python to build a tiny bot to manage the IRC channel of your open source project.

1.2. What Spring Python is NOT

Spring Python is NOT another web framework. I think there are plenty that are fine to use, like Django, TurboGears, Zope, CherryPy, Quixote, and more. Spring Python is meant to provide utilities to support any python application, including a web-based one.

So far, the demos have been based on CherryPy, but the idea is that these features should work with any python web framework. The Spring Python team is striving to make things reusable with any python-based web framework. There is always the goal of expanding the samples into other frameworks, whether they are web-based, RIA, or thick-client.

1.3. Support

1.3.1. Forums and Email

- You can read the messages on Spring Python's forums at the official Spring forum site.
- If you are interested, you can sign up for the springpython-developer mailing list.
- You can read the current archives of the spring-users mailing list.
- You can also read the old archives of the retired spring-developer mailing list.
- If you want to join this project, see How to become a team member

1.3.2. IRC

Sorry, I can't seem to get a long-term running IRC bot working for me. You'll have to resort to email to reach me for questions or issues. -- Greg

1.4. Downloads / Source Code

If you want a release, check out Spring's download site for Spring Python.

If you want the latest source code type:

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

That will create a new *springpython* folder. This includes both the source code and the demo applications

(PetClinic and SpringWiki).

You can browse the code at https://fisheye.springframework.org/browse/se-springpython-py.

1.5. Licensing

Spring Python is released under the Apache Server License 2.0 and the copyright is held by SpringSource.

1.6. Spring Python's team

Spring Python's official team (those with committer rights):

- Project Lead: Greg L. Turnquist
- SpringSource Sponsor: Mark Pollack
- Project Contributor: Russ Miles
- Project Contributor: Dariusz Suchojad

Many others have also contributed through reporting issues, raising questions, and even sending patches.

1.6.1. How to become a team member

We like hearing about new people interesting in joining the project. We are also excited in hearing from people interested in working on a particular jira feature.

The way we do things around here, we like to work through a few patches before granting you any committer rights. You can checkout a copy of the code anonymously, and then work on your patch. Email your patch to one of the official team members, and we will inspect things. From there we will consider committing your patch, or send you feedback.

If we decide to commit your changes, we may choose to create a new branch for your feature, based on the scope of the work, or simply commit it to the trunk. After testing, evaluation, and prioritization, we may eventually merge your patch to the trunk. After a few patches, if things are looking good, we will evaluate giving you committer rights.

Spring Python is a TDD-based project, meaning if you are working on code, be sure to write an automated test case and write the test case FIRST. For insight into that, take a trip into the code repository's test section to see how current things are run. Your patch can get sold off and committed much faster if you include automated test cases and a pasted sample of your test case running successfully along with the rest of the baseline test suite.

You don't have to become a team member to contribute to this project, but if you want to contribute code, then we ask that you follow the details of this process, because this project is focused on high quality code, and we want to hold everyone to the same standard.

Getting Started

1. First of all, I suggest you sign up on our springpython-developer mailing list. That way, you'll get notified about big items as well be on the inside for important developments that may or may not get published to

the web site. *NOTE: Use the springsource list, NOT the sourceforge one.*

2. Second, I suggest you register for a jira account, so you can leave comments, etc. on the ticket. I think that works (I don't manage jira, so if it doesn't let me know, and we will work from there) NOTE: I like notes and comments tracking what you have done, or what you think needs to be done. It gives us input in case someone else eventually has to complete the ticket. That would also be the place where you can append new files or patches to existing code.

3. Third, register at the SpringSource community forum, and if you want to kick ideas around or float a concept, feel free to start a thread in our Spring Python forum.

4. Finally, we really like to have supporting documentation as well as code. That helps other people who aren't as up-to-speed on your piece of the system. Go ahead and start your patch, but don't forget to look into the docs folder and update or add to relevant documentation. Our documentation is part of the source code, so you can submit doc mods as patches also. Include information such as dependencies, design notes, and whatever else you think would be valuable.

With all that said, happy coding!

1.7. Deprecated Code

To keep things up-to-date, we need to deprecate code from time to time. Python has built in functionality to put warnings into certain sections of code, so that if you import a deprecated module, you will be properly warned. With each major release (1.0, 2.0, 3.0, etc.), the Spring Python team has the option to remove any and all deprecated code.

Chapter 2. The IoC container

Background

In early 2004, Martin Fowler asked the readers of his site: when talking about Inversion of Control: “*the question is, what aspect of control are [they] inverting?*”. Fowler then suggested renaming the principle (or at least giving it a more self-explanatory name), and started to use the term *Dependency Injection*. His article then continued to explain the ideas underpinning the Inversion of Control (IoC) and Dependency Injection (DI) principle.

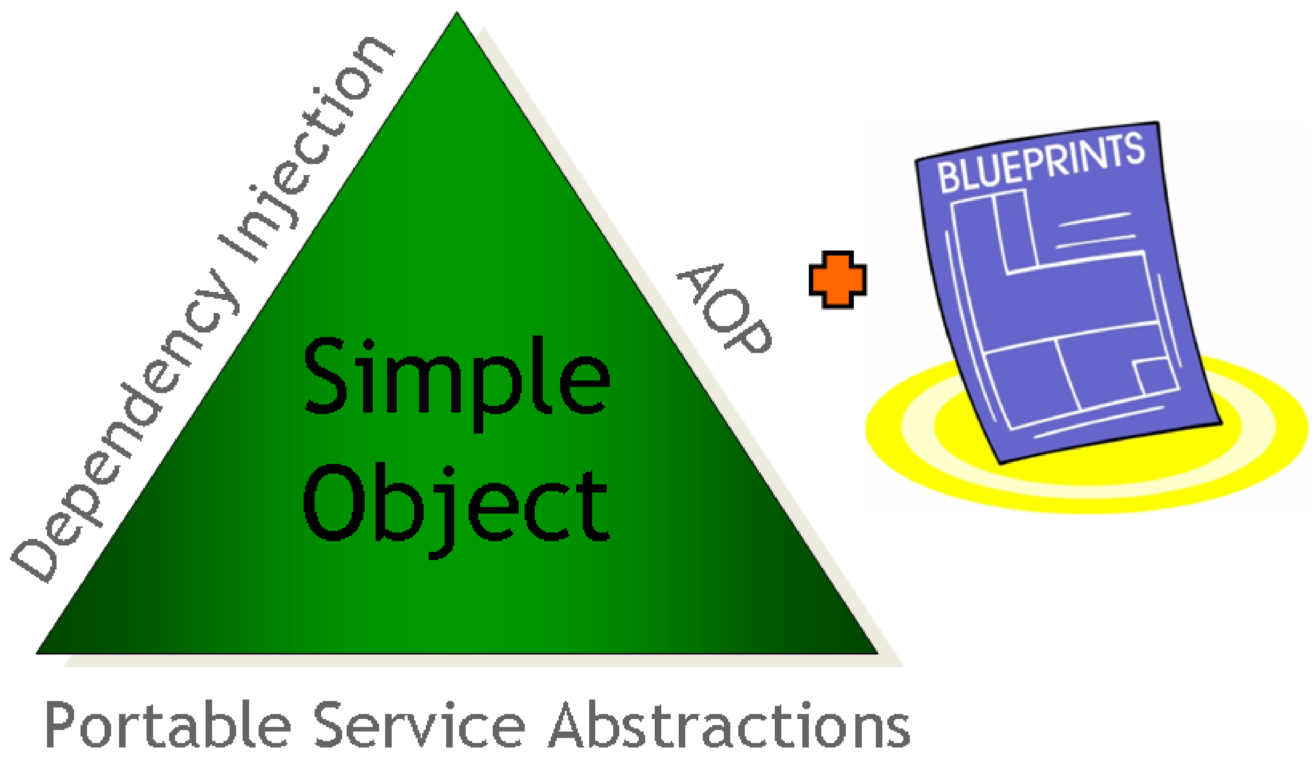
If you need a decent insight into IoC and DI, please do refer to said article : http://martinfowler.com/articles/injection.html.

Inversion Of Control (IoC), also known as dependency injection is more of an architectural concept than a simple coding pattern.

The idea is to decouple classes that depend on each other from inheriting other dependencies, and instead link them only at the interfacing level. This requires some sort of 3rd party software module to instantiate the concrete objects and "inject" them into the class that needs to call them.

In Spring, there are certain classes whose instances form the backbone of your application and that are managed by the Spring IoC container. While Spring Java calls them beans, Spring Python and Spring for .NET call them *objects*. An object is simply a class instance that was instantiated, assembled and otherwise managed by a Spring IoC container instead of directly by your code; other than that, there is nothing special about a object. It is in all other respects one of probably many objects in your application. These objects, and the dependencies between them, are reflected in the configuration meta-data used by a container.

The following diagram demonstrates a key Spring concept: building useful services on top of simple objects, configured through a container's set of blueprints, provides powerful services that are easier to maintain.



This chapter provides the basics of Spring Python's IoC container by using examples with explanations. If you are familiar with Spring Java, then you may notice many similarities. Also, this document points out key differences. It shows how to define the objects, read them into a container, and then fetch the objects into your code.

2.1. External dependencies

XML-based IoC configuration formats use ElementTree which is a part of Python's stantard library in Python 2.5 and newer. If you use Python 2.4 you can download ElementTree from here. YamlConfig requires installation of PyYAML which may be found here. No additional dependencies needs be installed if you choose PythonConfig.

2.2. Container

A *container* is an object you create in your code that receives the definitions for objects from various sources. Your code goes to the container to request the object, and the container then does everything it needs to create an instance of that.

Depending on the scope of the object definition, the container may create a new instance right there on the spot, or it may fetch a reference to a singleton instance that was created previously. If this is the first time a singleton-scoped object is requested, is created, stored, and then returned to you. For a prototype-scoped object, EVERY TIME you request an object, a new instance is created and NOT stored in the singleton cache.

Containers depend on various *object factories* to do the heavy lifting of construction, and then itself will set any additional properties. There is also the possibility of additional behavior outside of object creation, which can be defined by extending the `ObjectContainer` class.

The reason it is called a container is the idea that you are going to a central place to get your top level object. While it is also possible to get all your other objects, the core concept of dependency injection is that below your top-most object, all the other dependencies have been injected and thus not require container access. That is what we mean when we say most of your code does NOT have to be Spring Python-aware.

Present vs. Future Object Containers

Pay special note that there is no fixed requirement that a container actually be in a certain location. While the current solution is memory based, meaning your objects will be lost when your application shuts down, there is always the possibility of implementing some type of distributed, persistent object container. For example, it is within the realm of possibilities to implement a container that utilizes a back-end database to "contain" things or utilizes some distributed memory cache spread between nodes.

2.2.1. `ObjectContainer` vs. `ApplicationContext`

The name of the container is `ObjectContainer`. Its job is to pull in object meta-data from various sources, and then call on related object factories to create the objects. In fact, this container is capable of receiving object definitions from multiple sources, each of differing types such as XML, YAML, python code, and other future formats.

The following block of code shows an example of creating an object container, and then pulling an object out of the container.

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

container = ApplicationContext(XMLConfig("app-context.xml"))
service = container.get_object("sampleService")
```

The first thing you may notice is the fact that `ApplicationContext` was used instead of `ObjectContainer`. `ApplicationContext` is a subclass of `ObjectContainer`, and is typically used because it also performs additional pre- and post-creational logic.

For example, any object that implements `ApplicationContextAware` will have an additional `app_context` attribute added, populated with a copy of the `ApplicationContext`. If your object's class extends `ObjectPostProcessor` and defines a `post_process_after_initialization`, the `ApplicationContext` will run that method against every instance of that object.

If your singleton objects hold references to some external resources, e.g. connections to a resource manager of some sort, you may also want to subclass `springpython.context.DisposableObject` to have a means for the resources to get released. Any singleton subclassing `springpython.context.DisposableObject` may define a `destroy` method which is guaranteed to be executed on `ApplicationContext` shutdown. An alternative to creating a `destroy` method is to define the `destroy_method` attribute of an object which should be a name of the custom method to be invoked on `ApplicationContext` shutdown. If an object defines both `destroy` and `destroy_method` then the former will take precedence. It is an error to extend `springpython.context.DisposableObject` without providing either `destory` or `destroy_method`. If this occurs, an error will be written to Spring Python logs when the container shuts down.

2.2.2. Scope of Objects / Lazy Initialization

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, how long before they are destroyed, and whether or not to create them when the container is first started up.

Currently, two scopes are supported: SINGLETON and PROTOTYPE. A singleton-scoped object is cached in the container until application 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.

The default policy for the container is to make everything SINGLETON and also eagerly fetch all objects when the container is first created. The scope for each object can be individually overriden. Also, the initialization of each object can be shifted to "lazy", whereby the object is not created until the first time it is fetched or referenced by another object.

2.3. Configuration

Spring Python support different formats for defining objects. This project first began using the format defined by PyContainer, a now inactive project. The structure has been captured into an XSD spec. This format is primarily to support legacy apps that have already been built with Spring Python from its inception. There is no current priority to extend this format any further. Any new schema developments will be happening with `XMLConfig` and `YamlConfig`.

In the spirit of Spring JavaConfig and a blog posting by Rod Johnson, another format has been defined. By extending `PythonConfig` and using the `@Object` python decorator, objects may be defined with pure python code in a centralized class.

Due to limitations in the format of PyContainer, another schema has been developed called `XMLConfig` that more closely models the original Spring Java version. It has support for referenced objects in many more places than PyContainer could handle, inner objects as well, various collections (lists, sets, frozen sets, tuples, dictionaries, and java-style props), and values.

Spring Python also has a YAML-based parser called `YamlConfig`.

Spring Python is ultimately about choice, which is why developers may extend the `Config` class to define their own object definition scanner. By plugging an instance of their scanner into `ApplicationContext`, definitions can result in instantiated objects.

You may be wondering, amidst all these choices, which one to pick? Here are some suggestions based on your current solution space:

- New projects are encouraged to pick either `PythonConfig`, `XMLConfig`, or `YamlConfig`, based on your preference for pure python code, XML, or YAML.
- Projects migrating from Spring Java can use `SpringJavaConfig` to ease transition, with a long term goal of migrating to `XMLConfig`, and perhaps finally `PythonConfig`.
- Apps already developed with Spring Python can use `PyContainerConfig` to keep running, but it is highly suggested you work towards `XMLConfig`.
- Projects currently using `XMLConfig` should be pretty easy to migrate to `PythonConfig`, since it is basically a one-to-one translation. The pure python configuration may turn out much more compact, especially if you are using lists, sets, dictionaries, and props.

  It should also be relatively easy to migrate an `XMLConfig`-based configuration to `YamlConfig`. YAML tends to be more compact than XML, and some prefer not having to deal with the angle-bracket tax.

2.3.1. `XMLConfig` - Spring Python's native XML format

`XMLConfig` is a class that scans object definitions stored in the new XML format defined for Spring Python. It looks very similar to Spring Java's 2.5 XSD spec, with some small changes.

The following is a simple definition of objects. Later sections will show other options you have for wiring things together.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

        <object id="MovieLister" class="springpythontest.support.testSupportClasses.MovieLister" scope="prototyp
                <property name="finder" ref="MovieFinder"/>
                <property name="description"><ref object="SingletonString"/></property>
        </object>

        <object id="MovieFinder" class="springpythontest.support.testSupportClasses.ColonMovieFinder" scope="sin
                <property name="filename"><value>support/movies1.txt</value></property>
        </object>

        <object id="SingletonString" class="springpythontest.support.testSupportClasses.StringHolder" lazy-init=
                <property name="str" value="There should only be one copy of this string"></property>
        </object>
</objects>
```

The definitions stored in this file are fed to an `XMLConfig` instance which scans it, and then sends the meta-data to the `ApplicationContext`. Then, when the application code requests an object named *MovieLister* from the container, the container utilizes an object factory to create the object and return it.

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

container = ApplicationContext(XMLConfig("app-context.xml"))
service = container.get_object("MovieLister")
```

2.3.1.1. Referenced Objects

A referenced object is where an object is needed, but instead of providing the definition right there, there is, instead, a name, referring to another object definition.

Object definitions can refer to other objects in many places including: properties, constructor arguments, and objects embedded inside various collections. This is the way to break things down into smaller pieces. It also allows you more efficiently use memory and guarantee different objects are linked to the same backend object.

The following fragment, pulled from the earlier example, shows two different properties referencing other objects. It demonstrates the two ways to refer to another object.

```
<object id="MovieLister" class="springpythontest.support.testSupportClasses.MovieLister" scope="prototype">
        <property name="finder" ref="MovieFinder"/>
        <property name="description"><ref object="SingletonString"/></property>
</object>
```

This means that instead of defining the object meant to be injected into the `description` property right there, the container must look elsewhere amongst its collection of object definitions for an object named *SingletonString*.

Referenced objects don't have to be in same configuration

When a referenced object is encountered, finding its definition is referred back to the container. This means ANY of the input sources provided to the container can hold this definition, REGARDLESS of format.

Spring Python ONLY supports global references

While Spring Java has different levels of reference like *parent*, *local*, and *global*, Spring Python only supports *global* at this time.

In the following subsections, other types of object definitions are given. Each will also include information about embedding reference objects.

2.3.1.2. Inner Objects

Inner objects are objects defined inside another structure, and not at the root level of the XML document. The following shows an alternative configuration of a `MovieLister` where the `finder` uses a *named inner object*.

```
<object id="MovieLister3" class="springpythontest.support.testSupportClasses.MovieLister">
        <property name="finder">
                <object id="named" class="springpythontest.support.testSupportClasses.ColonMovieFinder">
                        <property name="filename"><value>support/movies1.txt</value></property>
                </object>
        </property>
        <property name="description"><ref object="SingletonString"/></property>
</object>
```

The `ColonMovieFinder` is indeed an inner object because it was defined inside the *MovieLister3* object. Objects defined at the top level have a container-level name that matches their `id` value. In this case, asking the container for a copy of *MovieLister3* will yield the top level object. However, named objects develop a path-like name based on where they are located. In this case, the inner `ColonMovieFinder` object will have a container-level name of *MovieLister3.finder.named*.

Typically, neither your code nor other object definitions will have any need to reference *MovieLister3.finder.named*, but there may be cases where you need this. The `id` attribute of `ColonMovieFinder` can be left out (it is optional for inner objects) like this:

```
<object id="MovieLister2" class="springpythontest.support.testSupportClasses.MovieLister">
        <property name="finder">
                <object class="springpythontest.support.testSupportClasses.ColonMovieFinder">
                        <property name="filename"><value>support/movies1.txt</value></property>
                </object>
        </property>
        <property name="description"><ref object="SingletonString"/></property>
</object>
```

That is slightly more compact, and usually alright because you usually wouldn't access this object from anywhere. However, if you must, the name in this case is *MovieLister2.finder.<anonymous>* indicating an anonymous object.

It is important to realize that inner objects have all the same privileges as top-level objects, meaning that they can also utilize reference objects, collections, and inner objects themselves.

2.3.1.3. Collections

Spring Java supports many types of collections, including lists, sets, frozen sets, maps, tuples, and java-style properties. Spring Python supports these as well. The following configuration shows usage of `dict`, `list`, `props`, `set`, `frozenset`, and `tuple`.

```
<object id="ValueHolder" class="springpythontest.support.testSupportClasses.ValueHolder">
        <constructor-arg><ref object="SingletonString"/></constructor-arg>
        <property name="some_dict">
                <dict>
                        <entry><key><value>Hello</value></key><value>World</value></entry>
                        <entry><key><value>Spring</value></key><value>Python</value></entry>
                        <entry><key><value>holder</value></key><ref object="SingletonString"/></entry>
                        <entry><key><value>another copy</value></key><ref object="SingletonString"/></entry>
                </dict>
        </property>
        <property name="some_list">
                <list>
                        <value>Hello, world!</value>
                        <ref object="SingletonString"/>
                        <value>Spring Python</value>
                </list>
        </property>
        <property name="some_props">
                <props>
                        <prop key="administrator">administrator@example.org</prop>
                        <prop key="support">support@example.org</prop>
                        <prop key="development">development@example.org</prop>
                </props>
        </property>
        <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>
</object>
```

- `some_dict` is a python dictionary with four entries.
- `some_list` is a python list with three entries.
- `some_props` is also a python dictionary, containing three values.
- `some_set` is an instance of python's mutable set.
- `some_frozen_set` is an instance of python's frozen set.
- `some_tuple` is a python tuple with three values.

Java uses maps, Python uses dictionaries

While java calls key-based structures *maps*, python calls them *dictionaries*. For this reason, the code fragment shows a "dict" entry, which is one-to-one with Spring Java's "map" definition.

Java also has a `Property` class. Spring Python translates this into a python dictionary, making it more like an alternative to the configuring mechanism of `dict`.

2.3.1.4. Constructors

Python functions can have both positional and named arguments. Positional arguments get assembled into a tuple, and named arguments are assembled into a dictionary, before being passed to a function call. Spring Python takes advantage of that option when it comes to constructor calls. The following block of configuration data shows defining positional constructors.

```
<object id="AnotherSingletonString" class="springpythontest.support.testSupportClasses.StringHolder">
        <constructor-arg value="attributed value"/>
</object>

<object id="AThirdSingletonString" class="springpythontest.support.testSupportClasses.StringHolder">
        <constructor-arg><value>elemental value</value></constructor-arg>
</object>
```

Spring Python will read these and then feed them to the class constructor in the same order as shown here.

The following code configuration shows named constructor arguments. Spring Python converts these into keyword arguments, meaning it doesn't matter what order they are defined.

```
<object id="MultiValueHolder" class="springpythontest.support.testSupportClasses.MultiValueHolder">
        <constructor-arg name="a"><value>alt a</value></constructor-arg>
        <constructor-arg name="b"><value>alt b</value></constructor-arg>
</object>

<object id="MultiValueHolder2" class="springpythontest.support.testSupportClasses.MultiValueHolder">
        <constructor-arg name="c"><value>alt c</value></constructor-arg>
        <constructor-arg name="b"><value>alt b</value></constructor-arg>
</object>
```

This was copied from the code's test suite, where a test case is used to prove that order doesn't matter. It is important to note that positional constructor arguments are fed before named constructors, and that overriding a the same constructor parameter both by position and by name is not allowed by Python, and will in turn, generate a run-time error.

It is also valuable to know that you can mix this up and use both.

2.3.1.5. Values

For those of you that used Spring Python before `XMLConfig`, you may have noticed that expressing values isn't as succinct as the old format. A good example of the old PyContainer format would be:

```
<component id="user_details_service" class="springpython.security.userdetails.InMemoryUserDetailsService">
        <property name="user_dict">
                {
                        "basichiblueuser"  : ("password1", ["ROLE_BASIC", "ASSIGNED_BLUE",   "LEVEL_HI"], True),
                        "basichiorangeuser": ("password2", ["ROLE_BASIC", "ASSIGNED_ORANGE", "LEVEL_HI"], True),
                        "otherhiblueuser"  : ("password3", ["ROLE_OTHER", "ASSIGNED_BLUE",   "LEVEL_HI"], True),
                        "otherhiorangeuser": ("password4", ["ROLE_OTHER", "ASSIGNED_ORANGE", "LEVEL_HI"], True),
                        "basicloblueuser"  : ("password5", ["ROLE_BASIC", "ASSIGNED_BLUE",   "LEVEL_LO"], True),
                        "basicloorangeuser": ("password6", ["ROLE_BASIC", "ASSIGNED_ORANGE", "LEVEL_LO"], True),
                        "otherloblueuser"  : ("password7", ["ROLE_OTHER", "ASSIGNED_BLUE",   "LEVEL_LO"], True),
                        "otherloorangeuser": ("password8", ["ROLE_OTHER", "ASSIGNED_ORANGE", "LEVEL_LO"], True)
                }
        </property>
</component>
```

Why do I see components and not objects?

In the beginning, PyContainer was used and it tagged the managed instances as *components*. After replacing PyContainer with a more sophisticated IoC container, the instances are now referred to as *objects*, however, to maintain this legacy format, you will see *component* tags inside `PyContainerConfig`-based definitions.

While this is very succinct for expressing definitions using as much python as possible, that format makes it very hard to embed referenced objects and inner objects, since `PyContainerConfig` uses python's `eval` method to convert the material.

The following configuration block shows how to configure the same thing for `XMLConfig`.

```
<object id="user_details_service" class="springpython.security.userdetails.InMemoryUserDetailsService">
        <property name="user_dict">
                <dict>
                        <entry>
                                <key><value>basichiblueuser</value></key>
                                <value><tuple>
                                        <value>password1</value>
                                        <list><value>ROLE_BASIC</value><value>ASSIGNED_BLUE</value><value>LEVEL_
                                        <value>True</value>
                                </tuple></value>
                        </entry>
                        <entry>
                                <key><value>basichiorangeuser</value></key>
                                <value><tuple>
                                        <value>password2</value>
                                        <list><value>ROLE_BASIC</value><value>ASSIGNED_ORANGE</value><value>LEVE
                                        <value>True</value>
                                </tuple></value>
                        </entry>
                        <entry>
                                <key><value>otherhiblueuser</value></key>
                                <value><tuple>
                                        <value>password3</value>
                                        <list><value>ROLE_OTHER</value><value>ASSIGNED_BLUE</value><value>LEVEL_
                                        <value>True</value>
                                </tuple></value>
                        </entry>
                        <entry>
                                <key><value>otherhiorangeuser</value></key>
                                <value><tuple>
                                        <value>password4</value>
                                        <list><value>ROLE_OTHER</value><value>ASSIGNED_ORANGE</value><value>LEVE
                                        <value>True</value>
                                </tuple></value>
                        </entry>
                        <entry>
                                <key><value>basicloblueuser</value></key>
                                <value><tuple>
                                        <value>password5</value>
                                        <list><value>ROLE_BASIC</value><value>ASSIGNED_BLUE</value><value>LEVEL_
                                        <value>True</value>
                                </tuple></value>
                        </entry>
                        <entry>
                                <key><value>basicloorangeuser</value></key>
                                <value><tuple>
                                        <value>password6</value>
                                        <list><value>ROLE_BASIC</value><value>ASSIGNED_ORANGE</value><value>LEVE
                                        <value>True</value>
                                </tuple></value>
                        </entry>
                        <entry>
                                <key><value>otherloblueuser</value></key>
                                <value><tuple>
                                        <value>password7</value>
                                        <list><value>ROLE_OTHER</value><value>ASSIGNED_BLUE</value><value>LEVEL_
                                        <value>True</value>
                                </tuple></value>
                        </entry>
                        <entry>
```

```
                                <key><value>otherloorangeuser</value></key>
                                <value><tuple>
                                        <value>password8</value>
                                        <list><value>ROLE_OTHER</value><value>ASSIGNED_ORANGE</value><value>LEVE
                                        <value>True</value>
                                </tuple></value>
                        </entry>
                </dict>
        </property>
</object>
```

Of course this is more verbose than the previous block. However, it opens the door to having a much higher level of detail:

```
<object id="user_details_service2" class="springpython.security.userdetails.InMemoryUserDetailsService">
        <property name="user_dict">
                <list>
                        <value>Hello, world!</value>
                        <dict>
                                <entry>
                                        <key><value>yes</value></key>
                                        <value>This is working</value>
                                </entry>
                                <entry>
                                        <key><value>no</value></key>
                                        <value>Maybe it's not?</value>
                                </entry>
                        </dict>
                        <tuple>
                                <value>Hello, from Spring Python!</value>
                                <value>Spring Python</value>
                                <dict>
                                        <entry>
                                                <key><value>yes</value></key>
                                                <value>This is working</value>
                                        </entry>
                                        <entry>
                                                <key><value>no</value></key>
                                                <value>Maybe it's not?</value>
                                        </entry>
                                </dict>
                                <list>
                                        <value>This is a list element inside a tuple.</value>
                                        <value>And so is this :)</value>
                                </list>
                        </tuple>
                        <set>
                                <value>1</value>
                                <value>2</value>
                                <value>1</value>
                        </set>
                        <frozenset>
                                <value>a</value>
                                <value>b</value>
                                <value>a</value>
                        </frozenset>
                </list>
        </property>
</object>
```

2.3.1.6. XMLConfig and basic Python types

Objects of most commonnly used Python types - `str`, `unicode`, `int`, `long`, `float`, `decimal.Decimal`, `bool` and `complex` - may be expressed in XMLConfig using a shorthand syntax which allows for a following XMLConfig file:

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

    <str id="MyString">My string</str>

    <unicode id="MyUnicode">Za#ó## g##l# ja##</unicode>

    <int id="MyInt">10</int>

    <long id="MyLong">100000000000000000000000</long>

    <float id="MyFloat">3.14</float>

    <decimal id="MyDecimal">12.34</decimal>

    <bool id="MyBool">False</bool>

    <complex id="MyComplex">10+0j</complex>

</objects>
```

2.3.1.7. Object definition inheritance

XMLConfig's definitions may be stacked up into hierarchies of abstract parents and their children objects. A child object not only inherits all the properties and constructor arguments from its parent but it can also easily override any of the inherited values. This can save a lot of typing when configuring non-trivial application contexts which would otherwise need to repeat the same configuration properties over many objects definitions.

An abstract object is identified by having an *abstract="True"* attribute and the child ones are those which have a *parent* attribute set to ID of an object from which the properties or constructor arguments should be inherited. Child objects must not specify the *class* attribute, its value is taken from their parents.

An object may be both a child and an abstract one.

Here's a hypothetical configuration of a set of services exposed by a server. Note how you can easily change the CRM environment you're invoking by merely changing the concrete service's (get_customer_id or get_customer_profile) parent ID.

```

<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

    <object id="service" class="springpythontest.support.testSupportClasses.Service" scope="singleton" abstract=
        <property name="ip"><value>192.168.1.153</value></property>
    </object>

    <object id="crm_service_dev" parent="service" abstract="True">
        <property name="port"><value>3392</value></property>
    </object>

    <object id="crm_service_test" parent="service" abstract="True">
        <property name="port"><value>3393</value></property>
    </object>

    <object id="get_customer_id" parent="crm_service_dev">
        <property name="path"><value>/soap/invoke/get_customer_id</value></property>
    </object>

    <object id="get_customer_profile" parent="crm_service_test">
        <property name="path"><value>/soap/invoke/get_customer_profile</value></property>
    </object>

</objects>
```

Here's how you can override inherited properties; both get_customer_id and get_customer_profile object definitions will inherit the *path* property however the actual objects returned by the container will use local, overridden, values of the property.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

    <object id="service" class="springpythontest.support.testSupportClasses.Service" scope="singleton" abstract=
        <property name="ip"><value>192.168.1.153</value></property>
        <property name="port"><value>3392</value></property>
        <property name="path"><value>/DOES-NOT-EXIST</value></property>
    </object>

    <object id="get_customer_id" parent="service">
        <property name="path"><value>/soap/invoke/get_customer_id</value></property>
    </object>

    <object id="get_customer_profile" parent="service">
        <property name="path"><value>/soap/invoke/get_customer_profile</value></property>
    </object>

</objects>
```

If you need to get an abstract object from a container, use the .get_object's *ignore_abstract* parameter, otherwise *springpython.container.AbstractObjectException* will be raised. Observe the difference:

```
# .. skip creating the context

# No exception will be raised, even though 'service' is an abstract object
service = ctx.get_object("service", ignore_abstract=True)

# Will show the object
print service

# Will raise AbstractObjectException
service = ctx.get_object("service")
```

2.3.2. `YamlConfig` - Spring Python's YAML format

`YamlConfig` is a class that scans object definitions stored in a YAML 1.1 format using the PyYAML project.

The following is a simple definition of objects, including scope and lazy-init. Later sections will show other options you have for wiring things together.

```
objects:
    - object: MovieLister
      class: springpythontest.support.testSupportClasses.MovieLister
      scope: prototype
      properties:
          finder: {ref: MovieFinder}
          description: {ref: SingletonString}

    - object: MovieFinder
      class: springpythontest.support.testSupportClasses.ColonMovieFinder
      scope: singleton
      lazy-init: True
      properties:
          filename: support/movies1.txt

    - object: SingletonString
      class: springpythontest.support.testSupportClasses.StringHolder
      lazy-init: True
      properties:
          str: There should only be one copy of this string
```

The definitions stored in this file are fed to an `YamlConfig` instance which scans it, and then sends the meta-data to the `ApplicationContext`. Then, when the application code requests an object named *MovieLister* from the container, the container utilizes an object factory to create the object and return it.

```
from springpython.context import ApplicationContext
from springpython.config import YamlConfig

container = ApplicationContext(YamlConfig("app-context.xml"))
service = container.get_object("MovieLister")
```

2.3.2.1. Referenced Objects

A referenced object is where an object is needed, but instead of providing the definition right there, there is, instead, a name, referring to another object definition.

Object definitions can refer to other objects in many places including: properties, constructor arguments, and objects embedded inside various collections. This is the way to break things down into smaller pieces. It also allows you more efficiently use memory and guarantee different objects are linked to the same backend object.

The following fragment, pulled from the earlier example, shows two different properties referencing other objects. It demonstrates the two ways to refer to another object.

```
object: MovieLister
class: springpythontest.support.testSupportClasses.MovieLister
scope: prototype
properties:
    finder: {ref: MovieFinder}
    description: {ref: SingletonString}
```

This means that instead of defining the object meant to be injected into the `description` property right there, the container must look elsewhere amongst its collection of object definitions for an object named *SingletonString*.

Referenced objects don't have to be in same configuration

When a referenced object is encountered, finding its definition is referred back to the container. This means ANY of the input sources provided to the container can hold this definition, REGARDLESS of format.

Spring Python ONLY supports global references

While Spring Java has different levels of reference like *parent*, *local*, and *global*, Spring Python only supports *global* at this time.

In the following subsections, other types of object definitions are given. Each will also include information about embedding reference objects.

2.3.2.2. Inner Objects

Inner objects are objects defined inside another structure, and not at the root level of the YAML document. The following shows an alternative configuration of a `MovieLister` where the `finder` uses a *named inner object*.

```
object: MovieLister3
class: springpythontest.support.testSupportClasses.MovieLister
properties:
    finder:
        object: named
        class: springpythontest.support.testSupportClasses.ColonMovieFinder
        properties:
            filename: support/movies1.txt
    description: {ref: SingletonString}
```

The `ColonMovieFinder` is indeed an inner object because it was defined inside the *MovieLister3* object. Objects defined at the top level have a container-level name that matches their `id` value. In this case, asking the container for a copy of *MovieLister3* will yield the top level object. However, named objects develop a path-like name based on where they are located. In this case, the inner `ColonMovieFinder` object will have a container-level name of *MovieLister3.finder.named*.

Typically, neither your code nor other object definitions will have any need to reference *MovieLister3.finder.named*, but there may be cases where you need this. The value of the `object` key of `ColonMovieFinder` can be left out (it is optional for inner objects) like this:

```
object: MovieLister2
class: springpythontest.support.testSupportClasses.MovieLister
properties:
    finder:
        object:
        class: springpythontest.support.testSupportClasses.ColonMovieFinder
        properties:
            filename: support/movies1.txt
    description: {ref: SingletonString}
```

That is slightly more compact, and usually alright because you usually wouldn't access this object from anywhere. However, if you must, the name in this case is *MovieLister2.finder.<anonymous>* indicating an anonymous object.

It is important to realize that inner objects have all the same privileges as top-level objects, meaning that they can also utilize reference objects, collections, and inner objects themselves.

2.3.2.3. Collections

Spring Java supports many types of collections, including lists, sets, frozen sets, maps, tuples, and java-style properties. Spring Python supports these as well. The following configuration shows usage of `dict`, `list`, `set`, `frozenset`, and `tuple`.

```
      object: ValueHolder
      class: springpythontest.support.testSupportClasses.ValueHolder
      constructor-args:
          - {ref: SingletonString}
      properties:
          some_dict:
                Hello: World
                Spring: Python
                holder: {ref: SingletonString}
                another copy: {ref: SingletonString}
          some_list:
              - Hello, world!
              - ref: SingletonString
              - Spring Python
          some_props:
              administrator: administrator@example.org
              support: support@example.org
              development: development@example.org
          some_set:
              set:
                  - Hello, world!
                  - ref: SingletonString
                  - Spring Python
          some_frozen_set:
              frozenset:
                  - Hello, world!
                  - ref: SingletonString
                  - Spring Python
          some_tuple:
              tuple:
                  - Hello, world!
                  - ref: SingletonString
                  - Spring Python
```

- `some_dict` is a python dictionary with four entries.
- `some_list` is a python list with three entries.
- `some_props` is also a python dictionary, containing three values.
- `some_set` is an instance of python's mutable set.
- `some_frozen_set` is an instance of python's frozen set.
- `some_tuple` is a python tuple with three values.

Java uses maps, Python uses dictionaries

While java calls key-based structures *maps*, python calls them *dictionaries*. For this reason, the code fragment shows a "dict" entry, which is one-to-one with Spring Java's "map" definition.

Java also has a `Property` class. Since YAML already supports a key/value structure as-is, `YamlConfig` does not have a separate structural definition.

2.3.2.4. Support for Python builtin types and mappings of other types onto YAML syntax

Objects of commonly used Python builtin types may be tersely expressed in YamlConfig. Supported types are `str`, `unicode`, `int`, `long`, `float`, `decimal.Decimal`, `bool`, `complex`, `dict`, `list` and `tuple`.

Here's a sample YamlConfig featuring their usage. Note that with the exception of `decimal.Decimal`, names of the YAML attributes are the same as the names of Python types.

```
objects:
    - object:  MyString
      str: My string

    - object:  MyUnicode
      unicode: Za#ó## g##l# ja##

    - object:  MyInt
      int: 10

    - object:  MyLong
      long: 100000000000000000000000

    - object:  MyFloat
      float: 3.14

    - object:  MyDecimal
      decimal: 12.34

    - object:  MyBoolean
      bool: False
    - object:  MyComplex
      complex: 10+0j

    - object:  MyList
      list: [1, 2, 3, 4]

    - object:  MyTuple
      tuple: ["a", "b", "c"]

    - object: MyDict
      dict:
        1: "a"
        2: "b"
        3: "c"

    - object: MyRef
      decimal:
        ref: MyDecimal
```

Under the hood, while parsing the YAML files, Spring Python will translate the definitions such as the one above into the following one:

```
objects:
    - object:  MyString
      class: types.StringType
      constructor-args: ["My string"]

    - object:  MyUnicode
      class: types.UnicodeType
      constructor-args: ["Za#ó## g##l# ja##"]

    - object:  MyInt
      class: types.IntType
      constructor-args: [10]

    - object:  MyLong
      class: types.LongType
      constructor-args: [10000000000000000000000]

    - object:  MyFloat
      class: types.FloatType
      constructor-args: [3.14]

    - object:  MyDecimal
      class: decimal.Decimal
      constructor-args: ["12.34"]

    - object: MyBoolean
      class: types.BooleanType
      constructor-args: [False]

    - object: MyComplex
      class: types.ComplexType
      constructor-args: [10+0j]

    - object: MyList
      class: types.ListType
      constructor-args: [[1,2,3,4]]

    - object: MyTuple
      class: types.TupleType
      constructor-args: [["a", "b", "c"]]

    - object: MyDict
      class: types.DictType
      constructor-args: [{1: "a", 2: "b", 3: "c"}]

    - object: MyRef
      class: decimal.Decimal
      constructor-args: [{ref: MyDecimal}]
```

Configuration of how YAML elements are mapped onto Python types is stored in the `springpython.config.yaml_mappings` dictionary which can be easily customized to fulfill one's needs. The dictionary's keys are names of the YAML elements and its values are the coresponding Python types, written as strings in the form of `"package_name.module_name.class_name"` - note that the `"package_name.module_name."` part is required, it needs to be a fully qualified name.

Let's assume that in your configuration you're frequently creating objects of type `interest_rate.InterestRateFrequency`, here's how you can save yourself a lot of typing by customizing the mappings dictionary. First, on Python side, create an `InterestRate` class, such as:

```
class InterestRate(object):
    def __init__(self, value=None):
        self.value = value
```

.. which will allow you to create such a YAML context

```
objects:
    - object: base_interest_rate
      interest_rate: "7.35"
```

.. then, before creating the context, update the mappings dictionary as needed and next you'll be able to access the `base_interest_rate` object as if it had been defined using the standard syntax:

```
from springpython.context import ApplicationContext
from springpython.config import YamlConfig, yaml_mappings

yaml_mappings.update({"interest_rate": "interest_rate.InterestRate"})

# .. create the context now
container = ApplicationContext(YamlConfig("./app-ctx.yaml"))

# .. fetch the object
base_interest_rate = container.get_object("base_interest_rate")

# .. will show "7.35", as defined in the "./app-ctx.yaml" config
print base_interest_rate.value
```

2.3.2.5. Constructors

Python functions can have both positional and named arguments. Positional arguments get assembled into a tuple, and named arguments are assembled into a dictionary, before being passed to a function call. Spring Python takes advantage of that option when it comes to constructor calls. The following block of configuration data shows defining positional constructors.

```
object: AnotherSingletonString
class: springpythontest.support.testSupportClasses.StringHolder
constructor-args:
    - position 1's constructor value
```

Spring Python will read these and then feed them to the class constructor in the same order as shown here.

The following code configuration shows named constructor arguments. Spring Python converts these into keyword arguments, meaning it doesn't matter what order they are defined.

```
object: MultiValueHolder
class: springpythontest.support.testSupportClasses.MultiValueHolder
```

```
constructor-args:
    a: alt a
    b: alt b
```

This was copied from the code's test suite, where a test case is used to prove that order doesn't matter. It is important to note that positional constructor arguments are fed before named constructors, and that overriding a the same constructor parameter both by position and by name is not allowed by Python, and will in turn, generate a run-time error.

It is also valuable to know that you can mix this up and use both.

2.3.2.6. Object definition inheritance

Just like XMLConfig, YamlConfig allows for wiring the objects definitions into hierarchies of abstract and children objects, thus this section is in most parts a repetition of what's documented here

Definitions may be stacked up into hierarchies of abstract parents and their children objects. A child object not only inherits all the properties and constructor arguments from its parent but it can also easily override any of the inherited values. This can save a lot of typing when configuring non-trivial application contexts which would otherwise need to repeat the same configuration properties over many objects definitions.

An abstract object is identified by having an *abstract* attribute equal to True and the child ones are those which have a *parent* attribute set to ID of an object from which the properties or constructor arguments should be inherited. Child objects must not specify the *class* attribute, its value is taken from their parents.

An object may be both a child and an abstract one.

Here's a hypothetical configuration of a set of services exposed by a server. Note how you can easily change the CRM environment you're invoking by merely changing the concrete service's (get_customer_id or get_customer_profile) parent ID.

```

objects:
    - object: service
      class: springpythontest.support.testSupportClasses.Service
      abstract: True
      scope: singleton
      lazy-init: True
      properties:
        ip: 192.168.1.153

    - object: crm_service_dev
      abstract: True
      parent: service
      properties:
        port: "3392"

    - object: crm_service_test
      abstract: True
      parent: service
      properties:
        port: "3393"

    - object: get_customer_id
      parent: crm_service_dev
      properties:
        path: /soap/invoke/get_customer_id

    - object: get_customer_profile
      parent: crm_service_test
      properties:
        path: /soap/invoke/get_customer_profile
```

Here's how you can override inherited properties; both get_customer_id and get_customer_profile object definitions will inherit the *path* property however the actual objects returned by the container will use local, overridden, values of the property.

```
objects:
    - object: service
      class: foo.Service
      abstract: True
      scope: singleton
      lazy-init: True
      properties:
        ip: 192.168.1.153
        port: "3392"
        path: /DOES-NOT-EXIST

    - object: get_customer_id
      parent: service
      properties:
        path: /soap/invoke/get_customer_id

    - object: get_customer_profile
      parent: service
      properties:
        path: /soap/invoke/get_customer_profile
```

If you need to get an abstract object from a container, use the .get_object's *ignore_abstract* parameter, otherwise *springpython.container.AbstractObjectException* will be raised. Observe the difference:

```
# .. skip creating the context

# No exception will be raised, even though 'service' is an abstract object
service = ctx.get_object("service", ignore_abstract=True)

# Will show the object
print service

# Will raise AbstractObjectException
service = ctx.get_object("service")
```

2.3.3. `PythonConfig` and `@Object` - decorator-driven configuration

By defining a class that extends `PythonConfig` and using the `@Object` decorator, you can wire your application using pure python code.

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

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

    @Object(scope.PROTOTYPE)
    def MovieLister(self):
        lister = MovieLister()
        lister.finder = self.MovieFinder()
        lister.description = self.SingletonString()
        self.logger.debug("Description = %s" % lister.description)
        return lister

    @Object(scope.SINGLETON)
    def MovieFinder(self):
        return ColonMovieFinder(filename="support/movies1.txt")

    @Object(lazy_init=True)    # scope.SINGLETON is the default
    def SingletonString(self):
        return StringHolder("There should only be one copy of this string")

    def NotExposed(self):
        pass
```

As part of this example, the method `NotExposed` is also shown. This indicates that using `get_object` won't fetch that method, since it isn't considered an object.

By using pure python, you don't have to deal with any XML. If you look closely, you will notice that the container code below is only different in the line actually creating the container. Everything else is the same.

```

from springpython.context import ApplicationContext

container = ApplicationContext(MovieBasedApplicationContext())
service = container.get_object("MovieLister")
```

PythonConfig's support for abstract objects is different to that of XMLConfig or YamlConfig. With PythonConfig, the children object do not automatically inherit nor override the parents' properties, they in fact *receive* the values returned by their parents and it's up to them to decide, using Python code, whether to use or to discard the values received.

A child object must have as many optional arguments as there are expected to be returned by its parent.

Observe that in the following example the child definitions must define an optional 'req' argument; in runtime they will be passed its value basing on what their parent object will return. Note also that *request* is of PROTOTYPE scope, if it were a SINGLETON then both get_customer_id_request and get_customer_profile_request would receive the very same Request instance which, in other circumstances, could be a desirable effect but not in the example.

```


# stdlib
import uuid4

# .. skip Spring Python imports

class Request(object):
    def __init__(self, nonce=None, user=None, password=None):
        self.nonce = nonce
        self.user = user
        self.password = password

    def __str__(self):
        return "<id=%s %s %s %s>" % (hex(id(self)), self.nonce, self.user, self.password)

class TestAbstractContext(PythonConfig):

    @Object(scope.PROTOTYPE, abstract=True)
    def request(self):
        return Request()

    @Object(parent="request")
    def request_dev(self, req=None):
        req.user = "dev-user"
        req.password = "dev-password"

        return req

    @Object(parent="request")
    def request_test(self, req=None):
        req.user = "test-user"
        req.password = "test-password"

        return req

    @Object(parent="request_dev")
    def get_customer_id_request(self, req=None):
        req.nonce = uuid4().hex

        return req

    @Object(parent="request_test")
    def get_customer_profile_request(self, req=None):
        req.nonce = uuid4().hex

        return req
```

Same as with the other configuration modes, if you need to get an abstract object from a container, use the .get_object's *ignore_abstract* parameter, otherwise *springpython.container.AbstractObjectException* will be raised:

```
# .. skip creating the context

# No exception will be raised, even though 'request' is an abstract object
request = ctx.get_object("request", ignore_abstract=True)

# Will show the object
print request

# Will raise AbstractObjectException
request = ctx.get_object("request")
```

2.3.4. `PyContainerConfig` - Spring Python's original XML format

`PyContainerConfig` is a class that scans object definitions stored in the format defined by PyContainer, which was the original XML format used by Spring Python to define objects.

An important thing to note is that PyContainer used the term *component*, while Spring Python uses *object*. In order to support this legacy format, *component* will show up in `PyContainerConfig`-based configurations.

PyContainer's format is deprecated

PyContainer's format and the original parser was useful for getting this project started. However, it has shown its age by not being easy to revise nor extend. So this format is being retired. This parser is solely provided to help sustain existing Spring Python apps until they can migrate to either the XMLConfig or the YamlConfig format.

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

    <component id="MovieLister" class="springpythontest.support.testSupportClasses.MovieLister" scope="prototype
            <property name="finder" local="MovieFinder"/>
            <property name="description" local="SingletonString"/>
        </component>

        <component id="MovieFinder" class="springpythontest.support.testSupportClasses.ColonMovieFinder" scope="
                <property name="filename">"support/movies1.txt"</property>
        </component>

    <component id="SingletonString" class="springpythontest.support.testSupportClasses.StringHolder">
            <property name="str">"There should only be one copy of this string"</property>
    </component>
</components>
```

The definitions stored in this file are fed in to a `PyContainerConfig` which scans it, and then sends the meta-data to the `ApplicationContext`. Then, when the application code requests an object named "MovieLister" from the container, the container utilizes an object factory to create an object and return it.

```
from springpython.context import ApplicationContext
from springpython.config import PyContainerConfig

container = ApplicationContext(PyContainerConfig("app-context.xml"))
service = container.get_object("MovieLister")
```

2.3.5. `SpringJavaConfig`

The `SpringJavaConfig` is a class that scans object definitions stored in the format defined by the Spring Framework's original java version. This makes it even easier to migrate parts of an existing Spring Java application onto the Python platform.

This is about configuring python objects NOT java objects

It is important to point out that this has nothing to do with configuring java-backed beans from Spring Python, or somehow injecting java-backed beans magically into a python object. This is PURELY for configuring python-backed objects using a format that was originally designed for pure java beans.

When ideas like "converting java to python" are mentioned, it is meant that re-writing certain parts of your app in python would require a similar IoC configuration, however, for the java and python parts to integrate, you must utilize interoperable solutions like web service or other remoting technologies.

```
<?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="MovieLister" class="springpythontest.support.testSupportClasses.MovieLister" scope="prototype"
                <property name="finder" ref="MovieFinder"/>
                <property name="description"><ref bean="SingletonString"/></property>
        </bean>

        <bean id="MovieFinder" class="springpythontest.support.testSupportClasses.ColonMovieFinder" scope="singl
                <property name="filename"><value>support/movies1.txt</value></property>
        </bean>

        <bean id="SingletonString" class="springpythontest.support.testSupportClasses.StringHolder">
                <property name="str" value="There should only be one copy of this string"></property>
        </bean>
</beans>
```

The definitions stored in this file are fed in to a `SpringJavaConfig` which scans it, and then sends the meta-data to the `ApplicationContext`. Then, when the application code requests an object named "MovieLister" from the container, the container utilizes an object factory to create an object and return it.

```
from springpython.context import ApplicationContext
from springpython.config import SpringJavaConfig

container = ApplicationContext(SpringJavaConfig("app-context.xml"))
service = container.get_object("MovieLister")
```

Again, the only difference in your code is using `SpringJavaConfig` instead of `PyContainerConfig` on one line. Everything is the same, since it is all inside the `ApplicationContext`.

What parts of Spring Java configuration are supported?

It is important to note that only spring-beans-2.5 has been tested at this point in time. It is possible that older versions of the XSD spec may also work.

Spring Java's other names spaces, like *tx* and *aop*, probably DON'T work. They haven't been tested, and there is no special code that will utilize their feature set.

How much of Spring Java will be supported? That is an open question, best discussed on Spring Python's community forum. Basically, this is meant to ease current Java developers into Spring Python and/or provide a means to split up objects to support porting parts of your application into Python. There isn't any current intention of providing full blown support.

2.3.6. Mixing Configuration Modes

Spring Python also supports providing object definitions from multiple sources, and allowing them to reference each other. This section shows the same app context, but split between two different sources.

First, the XML file containing the key object that gets pulled:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://www.springframework.org/springpython/schema/pycontainer-components"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/pycontainer-components
                       http://springpython.webfactional.com/schema/context/spring-python-pycontainer-context-1.

    <component id="MovieLister" class="springpythontest.support.testSupportClasses.MovieLister" scope="prototype
            <property name="finder" local="MovieFinder"/>
            <property name="description" local="SingletonString"/>
        </component>

    <component id="SingletonString" class="springpythontest.support.testSupportClasses.StringHolder">
            <property name="str">"There should only be one copy of this string"</property>
    </component>
</components>
```

Notice that *MovieLister* is referencing *MovieFinder*, however that object is NOT defined in this location. The definition is found elsewhere:

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

    @Object(scope.SINGLETON)
    def MovieFinder(self):
        return ColonMovieFinder(filename="support/movies1.txt")
```

Object ref must match function name

In this situation, an XML-based object is referencing python code by the name *MovieFinder*. It is of paramount importance that the python function have the same name as the referenced string.

With some simple code, this is all brought together when the container is created.

```
from springpython.context import ApplicationContext
from springpython.config import PyContainerConfig

container = ApplicationContext([MixedApplicationContext(),
                                PyContainerConfig("mixed-app-context.xml")])
movieLister = container.get_object("MovieLister")
```

In this case, the XML-based object definition signals the container to look elsewhere for a copy of the MovieFinder object, and it succeeds by finding it in MixedApplicationContext.

It is possible to switch things around, but it requires a slight change.

```
class MixedApplicationContext2(PythonConfig):
    def __init__(self):
        super(MixedApplicationContext2, self).__init__()

    @Object(scope.PROTOTYPE)
    def MovieLister(self):
        lister = MovieLister()
        lister.finder = self.app_context.get_object("MovieFinder")  # <-- only line that is different
        lister.description = self.SingletonString()
        self.logger.debug("Description = %s" % lister.description)
        return lister

    @Object    # scope.SINGLETON is the default
    def SingletonString(self):
        return StringHolder("There should only be one copy of this string")
```

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://www.springframework.org/springpython/schema/pycontainer-components"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/pycontainer-components
                       http://springpython.webfactional.com/schema/context/spring-python-pycontainer-context-1.

        <component id="MovieFinder" class="springpythontest.support.testSupportClasses.ColonMovieFinder" scope="
                <property name="filename">"support/movies1.txt"</property>
        </component>

</components>
```

An XML-based object definition can refer to a `@Object` by name, however, the python code has to change its direct function call to a container lookup, otherwise it will fail.

PythonConfig is ApplicationContextAware

In order to perform a `get_object`, the configuration needs a handle on the surrounding container. The base class `PythonConfig` provides this, so that you can easily look for any object (local or not) by using `self.app_context.get_object("name")`

2.4. Object Factories

Spring Python offers two types of factories, `ReflectiveObjectFactory` and `PythonObjectFactory`. These classes should rarely be used directly by the developer. They are instead used by the different types of configuration scanners.

2.5. Testable Code

One key value of using the IoC container is the how you can isolate parts of your code for better testing. Imagine you had the following configuration:

```
from springpython.config import *
from springpython.context import *
```

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

    @Object(scope.PROTOTYPE)
    def MovieLister(self):
        lister = MovieLister()
        lister.finder = self.MovieFinder()
        lister.description = self.SingletonString()
        self.logger.debug("Description = %s" % lister.description)
        return lister

    @Object(scope.SINGLETON)
    def MovieFinder(self):
        return ColonMovieFinder(filename="support/movies1.txt")

    @Object    # scope.SINGLETON is the default
    def SingletonString(self):
        return StringHolder("There should only be one copy of this string")
```

To inject a test double for `MovieFinder`, your test code would only have to extend the class and override the `MovieFinder` method, and replace it with your stub or mock object. Now you have a nicely isolated instance of `MovieLister`.

```
class MyTestableAppContext(MovieBasedApplicationContext):
    def __init__(self):
        super(MyTestableAppContext, self).__init__()

    @Object
    def MovieFinder(self):
        return MovieFinderStub()
```

2.6. Querying and modifying the ApplicationContext in runtime

ApplicationContext instances expose two attributes and an utility method which let you learn about their current state and dynamically alter them in runtime.

- `object_defs` is a dictionary of objects definitions, that is, the templates based upon which the container will create appropriate objects, e.g. your singletons,
- `objects` is a dictionary of already created objects stored for later use,
- `get_objects_by_type(type, include_type=True)` returns those ApplicationContext's objects which are instances of a given type or of its subclasses. If *include_type* is False then only instances of the type's *subclasses* will be returned.

Here's an example showing how you can easily query a context to find out what definitions and objects it holds. The context itself is stored using PythonConfig in the `sample_context.py` module and `demo.py` contains the code which examines the context.

```
#
# sample_context.py
#

from springpython.config import Object
from springpython.context import scope
from springpython.config import PythonConfig

class MyClass(object):
    pass
```

```
class MySubclass(MyClass):
    pass

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

    @Object
    def http_port(self):
        return 18000

    @Object
    def https_port(self):
        return self._get_https_port()

    def _get_https_port(self):
        return self.http_port() + 443

    @Object
    def my_class_object1(self):
        return MyClass()

    @Object
    def my_class_object2(self):
        return MyClass()

    @Object
    def my_subclass_object1(self):
        return MySubclass()

    @Object
    def my_subclass_object2(self):
        return MySubclass()

    @Object
    def my_subclass_object3(self):
        return MySubclass()


#
# demo.py
#

# Spring Python
from springpython.context import ApplicationContext

# Our sample code.
from sample_context import SampleContext, MyClass, MySubclass

# Create the context.
ctx = ApplicationContext(SampleContext())

# Do we have an 'http_port' object?
print "http_port" in ctx.objects

# Does the context have a definition of an 'ftp_port' object?
print "ftp_port" in ctx.object_defs

# How many objects are there? Observe the result is 7, that's because one of
# the methods - _get_https_port - is not managed by the container.
print len(ctx.objects)

# List the names of all objects defined.
print ctx.object_defs.keys()

# Returns all instances of MyClass and of its subclasses.
print ctx.get_objects_by_type(MyClass)

# Returns all instances of MyClass' subclasses only.
print ctx.get_objects_by_type(MyClass, False)

# Returns all integer objects.
print ctx.get_objects_by_type(int)
```

The .object_defs dictionary stores instances of `springpython.config.ObjectDef` class, these are the objects you need to inject into the container to later successfully access them as if they were added prior to the application's start. An `ObjectDef` allows one to specify the very same set of parameters an `@Object` decorator does. The next examples shows how to insert two definitions into a context, one will be a prototype - a new instance of `Foo` will be created on each request, the second one will be a singleton - only one instance of `Bar` will ever be created and stored in a cache of singletons. This time the example employs the Python's standard library `logging` module to better show in the DEBUG mode what is going on under the hood.

```
#
# sample_context2.py
#


# Spring Python
from springpython.config import PythonConfig

class SampleContext2(PythonConfig):
    def __init__(self):
        super(SampleContext2, self).__init__()


#
# demo2.py
#

# stdlib
import logging

# Spring Python
from springpython.config import Object, ObjectDef
from springpython.context import ApplicationContext
from springpython.factory import PythonObjectFactory
from springpython.context.scope import SINGLETON, PROTOTYPE

# Our sample code.
from sample_context2 import SampleContext2

# Configure logging.
log_format = "%(msecs)d - %(levelname)s - %(name)s - %(message)s"
logging.basicConfig(level=logging.DEBUG, format=log_format)

class Foo(object):
    def run(self):
        return "Foo!"

class Bar(object):
    def run(self):
        return "Bar!"

# Create the context - part 1. in the logs.
ctx = ApplicationContext(SampleContext2())

# Definitions of objects that will be dynamically injected into container.

@Object(PROTOTYPE)
def foo():
    """ Returns a new instance of Foo on each call.
    """
    return Foo()

@Object # SINGLETON is the default.
def bar():
    """ Returns a singleton Bar every time accessed.
    """
    return Bar()

# A reference to the function wrapping the actual 'foo' function.
foo_wrapper = foo.func_globals["_call_"]

# Create an object definition, note that we're telling to return
foo_object_def = ObjectDef(id="foo", factory=PythonObjectFactory(foo, foo_wrapper), scope=PROTOTYPE, lazy_init=f

# A reference to the function wrapping the actual 'bar' function.
bar_wrapper = foo.func_globals["_call_"]

bar_object_def = ObjectDef(id="foo", factory=PythonObjectFactory(bar, bar_wrapper), scope=SINGLETON, lazy_init=b
ctx.object_defs["foo"] = foo_object_def
ctx.object_defs["bar"] = bar_object_def

# Access "foo" - part 2. in the logs.
for x in range(3):
    foo_instance = ctx.get_object("foo")

# Access "bar" - part 3. in the logs.
for x in range(3):
    bar_instance = ctx.get_object("bar")
```

Here's how it shows in the logs. For clarity, the log has been divided into three parts. Part 1. reads the object definitions from SampleContext2, as we see, nothing has been read from it as it's still been empty at this point. After adding definitions to the .object_defs dictionary, we're now at parts 2. and 3. - in 2. the 'foo' object, a prototype one, is being created three times, as expected. In part 3. the singleton 'bar' object is created and stored in a singleton cache once only even though we're accessing it three times in our code.

```

# Part 1.

100 - DEBUG - springpython.config.PythonConfig - ==============================================================
100 - DEBUG - springpython.config.PythonConfig - Parsing <sample_context2.SampleContext2 object at 0x17e70d0>
101 - DEBUG - springpython.config.PythonConfig - ==============================================================
101 - DEBUG - springpython.container.ObjectContainer - === Done reading object definitions. ===

# Part 2.

102 - DEBUG - springpython.context.ApplicationContext - Did NOT find object 'foo' in the singleton storage.
102 - DEBUG - springpython.context.ApplicationContext - Creating an instance of id=foo props=[] scope=scope.PROT
102 - DEBUG - springpython.factory.PythonObjectFactory - Creating an instance of foo
102 - DEBUG - springpython.config.objectPrototype<function foo at 0x7f6d15db0a28> - ()scope.PROTOTYPE - This IS
102 - DEBUG - springpython.config.objectPrototype<function foo at 0x7f6d15db0a28> - ()scope.PROTOTYPE - Found <_

102 - DEBUG - springpython.context.ApplicationContext - Did NOT find object 'foo' in the singleton storage.
102 - DEBUG - springpython.context.ApplicationContext - Creating an instance of id=foo props=[] scope=scope.PROT
102 - DEBUG - springpython.factory.PythonObjectFactory - Creating an instance of foo
103 - DEBUG - springpython.config.objectPrototype<function foo at 0x7f6d15db0a28> - ()scope.PROTOTYPE - This IS
103 - DEBUG - springpython.config.objectPrototype<function foo at 0x7f6d15db0a28> - ()scope.PROTOTYPE - Found <_

103 - DEBUG - springpython.context.ApplicationContext - Did NOT find object 'foo' in the singleton storage.
103 - DEBUG - springpython.context.ApplicationContext - Creating an instance of id=foo props=[] scope=scope.PROT
103 - DEBUG - springpython.factory.PythonObjectFactory - Creating an instance of foo
103 - DEBUG - springpython.config.objectPrototype<function foo at 0x7f6d15db0a28> - ()scope.PROTOTYPE - This IS
103 - DEBUG - springpython.config.objectPrototype<function foo at 0x7f6d15db0a28> - ()scope.PROTOTYPE - Found <_

# Part 3.

103 - DEBUG - springpython.context.ApplicationContext - Did NOT find object 'bar' in the singleton storage.
103 - DEBUG - springpython.context.ApplicationContext - Creating an instance of id=foo props=[] scope=scope.SING
103 - DEBUG - springpython.factory.PythonObjectFactory - Creating an instance of bar
104 - DEBUG - springpython.config.objectSingleton<function bar at 0x17e5aa0> - ()scope.SINGLETON - This IS the t
104 - DEBUG - springpython.config.objectSingleton<function bar at 0x17e5aa0> - ()scope.SINGLETON - Found <__mair
104 - DEBUG - springpython.context.ApplicationContext - Stored object 'bar' in container's singleton storage
```

Please note that what has been shown above applies to runtime only, adding object definitions to the container doesn't mean the changes will be in any way serialized to the file system, they are transient and will be lost when the application will be shutting down. Another thing to keep in mind is that you'll be modifying a raw Python dictionary and if your application is multi-threaded, you'll have to serialize the access from concurrent threads yourself.

Chapter 3. Aspect Oriented Programming

Aspect oriented programming (AOP) is a horizontal programming paradigm, where some type of behavior is applied to several classes that don't share the same vertical, object-oriented inheritance. In AOP, programmers implement these *cross cutting concerns* by writing an *aspect* then applying it conditionally based on a *join point*. This is referred to as applying *advice*. This section shows how to use the AOP module of Spring Python.

3.1. External dependencies

Spring Python's AOP itself doesn't require any special external libraries to work however the IoC configuration format of your choice, unless you use PythonConfig, will likely need some. Refer to the IoC documentation for more details.

3.2. Interceptors

Spring Python implements AOP advice using *proxies* and *method interceptors*. NOTE: Interceptors only apply to method calls. Any request for attributes are passed directly to the target without AOP intervention.

Here is a sample service. Our goal is to wrap the results with "wrapped" tags, without modifying the service's code.

```
class SampleService:
    def method(self, data):
        return "You sent me '%s'" % data
    def doSomething(self):
        return "Okay, I'm doing something"
```

If we instantiate and call this service directly, the results are straightforward.

```
service = SampleService()
print service.method("something")

"You sent me 'something'"
```

To configure the same thing using the IoC container, put the following text into a file named `app-context.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

        <object id="service" class="SampleService"/>

</objects>
```

To instantiate the IoC container, use the following code.

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

container = ApplicationContext(XMLConfig("app-context.xml"))
service = container.get_object("service")
```

```

```

You can use either mechanism to define an instance of your service. Now, let's write an interceptor that will catch any results, and wrap them with <Wrapped> tags.

```
from springpython.aop import *
class WrappingInterceptor(MethodInterceptor):
    def invoke(self, invocation):
        results = "<Wrapped>" + invocation.proceed() + "</Wrapped>"
        return results
```

`invoke(self, invocation)` is a dispatching method defined abstractly in the `MethodInterceptor` base class. `invocation` holds the target method name, any input arguments, and also the callable target function. In this case, we aren't interested in the method name or the arguments. So we call the actual function using `invocation.proceed()`, and than catch its results. Then we can manipulate these results, and return them back to the caller.

In order to apply this advice to a service, a stand-in *proxy* must be created and given to the client. One way to create this is by creating a `ProxyFactory`. The factory is used to identify the target service that is being intercepted. It is used to create the dynamic proxy object to give back to the client.

You can use the Spring Python APIs to directly create this proxied service.

```
from springpython.aop import *
factory = ProxyFactory()
factory.target = SampleService()
factory.interceptors.append(WrappingInterceptor())
service = factory.getProxy()
```

Or, you can insert this definition into your `app-context.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

        <object id="targetService" class="SampleService"/>

        <object id="serviceFactory" class="springpython.aop.ProxyFactory">
                <property name="target" ref="targetService"/>
                <property name="interceptors">
                        <object class="WrappingInterceptor"/>
                </property>
        </object>

</objects>
```

If you notice, the original Spring Python "service" object has been renamed as "targetService", and there is, instead, another object called "serviceFactory" which is a Spring AOP ProxyFactory. It points to the target service and also has an interceptor plugged in. In this case, the interceptor is defined as an inner object, not having a name of its own, indicating it is not meant to be referenced outside the IoC container. When you get a hold of this, you can request a proxy.

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

container = ApplicationContext(XMLConfig("app-context.xml"))
serviceFactory = container.get_object("serviceFactory")
```

```
service = serviceFactory.getProxy()
```

Now, the client can call *service*, and all function calls will be routed to `SampleService` with one simple detour through `WrappingInterceptor`.

```
print service.method("something")

"<Wrapped>You sent me 'something'</Wrapped>"
```

Notice how I didn't have to edit the original service at all? I didn't even have to introduce Spring Python into that module. Thanks to the power of Python's dynamic nature, Spring Python AOP gives you the power to wrap your own source code as well as other 3rd party modules.

3.3. Proxy Factory Objects

The earlier usage of a `ProxyFactory` is useful, but often times we only need the factory to create one proxy. There is a shortcut called `ProxyFactoryObject`.

```
from springpython.aop import *
service = ProxyFactoryObject()
service.target = SampleService()
service.interceptors = [WrappingInterceptor()]
print service.method(" proxy factory object")

"You sent me a 'proxy factory object'"
```

To configure the same thing into your `app-context.xml` file, it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

        <object id="targetService" class="SampleService"/>

        <object id="service" class="springpython.aop.ProxyFactoryObject">
                <property name="target" ref="targetService"/>
                <property name="interceptors">
                        <object class="WrappingInterceptor"/>
                </property>
        </object>

</objects>
```

In this case, the `ProxyFactoryObject` acts as both a proxy and a factory. As a proxy, it behaves just like the target service would, and it also provides the ability to wrap the service with aspects. It saved us a step of coding, but more importantly, the `ProxyFactoryObject` took on the persona of being our service right from the beginning.

To be more pythonic, Spring Python also allows you to initialize everything at once.

```
from springpython.aop import *
service = ProxyFactoryObject(target = SampleService(), interceptors = [WrappingInterceptor()])
```

3.4. Pointcuts

Sometimes we only want to apply advice to certain methods. This requires definition of a *join point*. Join points are composed of rules referred to as *point cuts*.

In this case, we want to only apply our `WrappingInterceptor` to methods that start with "do".

```
from springpython.aop import *
pointcutAdvisor = RegexpMethodPointcutAdvisor(advice = [WrappingInterceptor()], patterns = [".*do.*"])
service = ProxyFactoryObject(target = SampleService(), interceptors = pointcutAdvisor)
print service.method("nothing changed here")

"You sent me 'nothing changed here'"

print service.doSomething()

"<Wrapped>Okay, I'm doing something</Wrapped"
```

The power of RegexpMethodPointAdvisor

`RegexpMethodPointAdvisor` is a powerful object in Spring Python that acts as *pointcut*, a *join point*, and a *method interceptor*. It fetches the fullpath of the target's module, class, and method name, and then checks to see if it matches any of the patterns in the *patterns* list using Python's regular expression module.

By plugging this into a `ProxyFactoryObject` as an interceptor, it evaluates whether to route method calls through the advice, or directly to the target service.

3.5. Interceptor Chain

You may have noticed by now that the `WrappingInterceptor` is being specified inside a Python list. That is because you can apply more than one piece of advice. When an interceptor calls `invocation.proceed()`, it is actually calling the next interceptor in the chain, until it gets to the end. Then it calls the actual target service. When the target method returns back, everything backtracks back out the set of nested interceptors.

Spring Python is coded to intelligently detect if you are assigning a single interceptor to the *interceptors* property, or a list. A single interceptor gets converted into a list of one. So, you can do either of the following:

```
service = ProxyFactoryObject()
service.interceptors = WrappingInterceptor()
```

or

```
service = ProxyFactoryObject()
factory.interceptors = [WrappingInterceptor()]
```

It produces the same thing.

3.6. Coding AOP with Pure Python

There is a long history of Spring being based on XML. However, Spring Python offers an easy to use alternative: a pure python decorator-based PythonConfig. Imagine you had set up a simple context like this:

```
from springpython.config import *
from springpython.context import *

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

    @Object(scope.PROTOTYPE)
    def MovieLister(self):
        lister = MovieLister()
        lister.finder = self.MovieFinder()
        lister.description = self.SingletonString()
        self.logger.debug("Description = %s" % lister.description)
        return lister

    @Object(scope.SINGLETON)
    def MovieFinder(self):
        return ColonMovieFinder(filename="support/movies1.txt")

    @Object    # scope.SINGLETON is the default
    def SingletonString(self):
        return StringHolder("There should only be one copy of this string")
```

From an AOP perspective, it is easy to intercept `MovieFinder` and wrap it with some advice. Because you have already exposed it as an injection point with this pure-python IoC container, you just need to make this change:

```
from springpython.aop import *
from springpython.config import *
from springpython.context import *

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

    @Object(scope.PROTOTYPE)
    def MovieLister(self):
        lister = MovieLister()
        lister.finder = self.MovieFinder()
        lister.description = self.SingletonString()
        self.logger.debug("Description = %s" % lister.description)
        return lister

    # By renaming the original service to this...
    def targetMovieFinder(self):
        return ColonMovieFinder(filename="support/movies1.txt")

    #...we can substitute a proxy that will wrap it with an interceptor
    @Object(scope.SINGLETON)
    def MovieFinder(self):
            return ProxyFactoryObject(target=self.targetMovieFinder(),
                                     interceptors=MyInterceptor())


    @Object    # scope.SINGLETON is the default
    def SingletonString(self):
        return StringHolder("There should only be one copy of this string")

class MyInterceptor(MethodInterceptor):
    def invoke(self, invocation):
        results = "<Wrapped>" + invocation.proceed() + "</Wrapped>"
        return results
```

Now, everything that was referring to the original `ColonMovieFinder` instance, is instead pointing to a wrapping interceptor. The caller and callee involved don't know anything about it, keeping your code isolated and clean.

Shouldn't you decouple the interceptor from the IoC configuration?

It is usually good practice to split up configuration from actual business code. These two were put together in the same file for demonstration purposes.

Chapter 4. Data Access

4.1. DatabaseTemplate

Writing SQL-based programs has a familiar pattern that must be repeated over and over. The DatabaseTemplate resolves that by handling the plumbing of these operations while leaving you in control of the part that matters the most, the SQL.

4.1.1. External dependencies

DatabaseTemplate requires the use of external libraries for connecting to SQL databases. Depending on which SQL connection factory you're about to use, you need to install following dependencies:

- `springpython.database.factory.MySQLConnectionFactory` - needs MySQLdb for connecting to MySQL,
- `springpython.database.factory.PgdbConnectionFactory` - needs PyGreSQL for connecting to PostgreSQL,
- `springpython.database.factory.Sqlite3ConnectionFactory` - needs PySQLite for connecting to SQLite 3, note that PySQLite is part of Python 2.5 and later so you need to install it separately only if you're using Python 2.4,
- `springpython.database.factory.cxoraConnectionFactory` - needs cx_Oracle for connecting to Oracle,
- `springpython.database.factory.SQLServerConnectionFactory` - needs PyODBC for connecting to SQL Server.

4.1.2. Traditional Database Query

If you have written a database SELECT statement following Python's DB-API 2.0, it would something like this (MySQL example):

```
conn = MySQL.connection(username="me", password"secret", hostname="localhost", db="springpython")
cursor = conn.cursor()
results = []
try:
    cursor.execute("select title, air_date, episode_number, writer from tv_shows where name = %s", ("Monty Pytho
    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
```

I know, you don't have to open and close a connection for every query, but let's look past that part. In every definition of a SQL query, you must create a new cursor, execute against the cursor, loop through the results, and most importantly (and easy to forget) *close the cursor*. Of course you will wrap this in a method instead of plugging in this code where ever you need the information. But every time you need another query, you have to repeat this dull pattern over and over again. The only thing different is the actual SQL code you must write and converting it to a list of objects.

I know there are many object relational mappers (ORMs) out there, but sometimes you need something simple and sweet. That is where `DatabaseTemplate` comes in.

4.1.3. Database Template

The same query above can be written using a `DatabaseTemplate`. The only thing you must provide is the SQL and a `RowMapper` to process one row of data. The template does the rest.

```
""" The following part only has to be done once."""
from springpython.database.core import *
from springpython.database.factory import *
connectionFactory = MySQLConnectionFactory(username="me", password="secret", hostname="localhost", db="springpyt
dt = DatabaseTemplate(connectionFactory)

class TvShowMapper(RowMapper):
    """This will handle one row of database. It can be reused for many queries if they
       are returning the same columns."""
    def map_row(self, row, metadata=None):
        return TvShow(title=row[0], airDate=row[1], episodeNumber=row[2], writer=row[3])


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

Well, no sign of a cursor anywhere. If you didn't have to worry about opening it, you don't have to worry about closing it. I know this is about the same amount of code as the traditional example. Where DatabaseTemplate starts to shine is when you want to write ten different TV_SHOW queries.

```
results = dt.query("select title, air_date, episode_number, writer from tv_shows where episode_number < %s", \
                   (100,), TvShowMapper())
results = dt.query("select title, air_date, episode_number, writer from tv_shows where upper(title) like %s", \
                   ("%CHEESE%",), TvShowMapper())
results = dt.query("select title, air_date, episode_number, writer from tv_shows where writer in ('Cleese', 'Gra
                   rowhandler=TvShowMapper())
```

You don't have to reimplement the rowhandler. For these queries, you can focus on the SQL you want to write, not the mind-numbing job of managing database cursors.

4.1.4. Mapping rows into objects using convention over configuration

A powerful feature provided by databases is the ability to look up column names. Spring Python harnesses this by providing an out-of-the-box row mapper that will automatically try matching a query column name to an class attribute name. This is known as *convention over configuration* because it relieves you of the need to code the `RowMapper` provided you follow the convention of naming the attributes of your POPO after query columns. The only requirement is that class have a default constructor that doesn't require any arguments.

```
results = dt.query("select title, air_date, episode_number, writer from tv_shows where episode_number < %s", \
                   (100,), SimpleRowMapper(TvShow))
results = dt.query("select title, air_date, episode_number, writer from tv_shows where upper(title) like %s", \
                   ("%CHEESE%",), SimpleRowMapper(TvShow))
results = dt.query("select title, air_date, episode_number, writer from tv_shows where writer in ('Cleese', 'Gra
                   rowhandler=SimpleRowMapper(TvShow))
```

Convention is based on query, not tables

Query metadata is based on the column names as defined in the query, NOT what is in the table. This is important when you use expressions like COUNT(*). These columns should be aliased to fit the attribute name.

4.1.5. Mapping rows into dictionaries

A convenient alternative to mapping database rows into python objects, it to map them into dictionaries. Spring Python offers `DictionaryRowMapper` as an out-of-the-box way to query the database, and return a list of dictionary entries, based on the column names of the queries. Using this mapper, you don't have to code a `TvRowMapper` as shown earlier.

```
results = dt.query("select title, air_date, episode_number, writer from tv_shows where episode_number < %s", \
                   (100,), DictionaryRowMapper())
results = dt.query("select title, air_date, episode_number, writer from tv_shows where upper(title) like %s", \
                   ("%CHEESE%",), DictionaryRowMapper())
results = dt.query("select title, air_date, episode_number, writer from tv_shows where writer in ('Cleese', 'Gra
                   rowhandler=DictionaryRowMapper())
```

Dictionary keys are based on query not original tables

Query metadata is based on the column names as defined in the query, NOT what is in the table. This is important when you use expressions like COUNT(*). These columns should be aliased in order to generate a useful key in the dictionary.

4.1.6. What is a Connection Factory?

You may have noticed I didn't make a standard connection in the example above. That is because to support Dependency Injection, I need to setup my credentials in an object before making the actual connection. `MySQLConnectionFactory` holds credentials specific to the MySQL DB-API, but contains a common function to actually create the connection. I don't have to use it myself. `DatabaseTemplate` will use it when necessary to create a connection, and then proceed to reuse the connection for subsequent database calls.

That way, I don't manage database connections and cursors directly, but instead let Spring Python do the heavy lifting for me.

4.1.7. Creating/altering tables, databases, and other DDL

Data Definition Language includes the database statements that involve creating and altering tables, and so forth. DB-API defines an execute function for this. `DatabaseTemplate` offers the same. Using the execute() function will pass through your request to a cursor, along with the extra exception handler and cursor management.

4.1.8. SQL Injection Attacks

You may have noticed in the first three example queries I wrote with the `DatabaseTemplate`, I embedded a "%s" in the SQL statement. These are called *binding variables*, and they require a tuple argument be included after the SQL statement. Do *NOT* include quotes around these variables. The database connection will handle that. This style of SQL programming is *highly recommended* to avoid SQL injection attacks.

For users who are familiar with Java database APIs, the binding variables are cited using "?" instead of "%s".

To make both parties happy and help pave the way for existing Java programmers to use this framework, I have included support for both. You can mix-and-match these two binding variable types as you wish, and things will still work.

4.1.9. Have you used Spring Framework's JdbcTemplate?

If you are a user of Java's Spring framework and have used the JdbcTemplate, then you will find this template has a familiar feel.

Table 4.1. `JdbcTemplate` operations also found in `DatabaseTemplate`

`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 DB-API tuples (or a dictionary if you use sqlWrappy)
`query_for_int(sql_query, args = None)`	query for a single column of a single row, and return an integer (throws exception otherwise)
`query_for_long(sql_query, args = None)`	query for a single column of a single row, and return a long (throws 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 possibly no checking
`update(sql_statement, args = None)`	update the database, return number of rows updated

Inserts are implemented through the execute() function, just like in JdbcTemplate.

4.1.10. Notes on using SQLServerConnectionFactory

`SQLServerConnectionFactory` uses ODBC for connecting to SQL Server instances and it expects you to pass the ODBC parameters when creating connection factories or when injecting factory settings through IoC. The ODBC parameters you provide are directly translated into an ODBC connection string.

That means that you use the exact ODBC parameters your ODBC provider understands and not the standard *username*, *password*, *hostname* and *db* parameters as with other connection factories.

A simple example will demonstrate this. Here's how you would create a `DatabaseTemplate` on Windows for running queries against an SQL Server instance.

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

driver = "{SQL Server}"
server = "localhost"
database = "springpython"
uid = "springpython"
pwd = "cdZS*RQRBdc9a"

factory = SQLServerConnectionFactory(DRIVER=driver, SERVER=server, DATABASE=database, UID=uid, PWD=pwd)
dt = DatabaseTemplate(factory)
```

SQLServerConnectionFactory is dictionary driven

Due to `SQLServerConnectionFactory`'s pass-through nature, it is coded to accept a dictionary. For pure python, this means you MUST name the arguments and NOT rely on argument position.

For an XML-based application context, you must populate the argument `odbc_info` with a dictionary. See the following example.

```xml

<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

        <object id="connection_factory" class="springpython.database.factory.SQLServerConnectionFactory">
        <property name="odbc_info">
            <dict>
                <entry>
                    <key><value>DRIVER</value></key>
                    <value>{SQL Server}</value>
                </entry>
                <entry>
                    <key><value>SERVER</value></key>
                    <value>localhost</value>
                </entry>
                <entry>
                    <key><value>DATABASE</value></key>
                    <value>springpython</value>
                </entry>
                <entry>
                    <key><value>UID</value></key>
                    <value>springpython</value>
                </entry>
                <entry>
                    <key><value>PWD</value></key>
                    <value>cdZS*RQRBdc9a</value>
                </entry>
            </dict>
        </property>
    </object>

</objects>
```

Chapter 5. Transaction Management

When writing a program with database operations, you may need to use transactions. Your code can get ugly, and it often becomes hard to read the business logic due to starting, committing, or rolling back for various reasons. Another risk is that some of the transaction management code you write will have all the necessary steps, while you may forget some important steps in others. Spring Python offers a key level of abstraction that can remove that burden and allow you to focus on the business logic.

5.1. External dependencies

If you choose to use DatabaseTemplate along with Spring Python's support for transaction management you need to install an appropriate SQL database driver module. Depending on the IoC configuration format you're going to use you may also need to install one of its documented dependencies.

5.2. Solutions requiring transactions

For simple transactions, you can embed them programmatically.

Seen anything like this before?

```
def transfer(transfer_amount, source_account_num, target_account_num):
    conn = MySQLdb.connection("springpython", "springpython", "localhost", "springpython")
    cursor = conn.cursor()
    cursor.execute("update ACCOUNT set BALANCE = BALANCE - %s where ACCOUNT_NUM = %s", (transfer_amount, source_
    cursor.execute("update ACCOUNT set BALANCE = BALANCE + %s where ACCOUNT_NUM = %s", (transfer_amount, target_
    cursor.close()
```

This business method defines a transfer between bank accounts. Notice any issues here? What happens if the target account doesn't exist? What about transferring a negative balance? What if the transfer amount exceeded the source account's balance? All these things require checks, and if something is wrong the entire transfer must be aborted, or you find the first bank account leaking money.

To wrap this function transactionally, based on DB-2.0 API specifications, we'll add some checks. I have also completed some refactorings and utilized the `DatabaseTemplate` to clean up my database code.

```
from springpython.database import *
from springpython.database.core import *
import types
class Bank:
    def __init__(self):
        self.factory = factory.MySQLConnectionFactory("springpython", "springpython", "localhost", "springpythor
        self.dt = DatabaseTemplate(self.factory)

    def balance(self, account_num):
        results = self.dt.query_for_list("select BALANCE from ACCOUNT where ACCOUNT_NUM = %s", (account_num,))
        if len(results) != 1:
            raise InvalidBankAccount("There were %s accounts that matched %s." % (len(results), account_num))
        return results[0][0]

    def checkForSufficientFunds(self, source_balance, amount):
        if source_balance < amount:
            raise InsufficientFunds("Account %s did not have enough funds to transfer %s" % (source_account_num,

    def withdraw(self, amount, source_account_num):
        self.checkForSufficientFunds(self.balance(source_account_num), amount)
        self.dt.execute("update ACCOUNT set BALANCE = BALANCE - %s where ACCOUNT_NUM = %s", (amount, source_acco

    def deposit(self, amount, target_account_num):
        # Implicitly testing for valid account number
        self.balance(target_account_num)
        self.dt.execute("update ACCOUNT set BALANCE = BALANCE + %s where ACCOUNT_NUM = %s", (amount, target_acco

    def transfer(self, transfer_amount, source_account_num, target_account_num):
        try:
            cursor = self.factory.getConnection().cursor() # DB-2.0 API spec says that creating a cursor implici
            self.withdraw(transfer_amount, source_account_num)
            self.deposit(transfer_amount, target_account_num)
            self.factory.getConnection().commit()
            cursor.close() # There wasn't anything in this cursor, but it is good to close an opened cursor
        except InvalidBankAccount, InsufficientFunds:
            self.factory.getConnection().rollback()
```

- This has some extra checks put in to protect from overdrafts and invalid accounts.
- `DatabaseTemplate` removes our need to open and close cursors.
- Unfortunately, we still have to tangle with them as well as the connection in order to handle transactions.

5.3. `TransactionTemplate`

We still have to deal with exceptions. What if another part of the code raised another exception that we didn't trap? It might escape our try-except block of code, and then our data could lose integrity. If we plug in the `TransactionTemplate`, we can really simplify this and also guarantee management of any exceptions.

The following code block shows swapping out manual transaction for `TransactionTemplate`.

```

from springpython.database.transaction import *
..
..
..
class Bank:
    def __init__(self):
        self.factory = factory.MySQLConnectionFactory("springpython", "springpython", "localhost", "springpython
        self.dt = DatabaseTemplate(self.factory)
        self.txManager = ConnectionFactoryTransactionManager(self.factory)
        self.txTemplate = TransactionTemplate(self.txManager)
..
..
..
    def transfer(self, transfer_amount, source_account_num, target_account_num):
        class txDefinition(TransactionCallbackWithoutResult):
                    def doInTransactionWithoutResult(s, status):
                       self.withdraw(transfer_amount, source_account_num)
                       self.deposit(transfer_amount, target_account_num)
        try:
            self.txTemplate.execute(txDefinition())
            print "If you made it to here, then your transaction has already been committed."
        except InvalidBankAccount, InsufficientFunds:
            print "If you made it to here, then your transaction has already been rolled back."
```

- We changed the init function to setup a `TransactionManager` (based on ConnectionFactory) and also a `TransactionTemplate`.
- We also rewrote the transfer function to generate a callback.

Now you don't have to deal with implicit cursors, commits, and rollbacks. Managing commits and rollbacks can really complicated especially when dealing with exceptions. By wrapping it into a nice callback, `TransactionTemplate` does the work for us, and lets us focus on business logic, while encouraging us to continue to define meaningful business logic errors.

5.4. `@transactional`

Another option is to use the `@transactional` decorator, and mark which methods should be wrapped in a transaction when called.

```
from springpython.database.transaction import *
..
..
..
class Bank:
    def __init__(self, connectionFactory):
        self.factory = connectionFactory):
        self.dt = DatabaseTemplate(self.factory)
..
..
..
    @transactional
    def transfer(self, transfer_amount, source_account_num, target_account_num):
        self.withdraw(transfer_amount, source_account_num)
        self.deposit(transfer_amount, target_account_num)
```

This needs to be wired together with a `TransactionManager` in an `ApplicationContext`. The following example shows a `PythonConfig` with three objects:

- the bank
- a `TransactionManager` (in this case `ConnectionFactoryTransactionManager`)
- an `AutoTransactionalObject`, which checks all objects to see if they have `@transactional` methods, and if so, links them with the `TransactionManager`.

The name of the method (i.e. component name) for `AutoTransactionalObject` doesn't matter.

```
class DatabaseTxTestDecorativeTransactions(PythonConfig):
    def __init__(self, factory):
            super(DatabaseTxTestDecorativeTransactions, self).__init__()
        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 TransactionalBank(self.factory)
```

This can also be configured using `XMLConfig`

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

        <object id="transactionalObject" class="springpython.database.transaction.AutoTransactionalObject">
                <constructor-arg ref="tx_mgr"/>
        </object>

        <object id="tx_mgr" class="springpython.database.transaction.ConnectionFactoryTransactionManager">
```

```
                <constructor-arg ref="factory"/>
        </object>

        <object id="factory" class="...your DB connection factory definition here..."/>

        <object id="bank" class="TransactionalBank">
                <constructor-arg ref="factory"/>
        </object>

</objects>
```

5.4.1. `@transactional(["PROPAGATION_REQUIRED"])`...

Declarative transactions includes the ability to define transaction propagation. This allows you to define when a transaction should be started, and which operations need to be part of transactions. There are several levels of propagation defined:

- PROPAGATION_SUPPORTS - Code can run inside or outside a transaction.
- PROPAGATION_REQUIRED - If there is no current transaction, one will be started.
- PROPAGATION_MANDATORY - Code MUST be run inside an already started transaction.
- PROPAGATION_NEVER - Code must NOT be run inside an existing transaction.

The following code is a revision of the Bank class, with this attribute plugged in:

```

class TransactionalBankWithLotsOfTransactionalArguments(object):
    """This sample application can be used to demonstrate the value of atomic operations. The transfer operation
    must be wrapped in a transaction in order to perform correctly. Otherwise, any errors in the deposit will
    allow the from-account to leak assets."""
    def __init__(self, factory):
        self.logger = logging.getLogger("springpython.test.testSupportClasses.TransactionalBankWithLotsOfTransac
        self.dt = DatabaseTemplate(factory)

    @transactional(["PROPAGATION_REQUIRED"])
    def open(self, accountNum):
        self.logger.debug("Opening account %s with $0 balance." % accountNum)
        self.dt.execute("INSERT INTO account (account_num, balance) VALUES (?,?)", (accountNum, 0))

    @transactional(["PROPAGATION_REQUIRED"])
    def deposit(self, amount, accountNum):
        self.logger.debug("Depositing $%s into %s" % (amount, accountNum))
        rows = self.dt.execute("UPDATE account SET balance = balance + ? WHERE account_num = ?", (amount, accour
        if rows == 0:
            raise BankException("Account %s does NOT exist" % accountNum)

    @transactional(["PROPAGATION_REQUIRED"])
    def withdraw(self, amount, accountNum):
        self.logger.debug("Withdrawing $%s from %s" % (amount, accountNum))
        rows = self.dt.execute("UPDATE account SET balance = balance - ? WHERE account_num = ?", (amount, accour
        if rows == 0:
            raise BankException("Account %s does NOT exist" % accountNum)
        return amount

    @transactional(["PROPAGATION_SUPPORTS","readOnly"])
    def balance(self, accountNum):
        self.logger.debug("Checking balance for %s" % accountNum)
        return self.dt.queryForObject("SELECT balance FROM account WHERE account_num = ?", (accountNum,), types.

    @transactional(["PROPAGATION_REQUIRED"])
    def transfer(self, amount, fromAccountNum, toAccountNum):
        self.logger.debug("Transferring $%s from %s to %s." % (amount, fromAccountNum, toAccountNum))
        self.withdraw(amount, fromAccountNum)
        self.deposit(amount, toAccountNum)

    @transactional(["PROPAGATION_NEVER"])
    def nonTransactionalOperation(self):
        self.logger.debug("Executing non-transactional operation.")

    @transactional(["PROPAGATION_MANDATORY"])
    def mandatoryOperation(self):
        self.logger.debug("Executing mandatory transactional operation.")

    @transactional(["PROPAGATION_REQUIRED"])
    def mandatoryOperationTransactionalWrapper(self):
        self.mandatoryOperation()
        self.mandatoryOperation()

    @transactional(["PROPAGATION_REQUIRED"])
    def nonTransactionalOperationTransactionalWrapper(self):
        self.nonTransactionalOperation()
```

You will notice several levels are being utilized. This class was pulled directly from the test suite, so some of the functions are deliberately written to generate controlled failures.

If you look closely at *withdraw*, *deposit*, and *transfer*, which are all set to PROPAGATION_REQUIRED, you can see what this means. If you use *withdraw* or *deposit* by themselves, which require transactions, each will start and complete a transaction. However, *transfer* works by re-using these business methods. *Transfer* itself needs to be an entire transaction, so it starts one. When it calls *withdraw* and *deposit*, those methods don't need to start another transaction because they are already inside one. In comparison, *balance* is defined as PROPAGATION_SUPPORTS. Since it doesn't update anything, it can run by itself without a transaction. However, if it is called in the middle of another transaction, it will play along.

You may have noticed that *balance* also has "readOnly" defined. In the future, this may be passed onto the RDBMS in case the relational engine can optimize the query given its read-only nature.

Chapter 6. Security

Spring Python's Security module is based on Acegi Security's architecture. You can read Acegi's detailed reference manual for a background on this module.

Spring Security vs. Acegi Security

At the time this module was implemented, Spring Security was still Acegi Security. Links include reference documentation that was used at the time to implement this security module.

6.1. External dependencies

`springpython.security.cherrypy3` package depends on CherryPy 3 being installed prior to using them. Other than that, there are no specific external libraries required by Spring Python's security system, however the IoC configuration format that you'll be using may need some, check IoC documentation for more details.

6.2. Shared Objects

The major building blocks of Spring Python Security are

- `SecurityContextHolder`, to provide any type access to the `SecurityContext`.
- `SecurityContext`, to hold the Authentication and possibly request-specific security information.
- `HttpSessionContextIntegrationFilter`, to store the `SecurityContext` in the HTTP session between web requests.
- `Authentication`, to represent the principal in an Acegi Security-specific manner.
- `GrantedAuthority`, to reflect the application-wide permissions granted to a principal.

These objects are needed for both authentication and authorization.

6.3. Authentication

The first level of security involves verifying your credentials. Most systems today use some type of username/password check. To configure Spring Python, you will need to configure one or more `AuthenticationProvider`'s. All `Authentication` implementations are required to store an array of `GrantedAuthority` objects. These represent the authorities that have been granted to the principal. The `GrantedAuthority` objects are inserted into the `Authentication` object by the `AuthenticationManager` and are later read by `AccessDecisionManager`'s when making authorization decisions. These are chained together inside an `AuthenticationManager`.

6.3.1. AuthenticationProviders

6.3.1.1. DaoAuthenticationProvider

This `AuthenticationProvider` allows you to build a dictionary of user accounts, and is very handy for integration testing without resorting to complex configuration of 3rd party systems.

To configure this using a pythonic, decorator-based IoC container...

```
class SampleContainer(PythonConfig):
    ...
    @Object
    def inMemoryDaoAuthenticationProvider(self):
        provider = DaoAuthenticationProvider()
        provider.user_details_service = inMemoryUserDetailsService()
        return provider

    @Object
    def inMemoryUserDetailsService(self):
        user_details_service = InMemoryUserDetailsService()
        user_details_service.user_dict = {
           "vet1": ("password1", ["VET_ANY"], False),
           "bdavis": ("password2", ["CUSTOMER_ANY"], False),
           "jblack": ("password3", ["CUSTOMER_ANY"], False),
           "disableduser": ("password4", ["VET_ANY"], True),
           "emptyuser": ("", [], False) }
        return user_details_service
```

XML configuration using `XMLConfig`:

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

        <object id="inMemoryUserDetailsService" class="springpython.security.userdetails.InMemoryUserDetailsServ
                <property name="user_dict">
                        <dict>
                                <entry>
                                        <key><value>user1</value></key>
                                        <value>
                                                <tuple>
                                                        <value>password1</value>
                                                        <list><value>role1</value><value>blue</value></list>
                                                        <value>True</value>
                                                </tuple>
                                        </value>
                                </entry>
                                <entry>
                                        <key><value>user2</value></key>
                                        <value>
                                                <tuple>
                                                        <value>password2</value>
                                                        <list><value>role1</value><value>orange</value></list>
                                                        <value>True</value>
                                                </tuple>
                                        </value>
                                </entry>
                                <entry>
                                        <key><value>adminuser</value></key>
                                        <value>
                                                <tuple>
                                                        <value>password3</value>
                                                        <list><value>role1</value><value>admin</value></list>
                                                        <value>True</value>
                                                </tuple>
                                        </value>
                                </entry>
                                <entry>
                                        <key><value>disableduser</value></key>
                                        <value>
                                                <tuple>
                                                        <value>password4</value>
                                                        <list><value>role1</value><value>blue</value></list>
                                                        <value>False</value>
                                                </tuple>
                                        </value>
                                </entry>
                                <entry>
                                        <key><value>emptyuser</value></key>
                                        <value>
                                                <tuple>
                                                        <value/>
                                                        <list/>
                                                        <value>True</value>
                                                </tuple>
                                        </value>
                                </entry>
                        </dict>
                </property>
        </object>

        <object id="inMemoryDaoAuthenticationProvider" class="springpython.security.providers.dao.DaoAuthenticat
                <property name="user_details_service" ref="inMemoryUserDetailsService"/>
        </object>

</objects>
```

This is the user map defined for one of the test cases. The first user, user1, has a password of password1, a list of granted authorities ("role1", "blue"), and is enabled. The fourth user, "disableduser", has a password and a list of granted authorities, but is NOT enabled. The last user has no password, which will cause authentication to fail.

6.3.1.2. LDAP Authentication Provider

Spring Python has an `LdapAuthenticationProvider` that is able to authenticate users against an LDAP server using either binding or password comparison. It will also search the LDAP server for groups in order to identify roles.

Spring Python's LDAP only works with CPython

Currently, Spring Python only provides LDAP support for CPython. There is on-going effort to extend support to Jython as well.

It is possible to the customize the query parameters, as well as inject an alternative version of authentication as well as role identification.

There are two ways to verify a password in ldap: binding to the server using the password, or fetching the password from ldap and comparing outside the server. Spring Python supports both. You can choose which mechanism by injecting either a `BindAuthenticator` or a `PasswordComparisonAuthenticator` into `LdapAuthenticationProvider`.

The following XML fragment demonstrates how to configure Spring Python's `LdapAuthenticationProvider` using a `BindAuthenticator` combined with a `DefaultLdapAuthoritiesPopulator`.

```

        <object id="context_source" class="springpython.security.providers.Ldap.DefaultSpringSecurityContextSour
            <property name="url" value="ldap://localhost:53389/dc=springframework,dc=org"/>
        </object>

        <object id="bindAuthenticator" class="springpython.security.providers.Ldap.BindAuthenticator">
            <property name="context_source" ref="context_source"/>
            <property name="user_dn_patterns" value="uid={0},ou=people"/>
        </object>

        <object id="authoritiesPopulator" class="springpython.security.providers.Ldap.DefaultLdapAuthoritiesPopu
            <property name="context_source" ref="context_source"/>
            <property name="group_search_filter" value="member={0}"/>
        </object>

        <object id="ldapAuthenticationProvider" class="springpython.security.providers.Ldap.LdapAuthenticationPr
            <property name="ldap_authenticator" ref="bindAuthenticator"/>
            <property name="ldap_authorities_populator" ref="authoritiesPopulator"/>
        </object>

        <object id="ldapAuthenticationManager" class="springpython.security.providers.AuthenticationManager">
            <property name="auth_providers">
                <list><ref object="ldapAuthenticationProvider"/></list>
            </property>
        </object>
```

- *context_source* - points to an ldap server, defining the base DN to start searching for users and groups.
- *bindAuthenticator* - configured to use the context_source, and does a user search based on sub-entry *uid={0},ou=people*. *{0}* is the variable where an entered username will be substituted before executing the ldap search.
- *authoritiesPopulator* - assuming the user is found, it uses the group_search_filter to find groups containing this attribute pointed at the user's DN.
- *ldapAuthenticationProvider* - combines together the bindAuthenticator and the authoritiesPopulator, in order to process a `UsernamePasswordAuthenticationToken`.
- *ldapAuthenticationManager* - just like the other examples, this `AuthenticationManager` iterates over the list of providers, giving them a chance to authenticate the user.

The following shows the same configuration in pure Python, using `PythonConfig`.

```
class LdapContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def context_source(self):
        return DefaultSpringSecurityContext(url="ldap://localhost:53389/dc=springframework,dc=org")

    @Object
    def bind_authenticator(self):
        return BindAuthenticator(self.context_source(), user_dn_patterns="uid={0},ou=people")

    @Object
    def authorities_populator(self):
        return DefaultLdapAuthoritiesPopulator(self.context_source(), group_search_filter="member={0}")

    @Object
    def provider(self):
        return LdapAuthenticationProvider(self.bind_authenticator(), self.authorities_populator())

    @Object
    def manager(self):
        return AuthenticationManager(auth_providers=[self.provider()])
```

To use the password comparison mechanism, substitute `PasswordComparisonAuthenticator` for `BindAuthenticator` as follows:

```
        <object id="context_source" class="springpython.security.providers.Ldap.DefaultSpringSecurityContextSour
            <property name="url" value="ldap://localhost:53389/dc=springframework,dc=org"/>
        </object>

        <object id="passwordAuthenticator" class="springpython.security.providers.Ldap.PasswordComparisonAuthent
            <property name="context_source" ref="context_source"/>
            <property name="user_dn_patterns" value="uid={0},ou=people"/>
        </object>

        <object id="authoritiesPopulator" class="springpython.security.providers.Ldap.DefaultLdapAuthoritiesPopu
```

```
            <property name="context_source" ref="context_source"/>
            <property name="group_search_filter" value="member={0}"/>
        </object>

        <object id="ldapAuthenticationProvider" class="springpython.security.providers.Ldap.LdapAuthenticationPr
            <property name="ldap_authenticator" ref="bindAuthenticator"/>
            <property name="ldap_authorities_populator" ref="authoritiesPopulator"/>
        </object>

        <object id="ldapAuthenticationManager" class="springpython.security.providers.AuthenticationManager">
            <property name="auth_providers">
                <list><ref object="ldapAuthenticationProvider"/></list>
            </property>
        </object>
```

The following block shows the same configuration using the pure Python container:

```
class LdapContext(PythonConfig):
    def __init__(self):
        PythonConfig.__init__(self)

    @Object
    def context_source(self):
        return DefaultSpringSecurityContext(url="ldap://localhost:53389/dc=springframework,dc=org")

    @Object
    def password_authenticator(self):
        return PasswordComparisonAuthenticator(self.context_source(), user_dn_patterns="uid={0},ou=people")

    @Object
    def authorities_populator(self):
        return DefaultLdapAuthoritiesPopulator(self.context_source(), group_search_filter="member={0}")

    @Object
    def provider(self):
        return LdapAuthenticationProvider(self.password_authenticator(), self.authorities_populator())

    @Object
    def manager(self):
        return AuthenticationManager(auth_providers=[self.provider()])
```

By default, `PasswordComparisonAuthenticator` handles SHA encrypted passwords as well passwords stored in plain text. However, you can inject a custom `PasswordEncoder` to support alternative password encoding schemes.

6.3.1.3. Future AuthenticationProviders

So far, Spring Python has implemented a `DaoAuthenticationProvider` than can link with any database or use an in-memory user data structure, as well as an `LdapAuthenticationProvider`. Future releases should include:

- `OpenIDAuthenticationProvider`
- Anonymous authentication provider - allows you to tag anonymous users, and constrain what they can access, even if they don't provide a password

6.3.2. AuthenticationManager

An AuthenticationManager holds a list of one or more AuthenticationProvider's, and will go through the list when attempting to authenticate. PetClinic configures it like this:

```
class SampleContainer(PythonConfig):
    ...
    @Object
    def authenticationManager(self):
        return AuthenticationManager(auth_providers = [self.authenticationProvider()])
```

XML-based configuration with `XMLConfig`:

```

<object id="authenticationManager" class="springpython.security.providers.AuthenticationManager">
    <property name="auth_providers">
            <list><ref object="authenticationProvider"/></list>
           </property>
</object>
```

This `AuthenticationManager` has a list referencing one object already defined in the `ApplicationContext`, authenticationProvider. The authentication manager is supplied as an argument to the security interceptor, so it can perform checks as needed.

6.4. Authorization

After successful authentication, a user is granted various roles. The next step of security is to determine if that user is authorized to conduct a given operation or access a particular web page. The `AccessDecisionManager` is called by the `AbstractSecurityInterceptor` and is responsible for making final access control decisions. The `AccessDecisionManager` interface contains two methods:

```

def decide(self, authentication, object, config)
def supports(self, attr)
```

As can be seen from the first method, the `AccessDecisionManager` is passed via method parameters all information that is likely to be of value in assessing an authorization decision. In particular, passing the secure object enables those arguments contained in the actual secure object invocation to be inspected. For example, let's assume the secure object was a `MethodInvocation`. It would be easy to query the `MethodInvocation` for any Customer argument, and then implement some sort of security logic in the `AccessDecisionManager` to ensure the principal is permitted to operate on that customer. Implementations are expected to throw an `AccessDeniedException` if access is denied.

Whilst users can implement their own `AccessDecisionManager` to control all aspects of authorization, Spring Python Security includes several `AccessDecisionManager` implementations that are based on voting. Using this approach, a series of `AccessDecisionVoter` implementations are polled on an authorization decision. The `AccessDecisionManager` then decides whether or not to throw an `AccessDeniedException` based on its assessment of the votes.

The `AccessDecisionVoter` interface has two methods:

```

def supports(self, attr)
def vote(self, authentication, object, config)
```

Concrete implementations return an integer, with possible values being reflected in the `AccessDecisionVoter` static fields *ACCESS_ABSTAIN*, *ACCESS_DENIED* and *ACCESS_GRANTED*. A voting implementation will return ACCESS_ABSTAIN if it has no opinion on an authorization decision. If it does have an opinion, it must return either ACCESS_DENIED or ACCESS_GRANTED.

There are three concrete `AccessDecisionManager`'s provided with Spring Python Security that tally the votes. The `ConsensusBased` implementation will grant or deny access based on the consensus of non-abstain votes. Properties are provided to control behavior in the event of an equality of votes or if all votes are abstain. The `AffirmativeBased` implementation will grant access if one or more ACCESS_GRANTED votes were received (ie a deny vote will be ignored, provided there was at least one grant vote). Like the `ConsensusBased` implementation, there is a parameter that controls the behavior if all voters abstain. The `UnanimousBased` provider expects unanimous ACCESS_GRANTED votes in order to grant access, ignoring abstains. It will deny access if there is any ACCESS_DENIED vote. Like the other implementations, there is a parameter that controls the behavior if all voters abstain.

It is possible to implement a custom `AccessDecisionManager` that tallies votes differently. For example, votes from a particular `AccessDecisionVoter` might receive additional weighting, whilst a deny vote from a particular voter may have a veto effect.

There are two concrete `AccessDecisionVoter` implementations provided with Spring Python Security. The `RoleVoter` class will vote if any config attribute begins with *ROLE_*. It will vote to grant access if there is a `GrantedAuthority` which returns a String representation exactly equal to one or more config attributes starting with *ROLE_*. If there is no exact match of any config attribute starting with *ROLE_*, the `RoleVoter` will vote to deny access. If no config attribute begins with *ROLE_*, the voter will abstain. `RoleVoter` is case sensitive on comparisons as well as the *ROLE_* prefix.

PetClinic has two `RoleVoter`'s in its configuration:

```
class SampleContainer(PythonConfig):
    ...
    @Object
    def vetRoleVoter(self):
        return RoleVoter(role_prefix="VET")

    @Object
    def customerRoleVoter(self):
        return RoleVoter(role_prefix="CUSTOMER")
```

XML-based configuration with `XMLConfig`:

```
<object id="vetRoleVoter" class="springpython.security.vote.RoleVoter">
        <property name="role_prefix"><value>VET</value></property>
</object>

<object id="customerRoleVoter" class="springpython.security.vote.RoleVoter">
        <property name="role_prefix"><value>CUSTOMER</value></property>
</object>
```

The first one votes on VET authorities, and the second one votes on CUSTOMER authorities.

The other concrete `AccessDecisionVoter` is the `LabelBasedAclVoter`. It can be seen in the test cases. Maybe later it will be incorporated into a demo.

Petclinic has a custom `AccessDecisionVoter`, which votes on whether a user "owns" a record.

```
class SampleContainer(PythonConfig):
    ...
    @Object
    def ownerVoter(self):
        return OwnerVoter(controller = self.controller())
```

XML-based configuration using `XMLConfig`:

```
<object id="ownerVoter" class="controller.OwnerVoter">
        <property name="controller" ref="controller"/>
</object>
```

This class is wired in the PetClinic controller module as part of the sample, which demonstrates how easy it is to plugin your own custom security handler to this module.

PetClinic wires together these `AccessDecisionVoter`'s into an `AccessDecisionManager`:

```
class SampleContainer(PythonConfig):
    ...
    @Object
    def accessDecisionManager(self):
        manager = AffirmativeBased()
        manager.allow_if_all_abstain = False
        manager.access_decision_voters = [self.vetRoleVoter(), self.customerRoleVoter(), self.ownerVoter()]
        return manager
```

XML-based configuration using `XMLConfig`:

```
<object id="accessDecisionManager" class="springpython.security.vote.AffirmativeBased">
        <property name="allow_if_all_abstain"><value>False</value></property>
        <property name="access_decision_voters">
                <list>
                        <ref object="vetRoleVoter"/>
                        <ref object="customerRoleVoter"/>
                        <ref object="ownerVoter"/>
                </list>
        </property>
</object>
```

Chapter 7. Remoting

Coupling Aspect Oriented Programming with different types of Python remoting services makes it easy to convert your local application into a distributed one. Technically, the remoting segment of Spring Python doesn't use AOP. However, it is very similar in the concept that you won't have to modify either your servers or your clients.

Distributed applications have multiple objects. These can be spread across different instances of the Python interpreter on the same machine, as well on different machines on the network. The key factor is that they need to talk to each other. The developer shouldn't have to spend a large effort coding a custom solution. Another common practice in the realm of distributed programming is that fact that programmers often develop standalone. When it comes time to distribute the application to production, the configuration may be very different. Spring Python solves this by making the link between client and server objects a step of configuration not coding.

In the context of this section of documentation, the term *client* refers to a client-application that is trying to access some remote service. The service is referred to as the *server* object. The term remote is subjective. It can either mean a different thread, a different interpretor, or the other side of the world over an Internet connection. As long as both parties agree on the configuration, they all share the same solution.

7.1. External dependencies

Spring Python currently supports and requires the installation of at least one of the libraries:

- Pyro (Python Remote Objects) - a pure Python transport mechanism
- Hessian - support for Hessian has just started. So far, you can call python-to-java based on libraries released from Caucho.

7.2. Remoting with PYRO (Python Remote Objects)

7.2.1. Decoupling a simple service, to setup for remoting

For starters, let's define a simple service.

```
class Service(object):
    def get_data(self, param):
        return "You got remote data => %s" % param
```

Now, we will create it locally and then call it.

```
service = Service()
print service.get_data("Hello")

"You got remote data => Hello"
```

Okay, imagine that you want to relocate this service to another instance of Python, or perhaps another machine on your network. To make this easy, let's utilize Inversion Of Control, and transform this service into a Spring service. First, we need to define an application context. We will create a file called *applicationContext.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

    <object id="service" class="Service"/>

</objects>
```

The client code is changed to this:

```
appContext = ApplicationContext(XMLConfig("applicationContext.xml"))
service = appContext.get_object("service")
print service.get_data("Hello")

"You got remote data => Hello"
```

Not too tough, ehh? Well, guess what. That little step just decoupled the client from directly creating the service. Now we can step in and configure things for remote procedure calls without the client knowing it.

7.2.2. Exporting a Spring Service Using Inversion Of Control

In order to reach our service remotely, we have to export it. Spring Python provides `PyroServiceExporter` to export your service through Pyro. Add this to your application context.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

    <object id="remoteService" class="Service"/>

    <object id="service_exporter" class="springpython.remoting.pyro.PyroServiceExporter">
        <property name="service_name" value="ServiceName"/>
        <property name="service" ref="remoteService"/>
    </object>

    <object id="service" class="springpython.remoting.pyro.PyroProxyFactory">
        <property name="service_url" value="PYROLOC://localhost:7766/ServiceName"/>
    </object>

</objects>
```

Three things have happened:

1. Our original service's object name has been changed to *remoteService*.

2. Another object was introduced called *service_exporter*. It references object *remoteService*, and provides a proxied interface through a Pyro URL.

3. We created a client called *service*. That is the same name our client code it looking for. It won't know the difference!

7.2.2.1. Hostname/Port overrides

Pyro defaults to advertising the service at *localhost:7766*. However, you can easily override that by setting the `service_host` and `service_port` properties of the `PyroServiceExporter` object, either through setter or constructor injection.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

    <object id="remoteService" class="Service"/>

    <object id="service_exporter" class="springpython.remoting.pyro.PyroServiceExporter">
        <property name="service_name" value="ServiceName"/>
        <property name="service" ref="remoteService"/>
        <property name="service_host" value="127.0.0.1"/>
        <property name="service_port" value="7000"/>
    </object>

    <object id="service" class="springpython.remoting.pyro.PyroProxyFactory">
        <property name="service_url" value="PYROLOC://127.0.0.1:7000/ServiceName"/>
    </object>

</objects>
```

In this variation, your service is being hosted on port 7000 instead of the default 7766. This is also key, if you need to advertise to another IP address, to make it visible to another host.

Now when the client runs, it will fetch the `PyroProxyFactory`, which will use Pyro to look up the exported module, and end up calling our remote Spring service. And notice how neither our service nor the client have changed!

Python doesn't need an interface declaration for the client proxy

If you have used Spring Java's remoting client proxy beans, then you may be used to the idiom of specifying the interface of the client proxy. Due to Python's dynamic nature, you don't have to do this.

We can now split up this application into two objects. Running the remote service on another server only requires us to edit the client's application context, changing the URL to get to the service. All without telling the client and server code.

7.2.3. Do I have to use XML?

No. Again, Spring Python provides you the freedom to do things using the IoC container, or programmatically.

To do the same configuration as shown above looks like this:

```
from springpython.remoting.pyro import PyroServiceExporter
from springpython.remoting.pyro import PyroProxyFactory

# Create the service
remoteService = Service()

# Export it via Pyro using Spring Python's utility classes
service_exporter = PyroServiceExporter()
service_exporter.service_name = "ServiceName"
service_exporter.service = remoteService
service_exporter.after_properties_set()

# Get a handle on a client-side proxy that will remotely call the service.
service = PyroProxyFactory()
```

```
service.service_url = "PYROLOC://127.0.0.1:7000/ServiceName"

# Call the service just you did in the original, simplified version.
print service.get_data("Hello")
```

Against, you can override the hostname/port values as well

```

...
# Export it via Pyro using Spring Python's utility classes
service_exporter = PyroServiceExporter()
service_exporter.service_name = "ServiceName"
service_exporter.service = remoteService
service_exporter.service_host = "127.0.0.1" # or perhaps the machines actual hostname
service_exporter.service_port = 7000
service_exporter.after_properties_set()
...
```

That is effectively the same steps that the IoC container executes.

Don't forget after_properties_set!

Since `PyroServiceExporter` is an `InitializingObject`, you must call `after_properties_set` in order for it to start the Pyro thread. Normally the IoC container will do this step for you, but if you choose to create the proxy yourself, you are responsible for this step.

7.2.4. Splitting up the client and the server

This configuration sets us up to run the server and the client in two different Python VMs. All we have to do is split things into two parts.

Copy the following into `server.xml`:

```

<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

    <object id="remoteService" class="server.Service"/>

    <object id="service_exporter" class="springpython.remoting.pyro.PyroServiceExporter">
        <property name="service_name" value="ServiceName"/>
        <property name="service" ref="remoteService"/>
        <property name="service_host" value="127.0.0.1"/>
        <property name="service_port" value="7000"/>
    </object>

</objects>
```

Copy the following into `server.py`:

```

import logging
from springpython.config import XMLConfig
from springpython.context import ApplicationContext

class Service(object):
    def get_data(self, param):
        return "You got remote data => %s" % param

if __name__ == "__main__":
    # Turn on some logging in order to see what is happening behind the scenes...
    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)

    appContext = ApplicationContext(XMLConfig("server.xml"))
```

Copy the following into `client.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

    <object id="service" class="springpython.remoting.pyro.PyroProxyFactory">
        <property name="service_url" value="PYROLOC://127.0.0.1:7000/ServiceName"/>
    </object>

</objects>
```

Copy the following into `client.py`:

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

if __name__ == "__main__":
    # Turn on some logging in order to see what is happening behind the scenes...
    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)

    appContext = ApplicationContext(XMLConfig("client.xml"))
    service = appContext.get_object("service")
    print "CLIENT: %s" % service.get_data("Hello")
```

First, launch the server script, and then launch the client script, both on the same machine. They should be able to talk to each other with no problem at all, producing some log chatter like this:

```
$ python server.py &
[1] 20854

2009-01-08 12:06:20,021 - springpython.container.ObjectContainer - DEBUG - === Scanning configuration <springpyt
2009-01-08 12:06:20,021 - springpython.config.XMLConfig - DEBUG - =============================================
2009-01-08 12:06:20,022 - springpython.config.XMLConfig - DEBUG - * Parsing server.xml
2009-01-08 12:06:20,025 - springpython.config.XMLConfig - DEBUG - =============================================
2009-01-08 12:06:20,025 - springpython.container.ObjectContainer - DEBUG - remoteService object definition does
2009-01-08 12:06:20,026 - springpython.container.ObjectContainer - DEBUG - service_exporter object definition do
2009-01-08 12:06:20,026 - springpython.container.ObjectContainer - DEBUG - === Done reading object definitions.
2009-01-08 12:06:20,026 - springpython.context.ApplicationContext - DEBUG - Eagerly fetching remoteService
2009-01-08 12:06:20,026 - springpython.context.ApplicationContext - DEBUG - Did NOT find object 'remoteService'
2009-01-08 12:06:20,026 - springpython.context.ApplicationContext - DEBUG - Creating an instance of id=remoteSer
2009-01-08 12:06:20,026 - springpython.factory.ReflectiveObjectFactory - DEBUG - Creating an instance of server.
2009-01-08 12:06:20,027 - springpython.context.ApplicationContext - DEBUG - Stored object 'remoteService' in con
2009-01-08 12:06:20,027 - springpython.context.ApplicationContext - DEBUG - Eagerly fetching service_exporter
2009-01-08 12:06:20,027 - springpython.context.ApplicationContext - DEBUG - Did NOT find object 'service_exporte
2009-01-08 12:06:20,027 - springpython.context.ApplicationContext - DEBUG - Creating an instance of id=service_e
2009-01-08 12:06:20,028 - springpython.factory.ReflectiveObjectFactory - DEBUG - Creating an instance of springp
2009-01-08 12:06:20,028 - springpython.context.ApplicationContext - DEBUG - Stored object 'service_exporter' in
```

```
2009-01-08 12:06:20,028 - springpython.remoting.pyro.PyroServiceExporter - DEBUG - Exporting ServiceName as a Py
2009-01-08 12:06:20,029 - springpython.remoting.pyro.PyroDaemonHolder - DEBUG - Registering ServiceName at 127.0
2009-01-08 12:06:20,029 - springpython.remoting.pyro.PyroDaemonHolder - DEBUG - Pyro thread needs to be started
2009-01-08 12:06:20,030 - springpython.remoting.pyro.PyroDaemonHolder._PyroThread - DEBUG - Starting up Pyro ser

$ python client.py

2009-01-08 12:06:26,291 - springpython.container.ObjectContainer - DEBUG - === Scanning configuration <springpyt
2009-01-08 12:06:26,292 - springpython.config.XMLConfig - DEBUG - =============================================
2009-01-08 12:06:26,292 - springpython.config.XMLConfig - DEBUG - * Parsing client.xml
2009-01-08 12:06:26,294 - springpython.config.XMLConfig - DEBUG - =============================================
2009-01-08 12:06:26,294 - springpython.container.ObjectContainer - DEBUG - service object definition does not ex
2009-01-08 12:06:26,294 - springpython.container.ObjectContainer - DEBUG - === Done reading object definitions.
2009-01-08 12:06:26,295 - springpython.context.ApplicationContext - DEBUG - Eagerly fetching service
2009-01-08 12:06:26,295 - springpython.context.ApplicationContext - DEBUG - Did NOT find object 'service' in the
2009-01-08 12:06:26,295 - springpython.context.ApplicationContext - DEBUG - Creating an instance of id=service p
2009-01-08 12:06:26,295 - springpython.factory.ReflectiveObjectFactory - DEBUG - Creating an instance of springp
2009-01-08 12:06:26,295 - springpython.context.ApplicationContext - DEBUG - Stored object 'service' in container

CLIENT: You got remote data => Hello
```

This shows one instance of Python running the client, connecting to the instance of Python hosting the server module. After that, moving these scripts to other machines only requires changing the hostname in the XML files.

7.3. Remoting with Hessian

Caucho's python library for Hessian is incomplete

Due to minimal functionality provided by Caucho's Hessian library for python, there is minimal documentation to show its functionality.

The following shows how to connect a client to a Hessian-exported service. This can theoretically be any technology. Currently, Java objects are converted into python dictionaries, meaning that the data and transferred, but there are not method calls available.

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.org/springpython/schema/objects/1.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/springpython/schema/objects/1.1
                       http://springpython.webfactional.com/schema/context/spring-python-context-1.1.xsd">

        <object id="personService" class="springpython.remoting.hessian.HessianProxyFactory">
                <property name="service_url"><value>http://localhost:8080/</value></property>
        </object>

</objects>
```

The Caucho library appears to only support Python being a client, and not yet as a service, so there is no `HessianServiceExporter` available yet.

7.4. High-Availability/Clustering Solutions

This props you up for many options to increase availability. It is possible to run a copy of the server on multiple machines. You could then institute some type of round-robin router to go to different URLs. You could easily run ten copies of the remote service.

```
pool = []
for i in range(10):
    service_exporter = PyroServiceExporter(service_name = "ServiceName%s" % i, service = Service())
    pool.append(service_exporter)
```

(Yeah, I know, you can probably do this in one line with a list comprehension).

Now you have ten copies of the server running, each under a distinct name.

For any client, your configuration is a slight tweak.

```

services = []
for i in range(10):
    services.append(PyroProxyFactory(service_url = "PYROLOC://localhost:7766/ServiceName%s" % i))
```

Now you have an array of possible services to reach, easily spread between different machines. With a little client-side utility class, we can implement a round-robin solution.

```

class HighAvailabilityService(object):
    def __init__(self, service_pool):
        self.service_pool = service_pool
        self.index = 0
    def get_data(self, param):
        self.index = (self.index+1) % len(self.service_pool)
        try:
            return self.service_pool[self.index].get_data(param)
        except:
            del(self.service_pool[i])
            return self.get_data(param)

service = HighAvailabilityService(service_pool = services)
service.get_data("Hello")
service.get_data("World")
```

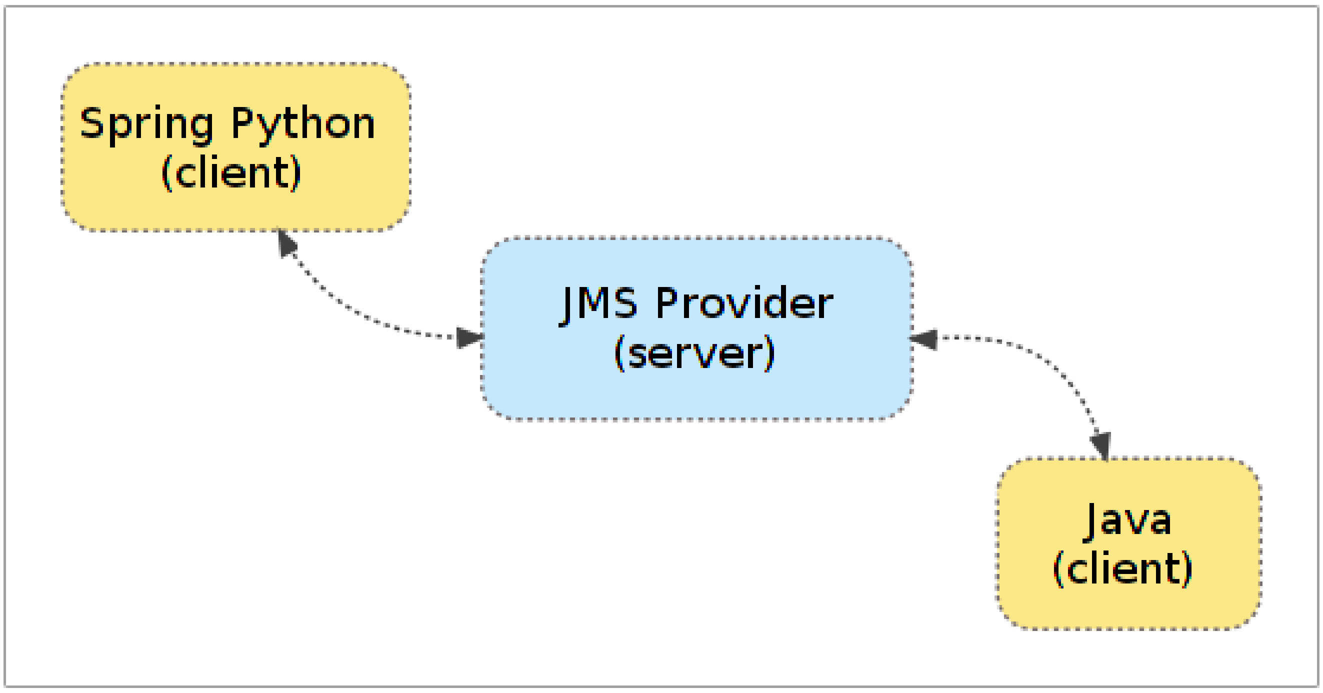
Notice how each call to the `HighAvailabilityService` class causes the internal index to increment and roll over. If a service doesn't appear to be reachable, it is deleted from the list and attempted again. A little more sophisticated error handling should be added in case there are no services available. And there needs to be a way to grow the services. But this gets us off to a good start.

Chapter 8. JMS Messaging

Java Message Service has been a well known means for decoupling the Java application's parts or to provide an integration service for otherwise disconnected Java applications. Thanks to JMS being purely an API, Spring Python offers a way for connecting to JMS providers and to participate in JMS messaging scenarios. JMS messages sent and received by a Spring Python powered application are no different than messages produced and consumed by Java applications, in fact, *you can use Spring Python and JMS with no Java applications participating in message exchanging at all*.

Spring Python works as a JMS client, you still need a JMS provider, the server part, for message brokering. The only JMS provider currently supported by Spring Python is IBM's WebSphere MQ, formerly known as MQSeries.

Although Spring Python's JMS API is loosely based on Spring Java's, it's not a direct port and features a highly Pythonic look and feel.



8.1. Introduction

Througout the chapter pure Python code or YAML syntax is used to illustrate the support for JMS however it only represents the author's preferences and it's worth noting that you can use any of Spring Python's formats to configure the IoC container. Or you can use no IoC at all as it's a completely optional feature and one that's not strictly required by JMS.

JMS messaging with Spring Python revolves around the idea of using a connection factory for obtaining a connection to a JMS provider and `springpython.jms.core.JmsTemplate` as a means for sending and receiving messages. A JmsTemplate instance is tied to a connection factory however a single connection factory may be safely reused across multiple JmsTemplates.

In addition to that, `springpython.jms.listener.SimpleMessageListenerContainer` allows for a purely configuration-driven way to set up background JMS listeners to receive messages from JMS providers.

8.2. Dependencies

Support for JMS messaging with WebSphere MQ is built on top of the CPython-only PyMQI library which provides Python applications an access to WebSphere MQ queue managers. You need to separately install PyMQI in order to use `springpython.jms.factory.WebSphereMQConnectionFactory`. PyMQI, in turn, needs a *WebSphere MQ client*, a runtime library which may be freely downloaded from IBM's site.

`SimpleMessageListenerContainer`, a Spring Python component which helps with running background JMS listeners, requires the installation of Circuits 1.2+ and threadpool 1.2.7 or newer.

8.3. Quick start

Here's a few quick examples that will get you started with Spring Python and JMS. Both Python code and IoC with YAML syntax are shown. It's assumed there's a QM.1 queue manager running on host 192.168.1.121 with its listener on port 1434 and connections are made through the server connection channel SVRCONN.1 to queues TEST.1 and TEST.2.

8.3.1. Sending

First, let's send a message using nothing but pure Python code.

```
from springpython.jms.core import JmsTemplate
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN1.1"
host = "192.168.1.121"
listener_port = "1434"
queue1 = "TEST.1"

# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# Every JmsTemplate uses a connection factory for actually communicating with a JMS provider.
jms_template = JmsTemplate(factory)

# And that's it, now we put the mandatory "Hello world" message on a queue.
jms_template.send("Hello world", queue1)

# We're not using an IoC so we must shut down the connection factory ourselves.
factory.destroy()
```

Now do the same but use an IoC container configured via `springpython.config.YamlConfig`. The configuration should be saved in a "jms-context.yml" file in the same directory the Python code using it will be saved in.

```
objects:
    - object: MyConnectionFactory
      class: springpython.jms.factory.WebSphereMQConnectionFactory
      properties:
          queue_manager: QM.1
          channel: SVRCONN.1
          host: 192.168.1.121
          listener_port: "1434"

    - object: MyTemplate
      class: springpython.jms.core.JmsTemplate
      properties:
          factory: {ref: MyConnectionFactory}

    - object: MyQueue
      str: TEST.1
```

And the Python code using the above IoC configuration.

```

from springpython.context import ApplicationContext
from springpython.config import YamlConfig

container = ApplicationContext(YamlConfig("./jms-context.yml"))

# Read the objects definitions from configuration.
queue1 = container.get_object("MyQueue")
jms_template = container.get_object("MyTemplate")

# Send the message.
jms_template.send("Hello world", queue1)

# The connection factory is now being managed by the IoC container which takes
# care of shutting down the factory. No need for manually destroying it.
```

An obvious change is that the configuration is now kept separately from the implementation but another advantage is that the container will shut down the connection factory on itself as `springpython.jms.factory.WebSphereMQConnectionFactory` is a subclass of `springpython.context.DisposableObject` which means its .destroy method will be executed when the container will be shutting down.

8.3.2. Receiving

The very same connection factory and JmsTemplate can be used for both sending and receiving. Examples below use the same definitions of objects as the sending examples do, they are repeated here for the sake of completness.

```

from springpython.jms.core import JmsTemplate
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"
queue1 = "TEST.1"

# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# Every JmsTemplate uses a connection factory for actually communicating with a JMS provider.
jms_template = JmsTemplate(factory)

# Get a message off the queue. The call to receive will by default time out
# after 1000ms and raise springpython.jms.NoMessageAvailableException then.
jms_template.receive(queue1)

# We're not using an IoC so we need to shut down the connection factory ourselves.
factory.destroy()
```

And here's a complementary example showing the usage of `YamlConfig`. The configuration should be saved in a "jms-context.yml" file in the same directory the Python code using it will be saved in. Note that it's the same configuration that was used in the sending example.

```

```

```
objects:
    - object: MyConnectionFactory
      class: springpython.jms.factory.WebSphereMQConnectionFactory
      properties:
          queue_manager: QM.1
          channel: SVRCONN.1
          host: 192.168.1.121
          listener_port: "1434"

    - object: MyTemplate
      class: springpython.jms.core.JmsTemplate
      properties:
          factory: {ref: MyConnectionFactory}

    - object: MyQueue
      str: TEST.1
```

The Python code used for receiving a message from a queue configured using the `YamlConfig`.

```

from springpython.context import ApplicationContext
from springpython.config import YamlConfig

container = ApplicationContext(YamlConfig("./jms-context.yml"))

# Read the objects definitions from configuration
queue1 = container.get_object("MyQueue")
jms_template = container.get_object("MyTemplate")

# Get a message off the queue. The call to receive will by default time out
# after 1000ms and raise springpython.jms.NoMessageAvailableException then.
jms_template.receive(queue1)

# The connection factory is now being managed by the IoC container which takes
# care of shutting down the factory. No need for manually destroying it.
```

Here's a sample YAML context utilizing the SimpleMessageListenerContainer component and an accompanying Python code using it. As you can see, a mere fact of providing the configuration allows for receiving the messages.

```

objects:
    - object: connection_factory
      class: springpython.jms.factory.WebSphereMQConnectionFactory
      properties:
          queue_manager: QM.1
          channel: SVRCONN.1
          host: 192.168.1.121
          listener_port: "1434"

    - object: message_handler
      class: app.MyMessageHandler

    - object: listener_container
      class: springpython.jms.listener.SimpleMessageListenerContainer
      properties:
          factory: {ref: connection_factory}
          handler: {ref: message_handler}
          destination: TEST.1
```

```


# app.py

from springpython.config import YamlConfig
from springpython.context import ApplicationContext

class MyMessageHandler(object):
    def handle(self, message):
        print "Got message!", message

```

```
if __name__ == "__main__":

    # Obtaining a context will automatically start the SimpleMessageListenerContainer and its listeners in backg
    container = ApplicationContext(YamlConfig("./context.yml"))

    while True:
        # Here goes the application's logic. Any JMS messages, as configured
        # in ./context.yml, will be passed in to a singleton MyMessageHandler instance.
        pass
```

8.4. Connection factories

8.4.1. springpython.jms.factory.WebSphereMQConnectionFactory

`WebSphereMQConnectionFactory` implements access to WebSphere MQ JMS provider. Along with `JmsTemplate` it's the class you'll be most frequently using for sending and receiving of messages.

Each `WebSphereMQConnectionFactory` object will hold at most one connection to WebSphere MQ, which will be lazily established when it'll be actually needed, e.g. when a message will need to be put on a queue for the first time. The connection will always be started in WebSphere MQ's client mode, there's no support for connecting in the bindings mode.

Like all Spring Python's classes `WebSphereMQConnectionFactory` can be configured using pure Python or you can use Spring Python's IoC to separate your business code from configuration. Using IoC has an added benefit of taking care of destroying any open queues and closing the connection when the IoC shuts down - we'll get to it in a moment.

`WebSphereMQConnectionFactory` provides several options that let you customize its behaviour and apart from the obvious ones which you must provide (like, the queue manager's host) all other options have sensible defaults which you'll rarely need to change, if at all.

Here's a full initializer method reproduced for convenience and the explanation of default values used:

```

    def __init__(self, queue_manager=None, channel=None, host=None, listener_port=None,
            cache_open_send_queues=True, cache_open_receive_queues=True,
            use_shared_connections=True, local_queue_template="SYSTEM.DEFAULT.MODEL.QUEUE"):
```

Table 8.1. `springpython.jms.factory.WebSphereMQConnectionFactory` customizable options

queue_manager	default: None *Must be set manually* Name of the queue manager, e.g. EAI.QM.1
channel	default: None *Must be set manually* Name of a server connection (SVRCONN) channel through which the connection will be established, e.g. EAI.SVRCONN.1

host	default: None *Must be set manually* Host name or IP on which the queue manager is running, e.g. 192.168.1.103
listener_port	default: None *Must be set manually* Port on which the queue manager's listener is accepting TCP connections, e.g. 1434
cache_open_send_queues	default: True By default, `WebSphereMQConnectionFactory` will keep references to open queues in a cache for later re-use. This speeds-up most operations as there's usually no need for closing a queue if it's going to be used in subsequent calls to queue manager. At times however, it's prefered to close the queues as soon as possible and *cache_open_send_queues* controls whether queues open for putting the messages on are to be kept in the cache.
cache_open_receive_queues	default: True This setting controls whether queues open for receving of messages should be kept in a cache. If set to *False*, they will be closed after the call to get a message off the queue will have finished.
use_shared_connections	default: True A single `WebSphereMQConnectionFactory` object may be shared between multiple threads to provide better performance. This setting allows for marking the underlying connection to a queue manager as a non-shareable and makes sure that only one thread will be able to use it, any call to the factory from a thread that didn't open the connection will result in a `springpython.jms.JMSException` being raised. The setting should only set to False when connecting to queue managers running on z/OS systems as it otherwise can hurt the performance of multi-threaded applications. It has no impact on performance of single-threaded applications.

dynamic_queue_template	default: SYSTEM.DEFAULT.MODEL.QUEUE The name of a model queue basing on which the dynamic queues will be created. It is usually desirable to override the default value as, unless customized, SYSTEM.DEFAULT.MODEL.QUEUE is a non-shared (NOSHARE in MQ speak) queue and doesn't allow for opening the dynamic queues for both sending and receiving.
ssl	default: False A boolean value which indicates whether connections to the queue manager should use a client SSL/TLS certificate. *ssl_cipher_spec* and *ssl_key_repository* must also be provided if *ssl* is True.
ssl_cipher_spec	default: None An SSL/TLS cipher spec to use for encrypted connections, its value must be equal to that of the MQ SVRCONN channel's SSLCIPH attribute.
ssl_key_repository	default: None On-disk location of an SSL/TLS client certificates repository. The repository must be of type CMS, such a repository can be created using gsk6cmd/gsk7cmd command line tools. Note that the value of this attribute must not contain a suffix; for instance, if there are following files in /var/mqm/security: client-repo.crl, client-repo.kdb, client-repo.rdb and client-repo.sth, then ssl_key_repository must be set to "/var/mqm/security/client-repo".

Here's an example of programatically creating a `WebSphereMQConnectionFactory` object:

```
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"

factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# ... use factory here.

# Always destroy the factory when not using an IoC container.
factory.destroy()
```

An example of using YamlConfig for configuring `WebSphereMQConnectionFactory` inside of an IoC container.

```
objects:
    - object: MyConnectionFactory
      class: springpython.jms.factory.WebSphereMQConnectionFactory
      properties:
          queue_manager: QM.1
          channel: SVRCONN.1
          host: 192.168.1.121
          listener_port: "1434"
```

All cached queues will not be closed by a factory until after its .destroy will have been called which will happen automatically if you're using an IoC container. If the factory is configured programatically in Python you must call .destroy yourself in your code. A call to .destroy also closes the factory's connection to a queue manager.

`WebSphereMQConnectionFactory` objects are thread-safe and may be shared between multiple threads if the queue manager supports sharing a single connection which is the case on all platforms except for z/OS.

For the curious one

`springpython.jms.factory.WebSphereMQConnectionFactory` and `springpython.jms.factory.MQRFH2JMS` wrap the WebSphere MQ's native MQRFH2 wire-level format in a set of Python classes and hide any intricate details of communicating with queue managers. From the programmer's viewpoint, `MQRFH2JMS` is irrelevant, however it might be of interest to anyone willing to improve or expand Spring Python's JMS support.

8.5. springpython.jms.core.JmsTemplate

JmsTemplate is the class to use for sending JMS messages; along with SimpleMessageListenerContainer it may also be used in order to receive them. A template must be associated with a connection factory and once configured, may be used for communicating in both directions. It's up to you to decide whether in your circumstances it makes sense to reuse a single template for all communications, to have a single template for each queue involved or perhaps to use separate, dedicated, templates, one for sending and one for receiving. Note however, *that `JmsTemplate` instances are not guaranteed to be thread-safe* and no attempt has been made to make them be so.

Remember that factories postpone connecting to a queue manager and creating a JmsTemplate instance doesn't necessarily mean there will be no connection errors when it will be first time used for sending or receiving.

Here's how a `JmsTemplate` may be instantiated using Python code:

```
from springpython.jms.core import JmsTemplate
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN1.1"
host = "192.168.1.121"
listener_port = "1434"

factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)
jms_template = JmsTemplate(factory)

# Always destroy the factory when not using IoC
factory.destroy()
```

An example of using YamlConfig to configure a JmsTemplate

```
objects:
    - object: MyConnectionFactory
      class: springpython.jms.factory.WebSphereMQConnectionFactory
      properties:
          queue_manager: QM.1
          channel: SVRCONN.1
          host: 192.168.1.121
          listener_port: "1434"

    - object: jms_template
      class: springpython.jms.core.JmsTemplate
      properties:
          factory: {MyConnectionFactory}
```

JmsTemplate allows for a number of options to customize its behaviour. The only options required to set manually is the *factory* parameter. Except for factory, all the parameters may be overriden by individual calls to sending or receiving of messages.

```
    def __init__(self, factory=None, delivery_persistent=None,
            priority=None, time_to_live=None, message_converter=None,
            default_destination=None):
```

Table 8.2. `springpython.jms.core.JmsTemplate` customizable options

factory	default: None *Must be set manually* A JMS connection factory associated with this JmsTemplate.
delivery_persistent	default: None Tells whether messages sent to a JMS provider are by default persistent. If not set, the persistency of messages is controlled on a per messages basis (and defaults to a persistent delivery).
priority	default: None Messages sent to the provider may be of different priority, usually on a scale from 1 to 9. The setting controls the default priority of all messages sent by this JmsTemplate, unless overridden by individual messages. A JMS provider will set the default priority if no value is given here or when sending the individual messages.
time_to_live	default: None JMS allows for expiry of messages after a certain time *expressed in milliseconds*. The time to live of a message may be set here and it will be applied to all messages sent or can be set per each message sent. If

	no value is provided here and when sending the message to a destination, the message expiry time is left to the discretion of a JMS provider.
message_converter	default: None It is sometimes desirable to not have to deal with raw messages taken from or sent to JMS provider from within a JmsTemplate object, it may make more sense to delegate converting the objects from and to JMS representation to an external helper class. A message converter is an object that helps decoupling the domain objects from the fact that JMS is the transportation layer used for communicating. A single converter may be used for converting the incoming as well as outgoing messages. See the section on message converters for more details and code examples. Setting the message converter here will take precedence over setting it on a per-message basis.
default_destination	default: None If all or most of the messages are sent to or received from the same JMS destination, it's usually useful to configure the destination's name here and have it used in any subsequent calls to a JMS provider. Note that *the value given is used as a default destination for both sending and receiving of messages*.

8.5.1. Sending

The basic approach is to send ASCII strings or unicode objects, which must allow for encoding into UTF-8.

```
# -*- coding: utf-8 -*-

from springpython.jms.core import JmsTemplate
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"
queue1 = "TEST.1"

# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# Every JmsTemplate uses a connection factory for actually communicating with a JMS provider.
jms_template = JmsTemplate(factory)
jms_template.default_destination = queue1

# Send some ASCII
jms_template.send("Hi, Spring Python here")

# Send unicode
jms_template.send(u"Cze##, z tej strony Spring Python")
```

```
# We're not using an IoC so we need to shut down the connection factory ourselves.
factory.destroy()
```

Note that in an example above the message's destination has been taken from JmsTemplate. We can also specify it on send time or we can combine both approaches, like here:

```

# -*- coding: utf-8 -*-

from springpython.jms.core import JmsTemplate
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"
queue1 = "TEST.1"
queue2 = "TEST.2"

# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# Every JmsTemplate uses a connection factory for actually communicating with a JMS provider.
jms_template = JmsTemplate(factory)
jms_template.default_destination = queue1

# Send some ASCII to one queue
jms_template.send("Hi, Spring Python here")

# Send unicode to another queue
jms_template.send(u"Cze##, z tej strony Spring Python", queue2)

# We're not using an IoC so we need to shut down the connection factory ourselves.
factory.destroy()
```

Sending is not limited to strings or unicode objects though. You can customize a lot of message's properties by sending a springpython.jms.core.TextMessage instead. The following example shows how a custom message ID and reply to destination can be specified for an outgoing message.

```

# stdlib
from uuid import uuid4

# Spring Python
from springpython.jms.core import JmsTemplate, TextMessage
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"
queue1 = "TEST.1"

# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# Every JmsTemplate uses a connection factory for actually communicating with a JMS provider.
jms_template = JmsTemplate(factory)
jms_template.default_destination = queue1

# Generate the correlation ID
jms_correlation_id = uuid4().hex

message = TextMessage("Hi, Spring Python here")
message.jms_correlation_id = jms_correlation_id
message.jms_reply_to = "REPLY.TO.QUEUE"

# Send the message
jms_template.send(message)

# We're not using an IoC so we need to shut down the connection factory ourselves.
factory.destroy()
```

Using TextMessage instances instead of plain strings or unicode objects is also recommended when you're interested in values a JMS provider has given to JMS properties of a message after the message had been sent. Here you can see the values which were assigned automatically by the provider to jms_timestamp and jms_message_id properties.

```
from springpython.jms.core import JmsTemplate, TextMessage
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"
queue1 = "TEST.1"

# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# Every JmsTemplate uses a connection factory for actually communicating with a JMS provider.
jms_template = JmsTemplate(factory)
jms_template.default_destination = queue1

# Create a TextMessage instance.
message = TextMessage("Hi, Spring Python here")

# Send the message
jms_template.send(message)

print "jms_timestamp = %s" % message.jms_timestamp
print "jms_message_id = %s" % message.jms_message_id

# We're not using an IoC so we need to shut down the connection factory ourselves.
factory.destroy()

#
# Shows the following here:
#
# $ python jms_properties_overriding.py
# jms_timestamp = 1255885622380
# jms_message_id = ID:414d5120514d2e312020202020202020283cdb4a02220020
# $
```

Take a look here for more information about how to use TextMessages.

8.5.2. Receiving

The same JmsTemplate instance may be used for both sending and receiving of messages. When you receive messages you may optionally provide a timeout value in milliseconds after exceeding which a `springpython.jms.NoMessageAvailableException` will be raised if no message will have been available for a given JMS destination. Default timeout is 1000 milliseconds.

JmsTemplate may use a default JMS destination for each call to .receive or you can explicitly specify the destination's name when you receive messages.

```
from springpython.jms.core import JmsTemplate, TextMessage
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"
queue1 = "TEST.1"
queue2 = "TEST.2"
```

```
# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# Every JmsTemplate uses a connection factory for actually communicating with a JMS provider.
jms_template = JmsTemplate(factory)
jms_template.default_destination = queue1

# Send a message to the first queue which is a default destination ..
jms_template.send("Hi there!")

# .. and now receive it.
print jms_template.receive()

# Now send a message to the second one ..
jms_template.send("Hi there again!", queue2)

# .. and now receive it ..
print jms_template.receive(queue2)

# .. try to receive a message again, this time requesting a timeout of 2 seconds.
print jms_template.receive(queue2, 2000)

# We're not using an IoC so we need to shut down the connection factory ourselves.
factory.destroy()
```

Note that SimpleMessageListenerContainer provides a complementary way for receiving the messages, particularly well suited for long-running processes, such as servers.

8.5.3. Dynamic queues

A dynamic queue is a usually short-lived object created on-demand by JMS applications, most often found in request-reply scenarios when there's no need for the response to be persistently stored. An application initiating the communication will create a dynamic temporary queue, send the request to the other side providing the name of the dynamic queue as a destination for the responses to be sent to and wait for a certain amount of time. *With Spring Python and WebSphere MQ, the requesting side must then explicitly close the dynamic queue* regardless of whether the response will be received or if the request timeouts.

The following example shows two JmsTemplate objects communicating via a dynamic queue and imitating an exchange of messages between two dispersed applications. You can observe than from the responding application's point of view a dynamic queue's name is like any other queue's name, the application doesn't need to be - and indeed isn't - aware that it's responding to a dynamic queue and not to a predefined one. For the requesting end a dynamic queue is also like a regular queue in that its name must be provided to the JmsTemplate's .receive method. Note that WebSphere MQ allows only non-persistent messages to be put on *temporary* dynamic queues which are the kind of dynamic queues you get by default with Spring Python.

```

from springpython.jms import DELIVERY_MODE_NON_PERSISTENT
from springpython.jms.core import JmsTemplate, TextMessage
from springpython.jms.factory import WebSphereMQConnectionFactory

qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"

exchange_queue = "TEST.1"

# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

requesting_side = JmsTemplate(factory)
requesting_side.default_destination = exchange_queue

responding_side = JmsTemplate(factory)
responding_side.default_destination = exchange_queue

# Create a dynamic queue.
```

```
dyn_queue_name = requesting_side.open_dynamic_queue()

# Note that we wrap the whole conversation in a try/finally block as we must
# always close a WebSphere MQ dynamic queue.

try:
    # Create a request message.
    message = TextMessage("Hey, what's up on the other side?")

    # WebSphere MQ messages sent to dynamic temporary queues must not
    # be persistent.
    message.jms_delivery_mode = DELIVERY_MODE_NON_PERSISTENT

    # Tell the other side where to send responses.
    message.jms_reply_to = dyn_queue_name

    # Send the request
    requesting_side.send(message)

    # Receive the request ..
    request = responding_side.receive()

    # .. prepare the response ..
    response = TextMessage("A bit stormy today!")
    response.jms_delivery_mode = DELIVERY_MODE_NON_PERSISTENT

    # .. and send our response to a jms_reply_to destination which as we know
    # is a dynamic queue in this example.
    responding_side.send(response, request.jms_reply_to)

    # Receive the response. It's being read as usual, as from any other queue,
    # there's no special JmsTemplate's method for getting messages
    # off dynamic queues.
    print requesting_side.receive(dyn_queue_name)

finally:
    requesting_side.close_dynamic_queue(dyn_queue_name)

# We're not using an IoC so we need to shut down the connection factory ourselves.
factory.destroy()
```

It's worth mentioning again that you must close WebSphere MQ dynamic queues yourself as Spring Python won't do that for you - it's a slight deviation from how Java JMS works.

8.5.4. Message converters

It's quite possible that you'll like to separate the code responsible for core JMS communication with outside systems from the logic needed for converting your business domain's objects back and forth to strings needed for passing into JmsTemplate's methods. You may utilize your own converting classes for it or you can use the Spring Python's converters for such a work. A converter is a subclass of `springpython.jms.core.MessageConverter` which must implement at least one of the *to_message* or *from_message* methods. There's nothing magical about MessageConverter objects and they won't do any automatic convertions for you, they're just interfaces you can implement as you'll likely need some sort of separation between the objects you deal with and the JMS API.

There's one difference you must take into account when using message converters - you don't use the standard send and receive methods but dedicated *convert_and_send* and *receive_and_convert* ones. Other than that, the JMS API and features are exactly the same.

The code below shows a sample usage of MessageConverters. Note that you don't need to implement both *to_message* and *from_message* if that's not appropriate in your situation however it makes sense for the example below to handle requests and responses using only one converter object.

```

from springpython.jms.factory import WebSphereMQConnectionFactory
from springpython.jms.core import JmsTemplate, MessageConverter, TextMessage
```

```
qm_name = "QM.1"
channel = "SVRCONN.1"
host = "192.168.1.121"
listener_port = "1434"

# Note that it's the same queue so we're going to later receive the same invoice we sent.
request_queue = response_queue = "TEST.1"

# One of the business domain's objects our application deals with.
class Invoice(object):
    def __init__(self, customer_account_id=None, month=None, amount=None):
        self.customer_account_id = customer_account_id
        self.month = month
        self.amount = amount

    def __str__(self):
        return "<%s at %s, customer_account_id=%s, month=%s, amount=%s>" % (
            self.__class__.__name__, hex(id(self)), self.customer_account_id,
            self.month, self.amount)

# Let's imagine the other side of a JMS link wants to receive and send CSV data.
class InvoiceConverter(MessageConverter):

    def to_message(self, invoice):
        """ Converts a business object to CSV.
        """
        text = ";".join((invoice.customer_account_id, invoice.month, invoice.amount))

        return TextMessage(text)

    def from_message(self, message):
        """ Produces a business object out of CSV data.
        """

        customer_account_id, month, amount = message.text.split(";")

        invoice = Invoice()
        invoice.customer_account_id = customer_account_id
        invoice.month = month
        invoice.amount = amount

        return invoice

# The connection factory we're going to use.
factory = WebSphereMQConnectionFactory(qm_name, channel, host, listener_port)

# Our JmsTemplate.
jms_template = JmsTemplate(factory)

# Here we tell the template to use our converter.
invoice_converter = InvoiceConverter()
jms_template.message_converter = invoice_converter

# See how we're now dealing only with business objects at the JmsTemplate level.

invoice = Invoice("00033010118", "200909", "136.32")
jms_template.convert_and_send(invoice, request_queue)

print jms_template.receive_and_convert(response_queue)

# We're not using an IoC so we need to shut down the connection factory ourselves.
factory.destroy()
```

8.6 springpython.jms.listener.SimpleMessageListenerContainer and background JMS listeners

SimpleMessageListenerContainer is a configuration-driven component which is used to receive messages from JMS destinations. Once configured, the container starts as many background listeners as requested and each listener gets assigned a pool of threads to handle the incoming requests. The number of listeners started and threads in a pool is fixed upon the configuration is read and the container is started, they cannot be dynamically altered in runtime.

The advantage of using SimpleMessageListenerContainer comes from the fact that all you need to do in order to receive the messages is to create your own handler class and to configure the container, no JMS coding is required so you're focusing on creating the business logic, not on the JMS boilerplate.

Table 8.3. `SimpleMessageListenerContainer` properties

factory	A reference to a JMS connection factory; defaults to None and must be set manually.
destination	Name of a JMS destination to read the messages off. Defaults to None and must be set manually.
handler	A reference to an object which will be receiving messages read from the JMS destination. A handler must implement *handle(self, message)* method, of which the *message* argument is a TextMessage instance. There is a convenience class, `springpython.jms.listener.MessageHandler`, which exposes such a method. The exact number of handlers available for message processing is controlled via the *handlers_per_listener* property. The *handler* parameter defaults to None and must be set manually.
concurrent_listeners	Sets a number of background processes that connect to a JMS provider and read messages off the destination. Default value is 1.
handlers_per_listener	Each concurrent listener is assigned a thread pool of a fixed size, given by the *handlers_per_listener* parameter. Upon receiving a message, it will be dispatched to a thread which will in turn invoke the message handler's *handle* method. The pool's size defaults to 2.
wait_interval	A value in milliseconds expressing how often each of the listeners will check for the arrival of a new message. Defaults to 1000 (1 second).

Here's an example showing SimpleMessageListenerContainer in action together with YamlConfig's abstract objects definitions. customer_queue, credit_account_queue and deposit_account_queue subclass the listener_container object which holds the information common to all definitions of JMS destinations. 4 listeners will be assigned to each of the JMS destination, every listener will be assigned a pool of 5 threads for handling the messages read; a wait interval of 700 milliseconds has been set.

```
objects:
    - object: connection_factory
      class: springpython.jms.factory.WebSphereMQConnectionFactory
      properties:
          queue_manager: QM.1
          channel: SVRCONN.1
```

```
            host: 192.168.1.121
            listener_port: "1434"

    - object: message_handler
      class: app.MyMessageHandler

    - object: listener_container
      abstract: True
      class: springpython.jms.listener.SimpleMessageListenerContainer
      concurrent_listeners: "4"
      handlers_per_listener: "5"
      wait_interval: "700"
      properties:
          factory: {ref: connection_factory}
          handler: {ref: message_handler}

    - object: customer_queue
      parent: listener_container
      properties:
          destination: CUST.QUEUE.1

    - object: credit_account_queue
      parent: listener_container
      properties:
          destination: CREDACCT.QUEUE.1

    - object: deposit_account_queue
      parent: listener_container
      properties:
          destination: DEPACCT.QUEUE.1
```

Here's a Python code using the above IoC configuration. Note that the fact of reading a configuration alone suffices for JMS listeners to be started and run in the background of the main application.

```

# app.py

from springpython.config import YamlConfig
from springpython.context import ApplicationContext

class MyMessageHandler(object):
    def handle(self, message):
        print "Got message!", message

if __name__ == "__main__":

    # Obtaining a context will automatically start the SimpleMessageListenerContainer
    # and its listeners in background.
    container = ApplicationContext(YamlConfig("./context.yml"))

    while True:
        # Here goes the main application's logic, which does nothing in this case.
        # However, the listeners have been already started and incoming messages
        # will be passed in to MyMessageHandler instance (as configured in YamlConfig).
        pass
```

8.7. springpython.jms.core.TextMessage

TextMessage objects encapsulate the data being sent to or received from a JMS provider. Even if you use the plain *jms_template.send("Foobar")* to send an ordinary text, there's still a TextMessage instance created automatically underneath for you.

If all you need from JMS is simply to send and receive some text then you're not likely to be required to use TextMessages. However, if you have to set or read JMS attributes or you're interested in setting custom JMS properties then TextMessage is what you're looking for.

In Spring Python there are no clumsy setters and getters as in Java JMS. If you need to set the property of a message, you just write it, like for instance *message.jms_correlation_id = "1234567"*. Here's the list of all TextMessage's attributes along with their explanation and usage notes.

Table 8.4. `springpython.jms.core.TextMessage` default attributes

text	The message contents, the actual business payload carried by a message. May be both read and written to. For messages sent to a JMS provider it must be either a string or a unicode object encodable into UTF-8.
	
	The following two code snippets are equivalent:
	
	`message = TextMessage("Hey")`
	
	`message = TextMessage()`
	`message.text = "Hey"`
	
	Here's how to get the content of a message received by a JmsTemplate.
	
	`# .. skip creating the connection factory and a JmsTemplate`
	
	`message = jms_template.receive()`
	`print message.text`

jms_correlation_id	Equivalent to Java's JMSCorrelationID message header. It must be a string instance when set manually - a good way to produce correlation identifiers is to use the Python's `uuid4` type, e.g.:
	
	`# stdlib`
	`from uuid import uuid4`
	
	`# Spring Python`
	`from springpython.jms.core import TextMessage`
	
	`# Prapare the JMS correlation ID`
	`jms_correlation_id = uuid4().hex`
	
	`message = TextMessage("Howdy")`
	`message.jms_correlation_id = jms_correlation_id`
	
	`# Now the message can be sent with a JMS correlation ID such`
	`# which is a 128 bits long identifier.`

jms_delivery_mode	Equivalent to Java's JMSDeliveryMode, can be both read and written to and must be equal to one of the following values `springpython.jms.DELIVERY_MODE_NON_PERSISTENT`, `springpython.jms.DELIVERY_MODE_PERSISTENT` or `springpython.jms.DEFAULT_DELIVERY_MODE`. The default value - `DEFAULT_DELIVERY_MODE` - equals to `DELIVERY_MODE_PERSISTENT`.

jms_destination	Equivalent to Java's JMSDestination, automatically populated by JmsTemplate objects on send or receive time. May be read from but *must not be set manually*.
jms_expiration	Same as Java's JMSExpiration - allow for a message to expire after a certain amount of time. The value is automatically set by JmsTemplate for received messages. For messages being sent the time expressed is in milliseconds, as in the following code: `message = TextMessage("I will expire in half a second")` `# Set the message's expiration time to 500 ms` `message.jms_expiration = 500`
jms_message_id	Same as Java's JMSMessageID. Automatically set by JmsTemplate for received messages, may be set manually but the value will be ignored by the JMS provider.
jms_priority	Equivalent to Java's JMSPriority, may be set to an integer value. If not set manually, the value will be automatically computed by the JMS provider. An incoming message will have the value set as given by the JMS provider.
jms_redelivered	Same as Java's JMSRedelivered header. Should not be set manually. Default value for incoming messages is *False*; for messages received from WebSphere MQ (which is currently the only supported JMS provider) it will be *True* if the underlying MQ message's *BackoutCount* attribute is 1 or greater.
jms_reply_to	Equivalent to Java's JMSReplyTo, the name of a JMS destination to which responses to the currently sent message should be delivered. `message = TextMessage("Please, reply to me.")` `# Set the reply to queue` `message.jms_reply_to = "REPLY.TO.QUEUE.1"` See here for an example of how to use *jms_reply_to* in request/reply scenarios.
jms_timestamp	Same as Java's JMSTimestamp, the timestamp of a message returned as a number of milliseconds with a centiseconds precision. Should not be set manually.
max_chars_printed	Specifies how many characters of the business payload (the `.text` attribute) will be returned by the TextMessage instance's `__str__` method, which is used, for instance, for logging purposes. Default value is 100 characters. Consider the code below, in both cases the message's

content is the same, the messages differ only by the value of the `max_chars_printed` attribute.

```
# Spring Python
from springpython.jms.core import TextMessage

payload = "Business payload. " * 8

msg = TextMessage(payload)
msg.max_chars_printed = 50

print msg

# Will show in the console:

JMS message class: jms_text
  jms_delivery_mode:  2
  jms_expiration:     None
  jms_priority:       None
  jms_message_id:     None
  jms_timestamp:      None
  jms_correlation_id: None
  jms_destination:    None
  jms_reply_to:       None
  jms_redelivered:    None
Business payload. Business payload. Business paylo
Another 94 character(s) omitted
```

```
# Spring Python
from springpython.jms.core import TextMessage

payload = "Business payload. " * 8

msg = TextMessage(payload)
msg.max_chars_printed = 130

print msg

# Will show in the console:

JMS message class: jms_text
  jms_delivery_mode:  2
  jms_expiration:     None
  jms_priority:       None
  jms_message_id:     None
  jms_timestamp:      None
  jms_correlation_id: None
  jms_destination:    None
  jms_reply_to:       None
  jms_redelivered:    None
Business payload. Business payload. Business payload. Busine
Another 14 character(s) omitted
```

Attributes shown in the table above are standard JMS headers, available regardless of the JMS provider used. For WebSphereMQ - which is currently the only JMS provider supported by Spring Python - following attributes are also available: JMS_IBM_Report_Exception, JMS_IBM_Report_Expiration, JMS_IBM_Report_COA, JMS_IBM_Report_COD, JMS_IBM_Report_PAN, JMS_IBM_Report_NAN, JMS_IBM_Report_Pass_Msg_ID, JMS_IBM_Report_Pass_Correl_ID, JMS_IBM_Report_Discard_Msg, JMSXGroupID, JMSXGroupSeq, JMS_IBM_Feedback, JMS_IBM_Last_Msg_In_Group, JMSXUserID, JMS_IBM_PutTime, JMS_IBM_PutDate and JMSXAppID. Refer to the IBM's Java JMS documentation for info on how to use them.

Creating custom JMS properties is simply a matter of assigning a value to an attribute, there are no special methods such as `setStringProperty/getStringProperty` which are used in Java JMS, thus the following code will create a custom *MESSAGE_NAME* property which can be read by `getStringProperty` on the Java side.

```
# Spring Python
from springpython.jms.core import TextMessage

msg = TextMessage("Hello!")
msg.MESSAGE_NAME = "HelloRequest"
```

Observe how custom properties will be printed to the console along with standard JMS headers:

```
# Spring Python
from springpython.jms.core import TextMessage

msg = TextMessage("Hello!")
msg.MESSAGE_NAME = "HelloRequest"
msg.CLIENT = "CRM"
msg.CUSTOMER_ID = "201888228"

print msg

# Will show:

JMS message class: jms_text
  jms_delivery_mode:  2
  jms_expiration:     None
  jms_priority:       None
  jms_message_id:     None
  jms_timestamp:      None
  jms_correlation_id: None
  jms_destination:    None
  jms_reply_to:       None
  jms_redelivered:    None
  CLIENT:CRM
  CUSTOMER_ID:201888228
  MESSAGE_NAME:HelloRequest
Hello!
```

Not all TextMessage's attributes can be set to a custom value, the exact list of reserved attributes' names is available as `springpython.jms.core.reserved_attributes`. There's a very slim chance you'll ever encounter the conflict with your application's message attributes, nevertheless be sure to check the list before using custom JMS properties in your code.

8.8. Exceptions

`springpython.jms.JMSException` is the base exception class for all JMS-related issues that may be raised by Spring Python's JMS and a pair of its specialized subclasses is also available: `springpython.jms.NoMessageAvailableException` and `springpython.jms.WebSphereMQJMSException`.

NoMessageAvailableException is raised when a call to *receive* or *receive_and_convert* timeouts, which indicates that there's no message available for a given JMS destination.

WebSphereMQJMSException is raised when the underlying error is known to be caused by a call to WebSphere MQ API, such as a call to connect to a queue manager. Spring Python tries to populate these attributes of a WebSphereMQJMSException object when an error condition arises:

- *message*
- *completion_code*

reason_code

Note that *message*, *completion_code* and *reason_code* are all optional and there's no guarantee they will be actually returned. Should you caught a WebSphereMQJMSException, you should always check for their existence before making any use of them.

8.9. Logging and troubleshooting

Spring Python's JMS uses standard Python's `logging` module for emitting the messages. In general, you can expect JMS to behave sane, it won't overflow your logs with meaningless entries, e.g. if you configure it to log the messages at the `ERROR` level then you'll be notified of only truly erratic situtations.

In addition to `logging`'s builtin levels, JMS uses one custom level - `springpython.util.TRACE1`, *enabling TRACE1 will degrade the performance considerably* and will result in a huge number of messages written to the logs. Use it sparingly at troubleshooting times when you'd like to see the exact flow of messages, raw bytes and JMS headers passing by the Spring Python's JMS classes involved. Do not ever enable it in production environments unless you have a very compelling reason and you're sure you're comfortable with paying the performance penalty. Consider using the `logging.DEBUG` level instead of `TRACE1` if all you're after is simply seeing the messages' payload.

JMS loggers currently employed by Spring Python are `springpython.jms.factory.WebSphereMQConnectionFactory`, `springpython.jms.listener.SimpleMessageListenerContainer` and `springpython.jms.listener.WebSphereMQListener(`*`LISTENER_INSTANCE_ID`*`)`.

Here's how the WebSphere MQ connection factory's logger can be configured to work at the `INFO` level:

```
# stdlib
import logging

log_format = "%(asctime)s - %(levelname)s - %(process)d - %(threadName)s - %(name)s - %(message)s"
formatter = logging.Formatter(log_format)

handler = logging.StreamHandler()
handler.setFormatter(formatter)

jms_logger = logging.getLogger("springpython.jms.factory.WebSphereMQConnectionFactory")

jms_logger.setLevel(level=logging.INFO)
jms_logger.addHandler(handler)
```

Here's how to configure it to log messages at the `TRACE1` level:

```
# stdlib
import logging

# Spring Python
from springpython.util import TRACE1

log_format = "%(asctime)s - %(levelname)s - %(process)d - %(threadName)s - %(name)s - %(message)s"
formatter = logging.Formatter(log_format)

handler = logging.StreamHandler()
handler.setFormatter(formatter)

jms_logger = logging.getLogger("springpython.jms.factory.WebSphereMQConnectionFactory")

jms_logger.setLevel(level=TRACE1)
jms_logger.addHandler(handler)
```

`springpython.jms.listener.SimpleMessageListenerContainer` is the logger used by the JMS listener container itself.

Each WebSphere MQ listener is assigned a `springpython.jms.listener.WebSphereMQListener(`*`LISTENER_INSTANCE_ID`*`)` logger, where *LISTENER_INSTANCE_ID* is an identifier uniquely associated with a listener to form a full name of a logger, such as `springpython.jms.listener.WebSphereMQListener(0xc7f5e0)`. To be precise, its value is obtained by invoking hex(id(self)) on the listener's instance. Note that the value is not guaranteed to be globally unique, it's just an identifier of the Python object so its value may be very well reused across application's restarts.

How much information is being logged depends on the logging level, the average message size, the messages' `max_chars_printed` attribute value and the message rate.

Here's an estimation of how fast log files will grow depending on the logging level. During the test, the message size was 5kB, there were a total of 10,000 messages sent, the `max_chars_printed` attribute had value of 100 and the log entries were written to an ordinary log file.

- `ERROR`
- `INFO`
- `DEBUGmax_chars_printed`
- `TRACE1`*tenfold*`DEBUG`

Chapter 9. Spring Python's plugin system

Spring Python's plugin system is designed to help you rapidly develop applications. Plugin-based solutions have been proven to enhance developer efficiency, with examples such as Grails and Eclipse being market leaders in usage and productivity.

This plugin solution was mainly inspired by the Grails demo presented by Graeme Rocher at the SpringOne Americas 2008 conference, in which he created a Twitter application in 40 minutes. Who wouldn't want to have something similar to support Spring Python development?

9.1. Introduction

Have you considered submitting your plugin as a Spring Extension?

Spring Extensions is the official incubator process for SpringSource. You can always maintain your own plugin separately, using whatever means you wish. But if want to get a larger adoption of your plugin, name association with SpringSource, and perhaps one day becoming an official part of the software suite of SpringSource, you may want to consider looking into the Spring Extensions process.

Spring Python will manage an approved set of plugins. These are plugins written by the committers of Spring Python and are verified to work with an associated version of the library. These plugins are also hosted by the same services used to host Spring Python downloads, meaning they have the same level of support as Spring Python.

However, being an open source framework, developers have every right to code their own plugins. We fully support the concept of 3rd party plugins. We want to provide as much support in the way of documentation and extension points for you to develop your own plugins as well.

9.2. Coily - Spring Python's command-line tool

Coily is the command-line tool that utilizes the plugin system. It is similar to grails command-line tool, in that through a series of installed plugins, you are able to do many tasks, including build skeleton apps that you can later flesh out. If you look at the details of this app, you will find a sophisticated, command driven tool to built to manage plugins. The real power is in the plugins themselves.

9.2.1. Commands

To get started, all you need is a copy of coily installed in some directory located on your path.

```
% coily --help
```

The results should list available commands.

```
Coily - the command-line management tool for Spring Python
==========================================================
Copyright 2006-2008 SpringSource (http://springsource.com), All Rights Reserved
Licensed under the Apache License, Version 2.0
```

```
Usage: coily [command]

        --help                          print this help message
        --list-installed-plugins        list currently installed plugins
        --list-available-plugins        list plugins available for download
        --install-plugin [name]         install coily plugin
        --uninstall-plugin [name]       uninstall coily plugin
        --reinstall-plugin [name]       reinstall coily plugin
```

- --help - Print out the help menu being displayed
- --list-installed-plugins - list the plugins currently installed in this account. It is important to know that each plugin creates a directly underneath the user's home directory in a hidden directory `.springpython`. If you delete this entire directory, you have effectively uninstalled all plugins.
- --list-available-plugins - list the plugins available for installation. Coily will check certain network locations, such as the S3 site used to host Spring Python downloads. It will also look on the local file system. This is in case you have a checked out copy of the plugins source code, and want to test things out without uploading to the network.
- --install-plugin - install the named plugin. In this case, you don't have to specify a version number. Coily will figure out which version of the plugin you need, download it if necessary, and finally copy it into `~/.springpython`.
- --uninstall-plugin - uninstall the named plugin by deleting its entry from `~/.springpython`
- --reinstall-plugin - uninstall then install the plugin. This is particulary useful if you are working on a plugin, and need a shortcut step to deploy.

In this case, no plugins have been installed yet. Every installed plugin will list itself as another available command to run. If you have already installed the *gen-cherrypy-app* plugin, you will see it listed.

```
Coily - the command-line management tool for Spring Python
==========================================================
Copyright 2006-2008 SpringSource (http://springsource.com), All Rights Reserved
Licensed under the Apache License, Version 2.0


Usage: coily [command]

        --help                          print this help message
        --list-installed-plugins        list currently installed plugins
        --list-available-plugins        list plugins available for download
        --install-plugin [name]         install coily plugin
        --uninstall-plugin [name]       uninstall coily plugin
        --reinstall-plugin [name]       reinstall coily plugin
        --gen-cherrypy-app [name]       plugin to create skeleton CherryPy applications
```

You should notice an extra option listed at the bottom: *gen-cherrypy-app* is listed as another command with one argument. Later on, you can read official documentation on the existing plugins, and also how to write your own.

9.3. Officially Supported Plugins

This section documents plugins that are developed by the Spring Python team.

9.3.1. gen-cherrypy-app

This plugin is used to generate a skeleton CherryPy application based on feeding it a command-line argument.

9.3.1.1. External dependencies

gen-cherrypy-app plugin requires the installation of CherryPy 3.

```
% coily --gen-cherrypy-app twitterclone
```

This will generate a subdirectory `twitterclone` in the user's current directory. Inside twitterclone are several files, including `twitterclone.py`. If you run the app, you will see a working CherryPy application, with Spring Python security in place.

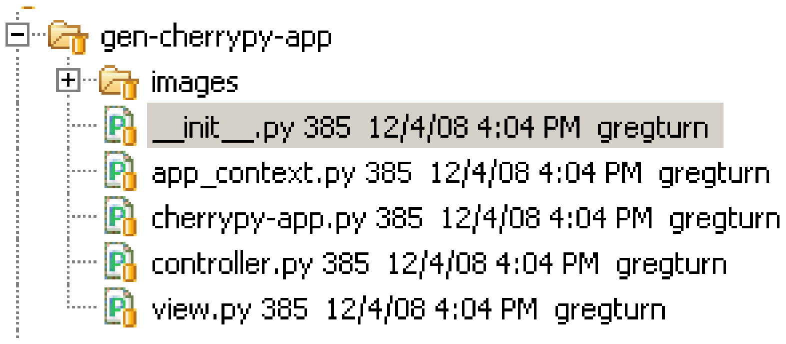
```
% cd twitterclone
% python twitterclone.py
```

You can immediately start modifying it to put in your features.

9.4. Writing your own plugin

9.4.1. Architecture of a plugin

A plugin is pretty simple in structure. It is basically a python package with some special things added on. *gen-cherrypy-app* plugin demonstrates this.



```
gen-cherrypy-app
  images
  __init__.py 385  12/4/08 4:04 PM  gregturn
  app_context.py 385  12/4/08 4:04 PM  gregturn
  cherrypy-app.py 385  12/4/08 4:04 PM  gregturn
  controller.py 385  12/4/08 4:04 PM  gregturn
  view.py 385  12/4/08 4:04 PM  gregturn
```

The special things needed to define a plugin are as follows:

- A root folder with the same name as your plugin and a `__init__.py`, making the plugin a python package
- A package-level variable named `__description__`

  This attribute should be assigned the string value description you want shown for your plugin when coily --help is run.
- A package-level function named either `create` or `apply`
  - If your plugin needs one command line argument, define a `create` method with the following signature:

```
def create(plugin_path, name)
```

- If your plugin doesn't need any arguments, define an `apply` method with the following signature:

```
def apply(plugin_path)
```

In either case, your plugin gets passed an extra argument, `plugin_path`, which contains the directory the plugin is actually installed in. This is typically so you can reference other files your plugin needs access to.

What does "package-level" mean?

The code needs to be in the `__init__.py` file. This file makes the enclosing directory a python package.

9.4.2. Case Study - gen-cherrypy-app plugin

gen-cherrypy-app is a plugin used to build a CherryPy web application using Spring Python's feature set. It saves the developer from having to re-configure Spring Python's security module, coding CherryPy's engine, and so forth. This allows the developer to immediately start writing business code against a working application.

Using this plugin, we will de-construct this simple, template-based plugin. This will involve looking line-by-line at `gen-cherrypy-app/__init__.py`.

9.4.2.1. Source Code

```
"""
   Copyright 2006-2008 SpringSource (http://springsource.com), All Rights Reserved

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
"""
import re
import os
import shutil

__description__ = "plugin to create skeleton CherryPy applications"

def create(plugin_path, name):
    if not os.path.exists(name):
        print "Creating CherryPy skeleton app %s" % name
        os.makedirs(name)

        # Copy/transform the template files
        for file_name in ["cherrypy-app.py", "controller.py", "view.py", "app_context.py"]:
            input_file = open(plugin_path + "/" + file_name).read()

            # Iterate over a list of patterns, performing string substitution on the input file
            patterns_to_replace = [("name", name), ("properName", name[0].upper() + name[1:])]
            for pattern, replacement in patterns_to_replace:
                input_file = re.compile(r"\$\{%s}" % pattern).sub(replacement, input_file)
            output_filename = name + "/" + file_name
            if file_name == "cherrypy-app.py":
                output_filename = name + "/" + name + ".py"

            app = open(output_filename, "w")
            app.write(input_file)
            app.close()

        # Recursively copy other parts
        shutil.copytree(plugin_path + "/images", name + "/" + "images")
    else:
        print "There is already something called %s. ABORT!" % name
```

9.4.2.2. Deconstructing the factory

- The opening section shows the copyright statement, which should tip you off that this is an official plugin.
- `__description__` is a required variable.

```
__description__ = "plugin to create skeleton CherryPy applications"
```

  It contains the description displayed when a user runs:

```
% coily --help
```

```
Usage: coily [command]
...
        --gen-cherrypy-app [name]       plugin to create skeleton CherryPy applications
```

- Opening line defines `create` with two arguments.

```
def create(plugin_path, name):
```

  The arguments allow both the plugin path to be fed along with the command-line argument that is filled in when the user runs the command:

```
% coily --gen-cherrypy-app [name]
```

  It is important to realize that `plugin_path` is needed in case the plugin needs to refer to any files inside its installed directory. This is because plugins are not installed anywhere on the `PYTHONPATH`, but instead, in the user's home directory underneath `.springpython`.

  This mechanism was chosen because it gives users an easy ability to pick which plugins they wish to use, without requiring system admin power. It also eliminates the need to deal with multiple versions of plugins being installed on your `PYTHONPATH`. This provides maximum flexibility which is needed in a development environment.

- This plugin works by creating a directory in the user's current working directory, and putting all relevant files into it. The argument passed into the command-line is used as the name of an application, and the directory created has the same name.

```
  if not os.path.exists(name):
      print "Creating CherryPy skeleton app %s" % name
      os.makedirs(name)
```

However, if the directory already exists, it won't proceed.

```
  else:
      print "There is already something called %s. ABORT!" % name
```

- This plugin then iterates over a list of filenames, which happen to match the names of files found in the plugin's directory. These are essentially template files, intended to be copied into the target directory. However, the files are not copied directly. Instead they are opened and read into memory.

```
      # Copy/transform the template files
      for file_name in ["cherrypy-app.py", "controller.py", "view.py", "app_context.py"]:
          input_file = open(plugin_path + "/" + file_name).read()
```

  Then, the contents are scanned for key phrases, and substituted. In this case, the substitution is a variant of the name of the application being generated.

```
          # Iterate over a list of patterns, performing string substitution on the input file
          patterns_to_replace = [("name", name), ("properName", name[0].upper() + name[1:])]
          for pattern, replacement in patterns_to_replace:
              input_file = re.compile(r"\$\{%s}" % pattern).sub(replacement, input_file)
```

  The substituted content is written to a new output file. In most cases, the original filename is also the target filename. However, the key file, `cherrypy-app.py` is renamed to the application's name.

```
          output_filename = name + "/" + file_name
          if file_name == "cherrypy-app.py":
              output_filename = name + "/" + name + ".py"

          app = open(output_filename, "w")
          app.write(input_file)
          app.close()
```

- Finally, the images directory is recursively copied into the target directory.

```
      # Recursively copy other parts
      shutil.copytree(plugin_path + "/images", name + "/" + "images")
```

9.4.2.3. Summary

All these steps effectively copy a set of files used to template an application. With this template approach, the major effort of developing this plugin is spent working on the templates themselves, not on this template factory. While this is mostly working with python code for a python solution, the fact that this is a template requires reinstalling the plugin everytime a change is made in order to test them.

Users are welcome to use *gen-cherypy-app*'s `__init__.py` file to generate their own template solutions, and work on other skeleton tools or solutions.

Chapter 10. Samples

10.1. PetClinic

PetClinic is a sample application demonstrating the usage of Spring Python.

- It uses CherryPy as the web server object.
- A detailed design document (NOTE: find latest version, and click on *raw*) is part of the source code. You can read it from here or by clicking on a hyperlink while running the application.

10.1.1. How to run

Assuming you just checked out a copy of the source code, here are the steps to run PetClinic.

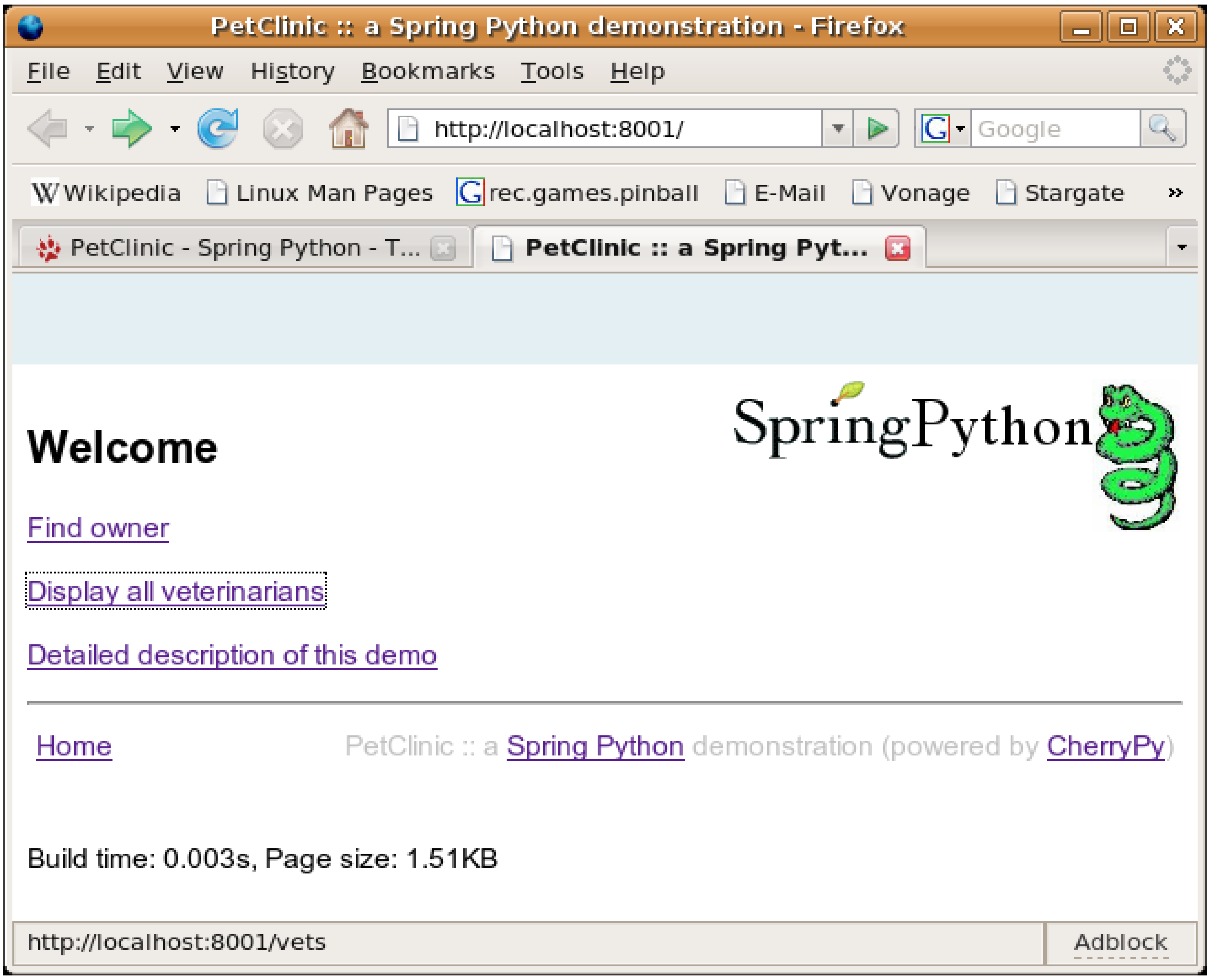
```
bash$ cd /path/you/checked/out/springpython
bash$ cd samples/petclinic
bash$ python configure.py
```

At this point, you will be prompted for MySQL's root password. This is NOT your system's root password. This assumes you have a MySQL server running. After that, it will have setup database *petclinic*.

```
bash$ cd cherrypy
bash$ python petclinic.py
```

This assumes you have CherryPy 3 installed. It probably *won't* work if you are still using CherryPy 2. NOTE: If you are using Python 2.5.2+, you must install CherryPy 3.1.2+. The older version of CherryPy (3.1.0) only works pre-2.5.2.

Finally, after launching it, you should see a nice URL at the bottom: http://localhost:8080. Well, go ahead! Things should look good now!


PetClinic :: a Spring Python demonstration - Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8001/
Google
Wikipedia Linux Man Pages rec.games.pinball E-Mail Vonage Stargate
PetClinic - Spring Python - T...
PetClinic :: a Spring Pyt...
SpringPython
Home
http://localhost:8001/vets
Adblock

Welcome

Find owner

Display all veterinarians

Detailed description of this demo

PetClinic :: a Spring Python demonstration (powered by CherryPy)

Build time: 0.003s, Page size: 1.51KB

Snapshot of PetClinic application

10.2. Spring Wiki

Spring Wiki is a wiki engine based that uses mediawiki's markup language. It utilizes the same stylesheets to have a very wikipedia-like feel to it.

TODO: Add persistence. Currently, Spring Wiki only stores content in current memory. Shutting it down will cause all changes to be lost.

10.3. Spring Bot

This article will show how to write an IRC bot to manage a channel for your open source project, like the one I have managing #springpython, the IRC chat channel for Spring Python.

10.3.1. Why write a bot?

I read an article, Building a community around your open source project, that talked about setting up an IRC channel for your project. This is a route to support existing users, and allow them to work with each other.

I became very interested in writing some IRC bot, and I since my project is based on Python, well, you can probably guess what language I wanted to write it in.

10.3.2. IRC Library

To build a bot, it pays to have use an already written library. I discovered python-irclib.

For Ubuntu users:

```
% sudo apt-get install python-irclib
```

This bot also sports a web page using CherryPy. You also need to install that as well.

10.3.2.1. Articles

Well, of course I started reading. The documentation from the project's web site was minimal. Thankfully, I found some introductory articles that work with python-irclib.

- http://www.devshed.com/c/a/Python/IRC-on-a-Higher-Level/
- http://www.devshed.com/c/a/Python/IRC-on-a-Higher-Level-Continued/
- http://www.devshed.com/c/a/Python/IRC-on-a-Higher-Level-Concluded/

10.3.3. What I built

Using this, I managed to get something primitive running. It took me a while to catch on that posting private messages on a channel name instead of a user is the way to publicly post to a channel. I guess it helped to trip through the IRC RFC manual, before catching on to this.

At this stage, you may wish to get familiar with regular expressions in Python. You will certainly need this in order to make intelligent looking patterns. Anything more sophisticated would probably require PLY.

What I really like is that fact that I built this application in approximately 24 hours, counting the time to learn how to use python-irclib. I already knew how to build a Spring Python/CherryPy web application. The history pages on this article should demonstrate how long it took.

NOTE: This whole script is contained in one file, and marked up as:

```
"""
   Copyright 2006-2008 SpringSource (http://springsource.com), All Rights Reserved

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
"""
```

10.3.3.1. IRC Bot

So far, this handy little bot is able to monitor the channel, log all communications, persistently fetch/store things, and grant me operator status when I return to the channel. My next task is to turn it into a web app using Spring Python. That should let me have a web page to go along with the channel!

```
class DictionaryBot(ircbot.SingleServerIRCBot):
    def __init__(self, server_list, channel, ops, logfile, nickname, realname):
        ircbot.SingleServerIRCBot.__init__(self, server_list, nickname, realname)
        self.datastore = "%s.data" % self._nickname
        self.channel = channel
        self.definition = {}
        try:
            f = open(self.datastore, "r")
            self.definition = cPickle.load(f)
            f.close()
        except IOError:
            pass
        self.whatIsR = re.compile(",?\s*[Ww][Hh][Aa][Tt]\s*[Ii][Ss]\s+([\w ]+)[?]?")
        self.definitionR = re.compile(",?\s*([\w ]+)\s+[Ii][Ss]\s+(.+)")
        self.ops = ops
        self.logfile = logfile

    def on_welcome(self, connection, event):
        """This event is generated after you connect to an irc server, and should be your signal to join your ta
        connection.join(self.channel)

    def on_join(self, connection, event):
        """This catches everyone who joins. In this case, my bot has a list of whom to grant op status to when t
        self._log_event(event)
        source = event.source().split("!")[0]
        if source in self.ops:
            connection.mode(self.channel, "+o %s" % source)

    def on_mode(self, connection, event):
        """No real action here, except to log locally every mode action that happens on my channel."""
        self._log_event(event)

    def on_pubmsg(self, connection, event):
        """This is the real meat. This event is generated everytime a message is posted to the channel."""
        self._log_event(event)

        # Capture who posted the messsage, and what the message was.
        source = event.source().split("!")[0]
        arguments = event.arguments()[0]

        # Some messages are meant to signal this bot to do something.
        if arguments.lower().startswith("!%s" % self._nickname):
            # "What is xyz" command
            match = self.whatIsR.search(arguments[len(self._nickname)+1:])
            if match:
                self._lookup_definition(connection, match.groups()[0])
                return

            # "xyz is blah blah" command
            match = self.definitionR.search(arguments[len(self._nickname)+1:])
            if match:
                self._set_definition(connection, match.groups()[0], match.groups()[1])
                return

        # There are also some shortcut commands, so you don't always have to address the bot.
        if arguments.startswith("!"):
            match = re.compile("!([\w ]+)").search(arguments)
            if match:
                self._lookup_definition(connection, match.groups()[0])
                return

    def getDefinitions(self):
        """This is to support a parallel web app fetching data from the bot."""
        return self.definition

    def _log_event(self, event):
        """Log an event to a flat file. This can support archiving to a web site for past activity."""
        f = open(self.logfile, "a")
        f.write("%s::%s::%s::%s\n" % (event.eventtype(), event.source(), event.target(), event.arguments()))
        f.close()

    def _lookup_definition(self, connection, keyword):
        """Function to fetch a definition from the bot's dictionary."""
        if keyword.lower() in self.definition:
            connection.privmsg(self.channel, "%s is %s" % self.definition[keyword.lower()])
        else:
            connection.privmsg(self.channel, "I have no idea what %s means. You can tell me by sending '!%s, %s

    def _set_definition(self, connection, keyword, definition):
        """Function to store a definition in cache and to disk in the bot's dictionary."""
        self.definition[keyword.lower()] = (keyword, definition)
        connection.privmsg(self.channel, "Got it! %s is %s" % self.definition[keyword.lower()])
        f = open(self.datastore, "w")
        cPickle.dump(self.definition, f)
        f.close()
```

I have trimmed out the instantiation of this bot class, because that part isn't relevant. You can go and immediately reuse this bot to manage any channel you have.

10.3.3.2. Web App

Well, after getting an IRC bot working that quickly, I want a nice interface to see what it is up to. For that, I will use Spring Python and build a Spring-based web app.

```

def header():
    """Standard header used for all pages"""
    return """
        <!--

            Coily :: An IRC bot used to manage the #springpython irc channel (powered by CherryPy/Spring Python)

        -->

        <html>
        <head>
        <title>Coily :: An IRC bot used to manage the #springpython irc channel (powered by CherryPy/Spring Pyth
            <style type="text/css">
                    td { padding:3px; }
                    div#top {position:absolute; top: 0px; left: 0px; background-color: #E4EFF3; height: 50px; wi
                    div#image {position:absolute; top: 50px; right: 0%; background-image: url(images/spring_pyth
            </style>
        </head>

        <body>
            <div id="top">&nbsp;</div>
            <div id="image">&nbsp;</div>
            <br clear="all">
            <p>&nbsp;</p>
        """

def footer():
    """Standard footer used for all pages."""
    return """
        <hr>
        <table style="width:100%"><tr>
                <td><A href="/">Home</A></td>
                <td style="text-align:right;color:silver">Coily :: a <a href="http://springpython.webfactional.c
        </tr></table>

        </body>
        """

def markup(text):
    """Convert any http://xyz references into real web links."""
    httpR = re.compile(r"(http://[\w.:/?-]*\w)")
    alteredText = httpR.sub(r'<A HREF="\1">\1</A>', text)
    return alteredText

class CoilyView:
    """Presentation layer of the web application."""

    def __init__(self, bot = None):
        """Inject a controller object in order to fetch live data."""
        self.bot = bot

    @cherrypy.expose
    def index(self):
        """CherryPy will call this method for the root URI ("/") and send
        its return value to the client."""

        return header() + """
            <H2>Welcome</H2>
            <P>
            Hi, I'm Coily! I'm a bot used to manage the IRC channel <a href="irc://irc.ubuntu.com/#springpython"
            <P>
            If you visit the channel, you may find I have a lot of information to offer while you are there. If
            <small>
                <TABLE border="1">
                    <TH>Command</TH>
                    <TH>Description</TH>
                    <TR>
                        <TD>!coily, what is <i>xyz</i>?</TD>
                        <TD>This is how you ask me for a definition of something.</TD>
                    </TR>
                    <TR>
                        <TD>!<i>xyz</i></TD>
                        <TD>This is a shortcut way to ask the same question.</TD>
                    </TR>
                    <TR>
                        <TD>!coily, <i>xyz</i> is <i>some definition for xyz</i></TD>
                        <TD>This is how you feed me a definition.</TD>
                    </TR>
                </TABLE>
            </small>
            <P>
            To save you from having to query me for every current definition I have, there is a link on this web
            that lists all my current definitions. NOTE: These definitions can be set by other users.
            <P>
            <A href="listDefinitions">List current definitions</A>
            <P>
            """ + footer()

    @cherrypy.expose
    def listDefinitions(self):
        results = header()
        results += """
                <small>
                <TABLE border="1">
                    <TH>Keyword</TH>
                    <TH>Definition</TH>
            """
        for key, value in self.bot.getDefinitions().items():
            results += markup("""
                <TR>
                    <TD>%s</TD>
                    <TD>%s</TD>
                </TR>
                """ % (value[0], value[1]))
        results += "</TABLE></small>"
        results += footer()
        return results
```

10.3.3.3. Putting it all together

Well, so far, I have two useful classes. However, they need to get launched inside a script. This means objects need to be instantiated. To do this, I have decided to make this a Spring app and use inversion of control.

So, I defined two contexts, one for the IRC bot and another for the web application.

10.3.3.3.1. IRC Bot's application context

```
class CoilyIRCServer(PythonConfig):
    """This container represents the context of the IRC bot. It needs to export information, so the web app can
    def __init__(self):
        super(CoilyIRCServer, self).__init__()

    @Object
    def remoteBot(self):
        return DictionaryBot([("irc.ubuntu.com", 6667)], "#springpython", ops=["Goldfisch"], nickname="coily", r

    @Object
    def bot(self):
        exporter = PyroServiceExporter()
        exporter.service_name = "bot"
        exporter.service = self.remoteBot()
        return exporter
```

10.3.3.3.2. Web App's application context

```
class CoilyWebClient(PythonConfig):
    """
    This container represents the context of the web application used to interact with the bot and present a
    nice frontend to the user community about the channel and the bot.\
    """
    def __init__(self):
        super(CoilyWebClient, self).__init__()

    @Object
    def root(self):
        return CoilyView(self.bot())

    @Object
    def bot(self):
        proxy = PyroProxyFactory()
        proxy.service_url = "PYROLOC://localhost:7766/bot"
        return proxy
```

10.3.3.3.3. Main runner

I fit all this into one executable. However, I quickly discovered that both CherryPy web apps and irclib bots like to run in the main thread. This means I need to launch two python shells, one running the web app, the other running the ircbot, and I need the web app to be able to talk to the irc bot. This is a piece of cake with Spring Python. All I need to utilize is a remoting technology.

```
if __name__ == "__main__":
    # Parse some launching options.
    parser = OptionParser(usage="usage: %prog [-h|--help] [options]")
    parser.add_option("-w", "--web", action="store_true", dest="web", default=False, help="Run the web server ob
    parser.add_option("-i", "--irc", action="store_true", dest="irc", default=False, help="Run the IRC-bot objec
    parser.add_option("-d", "--debug", action="store_true", dest="debug", default=False, help="Turn up logging l
    (options, args) = parser.parse_args()

    if options.web and options.irc:
        print "You cannot run both the web server and the IRC-bot at the same time."
        sys.exit(2)

    if not options.web and not options.irc:
        print "You must specify one of the objects to run."
        sys.exit(2)

    if options.debug:
        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)

    if options.web:
        # This runs the web application context of the application. It allows a nice web-enabled view into
        # the channel and the bot that supports it.
        applicationContext = ApplicationContext(CoilyWebClient())

        # Configure cherrypy programmatically.
        conf = {"/":                {"tools.staticdir.root": os.getcwd()},
                    "/images":          {"tools.staticdir.on": True,
                                         "tools.staticdir.dir": "images"},
                    "/html":            {"tools.staticdir.on": True,
                                         "tools.staticdir.dir": "html"},
                    "/styles":          {"tools.staticdir.on": True,
                                         "tools.staticdir.dir": "css"}
                    }

        cherrypy.config.update({'server.socket_port': 9001})

        cherrypy.tree.mount(applicationContext.get_object(name = "root"), '/', config=conf)

        cherrypy.engine.start()
        cherrypy.engine.block()

    if options.irc:
        # This runs the IRC bot that connects to a channel and then responds to various events.
        applicationContext = ApplicationContext(CoilyIRCServer())
        coily = applicationContext.get_object("bot")
        coily.service.start()
```

10.3.3.4. Releasing your CherryPy web app to the internet

Now that you have a CherryPy web app running, how about making it visible to the internet?

If you already have an Apache web server running, and are using a Debian/Ubuntu installation, you just need to create a file in */etc/apache2/sites-available* like *coily.conf* with the following lines:

```

RedirectMatch ^/coily$ /coily/

ProxyPass /coily/ http://localhost:9001/
ProxyPassReverse /coily/ http://localhost:9001/

<LocationMatch /coily/.*>
    Order allow,deny
    Allow from all
</LocationMatch>
```

Now need to softlink this into */etc/apache2/sites-enabled*.

```

% cd /etc/apache2/sites-enabled
% sudo ln -s /etc/apache2/sites-available/coily.conf 001-coily
```

This requires that enable *mod_proxy*.

```

% sudo a2enmod proxy proxy_http
```

Finally, restart apache.

```

% sudo /etc/init.d/apache2 --force-reload
```

It should be visible on the site now.

10.3.3.5. Come and visit Coily

If you haven't figured it out yet, I use this code to run my own bot, Coily. Unfortunately, at this time, I don't have a mechanism to make it run persistently.

10.3.4. External Links

- See this article reported in LinuxToday