A Lightweight Introduction to the Spring Framework

Just Spring

O'REILLY®

Madhusudhan Konda

Just Spring

Get a concise introduction to Spring, the increasingly popular open source framework for building lightweight enterprise applications on the Java platform. This example-driven book for Java developers delves into the framework's basic features, as well as advanced concepts such as containers. You'll learn how Spring makes Java Messaging Service easier to work with, and how its support for Hibernate helps you work with data persistence and retrieval.

Throughout *Just Spring*, you'll get your hands deep into sample code, beginning with a problem that illustrates Spring's core principle: dependency injection. In the chapters that follow, author Madhusudhan Konda walks you through features that underlie the solution.

- Learn dependency injection through a simple object coupling problem, along with different injection types
- Tackle the framework's core fundamentals, including beans and bean factories
- Dive into containers and other advanced concepts, such as event handling and autowiring beans
- Discover how Spring makes the Java Messaging Service API easier to use
- Learn how Spring has revolutionized data access with Java DataBase Connectivity (JDBC)
- Use Spring with the Hibernate framework to manipulate data as objects

Twitter: @oreillymedia
facebook.com/oreilly

US $19.99 CAN $22.99
ISBN: 978-1-449-30640-3

O'REILLY®
oreilly.com

Just Spring

Just Spring

Madhusudhan Konda

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Just Spring
by Madhusudhan Konda

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: O'Reilly Publishing Services
Cover Designer: Karen Montgomery
Interior Designer: David Futato

Printing History:
July 2011: First Edition.

ISBN: 978-1-449-31146-9

[LSI]

1311270898

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Just Spring* by Madhusudhan Konda (O'Reilly). Copyright 2011 Madhusudhan Konda, 978-1-449-30640-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9781449306403

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I sincerely wish to thank my editor, Mike Loukides, for keeping faith in me and directing me when lost. Also to all of those in the O'Reilly team, especially Meghan Blanchette, Holly Bauer, Sarah Schneider. and Dan Fauxsmith, for helping shape this book.

Sincere thanks to my loving wife, Jeannette, for being very patient and supportive throughout the writing of this book. Also to my wonderful four-year-old son, Joshua, who surprisingly sacrificed his free time, allowing me to write when I explained to him what I was doing!

CHAPTER 1

Spring Basics

Introduction

The Spring Framework has found a very strong user base over the years. Software houses and enterprises found the framework to be the best fit for their plumbing needs. Surprisingly, the core principle that Spring has been built for—the *Dependency Injection* (DI)—is very simple to understand. This chapter discusses the fundamentals of the framework from a high ground. It helps the reader understand the dependency injection principles. The chapter opens up with a simple problem of object coupling and dependencies. It then explains how to solve the them using Spring Framework.

Object Coupling Problem

Let us consider a simple program whose objective is to read data from various data sources. It can read data from a file system or database or even from an FTP server. For simplicity, we will start writing a program that reads the data from a file system for now. The following example code is written without employing any best practices or dependency injection patterns—it's just a simple and plain program that works.

Example 1-1 shows a `Client` program that uses `FileReader` to fetch the data.

Example 1-1.

```
public class DataReaderClient {
    private FileReader fileReader =  null;
    private String fileName = "res/myfile.txt";
    public DataReaderClient() {
      fileReader = new FileReader(fileName);
    }
    private String fetchData() {
      return fileReader.read();
    }
    public static void main(String[] args) {
      DataReaderClient dataReader = new DataReaderClient();
      System.out.println("Got data: "+dataReader.fetchData());
```

```
    }
}
```

As the name suggests, the `DataReaderClient` is the client that fetches the data from a data source. When the program is executed, the `DataReaderClient` gets instantiated along with a referenced `FileReader` object. It then uses the `FileReader` object to fetch the result.

Example 1-2 is the implementation of `FileReader` class.

Example 1-2.

```
public class FileReader {
  private StringBuilder builder = null;
  private Scanner scanner = null;
  public FileReader(String fileName) {
    scanner = new Scanner(new File(fileName));
    builder= new StringBuilder();
  }
  public String read() {
    while (scanner.hasNext()) {
      builder.append(scanner.next());
    }
    return builder.toString();
  }
}
```

The limitation of the above client is that it can only read the data from file system. Imagine, one fine morning, your manager asks you to improvise the program to read data from a Database or Socket instead of File! With the current design, it is not possible to incorporate these changes without refactoring the code. Lastly (and very importantly), you can see the client and the reader are coupled tightly. That is, client depends on `FileReader`'s contract, meaning if `FileReader` changes, so does the client. If the client has already been distributed and used by, say, 1000 users across the globe, you will have fun refactoring the client!

If one's intention is to build good scalable and testable components, then coupling is a bad thing.

So, let's work out on your manager's demands and make the program read the data from any source. For this, we will take this program one step further—refactoring so the client can read from any datasource. For this refactoring, we have to rely on our famous *design to interfaces* principle.

Designing to Interfaces

Writing your code against interfaces is a very good practice. I am not going to sing praises about the best practises of designing here. The first step in designing to interfaces is to create an interface. The concrete classes will implement this interface, binding themselves to the interface rather than to an implementation. As long as we keep the

interface contract unchanged, the implementation can be modified any number of times without affecting the client.

For our data reader program, we create a `Reader` interface. This has just one method:

```
public interface Reader {
  String read();
}
```

The next step is to implement this contract. As we have to read the data from different sources, we create respective concrete implementations such as `FileReader` for reading from a File, `DatabaseReader` for reading from a Database, and `FtpReader` for reading from FtpServer. The template for concrete implementation goes in the form of *`XXXReader`* as shown below:

```
private class XXXReader implements Reader {
  public String read(){
    //impl goes here
  }
}
```

Once you have the `XXXReader` ready, the next step is to use it in the client program. However, instead of using the concrete class reference, use the interface reference.

For example, the modified client program shown below has a `Reader` variable reference, rather than `FileReader` or `FtpReader`. It has a constructor that takes in the `Reader` interface as a parameter.

```
public class DataReaderClient {
    private Reader reader =  null;
    public DataReaderClient(Reader reader) {
      this.reader = reader;
    }
    private String fetchData() {
      return reader.read();
    }
    public static void main(String[] args) {
      ...
    }
  }
```

Looking at the client code, if I ask you to tell me the actual reader that has been used by the client, would you be able to tell me? You can't! The `DataReaderClient` does not know where it is fed the data until runtime. The `Reader` class will only be resolved at runtime using Polymorphism. All we know is that the client can get any of the concrete implementations of `Reader` interface. The interface methods that were implemented in concrete incarnations of `Reader` are invoked appropriately.

The challenge is to provide the appropriate `Reader` to the client. One way to do this is to create a concrete implementation of `Reader` in the client program. It is shown below:

```
public class DataReaderClient {
  ...
  public static void main(String[] args) {
```

```
        Reader reader = new FileReader(); //Ummh..still hard wired, isn't it?
        DataReaderClient client = new DataReaderClient(reader); ...
      }
}
```

Well, it is still hard wired, and the client will have to know about which `Reader` it is going to use. Of course, you could swap `FileReader` with `DatabaseReader` or `FtpReader` without much hassle, as they all implement `Reader` interface. So, we are in much better position when the Manager comes along and changes his mind!

However, we still have the concrete `Reader` coupled to the client. Ideally, we should eliminate this coupling as much as possible. The question is how can we can provide an instance of `Reader` to `DataReader` without hardwiring? Is there a way we can abstract the creation of this `FileReader` away from the client?

Before I ask you more questions, let me tell you the answer: yes! Any *Dependency Injection* framework can do this job for us. One such framework is Spring Framework.

The Spring Framework is one of the Dependency Injection (or Inversion of Control) frameworks that provides the dependencies to your objects at runtime very elegantly. I won't explain the framework details yet, because I'm sure you're eager to find the solution to the above problem using Spring first.

Introducing Spring

The object interaction is a part and parcel of software programs. The good design allows you to replace the moving parts with no or minimal impact to the existing code. We say the objects are coupled tightly when the moving parts are knitted closely together. However, this type of design is inflexible—it is in a state where it cannot be scalable or testable in isolation or even maintainable without some degree of code change. Spring Framework can come to the rescue in designing the components eliminating dependencies.

Dependency Injection

Spring Framework works on one single mantra: *Dependency Injection*. This is sometimes interchangeable with the *Inversion of Control* (IoC) principle. When a standalone program starts, it starts the main program, creates the dependencies, and then proceeds to execute the appropriate methods. However, this is exactly the reverse if IoC is applied. That is, all the dependencies and relationships are created by the IoC container and then they are injected into the main program as properties. The program is then ready for action. This is essentially the reverse of usual program creation and hence is called *Inversion of Control* principle. The DI and IoC are often used interchangeably.

Refactoring Reader Using Framework

Coming back to our `Reader` program, the solution is to inject a concrete implementation of `Reader` into the client on demand.

Let us modify the `DataReaderClient`. The full listing is shown below. Don't worry about new classes you see in the client program; you will learn about them in no time.

```
public class DataReaderClient {
  private ApplicationContext ctx = null;
  private Reader reader =  null;
  public DataReaderClient() {
    ctx = new ClasspathXmlApplicationContext("reader-beans.xml");
  }
  public String getData() {
    reader = (Reader) ctx.getBean("fileReader");
    reader.fetchData();
  }
  public static void main(String[] args) {
    DataReaderClient client = new DataReaderClient();
    System.out.println("Data:"+client.getData());
  }
}
```

So, there are couple of notable things in the client program: a new variable referring to `ApplicationContext`. This is then assigned an instance of `ClasspathXmlApplicationContext` passing an XML file to the constructor of the class of its instantiation.

These two bits are the key to using Spring Framework. The instantiation of the `ApplicationContext` creates the container that consists of the objects defined in that XML file. I will discuss the framework fundamentals later in the chapter, but for now, let's continue with our reader example.

After creating the client class, create an XML file that consists of definitions of our `FileReader`. The XML file is shown below:

```
<bean name="fileReader" class="com.oreilly.justspring.ch1.FileReader"
    <constructor-arg value="src/main/resources/myfile.txt"/>
</bean>
```

The purpose of this XML file is to create the respective beans and their relationship. This XML file is then provided to the `ApplicationContext` instance, which creates a container with these beans and their object graphs along with relationships. The Spring container is simply a holder of the bean instances that were created from the XML file. An API is provided to query these beans and use them accordingly from our client application.

The `ctx = new ClasspathXmlApplicationContext("reader-beans.xml")` statement creates this container of beans defined in the *reader-beans.xml*. In our XML file, we have defined a single bean: FileReader. The bean was given an unique name: `fileReader`. Once the container is created, the client will have to use an API provided by the Context in order to access all the beans that were defined in the XML file.

For example, using the API method `ctx.getBean("fileReader")`, you can access the respective bean instance. That's exactly what we're doing in our client, as shown below:

```
reader = (Reader) ctx.getBean("fileReader");
```

The bean obtained is a fully instantiated `FileReader` bean, so you can invoke the methods normally: `reader.fetchData()`.

So, to wrap up, here are the things that we have done to make our program work without dependencies using Spring:

- We created a concrete `Reader` implementation, the `FileReader` as a simple POJO.
- We created the XML file to configure this bean.
- We then created a container with this bean in our client that loads it by reading the XML file.
- We queried the container to obtain our bean so we can invoke the respective methods.

Simple and Straightforward, eh?

Currently the client will always be injected with a type of `Reader` defined in configuration. One last thing you can do to improve your program is to create a service layer. However, if we introduce a service that would be glued to the client rather than the `Reader`, it is the desirable solution. Let's do this by creating a service.

Creating ReaderService

The `ReaderService` is an interface between the client and the `Reader`s. It abstracts away the implementation details and the `Reader` interface from the client. The client will only have knowledge of the service; it will know nothing about where the service is going to return the data. The first step is to write the service:

```
public class ReaderService {
  private Reader reader =  null;
  public ReaderService (Reader reader) {
    this.reader = reader;
  }
  private String fetchData() {
    return reader.read();
  }
}
```

The service has a class variable `of Reader` type. It is injected with a `Reader` implementation via the constructor. It has just one method—`fetchData()`—which delegates the call to respective implementation to return the data.

Wire the `ReaderService` with appropriate Reader in our *reader-beans.xml* file:

```
<bean name="readerService" class="com.oreilly.justspring.ch1.ReaderService">
    <constructor-arg ref="fileReader" />
</bean>
```

```
<bean name="fileReader" class="com.oreilly.justspring.ch1.FileReader"
    <constructor-arg value="src/main/resources/myfile.txt"/>
</bean>
```

When this config file is read by the Spring's `ApplicationContext`, the `ReaderService` and `FileReader` beans are instantiated. However, as the `ReaderService` has a reference to fileReader (`constructor-arg ref="fileReader"`), the `fileReader` is instantiated first and injected into `ReaderService`.

The modified client that uses `ReaderService` is given below:

```
public class DataReaderClient {
  private ApplicationContext ctx = null;
  private ReaderService service =  null;
  public DataReaderClient() {
    ctx = new ClasspathXmlApplicationContext("reader-beans.xml");
  }
  public String getData() {
    service = (ReaderService) ctx.getBean("readerService");
    service.fetchData();
  }
  public static void main(String[] args) {
    DataReaderClient client = new DataReaderClient();
    System.out.println("Data:"+client.getData());
  }
}
```

The notable thing is that the client will only have knowledge of the service—no `Reader`s whatsoever. If you wish to read data from a database, no code changes are required except config changes as shown below:

```
<bean name="readerService" class="com.oreilly.justspring.ch1.ReaderService">
  <property name="reader" ref="databaseReader"/>
  <!--
    <property name="reader" ref="fileReader"/>
    <property name="reader" ref="ftpReader"/>
  -->
</bean>
<bean name="databaseReader" class="com.oreilly.justspring.ch1.DatabaseReader"
  <property name="dataSource" ref="mySqlDataSource" />
</bean>
<bean name="ftpReader" class="com.oreilly.justspring.ch1.FTPReader"
  <property name="ftpHost" value="oreilly.com" />
  <property name="ftpPort" value="10009" />
 </bean>
<bean name="fileReader" class="com.oreilly.justspring.ch1.FileReader"
 ...
</bean>
```

We have defined all the `Reader` beans in the above configuration. The `readerService` is given a reference to the respective `Reader` without having to make any code change!

Injection Types

Spring allows us to inject the properties via constructors or setters. While both types are equally valid and simple, it's a matter of personal choice in choosing one over the other. One advantage to using constructor types over setters is that we do not have to write additional setter code. Having said that, it is not ideal to create constructors with lots of properties as arguments. I detest writing a class with a constructor that has more than a couple of arguments!

Constructor Type Injection

In the previous examples, we have seen how to inject the properties via constructors by using the `constructor-arg` attribute. Those snippets illustrate the constructor injection method. The basic idea is that the class will have a constructor that takes the arguments, and these arguments are wired via the config file.

See `FtpReader` shown below:

```
public class FtpReader implements Reader {
  private String ftpHost = null;
  private int ftpPort = null;
  // Constructor with arguments
  public FtpReader(String host, String port) {
    this.ftpHost = host;
    this.ftpPort = port;
  }
  ...
}
```

The `host` and `port` arguments are then wired using `constructor-arg` attributes defined in the config file:

```
<bean name="ftpReader" class="com.oreilly.justspring.ch1.FtpReader"
  <constructor-arg value="oreilly.com" />
  <constructor-arg value="10009" />
</bean>
```

You can set references to other beans, too. For example, the following snippet injects a reference to FtpReader into the ReaserService constructor:

```
<bean name="readerService" class="com.oreilly.justspring.ch1.ReaderService">
    <constructor-arg ref="ftpReader" />
</bean>
```

Setter Type Injection

In addition to injecting the dependent beans via constructors, Spring also allows them to be injected using setters, too. We will modify the `ReaderService` that can have the `FileReader` dependency injected using a setter. In order to use the setter injection, we have to provide setters and getters on the respective variables.

So, in our ReaderService class, create a variable of Reader type and a matching setter/getter for that property. The constructor is left empty as the properties are now populated using the setters. You should follow the normal bean conventions when creating setters and getters. Modified ReaderService is given below:

```
public class ReaderService {
  private Reader reader =  null;
  public ReaderService() { /* empty constructor */}
  public void setReader(Reader reader) {
    this.reader = reader;
  }
  public Reader getReader() {
    return reader;
  }
}
```

The notable change to the previous version is the omission of constructor. Instead, the setter and getter of the Reader variable will set and access the object. We should also refactor our XML file:

```
<bean name="readerService" class="com.oreilly.justspring.ch1.ReaderService" >
  <property name="reader" ref="fileReader" />
</bean>
<bean name="fileReader" class="com.oreilly.justspring.ch1.FileReader" >
 ...
</bean>
```

The notable change is to create a property called reader and set it with a reference to fileReader. The framework will check the ReaderService for a reader property and invokes setReader by passing the fileReader instance.

Mixing Constructor and Setter

You can mix and match the injection types, too. The revised FileReader class listed below has a constructor as well as a few other properties. The componentName is initialized using constructor, while fileName is set via setter.

```
<bean name="fileReader" class="com.oreilly.justspring.ch1.FileReader"
  <constructor-arg componentName="TradeFileReader" />
  <property name="fileName" value="src/main/resources/myfile.txt" />
</bean
```

Although mixing and matching the injection types is absolutely possible, I would recommend sticking one or the other of them, rather than both, to avoid complicating matters.

Property Files

I will show one more good point before wrapping up the chapter. When defining the FileReader or FtpReader, we have set the hard-coded properties in the XML file. Is there a way that we can resolve properties mentioned in the config file to be abstracted away?

If I need to change the properties from one environment to another, I will have to modify this XML file. This is definitely not a good practice. Spring gives us another way of injecting these properties, too. Let's see how we can do this.

Create a property file called `reader-beans.properties` and add the properties and their values:

```
#property=value
file-name="/src/main/resources/myfile.txt"
ftp-host = "oreilly.com"
ftp-port="10009"
```

Edit the *reader-beans.xml* file to add the Framework's class named `PropertyPlaceholderConfigurer`. This class has a property called location, which should be pointing to your properties file:

```
<bean id="placeholderConfig"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location" value="classpath:reader-beans.properties" />
</bean>
```

This bean can pick up the property file as long as it is present in the classpath.

The next step is to parameterize the `FileReader` property:

```
<bean name="fileReader" class="com.oreilly.justspring.ch1.FileReader"
 <property name="fileName" value="${file-name}" />
</bean>
<bean name="ftpReader"class="com.oreilly.justspring.ch1.FtpReader"
 <property name="ftpHost" value="${ftp-host}" />
 <property name="ftpPort" value="${ftp-port}" />
</bean>
```

The `${file-name}` resolves to the name-value pair defined in the `reader-beans.properties` file. So do the `${ftpHost}` and `${ftpPort}` properties.

Summary

This chapter introduced the Spring framework from the 10,000-foot view. We have seen the problem of object coupling and how the framework solved the dependency issues. We also glanced at framework's containers and injection types. We have scratched the surface of the framework's usage, leaving many of the fundamentals to the upcoming chapters.

We are going to see the internals of the framework in depth in the next chapter.

CHAPTER 2

Spring Beans

We saw the bare-minimum basics of Spring Framework in the last chapter. We worked with new things such as beans, bean factories, and containers. This chapter will explain them in detail. It discusses writing beans, naming conventions, how they are wired into containers, etc. This chapter forms the basis to understanding the details of the Spring Framework in depth.

Introduction to Beans

For Spring, all objects are beans! The fundamental step in the Spring Framework is to define your objects as beans. Beans are nothing but object instances that would be created by the spring framework by looking at their class definitions. These definitions basically form the configuration metadata. The framework then creates a plan for which objects need to be instantiated, which dependencies need to be set and injected, the scope of the newly created instance, etc., based on this configuration metadata.

The metadata can be supplied in a simple XML file, just like in the first chapter. Alternatively, one could provide the metadata as Annotation or Java Configuration.

Configuring using XML

Let's define a bean with a name `myBean` that corresponds to a class `com.oreilly.justspring.ch2.MyBean`. The `MyBean` expects a `String` value as a constructor argument. It also defines two properties, `property1` and `property2`.

Example 2-1 shows the simple XML file.

Example 2-1.

```
<?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-3.0.xsd">

  <bean name="myBean" class="com.oreilly.justspring.ch2.MyBean">
    <constructor-arg value="MyConstructorArgument"/>
    <property name="property1" value="value1"/>
    <property name="otherBean" ref="myReferenceBean"/>
  </bean>
</beans>
```

The topmost node declares `<beans>` as your root element. All bean definitions would then follow using a `<bean>` tag. Usually, the XML file consists of at least one bean. Each bean definition may contain sets of information, most importantly the name and the class tags. It may also have other information, such as the scope of the bean instantiated, the dependencies, and others. Basically, when the config file is loaded at runtime, the framework would pick up the definitions and create the instance of `MyBean`. It then gives a name as `myBean`. The developer should use the name as an API starting point to query the bean instance.

You can split the bean definitions across multiple files. For example, you create all the beans that deliver the business functions in a file called *business-beans.xml*, the utility beans in *util-beans.xml*, data access beans in *dao-beans.xml*, etc. We will see how to instantiate the Spring container using multiple files later in the chapter. Usually, I follow the convention of creating the files using two parts separated by a hyphen. The first part usually represents the business function, while the second part simply indicates that these are spring beans. There is no restriction on the naming convention, so feel free to name your beans as you wish.

Each bean should either have a name or id field attached to it. You can create the beans with neither of these things, making them anonymous beans (which are not available to query in your client code). The name and id fields both serve the same purpose, except that the id field corresponds to XML spec's id notation. This means that checks are imposed on the id (for example, no special characters in the id value, etc.). The name field does not attract any of these restrictions.

The class field declares the fully qualified name of the class. If the instantiation of the class requires any data to be initialized, it is set via properties or a constructor argument. As you can see in Example 2-1, the `MyBean` object is instantiated with both types: constructor argument and property setters. The value fields can be simple values or references to other beans. A `ref` tag is used if a the bean needs another bean, as is seen for `otherBean`.

You can name the bean as you wish. However, I would suggest sticking to camelCase class name, with the first letter being lowercase. So, myBean suits well, as indicated in the above example.

Creating Beans

The beans are the instances wired together to achieve an application's goal. Usually in a standard Java application, we follow a specific life cycle of the components, including their dependencies and associations. For example, when you start a main class, it automatically creates all the dependencies, sets the properties, and instantiates the instances for your application to progress. However, the responsibility of creating the dependency and associating this dependency to the appropriate instance is given to your main class, whereas in Spring, this responsibility is taken away from you and given to the Spring Container. The instances (aka beans) are created, the associations are established, and dependencies are injected by the Spring Framework entirely. These beans are then contained in a Container for the application to look up and act upon them. Of course, you would have to declare these associations and other configuration metadata either in an XML file or provide them as Annotations for the Framework to understand what it should do.

Life Cycle

The Spring Framework does quite a few things behind the scenes. The life cycle of a bean is easy to understand, yet different from the life cycle exposed in a standard Java application. In a normal Java process, a bean is usually instantiated using a new operator. The Framework does a few more things in addition to simply creating the beans. Once they are created, they are loaded into the appropriate container (we will learn about containers later in this chapter). They are listed below:

- The framework factory loads the bean definitions and creates the bean.
- The bean is then populated with the properties as declared in the bean definitions. If the property is a reference to another bean, that other bean will be created and populated, and the reference is injected prior to injecting it into this bean.
- If your bean implements any of Spring's interfaces, such as `BeanNameAware` or `BeanFactoryAware`, appropriate methods will be called.
- The framework also invokes any `BeanPostProcessor`'s associated with your bean for pre-initialzation.
- The init-method, if specified, is invoked on the bean.
- The post-initialization will be performed if specified on the bean.

We will discuss these points in the coming sections.

Look at the following XML code snippet:

Example 2-2. `FileReader` without a dependency

```
<bean name="fileReader" class="com.oreilly.justspring.ch2.FileReader">
  <property name="fileName" value="/opt/temp/myfile.txt"/>
</bean>
```

When the factory reads the definition, it creates the class using the new operator (in reality, the bean is instantiated using Java Reflection). After the bean is created, the property `fileName` is injected. In this case, a setter called `setFileName` is invoked and given a value of `/opt/temp/myfle.txt` as an argument. The bean is now instantiated and ready to be used.

However, if the `FileReader` bean has a dependency on another bean, the other bean will be created and instantiated. See Example 2-3. The `FileReader` has to be injected with a location object.

Example 2-3. `FileReader` with a dependency

```
<bean name="fileReader" class="com.oreilly.justspring.ch2.FileReader">
  <property name="location" ref="fileLocation"/>
</bean>
```

The property `location` references another bean called `fileLocation`. The bean definition is given in Example 2-4.

Example 2-4. `Location` definition

```
<bean name="fileLocation" class="com.oreilly.justspring.ch2.Location">
  <property name="fileName"  value="myfile.txt"/>
  <property name="filePath"  value="/opt/temp"/>
</bean>
```

The order of creation is important for Spring. After digesting the configuration metadata, Spring creates a plan (it allocates certain priorities to each bean) with the order of beans that needs to be created to satisfy dependencies. Hence, the `Location` object is created first, before the `FileReader`. If Spring encounters any exception while creating `Location` object, it will fail fast and quit. It does not create any further beans and lets the developer know why it won't progress further.

Method Hooks

Spring Framework provides a couple of hooks in the form of callback methods. These methods provide opportunity for the bean to initialize properties or clean up resources. There are two such method hooks: `init-method` and `destroy-method`.

init-method

When the bean is created, you can ask Spring to invoke a specific method on your bean to initialize. This method provides a chance for your bean to do housekeeping stuff and to do some initialization, such as creating data structures, creating thread pools, etc. You have to declare the method in the XML file as shown in Example 2-5.

Example 2-5. `init-method` declaration

```
<bean name="fileReader" class="com.oreilly.justspring.ch2.FileReader" init-method="init">
```

The `FileReader` class with `init-method` is provided in Example 2-6.

Example 2-6. `FileReader` with init-method

```
public class FileReader implements Reader {
    private List<Location> locations = null;
    // This method is called give us opportuniting to custom initialize
    public void init(){
       locations = new ArrayList<Locations>();
    }
}
```

Once the `FileReader` is created, its `init-method` (in this case, `init`) as declared in the config is invoked. The `init` method in this case creates a List of `Location`s.

destroy-method

Similar to the initialization, framework also invokes a destroy method to clean up before destroying the bean. Framework provides a hook with the name `destroy-method`, as shown below:

Example 2-7. `FileReader` with `destroy-method` method

```
public class FileReader implements Reader {
    private List<Location> locations = null;
    // This method is invoked by the Spring Framework before destroying the bean
    public void cleanUp(){
        locations = null;
    }
}
```

You should refer the `cleanUp` method as your destroy method in the XML declaration as shown below.

```
<bean name="fileReader" class="com.oreilly.justspring.ch2.FileReader"
destroy-method="cleanUp">
```

When the program quits, the framework destroys the beans. During the process of destroying, as the config metadata declares `cleanUp` as the destroy method, the `cleanUp` method is invoked. This gives the bean a chance to do some housekeeping activities.

Bean Post Processors

Spring provides a couple of interfaces that your class can implement in order to achieve the bean initialization and housekeeping. Those interfaces are `InitializingBean` or `DisposableBean`, which has just one method in each of them. Example 2-8 shows the `Connection` bean implementing these interfaces. The `InitializingBean`'s `afterPropertiesSet` method is called so the bean gets an opportunity to initialize. Similarly, the `DisposableBean`'s `destroy` method is called so the bean can do housekeeping when the bean is removed by the Spring.

Example 2-8. `Connection` class implementing Spring's post .processors

```
public class Connection implements InitializingBean, DisposableBean {
  private ObjectName objectName;
  //InitializingBean's method implementation
  public void afterPropertiesSet(){
    connection.registerToJmx(objectName);
  }
  //DisposableBean's method implementation
  public void destroy(){
    connection.unregisterFromJmx(objectName);
  }
}
```

The downside of using these interfaces is that we are locking our implementation to Spring's API. I would not recommend this; however, it is your choice. The same functionality can be achieved using the init-method and destroy-method (see section 3.1), so why lock to vendor unnecessarily?

Bean Scopes

Did you wonder how many instances will be created when the container is queried for your bean? If you have a case where you have one and only one bean (such as a service or a factory) irrespective of the number of times you call the container, does it return the same bean? Or for every call, do you want to have a new bean instantiated? How does Spring achieve this? Well, it turns out to be a simple config tag that dictates these types: `singleton` and `prototype`.

Singleton Scope

When you need one and only one instance of a bean, you should set the `singleton` property to true, as shown below:

```
<bean name="fileReader" class="com.oreilly.justspring.ch2.FileReader" singleton="true">
```

However, the default scope is always singleton. Hence, you can ignore the declaration of `singleton` property as shown below:

```
<bean name="fileReader" class="com.oreilly.justspring.ch2.FileReader">
```

Every time a `fileReader` is injected into another bean or queried using the `getBean()` method, the same instance is returned. Note that there is a subtle difference between the instance obtained using the Java Singleton pattern and Spring Singleton. Spring Singleton is a singleton per context or container, whereas the Java Singleton is per process and per class loader.

Prototype Scope

Prototype scope creates a new instance every time a call is made to fetch the bean. For example, we define a `Trade` object that gets created with a unique id every time it is instantiated.

```
public class Trade {
  private AtomicInteger uniqueId = -1;
  public void Trade() { }
  public int initId() {
    uniqueId = AtomicInteger.incrementAndGet();//RECHECK
  }
}
```

In order to achieve this functionality, all we have to do is set the `singleton` property to `false` in our config file. Spring would then create a new `Trade` object for every invocation.

Example 2-9. `Trade` bean definition

```
<bean name="trade" ref="com.oreilly.justspring.ch2.Trade" singleton="false"
init-method="initId"/>
```

You may notice no difference in the code base, whether using singleton or prototype. The difference is seen just in the declarative part. In order to obtain a new trade instance, the singleton tag is set to false on the Trade bean definition. This tag is the key to creating a new instance every time a request comes along.

Property Editors

I'm sure the question of what type of values should be set on the properties declared in the XML file might have crossed your mind. From the declaration, it sounds like the properties are just Strings or Java Primitives. What if we have other types of properties, such as object references, Lists, Custom classes, and other Collections? Well, it turns out that the Spring follows the Java Bean style property editor mechanism to resolve the actual types.

If the property is another bean reference (a dependency), the actual type of the bean is resolved when injecting the dependency.

Injecting Java Collections

Injecting an initialized collection such as a `List`, `Set`, or `Map` couldn't be any easier in Spring. There is a specific syntax to follow in the config file so the collection is initialized before injecting.

Using Properties

Consider the following snippet, which reads data from the `java.util.Properties` object (property-value pairs).

Example 2-10. Setting Properties type object

```
public class JMSSource{
  private Properties sourceProps = null;
  public void setProperties(Properties props){
    sourceProps = props;
  }
}
```

In order to set the values to your `sourceProps` properties variable, use Example 2-11 to set the values. The `sourceProps` is then populated using `<props>` and `<prop>` tags.

Example 2-11.

```
<bean name="jmsSource" class="com.oreilly.justspring.ch2.JMSSource">
  <props name="sourceProps">
    <props>
      <prop key="1">system_1</prop>
      <prop key="2">system_2</prop>
      <prop key="3">system_3</prop>
    </list>
  </props>
</bean>
```

Using Lists, Sets and Maps

In order to use `Lists`, use the following config metadata.

Example 2-12.

```
<bean name="jmsSource" class="com.oreilly.justspring.ch2.JMSSource">
  <props name="sourceProps">
    <list>
      <value>system_1</value>
      <value>system_2</value>
      <value>system_3</value>
    </list>
  </props>
</bean>
```

Of course, you will have a `java.util.List` variable defined in your class:

```
    public class JMSSource {
      private List sourceProps = null;
      public void setProperties(List props) {
      sourceProps = props;
      }
    }
```

Similarly, if you have to pump in `Set` values, swap the list tag with `<set>`, as shown in Example 2-13. Obviously, you cannot insert duplicate data in `Set` implementations.

Example 2-13.

```
<bean name="jmsSource" class="com.oreilly.justspring.ch2.JMSSource">
  <props name="sourceProps">
    <set>
      <value>u1</value>
      <value>u2</value>
    </set>
  </props>
</bean>
```

Swap the List type of the sourceProps with `Set` type:

```
private Set sourceProps = null;
```

When using `Map`s, on the class side, as expected, you need to use the `java.util.Map` implementation. However, when configuring, there are certain subtle differences. The element in a map is made of an `Entry` node, which has name-value pairs. Note that you can set bean references in your collections, too, as was shown for the property `uniqueKeyGen`.

```
<bean name="jmsSource" class="com.oreilly.justspring.ch2.JMSSource">
  <props name="sourceProps">
    <map>
      <entry key="threads">
        <value>10</value>
      </entry>
      <entry key="numberOfRetries">
        <value>3</value>
      </entry>
      <entry key="uniqueKeyGen">
        <ref bean="com.oreilly.justspring.ch2.JMSSourceKeyGen"/>
      </entry
    </map>
  </props>
</bean>
```

As expected, replace the List type with `Map` type.

```
private Map sourceProps = null;
```

Summary

This chapter discussed the Spring framework in detail. It explained the concept of beans and bean factories. We have also learned about the life cycle of the bean scopes and touched upon the property editors used in injecting Java Collections and other types of objects.

We will be discussing the containers and application contexts in the next chapter, which forms a crucial concept in putting the framework to work.

CHAPTER 3

Advanced Concepts

A Spring container is the backbone of the framework. A container is basically a pool of beans created in a memory space by the framework when the application starts up. An API provided by the framework exposes methods to query the container beans. We'll start the chapter by looking at the containers and different categories. We'll also look at details about advanced concepts such as AutoWiring.

Containers

Understanding containers is a very important task that needs to happen before you can start working with the framework. During this process, the beans are instantiated, the associations and relationships are created, all the relationships are satisfied, and dependencies are injected. All named beans are available for querying using an API. Note that beans are loaded lazily into the container. This means that unless the bean is requested by yours or another bean (as a part of dependency), the factory will not instantiate the bean. This strategy works the same for any container in Spring, except loading Singletons in the ApplicationContext container.

Spring containers primarily fall into two categories: Bean Factories and Application Contexts. The names are sort of misnomers, as they do not give any clue as to whether they're containers and what they do. The `BeanFactory` is a simple container supporting basic dependency injection, whereas `ApplicationContext` is an extension of `BeanFactory`, which has a few additional bells and whistles.

BeanFactory Container

This is the simplest of all types of containers Spring has provided. The factory implements the `org.springframework.beans.factory.BeanFactory` interface. The crux of this factory is that it creates and instantiates beans with all the dependent configurations. So, when you query a bean, it is obtained in your client as a fully functional instance, meaning all the associations and relationships have already been created. The bean factory implementor can also invoke the customized `init-method` and `destroy-method`s.

Spring provides a few of the implementations of `BeanFactory` out of the box, the most common being `XmlBeanFactory`. As the name suggests, it reads the configuration metadata from an XML file. Using the `XmlBeanFactory` in your client is very easy and straightforward (similar to what we've seen in our previous examples of using `ApplicationContext`). Example 3-1 shows the code to instantiate the factory. The constructor takes in an XML file passed into a `FileInputStream` object.

Example 3-1. `BeanFactory` usage

```
// Instantiate the factory with your beans config file
BeanFactory factory = XmlBeanFactory(new FileInputStream("trade-beans.xml"));

// Use the factory to obtain your TradeService bean
TradeService service = (TradeService) factory.getBean("tradeService");
```

The `BeanFactory` is usually preferred in small device applications such as mobile phones, etc., where the resources are limited. If you are writing any standard Java or JEE application, you would ideally ditch `BeanFactory` to go with `ApplicationContext` implementation.

ApplicationContext Container

Simply put, the extension of `BeanFactory` is the `ApplicationContext`, which provides application events as an added bonus. The `ApplicationContext` does extend the `BeanFactory` and hence all the functionality of `BeanFactory` is already embedded in the `ApplicationContext` container. Ideally, one can use this container straight away due to its extended functionality. Similar to `BeanFactory` implementations, there are three concrete implementations of `ApplicationContext`.

- `FileSystemXmlApplicationContext`: This container loads the definitions of the beans found in the XML file located at a specified file system location. You should provide the full path of the file to the constructor.
- `ClassPathXmlApplicationContext`: In this container, the XML config file is read from the classpath by the container. The only difference between the FileSystemXmlApplicationContext and this one is that this context does not require the full path of the file.
- `WebXmlApplicationContext`: This container loads the XML file with definitions of all beans from within a web application.

  We have already seen the usage of any of the `ApplicationContext` type containers in the first chapter. Let me recap the usage again here:

  Instantiating the `FileSystemXmlApplicationContext` takes the exact location of the beans file, as shown in Example 3-2.

Example 3-2.

```
ApplicationContext ctx =
new FileSystemXmlApplicationContext("/opt/temp/trade-beans.xml");
```

Whereas instantiating the `ClassPathXmlApplicationContext` expects the location of the beans file in the classpath, so we do not need to give the exact path. Example 3-3 shows this:

Example 3-3. Using `ClassPathXmlApplicationContext`

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("trade-beans.xml");
```

Instantiating Beans

Beans are instantiated by the framework in more than one way. We have already seen in previous chapters how the beans were instantiated using Constructors. You should ideally provide a constructor in your bean and corresponding metadata in the XML file. If the constructor takes any arguments, we should pass them as constructor-arg attributes in our config file. We are not going to discuss instantiating beans using constructors, as it would be repetitive. Instead, we'll look into two other types to instantiate them.

Using Static Methods

This type of instantiating is well suited if your classes have static factories for creating the object instances. The procedure is simple: write a class with a static method that would create an instance of that class or whatever you wish to do in that method. In your bean config, once you have declared the class attribute, define an attribute called `factory-method`. Let's see how we can create a `TradeService` using the static method. First, define the `TradeService` class (Example 3-4):

Example 3-4. `TradeService` client code snippet

```
public class TradeService {
  private static TradeService service = new TradeService();
  public static TradeService getTradeService(){
    return service;
  }
}
```

Declare the above service client in the XML file definition as shown in ??? on page 23.

Example 3-5. `TradeService` client XML config

```
<bean name="tradeService" class="com.oreilly.justspring.ch3.TradeService"
factory-method="getTradeService"/>
```

The `factory-method` attribute invokes the respective static method on the class. Note that the declaration does not mention the `factory-method` being static anywhere. However, it must be declared static if you wish to encounter no errors. Also note that the return type method is not imperative; it is only known if you see the implementation of that method in the declared class, unfortunately.

Using Factory Methods

What if you wish to create the instance using non-static methods? Well, Spring Framework does allow us to instantiate the beans using non-static factory methods, too. Although it's not straightforward, it's not hard to grasp either. The `EmployeeCreator` is a simple class that creates either employees or executives using two factory methods. See Example 3-6 for the code snippet.

Example 3-6. `EmployeeCreator` that has two factory methods

```
public class EmployeeCreator {
  public Employee createEmployee() {
  Employee emp = new Employee();
  return emp;
}

public Employee createExecutive() {
  Employee emp = new Employee();
  emp.setTitle("EXEC");
  emp.setBonusGrade("GRADE-A");
  return emp;
 }
}
```

Declare the bean config (Example 3-7) as shown below:

Example 3-7. `EmployeeCreator` metadata

```
<bean name="empCreator" factor-bean="employeeCreator" factory-method="createEmployee"/>
<bean name="execCreator" factor-bean="employeeCreator" factory-method="createExecutive"/>
<bean name="employeeCreator" class="com.oreilly.justspring.ch3.EmployeeCreator"/>
```

Did you notice that we do not include the class attribute at all? However, the `factory-bean` should refer to the actual bean on which these factory methods were declared. In our case, the reference is `employeeCreator`, however, we need to declare that bean, too. As you may have noticed, the factory methods are not static in the above implementations.

Bean Post Processors

Spring allows us to poke into the bean's life cycle by implementing bean post processors. We get a chance to alter the configuration or set custom values as the properties on the bean or implement sophisticated init processes. A `BeanPostProcessor` interface

is used to exploit this functionality. It has two callback methods: `postProcessBeforeInitialization` and `postProcessAfterInitialization`.

As the names indicate, the `postProcessBeforeInitialization` method is invoked just before calling your `init-method` or `afterPropertiesSet` method on the bean. Your bean is just about to be implemented, but you have been given a chance to do something before the final call! Similarly, the `postProcessAfterInitialization` method is called just after the initialization of the bean is completed.

We need to tell the framework that we're using post processors. If the container is BeanFactory, we have to invoke `addBeanPostProcessor` (ourProcessor) on the BeanFactory in order to set the post processor instance. However, if the container is `ApplicationContext`, defining in the config file is enough! The container automatically looks at the class to see if it's a post processor. It then invokes the appropriate methods during the instantiation of the bean. We should define the config as shown below:

```
<bean name="beanPostProcessor" class="com.oreilly.justspring.ch3.TradePostProcessor"/>
```

As you can see, you do not have to define the bean as a post processor anywhere in the config, the container will automatically find all the post processors and load them for further work.

Event Handling

Sometimes, you may want to react to an event that happened in the container so you can do some custom processing. I'm sure you must be wondering how to get notified upon a bean's initialization or when its context is refreshed. Also, if you wish to let other beans know that you have finished processing a huge file, for example, how can you notify? Spring provides a way to listen to the events that may allow you to react on the context. And yes, you can use Spring's framework to publish your custom events, too.

The `ApplicationContext` publishes certain types of events when loading the beans. For example, a `ContextStartedEvent` is published when the context is started and `ContextStoppedEvent` is published when the context is stopped. We can have our beans receive these events if we wish to do some processing on our side based on these events. We can also publish our own events, too.

Let's see the procedure involved in listening and publishing events.

Listening to Context Events

In order to listen to the context events, our bean should implement the `ApplicationListener` interface. This interface has just one method: `onApplicationEvent(ApplicationEvent event)`.

```
public class TradeContextEventListener implements ApplicationListener {
  public onApplicationEvent(ApplicationEvent event) {
```

```
    //handle the event here
  }
}
```

The next step is to bind the listener to the context. You would do this by declaring the bean in your config file:

```
<bean id="tradeCtxListener"
class="com.oreilly.justspring.ch3.TradeContextEventListener"/>
```

Spring context publishes the following types of events:

- `ContextStartedEvent`: This event is published when the `ApplicationContext` is started. The beans receive a start signal once the `ApplicationContext` is started. The activities such as polling to database or observing a file system can be started once you receive this type of event.
- `ContextStoppedEvent`: This is the opposite of the start event. This event is published when the `ApplicationContext` is stopped. Your bean receives a stop signal from the framework so you can do housekeeping if you wish.
- `ContextRefreshedEvent`: A refresh event is emitted when the context is either refreshed or initialized.
- `ContextClosedEvent`: This event occurs when the `ApplicationContext` is closed
- `RequestHandledEvent`: This is a web-specific event informing the receivers that a web request has been received.

Publishing Custom Events

It's a fairly simple task to publish custom events. First you have to create your event by extending `ApplicationEvent`. The `TradePersistedEvent`, for example, is emitted once the `TradePersistor` persists the trade. So let's define the `TradePersistedEvent` as shown below:

```
public class TradePersistedEvent extends ApplicationEvent{
  private Trade t = null;
  private persistedTime = null;
  private source = null;
  public TradePersisteEvent(String source, Trade t) { ... }
  ....
}
```

The `TradePersistor` should implement an interface called `ApplicationEventPublisherAware`. This interface lets the framework set the `ApplicationEventPublisher` instance on the class by calling the `setApplicationEventPublisher` method. The `ApplicationEventPublisher` has one method—`publish(ApplicationEvent event)`—which is used to publish events.

```
public class TradePersistedEventPublisher implements ApplicationEventPublisherAware{
  private ApplicationEventPublisher tradeEventPublisher;
  ...
```

```
    public void setApplicationEventPublisher(ApplicationEventPublisher Publisher) {
        this.tradeEventPublisher = Publisher;
    }
    public void publish(TradePersistedEvent event) {
      tradeEventPublisher.publish(event);
    }
}
```

Declare `TradePersistedEventPublisher` in the XML file as a normal bean. The container can identify the bean as an event publisher because it implements the `ApplicationEventPublisherAware` interface. The container then sets the publisher automatically on the instance of `TradePersistedEventPublisher`. The `TradePersistedEventPublisher` can then use the injected `ApplicationEventPublisher` to publish the `TradePersistedEvent`.

```
public class TradePersistedEventListener implements ApplicationListener {
  public onApplicationEvent(TradePersistedEvent event) {
    //handle the event here
  }
}
```

Receiving Custom Events

Receiving the `TradePersistedEvent` is straightforward. Create a class that extends the `ApplicationListener` and handle the `onApplicationEvent` method that receives `TradePersistedEvent` object.

```
public class TradePersistedEventListener implements ApplicationListener {
  public onApplicationEvent(TradePersistedEvent event) {
    //handle the event here
  }
}
```

Single Threaded Event Model

One important thing to keep in mind when working with Spring events handling is that Spring's event handling is *single-threaded*. It is primarily synchronous in nature. That is, if an event is published, until and unless all the receivers get the message, the processes are blocked and the flow will not continue. This proves to be disadvantageous if you have multiple listeners listening for an event. So the single-thread model hampers the performance of the application. Hence, care should be taken when designing your application if event handling is to be used.

Auto Wiring

When creating a bean, we used to set the properties of the bean either using `property` or `constructor-arg` attributes. However, Spring has an advanced concept of autowiring the relationships and dependencies. This means that you don't have to explicitly mention the properties and their values, but setting autowire property to a value allows the

framework to wire them with appropriate properties. There are fundamentally three variations of autowiring, explained in the following sections.

Autowiring byName

When autowiring `byName` is enabled, the framework tries to inject the dependencies by matching the names of the beans. You have to set the value of `autowire` to `byName` when defining the bean in the config. The container looks at the properties of the respective bean on which autowiring `byName` is set. It then tries to match with the beans defined by the same name in the config file. If matches are found, it will inject those beans straight away; otherwise, it will throw exceptions

For example, see the definition of the `TradeReceiver` class below:

```
public class TradeReceiver{
  private Persistor tradePersistor = null;
  private Transformer tradeTransformer = null;
  //setters and getters of the above two variables
}
```

The `TradeReceiver` depends on two other beans: the `TradePersistor` and `TradeTransformer`. Our usual way is to define all three beans in the XML config file, then pass the references of persistor and transformer to TradeReceiver. However, with autowiring, you don't have to go this far.

In order to enable autowiring, we should first tell the framework to do so. We use a property called `autowire` in order to tell the container what beans it should wire automatically. See the config below for all the three beans.

```
<bean name="tradeReceiver" class="com.oreilly.justspring.ch3.TradeReceiver"
autowire="byName"/>
<bean name="tradePersistor" class="com.oreilly.justspring.ch3.TradePersistor"/>
<bean name="tradeTransformer" class="com.oreilly.justspring.ch3.TradeTransformer"/>
```

Surprisingly, we did not declare any properties such as `tradePesistor` or `tradeTransformer` for `tradeReceiver`! How does this work then? Well, the attribute `autowire="byName"` does the magic for you behind the scenes. In our `TradeReceiver`, we defined two properties named `tradePersistor` and `tradeTransformer`. Can you see we also defined two beans with the same names in our config? Hence the `tradeReceiver` is injected with the matching beans.

Autowiring byType

Similar to `byName`, we need to set the autowire property to `byType` in order to enable this type of autowiring. In this case, instead of looking for a bean with the same names, the container searches for the same types. Taking the same example of `TradeReceiver`, setting the `autowire="byType"` tells the container that it should look for a bean of type `TradePersistor` and another one with a type of `TradeTransformer`. If the container finds

the appropriate types, it will inject them into the bean. However, if it finds more than one bean with the same type defined in the config, a fatal exception is thrown.

Autowiring by Constructor

You may have already gotten the gist of using autowire by constructor. If a bean has a constructor that takes an argument of another bean type, the container looks for that reference and injects it.

For example, we define a `TradePersistor` class with a single constructor that takes a `datasource` object,

```
public class TradePersistor {
  public TradePersistor (DataSource datasource){ ..}
}
```

If we enable autowiring by constructor, the container looks for an object of type `DataSource` and injects into the `TradePersistor` bean. You enable autowiring by constructor as shown below:

```
<bean name="tradePersistor" class="com.oreilly.justspring.ch3.TradePersitor"
autowire="constructor"/>
```

Mixing Autowiring with Explicit Wiring

You can get the best of both worlds using auto and explicit wiring. Any ambiguities encountered while autowiring can be dealt with using explicit wiring. For example, the `TradePersitor` can be injected explicitly while the `TradeTransformer` can be wired automatically using `byName` variation.

```
<bean name="tradeReceiver" class="com.oreilly.justspring.ch3.TradeReceiver"
autowire="byName">
  <property name="tradePersistor" ref="tradePersistor"/>
</bean>
<bean name="tradePersistor" class="com.oreilly.justspring.ch3.TradePersistor"/>
<bean name="tradeTransformer" class="com.oreilly.justspring.ch3.TradeTransformer"/>
```

Summary

This chapter completes our journey into core Spring. It explains the fundamentals of Containers and their usage. It then delves into using autowiring beans where you do not have to set properties on the beans. It then explains various ways of instantiating the beans, such as static methods or factory methods. We have also seen event handling supported by the framework.

One of the important aspect of Spring is its support for enterprise features such as Spring JMS and Database. The next chapter explains the simplification Spring has brought into the Java messaging world.

CHAPTER 4

Spring JMS

The Java Messaging Service (JMS) API, which came into existence a decade ago, was an instant hit in the area of distributed messaging and enterprise applications. The demand for easier and standardized integration with the Messaging Middleware was the drive for creating this technology. Although the API is straightforward and relatively easy, Spring has taken a step forward in creating an even easier framework around it. This chapter will introduce you to the JMS basics. It then explains Spring's take on JMS and how it simplified developers' lives even further.

Two-Minute JMS

JMS provides a mechanism of communication between applications without being coupled. The applications can talk to each other via the JMS provider without even knowing who is on the other side of the fence.

For example, a `PriceQuote` service can publish price quotes onto a channel expecting someone to receive them. However, the producer is not aware of any consumers that would consume these quotes. Similarly, the consumers will not have any clue about the source of data or the producers. In a way, they are hidden behind the walls, just interacting via destinations. Producers and consumers don't have to bother about each other's existence as long as they know their expected messages are delivered or received, respectively.

There are three important moving pieces in JMS: the Provider, the Producer, and the Consumer. The provider is the central piece, while the other two are either data providers or consumers. The architecture is based on hub and spoke; that is, the server maintains a central ground like a hub while producers and consumers act like the spokes.

The overall responsibility in this architecture is taken by the provider piece. The provider will make sure all the consumers receive the messages, while at the same time, all providers are able to publish the messages. Pick up any book on the subject to understand the JMS in detail.

Messaging Models

In JMS, there are primarily two messaging models: one is point-to-point (or P2P, as it is sometimes called) and the other is publish-subscribe (Pub/Sub). There are specific use cases where you should use these models. Although both are different fundamentally in delivery and reception mechanisms, a unified API is used to access both the models.

Point-to-Point Messaging

In point-to-point mode, a message is delivered to a single consumer via a destination. A publisher publishes a message onto a destination, while a consumer consumes the message off that destination. When a message is published in P2P mode to a queue, one and only one consumer can receive the message. Even if there are hundreds of consumers connected to that queue, still only one consumer will get that message delivered. Of course, you never know who's the lucky winner though!

The destination point-to-point model is called `Queue.`

Pub/Sub Messaging

On the other hand, if a message is published to a destination, there could be several subscribers each receiving a copy of the message. The publisher obviously publishes the message once. The subscribers interested will listen to the same destination to consume that message. As mentioned earlier, the JMS Provider will make sure each of the subscribers receives a copy of the message.

The destination in a Pub/Sub model is called a `Topic`.

Spring JMS

The Spring framework has simplified the JMS API quite a bit. There's a simple software pattern to follow in order to understand the Spring JMS usage. Before we go into code examples, let me explain the high-level classes and their usage.

If you understand just one class in the Spring JMS framework, you more or less have a grip on Spring JMS. That single class is `JmsTemplate`. It is heavily used in Spring JMS for both message consumption and production. However, it is not used for a particular use case: the asynchronous message consumption. I will discuss this case a bit later in the chapter. First, let's start digging into the `JmsTemplate` class.

Mother of All: the JmsTemplate class

You can use the `JmsTemplate` class for both sending and receiving the messages. The template class hides all the plumbing needed for connecting to a provider and publishes or receives messages. It has to be wired with a few properties, of which the `Connection`

Factory is a must. There are other properties that are required for more features (such as defaultDestination and receiveTimeout parameters), which we will see in the following sections. Basically, it uses callbacks such as MessageCreator for the message creation, SessionCallback for associating with a Session, and ProducerCallback for creating a message producer.

The JmsTemplate instance is a thread-safe class, so it can be shared across different classes without having to worry about session corruption. There are two ways of creating or instantiating JmsTemplate. One is to create the class using a new operator in your code and passing a ConnectionFactory instance to it. This type is shown below:

```
public class TradePublisher {
  private JmsTemplate jmsTemplate = null;
  private ConnectionFactory connectionFactory = null;
  public void setConnectionFactory(ConnectionFactory connectionFactory) {
    this.connectionFactory = connectionFactory;
    //here as you got the ConnectionFactory injected, create the JmsTemplate
    jmsTemplate = new JmsTemplate(connectionFactory);
  }
  // access the template when publishing the message
  public void publishTrade(Trade t) {
    jmsTemplate.send(..);
  }
}
```

The only requirement in this case is to wire the ConnectionFactory to the publisher in your config so it can be injected when the bean is created. The ConnectionFactory is the gateway to the JMS Provider. Of course, you need to instantiate the JmsTemplate object using this connection factory.

The other way is to wire the template with a connection factory and inject into your bean. This is all done in the config. Your bean will have a reference to JmsTemplate via a setter method.

```
<bean id="tradePublisher" class="com.oreilly.justspring.jms.TradePublisher">
  <property name="jmsTemplate" ref="jmsTemplate"/>
  ...
</bean>
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>
<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
 ...
</bean>
```

The JmsTemplate is ideally injected into a related class that requires publishing or receiving messages from a JMS server. For sending messages, it exposes three methods:

```
send(MessageCreator msgCreator)
send(String destinationName, MessageCreator msgCreator)
send(Destination destinationName, MessageCreator msgCreator)
```

The first one assumes that messages should end up in a default destination, while the next two specify the destination specifically. The second argument, `MessageCreator`, is a callback used to create the JMS `Message`.

In the JMS world, all messages should be declared as one of the predefined five types before attempting to publish or receive them. The five types are `TextMessage`, `BytesMessage`, `ObjectMessage`, `StreamMessage`, and `MapMessage`.

Publishing Messages

Let's develop an example to understand the mechanics.

The `TradePublisher` is the publishing component in the workflow that would be invoked when the `Trade` is ready for publication. The aim is to publish the already constructed `Trade` onto a JMS destination so the parties interested would consume and act on them. Example 4-1 shows the code:

Example 4-1. `TradePublisher`

```
public class TradePublisher {
  private JmsTemplate jmsTemplate = null;
  private String destinationName = null;
  public void publish(final Trade t) {
    getJmsTemplate().send(getDestinationName(), new MessageCreator() {
    @Override
    public Message createMessage(Session msg) throws JMSException {
    TextMessage m = session.createTextMessage();
    m.setText("Trade Stuff: "+t.toString());
    return m;
  }
 });
}
//setters and getters for the jmsTemplate and destinationName variables
}
```

The `TradePublisher` has two instance variables: `jmsTemplate` and `destinationName`. The `JmsTemplate` is used to publish the messages. The `destinationName` defines the location (queue or topic) where the messages should be sent.

The `TradePublisher` instance is injected with a `JmsTemplate` instance and a `destinationName` value at runtime. The publish method on the `TradePublisher` instance is invoked when a client wishes to publish a `Trade` message. One important feature you should note concerns the MessageCreator interface. Spring uses the implementation of this interface to create a new `Message`. The template class expects a new instance of this callback with `createMessage()` implemented:

```
public Message createMessage(Session session) throws JMSException {
  TextMessage m = session.createTextMessage();
  m.setText("Trade Stuff: "+t.toString());
  return m;
}
```

The appropriate message is created using the session, in this instance a `TextMessage`.

Before injecting the `JmsTemplate` into `TradePublisher`, it is configured in the Spring's XML (for this example, it is *jms-beans.xml*) file. In order to work as expected, the template has a variable called `connectionFactory` that needs to be defined and referenced.

See the snippet below showing the configuration of `JmsTemplate`:

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

In the config file, you need to define the `ConnectionFactory`, too. The connection factory is specific to individual JMS Providers. I am going to use ActiveMQ as the JMS Provider for the rest of the chapter, but you can use any other providers that are JMS-compliant (so the code should work without tweaking for each provider).

The ActiveMQ connection factory requires a `brokerUrl` property to be set:

```
<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL">
    <value>tcp://localhost:61616</value>
  </property>
</bean>
```

The last piece is the declaration of the `TradePublisher` bean itself in the config file.

```
<bean id="tradePublisher" class="com.oreilly.justspring.jms.TradePublisher">
  <property name="jmsTemplate" ref="jmsTemplate"/>
  <property name="destinationName" value="justspring.jms.testQueue"/>
</bean>
```

As you may have guessed, the two properties `jmsTemplate` and `destinationName` were wired.

Write a simple test client that creates a Spring container by loading the *jms-beans.xml* config file.

```
public class TradePublisherTest {
  …
  public void init(){
    ctx = new ClassPathXmlApplicationContext("jms-beans.xml");
    pub = (TradePublisher) ctx.getBean("tradePublisher");
  }

  public static void main(String[] args) {
    TradePublisherTest test = new TradePublisherTest();
    test.init();
    test.pub.publish(new Trade());
    System.out.println("Messages published");
 }
}
```

The client gets a hold of the `TradePublisher` instance to invoke the publish method with a new `Trade` object. The `TradePublisher` was already injected with the dependencies such as `jmsTemplate` and `destinationName` (see the config declaration).

Now, run the ActiveMQ server. The server is started and running on my local machine (hence localhost) on a default port 61616.

Run the client and if all is set correctly, you should see a message landing up in the ActiveMQ destination.

Sending Messages to Default Destination

If you wish to send the messages to a default destination, use the `JmsTemplate`'s `send(MessageCreator msgCreator)` method. However, you need to wire the default destination in the config file:

```
<bean id="defaultDestination" class="org.apache.activemq.command.ActiveMQQueue">
  <constructor-arg value="justspring.jms.testQueue2" />
</bean>
```

Then add a property `defaultDestination` in `JmsTemplate`:

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="defaultDestination" ref="defaultDestination" />
</bean>
```

Delete the `destinationName` variable from the `TradePublisher`, as it is now publishing messages to the default destination. See the code snippet below. Note that the `send` method does not have any reference to the destination.

```
public void publishToDefaultDestination(final Trade t) {
  getJmsTemplate().send(new MessageCreator() {
  @Override
  public Message createMessage(Session session) throws JMSException {
  TextMessage m = session.createTextMessage();
  m.setText("Trade Stuff: "+t.toString());
  return m;
 }
 });
}
```

Destination Types

By default, the `JmsTemplate` assumes that your messaging mode is point-to-point and hence a destination to be a `Queue`. However, if you wish to change this mode to pub/sub, all you are required to do is wire in a property called `pubSubDomain`, setting it to true. This way, you are expecting the messages to be published onto a `Topic`.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  ...
```

```
  <property name="pubSubDomain" value="true" />
</bean>
```

Receiving Messages

Using `JmsTemplate` makes consuming the messages simple. However, there are two modes in which you can receive messages: synchronous and asynchronous modes.

In synchronous mode, the thread that calls the receive method will not return, but waits indefinitely to pick the message. I strongly recommend not using this mode unless you have a strong case. Should you have no alternatives other than using synchronous receive method, use it by setting a value on timeout.

In the asynchronous mode, the client will let the provider know that it would be interested in receiving the messages from a specific destination or set of destinations. When a message arrives at the given destination, the provider checks the list of clients interested in the message and will send the message to that list of clients.

Receiving Messages Synchronously

See the `TradeReceiver` given below to receive a message (the instance has been injected with a `JmsTemplate` object).

```
public void receiveMessages() {
  Message msg = jmsTemplate.receive("justspring.jms.testQueue");
  System.out.println("Message Received: "+msg);
}
```

In the above code snippet, the receive method has been given a destination name in a string format.

The receive method waits until the queue holds at least a message. As explained earlier, the receive is a blocking call, which may waste CPU cycles if no message exists in the queue. Hence, use the `receiveTimeout` variable with the appropriate value. The `receiveTimeout` is an attribute on `JmsTemplate` that needs to be declared in config file, which is shown below:

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="receiveTimeout" value="2000" />
  ...
</bean>
```

In the above snippet, the `TradeReceiver` will timeout after two seconds if it does not receive any messages in that time period.

You can also receive messages from a `defaultDestination`, too. As we did earlier, what you have to do is wire the `jmsTemplate` with a `defaultDestination` property.

Receiving Messages Asynchronously

In order to receive the messages asynchronously, you have to do couple of things:

- Create a listener class that implements the `MessageListener` interface.
- Wire a `XXXContainer` in your spring beans XML file with a reference to your listener. We will talk about `XXXContainer`s in a minute.

So, the asynchronous client must implement JMS API's interface `MessageListener`. This interface has one method called `onMethod` that must be implemented by your class. The `TradeMessageListener`, for example, is a simple class that implements the `MessageListener`. The method would not do much except print out the message to the console.

```
public class TradeMessageListener implements MessageListener{
  @Override
  public void onMessage(Message msg) {
    System.out.println("TradeMessageListener:Message received:"+msg.toString());
 }
}
```

The second part is the task of wiring the containers. Don't confuse yourself with the ApplicationContext or BeanFactory containers. These containers are utility classes provided by the framework used in clients that are destined to receive messages. The containers are simple yet powerful classes that hide away all the complexities of connections and sessions. They are responsible for fetching the data from the JMS Provider and pushing it to your listener. It does this by having a reference to ConnectionFactory (and hence JMS Provider) and a reference to your listener class.

Now, let's looks at the workings in detail. Wire up the container and the listener as shown in the code snippet below. Define an instance of `DefaultMessageListenerContainer` and inject it with a `connectionFactory`, `destination`, and `messageListener` instances. Note that the `messageListener` class refers to your listener class.

```
<bean id="defaultListenerContainer"
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="defaultDestination" />
  <property name="messageListener" ref="tradeMessageListener" />
</bean>

<bean id="tradeMessageListener"
class="com.oreilly.justspring.jms.consumer.TradeMessageListener" />
```

Once the wiring is done, fire up your client, which loads up the above beans. It would start up your `messageListener` instance, which waits to receive messages from the JMS server.

Publish a message onto the `Destination` and you can see that message popping up at the `messageListener` client.

Spring Message Containers

We have seen the usage of `DefaultMessageListenerContainer` in the previous section. Spring provides three different types of containers for receiving messages asynchronously, including the `DefaultMessageListenerContainer`. The other two are `SimpleMessageListenerContainer` and `ServerSessionMessageListenerContainer`.

The `SimpleMessageListenerContainer` is basically the simplest of all and is not recommended for production use. On the other hand, `ServerSessionMessageListenerContainer` is one level higher than `DefaultMessageListenerContainer` in complexity as well as features. This is suited if you wish to work with JMS sessions directly. It is also used in the situation where XA transactions are required.

The `DefaultMessageListenerContainer` does allow you to participate in external transactions. It is well-suited for most of the applications, but obviously choose the appropriate one based on your application's requirement.

Message Converters

One of the requirements when publishing a message is to convert your domain object into five pre-defined JMS message types. You cannot simply publish or receive domain objects such as `Trade` or `Order`, even if they're serialized. So, if you wish to publish your domain objects, you need to convert them into the appropriate JMS Message type.

Spring framework comes in quite handy in doing this without having to sweat. The developer will not have to worry about the conversions. Let's see how the converters work.

You create a class that implements the `MessageConverter` interface. This interface has two methods: `fromMessage` and `toMesage` methods. As the name indicates, you implement these methods either to convert a JMS message to a domain object or vice versa.

The following code shows a typical converter used for `Trade` objects:

```
public class TradeMessageConverter implements MessageConverter {
  @Override
  public Object fromMessage(Message msg)
throws JMSException, MessageConversionException {
    Trade t = (Trade) ((ObjectMessage)msg).getObject();
    System.out.println("fromMessage: "+msg.toString());
    return t;
  }

  @Override
  public Message toMessage(Object obj, Session session)
throws JMSException,MessageConversionException {
    ObjectMessage objMsg = session.createObjectMessage();
    objMsg.setObject((Trade)obj);
    System.out.println("toMessage: "+objMsg.toString());
    return objMsg;
```

```
    }
}
```

The `TradeMessageConverter` class implements the `MessageConverter` interface. In the `fromMesage` method, the `Trade` object is grabbed from JMS `ObjectMessage`.

In `toMessage` method, use the passed-in session object to create an `ObjectMessage` and push the domain object using `setObject` method.

Once you have created the converter, there are couple of changes required. In the publisher and receiver code, you need to change the send and receive methods to `convertAndSend()` and `receiveAndConvert()` so the converter is used at publishing and receiving end.

The last thing you need to do is wire the converter, along with giving a reference of it to `JmsTemplate`:

```
<bean id="tradeMessageConverter"
class="com.oreilly.justspring.jms.converter.TradeMessageConverter"/>
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  ...
  <property name="messageConverter" ref="tradeMessageConverter" />
</bean>
```

Whenever you publish a `Trade` message, the `jmsTemplate` uses the converter to convert the `Trade` to `ObjectMessage`. Similarly when receiving, the template calls the converter to do the conversion. This way, you write the converter once and use it everywhere and at all times.

Summary

We have seen Java Messaging in action in this chapter. We briefly touched the subject of JMS and delved into using Spring's `JmsTemplate` class. We learned how we can publish the messages using the template class. We also saw how we can receive messages synchronously and asynchronously using Spring's framework classes called Message Containers.

The next chapter deals with persistence and retrieval of Data using Spring's JDBC and Hibernate support.

CHAPTER 5

Spring Data

Data persistence and retrieval are inevitable operations in an enterprise world. The advent of JDBC paved a way to interact with multiple databases with ease and comfort. It gained popularity in no time because of its unified API to access any database, be it MySQL or Oracle or Sybase. Spring created a lightweight framework abstracting the JDBC behind the scenes. Although the JDBC and Spring marriage makes a happy family, there are some unsophisticated or unavailable features from the joint venture. One feature that springs to mind is the support for Object Relational mappings. You still have to write plain old SQL statements to access the data. Hibernate came into existence utilizing this opportunity. It is now a popular and powerful open source framework. Spring added more abstraction on top of the already powerful Hibernate to make it even better.

This chapter explains how the Spring framework can be used effectively for accessing the databases, without even having to worry about connections and statements. We then continue on to Spring's ORM support using Hibernate.

JDBC and Hibernate

The joint venture did not attempt to bridge the gap between Objects and Relational Data. JDBC is certainly one of the first-hand choices for a Java developer when working with databases. JDBC abstracts away the intricacies involved in accessing different databases. It gives a clear and concise API to do the job easily. However, as many developers who worked with JDBC will moan about, there is a lot of redundant code that needs to be written, even if your intention is to fetch a single row of data. The advent of Spring Framework has changed this scenario drastically. Using a simple template pattern, Spring has revolutionized database access, digesting the boilerplate code altogether into the framework. We do not have to worry about the unnecessary bootstrapping and resource management code, and can write just the business logic. We will see in this chapter how the framework has achieved this objective.

There is a second scenario to consider: working with relational entities as if they are objects in your code. The simple name for this type of framework is Object Relational Mapping (ORM) framework. Hibernate, Java Data Objects (JDO), iBatis, and TopLink belong to this category. Using these ORM tools, we do not have to work at a low level as exposed by JDBC; instead we manipulate the data as objects. For example, a table called `MOVIES` consists of rows, each represented as a `MOVIE` relational entity. The same would be modeled as a `Movie` object in your code, and the mapping of the mapping of the `MOVIE` row to `Movie` domain object is performed by the framework behind the scenes. Hibernate with Spring framework is a truly cost-effective solution.

Spring JDBC

I agree that JDBC is a simple API for data access. However, when it comes to coding, it is still cumbersome, as you still have to write unnecessary code. Some say that about 80 percent of the code is repetitious. In a world of reusabiltiy, this is unacceptable. I myself have written and seen some homegrown frameworks to abstract the redundancies away from the developer. Spring does exactly this—abstracts away all the resource management so we can concentrate on the meat of the application. It might not surprise you to learn that Spring "reuses" its template design pattern, allowing us to interact with the databases in a clean and easy manner. The core of the JDBC package revolves around one class: `JdbcTemplate`. This class plays the key role in accessing data from your components.

JdbcTemplate

The basic and most useful class from the framework is the `JdbcTemplate`. This call should serve to do most of your work. But should you require a bit more sophistication, the two variants of `JdbcTemplate`—the `SimpleJdbcTemplate` and `NamedParameterJdbcTemplate`—should provide you that. The `JdbcTemplate` class provides the common aspects of database operations, such as inserting and updating data using prepared statements, querying tables using standard SQL queries, invoking stored procedures, etc. It can also iterate over the `ResultSet` data. The connection management is hidden from the user, and so is the resource pooling and exception management. Regarding the exceptions, one does not have to clutter the code with `try-catch` blocks because the database-specific exceptions are wrapped by Spring's Runtime Exceptions.

Following the template design pattern, the `JdbTemplate` provides some callback interfaces for you to implement. In these callbacks, you create the necessary business logic. For example, `PreparedStatementCallback` is used for creating `PreparedStatement`s, while `RowCallbackHandler` is where you extract the `ResultSet` into your domain objects. The `CallableStatementCallback` is used when executing a stored procedure. We will work briefly with these callbacks in the next few sections.

Configuring JdbcTemplate

Before we can jump into working with the `JdbcTemplate`, we need to take care of a few details. First, we need to supply a `DataSource` to the `JdbcTemplate` so it can configure itself to get database access. You can configure the `DataSource` in the XML file as shown in Example 5-1. I am using an open source JDBC framework—*Apache commons DBCP*—for creating my datasources.

Example 5-1.

```
<bean id="movieDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
  <property name="driverClassName" value="${jdbc.driver}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>
```

As you can see, creating the datasource is easy. Provide respective properties related to your database provider to fill in the properties shown above.

Once you have the datasource created, your next job is to create the `JdbcTemplate`. You have primarily two options: First, you can instantiate a `JdbcTemplate` in your code base (in your DAO), injecting the `DataSource` into it. Alternatively, you can define the `JdbcTemplate` in the XML file, wiring the datasource to it. You then inject the `JdbcTemplate` reference into your DAO class. The `JdbcTemplate` is a threadsafe object, so you can inject it into any number of DAOs.

Let us define the DAO interface for accessing the `MOVIES` database. Example 5-2 shows the simple API, which is self-explanatory:

Example 5-2.

```
public interface IMovieDAO {
  public Movie getMovie(String id);
  public String getStars(String title);
  public List<Movie> getMovies(String sql);
  public List<Movie> getAllMovies();
  public void insertMovie(Movie m);
  public void updateMovie(Movie m);
  public void deleteMovie(String id);
  public void deleteAllMovies();
}
```

The concrete class `MovieDAO` implements the `IMovieDAO` interface. It has `JdbcTemplate` as a member variable. It is configured and wired with a datasource in the XML file and injected into our concrete DAO. The XML file is shown below:

```
    <bean id="movieDao" class="com.oreilly.justspring.data.dao.MovieDAO"
    destroy-method="close">
      <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>
```

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="movieDataSource"/>
</bean>

<bean id="movieDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
  ....
</bean>
```

The listing shown below is the concrete implementation of the DAO:

```
public class MovieDAO implements IMovieDAO {
  private JdbcTemplate jdbcTemplate = null;
  private void setJdbcTemplate(JdbcTemplate jdbcTemplate){
    this.jdbcTemplate = jdbcTemplate;
  }
  private JdbcTemplate getJdbcTemplate() {
    return this.jdbcTemplate;
  }
  ...
}
```

That's it! Your `JdbcTemplate` is configured and ready to be used straight away. Let's concentrate on what the template can do for us.

Manipulating Data Using JdbcTemplate

The simplest operation is to fetch movie stars using a criteria such as movie title. You can write a simple SQL query like `"select stars from MOVIES where title='Dumbo'"` to retrieve the movie actors. Use the same query to invoke a `queryForObject` method on `JdbcTemplate` as shown below:

```
String stars = getJdbcTemplate().queryForObject("select stars from MOVIES
where title= 'Dumbo'", String.class);
```

This method takes two parameters, a SQL query without bind variables and an expected type of the result. The expected result is a comma-separated stars list. You can improve this query by parameterising the query. The `where` clause will have a bind variable that will change on queries. It is shown in the listing below:

```
String stars = getJdbcTemplate().queryForObject("select stars from MOVIES
where title=?", new Object[]{"Dumbo"}, String.class);
```

Here, ideally the second argument is passed via method arguments. For example, the title is passed in as a method parameter, as shown in the following code snippet:

```
public String getStars(String title) {
  String stars = getJdbcTemplate().queryForObject("select stars from MOVIES
where title=?", new Object[]{title}, String.class);
  return stars;
}
```

There is a plethora of `queryForXXX` methods defined on the `JdbcTemplate`, such as `queryForInt`, `queryForList`, `queryForMap`, etc. Refer to the Spring Framework's API to understand the workings.

Returning Domain Objects

The above queries returned a single piece of data, in this case, movie stars. How would I retrieve a `Movie` object for a given an id or criteria? Well, you can use `JdbcTemplate`'s `queryForObject` method passing additionally with a `RowMapper` instance. As we know, the JDBC API returns a `ResultSet` and we need to map each and every column data from the `ResultSet` into our domain objects. The Spring framework eliminates this repetitious process by providing `RowMapper` interface. Simply put, `RowMapper` is an interface for mapping table rows to a domain object. It has one method called `mapRow` that should be implemented by the concrete implementations.

What we need to do is implement this interface to map our table columns to a `Movie` object. Let's implement for our `Movie` domain object.

```
public class MovieRowMapper implements RowMapper{
  public Object mapRow(ResultSet rs, int rowNum){
  Movie movie = new Movie();
  movie.setID(rs.getString("id"));
  movie.setTitle(rs.getString("title"));
  movie.setStars(rs.getString("stars"));
  movie.setReleaseData(rs.getDate("release_data"));
  ....
  return movie;
 }
}
```

Basically, the idea is to extract the relevant columns from the `ResultSet` and populate our `Movie` domain object and return it.

Now that our `MovieRowMapper` is implemented, use `jdbcTemplate` to retreive the results.

```
public Movie getMovie(String id){
  String sql = "select * from MOVIES where id=?";
  return getJdbcTemplate().queryForObject(sql,new Object[]{id},new MovieRowMapper());
}
```

The `JdbcTemplate` executes the query by binding the argument and invoking the `Movie RowMapper` with a returned `ResultSet` from the query.

You can use the same `MovieRowMapper` for returning all movies. It should be wrapped in `RowMapperResultSetExtractor` as shown below:

```
public List getAllMovies(){
  RowMapper mapper = new MovieRowMapper();
  String sql = "select * from MOVIES";
  return getJdbcTemplate().query(sql, RowMapperResultSetExtractor(mapper,10));
```

We can use the `jdbceTemplate.update()` method to insert/update or delete the data. The following code shows the insertion of `Movie` into our database.

```
public void insertMovie(Movie m){
  String sql = "insert into MOVIES (ID, TITLE,GENRE, SYNOPSIS) values(?,?,?,?)";
  Object[] params = new Object[]{m.getId(),m.getTitle(),m.getGenre(),m.getSynopsis()};
  int[] types = new int[] {Types.VARCHAR,Types.VARCHAR,Types.VARCHAR,Types.VARCHAR);
  jdbcTemplate.update(sql, params, types);
}
```

Similarly, deleting a single movie from the database is straightforward:

```
public void deleteMovie(String id){
  String sql = "delete from MOVIES where ID=?";
  Object[] params = new Object[]{id};
  jdbcTemplate.update(sql, params);
}
```

In order to delete all movies, use the following code:

```
public void deleteAllMovies(){
  String sql = "delete from MOVIES";
  jdbcTemplate.update(sql);
}
```

Calling Stored Procedures is also an easy thing using the update method:

```
public void deleteAllMovies(){
  String sql = "call MOVIES.DELETE_ALL_MOVIES";
  jdbcTemplate.update(sql);
}
```

As we have noticed, the `JdbcTemplate` has eased our burden in accessing the database dramatically. I will advise you to refer Spring's API for the template methods and their usage.

Hibernate

Hibernate provides a mapping of database columns to the objects by reading a configuration file. We define the mapping of our domain objects to the table columns in the XML configuration file. The configuration file for each of the mappings should have an extension of *".hbm.xml"*. Spring abstracts the framework one step more and provides us with classes like `HibernateTemplate` to access the database. For example, let's define our `MOVIE` object using hibernate mapping rules.

```
<hibernate-mapping>
  <class name="com.oreilly.justspring.springdata.domain.Movie" table="MOVIES">
  <id name="id" column="ID">
    <generator class="assigned"/>
  </id>
  <property name="title" column="TITLE"/>
  <property name="genre" column="GENRE"/>
  <property name="synopsis" column="SYNOPSIS"/>
```

```
 </class>
</hibernate-mapping>
```

The `class` attribute defines the actual domain class, `Movie` in this case. The `id` attribute is the primary key and is assigned, meaning it is the application's responsibility to set the primary key. The rest of the properties are mapped against the respective columns in the `MOVIES` table.

Hibernate requires a `Session` object in order to access the database. A `Session` is created from the `SessionFactory`. When using Spring framework, you can use `LocalSessionFactoryBean` to create a `SessionFactory`. The `LocalSessionFactoryBean` requires a datasource to be wired in, along with hibernate properties and mapping resources. The hibernateProperties enables the properties such as database dialect, pool sizes, and other options. The mappingResources property loads the mapping config files (*Movie.hbm.xml* in our case).

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="movieDataSource"/>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
      <prop key="hibernate.show_sql">false</prop>
    </props>
  </property>
  <property name="mappingResources">
    <list>
      <value>Movie.hbm.xml</value>
    </list>
  </property>
  <bean id="movieDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
   ...
  </bean>
</bean>
```

Now that the `sessionFactory` is defined, the next bit is to define `HibernateTemplate`. The `HibernateTemplate` requires a `SessionFactory` instance, so the following declaration wires the `sessionFactory` we defined earlier.

```
<bean id="hibernateTemplate"
class="org.springframework.orm.hibernate.HibernateTemplate">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="movieDao" class="com.oreilly.justspring.data.dao.MovieDAO">
  <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
```

The configuration is completed. Let's implement a few methods for retrieving the data. The `getMovie` method shown below uses the template's load method.

```
public Movie getMovie(String id){
  return (Movie)getHibernateTemplate().load (Movie.class, id);
}
```

As you can see, there's no SQL that retrieves a movie in this method. We can feel that we are working with Java objects rather than data!

The load method accesses the database to load the matching row based on the id passed. Updating a Movie is simple as well:

```
public void updateMovie(Movie m){
  getHibernateTemplate().update (m);
}
```

As you can see, the single statement above will do the job! Deleting a row is as simple as invoking the delete method.

```
public void deleteMovie(Movie m){
  getHibernateTemplate().delete (m);
}
```

Running queries is straightforward, too. Hibernate introduces Hibernate Query Language (HQL) for writing queries. Use find methods to execute the queries. For example, returning a Movie based on a ID is shown below:

```
public Movie getMovie(String id){
  return (Movie)getHibernateTemplate().find("from MOVIES as movies
where movies.id=?",id);
}
```

Summary

In this chapter, we discussed Spring's support of JDBC and Hibernate. As we have seen in the examples, Spring has truly simplified our lives by providing a simple yet powerful API to work with. We can concentrate on the business logic rather than writing reams of repetitive code fragments.

About the Author

Madhusudhan Konda is an experienced Java consultant working in London, primarily with investment banks and financial organizations. Having worked in enterprise and core Java for last 12 years, his interests lie in distributed, multi-threaded, n-tier scalable, and extensible architectures. He is experienced in designing and developing high-frequency and low-latency application architectures. He enjoys writing technical papers and is interested in mentoring.

Colophon

The bird on the cover of *Just Spring* is a Tree Swift.

The cover image is from Cassell's *Natural History*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.