Mockito for Spring

Learn all you need to know about the Spring Framework and how to unit test your projects with Mockito

Sujoy Acharya
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 ‘Getting Familiar with the Spring Framework’
- A synopsis of the book’s content
- More information on Mockito for Spring

About the Author

Sujoy Acharya works as a software architect with Siemens Technology and Services Pvt. Ltd. (STS). He grew up in a joint family and pursued his graduation in the fields of computer science and engineering. His hobbies are watching movies and sitcoms, playing outdoor sports, and reading books.

Sujoy likes to research upcoming technologies. His major contributions are in the fields of Java, J2EE, SOA, Ajax, GWT, and the Spring Framework.

He has authored three books for Packt Publishing, namely Test-Driven Development with Mockito, Mastering Unit Testing using Mockito and JUnit, and Mockito Essentials.

He designs and develops healthcare software products. He has over 11 years of experience in the industry and has architected and implemented large-scale enterprise solutions.

I’d especially like to thank my wife, Sunanda, firstly for pushing me to man up and finish the book, and additionally, for her patience and endless support in the many hours spent on reviewing my draft and providing valuable inputs.

I would also like to thank my mother and late father for their support, blessings, and encouragement.

To my 23-month-old kid, Abhigyan, I am sorry I couldn't be around as much as we all wanted and for the many times I had to get you away from the laptop. I love you very much.
Mockito for Spring

When I was writing the first draft of this book, I was eager to compare the manuscript with other books on the Spring Framework. Here are the features that distinguish this book from others:

- This book is not only about the Spring Framework. It also describes the basics of Spring, Spring's test module, Spring's integration testing, JUnit testing, how to mock Spring beans with Mockito, and advanced Spring 4.1 features. I couldn't find any books that cover these topics.
- This book explains JUnit testing and mocking in the context of Spring.
- The book covers Spring's test module and Spring integration testing in detail. These are the most difficult parts in the Spring testing world. I have taken a hands-on approach here by combining theories with examples to explain the topics.

What This Book Covers

Chapter 1, Getting Familiar with the Spring Framework, covers the basics of Spring, Spring projects, and especially the Spring Framework. It explores the Spring container, the life cycle of beans, dependency injection, AOP, Spring MVC, and Spring transaction management.

Chapter 2, Working with JUnit and Mockito, covers both basic and advanced JUnit usages. It covers annotation-based JUnit testing, assertion, the @RunWith annotation, exception handling, and the Eclipse setup to run JUnit tests, matchers, and assertThat, as well as the custom lessThanOrEqual() matcher. The Working with Mockito section explores the Mockito framework and provides technical examples to demonstrate the capability of Mockito.

Chapter 3, Working with Spring Tests, illustrates every aspect of unit testing your Spring applications. It starts with TestContext and explores the JUnit 4 enabled SpringJUnit4ClassRunner. Then, it explores Spring profiles that can be used to work with the different sets of configuration files as well as the Spring environment and how to mock the environment with MockEnvironment and MockPropertySource. We use the ReflectionTestUtils method to access the private fields of the Spring beans. The chapter provides usage examples of Spring annotations for testing, unit tests the MVC application with MockHttpServletRequest, MockHttpSession, and ModelAndViewAssert, and mocks the servlet container with MockMvc to handle actual requests and responses, as they will be at runtime. You will also perform real Spring integration and transaction management with annotations such as @Transactional, @TransactionConfiguration, and @Rollback.
Chapter 4, *Resolving Out-of-container Dependencies with Mockito*, deals with unit testing the service layer in isolation from the data access layer with Mockito, unit testing the Spring data access layer with Mockito, and unit testing the Spring presentation layer (MVC) with Mockito.

Chapter 5, *Time Travelling with Spring*, starts by covering the features of the new major Spring release 4.0, such as Java 8 support and so on. Then, we pick the four Spring 4 topics and explore them one by one. The *Working with asynchronous tasks* section showcases the execution of long running methods asynchronously and provides examples to handle asynchronous processing. The *Exploring @RestController* section eases RESTful web service development with the advent of the `@RestController` annotation. The *Learning AsyncRestTemplate* section explains the RESTful client code to invoke RESTful web services asynchronously. Caching is inevitable for high performant, scalable web applications. This section explains EhCache and Spring integration to achieve a high availability caching solution.
Getting Familiar with the Spring Framework

Spring is a popular enterprise application development framework. This chapter covers the following topics:

- Spring Framework fundamentals
- Spring projects
- The Spring architecture and modules
- **Inversion of control (IoC) and dependency injection (DI)**
- Setting up a Spring development environment, a Hello World program, and autowiring
- **Aspect-oriented Programming (AOP)**
- Spring JDBC
- Transaction management
- Spring MVC

**Getting started with Spring**

Spring is an open source enterprise application development framework for Java. It was first written by Rod Johnson and released under the Apache 2.0 license in June 2003.

Spring Framework provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure for us and allows us to focus on our application logic. Spring enables us to build applications from **Plain Old Java Objects (POJOs)** and apply enterprise services non-invasively to POJOs.
Getting Familiar with the Spring Framework

The following are examples of POJO-based application development:

• A Java method can handle an HTTP POST/GET request; you don't have to write a servlet or work with servlet APIs
• A Java method can act as a RESTful web service without dealing with web service APIs
• A Java method can execute a database transaction without dealing with transaction APIs
• A local Java method can participate in a remote procedure call (RPC) without having to deal with remote APIs
• A Java method can consume or handle messages without having to deal with JMS APIs
• A Java method can work as a management extension without dealing with JMX APIs

In a nutshell, Spring can be described as follows:

• An open source application framework
• One of the available enterprise application frameworks and a lightweight solution for enterprise applications
• Non-invasive (POJO-based)
• Modular
• Extensible for other frameworks
• The de facto standard of Java enterprise applications

The following are advantages of Spring:

• Lightweight and minimally invasive development with POJOs
• Loose coupling through dependency injection and interface-orientation
• Declarative programming through aspects and common conventions
• Boilerplate code reduction through aspects and templates

Spring projects provide infrastructure for building security configuration, web applications, big data, LDAP, and so on. Spring Framework is one of the Spring projects.
There are various Spring projects that can be used. In this book, we’ll be using Spring 4.

The following are the icons of some Spring projects:

The following are all Spring projects as of September 2014:

- **The Spring IO platform**: Spring IO brings together the core Spring APIs into a cohesive and versioned foundational platform for modern applications. Spring IO is comprised of the Spring IO Foundation and Spring IO Execution layers.

- **Spring Boot**: This helps in creating production-grade Spring applications that can be run any time with the minimal Spring configuration. It follows the convention-over-configuration approach.

- **Spring Framework**: This is an open source framework for Java enterprise applications. It provides an inversion of control container for Java beans. The framework offers a number of templates for the developers; the templates hide the infrastructure code and allow us to concentrate on the business logic.

- **Spring XD**: This is a unified, distributed, and extensible system for data ingestion, real-time analytics, batch processing, and data export. The goal of the project is to simplify the development of big data applications.

- **Spring Cloud**: Spring Cloud builds on Spring Boot by providing a bunch of libraries that enhance the behavior of an application when added to the classpath. You can take advantage of the basic default behavior to get started really quickly, and then when you need to, you can configure or extend it to create a custom solution.

- **Spring Data**: This simplifies data access, offers APIs to work with the relational databases, NoSQL or non-relational databases, big data or the map-reduce algorithm, and so on.
Getting Familiar with the Spring Framework

- **Spring Integration**: This follows Enterprise Integration Patterns (EIP) to enable us lightweight, POJO-based messaging for Spring applications to integrate with external systems.

- **Spring Batch**: This is a lightweight, comprehensive batch framework designed to enable the development of robust batch applications vital for the daily operations of enterprise systems.

The following image displays the icons of the following spring projects: security, HATEOAS, social, AMQP, web services, Mobile, Android, web flow, Spring LDAP and Grails

- **Spring Security**: This is a powerful and highly customizable authentication and access-control framework. It is the de facto standard for securing Spring-based applications.

- **Spring HATEOAS**: This allows you to create REST representations that follow the HATEOAS principle from your Spring-based applications.

- **Spring Social**: Connect your Spring application with Software as a Service (SaaS) API providers such as Facebook, Twitter, and LinkedIn.

- **Spring AMQP**: The Advanced Message Queuing Protocol (AMQP) is an open standard for messaging. Spring AMQP offers solutions for AMQP-based messaging, for example, it can be used with the AMQP broker RabbitMQ.

- **Spring Mobile**: This is an extension to Spring MVC that aims to simplify the development of mobile web applications.

- **Spring for Android**: This is an extension of Spring Framework that aims to simplify the development of native Android applications.
• **Spring Web Flow**: This provides the infrastructure to build process workflows for web-based Spring applications, such as page navigation, navigation triggers, application state, and services to invoke. This is stateful and can be a short-lived process flow or long-running flow.

• **Spring Web Services**: This aims to facilitate contract-first SOAP service development, and this allows the creation of flexible web services using one of the many ways to manipulate XML payloads.

• **Spring LDAP**: This makes it easier to build Spring-based applications that use the Lightweight Directory Access Protocol (LDAP).

### Exploring the Spring architecture

Spring Framework is modular, and its features are organized into different modules. This section talks about core Spring modules. The following are the Spring 4 modules:

#### The core container

The core container holds the backbone of Spring Framework. The following are the submodules in the core container:

- **Core and Beans**: These provide the fundamental parts of the framework, including IoC and dependency injection features

- **Context**: This is a means to access objects in a framework-style manner that is similar to the JNDI registry

- **Expression Language**: This is also known as SpEL; it is an expression language used to query and modify an object graph and evaluate mathematical expressions

#### The AOP module

AOP is an aspect-oriented programming implementation of Spring. It decouples the business logic from the cross-cutting infrastructure code, such as logging and security.

#### The instrumentation module

The instrumentation module provides class instrumentation support for the Spring application. Instrumentation exposes container resources through MBean and helps in JMX management.
The messaging module

The messaging module comes with key abstractions from the Spring Integration project such as Message, MessageChannel, and MessageHandler to serve as a foundation for messaging-based applications.

The data access module

The following are the submodules in the data access module:

- **JDBC**: This provides a JDBC abstraction layer
- **ORM**: This provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBATIS
- **OXM**: This provides an abstraction layer that supports object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX, and Xstream
- **JMS**: This contains features to produce and consume messages
- **Transactions**: This supports programmatic and declarative transaction management

The web layer

The web layer consists of the web, webmvc/servlet, WebSocket, and webmvc-portlet modules:

- **Web**: This module provides basic web-oriented integration features such as multipart file upload functionality and initialization of the IoC container using servlet listeners and web-oriented application context. It also contains the web-related parts of Spring's remoting support.
- **Webmvc**: This module (also known as the web-servlet module) contains Spring's model-view-controller implementation for web applications. Spring's MVC framework provides a clean separation between the domain model code and web forms and integrates with all the other features of Spring Framework.
- **Portlet**: This module (also known as the web-portlet module) provides the MVC implementation to be used in a portlet environment and mirrors the functionality of the webmvc module.
- **WebSocket**: This module provides APIs for two-way communication between client and server. It is extremely useful when the client and server need to exchange events at high frequency and low latency. Prime candidates include applications in finance, games, collaboration, and so on.
The test module

The test module supports the unit testing and integration testing of Spring components with JUnit or TestNG.

The following figure represents the Spring 4 modules:

Learning the Inversion of Control

Inversion of Control (IoC) and dependency injection (DI) are used interchangeably. IoC is achieved through DI. DI is the process of providing dependencies and IoC is the end result of DI. Spring's IoC container enforces the DI pattern for your components, and this leaves them loosely coupled and allows you to code to abstractions.

Dependency injection is a style of object configuration in which an object's fields and collaborators are set by an external entity. In other words, objects are configured by an external entity. Dependency injection is an alternative to having the object configure itself. This might sound a bit vague, so let's look at a simple example.

After visiting the Packt Publishing website, you can search books by the author's name or different criteria. We'll look at the service that lists books by author.
The following interface defines book retrieval:

```java
public interface BookService {
    List<Book> findAll();
}
```

The following class lists books by author names:

```java
public class BookLister {

    private BookService bookFinder = new BookServiceImpl();

    public List<Book> findByAuthor(String author){
        List<Book> books = new ArrayList<>();

        for(Book aBook:bookFinder.findAll()){
            for(String anAuthor:aBook.getAuthors()){
                if(anAuthor.equals(author)){
                    books.add(aBook);
                    break;
                }
            }
        }

        return books;
    }
}
```

The BookLister class needs a BookService implementation; this means that the BookLister class depends on it. It cannot carry out its work without a BookService implementation. Therefore, BookLister has a dependency on the BookService interface and on some implementation of it. The BookLister class itself instantiates BookServiceImpl as its BookService implementation. Therefore, the BookLister class is said to satisfy its own dependencies. When a class satisfies its own dependencies, it automatically also depends on the classes it satisfies the dependencies with. In this case, BookLister now also depends on BookServiceImpl, and if any, on the other values passed as a parameter to the BookServiceImpl constructor. The BookService interface can have many implementations such as Spring JDBC-based data access and JPA-based data access implementation. We cannot use a different implementation of the BookService interface without changing the code.
To refactor this tight coupling, we can move the `BookService` instantiation to the constructor of the class. The following is the modified `BookLister` class:

```java
public class BookLister {

    private final BookService bookFinder;

    public BookLister(BookService bookFinder) {
        this.bookFinder = bookFinder;
    }

    public List<Book> findByAuthor(String author){
        List<Book> books = new ArrayList<>();

        for(Book aBook:bookFinder.findAll()){
            for(String anAuthor:aBook.getAuthors()){
                if(anAuthor.equals(author)){
                    books.add(aBook);
                    break;
                }
            }
        }

        return books;
    }
}
```

Note that the `BookService` dependency is passed to the `BookLister` constructor as a constructor argument. Now, `BookLister` is only depending on `BookService`. Whoever instantiates the `BookLister` constructor will satisfy the dependency. The `BookService` dependency is said to be injected into the `BookLister` constructor, hence the term dependency injection. It is now possible to change the `BookService` implementation used by the `BookLister` class without changing the `BookLister` class.

There are two types of dependency injections:

- Constructor injection
- Setter injection
A Spring configuration file creates/defines and configures (resolves dependencies) beans. In the Spring configuration file, the constructor injection is constructed as follows:

```xml
<bean id="bookLister" class="com.packt.di.BookLister">
  <constructor-arg ref="bookService"/>
</bean>
<bean id="bookService" class="com.packt.di.BookServiceImpl" />
```

The preceding code is equivalent to the following:

```java
BookService service = new BookServiceImpl();
BookLister bookLister = new BookLister(service);
```

The setter injection is carried out by setting a property. In a setter injection, instead of passing `bookService` as a constructor argument, we change the class to pass as a setter method argument.

The Spring configuration is as follows:

```xml
<bean id="bookListerSetterInjection" class="com.packt.di.BookLister">
  <property name="bookService" ref="bookService" />
</bean>
<bean id="bookService" class="com.packt.di.BookServiceImpl" />
```

The preceding code snippet is equivalent to the following:

```java
BookService service = new BookServiceImpl();
BookLister bookLister = new BookLister();
bookLister.setBookService(service);
```

The Spring IoC container is known as `ApplicationContext`. The objects that are used in our application, defined in `ApplicationContext`, and managed by the Spring IoC container are called beans; for example, `bookService` is a bean.

A bean is an object that is managed by the Spring IoC container; beans are created with the configuration metadata that you supply to the container, such as in the form of XML `<bean/>` definitions or using Java annotations.

A bean definition describes a bean instance. The bean definition contains the information called configuration metadata, which is needed by the container to know how to create the bean, the life cycle of the bean, and the dependencies of the bean.
The following properties are used to define a bean:

- **class**: This is mandatory and provides the fully qualified bean class name required for the container to create the bean instance.
- **name**: This attribute (also known as id) uniquely identifies a bean.
- **scope**: This provides the scope of the objects created from a bean definition, such as prototype and singleton. We’ll learn about them later.
- **constructor-arg**: This injects a dependency as a bean's constructor argument.
- **properties**: This injects a dependency as a setter method argument.
- **lazy-init**: If this is set as true, the IoC container creates the bean instance when it is first requested, rather than at startup, which means any configuration error is not discovered until the bean is eventually instantiated inside the Spring context.
- **init-method**: This provides the method name of the bean that is being invoked just after all necessary properties on the bean are set by the IoC container. This is useful when we need to initialize/compute something after the bean is instantiated.
- **destroy-method**: The container calls this method when the bean is destroyed; this is necessary when we need to clean up something before the bean is destroyed.

The following are the bean scopes:

- **singleton**: A single instance of the bean per IoC container. This is not actually the same as in the singleton design pattern (that is, one instance per classloader).
- **prototype**: A single bean definition to have any number of object instances. A new bean instance is created each time one is needed.
- **request**: A bean instance per HTTP request, only valid in the web-aware application context.
- **session**: A bean instance per HTTP session, only valid in the web-aware application context.
- **global-session**: A bean instance per global HTTP session, only valid in the web-aware application context.
The following are the steps in a bean's life cycle:

1. The first step is to find and instantiate the beans. The Spring IoC container reads the bean definitions from the XML and then instantiates them.
2. The next step is to populate the bean properties and satisfy the dependencies. The IoC container uses dependency injection to set the properties.
3. After setting the dependencies, the `setBeanName` method is invoked on the beans; if they implement the `BeanNameAware` interface, the `setBeanName()` method is invoked by passing the ID of the bean.
4. After this, if a bean implements the `BeanFactoryAware` interface, the `setBeanFactory()` method is called with an instance of itself.
5. The pre-initialization of `BeanPostProcessor` is done. If a bean has any `BeanPostProcessor` interface associated with it, the `processBeforeInitialization()` methods are called on the post processors.
6. The init method is called; if a bean specifies an `init-method`, it will be called.
7. Finally, the post-initialization is done; if there are any `BeanPostProcessors` associated with the bean, their `postProcessAfterInitialization()` methods are invoked.

Note that a POJO doesn't need to depend on anything Spring-specific. For particular cases, Spring provides hooks in the form of these interfaces. Using them means introducing a dependency on Spring. The following figure depicts the bean's life cycle:

To learn more about DI and IoC, visit the Martin Fowler site at http://martinfowler.com/articles/injection.html.
Printing Hello World

In this section, we'll create a hello world example and set up the Eclipse environment for Spring. You can download the latest Eclipse version from http://www.eclipse.org/downloads/.

Note that Spring provides a specific Eclipse distribution for Spring, known as **Spring Tool Suite (STS)**. STS is customized for developing Spring applications. You can download STS from http://spring.io/tools/sts.

Download the Spring 4.1.0 JAR from the Maven repository at http://search.maven.org/ or http://mvnrepository.com/artifact/org.springframework.

1. Launch Eclipse and create a Java project and name it `SpringOverview`.
2. Add the following dependencies:

<table>
<thead>
<tr>
<th>JARs and class folders on the build path:</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-collections-3.1.jar - SpringOverview</td>
</tr>
<tr>
<td>commons-logging.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-aop-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-beans-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-context-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-context-support-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-core-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-expression-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-jdbc-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-messaging-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-test-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-tx-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-web-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>spring-webmvc-4.1.0.RELEASE.jar - SpringOverview</td>
</tr>
<tr>
<td>JRE System Library [JavaSE-1.7]</td>
</tr>
</tbody>
</table>

3. Create a `com.packt.lifecycle` package under `src`.
4. Add a `HelloWorld` class with following details:

```java
public class HelloWorld {
    private String message;
    public String getMessage() {
        return message;
    }
}
```
Getting Familiar with the Spring Framework

```java
public void setMessage(String message) {
    this.message = message;
}
```

5. Add an XML file, applicationContext.xml, directly under the src folder and add the bean definition as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="helloWorld" class="com.packt.lifecycle.HelloWorld">
        <property name="message" value="Welcome to the Spring world"/>
    </bean>
</beans>
```

6. Create a Java class, HelloWorldExample, and add the following lines to check the bean configuration:

```java
public class HelloWorldExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        HelloWorld world = (HelloWorld) context.getBean("helloWorld");
        System.out.println(world.getMessage());
    }
}
```

We load the Spring bean configuration from an XML file, which is kept in the classpath and named applicationContext.xml, and then ask the context to find a bean with a name or ID as helloWorld. Finally, we call the getMessage() method on the bean to check the value we set in the application context.
7. When we run the HelloWorldExample program, the following output is displayed:


Welcome to the Spring world

Exchanging life cycle messages

We read about the bean's life cycle; why don't we try to examine the life cycle?

Modify the HelloWorld class and implement the following Spring Framework interfaces:

- **ApplicationContextAware**: This will ask you to implement the `setApplicationContext` method
- **BeanNameAware**: This will tell you to implement the `setBeanName` method
- **InitializingBean**: This will force you to implement the `afterPropertiesSet()` method
- **BeanFactoryAware**: This will request you to implement the `setBeanFactory` method
- **BeanPostProcessor**: This needs you to implement the `postProcessBeforeInitialization` and `postProcessAfterInitialization` methods
- **DisposableBean**: This needs to implement the `destroy()` method

Add the `System.out.println` statement in all the implemented methods. Now, add the following two methods:

```java
public void myInit() {
    System.out.println("custom myInit is called ");
}

public void myDestroy() {
    System.out.println("custom myDestroy is called ");
}
```
Modify the bean definition to call the *init*-method and *destroy*-method methods. The following is the modified bean definition:

```xml
<bean id="helloWorld" class="com.packt.lifecycle.HelloWorld"
    init-method="myInit" destroy-method="myDestroy">
    <property name="message" value="Welcome to the Spring world"/>
</bean>
```

Now, modify HelloWorldExample to destroy the application context by registering to shutdown hook. The following is the modified code:

```java
AbstractApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
HelloWorld world = (HelloWorld) context.getBean("helloWorld");
System.out.println(world.getMessage());
context.registerShutdownHook();
```

When we run the application, the following output is displayed:

```
SetBeanName is called with helloWorld
setBeanFactory is called
setApplicationContext is called
afterPropertiesSet is called
custom myInit is called
Welcome to the Spring world
destroy is called
custom myDestroy is called
```

Note that the `setBeanName` method is invoked first, then the `setBeanFactory`, `setApplicationContext`, and `afterPropertiesSet` methods are called, and then the custom `init` method is invoked. During destruction, the `destroy` method is called first and then the custom `destroy-method` is invoked.

**Working with autowiring and annotations**

The Spring container can autowire dependencies between the collaborating beans without using the `<constructor-arg>` and `<property>` elements that simplify the application context XML configuration.

The following autowiring modes can be used to instruct a Spring container to use autowiring for dependency injection:

- **no**: By default, the settings is `no`. This means no autowiring.
- **byName**: The container tries to match and wire bean properties with the beans defined by the same name in the configuration file.
• byType: The container tries to match a property if its type matches with exactly one of the bean names in the configuration file. If more than one such bean exists, an exception is thrown.

• constructor: This is similar to type but looks at the constructor type matching. If more than one bean of the constructor argument type is found in the container, an exception is thrown.

• default: This tries to wire using autowire by constructor; if it does not work, then it tries autowire by byType.

Let's modify our HelloWorld example and try wiring by name:

```xml
<bean name="message" class="java.lang.String">
  <constructor-arg value="auto wired" />
</bean>

<bean id="helloWorld" class="com.packt.lifecycle.HelloWorld"
  autowire="byName">
</bean>
```

It will print auto wired.

Spring provides annotations to wire collaborators. The following are the annotations:

• @Required: This annotation applies to the bean property setter method

• @Autowired: This can be applied to bean property setter methods, constructor, and properties

• @Qualifier: This annotation along with @Autowired can be used to wire a bean with the qualifier name

To enable autowiring through an annotation, the application context needs to be configured to indicate the annotation. Add the following entry to the application context:

```xml
<context:annotation-config/>
```

Modify the application context to enable an annotation:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  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.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context"
..."/>
Getting Familiar with the Spring Framework

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<bean name="message" id="message" class="java.lang.String">
   <constructor-arg value="auto wired" />
</bean>

<bean id="helloWorld" class="com.packt.lifecycle.HelloWorld">
</bean>

</beans>

Modify the HelloWorld class to annotate the setter method (setMessage) or the private message property with @Autowired:

public class HelloWorld implements ApplicationContextAware,BeanNameAware, InitializingBean, BeanFactoryAware,BeanPostProcessor, DisposableBean {

private String message;

public String getMessage() {
   return message;
}

@Autowired
public void setMessage(String message) {
   this.message = message;
}
   //code omitted for brevity
}

Rerun the application; you will see the auto wired message.

Working with aspects

AOP is one of the key components of Spring Framework. Object-oriented programming fails to deal with technical and functional cross-cutting concerns, such as generic functionalities that are needed in many places in our application.
The following are a few examples of cross-cutting concerns:

- Logging and tracing
- Transaction management
- Security
- Caching
- Error handling
- Performance monitoring
- Custom business rules
- Event handling

In our application, we need logging to debug or troubleshoot, so we put debug messages in every method; this is a cross-cutting concern. Similarly, we secure methods for unauthorized access.

AOP overlays a new layer onto the data-driven composition of OOP. This layer corresponds to the cross-cutting functionalities that are difficult to integrate through the OOP paradigm.

AOP is implemented with AspectJ and Spring AOP:

- **AspectJ**: This is the original AOP technology (the first version dates from 1995) that offers a full-blown, aspect-oriented programming language and uses bytecode modification for aspect weaving.
- **Spring AOP**: This is a Java-based AOP framework and it uses dynamic proxies for aspect weaving. This focuses on using AOP to solve enterprise problems.

The following example demonstrates a cross-cutting concern:

```java
class Account{
    private double balance;
    public void withdraw(double amount){
        logger.debug("Withdraw -"+amount);
        tx.begin();
        balance = this.balance-amount;
        accountDao.saveBalance(balance);
        tx.commit();
    }
}
```
Getting Familiar with the Spring Framework

The withdraw method logs debug information, begins a transaction, performs a database transaction, and finally commits the transaction. In each method, we will introduce duplicate code for debugging and opening and committing a transaction. These are cross-cutting concerns as the conceptually duplicate code will be scattered to all modules in the application. This is bad in the sense that if we need to change any settings, we have to manually change all methods in all modules, such as instead of `logger.debug`, and if we need to change the logging to `logger.info`, we need to modify all methods.

Before we dig deep into AOP, let’s get familiar with the terminology:

- **Join point**: This is a well-defined point during the execution of your application. You can insert additional logic at join points. Examples of join points are as follows:
  - Method invocation
  - Class initialization
  - Object initialization

- **Advice**: This is the code that is executed at a specific join point. The three types of advice are as follows:
  - The before advice is executed before a join point.
  - The after advice is executed after a join point.
  - The around advice is executed around a join point. The around advice spans the before and after advice.

- **Pointcut**: This is a collection of join points to execute an advice. A join point is a possibility of executing an advice, whereas a pointcut is a set of selected join points where actually the advice is executed.

- **Aspect**: This defines the implementation of the cross-cutting concern. An aspect is the combination of advice and pointcuts. An application can have any number of aspects, depending on the requirement.

- **Weaving**: This is the process of applying aspects into the code at the appropriate join points. There are three types of weaving:
  - Compile-time weaving
  - Class load-time weaving
  - Runtime weaving
• **Target**: This is the object that is advised by one or more aspects.

• **Introduction**: This is the process by which you can modify the structure of an object by introducing additional methods or fields to it. You use the introduction to make any object implement a specific interface without needing the object's class to implement that interface explicitly.

There are two types of AOP:

• **Static AOP**
  - The weaving process forms another step in the build process for an application
  - For example, in a Java program, you can achieve the weaving process by modifying the actual bytecode of the application by changing and modifying the code as necessary

• **Dynamic AOP**
  - The weaving process is performed dynamically at runtime
  - It is easy to change the weaving process without recompilation

Spring AOP is based on proxies. To know more about proxies, read about the proxy pattern or visit [http://en.wikipedia.org/wiki/Proxy_pattern](http://en.wikipedia.org/wiki/Proxy_pattern).

We'll display Hello World! through AOP. The following are the steps to create the hello world message:

1. **Create an interface called IMessageWriter**:
   ```java
   package com.packt.aop;
   
   public interface IMessageWriter {
       void writeMessage();
   }
   
   package com.packt.aop;
   
   public class MessageWriter implements IMessageWriter {
       @Override
       public void writeMessage() {
           System.out.print("World");
       }
   }
   ```

2. **Create a class called MessageWriter and implement the IMessageWriter interface**:
   ```java
   package com.packt.aop;
   
   public class MessageWriter implements IMessageWriter {
       @Override
       public void writeMessage() {
           System.out.print("World");
       }
   }
   ```
3. The join point is the invocation of the `writeMessage()` method. What we need is an around advice as we'll prepend *Hello* before *World* and append the exclamation after *World* to make it *Hello World!* The `MethodInterceptor` interface is AOP Alliance standard interface for around interface. The `MethodInvocation` object represents the method invocation that is being advised. We'll create an advice as follows:

```java
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class MessageDecorator implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation)
        throws Throwable {
        System.out.print("Hello ");
        Object retVal = invocation.proceed();
        System.out.println("!");
        return retVal;
    }
}
```

4. We'll use the `ProxyFactory` class to create the proxy of the target object:

```java
import org.springframework.aop.framework.ProxyFactory;
public class AOPTest {
    public static void main(String[] args) {
        MessageWriter target = new MessageWriter();
        // create the proxy
        ProxyFactory pf = new ProxyFactory();
        // Add the given AOP Alliance advice to the tail
        // of the advice (interceptor) chain
        pf.addAdvice(new MessageDecorator());
        // Set the given object as target
        pf.setTarget(target);
        // Create a new proxy according to the
        // settings in this factory
        MessageWriter proxy = (MessageWriter) pf.getProxy();
        // write the messages
        target.writeMessage();
        System.out.println(" ");
        // use the proxy
        proxy.writeMessage();
    }
}
```
When we run the program, the MessageDecorator around advice is applied on the proxy object. When `proxy.writeMessage` is called, the correct output is displayed.

**Exploring Spring JDBC**

The Spring Data Access Object (DAO) support makes it easy to work with data access technologies such as JDBC, Hibernate, or JDO in a standardized way. Spring Framework provides APIs to reduce JDBC code duplication. Spring JDBC hides the low-level details and allows us to concentrate on business logic, which makes switching between databases easy and simple.

In a normal JDBC code, we catch a series of checked exceptions such as SQLException while acquiring a connection or executing a SQL statement; with Spring, we can code without worrying about catching exceptions, as Spring does the exception handling for us. Spring is not throwing away or eating the checked exceptions but is instead translating them to unchecked/runtime ones.

Spring provides a set of abstract DAO classes that one can extend; these abstract classes have methods to provide the data source and any other configuration settings that are specific to the technology one is currently using.

The following are the DAO support classes:

- JdbcDaoSupport
- HibernateDaoSupport
- JdoDaoSupport
- JpaDaoSupport

In normal JDBC code, we write the code in the following way to access the database:

1. Define the connection parameters.
2. Open the connection.
3. Specify the statement.
4. Prepare and execute the statement.
5. Set up the loop to iterate through the results (if any).
6. Do the work for each iteration.
7. Process any exception.
8. Handle transactions.
9. Close the connection.
Spring Framework relaxes the requirement to write numerous JDBC code lines. We need to write only the code to perform the following:

- Specify the statement
- Do the work for each iteration

Spring takes care of all the grungy, low-level details that can make JDBC such a tedious API to develop against.

The Spring-JDBC abstraction framework consists of four different packages:

- `org.springframework.jdbc.core`
- `org.springframework.jdbc.datasource`
- `org.springframework.jdbc.object`
- `org.springframework.jdbc.support`

The `org.springframework.jdbc.core` package contains the following:

- The `JdbcTemplate` class
- Various callback interfaces
- A variety of related classes

The `org.springframework.jdbc.datasource` package contains the following classes:

- A utility class for easy `DataSource` access
- Various simple `DataSource` implementations that can be used to test and run unmodified JDBC code outside of a J2EE container
- The utility class provides static methods to obtain connections from JNDI and to close connections if necessary
- It has support for thread-bound connections, for example, to use with `DataSourceTransactionManager`

The `org.springframework.jdbc.object` package contains the following:

- Classes that represent RDBMS queries, updates, and stored procedures as thread-safe, reusable objects
- This approach is modeled by JDO, although of course, objects returned by queries are disconnected from the database
- This higher level of JDBC abstraction depends on the lower-level abstraction in the `org.springframework.jdbc.core` package
The `org.springframework.jdbc.support` package contains the following:

- The `SQLException` translation functionality and some utility classes
- Exceptions thrown during JDBC processing are translated to exceptions defined in the `org.springframework.dao` package
- The code using the Spring JDBC abstraction layer does not need to implement JDBC-or RDBMS-specific error handling
- All translated exceptions are unchecked giving you the option of catching the exceptions that you can recover from while allowing other exceptions to be propagated to the caller

The `JdbcTemplate` class is the main class in the `org.springframework.jdbc.core` package. It simplifies the use of JDBC since it handles the creation and release of resources. This helps avoid common errors such as not closing the connection, and it executes the core JDBC workflow such as statement creation and execution leaving application code to provide SQL and extract results.

We'll build a phone book application and store phone numbers using Spring JDBC and normal JDBC and realize the simplicity and usability of Spring JDBC. We'll use the Apache Derby database for persistence. Derby can be downloaded from [http://db.apache.org/derby/](http://db.apache.org/derby/).

You can use better built-in databases such as H2. It has more features and less restriction than Derby. However, we're using Derby for simplicity.

The following are the steps to run Derby:

1. Download the binary media file and extract media to a location. We'll refer to it as `DERBY_HOME` in the next steps.
2. On a Windows machine, go to `DERBY_HOME\bin` and execute `startNetworkServer.bat`.
3. It will launch Command Prompt and print to the console that the database server is started, such as the following:

   started and ready to accept connections on port 1527.

Perform the following steps to implement Spring JDBC and simplify the code:

1. Launch Eclipse and create a Java project named DatabaseAccess.
2. Add a class PhoneEntry to store phone details. The following are the class details:

   ```java
   package com.packt.database.model;

   public class PhoneEntry implements Serializable {
       private static final long serialVersionUID = 1L;

       private String phoneNumber;
       private String firstName;
       private String lastName;

       // getters and setters
   }
   ```

3. Create a data access interface for the phone book. The following are the API details:

   ```java
   package com.packt.database.dao;

   import java.util.List;
   import com.packt.database.model.PhoneEntry;

   public interface PhoneBookDao {
       boolean create(PhoneEntry entry);
       boolean update(PhoneEntry entryToUpdate);
       List<PhoneEntry> searchByNumber(String number);
       List<PhoneEntry> searchByFirstName(String firstName);
       List<PhoneEntry> searchByLastName(String lastName);
       boolean delete(String number);
   }
   ```
4. Edit .classpath to add the following Spring dependencies:

```xml
+ commons-lang-3.2.1.jar - DatabaseAccess
+ commons-logging.jar - DatabaseAccess
+ derby.jar - DatabaseAccess
+ junit-4.11.jar - DatabaseAccess
+ mockito-all-1.9.0.jar - DatabaseAccess
+ org.springframework.asm-3.1.1.RELEASE.jar - DatabaseAccess
+ org.springframework.beans-3.1.1.RELEASE.jar - DatabaseAccess
+ org.springframework.core-3.1.1.RELEASE.jar - DatabaseAccess
+ org.springframework.jdbc-3.1.1.RELEASE.jar - DatabaseAccess
+ org.springframework.transaction-3.1.1.RELEASE.jar - DatabaseAccess
```

5. Create a database access interface implementation to communicate with the database. The following are the data access object details:

```java
public class PhoneBookDerbyDao implements PhoneBookDao {

  private String driver = "org.apache.derby.jdbc.EmbeddedDriver";
  private String protocol = "jdbc:derby:";
  private String userId = "dbo";
  private String dbName = "phoneBook";

  public PhoneBookDerbyDao() {
    loadDriver();
  }

  protected void loadDriver() {
    try {
      Class.forName(driver).newInstance();
    } catch (ClassNotFoundException cnfe) {
      cnfe.printStackTrace(System.err);
    } catch (InstantiationException ie) {
      ie.printStackTrace(System.err);
    } catch (IllegalAccessException iae) {
      iae.printStackTrace(System.err);
    }
  }

  protected Connection getConnection() throws SQLException {
    Connection conn = null;
    Properties props = new Properties();
    props.put("user", userId);
```
Getting Familiar with the Spring Framework

```java
conn = DriverManager.getConnection(protocol + dbName + 
";create=true", props);
conn.setAutoCommit(false);
return conn;
}
```

Note that the PhoneBookDerbyDao class is a derby implementation of the DAO. It has configuration attributes such as driver, protocol, and dbName, and getters/setters. The loadDriver() method loads the database driver and gets invoked from the PhoneBookDerbyDao constructor. The getConnection() method connects to a Derby database and establishes a connection.

6. Implement the create behavior:

```java
@Override
public boolean create(PhoneEntry entry) {
    PreparedStatement preparedStmt = null;
    Connection conn = null;
    try {
        conn = getConnection();
        preparedStmt = conn
            .prepareStatement("insert into PhoneBook values 
            (?, ?, ?)"odable);
        preparedStmt.setString(1, entry.getPhoneNumber());
        preparedStmt.setString(2, entry.getFirstName());
        preparedStmt.setString(3, entry.getLastName());
        preparedStmt.executeUpdate();
        // Note that it can cause problems on some dbs if
        // autocommit mode is on
        conn.commit();
        return true;
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        if (preparedStmt != null) {
            try {
                preparedStmt.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```
if (conn != null) {
    try {
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

return false;
}

The `create` method first acquires a database connection and creates a prepared statement from `connection`; it then populates the prepared statement with the `PhoneEntry` values, executes the prepared statement, and then commits the connection. The `finally` block closes the resources, which closes the prepared statement and the connection.

7. Create a class named `PhoneBookDerbySpringDao` that implements the `PhoneBookDao` interface. The following is the Spring implementation of the `create` method:

```java
public class PhoneBookDerbySpringDao implements PhoneBookDao {
    private final JdbcTemplate jdbcTemplate;

    public PhoneBookDerbySpringDao(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public boolean create(PhoneEntry entry) {
        int rowCount = jdbcTemplate.update("insert into PhoneBook values (?, ?, ?)",
                                             new Object[]{entry.getPhoneNumber(),
                                                         entry.getFirstName(),
                                                         entry.getLastName()});
        return rowCount == 1;
    }
}
```
The `JdbcTemplate` class simplifies the use of JDBC; it handles the resources and helps avoid common errors such as not closing the connection. It creates and populates the statement object, iterates through `ResultSet`, leaving the application code to provide SQL and extract results.

PhoneBookDerbySpringDao contains a `JdbcTemplate` instance and delegates the database tasks to `jdbcTemplate`. `JdbcTemplate` uses data source definition from the `applicationContext` file.

`JdbcTemplate` has an `update` method for the insert and update operations. It takes a SQL query and parameters. The new Spring version of the `create()` method invokes the `update()` method on `jdbcTemplate` and passes the `PhoneEntry` details. Now the `create` method looks simple; it is just two lines of code. Spring Framework handles the resource life cycle.

Look at the Spring DAO class; it has only 54 lines. The class looks neat, simple, and readable. It doesn't handle resources; rather, it concentrates on data access.

**Handling a transaction with Spring**

Spring Framework provides supports for transaction management. The following are characteristics of the Spring transaction management framework:

- Offers abstraction for transaction management
- Defines a programming model that supports different transaction APIs, such as JDBC, JTA, and JPA
- Declarative transaction management is supported
- Provides a simpler programmatic transaction management API
- Easily integrates with Spring's data access abstractions

Two transaction management options are available for the J2EE developers. The following are the two options:

- The application server manages global transactions, using the **Java Transaction API (JTA)**. It supports multiple transaction resources, such as database transactions, JMS transactions, and XA transactions.
- Resource-specific local transactions, such as a transaction associated with a JDBC connection.

Both transaction models have downsides. The global transaction needs an application server and JNDI to manage transactions; it uses JTA but the JTA API is cumbersome and has a complex exception model. The need for an application server, JNDI, and JTA limits the reusability of code.
The local transactions have the following disadvantages:

- Cannot handle multiple transactional resources
- Invasive to the programming model

Spring's transaction model solves the problems associated with the global and local transactions, and it offers a consistent programming model for developers that can be used in any environment.

Spring Framework supports both declarative and programmatic transaction management. Declarative transaction management is the recommended one, and it has been well accepted by the development community.

The programmatic transaction model provides an abstraction that can be run over any underlying transaction infrastructure. The concept of transaction strategy is the key to the transaction abstraction. The `org.springframework.transaction.PlatformTransactionManager` interface defines the strategy.

The following is the `PlatformTransactionManager` interface:

```java
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(
            TransactionDefinition definition) throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

The following are the characteristics of `PlatformTransactionManager`:

- `PlatformTransactionManager` is not a class; instead, it is an interface, and thus it can be easily mocked or stubbed to write tests.
- It doesn't need a JNDI lookup strategy, as its implementations can be defined as Spring beans in Spring Framework's IoC container.
- Methods defined in `PlatformTransactionManager` throw `TransactionException`. However, this is an unchecked exception, so programmers are not forced to handle the exception. But in reality, the exception is fatal in nature; when it is thrown, there is very little chance that the failure can be recovered.
- The `getTransaction()` method takes a `TransactionDefinition` parameter and returns a `TransactionStatus` object. The `TransactionStatus` object can be a new or an existing transaction.
The `TransactionDefinition` interface defines the following:

```java
public interface TransactionDefinition {
    int getIsolationLevel();
    int getPropagationBehavior();
    String getName();
    int getTimeout();
    boolean isReadOnly();
}
```

- **Isolation**: This returns the degree of isolation of this transaction from other transactions. The following are the Spring propagations:
  - `ISOLATION_DEFAULT`
  - `ISOLATION_READ_COMMITTED`
  - `ISOLATION_READ_UNCOMMITTED`
  - `ISOLATION_REPEATABLE_READ`
  - `ISOLATION_SERIALIZABLE`

- **Propagation**: This returns the transaction propagation behavior. The following are the allowable values:
  - `PROPAGATION_MANDATORY`: This needs a current transaction and raises an error if no current transaction exists
  - `PROPAGATION_NESTED`: This executes the current transaction within a nested transaction
  - `PROPAGATION_NEVER`: This doesn't support a current transaction and raises an error if a current transaction exists
  - `PROPAGATION_NOT_SUPPORTED`: This executes code non-transactionally
  - `PROPAGATION_REQUIRED`: This creates a new transaction if no transaction exists
  - `PROPAGATIONQUIRES_NEW`: This suspends the current transaction and creates a new transaction
  - `PROPAGATION_SUPPORTS`: If the current transaction exists, then this supports it; otherwise, it executes the code non-transactionally
  - `TIMEOUT_DEFAULT`: This uses the default timeout
• **Timeout**: This returns the maximum time in seconds that the current transaction should take; if the transaction takes more than that, then the transaction gets rolled back automatically.

• **Read-only status**: This returns whether the transaction is a read-only transaction. A read-only transaction does not modify any data.

The `TransactionStatus` interface provides a simple way for transactional code to control the transaction execution and query the transaction status; it has the following signature:

```java
public interface TransactionStatus {
    boolean isNewTransaction();
    void setRollbackOnly();
    boolean isRollbackOnly();
}
```

The `PlatformTransactionManager` implementations normally require knowledge of the environment in which they work, such as JDBC, JTA, Hibernate, and so on.

A local `PlatformTransactionManager` implementation defines a JDBC data source and then uses the Spring `DataSourceTransactionManager` class, which gives it a reference to `DataSource`. The following Spring context defines a local transaction manager:

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

Here, `jdbc.xxxx` represents the values defined in the properties file. Usually, the convention is that the JDBC properties are defined in a properties file that is then loaded from `applicationContext`, and then the JDBC properties are accessed using the key such as `jdbc.xxxxx`. The following is the XML configuration of transaction manager:

```xml
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```
When we use JTA in a J2EE container and use a container DataSource obtained via the JNDI lookup, in conjunction with Spring's JtaTransactionManager, then JtaTransactionManager doesn't need to know about DataSource, or any other specific resources, as it will use the container's global transaction management infrastructure.

The following is the JtaTransactionManager definition in Spring context:

```xml
<jee:jndi-lookup id="dataSource" jndi-name="myDataSource "/>
<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

The benefit of Spring transaction manager is that in all cases, the application code will not need to change at all. We can change how transactions are managed merely by changing the configuration, even if that change means moving from local to global transactions or vice versa.

Declarative transaction management is preferred by most users; it is the option with the least impact on the application code. It is most consistent with the ideals of a non-invasive lightweight container. Spring's declarative transaction management is made possible with Spring AOP.

The similarities between the EJB CMT and Spring declarative transaction are as follows:

- It is possible to specify transaction behavior down to the individual method level
- It is possible to make a setRollbackOnly() call within a transaction context if necessary

**Working with declarative Spring transaction**

We'll create a simple Spring transaction management project and learn about the basics. The following are the steps to create the project:

1. Create an empty class, Foo, under the com.packt.tx package. The following is the class body:

   ```java
   package com.packt.tx;

   public class Foo {
   }
   ```
2. Create an interface, FooService, to handle the CRUD operations on Foo:

```java
package com.packt.tx;

public interface FooService {
    Foo getFoo(String fooName);
    void insertFoo(Foo foo);
    void updateFoo(Foo foo);
}
```

3. Create a default implementation of FooService, and from each method, throw UnsupportedOperationException to impersonate a rollback transaction:

```java
public class FooServiceImpl implements FooService {

    @Override
    public Foo getFoo(String fooName) {
        throw new UnsupportedOperationException();
    }

    @Override
    public void insertFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

    @Override
    public void updateFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

}
```

4. Create an application context file called applicationContextTx.xml directly under the src folder and add the following entries:

Define the fooService bean:

```xml
<bean id="fooService" class="com.packt.tx.FooServiceImpl" />
```

Define a Derby data source:

```xml
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close">
```

<property name="driverClassName" value="org.apache.derby.jdbc.EmbeddedDriver" />
<property name="url" value="jdbc:derby:derbyDB;create=true" />
<property name="username" value="dbo" />
<property name="password" value="" />
</bean>

Define a transaction manager with the data source:

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>

Define an advice with transaction manager so that all get methods will have a read-only transaction:

<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="get*" read-only="true" />
  </tx:attributes>
</tx:advice>

Define the AOP configuration to apply the advice on pointcut:

<aop:config>
  <aop:pointcut id="fooServiceOperation" expression="execution(* com.packt.tx.FooService.*(..))" />
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation" />
</aop:config>
</beans>

5. Create a test class to get the FooService bean and call the getFoo method on the FooService bean. The following is the class:

```java
public class TransactionTest {

  public static void main(String[] args) {
    AbstractApplicationContext context = new ClassPathXmlApplicationContext("applicationContextTx.xml");

    FooService fooService = (FooService) context.getBean("fooService");
```
6. When we run the program, Spring creates a transaction and then rolls back the transaction as it throws `UnsupportedOperationException`. Check the log to get the details. The following is the log:

- Creating new transaction with name `[com.packt.tx.FooServiceImpl.getFoo]`: PROPAGATION_REQUIRED, ISOLATION_DEFAULT, readOnly
- Acquired Connection `[341280385, URL=jdbc:derby:derbyDB, UserName=dbo, Apache Derby Embedded JDBC Driver]` for JDBC transaction
- Setting JDBC Connection `[341280385, URL=jdbc:derby:derbyDB, UserName=dbo, Apache Derby Embedded JDBC Driver]` read-only
- Switching JDBC Connection `[341280385, URL=jdbc:derby:derbyDB, UserName=dbo, Apache Derby Embedded JDBC Driver]` to manual commit
- Bound value `[org.springframework.jdbc.datasource.ConnectionHolder@6b58ba2b]` for key `[org.apache.commons.dbcp2.BasicDataSource@680624c7]` to thread `[main]`
- Initializing transaction synchronization
- Getting transaction for `[com.packt.tx.FooServiceImpl.getFoo]`
- Applying rules to determine whether transaction should rollback on `java.lang.UnsupportedOperationException`
- Winning rollback rule is: null
- No relevant rollback rule found: applying default rules
- Triggering beforeCompletion synchronization
- Initiating transaction rollback
- Rolling back JDBC transaction on Connection `[341280385, URL=jdbc:derby:derbyDB, UserName=dbo, Apache Derby Embedded JDBC Driver]`

**Exploring transaction attributes**

We declared a transaction advice and its attributes in the preceding example. This section examines the transaction attributes such as propagation, isolation, read-only, timeout, and rollback rules.

Transaction propagation has seven levels:

- **PROPAGATION_MANDATORY**: Method should run in a transaction and if nothing exists, an exception will be thrown.
- **PROPAGATION_NESTED**: Method should run in a nested transaction.
Getting Familiar with the Spring Framework

- **PROPAGATION_NEVER**: The current method should not run in a transaction. If this exists, an exception will be thrown.
- **PROPAGATION_NOT_SUPPORTED**: Method should not run in a transaction. The existing transaction will be suspended till the method completes the execution.
- **PROPAGATION_REQUIRED**: Method should run in a transaction. If this already exists, the method will run in that, and if not, a new transaction will be created.
- **PROPAGATION_REQUIRES_NEW**: Method should run in a new transaction. If this already exists, it will be suspended till the method finishes.
- **PROPAGATION_SUPPORTS**: Method need not run in a transaction. If this already exists, it supports one that is already in progress.

The following are the isolation levels:

- **ISOLATION_DEFAULT**: This is the default isolation specific to the data source.
- **ISOLATION_READ_UNCOMMITTED**: This reads changes that are uncommitted. This leads to dirty reads, phantom reads, and non-repeatable reads.

  A dirty read happens when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.

  Data getting changed in the current transaction by other transactions is known as a phantom read.

  A non-repeatable read means data that is read twice inside the same transaction cannot be guaranteed to contain the same value.

- **ISOLATION_READ_COMMITTED**: This reads only committed data. Dirty reads are prevented but repeatable and non-repeatable reads are possible.
- **ISOLATION_REPEATABLE_READ**: Multiple reads of the same field yield the same results unless modified by the same transaction. Dirty and non-repeatable reads are prevented but phantom reads are possible as other transactions can edit the fields.
- **ISOLATION_SERIALIZABLE**: Dirty, phantom, and non-repeatable reads are prevented. However, this hampers the performance of the application.

The read-only attribute specifies that the transaction is only going to read data from a database. It can be applied to only those propagation settings that start a transaction, that is, **PROPAGATION_REQUIRED**, **PROPAGATION_REQUIRES_NEW**, and **PROPAGATION_NESTED**.
The timeout specifies the maximum time allowed for a transaction to run. This is required for the transactions that run for very long and hold locks for a long time. When a transaction reaches the timeout period, it is rolled back. The timeout needs to be specified only on propagation settings that start a new transaction.

We can specify that transactions will roll back on certain exceptions and do not roll back on other exceptions by specifying the rollback rules.

**Using the @Transactional annotation**

The functionality offered by the `@Transactional` annotation and the support classes is only available in Java 5 (Tiger) and above. The `@Transactional` annotation can be placed before an interface definition, a method on an interface, a class definition, or a public method on a class. A method in the same class takes precedence over the transactional settings defined in the class-level annotation.

The following example demonstrates the method-level precedence:

```java
@.Transactional(readOnly = true)
public class FooServiceImpl implements FooService {
    public Foo getFoo(String fooName) {
    }
    // This settings has precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
    }
}
```

However, the mere presence of the `@Transactional` annotation is not enough to actually turn on the transactional behavior; the `@Transactional` annotation is simply metadata that can be consumed by something that is aware of `@Transactional` and that can use the metadata to configure the appropriate beans with the transactional behavior.

The default `@Transactional` settings are as follows:

- The propagation setting is `PROPAGATION_REQUIRED`
- The isolation level is `ISOLATION_DEFAULT`
- The transaction is read/write
The transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported.

Any `RuntimeException` will trigger a rollback and any checked exception will not trigger a rollback.

When the previous POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding one line of XML configuration. We’ll examine the `@Transactional` annotation in the following example:

1. Create an application context file called `applicationContextTxAnnotation.xml` and add the following lines (no need for AOP and advice):

   ```xml
   <context:annotation-config />
   <bean id="fooService" class="com.packt.tx.FooServiceImpl"/>

   <!-- enable the configuration of transactional behavior based on annotations -->
   <tx:annotation-driven transaction-manager="txManager" />

   <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
         destroy-method="close">
     <property name="driverClassName" value="org.apache.derby.jdbc.EmbeddedDriver"/>
     <property name="url" value="jdbc:derby:derbyDB;create=true"/>
     <property name="username" value="dbo"/>
     <property name="password" value=""/>
   </bean>

   <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
     <property name="dataSource" ref="dataSource"/>
   </bean>

2. Annotate `FooServiceImpl` with the `@Transactional` annotation:

   ```java
   @Transactional
   public class FooServiceImpl implements FooService {

   @Override
   public Foo getFoo(String fooName) {
     throw new UnsupportedOperationException();
   }
   ```
@Override public void insertFoo(Foo foo) {
    throw new UnsupportedOperationException();
}

@Override public void updateFoo(Foo foo) {
    throw new UnsupportedOperationException();
}

3. Create a class called TransactionTestAnnotation, load applicationContextTxAnnotation, and examine whether the same log appears. The following is the class:
   public class TransactionTestAnnotation {

       public static void main(String[] args) {
           AbstractApplicationContext context = new ClassPathXmlApplicationContext("applicationContextTxAnnotation.xml");

           FooService fooService = (FooService) context.getBean("fooService");
           System.out.println(fooService);
           fooService.getFoo(null);
       }
   }

**Working with a programmatic Spring transaction**
Spring provides two means of programmatic transaction management:

- Using TransactionTemplate
- Using a PlatformTransactionManager implementation directly

The Spring team generally recommends the first approach (using TransactionTemplate).

The second approach is similar to using the JTA UserTransaction API (although exception handling is less cumbersome).
Using TransactionTemplate

The following are the characteristics of TransactionTemplate:

- It adopts the same approach as other Spring templates such as JdbcTemplate and HibernateTemplate
- It uses a callback approach
- A TransactionTemplate instance is threadsafe

The following code snippet demonstrates TransactionTemplate with a callback:

```java
Object result = transTemplate.execute(new TransactionCallback()
    {
        public Object doInTransaction(TransactionStatus status) {
            updateOperation();
            return resultOfUpdateOperation();
        }
    });
```

If there is no return value, use the convenient TransactionCallbackWithoutResult class via an anonymous class, as follows:

```java
transTemplate.execute(new TransactionCallbackWithoutResult()
    {
        protected void doInTransactionWithoutResult(TransactionStatus status) {
            updateOperation1();
            updateOperation2();
        }
    });
```

Application classes wishing to use TransactionTemplate must have access to PlatformTransactionManager, which will typically be supplied to the class via a dependency injection. It is easy to unit test such classes with a mock or stub PlatformTransactionManager. There is no JNDI lookup here; it is a simple interface. As usual, you can use Spring to greatly simplify your unit testing.

Using PlatformTransactionManager

A PlatformTransactionManager implementation can be directly used to manage a transaction:

1. Simply pass the implementation of the PlatformTransactionManager to your bean via a bean reference.
2. Then, using the TransactionDefinition and TransactionStatus objects, you can initiate transactions and perform a rollback or commit.
Building an MVC application with Spring

The Model View Controller (MVC) is a widely used web development pattern. The MVC pattern defines three interconnected components, namely model, view, and controller.

The model represents the application data, logic, or business rules.

The view is a representation of information or a model. A model can have multiple views, for example, marks of a student can be represented in a tabular format or graphical chart.

The controller accepts client requests and initiates commands to either update the model or change the view.

The controller controls the flow of the application. In JEE applications, a controller is usually implemented as a servlet. A controller servlet intercepts requests and then maps each request to an appropriate handler resource. In this section, we will build a classic MVC front controller servlet to redirect requests to views.
Spring MVC is a web application framework that takes advantage of Spring design principles:

- Dependency injection
- Interface-driven design
- POJO without being tied up with a framework

Spring MVC is used for the following advantages:

- Testing through dependency injection
- Binding of request data to domain objects
- Form validation
- Error handling
- Multiple view technologies
- Supports different formats such as JSP, Velocity, Excel, and PDF
- Page workflow

In Spring MVC, the following is a simplified request-handling mechanism:

1. DispatcherServlet receives a request and confers with handler mappings to find out which controller can handle the request, and it then passes the request to that controller.

2. The controller performs the business logic (can delegate the request to a service or business logic processor) and returns some information back to DispatcherServlet for user display/response. Instead of sending the information (model) directly to the user, the controller returns a view name that can render the model.

3. DispatcherServlet then resolves the physical view from the view name and passes the model object to the view. This way DispatcherServlet is decoupled from the view implementation.

4. The view renders the model. A view can be a JSP page, a servlet, a PDF file, an Excel report, or any presentable component.
The following sequence diagram represents the flow and interaction of Spring MVC components:

We will build a Spring web application and unit test code using JUnit by performing the following steps:

1. Launch Eclipse and create a dynamic web project called SpringMvcTest.
2. Open web.xml and enter the following lines:

```xml
<display-name>SpringMVCTest</display-name>
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/dispatcher-servlet.xml
    </param-value>
</context-param>
</web-app>
```
Getting Familiar with the Spring Framework

The dispatcher is a DispatcherServlet and it maps all requests. Note the contextConfigLocation parameter. This indicates that the Spring beans are defined in /WEB-INF/dispatcher-servlet.xml.

3. Create an XML file called dispatcher-servlet.xml in WEB-INF and add the following lines:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
   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
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">
   <context:component-scan base-package="com.packt" />
   <bean class= "org.springframework.web.servlet.view.InternalResourceViewResolver">
     <property name="prefix">
       <value>/WEB-INF/pages/</value>
     </property>
     <property name="suffix">
       <value>.jsp</value>
     </property>
   </bean>
</beans>
```

This XML defines a Spring view resolver. Any view will be found under the /WEB-INF/pages location with the .jsp suffix, and all beans are configured under the com.packt package with Spring annotations.

4. Create a LoginInfo class in the com.packt.model package. This class represents the login information. Add two private string fields, userId and password, generate getters and setters

5. Create a JSP page called login.jsp under /WEB-INF/pages and add the following lines to create a form using the Spring tag library. Modify the form and add normal HTML input for username and password:

```jsp
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<sf:form method="POST" modelAttribute="loginInfo" action="/onLogin">

</sf:form>
```
6. Create a controller class called com.packt.controller.LoginController to handle the login request. Add the following lines:

```java
@Controller
@Scope("session")
public class LoginController implements Serializable {
    @RequestMapping({ "/", "/login" })
    public String onStartUp(ModelMap model) {
        model.addAttribute("loginInfo", new LoginInfo());
        return "login";
    }
}
```

The @Controller annotation indicates that the class is a Spring MVC controller class. In sample-servlet.xml, we defined `<context:component-scan base-package="com.packt" />`, so Spring will scan this @Controller annotation and create a bean. @RequestMapping maps any request with the default path /SpringMvcTest/ or /SpringMvcTest/login to the onStartUp method. This method returns a view named login. The view resolver defined in the XML file will map the login request to /WEB-INF/pages/login.jsp page.

7. Create another method in the Login class to handle the login submit request:

```java
@RequestMapping({ "/onLogin" })
public String onLogin(@ModelAttribute("loginInfo") LoginInfo loginInfo, ModelMap model) {
    LoginInfo loginInfo, ModelMap model) {
        if(!"junit".equals(loginInfo.getUserId())) {
            model.addAttribute("error", "invalid login name");
            return "login";
        }
        if(!"password".equals(loginInfo.getPassword())) {
            model.addAttribute("error", "invalid password");
            return "login";
        }
        model.addAttribute("name", "junit reader!");
        return "greetings";
    }
```

The method is mapped with /onLogin. @ModelAttribute("loginInfo") is the model submitted from the login.jsp form. This method checks whether the username is junit and password is password. If the user ID or password does not match, then an error message is shown in the login page; otherwise, the greetings view is opened.
8. Change `login.jsp` to submit the form to `/SpringMvcTest/onLogin`, and the modelattribute name is `loginInfo`:

```html
<sf:form method="POST" modelAttribute="loginInfo" action="/SpringMvcTest/onLogin">

Also, add the following JSTL expression to display the error message:

```html
<h1>${error}</h1>
```

9. Create a JSP file called `greetings.jsp` and add the following lines:

```html
<h1>Hello :${name}</h1>
```

10. In the browser, enter `http://localhost:8080/SpringMvcTest/`. This will open the login page. In the login page, do not enter any value; just hit Submit. It will show the error message Invalid login name. Now, enter `junit` in the user Id field and password in the Password field and hit Enter; the application will greet you with following message:

```
Hello :junit reader!
```

**Resources:** Spring Framework Reference Documentation

**Summary**

This chapter covered the Spring basics. It discussed the Spring projects and in particular Spring Framework. It explored the Spring container, Spring bean life cycle, dependency injection, AOP, Spring MVC, and Spring transaction management.

The next chapter will focus on getting the reader quickly started with JUnit 4 and the Mocking framework. It provides an overview of JUnit testing and explores the Mockito APIs.
Where to buy this book
You can buy Mockito for Spring from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.