Core Spring Framework Workshop

Version 1.0.4 June 2016



Copyright © Descriptor Systems, 2016
All Rights Reserved.

This document may not be copied or reproduced in any form without prior written consent of Joel Barnum of Descriptor Systems.

All trademarks belong to their respective companies.

You can contact Descriptor Systems at:

http://www.descriptor.com
sales@descriptor.com
319-362-3906
P.O. Box 461
Marion, IA 52302

Core Spring Framework Workshop

Chapter 1: Introduction to the Course

Core Spring Framework Workshop 1-1
Legal Information 1-2
Core Spring Framework Workshop 1-3
Introductions 1-4
Course Description 1-5
Course Objectives 1-6
Sample Agenda 1-7
Sample Agenda, cont'd 1-8
Sample Agenda, cont'd 1-9
Course Logistics 1-10

Chapter 2: Introduction to the Spring Framework

Introduction to the Spring Framework 2-1
What is the Spring Framework? 2-2
History of Spring 2-3
Spring Fundamentals 2-4
Spring Modules 2-5
Writing Spring Applications 2-6
What is a Spring Bean? 2-7
The Spring Container 2-8
Configuring the Container 2-9
Programming To Interfaces 2-10
Dependent Objects Without Spring 2-11
Introduction to Dependency Injection 2-12
Configuring with Annotations 2-13
Configuring with Java Config 2-14
Configuring with Java Config, cont'd 2-15
Configuring with XML 2-16
Injecting Simple Values 2-17
Configuring Annotation-Based Apps 2-18

Creating the Container (Annotation or XML)....2-19
Creating the Container (Java Configuration)....2-20
Steps for a Simple Spring Application....2-21
Hello World From Spring....2-22
Chapter Summary....2-23

Chapter 3: Spring Annotations

Spring Annotations....3-1
Configuring Spring Beans....3-2
Introduction to Annotations....3-3
Annotations and Spring....3-4
Enabling Annotations....3-5
The @Value Annotation....3-6
Using Annotations for Dependency Injection....3-7
The @Component Annotation....3-8
The @Resource Annotation: By Name....3-9
The @Resource Annotation: By Type....3-10
Autowiring with Annotations....3-11
@Resource vs @Autowired....3-12
The @Qualifier Annotation....3-13
Other Annotations....3-14
Spring and JSR-330....3-15
Chapter Summary....3-16

Chapter 4: Spring Java Configuration

Spring Java Configuration....4-1
Configuring Spring Beans....4-2
Introduction to Java-Based Configuration....4-3
Java-Based Configuration Example....4-4
Why Use Java Configuration?....4-5
The @Bean Annotation....4-6
Performing Setter Injection....4-7
Referencing Other Beans....4-8
Performing Constructor Injection....4-9
Using Autowiring in a Configuration Class....4-10

Autowiring Bean-Creation Methods....4-11
Multiple @Configuration Classes....4-12
Referencing Multiple Configuration Classes....4-13
Importing a Configuration....4-14
Importing Other Configuration Types....4-15
Complete Example....4-16
Using the Groovy DSL for Configuration....4-17
What is Groovy?....4-18
Groovy DSL Example....4-19
Chapter Summary....4-20

Chapter 5: Spring Beans

Spring Beans....5-1
Configuring Spring Beans....5-2
Bean Scopes....5-3
Singleton Scope....5-4
Prototype Scope....5-5
Singleton vs Prototype Java Config....5-6
Bean Lifecycle....5-7
Lifecycle Annotations....5-8
Lifecycle With Java Config....5-9
Creation Lifeycle....5-10
Creation Lifeycle Example....5-11
Destruction Lifecycle....5-12
Destruction Lifecycle Example....5-13
BeanPostProcessors....5-14
Coding A BeanPostProcessor....5-15
BeanPostProcessor Example....5-16
Factory Methods....5-17
Factory Classes....5-18
Factory Classes, cont'd....5-19
Chapter Summary....5-20

Chapter 6: XML Dependency Injection

XML Dependency Injection....6-1

Introduction to Dependency Injection....6-2
Configuring Dependency Injection....6-3
Inversion of Control....6-4
Sample Application for this Chapter....6-5
Setter Injection....6-6
Property Conversions....6-7
Constructor Injection....6-8
Constructor Injection Resolution....6-9
Constructor Injection Resolution, cont'd....6-10
Mixing Injection Types....6-11
Setter vs Constructor Injection....6-12
Chapter Summary....6-13

Chapter 7: Introduction to Spring MVC

Introduction to Spring MVC....7-1
Spring and the Web....7-2
The Spring WebApplicationContext....7-3
The Spring WebApplicationContext, cont'd....7-4
Introduction to Spring MVC....7-5
What is the MVC Design Pattern?....7-6
Spring MVC Architecture....7-7
Spring MVC Controllers....7-8
Form Processing Flow: Phase 1....7-9
Form Processing Flow: Phase 2....7-10
Spring MVC Development Steps....7-11
Configuring the Web Application....7-12
A Service Bean....7-13
A Spring MVC Input Form....7-14
A Spring MVC Result Page....7-15
Configuring Controller Annotation Processing....7-16
What is a ViewResolver?....7-17
A Spring MVC Command Controller....7-18
The Spring Configuration File....7-19
MVC With Java Configuration....7-20

Using Thymeleaf Instead of JSP 7-21
Thymeleaf/Spring Recipe 7-22
Chapter Summary 7-23

Chapter 8: Spring and JDBC

Spring and JDBC 8-1
The DAO Design Pattern 8-2
What's Wrong with JDBC? 8-3
Introducing the Spring JDBC Module 8-4
The JdbcTemplate Class 8-5
Spring JDBC Exceptions 8-6
JDBC Connection Review 8-7
Working with Data Sources in Spring 8-8
Configuring a DriverManager Source (XML) 8-9
Configuring a DriverManager (Java Config) 8-10
Configuring a JNDI-Based DataSource (XML) 8-11
Configuring JNDI (Java Config) 8-12
Writing a DAO 8-13
JdbcTemplate Query Methods 8-14
Example: Querying Row Count 8-15
Mapping Rows to Objects 8-16
Example: Query All 8-17
Example: Find Student By ID 8-18
JdbcTemplate Insert, Update and Delete 8-19
Example: Insert 8-20
Complete DAO Example 8-21
Chapter Summary 8-22

Chapter 9: Introduction to Spring AOP

Introduction to AOP 9-1
What is Aspect-Oriented Programming? 9-2
AOP Terminology 9-3
Spring Support for AOP 9-4
Steps for Using Spring AOP 9-5
Defining a Pointcut 9-6

Basic Pointcut Syntax....9-7
Applying Advice....9-8
Configuring Spring AOP....9-9
Spring AOP Behind the Scenes....9-10
Complete Hello, World Example....9-11
Chapter Summary....9-12

Chapter 10: Transactions in Spring

Transactions in Spring....10-1
Introduction to Transactions....10-2
A Program with No Transactions....10-3
A Program with Transactions....10-4
ACID Transactions....10-5
Global and Local Transactions....10-6
Distributed Transactions....10-7
Spring and Transactions....10-8
Spring Transaction Managers....10-9
Configuring a DataSource Transaction Manager (XML)....10-10
Configuring a DataSource Transaction Manager (Java Config)....10-11
Configuring a JTA Transaction Manager (XML)....10-12
Configuring a JTA Transaction Manager (Java Config)....10-13
Using a Vendor-Specific Transaction Manager....10-14
Transaction Demarcation and Management....10-15
Using Declarative Transaction Management....10-16
Spring Transaction Exception Handling....10-17
Transaction Propagation....10-18
Transaction Propagation Attributes....10-19
Transaction Propagation Example....10-20
A Typical Architecture....10-21
Exceptions and Rolling Back....10-22
Transaction Timeouts....10-23
Transaction Isolation....10-24
Transaction Isolation Levels....10-25
Read Only Transactions....10-26

Specifying Transaction Attributes....10-27
Transaction Annotations....10-28
The @Transactional Annotation....10-29
Complete Declarative Transaction Example....10-30
Chapter Summary....10-31

Chapter 11: Testing Spring Applications

Testing Spring Applications....11-1
Spring Testing Issues....11-2
Unit Testing Spring Beans....11-3
Example: Simple Unit Test with JUnit4....11-4
Integration Testing Spring Beans....11-5
Integration Test Example....11-6
Injecting Spring Beans....11-7
Using Tests with Transactions and Database....11-8
Transaction Test Example....11-9
Chapter Summary....11-10

Chapter 12: Spring Remoting with RMI, HttpInvoker and JMS

Spring Remoting with RMI, HttpInvoker and JMS....12-1
Introduction to Remote Objects....12-2
Introduction to RMI....12-3
RMI Shortcomings....12-4
Spring RMI Support....12-5
Spring RMI Example....12-6
Client/Server Firewall Issues....12-7
Spring HTTP Invoker Support....12-8
Spring HttpInvoker Example....12-9
What is the Java Message Service?....12-10
Point to Point Messaging....12-11
Publish and Subscribe Messaging....12-12
JMS Message Types....12-13
JMS Shortcomings....12-14
Spring and JMS....12-15
The JmsTemplate Type....12-16

Accessing JNDI Resources 12-17
Configuring the JmsTemplate 12-18
JmsTemplate Examples 12-19
Converting Messages 12-20
Subclassing JmsGatewaySupport 12-21
Receiving Messages 12-22
The MessageListener Interface 12-23
Configuring a Message Listener Container 12-24
POJO Listener Example 12-25
Chapter Summary 12-26

Chapter 13: Introduction to Spring Boot

Introduction to Spring Boot 13-1
What is Spring Boot? 13-2
Building Spring Boot Projects 13-3
Generating a Starter Project 13-4
Gradle Project Structure 13-5
The Generated Build Script 13-6
Generating an Eclipse Project 13-7
The 'main' Program 13-8
Spring Boot Autoconfiguration 13-9
Configuring SpringMVC JSP Views 13-10
Running the Project 13-11
Complete Example 13-12
Chapter Summary 13-13

Chapter 14: Introduction to REST

Introduction to REST 14-1
What is REST? 14-2
REST vs SOAP 14-3
Principles of REST 14-4
Resource IDs 14-5
Resources With Multiple Representations 14-6
Use Links to Connect Resources 14-7
Use Standard HTTP Methods 14-8

Stateless Communication 14-9
REST in Java 14-10
What is Restlet? 14-11
A Simple Restlet Resource 14-12
JAX-WS and REST 14-13
JAX-RS 14-14
A JAX-RS Service 14-15
Spring MVC REST 14-16
Chapter Summary 14-17

Chapter 15: RESTful CRUD Services

RESTful CRUD Services 15-1
The Richardson Maturity Model 15-2
RESTful CRUD Service Overview 15-3
HTTP Review 15-4
Media Types 15-5
Implementing Create 15-6
Implementing Retrieve 15-7
Implementing Update 15-8
Implementing Delete 15-9
Complete JAX-RS CRUD Example 15-10
Chapter Summary 15-11

Chapter 16: Spring and REST

Spring and REST 16-1
Spring Support for REST 16-2
Message Converters 16-3
Content Negotiation 16-4
Negotiation via HTTP Accept Header 16-5
Spring REST Annotations 16-6
A Simple RESTful Controller 16-7
Responding to POST Requests 16-8
Responding to PUT Requests 16-9
Configuring a Spring REST Application 16-10
Chapter Summary 16-11

Core Spring Framework Workshop

Version 1.0.4 June 2016



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Legal Information

Copyright (C) Descriptor Systems 2001, 2016

All Rights Reserved

All trademarks are owned by their respective companies

You can contact Descriptor Systems at:
http://www.descriptor.com
jbarnum@descriptor.com

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Core Spring Framework Workshop

- Introductions
- Objectives

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introductions

- Please introduce yourself:
  - Name:
  - Java Background:
  - Spring Background (if any):
  - Your goal for the course:

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Course Description

Description: This course covers how to write programs that use the Spring and Framework.

Prerequisites: Basic Java programming experience is assumed.

Audience: Business analysts, developers, managers and other people interested in learning how to program Spring.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Course Objectives

After taking this course, you will be able to:

- Write Spring programs that use dependency injection
- Understand the how the Spring container manages objects
- Write Web applications using the SpringMVC module
- Write Spring classes that access relational data

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Agenda

Day 1
----

1. Introduction to the Course

2. Introduction to the Spring Framework

3. Additional Configuration and Annotations

4. Java Configuration

5. Spring Beans

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Agenda, cont'd

Day 2
----

6. XML Dependency Injection

7. Introduction to Spring MVC

8. Spring and JDBC

9. Introduction to Spring AOP

10. Transactions in Spring

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Agenda, cont'd

Day 3
---

11. Testing Spring Applications

12. Spring Remoting with RMI, HttpInvoker and JMS

13. Introduction to Spring Boot

14. Introduction to REST

15. RESTful CRUD Services

16. Spring and REST

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Course Logistics

- Schedule
  - Start time
  - Lunch
  - End time
  - Breaks

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to the Spring Framework

- What is the Spring Framework?
- Spring Architecture
- Spring Hello, World

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is the Spring Framework?

- The **Spring Framework** is an open-source, Java and JEE framework that aims to make developing applications easier
- Spring applications are based on **plain old Java objects** (POJO) also known as JavaBeans
- Objects in a Spring application should not depend on Spring and should be easy to test



The homepage for the Spring Framework is at:

http://www.springsource.org/

Note that Spring is also available for Microsoft's .NET.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

History of Spring

- The Spring Framework was created in 2003 by Rod Johnson based on material from his book *Expert One-on-One J2EE Design and Development*
- Spring 2 was released in 2006 and added basic support for Java 5 and configuration extensions
- Spring 2.5 adds support for Java 5 annotations
- Spring 3 contains support for RESTful Web services and other new features including **Java-based configuration**
- Spring 4 adds support for Java 8 and removes many old deprecated classes and methods

Rod Johnson publicized his new framework with a post to theserverside.com:
http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Fundamentals

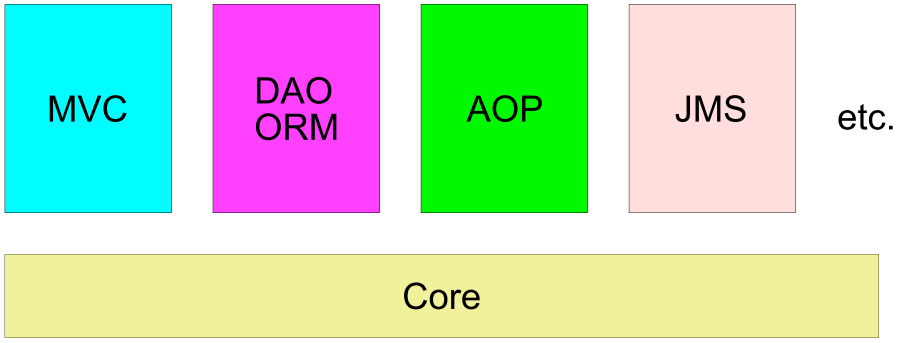
- Use of Spring beans (POJOs) instead of requiring classes to implement framework interfaces
- Loose coupling between objects
- **Dependency injection** (DI) to give objects resources instead of requiring lookups
- **Aspect-oriented programming** (AOP) to free objects from performing system functions such as logging and transactions
- Spring doesn't re-invent the wheel, but can integrate with other frameworks (e.g. Java Persistence)

Dependency injection is also known as "Inversion of Control" (IoC).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Modules

- The Spring Framework itself is modular so you can use only the pieces needed by your application
- This makes the Spring runtime lightweight



This notion of modular design allows Spring developers to use only the pieces they need for their application.

Note that there are additional modules not shown here.

Many developers use a build tool such as Maven or Gradle to help manage the Spring JARs and their dependencies.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Writing Spring Applications

- To write Spring programs, you:
  - Write Spring **bean** classes
  - Configure the **container** so it can manage and initialize the beans
  - Bootstrap the container, retrieve bean references and write application code

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is a Spring Bean?

- Unlike in earlier frameworks, Spring **beans** don't need to implement any non-domain interfaces
- These POJOs are like JavaBeans, though they don't need to implement Serializable and such

```
1    @Component
2    public class Student
3    {
4       private int id;
5       . . .
6
7       public int getId() {return id;}
8       public void setId(int i) {id = i;}
9       . . .
10   }
```

One of the "knocks" against frameworks like EJB2 and Struts is that they require you to write classes that implement framework interfaces and then require the framework's runtime behavior to exist. This coupling between the objects and the framework makes it difficult to re-use business objects and also makes testing outside of the runtime difficult or impossible.

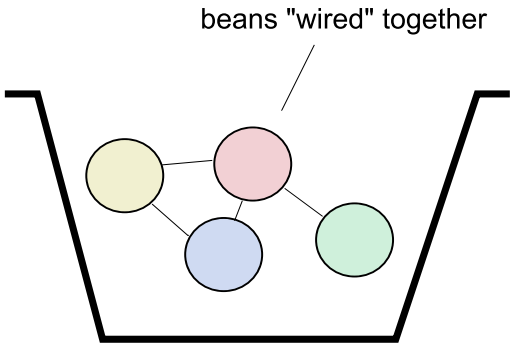
Spring avoids that by allowing the use of POJOs that have no dependency on Spring itself, except perhaps for the @Component annotation that names the bean.

JavaBeans is a native Java specification that defines the notion of Java components. Spring POJOs typically follow a relaxed version of the full JavaBean specification.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Spring Container

- To manage the lifecycle of the beans, Spring provides a lightweight **container**
- Spring apps create the container, then retrieve bean references from the container
- The container can **wire**, or hook up beans to each other via *dependency injection*



A Spring container

A container is a set of runtime that performs useful tasks. In Spring, one such useful task is ensure that beans have their dependencies satisfied. This is quite different from traditional JEE, where objects use JNDI to find their own dependencies. Here, the container injects the dependencies so the object doesn't need to do any look ups.

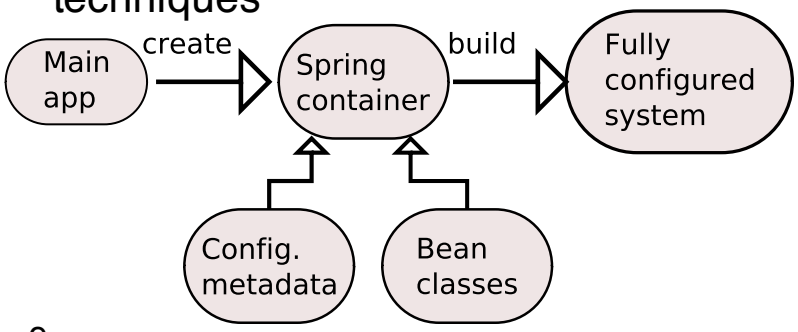
Beans can have references to other beans or to external resources. The container can connect a bean to its referenced objects as part of creating the set of managed objects.

This process, known as "wiring" is typically configured using annotations or in an XML file that the container reads when it starts.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring the Container

- Spring provides three* ways to configure Spring beans:
  - **Annotations** - Markup bean classes themselves
  - **Java configuration** - Write one or more classes that perform configuration
  - **XML** - Write one or more XML configuration files
- Spring applications can use any mixture of these techniques



---

* - Spring 4 added the ability to use the Groovy language both for writing and configuring Spring beans.

XML-based configuration was the first approach and was the only way to configure in early versions of Spring. It has the advantage that you can keep your beans configuration-free and consolidate all of the configuration, but does require you to learn and use XML.

Annotation-based configuration reduces the amount of XML required.

Java-based configuration is relatively new. It has the advantage that you can keep your beans configuration-free and perform configuration in the same language that you are using to write the beans (Java).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Programming To Interfaces

- To decrease coupling between objects, Spring developers often start by writing an **interface** for some of their beans
- This eases maintenance, and makes beans easier to test

```
1    public interface MyService
2    {
3        public void myMethod();
4    }
5
6    public class MyClass implements MyService
7    {
8       public void myMethod() {...};
9    }
10
11   public class OtherClass
12   {
13      private MyService theService;
14   . . .
15   }
```


Writing both an interface and an implementation is not required by Spring, but it's considered a best practice.

Note that dependent classes maintain a reference that's typed to the interface, not the implementation.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Dependent Objects Without Spring

- Most real-world objects use other objects; we refer to this as a **dependency**
- In non-Spring environments, objects typically satisfy their own dependencies

```
1    public class OtherClass
2    {
3       // MyService is an interface
4       private MyService theService;
5
6       public OtherClass()
7       {
8          // MyClass implements MyService
9          theService = new MyClass();
10         . . .
11      }
12   }
```

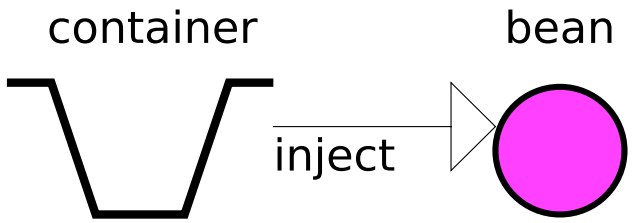
Here we show the OtherClass constructor explicitly and manually satisfying a dependency to a MyService object.

While this is straightforward and simple, it presents a few problems, most notably that OtherClass must know the name of the MyService implementation class. That makes it difficult to subsitute some other implementation, perhaps for testing.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Dependency Injection

- **Dependency injection** is a technique in which the container instantiates and provides references for beans
- To inject a bean reference, the container directly accesses a field or calls a bean's "set" method or constructor
- DI lessens coupling between beans and decouples beans from services such as JNDI



Dependency injection is a form of Inversion of Control</i> where the container supplies a bean with its dependencies.

The term "Dependency injection" was coined by Martin Fowler.

Spring developers typically write annotations and/or an XML file to configure their beans and how the Spring container should inject dependencies. Another option is to use the "Java configuration" technique added in Spring 3.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring with Annotations

- Spring lets you configure injection using **annotations**

```
1    package sample;
2
3    @Component("myService")
4    public class MyClass implements MyService
5    {
6       public void myMethod() {...};
7    }
```

```
1    package sample;
2
3    @Component("other")
4    public class OtherClass
5    {
6       @Resource(name="myService")
7       private MyService theService;
8       public void setTheService(MyService svc) {...}
9    . . .
10   }
```

Here we show injecting the MyClass reference into the OtherClass bean.

Note how in the Java code, OtherClass doesn't know about the implementation class, only the interface.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring with Java Config

- Java configuration is the newest approach and lets you configure using the same language you use to write the beans themselves
- To use Java config, create one or more Java classes annotated with **@Configuration** containing bean-creation methods annotated with **@Bean**

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring with Java Config, cont'd

```
1    @Configuration
2    public class MyConfig
3    {
4      @Bean
5      public MyService myService()
6      {
7         return new MyClass();
8      }
9
10     @Bean
11     public OtherClass otherClass()
12     {
13       OtherClass other = new OtherClass();
14       other.setTheService(myService());
15       return other;
16     }
17   }
```

Here we provide two bean-creation methods to create the same beans we configured with annotations and XML-based config.

Note in the otherClass() method, we perform dependency injection by calling the myService() method and passing its return to the OtherClass bean's setTheService() method.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring with XML

- XML configuration was the original configuration approach in Spring

```
1    <!-- sample.xml -->
2
3    <?xml version="1.0" encoding="UTF-8"?>
4    <beans ...>
5
6      <bean id="myService" class="sample.MyClass"/>
7
8      <bean id="other" class="sample.OtherClass">
9        <property name="theService" ref="myService" />
10     </bean>
11
12   </beans>
```


In older versions of Spring, there was no support for annotations, and most developers used XML. Even in modern Spring, some developers still prefer the XML approach since it lets you perform all of the configuration in a central location.

Note how the property name, "theService" maps to the "setTheService()" method in the bean. When using XML injection, Spring can call such "set" methods to perform the injection. With annotations, Spring doesn't need the "set" methods.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Injecting Simple Values

- You can use annotations, XML or Java config to intialize properties to values

```
1    // using annotations
2    public class MyClass
3    {
4       @Value("1234")
5       private int myInteger;
6       public void setMyInteger(int myInt) {...}
7    }
```

```
1    <!-- using XML -->
2
3    <bean id="myBean" class="sample.MyClass">
4      <property name="myInteger" value="1234" />
5    </bean>
```

Whether you use annotations or XML, Spring must convert the supplied textual value to the appropriate Java type. If it can't convert, the container throws org.springframework.beans.factory.BeanCreationException.

If you plan to use the XML approach, the "set" method is required, but not if you use annotations.

The @Value annotation is from the org.springframework.beans.factory.annotation package.

Injecting values with Java config is trivial – just call the "setter" method.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Annotation-Based Apps

- To enable Spring to scan your beans for annotations, you need to provide at least a minimal XML file

```
1    <!-- sample.xml -->
2
3    <?xml version="1.0" encoding="UTF-8"?>
4    <beans ...>
5
6      <context:component-scan base-package="sample"/>
7
8    </beans>
```

This simple XML file configures Spring so it can process the @Component and @Resource annotations shown previously. Note how Spring needs to know which packages to scan for the annotations; here the MyClass and OtherClass classes are in the "sample" package.

We don't need the component-scan element if we use XML or Java configuration to configure the beans.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Creating the Container (Annotation or XML)

- Spring applications with a *main* method can create a Spring container and then retrieve beans from it
- Spring provides several techniques for creating a container - here we retrieve the configuration file from the file system and use it to create the container

```
1    AbstractApplicationContext ctx =
2     new FileSystemXmlApplicationContext("sample.xml");
3
4    OtherClass oc =
5       (OtherClass)ctx.getBean("other");
6
7    . . .
8
9    ctx.close();
```

The AbstractApplicationContext interface specifies the basic functionality of a Spring container. Note that once we've created the container, we can use it to retrieve managed bean references.

The imports for this program might look like:

```
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Creating the Container (Java Configuration)

- If you are using Java configuration, you do not need any XML configuration - just reference the configuration class

```
1    AnnotationConfigApplicationContext ctx =
2      new AnnotationConfigApplicationContext(
3         MyConfig.class);
4
5    OtherClass oc =
6       (OtherClass)ctx.getBean("other");
7
8    . . .
9
10   ctx.close();
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Steps for a Simple Spring Application

- As desired, create interfaces for bean classes
- Implement the beans, using dependency injection as required
- Write an XML configuration file (if necessary)
- Write a class with a *main* method that creates a container and retrieves bean instances
- Invoke the beans' methods to perform the application function

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Hello World From Spring

HelloService.java

HelloImpl.java

Greeter.java

HelloApp.java

hello.xml

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package hello;
2
3    public interface HelloService
4    {
5        public void sayHello();
6    }
```

```
1    package hello;
2
3    import org.springframework.stereotype.Component;
4
5    @Component("helloService")
6    public class HelloImpl implements HelloService
7    {
8        public void sayHello()
9        {
10           System.out.println("Hello, from Spring");
11       }
12   }
```

```
1    package hello;
2
3    import javax.annotation.Resource;
4
5    import org.springframework.stereotype.Component;
6
7    @Component("greeter")
8    public class Greeter
9    {
10       @Resource(name="helloService")
11       private HelloService helloService;
12
13       public void sendGreetings()
14       {
15           helloService.sayHello();
16       }
17   }
```

```
1    package hello;
2
3    import org.springframework.context.support.AbstractApplicationContext;
4    import org.springframework.context.support.FileSystemXmlApplicationContext;
5
6    public class HelloApp
7    {
8        public static void main(String[] args) throws Exception
9        {
10           AbstractApplicationContext factory =
11              new FileSystemXmlApplicationContext("hello.xml");
12
13           Greeter greeter = (Greeter) factory
14                   .getBean("greeter");
15           greeter.sendGreetings();
16
17           ctx.close();
18       }
19   }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xmlns:context="http://www.springframework.org/schema/context"
5        xsi:schemaLocation="
6     http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
8     http://www.springframework.org/schema/context
9     http://www.springframework.org/schema/context/spring-context-3.1.xsd">
10
11       <context:component-scan base-package="hello"/>
12
13
14   </beans>
```

Chapter Summary

In this chapter, you learned

- About the basic architecture of the Spring Framework
- How to use Dependency Injection to inject resources into a bean

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Annotations

- Enabling Annotations
- Annotations for Dependency Injection

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Spring Beans

- Spring provides several techniques so that developers can provide metadata for beans, including:
  - XML
  - Annotations
  - Java Configuration
- XML configuration was the original approach, but this chapter focuses on annotations

Spring version 4 added Groovy configuration as well.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Annotations

- **Annotations** were added to Java in JDK5 and let you provide **metadata** in your source code
- You can use annotations to decorate classes, fields or methods
- Java defines several built-in annotations and frameworks like Spring can define their own "custom" annotations
- The Java compiler ignores custom annotations, so they need to be processed by the framework

```
1    @MyCustomAnnotation
2    public class Test
3    {
4      @SuppressWarnings("unused")
5      private int x;
6
7       @Override
8       public String toString() {...}
9    }
```


JDK 5 introduced a few standard, or built-in annotations:

@Override, @Deprecated, @SuppressWarnings,@Retention,@Target

The "parameters" to annotations are referred to as "elements". Annotations can accept no elements, a single element or multiple elements.

Technically, an annotation is similar to an interface, in that it defines a new type and must be fully qualified or imported when used.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Annotations and Spring

- Annotations were first added to Spring in version 2.0 and expanded in version 2.5
- Spring defines its own annotations and also uses annotations from other Java specifications (e.g. EJB3 and AOP)

```
1    import org.springframework.stereotype.Component;
2    import javax.annotation.Resource;
3
4    @Component("theStudent")
5    public class Student
6    {
7      @Resource(name="theAdvisor")
8      private Advisor theAdvisor;
9      public void setTheAdvisor(Advisor theAdvisor)
10       {...}
11   }
```

This code snippet shows using the @Resource annotation, which Spring borrows from JEE, and the @Component annotation, which is Spring-specific.

We will cover what these particular annotations do in a moment.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Enabling Annotations

- In Spring, annotations are processed by BeanPostProcessors, so you must configure the post processor(s) in the Spring configuration
- Spring lets you enable ALL annotations or selectively enable only the ones you are using

```
<!-- option 1: enable individual BeanPostProcessors -->
<bean class=
  "org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>
<bean class=
  "org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"

<!-- option 2: enable ALL BeanPostProcessors -->
<context:annotation-config/>

<!-- option 3: enable ALL BeanPostProcessors AND @Component scanning -->
<context:component-scan base-package="mypackage" />
```

A BeanPostProcessor is a Spring component that runs as part of the container's startup sequence. The BeanPostProcessors mentioned here scan your beans looking for annotations and then taking appropriate action.

Configuring all of the annotation processors adds overhead if your Spring program is only using a few types of annotations, but is certainly is easier.

Programs that use @Component have no options here - they must use the third approach. But programs that primarily use XML for configuration and only use annotations besides @Component can decrease container startup time by selectively enabling on the BeanPostProcessors they need.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The @Value Annotation

- This is a Spring-specific annotation that lets you inject a simple value

```
1    public class Student
2    {
3    . . .
4
5      @Value("3.33")
6      public void setGpa(double gpa)
7      {. . .}
8    }
```

Spring also provides the Spring Expression Language (SpEL) that lets you perform sophisticated injection using @Value. For example, this expression injects a random number:

```
#{ T(java.lang.Math).random() * 100.0 }
```

The "T" in the expression means Type and lets you assign a Java type to a part of the expression, in this case the java.lang.Math class.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Annotations for Dependency Injection

- As we've already seen, a common use of annotations in Spring is to perform **dependency injection**
- You use @Component to configure beans into the container and then @Resource, @Autowire or @Inject to inject references

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The @Component Annotation

- This is a Spring-specific annotation that obviates the need to define beans in the XML configuration file
- You must write the **context:component-scan** element in the configuration file

```
1    package university.faculty;
2    import org.springframework.stereotype.Component;
3
4    @Component("advisor1")
5    public class Advisor {. . . .}
```

```
1    <beans ...>
2
3       <!-- <context:annotation-config/> -->
4       <context:component-scan base-package="university"/>
5
6    </beans>
```

The context:component-scan XML element is a superset of context:annotation-config that additionally scans for @Component, @Resource and other annotations. If you have multiple "base packages", you can separate them with commas. Note that you can also fine-tune which packages it scans using context:include-filter and context:exclude-filter child elements.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The @Resource Annotation: By Name

- This is a standard JEE annotation that Spring uses to perform dependency injection
- @Resource injects either by **name** or by **type**, depending on how you use it

```
1    package university;
2
3    import javax.annotation.Resource;
4    import org.springframework.stereotype.Component;
5
6    @Component(name="student1")
7    public class Student
8    {
9      // inject bean reference by name
10     @Resource(name="advisor1")
11     public void setTheAdvisor(Advisor theAdvisor)
12     {. . .}
13   }
```

Here are assuming that there is an Advisor bean defined named "advisor1". That bean could either be defined in the XML, or using the @Component annotation (covered on a later page).

The placement for this annotation is either on the field to be injected or on its "set" method.

The @Resource annotation is from JSR-250 and is also used by JEE application servers that implement JEE Version 5 or later.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The @Resource Annotation: By Type

- You can use @Resource to inject by **type**
- You can use it to inject bean references or references to standard Spring types such as ApplicationContext

```
1    package university;
2
3    import javax.annotation.Resource;
4    import org.springframework.stereotype.Component;
5
6    @Component(name="student1")
7    public class Student
8    {
9      // inject reference by type
10     @Resource
11     private ApplicationContext ctx;
12     . . .
13   }
```

This usage of @Resource accepts no elements (parameters). The container determines what to look up by examining the type of the property.

The "standard" Spring objects include the BeanFactory interface, the ApplicationContext interface, the ResourceLoader interface, the ApplicationEventPublisher interface and the MessageSource interface.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Autowiring with Annotations

- @Autowired is a Spring-specific annotation that injects by **type**

```
1    public class MyBean
2    {
3      @Autowired
4      private ServiceBean1 sob1;
5
6      @Autowired
7      public void doSomething(ServiceBean sob2)
8      {
9          . . .
10     }
11
12     @Autowired
13     public MyBean(ServiceBean sob3) { . . . }
14   }
```

You can use the @Autowired annotation on fields, "set" methods, constructors, and other methods that accept bean references as parameters. If the wiring fails, the container will throw NoSuchBeanDefinitionException.

Note that @Autowired wires by type, at least by default. You can use the @Qualifier annotation in conjunction with @Autowired to have finer control over how it works.

@Autowired will fail if there's no candidate bean to wire, but you can specify @Autowired(required=false) to make it optional.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

@Resource vs @Autowired

- There is some overlap between the functionality of @Resource and @Autowired - both can inject by type
- A good rule of thumb is to use @Resource only for injecting by name and to use @Autowired to inject by type

Here's a note from the Spring documentation:

Tip: If you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if is technically capable of referring to a bean name through @Qualifier values. Instead, use the JSR-250 @Resource annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process.

You can read the docs at:

http://docs.spring.io/spring/docs/3.0.0.M4/spring-framework-reference/html/ch03s09.html

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The @Qualifier Annotation

- You can use **@Qualifier** with autowiring when there are multiple candidate beans

```
1    @Component("firstImpl")
2    public class FirstImpl implements MyService
3    {
4    . . .
5    }
6    @Component("secondImpl")
7    public class SecondImpl implements MyService
8    {
9    . . .
10   }
11   public class Client
12   {
13     @Autowired @Qualifier("secondImpl")
14     private MyService theService;
15    . . .
16   }
```

This essentially is the same as autowiring by name.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Other Annotations

- Spring defines many more annotations, including:

Annotation	Notes
@Scope	Configures bean scope
@PostConstruct	For bean lifecycle
@PreDestroy	For bean lifecycle
@Controller	For Spring MVC @Component stereotype
@RequestMapping	For Spring MVC
@RequestParameter	For Spring MVC
@Service	@Component stereotype for REST
@Respository	@Compoment stereotype for DAOs
@Transactional	Configure transactions
@Pointcut	For Spring AOP
@Before	For Spring AOP
@RunWith	For testing

The "stereotype" annotations are special cases of @Compoment for components that reside in the tiers of a typical JEE application. @Controller is for the Web tier, @Service for the business-logic tier and @Repository is for the data tier. You use the stereotypes instead of @Component mainly for documentation, though @Repository does provide additional function (it translates database exceptions into Spring database exceptions).

Note that modern versions of JEE support the Common Dependency Injection (CDI) specification, which defines alternatives to many of these Spring-specific annotations.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring and JSR-330

- The **Dependency Injection for Java** (JSR-330) specification defines a standard set of annotations for DI in Java
- JSR-330 defines @Named and @Inject annotatations similar to Spring's @Component and @Autowired
- Spring applications that need to be JEE standards compliant can use the JSR-330 annotations if they include the **javax-inject.jar** in the CLASSPATH

JSR-330's design was heavily influenced by Spring and the Google Guice framework.

The Contexts and Dependency Injection (JSR-299) specification extends JSR-330 and is a standard part of JEE 6 (and later) designed to help you integrate tiers, e.g. Web and domain logic (EJB)

CDI provides additional annoations such as @RequestScoped and @ConversationScoped that are useful in Web applications.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- How to inject collection properties
- About autowiring
- How to use annotations in Spring 2.x or later as an alternative way to provide configuration metadata

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Java Configuration

- Why Use Java Configuration?
- Writing a Configuration Class
- Using Dependency Injection

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Spring Beans

- Spring provides several techniques so that developers can provide metadata for beans, including:
  - XML
  - Annotations
  - Java Configuration
- XML configuration is the traditional approach

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Java-Based Configuration

- String in Spring 3, you can provide configuration metadata by writing Java code instead of XML, giving you fine-grained control over how the container creates objects
- Spring provides the **@Configuration** and **@Bean** annotations as well as the **AnnotationConfigApplicationContext** application context type

This approach lets you avoid writing ANY XML at all.

Other Java-based configuration annotations include @Import (to import configuration from another class) and @Scope to specify the scope.

Note that in versions of Spring prior to 3.2, Java-based configuration requires the CGLIB JAR to be in the program's CLASSPATH.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java-Based Configuration Example

```
1    @Configuration
2    public class HelloWorldConfig
3    {
4       @Bean
5       public HelloWorld helloWorld()
6       {
7          return new HelloWorld();
8       }
9    }
```

```
1    AnnotationConfigApplicationContext ctx =
2      new AnnotationConfigApplicationContext(
3         HelloWorldConfig.class);
4
5    HelloWorld helloWorld =
6       ctx.getBean(HelloWorld.class);
7
8    . . .
```

This example shows configuration and using a HelloBean object (HelloBean class is not shown).

Note how the main program uses the AnnotationConfigApplicationContext and retrieves the bean by specifying its class.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Why Use Java Configuration?

- Java configuration has some of the same advantages as XML configuration:
  - Centralized configuration
  - Cleaner, pure-POJO beans (no annotations)
- Main advantage of Java configuration over XML is compiler-time checking for typos and type-safety

One big issue with XML configuration is that you don't discover problems until runtime.

A potential downside to Java configuration is that Java itself is somewhat verbose. A potential solution is to use the Groovy language for configuration instead of Java. Groovy is a simplified syntax language based on Java. For more info on using Groovy for Spring configuration, see:

```
https://spring.io/blog/2014/03/03/groovy-bean-configuration-in-spring-framework-4
http://hantsy.blogspot.com/2013/12/spring-4-groovy-dsl-bean-definition.html
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The @Bean Annotation

- You use the **@Bean** annotation on methods that produce Spring beans
- The container calls such methods, then adds the returned reference to its internal bean list
- By default, the new bean's name is taken from the bean-creation method's name
- This annotation has optional elements to let you specify the bean's name, scope and initialize and destroy methods

```
1    @Bean
2    public Student myStudent()
3    {
4      Student s = new Student();
5
6      return s;
7    }
```

For the example here to work, the Student class would need to have a public, zero-argument constructor.

We will cover scope and init and destroy methods later in the class.

Note that is possible to assign beans names and then retrieve them by name:

```
@Bean(name="student01")
public Student createStudent() {....}

. . .

Student s = (Student)ctx.getBean("student01");
```

If you don't assign a name, Spring uses the bean-creation method's name.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Performing Setter Injection

- In the bean-creation method, you can easily call **set()** methods to inject dependencies

```
1    @Bean
2    public Advisor myAdvisor()
3    {
4       Advisor a = new Advisor();
5       a.setAdvisorID(12);
6       a.setName("Sue Smith");
7       a.setSalary(40000);
8
9       return a;
10   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Referencing Other Beans

- If the bean-creation methods are in the same configuration class, you can call them directly to obtain references to other beans

```
1    @Configuration
2    public class MyConfig
3    {
4       @Bean
5       public Student myStudent()
6       {
7           Student s = new Student();
8           s.setAdvisor(myAdvisor());
9           return s;
10      }
11
12      @Bean
13      public Advisor myAdvisor() {....}
14   }
```

We will see how to use multiple configuration classes later in this chapter.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Performing Constructor Injection

- In the bean-creation method, you can provide dependencies via **constructor injection** if the bean class supports it

```
1    @Bean
2    public Student myStudent()
3    {
4       Student s = new Student(myAdvisor());
5       s.setName("Harry Wolfe");
6       s.setGpa(3.33);
7       s.setStudentID(45);
8
9       return s;
10   }
```

For the above code to work, the Student class would need a one-argument constructor that accepts an Advisor reference.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Autowiring in a Configuration Class

- Instead of directly calling other bean-creation methods, configuration classes can autowire related beans by type or name

```
1    @Configuration
2    public class StudentConfiguration
3    {
4      @Autowired private Advisor a;  // autowire by type
5
6      @Bean
7      public Student myStudent()
8      {
9        Student s = new Student(a);
10
11       return s;
12     }
13   }
```

This is especially useful if you write multiple configuration classes and the related bean is defined in a different configuration class.

Note that you could instead autowire by name:

```
@Resource(name="myAdvisor")
private Advisor a;
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Autowiring Bean-Creation Methods

- The bean-creation methods themselves can take advantage of dependency injection by defining arguments that Spring will inject

```
1    @Configuration
2    public class StudentConfig
3    {
4       @Bean
5       public Student myStudent(Advisor a)
6       {
7           Student s = new Student(a);
8           return s;
9       }
10   }
```


In this case, Spring will find a reference to an Advisor and pass it into the myStudent() method. Using this technique, you don't need to define any fields for autowiring.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Multiple @Configuration Classes

- The **@Configuration** annotation identifies a class as a providing beans
- You can write multiple Configuration classes to modularize the bean definitions

```
1    @Configuration
2    public class StudentConfig
3    {
4       @Bean public Student myStudent() {.....}
5    }
```

```
1    @Configuration
2    public class AdvisorConfig
3    {
4       @Bean public Advisor myAdvisor() {.....}
5    }
```

Be sure to never put any application code in Configuration beans!

@Configuration is actually an @Component stereotype.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Referencing Multiple Configuration Classes

- Applications can use **AnnotationConfigApplicationContext** or **AnnotationConfigWebApplicationContext** to reference configuration class(es)

```
1    AnnotationConfigApplicationContext ctx =
2     new AnnotationConfigApplicationContext(
3        StudentConfig.class,
4        AdvisorConfig.class);
5
6    Student s = (Student)ctx.getBean(Student.class);
7    Advisor a = (Advisor)ctx.getBean(Advisor.class);
```


Note that instead of providing the configuration classes when you create the context, you can call its "register()" method to provide the classes. You should then call refresh() on the context to ensure it's in a stable state.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Importing a Configuration

- You can use the **@Import** annotation to reference other Configurations
- Then, the "main" program only needs to specify the "root" Configuration

```
1    @Configuration
2    @Import(AdvisorConfig.class)
3    public class StudentConfig
4    {
5    . . .
6    }
```

```
1    AnnotationConfigApplicationContext ctx =
2      new AnnotationConfigApplicationContext(
3        StudentConfig.class);
4
5    Student s = (Student)ctx.getBean(Student.class);
6    Advisor a = (Advisor)ctx.getBean(Advisor.class);
```

In the bottom listing, we can use Advisors even though we didn't specify the AdvisorConfig class when we created the context.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Importing Other Configuration Types

- You can use the **@ImportResource** annotation to reference configurations defined in XML or Groovy
- You can use the **@ComponentScan** annotation to enable the same functionality as in XML's **context:component-scan** element
- These techniques lets you mix configuration approaches and migrate from XML and/or annotation-based configuration to Java configuration

```
1    @Configuration
2    @ImportResource({"classpath:spring.xml"})
3    @ComponentScan(basePackages="mypackage")
4    public class MyConfig
5    {
6    . . .
7    }
```

@ImportResource examines the file extension - if it ends in .groovy, then Spring uses a GroovyBeanDefinitionReader. If it's XML, then Spring uses XmlBeanDefinitionReader.

The string(s) you provide to this annotation can contain prefixes such as "file:" to read from the file system, "classpath:" to read from the CLASSPATH and so forth. See the Spring docs for Resources for details:

http://docs.spring.io/autorepo/docs/spring/3.2.x/spring-framework-reference/html/resources.html

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Complete Example

Advisor.java

Professor.java

Student.java

AdvisorConfig.java

StudentConfig.java

TestUniversity.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package university;
2
3    import java.util.ArrayList;
4    import java.util.Collection;
5
6    public class Advisor
7    {
8        private int advisorID;
9        private String name;
10       private double salary;
11       private Professor theProf;
12
13       private Collection<Student> students = new ArrayList<Student>();
14
15       public Professor getTheProf()
16       {
17           return theProf;
18       }
19
20       public void setTheProf(Professor theProf)
21       {
22           this.theProf = theProf;
23       }
24
25       public int getAdvisorID()
26       {
27           return advisorID;
28       }
29
30       public void setAdvisorID(int advisorID)
31       {
32           this.advisorID = advisorID;
33       }
34
35       public String getName()
36       {
37           return name;
38       }
39
40       public void setName(String name)
41       {
42           this.name = name;
43       }
44
45       public double getSalary()
```

```
46      {
47          return salary;
48      }
49
50      public void setSalary(double salary)
51      {
52          this.salary = salary;
53      }
54
55      public Collection<Student> getStudents()
56      {
57          return students;
58      }
59
60      public void setStudents(Collection<Student> students)
61      {
62          this.students = students;
63      }
64
65  }
```

```
1    package university;
2
3    public class Student
4    {
5        private int studentID;
6        private String name;
7        private double gpa;
8
9        private Advisor advisor;
10
11       public Student() {}
12
13       public Student(Advisor advisor)
14       {
15           this.advisor = advisor;
16       }
17
18       public int getStudentID()
19       {
20           return studentID;
21       }
22
23       public void setStudentID(int studentID)
24       {
25           this.studentID = studentID;
26       }
27
28       public String getName()
29       {
30           return name;
31       }
32
33       public void setName(String name)
34       {
35           this.name = name;
36       }
37
38       public double getGpa()
39       {
40           return gpa;
41       }
42
43       public void setGpa(double gpa)
44       {
45           this.gpa = gpa;
```

```
46        }
47
48        public Advisor getAdvisor()
49        {
50            return advisor;
51        }
52
53        public void init()
54        {
55            System.out.println("In init method");
56        }
57
58    }
```

```
1    package university.config;
2
3    import org.springframework.context.annotation.Bean;
4    import org.springframework.context.annotation.Configuration;
5
6    import university.Advisor;
7    import university.Professor;
8
9    @Configuration
10   public class AdvisorConfig
11   {
12       @Bean
13       public Advisor myAdvisor()
14       {
15           Advisor a = new Advisor();
16           a.setAdvisorID(12);
17           a.setName("Sue Smith");
18           a.setSalary(40000);
19
20           a.setTheProf(theProfessor());
21
22           return a;
23       }
24
25       @Bean
26       public Professor theProfessor()
27       {
28           Professor p = new Professor();
29           p.setName("I.M. Mean");
30           p.setProfessorID(42);
31
32           return p;
33       }
34   }
```

```
1    package university.config;
2
3    import javax.annotation.Resource;
4
5    import org.springframework.context.annotation.Bean;
6    import org.springframework.context.annotation.Configuration;
7    import org.springframework.context.annotation.Import;
8
9    import university.Advisor;
10   import university.Student;
11
12   @Configuration
13   @Import(AdvisorConfig.class)
14   public class StudentConfig
15   {
16     @Autowired
17     private Advisor a;
18
19   //    @Resource(name="myAdvisor")
20   //    private Advisor a;
21
22       @Bean(name="student01")
23   //  public Student myStudent(Advisor a)
24       public Student myStudent()
25       {
26           Student s = new Student(a);
27           s.setName("Harry Wolfe");
28           s.setGpa(3.33);
29           s.setStudentID(45);
30
31           return s;
32       }
33   }
```

```
1    package university.test;
2
3    import org.springframework.context.annotation.AnnotationConfigApplicationContext;
4    import org.springframework.context.support.AbstractApplicationContext;
5
6    import university.Student;
7    import university.config.AdvisorConfig;
8    import university.config.StudentConfig;
9
10   public class TestUniversity
11   {
12
13       public static void main(String[] args)
14       {
15           AbstractApplicationContext ctx = new AnnotationConfigApplicationContext(
16                   StudentConfig.class, AdvisorConfig.class);
17
18           // Student s = (Student)ctx.getBean(Student.class);
19           Student s = (Student) ctx.getBean("student01");
20           System.out.println("Student name: " + s.getName());
21           System.out.println("Advisor name: " + s.getAdvisor().getName());
22           System.out.println("Professor name: " +
23              s.getAdvisor().getTheProf().getName());
24
25           ctx.close();
26       }
27
28   }
```

```
1    package university;
2
3    import java.util.ArrayList;
4
5    public class Professor
6    {
7        private int professorID;
8        private String name;
9        private ArrayList<Advisor> advisors;
10
11       public int getProfessorID()
12       {
13           return professorID;
14       }
15
16       public void setProfessorID(int professorID)
17       {
18           this.professorID = professorID;
19       }
20
21       public String getName()
22       {
23           return name;
24       }
25
26       public void setName(String name)
27       {
28           this.name = name;
29       }
30
31       public ArrayList<Advisor> getAdvisors()
32       {
33           return advisors;
34       }
35
36       public void setAdvisors(ArrayList<Advisor> advisors)
37       {
38           this.advisors = advisors;
39       }
40   }
```

Using the Groovy DSL for Configuration

- Starting in Spring 4, you can use the Groovy language to define Spring beans and/or to provide configuration metadata
- Steps for using Groovy:
  - Add Groovy dependency to Gradle or Maven build script
  - Optionally use Groovy to create Spring bean classes
  - Optionally use Spring's Groovy DSL to configure beans written in Java or Groovy

Spring 4 adapted the Domain Specific Language (DSL) from the Grails project so you can configure beans using Groovy. Using the DSL is nice because its syntax is simpler than Java, but still gives the expressive power of standard Java configuration.

We will not cover how to create Spring bean classes in Groovy in this chapter, since this chapter is all about configuration.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Groovy?

- Groovy is a dynamic, typing optional language with syntax similar to Java that runs on the JVM
- Some differences from Java:
  - Semicolons optional
  - Delimit strings with single or double quotes (double quoted support **interpolation**)
  - Can omit parenthesis for method calls
  - **print** and **println** shortcuts for System.out.println



The home page for Groovy is at: http://www.groovy-lang.org/index.html

Like Java 8, Groovy supports functional programming and closures.

Interpolated strings can contain variable references, for example:

```
def name = "John Adams"
println "Hello, $name"
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Groovy DSL Example

```
1    // build.gradle
2    dependencies {
3     compile 'org.springframework:spring-context:4.2.6.RELEASE'
4     compile "org.codehaus.groovy:groovy-all:2.1.0"
5    }
```

```
1    // spring.groovy
2    import university.TeachingAssistant
3
4    beans {
5      ta23 (TeachingAssistant) {  // bean-name (bean-type)
6        taID = 465                // set property
7        name = "Sue Mi"           // set property
8      }
9    }
```

```
1    // StudentConfig.java
2    @ImportResource({"classpath:spring.groovy"})
3    public class StudentConfig {. . .}
```

"beans" is part of the DSL. Behind the scenes, Spring creates a GroovyBeanDefinitionReader that parses the DSL and configures Spring beans into the container.

For more information on the Groovy DSL, see:

https://spring.io/blog/2014/03/03/groovy-bean-configuration-in-spring-framework-4

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the advantanges of Spring Java Configuration
- How to use Java Configuration to perform dependency injection

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Beans

- Scopes
- Lifecycle

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Spring Beans

- Spring provides several techniques so that developers can provide metadata for beans, including:
  - XML
  - Annotations
  - Spring 3 Java Configuration
- XML configuration is the traditional approach

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Bean Scopes

- In Spring, the notion of scope defines how many objects a Spring container instantiates for a given configured bean
- Basic Spring supports **singleton** and **prototype**
- ApplicationContexts used in Web applications also support **request**, **session** and **global session** scopes

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Singleton Scope

- By default, Spring treats beans as **singletons**, which means that a container instantiates only one instance of a defined bean

```
1    @Component("student5")
2    public class Student
3    {
4       @Resource(name="advisor1")
5       private Advisor theAdvisor;
6    }
7    @Component("advisor1")
8    public class Advisor{...}
```

```
1    Advisor advisor1 = (Advisor)ctx.getBean("advisor1");
2    Student student5 = (Student)ctx.getBean("student5");
3    // prints "Singleton!"
4    if (advisor1 == student5.getTheAdvisor())
5      System.out.println("Singleton!");
6    else
7      System.out.println("Prototype!");
```

Spring potentially instantiates whenever you "get" the bean programmatically or inject the bean into another. In either case, if it's a singleton, the container only instantiates one bean.

Note that Spring's idea of a singleton is a bit different than the Singleton Design Pattern from the "Gang of Four" book. A GoF Singleton only allows a single instance per classloader and uses a private constructor to enforce that. Spring singletons are unique in a given container and can have normal constructors.

The equivalent XML configuration is:

```
<bean id="advisor1" scope="singleton"
   class="university.Advisor"/>

<bean id="student5" class="university.Student">
  <property name="theAdvisor" ref="advisor1" />
</bean>
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Prototype Scope

- If you configure a bean as a **prototype**, then a container instantiates a new object whenever a request, e.g. injection or getBean(), is made

```
1    @Component("student5")
2    public class Student
3    {
4       @Resource(name="advisor1")
5       private Advisor theAdvisor;
6    }
7    @Component("advisor1") @Scope("prototype")
8    public class Advisor{...}
```

```
1    Advisor advisor1 = (Advisor)ctx.getBean("advisor1");
2    Student student5 = (Student)ctx.getBean("student5");
3    // prints "Prototype!"
4    if (advisor1 == student5.getTheAdvisor())
5      System.out.println("Singleton!");
6    else
7      System.out.println("Prototype!");
```

The @Scope annotation is from the org.springframework.context.annotation package.

The equivalent XML configuration is:

```
<bean id="advisor1" scope="prototype"
   class="university.Advisor"/>

<bean id="student5" class="university.Student">
  <property name="theAdvisor" ref="advisor1" />
</bean>
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Singleton vs Prototype Java Config

- By default, beans created during Java configuration are singletons, but you can specify prototype scope on the bean-creation method using **@Scope**

```
1    @Bean
2    @Scope("prototype")
3    public Advisor createAdvisor()
4    {
5      Advisor a = new Advisor();
6      a.setAdvisorID(12);
7      a.setName("Sue Smith");
8      a.setSalary(40000);
9
10     return a;
11   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Bean Lifecycle

- Similar to EJBs, Spring beans have a well-defined creation and destruction lifecycle
- If you wish, you can hook into the lifecycle so that the container calls methods as the bean moves through its lifecycle
- Spring provides serveral ways to configure notification of lifecycle events:
    - Annotation configuration
    - Java configuration
    - Implementing optional Spring interfaces
    - XML configuration

Implementing interfaces ties you to Spring, which goes against the general philosphy of writing plain POJOs. So it's preferable to use the configuration or annotation approach if possible.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Lifecycle Annotations

- **@PostConstruct** and **PreDestroy** are standard JEE annotations that Spring uses to let you hook lifecycle events
- These annotations are processed by the CommonAnnotationBeanPostProcessor

```
1    @Component
2    public class Student
3    {
4      @PostConstruct
5      public void setup() {....}
6
7      @PreDestroy
8      public void cleanup() {....}
9    }
```

This is an alternative to implementing Spring-specific lifecycle interfaces or configuring "init-method" and "destroy-method" in the Spring configuration file.

These annotation are from JSR-250 and are also used by JEE application servers that implement JEE Version 5 or later.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

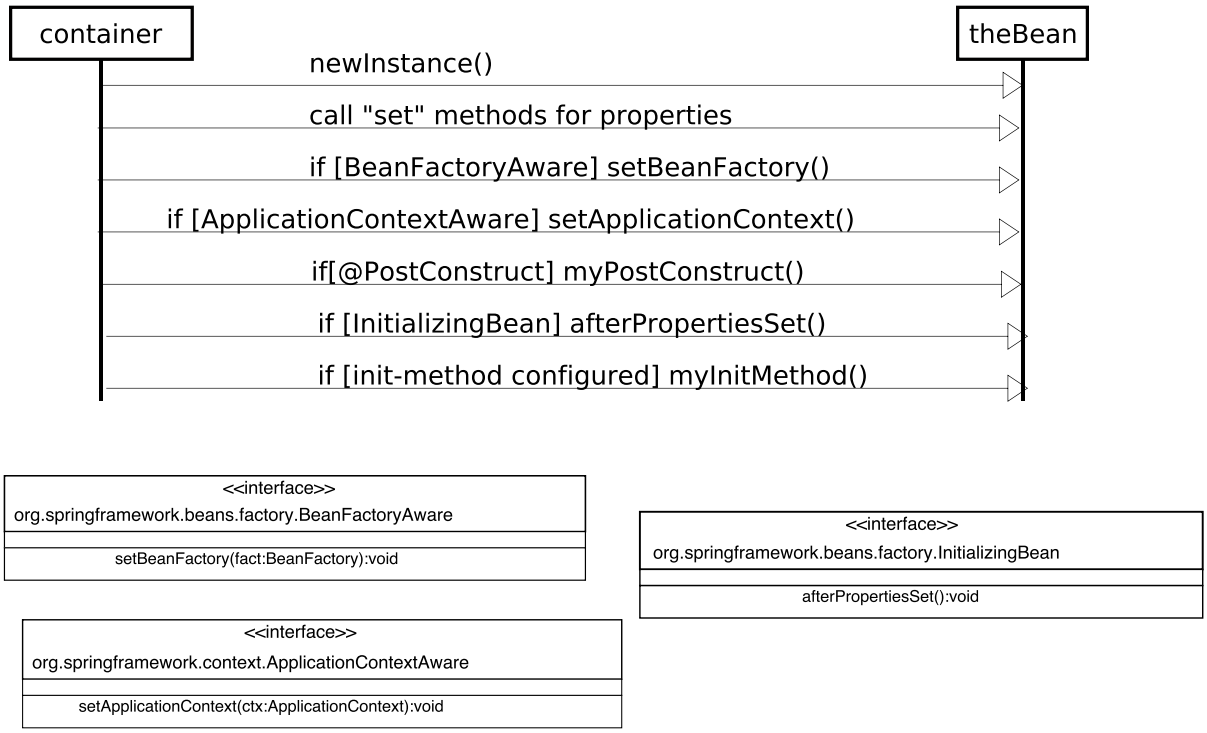
Lifecycle With Java Config

- Optional elements on @Bean let you specify **init** and **destroy** methods

```
1    @Bean(initMethod="init", destroyMethod="cleanup")
2    public Student createStudent()
3    {
4      Student s = new Student();
5
6      return s;
7    }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Creation Lifecycle



BeanFactoryAware, ApplicationContextAware and InitializingBean are all interfaces that the bean can optionally implement so the container can notify the bean of lifecycle events. Note that the setBeanFactory and setApplicationContext() methods pass the specified object to the bean - that lets the bean "know" about its container. Also note that setApplicationContext() only works if the bean resides in an application context rather than a bean factory.

In contrast, to use an "init-method", in the XML, you configure any method you wish in the bean. The container then calls that method after the bean is fully initialized. The syntax is like:

```
<bean ... init-method="myInitMethod" ... >
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Creation Lifeycle Example

MyCreationBean.java

TestCreation.java

create.xml

Instead of, or in addition to using "init-method", you can define a "global" definition in the XML file for all beans:

```
<beans default-init-method="setup" ... >
```

Note that this only works if the "init" method is named "setup" in all of the beans.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package mypackage;
2
3    import org.springframework.beans.BeansException;
4    import org.springframework.beans.factory.BeanFactory;
5    import org.springframework.beans.factory.BeanFactoryAware;
6    import org.springframework.beans.factory.InitializingBean;
7    import org.springframework.context.ApplicationContext;
8    import org.springframework.context.ApplicationContextAware;
9
10   public class MyCreationBean implements BeanFactoryAware,
11           ApplicationContextAware, InitializingBean
12   {
13       private int prop1;
14       private String prop2;
15
16       public int getProp1()
17       {
18           return prop1;
19       }
20
21       public void setProp1(int prop1)
22       {
23           this.prop1 = prop1;
24       }
25
26       public String getProp2()
27       {
28           return prop2;
29       }
30
31       public void setProp2(String prop2)
32       {
33           this.prop2 = prop2;
34       }
35
36       @Override
37       public void setBeanFactory(BeanFactory factory) throws BeansException
38       {
39           System.out.println("In setBeanFactory");
40       }
41
42       @Override
43       public void setApplicationContext(ApplicationContext ctx)
44               throws BeansException
45       {
```

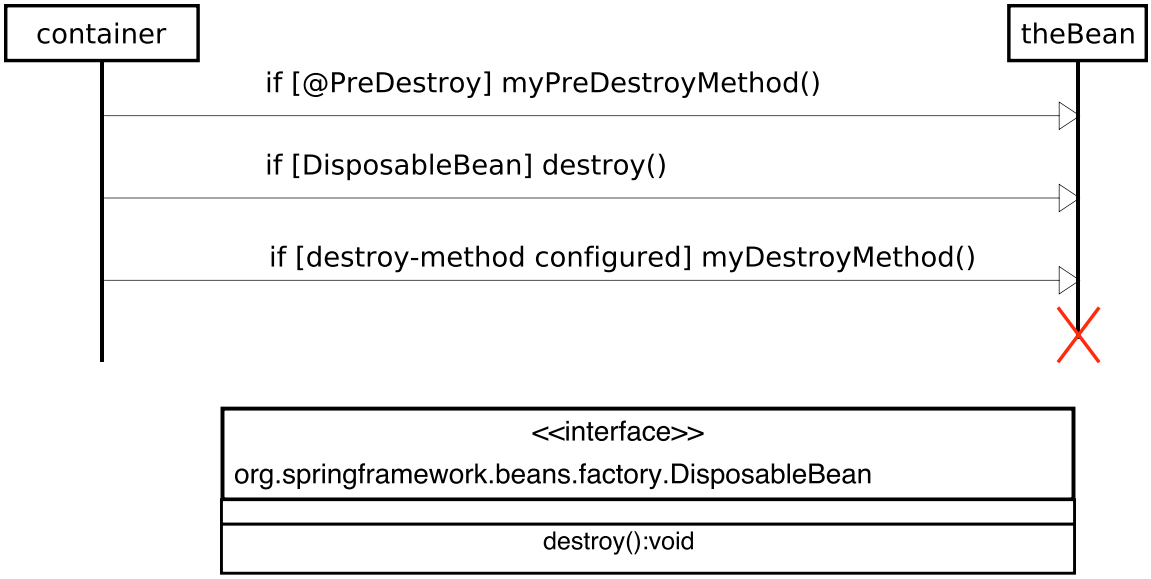
```
46          System.out.println("In setApplicationContext");
47      }
48
49      @Override
50      public void afterPropertiesSet() throws Exception
51      {
52          System.out.println("In afterPropertiesSet, prop1=" + prop1);
53      }
54
55      public void myInitMethod()
56      {
57          System.out.println("In myInitMethod");
58      }
59
60      @PostConstruct
61      public void myPostConstructMethod()
62      {
63          System.out.println("In myPostConstructMethod");
64      }
65  }
```

```
1    package main;
2
3    import mypackage.MyCreationBean;
4
5    import org.springframework.context.ApplicationContext;
6    import org.springframework.context.support.FileSystemXmlApplicationContext;
7
8    public class TestCreation
9    {
10       public static void main(String[] args)
11       {
12           ApplicationContext ctx =
13               new FileSystemXmlApplicationContext("create.xml");
14
15           MyCreationBean mcb = (MyCreationBean)ctx.getBean("myCreateBean");
16       }
17
18   }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xmlns:aop="http://www.springframework.org/schema/aop"
5        xmlns:context="http://www.springframework.org/schema/context"
6        xmlns:jee="http://www.springframework.org/schema/jee"
7        xmlns:jms="http://www.springframework.org/schema/jms"
8        xmlns:lang="http://www.springframework.org/schema/lang"
9        xmlns:tx="http://www.springframework.org/schema/tx"
10       xmlns:util="http://www.springframework.org/schema/util"
11       xsi:schemaLocation="http://www.springframework.org/schema/aop
12           http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
13           http://www.springframework.org/schema/beans
14           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
15           http://www.springframework.org/schema/context
16           http://www.springframework.org/schema/context/spring-context-3.1.xsd
17           http://www.springframework.org/schema/jee
18           http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
19           http://www.springframework.org/schema/lang
20           http://www.springframework.org/schema/lang/spring-lang-3.1.xsd
21           http://www.springframework.org/schema/tx
22           http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
23           http://www.springframework.org/schema/util
24           http://www.springframework.org/schema/util/spring-util-3.1.xsd">
25
26       <context:annotation-config/>
27       <context:component-scan base-package="mypackage"/>
28
29       <bean id="myCreateBean" class="mypackage.MyCreationBean"
30          init-method="myInitMethod">
31        <property name="prop1" value="12"/>
32        <property name="prop2" value="abcd"/>
33       </bean>
34
35   </beans>
```

Destruction Lifecycle

- This only applies to singleton-scoped beans since the container "forgets" about prototypes after creation



For this to work, you need to explicitly close the bean factory or context. Another possibility is to install a "JVM shutdown" hook in the factory or context. See the Spring docs for details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Destruction Lifecycle Example

MyDestroyBean.java

TestDestroy.java

destroy.xml

Instead of, or in addition to using "destroy-method", you can define a "global" definition in the XML file for all beans:

```
<beans default-destroy-method="cleanup" ... >
```

Note that this only works if the "destroy" method is named "cleanup" in all of the beans.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

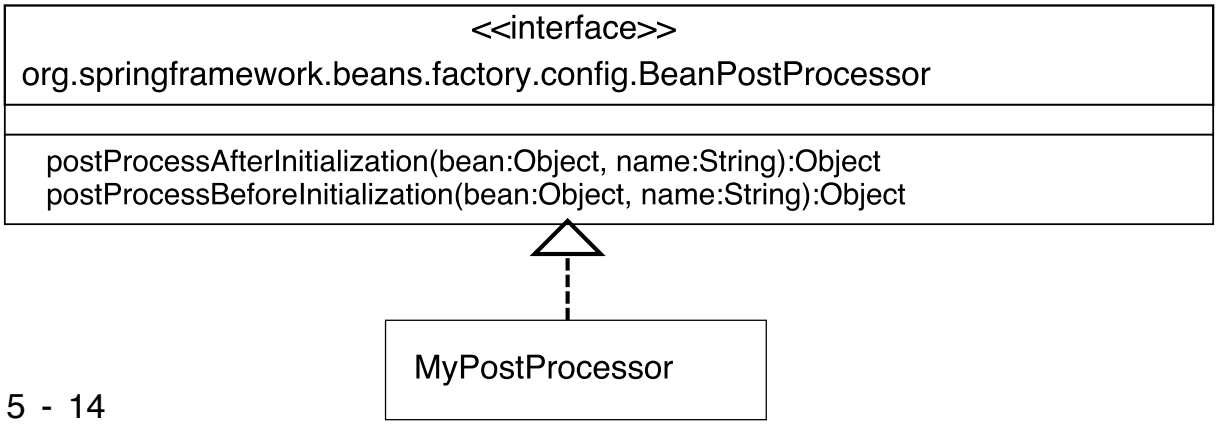
```
1    package mypackage;
2
3    import org.springframework.beans.factory.DisposableBean;
4
5    public class MyDestroyBean implements DisposableBean
6    {
7
8        @Override
9        public void destroy() throws Exception
10       {
11           System.out.println("In destroy");
12       }
13
14       public void myDestroyMethod()
15       {
16           System.out.println("In myDestroyMethod");
17       }
18
19       @PreDestroy
20       public void myPreDestroyMethod()
21       {
22           System.out.println("In myPreDestroyMethod");
23       }
24
25   }
```

```
1    package main;
2
3    import mypackage.MyDestroyBean;
4
5    import org.springframework.context.support.AbstractApplicationContext;
6    import org.springframework.context.support.FileSystemXmlApplicationContext;
7
8    public class TestDestroy
9    {
10       public static void main(String[] args)
11       {
12           AbstractApplicationContext ctx =
13               new FileSystemXmlApplicationContext("destroy.xml");
14
15           MyDestroyBean mdb = (MyDestroyBean)ctx.getBean("myDestroyBean");
16           ctx.close();
17       }
18
19   }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xmlns:aop="http://www.springframework.org/schema/aop"
5        xmlns:context="http://www.springframework.org/schema/context"
6        xmlns:jee="http://www.springframework.org/schema/jee"
7        xmlns:jms="http://www.springframework.org/schema/jms"
8        xmlns:lang="http://www.springframework.org/schema/lang"
9        xmlns:tx="http://www.springframework.org/schema/tx"
10       xmlns:util="http://www.springframework.org/schema/util"
11       xsi:schemaLocation="http://www.springframework.org/schema/aop
12           http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
13           http://www.springframework.org/schema/beans
14           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
15           http://www.springframework.org/schema/context
16           http://www.springframework.org/schema/context/spring-context-3.1.xsd
17           http://www.springframework.org/schema/jee
18           http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
19           http://www.springframework.org/schema/lang
20           http://www.springframework.org/schema/lang/spring-lang-3.1.xsd
21           http://www.springframework.org/schema/tx
22           http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
23           http://www.springframework.org/schema/util
24           http://www.springframework.org/schema/util/spring-util-3.1.xsd">
25
26       <context:annotation-config/>
27       <context:component-scan base-package="mypackage"/>
28
29       <bean id="myDestroyBean" class="mypackage.MyDestroyBean"
30          destroy-method="myDestroyMethod">
31       </bean>
32
33   </beans>
```

BeanPostProcessors

- A **BeanPostProcessor** can hook certain lifecycle events on **all** of the beans in a container
- You must register a BeanPostProcessor - in an application context you can register via annotations or XML, but in a bean factory, you must register programmatically



Spring itself uses bean post processors for things like annotation processing and aspect-oriented programming (AOP).

Bean post processors generally do one of two things:

1. Modify the input bean in some way

2. Substitute a different bean for the input bean

As example of what Spring itself does, for #1, Spring uses bean-post processors to read, process and apply annotations to beans. For #2, Spring AOP substitutes a proxy for the input bean so Spring can intercept method invocations on the bean.

As an application example of #1, imagine you have a bean with a text property that's initialized with encrypted text. A bean post processor could have code in postProcessAfterInitialization() to decrpyt the text.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Coding A BeanPostProcessor

- Since a BeanPostProcessor "sees" ALL beans in the container, if it only wants to operate on certain beans, it must be able to distinguish them
- Distinguishing strategies include using a **marker** interface or by using bean naming conventions
- The post-processor methods can return either the input object or substitute another object with the same interface

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

BeanPostProcessor Example

Encrypted.java

EncryptedBean.java

MyBeanPostProcessor.java

TestBeanPostProcessor.java

beanpp.xml

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package mypackage;
2
3    public interface Encrypted
4    {
5        public String getText();
6        public void setText(String text);
7    }
```

```
1    package mypackage;
2
3    import org.springframework.stereotype.Component;
4
5    @Component
6    public class EncryptedBean implements Encrypted
7    {
8        private String text = "Uryyb, jbeyq";
9
10       public String getText()
11       {
12           return text;
13       }
14
15       public void setText(String text)
16       {
17           this.text = text;
18       }
19   }
```

```
1    package mypackage;
2
3    import org.springframework.beans.BeansException;
4    import org.springframework.beans.factory.config.BeanPostProcessor;
5    import org.springframework.stereotype.Component;
6
7    @Component
8    public class MyBeanPostProcessor implements BeanPostProcessor
9    {
10
11       @Override
12       public Object postProcessAfterInitialization(Object bean, String name)
13               throws BeansException
14       {
15           System.out.println("In postProcessAfterInitialization");
16           if (bean instanceof Encrypted)
17           {
18               Encrypted eb = (Encrypted)bean;
19               String s1 = eb.getText();
20               String s2 = decrypt(s1);
21               eb.setText(s2);
22               System.out.println("Decrypted '" + s1 + "' to '" + s2 + "'");
23           }
24
25           return bean;
26       }
27
28       @Override
29       public Object postProcessBeforeInitialization(Object bean, String name)
30               throws BeansException
31       {
32           return bean;
33       }
34
35       private String decrypt(String s)
36       {
37           StringBuilder out = new StringBuilder(s.length());
38           char[] ca = s.toCharArray();
39           for (char c : ca)
40           {
41               if (c >= 'a' && c <= 'm')
42                   c += 13;
43               else if (c >= 'n' && c <= 'z')
44                   c -= 13;
45               else if (c >= 'A' && c <= 'M')
```

```
46                  c += 13;
47              else if (c >= 'N' && c <= 'Z')
48                  c -= 13;
49              out.append(c);
50          }
51          return out.toString();
52      }
53
54      public static void main(String[] args)
55      {
56          MyBeanPostProcessor mbpp = new MyBeanPostProcessor();
57          String s1 = mbpp.decrypt("Hello, world");
58          String s2 = mbpp.decrypt(s1);
59          System.out.println(s1);
60          System.out.println(s2);
61      }
62  }
```

```
1    package main;
2
3    import mypackage.EncryptedBean;
4
5    import org.springframework.context.support.AbstractApplicationContext;
6    import org.springframework.context.support.FileSystemXmlApplicationContext;
7
8    public class TestBeanPostProcessor
9    {
10
11       public static void main(String[] args)
12       {
13           AbstractApplicationContext ctx =
14               new FileSystemXmlApplicationContext("beanpp.xml");
15
16           EncryptedBean eb = (EncryptedBean)ctx.getBean("encryptedBean");
17           System.out.println("Main: " + eb.getText());
18       }
19
20   }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xmlns:aop="http://www.springframework.org/schema/aop"
5        xmlns:context="http://www.springframework.org/schema/context"
6        xmlns:jee="http://www.springframework.org/schema/jee"
7        xmlns:jms="http://www.springframework.org/schema/jms"
8        xmlns:lang="http://www.springframework.org/schema/lang"
9        xmlns:tx="http://www.springframework.org/schema/tx"
10       xmlns:util="http://www.springframework.org/schema/util"
11       xsi:schemaLocation="http://www.springframework.org/schema/aop
12           http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
13           http://www.springframework.org/schema/beans
14           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
15           http://www.springframework.org/schema/context
16           http://www.springframework.org/schema/context/spring-context-3.1.xsd
17           http://www.springframework.org/schema/jee
18           http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
19           http://www.springframework.org/schema/lang
20           http://www.springframework.org/schema/lang/spring-lang-3.1.xsd
21           http://www.springframework.org/schema/tx
22           http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
23           http://www.springframework.org/schema/util
24           http://www.springframework.org/schema/util/spring-util-3.1.xsd">
25
26       <context:annotation-config/>
27       <context:component-scan base-package="mypackage"/>
28
29   </beans>
```

Factory Methods

- Some Java classes define a **static** method that creates and initializes instances
- If you wish to use such a class as a Spring bean configured in XML, then you must tell the container to use the static method

```
1    <bean id="bean1" factory-method="getInstance"
2      class="mypackage.MyClassWithFactoryMethod" />
```

```
1    public static MyClassWithFactoryMethod getInstance()
2    {
3     . . .
4    }
```

If needed, you can pass arguments to the factory method:

```
<bean ... factory-method="getInstance" ...>
  <constructor-arg value="myval" />
</bean>
```

If you don't want to use XML, then you can investigate the new "Java-based container configuration" in Spring 3 and later.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Factory Classes

- Some Java classes use a separate **factory class** to create instances
- The factory itself defines **non-static** methods that create and return object references
- If you wish to use such a factory class to create Spring beans configured in XML, then you must configure the factory class and its factory method

Often in this scenario, in normal Java, you first create the factory object, then set characteristics on the factory to configure it. Then you can retrieve objects from the factory by calling an instance method on the factory.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Factory Classes, cont'd

```
1    public class StudentFactory
2    {
3      public Student getStudent()
4      {
5          return new Student(12, 3.44);
6      }
7    }
```

```
1    <bean id="myFactory"
2      class="mypackage.StudentFactory"/>
3
4    <bean id="student1" factory-bean="myFactory"
5        factory-method="getStudent" />
```


This factory class knows how to create Student objects.

In the XML, we define a bean for the factory itself, then the Student bean, which will be created by the factory. Note that we don't need to tell Spring the class of the Student bean – the container can determine the class by looking at the return type of the factory method.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About names and IDs for beans
- How bean scopes work
- About the lifecycle of beans

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

XML Dependency Injection

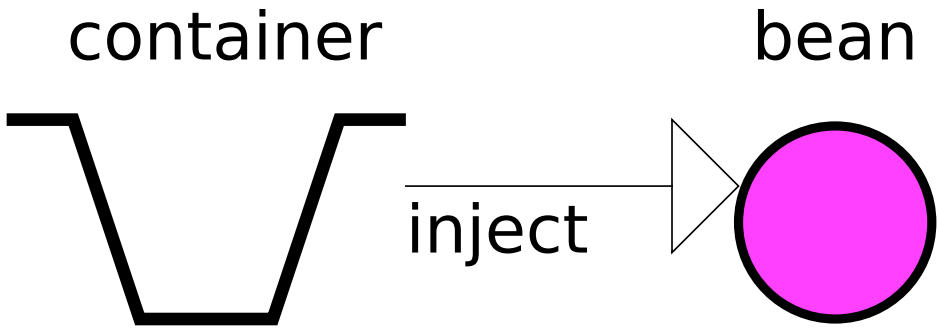
- Setter Injection
- Constructor Injection

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Dependency Injection

- **Dependency injection** is a technique in which the container instantiates and provides references for beans
- To inject a reference, the container either calls a bean's "set" method or constructor
- DI lessens coupling between beans and decouples beans from services such as JNDI



Dependency injection is a form of Inversion of Control</i> where the container supplies a bean with its dependencies.

The term "Dependency injection" was coined by Martin Fowler.

The traditional approach is to write an XML file to configure their beans and how the Spring container should inject dependencies.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Dependency Injection

- Spring provides several techniques so that developers can configure DI, including:
  - XML
  - Annotations
  - Spring 3 Java Configuration
- This chapter covers XML DI configuration

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Inversion of Control

- IoC is an architectural pattern in which the flow of control of a system is inverted in comparison to procedural programming
- Sometimes referred to as "The Hollywood Principle"
- Dependency injection is a form of IoC in which a container provides resources instead of the traditional practice of performing lookups

For a good read on Inversion of Control and Dependency Injection, go to:

`http://martinfowler.com/articles/injection.html`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Application for this Chapter

Student.java

Advisor.java

TestDI.java

spring.xml

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package university;
2
3    public class Student
4    {
5        private int id;
6        private String name;
7        private double gpa;
8        private Advisor theAdvisor;
9
10       public Student(String name, Advisor advisor)
11       {
12           this.name = name;
13           this.theAdvisor = advisor;
14       }
15
16       public Student(int id, double gpa)
17       {
18           this.id = id;
19           this.gpa = gpa;
20       }
21
22       public Student() {}
23
24       public Advisor getTheAdvisor()
25       {
26           return theAdvisor;
27       }
28
29       public void setTheAdvisor(Advisor theAdvisor)
30       {
31           this.theAdvisor = theAdvisor;
32       }
33
34       public int getId()
35       {
36           return id;
37       }
38
39       public void setId(int id)
40       {
41           this.id = id;
42       }
43
44       public String getName()
45       {
```

```
46          return name;
47      }
48
49      public void setName(String name)
50      {
51          this.name = name;
52      }
53
54      public double getGpa()
55      {
56          return gpa;
57      }
58
59      public void setGpa(double gpa)
60      {
61          this.gpa = gpa;
62      }
63
64      @Override
65      public String toString()
66      {
67          return "ID: " + id + " name: " + name + " GPA: " + gpa;
68      }
69  }
```

```
1    package university;
2
3    public class Advisor
4    {
5        private String name;
6        private double salary;
7
8        public String getName()
9        {
10           return name;
11       }
12
13       public void setName(String name)
14       {
15           this.name = name;
16       }
17
18       public double getSalary()
19       {
20           return salary;
21       }
22
23       public void setSalary(double salary)
24       {
25           this.salary = salary;
26       }
27
28       @Override
29       public String toString()
30       {
31           return "Name: " + name + " salary: " + salary;
32       }
33   }
```

```java
package main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import university.Student;

public class TestDI
{

    public static void main(String[] args)
    {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("spring.xml");

        Student s0 = (Student)ctx.getBean("student0");
        System.out.println(s0);

        Student s1 = (Student)ctx.getBean("student1");
        System.out.println(s1);

        Student s2 = (Student)ctx.getBean("student2");
        System.out.println(s2);

        Student s3 = (Student)ctx.getBean("student3");
        System.out.println(s3);

        Student s4 = (Student)ctx.getBean("student4");
        System.out.println(s4);

        Student s5 = (Student)ctx.getBean("student5");
        System.out.println(s5);
    }
}
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xsi:schemaLocation="http://www.springframework.org/schema/beans
5            http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6
7       <bean id="advisor1" class="university.Advisor"/>
8
9       <bean id="student0" class="university.Student">
10        <property name="name" value="Harry"/>
11        <property name="id" value="77"/>
12        <property name="gpa" value="3.14"/>
13        <property name="theAdvisor" ref="advisor1"/>
14      </bean>
15
16      <bean id="student1" class="university.Student">
17        <constructor-arg value="Joe"/>
18        <constructor-arg ref="advisor1"/>
19      </bean>
20
21      <bean id="student2" class="university.Student">
22        <constructor-arg ref="advisor1"/>
23        <constructor-arg value="Sue"/>
24      </bean>
25
26      <bean id="student3" class="university.Student">
27        <constructor-arg value="12"/>
28        <constructor-arg value="3.33"/>
29      </bean>
30
31      <bean id="student4" class="university.Student">
32        <constructor-arg value="3.33" index="1"/>
33        <constructor-arg value="12" index="0"/>
34      </bean>
35
36      <bean id="student5" class="university.Student">
37        <constructor-arg value="3.33" index="1"/>
38        <constructor-arg value="12" index="0"/>
39        <property name="name" value="George"/>
40        <property name="theAdvisor" ref="advisor1">
41        </property>
42      </bean>
43   </beans>
```

Setter Injection

- The JavaBean specification defines a **property** as data exposed via get/set methods
- Spring can use the "set" method on JavaBean-style properties to inject simple values and bean references



Student bean

```
public class Student
{
  public void setId(int id) {...}
  public void setName(String name) {...}
  public void setTheAdvisor(Advisor advisor) {...}
  public void setGpa(double gpa) {...}
 . . .
}
```

spring.xml

```
<bean id="advisor1" class="university.Advisor"/>

<bean id="student0" class="university.Student">
  <property name="name" value="Harry"/>
  <property name="id" value="77"/>
  <property name="gpa" value="3.14"/>
  <property name="theAdvisor" ref="advisor1"/>
</bean>
```


This has tradiitionally been the most common form of injection.

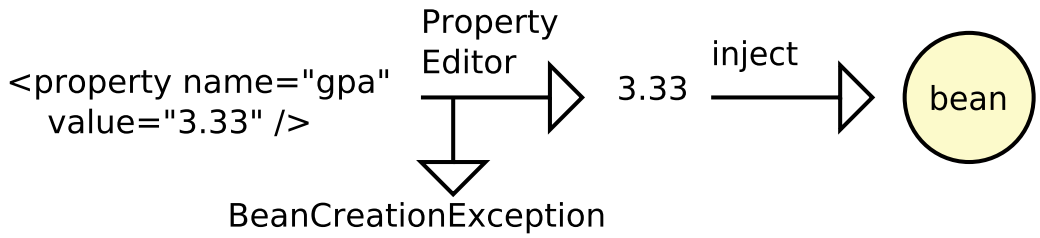
Note that for this to work, the bean must have a "set" method for the property that matches the JavaBean rules. The order of the "property" elements in the XML is not important.

Spring only uses the "set" method, so a "get" method is only required if your own logic needs it.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Property Conversions

- In the XML configuration, values are defined as character strings
- Spring converts the character strings as required to match the property type using standard JavaBean **property editors**



PropertyEditors are part of JavaBeans spec, but the spec just specifies the interface. However, Spring provides several PropertyEditors, many of which are from the sun.beans.editors package:

```
BooleanEditor.java
ByteEditor.java
ColorEditor.java
DoubleEditor.java
FloatEditor.java
FontEditor.java
IntegerEditor.java
LongEditor.java
NumberEditor.java
ShortEditor.java
StringEditor.java
```

If you search for "DoubleEditor.java" with Google, you can find the source code for this class and see how simple the conversion process is.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Constructor Injection

- Instead of, or in addition to using **setter** injection, Spring supports **constructor** injection that provides properties to a bean during instantiation
- The bean must define an appropriate constructor

Student bean

```
public Student(int id, double gpa){...}
```

spring.xml

```
<bean id="student3" class="university.Student">
  <constructor-arg value="12"/>
  <constructor-arg value="3.33"/>
</bean>
```


If you do NOT use constructor injection, then the bean MUST have a public zero-argument constructor.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Constructor Injection Resolution

- When you specify constructor arguments, Spring examines the bean for a matching constructor
- Spring attempts to use the types used in the XML file, but note that **values** don't have a type (they are character strings in XML)

Student bean

```
public Student(String name, Advisor advisor) {...}
```

spring.xml

```
<bean id="student1" class="university.Student">
  <constructor-arg value="Joe"/>
  <constructor-arg ref="advisor1"/>
</bean>

<bean id="student2" class="university.Student">
  <constructor-arg ref="advisor1"/>
  <constructor-arg value="Sue"/>
</bean>
```


In this case, since one of the constructor-args is a reference to an Advisor bean, Spring doesn't care about the order in which you write the constructor-arg elements. In other words, since Spring can determine the type of the "ref" constructor-arg, and there are two constructor-args, Spring can match up with the actual Java constructor.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Constructor Injection Resolution, cont'd

- To aid resolution, you can use the optional **type** and/or **index** attributes in the XML
- Note that if you specify **constructor-args** in the same order as they appear in the bean's constructor, this is often not necessary

Student bean

```
public Student(int id, double gpa){...}
```

spring.xml

```
<bean id="student3" class="university.Student">
  <constructor-arg value="12"/>
  <constructor-arg value="3.33"/>
</bean>

<bean id="student4" class="university.Student">
  <constructor-arg value="3.33" index="1"/>
  <constructor-arg value="12" index="0"/>
</bean>
```


For "student4", both of the constructor-args are values, not references, so Spring cannot determine the type. And since we put a string with a decimal point ("3.3") as the first argument, Spring would try to convert it to an integer and fail. Thus, the "index" attribute is necessary.

Note also that there's a "type" attribute you can write on a constructor-arg element that lets you explicitly specify an argument's type. That may help resolve ambiguity.

```
<bean id="student4" class="university.Student">
  <constructor-arg value="3.33"/>
  <constructor-arg value="12" type="int"/>
</bean>
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mixing Injection Types

- It's OK to use both setter and constructor injection on a bean
- Spring will first create the bean using the constructor arguments, then invoke the set method(s)

Student bean

```
public Student(int id, double gpa) {...}
public void setName(String name) {...}
public void setTheAdvisor(Advisor advisor) {...}
```

spring.xml

```
<bean id="student5" class="university.Student">
  <constructor-arg value="3.33" index="1"/>
  <constructor-arg value="12" index="0"/>
  <property name="name" value="George"/>
  <property name="theAdvisor" ref="advisor1">
  </property>
</bean>
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Setter vs Constructor Injection

- The Spring documentation favors setter injection over constructor injection since it's simpler and more intuitive
- Constructor injection does allow you to guarantee intialization (especially if you ONLY allow constructor injection)
- In Spring 2.5 and later, you can use the **@Required** annotation on a "set" method to guarantee initialization





Note that if you want to use the @Required annotation, you will also need to define this bean so that Spring can process the annotation:

```
<bean class=
  "org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- How to inject values and references via **setter** injection
- How to inject values and references via **constructor** injection

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Spring MVC

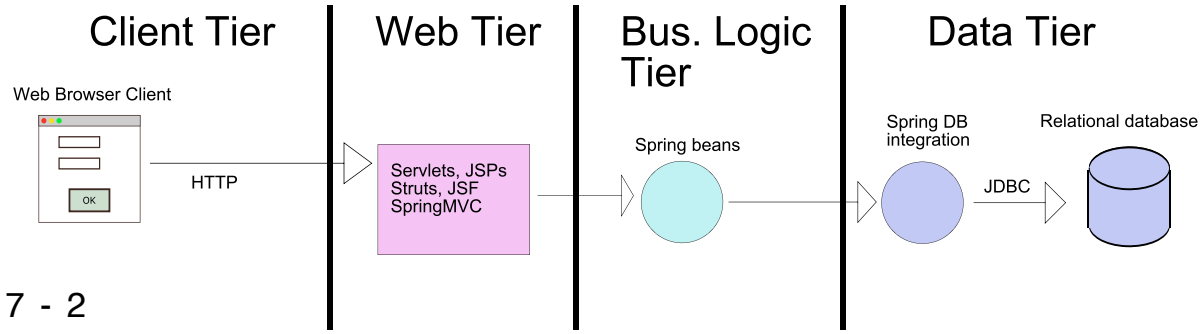
- MVC Fundamentals
- Spring MVC Architecture
- Spring MVC Sample Application

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring and the Web

- There are three basic strategies for fronting Spring applications with a Web front end:
  - Use standard JEE servlets and JSPs as the Web tier
  - Use a JEE Web framework such as Struts or JSF as the Web tier
  - Use the SpringMVC framework as the Web tier



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Spring WebApplicationContext

- If you simply want to use a Spring business layer from a servlet/JSP, Struts or JSF Web app, Spring provides a **context listener** and a **WebApplicationContext**

```
1    web.xml fragment
2    ----------------
3
4    <context-param>
5      <param-name>contextConfigLocation</param-name>
6      <param-value>/WEB-INF/spring.xml</param-value>
7    </context-param>
8    <listener>
9      <display-name>ContextLoaderListener</display-name>
10     <listener-class>
11      org.springframework.web.context.ContextLoaderListener
12     </listener-class>
13   </listener>
```

A ServletContextListener is a standard part of JEE Web apps it's a class that implements that interface and is configured in web.xml. The interface has two methods: contextCreated and contextDestroyed the servlet container calls those methods when the Web app starts and is about to go away. Spring provides a ServletContextListener that creates a Spring ApplicationContext.

The bad news here is the unfortunate overloading of the term context. We have a servlet context (aka Web application context) and we have a Spring application context. They are different things. The servlet context is a part of the JEE specification, while, as we've already seen, the Spring application context represents the Spring container and does stuff like letting you retrieve beans by name.

The "context-param" element tells the listener where the Spring configuration files are located, typically within the WEB-INF folder of the Web application.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Spring WebApplicationContext, cont'd

```
1    Servlet class fragment
2    ----------------------
3
4    protected void doGet(HttpServletRequest request,
5     HttpServletResponse response)
6       throws ServletException, IOException
7    {
8      ServletContext servletContext = getServletContext();
9      WebApplicationContext ctx =
10       WebApplicationContextUtils.
11          getRequiredWebApplicationContext(
12           servletContext);
13
14     Student s1 = (Student)ctx.getBean("student1");
15
16     response.setContentType("text/html");
17     PrintWriter out = response.getWriter();
18     out.println("Student name: " + s1.getName() + "<br>");
19   }
```

A WebApplicationContext is an extension of the "normal" application context covered earlier in the course. The primary difference is that it can work with JEE Web application "scopes" such as request and session.

Here we show a fragment of a servlet that first retrieves the WebApplicationContext reference from the location where the Spring context-listener stored it. We use the Spring-provided WebApplicationContextUtils class to retrieve the context.

Once we have the WebApplicationContext, we can then retrieve and access beans in the normal Spring fashion.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

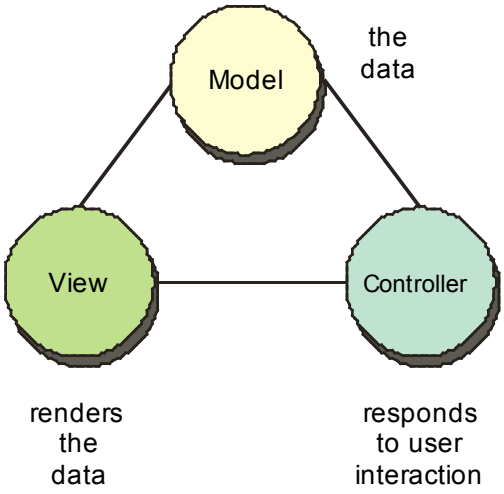
Introduction to Spring MVC

- Spring MVC is a Spring module that makes it fairly easy to create sophisticated Web front ends that use the Model-View-Controller design pattern
- You can using Spring MVC as an alternative to standard servlets/JSPs, Struts, JSF or other JEE Web frameworks
- Version 2.5 and later support using **annotations** to configure how Spring MVC maps *controllers* and maps requests

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is the MVC Design Pattern?

- The **Model-View-Controller** (MVC) architecture originated in the 1980s in the Smalltalk language, where it was used to build graphical user interfaces
- MVC is effective since it decomposes applications into components that you can develop, debug and maintain separately

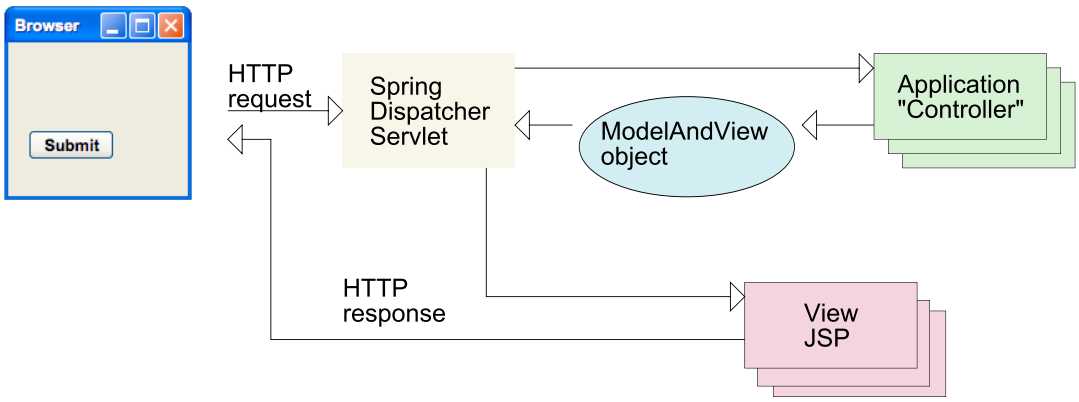


Other Java Web frameworks such as Struts, JSF and JEE 8 MVC use this pattern to varying degrees.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring MVC Architecture

- Spring MVC supplies a **front controller** servlet that acts as a single point of entry to an application
- Developers write "command controllers" that respond to requests from client



Note that if you use annotations, you don't need to explicitly create the ModelAndView object.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring MVC Controllers

- An application **controller** (not to be confused with a *front* controller), is an object that responds to a request, typically by populating a model and specifying a view
- In Spring 2.5 and later, controllers can be POJOs that you configure using annotations

```
1    @Controller
2    @RequestMapping("/formula")
3    public class QuadraticController
4    {
5      . . .
6    }
```

The @Controller annotation is a special case of the @Component annotation and configures the class as a Spring MVC controller - for it to work, you need to write an entry in a Spring configuration XML file:

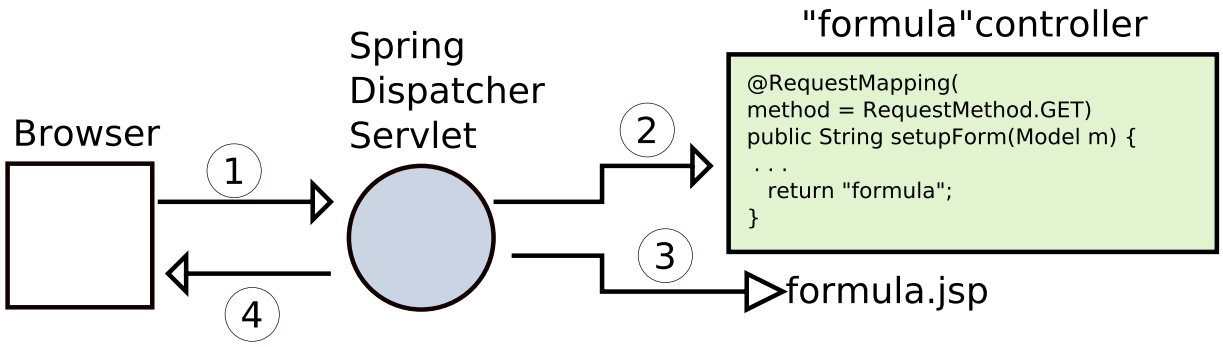
```
<context:component-scan base-package="my-controller-package" />
```

The @RequestMapping annotation assigns a URL to the controller.

In earlier versions of SpringMVC, controllers typically subclass either AbstractController and override handleRequestInternal(), or SimpleFormController and override onSubmit().

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Form Processing Flow: Phase 1



1. Browser sends GET /formula.html
2. DispatcherServlet finds controller with "formula" URL and invokes a "setup" method configured with @RequestMapping(method = RequestMethod.GET). The "setup" method optionally initializes a model and returns a String with a JSP name.
3. The DispatcherServlet executes the JSP, initializing form fields from model
4. DispatcherServlet sends response with form back to browser

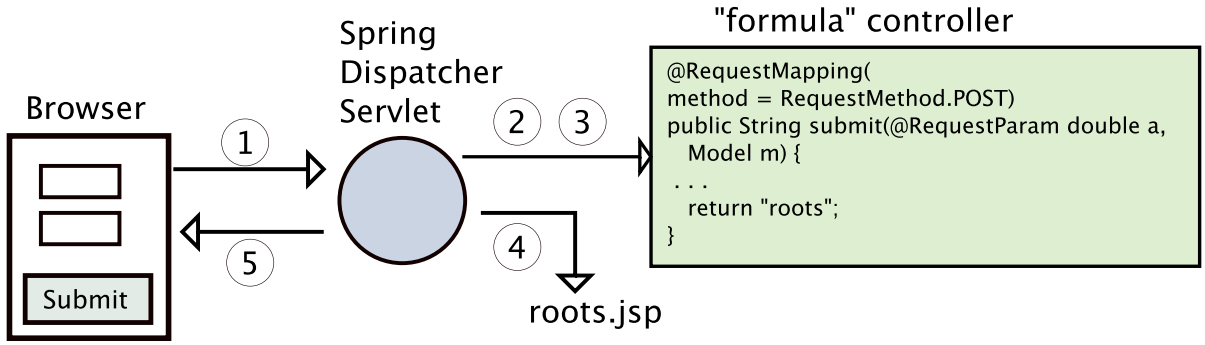
---

The first phase of the flow happens when a browser issues a GET request. The DispatcherServlet finds the matching controller bean, finds a "setup" method, which returns a String that specifies a JSP name. The servlet generates a HTML response back to the browser using the JSP as a template.

After this phase is complete, the browser displays a user input form, complete with a submit button. The user then completes the form and presses the submit button, which initiates Phase 2.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Form Processing Flow: Phase 2



1. Browser sends POST /formula.html
2. DispatcherServlet maps HTTP request parameters to "submit" method parameters
3. DispatcherServlet invokes "submit" method annotated with @RequestMapping(method = RequestMethod.POST). The "submit" method processes request, populates model and specifies view JSP
4. DispatcherServlet forwards request to view JSP - the JSP accesses the model and generates a response
5. DispatcherServlet returns response to browser

Phase 2 entails actually processing the request and generating a response. The DispatcherServlet can differentiate between the phases since Phase 1 is an HTTP GET, while Phase 2 is a POST.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring MVC Development Steps

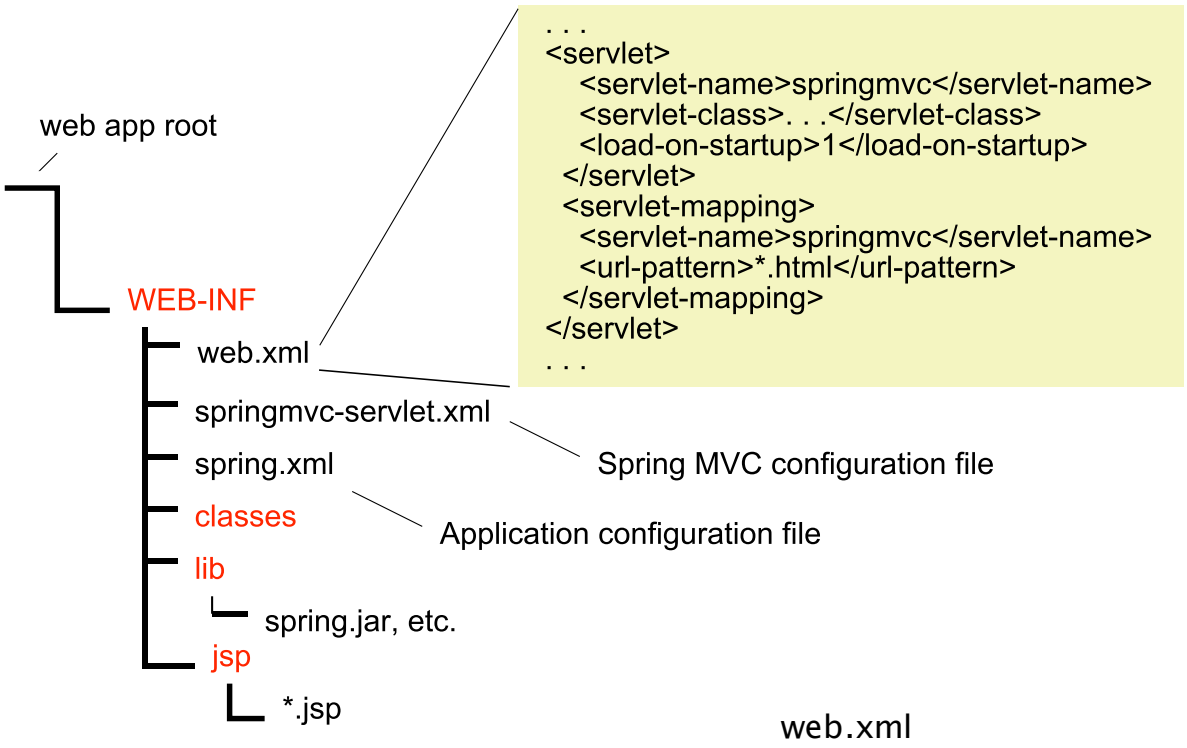
For a Spring SimpleFormController app:

- Configure the Web application:
  - Configure Spring MVC Dispatcher servlet in web.xml
  - Add Spring MVC JARS to WEB-INF/lib
- Create an HTML input form, optionally using Spring MVC custom actions
- Create an application **controller** class to process the form
- Write Spring configuration XML file(s)

Note that if we use annotations or Java configuration, the Spring XML file(s) can be minimal.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring the Web Application



Here we show the Web application configuration for a simple application. It's a good idea to create a separate folder in the WEB-INF directory for the JSPs - that will make it easier to map them as Spring MVC views.

We show a fragment of the web.xml deployment descriptor. Note the servlet configuration - we used the name "springmvc" and set up a URL pattern of *.html. That will route all requests ending in .html to the Spring MVC servlet.

Note also that since we named the servlet "springmvc", the Spring MVC configuration file is named "springmvc-servlet.xml". You can define additional "application" Spring configuration files using the context-listener approach shown earlier in this chapter.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3        xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/n
4        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/
5        id="WebApp_ID" version="3.0">
6
7        <listener>
8            <listener-class>org.springframework.web.util.Log4jConfigListener</listener
9        </listener>
10       <listener>
11           <display-name>ContextLoaderListener</display-name>
12           <listener-class>
13               org.springframework.web.context.ContextLoaderListener
14           </listener-class>
15       </listener>
16       <context-param>
17           <param-name>contextConfigLocation</param-name>
18           <param-value>/WEB-INF/spring.xml</param-value>
19       </context-param>
20       <servlet>
21           <servlet-name>springmvc</servlet-name>
22           <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
23           <load-on-startup>1</load-on-startup>
24       </servlet>
25       <servlet-mapping>
26           <servlet-name>springmvc</servlet-name>
27           <url-pattern>*.html</url-pattern>
28       </servlet-mapping>
29       <welcome-file-list>
30           <welcome-file>index.jsp</welcome-file>
31       </welcome-file-list>
32   </web-app>
```

A Service Bean

- In the normal fashion, you can write Spring beans that provide application services
- As always, it's a good idea to write an interface first for application service beans

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

SolveQuadratic.java

SolveQuadraticImpl.java

This service bean represents the quadratic formula, which solves equations of the form ax2 + bx +c.

Note how we annotate the implementation class with @Component so we don't need to configure it in an XML file.

We will need to configure annotation processing in the "application" spring.xml file:

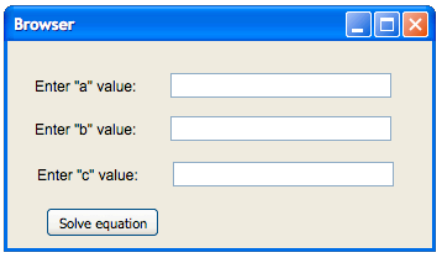
```
<context:annotation-config/>
<context:component-scan base-package="equations"/>
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package equations;
2
3    public interface SolveQuadratic
4    {
5        public double getRoot1(double a, double b, double c);
6        public double getRoot2(double a, double b, double c);
7    }
```

```
1    package equations;
2
3    @Component("solveQuadratic")
4    public class SolveQuadraticImpl implements SolveQuadratic
5    {
6
7        public double getRoot1(double a, double b, double c)
8        {
9            return (-b + Math.sqrt(b * b - 4 * a * c)) / 2 * a;
10       }
11
12       public double getRoot2(double a, double b, double c)
13       {
14           return (-b - Math.sqrt(b * b - 4 * a * c)) / 2 * a;
15       }
16   }
```

A Spring MVC Input Form



formula.jsp

Here we show formula.jsp, which resides in the "jsp" folder and defines a Spring MVC view containing an HTML input form.

Note the form's "action" – formula.html. We map all URLs ending *.html to the Spring MVC servlet, so it will process the request and invoke our controller.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    <html>
2      <head>
3        <title>Solve The Quadratic Formula</title>
4      </head>
5      <body>
6        <h3>Solve The Quadratic Formula</h3>
7        <p>
8          The Quadratic Formula solves equations of the form   
9          <img src="equation.png"> using this formula:
10       </p>
11       <p>
12         <img src="QuadraticFormula.jpg">
13       </p>
14       <form action="formula.html" method="post">
15         <table>
16           <tr>
17             <td>Enter <i>a</i> value:</td>
18             <td><input type="text" name="a" value="${a}" /></td>
19           </tr>
20           <tr>
21             <td>Enter <i>b</i> value:</td>
22             <td><input type="text" name="b" value="${b}" /></td>
23           </tr>
24           <tr>
25             <td>Enter <i>c</i> value:</td>
26             <td><input type="text" name="c" value="${c}" /></td>
27           </tr>
28           <tr>
29             <td>
30               <input type="submit" value="Solve equation">
31             </td>
32           </tr>
33         </table>
34       </form>
35     </body>
36   </html>
```

A Spring MVC Result Page

```
1    <html>
2      <head>
3        <title>Equation Solved</title>
4      </head>
5      <body>
6         <h3>Equation Solved</h3>
7         <p>For input values:</p>
8
9          a=${a}<br>
10         b=${b}<br>
11         c=${c}<br>
12
13        <p><b>Root 1:</b> ${root1}</p>
14        <p><b>Root 2:</b> ${root2}</p>
15     </body>
16   </html>
```

Here we show the view JSP that displays the results of solving the equation.

Note that it uses the JSP Expression Language to display values named "a", "b", "c", "root1" and "root2" - our controller will need to store those values into the model. Then the dispatcher servlet will write them into request scope.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Controller Annotation Processing

- Since our controller uses annotations, we must enable annotation processing in the Spring MVC configuration file

```
1    Spring MVC configuration file
2    ----------------------------
3
4    <context:component-scan
5         base-package="controllers" />
```


This configuration is in the springmvc-servlet.xml file, and tells Spring MVC which packages potentially contain controllers annotated with @Controller, @ReqestMapping and @RequestParam.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is a ViewResolver?

- So that controllers do not need to hard-code the file names of JSPs, SpringMVC provides **ViewResolver** classes that map logical strings into actual file names

```
1    Spring MVC configuration file
2    -----------------------------
3
4    <bean id="viewResolver"
5      class=
6    "org.springframework.web.servlet.view.InternalResourceViewResolver">
7      <property name="viewClass"
8        value="org.springframework.web.servlet.view.JstlView" />
9      <property name="prefix" value="/WEB-INF/jsp/" />
10     <property name="suffix" value=".jsp" />
11   </bean>
```

For example, if a controller method returns "roots" as a view logical string, this resolver will map to a JSP named "/WEB-INF/jsp/roots.jsp"

This configuration is in the springmvc-servlet.xml file.

The InternalResourceViewResolver does a simple mapping by applying both a prefix and a suffix to a logical view name.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Spring MVC Command Controller

QuadraticEquationController.java

Here we show a listing for a controller for the quadratic equation processing application.

We provide a "setupForm" method to provide a model that will intialize the form fields. Note that we pre-initialize it with sample values for the equation.

We provide a "submitForm" method to process the form. The Spring MVC servlet maps form HTTP parameters to method parameters, then invokes our method when the user posts the form.

After solving the equation, we put the original request values and the two roots into the model, then return a view JSP name. The Spring MVC servlet stores any such model attributes at request scope so that the view JSP can access the model data.

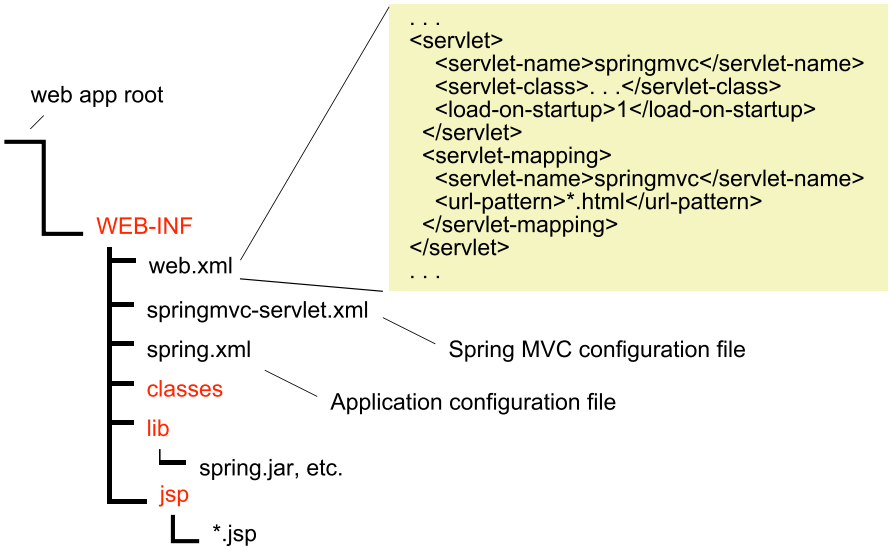
Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package controllers;
2
3    import org.springframework.beans.factory.annotation.Autowired;
4    import org.springframework.context.ApplicationContext;
5    import org.springframework.stereotype.Controller;
6    import org.springframework.ui.Model;
7    import org.springframework.web.bind.annotation.RequestMapping;
8    import org.springframework.web.bind.annotation.RequestMethod;
9    import org.springframework.web.bind.annotation.RequestParam;
10
11   import equations.SolveQuadratic;
12
13   @Controller
14   @RequestMapping("/formula")
15   public class QuadraticEquationController
16   {
17       @Autowired private ApplicationContext ctx;
18
19       @RequestMapping(method = RequestMethod.GET)
20       public String setupForm(Model model)
21       {
22           model.addAttribute("a", "1");
23           model.addAttribute("b", "3");
24           model.addAttribute("c", "-4");
25
26           return "formula";
27       }
28
29       @RequestMapping(method = RequestMethod.POST)
30       public String submitForm(@RequestParam double a, @RequestParam double b,
31               @RequestParam double c, Model model)
32       {
33           SolveQuadratic solver = (SolveQuadratic)ctx.getBean("solveQuadratic");
34
35           model.addAttribute("a", a);
36           model.addAttribute("b", b);
37           model.addAttribute("c", c);
38
39           model.addAttribute("root1", solver.getRoot1(a, b, c));
40           model.addAttribute("root2", solver.getRoot2(a, b, c));
41
42           return "roots";
43       }
44   }
```

The Spring Configuration File

- By default, the Spring MVC servlet looks in the WEB-INF folder for a file with name xxxxx-servlet.xml where xxxxx is the servlet name in web.xml

springmvc-servlet.xml



Here we show the complete Spring MVC configuration file. The first bean, the "viewResolver", tells Spring MVC how to map view names to JSPs. We've configured it so that a view named "myview" corresponds to a file named "myview.jsp" in a directory named "jsp" in the Web application's WEB-INF directory.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.sp
4        xmlns:context="http://www.springframework.org/schema/context"
5        xmlns:jee="http://www.springframework.org/schema/jee" xmlns:jms="http://www.sp
6        xmlns:lang="http://www.springframework.org/schema/lang" xmlns:tx="http://www.s
7        xmlns:util="http://www.springframework.org/schema/util"
8        xsi:schemaLocation="http://www.springframework.org/schema/aop
9            http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
10           http://www.springframework.org/schema/beans
11           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
12           http://www.springframework.org/schema/context
13           http://www.springframework.org/schema/context/spring-context-2.5.xsd
14           http://www.springframework.org/schema/jee
15           http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
16           http://www.springframework.org/schema/lang
17           http://www.springframework.org/schema/lang/spring-lang-2.5.xsd
18           http://www.springframework.org/schema/tx
19           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
20           http://www.springframework.org/schema/util
21           http://www.springframework.org/schema/util/spring-util-2.5.xsd">
22
23       <context:component-scan base-package="controllers" />
24
25       <bean id="viewResolver"
26           class="org.springframework.web.servlet.view.InternalResourceViewResolver">
27           <property name="viewClass"
28               value="org.springframework.web.servlet.view.JstlView" />
29           <property name="prefix" value="/WEB-INF/jsp/" />
30           <property name="suffix" value=".jsp" />
31       </bean>
32
33   </beans>
```

MVC With Java Configuration

- If you use Java configuration for Spring MVC, you can avoid writing any Spring XML
- Recipe:
    - Write a configuration class annotated with @Configuration, @ComponentScan and @EnableWebMvc
    - In the configuration class, write bean-creation methods to configure a **view resolver** and any other application beans
    - Configure **contextClass** and **contextConfigLocation** in web.xml

javaconfg-web.xml

QuadraticConfig.java

If you use Java configuration, you don't need any Spring configuration files, but you may need to write elements in the standard web.xml.

The proper contexClass is org.springframework.web.context.support.AnnotationConfigWebApplicationContext.

If you are deploying to a JEE Web container that supports the servlet specification version 3 or later, you can even avoid writing the web.xml file and do all of your configuration in Java. For an example, see:

https://stackoverflow.com/questions/22315672/

how-to-configure-spring-mvc-with-pure-java-based-configuration

NOTE: The name of the Web deployment descriptor must be web.xml. We have changed its name for this presentation to avoid a name conflict, so if you use the one shown here, be sure to rename it web.xml.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3        xmlns="http://java.sun.com/xml/ns/javaee"
4        xmlns:web="http://java.sun.com/xml/ns/javaee"
5        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/
6        id="WebApp_ID" version="3.0">
7        <servlet>
8            <servlet-name>springmvc</servlet-name>
9            <servlet-class>
10             org.springframework.web.servlet.DispatcherServlet
11             </servlet-class>
12           <init-param>
13               <param-name>contextClass</param-name>
14               <param-value>
15                 org.springframework.web.context.support.AnnotationConfigWebApplicati
16               </param-value>
17           </init-param>
18           <init-param>
19               <param-name>contextConfigLocation</param-name>
20               <param-value>config.QuadraticConfig</param-value>
21           </init-param>
22           <load-on-startup>1</load-on-startup>
23       </servlet>
24       <servlet-mapping>
25           <servlet-name>springmvc</servlet-name>
26           <url-pattern>*.html</url-pattern>
27       </servlet-mapping>
28       <welcome-file-list>
29           <welcome-file>index.jsp</welcome-file>
30       </welcome-file-list>
31   </web-app>
```

```
1    package config;
2
3    import org.springframework.context.annotation.Bean;
4    import org.springframework.context.annotation.ComponentScan;
5    import org.springframework.context.annotation.Configuration;
6    import org.springframework.web.servlet.ViewResolver;
7    import org.springframework.web.servlet.config.annotation.EnableWebMvc;
8    import org.springframework.web.servlet.view.InternalResourceViewResolver;
9
10   import equations.SolveQuadratic;
11   import equations.SolveQuadraticImpl;
12
13   @Configuration
14   @EnableWebMvc
15   @ComponentScan("controllers")
16   public class QuadraticConfig
17   {
18       @Bean
19       public ViewResolver configureViewResolver()
20       {
21           InternalResourceViewResolver viewResolve =
22               new InternalResourceViewResolver();
23           viewResolve.setPrefix("/WEB-INF/jsp/");
24           viewResolve.setSuffix(".jsp");
25
26           return viewResolve;
27       }
28
29       @Bean
30       public SolveQuadratic solveQuadratic()
31       {
32           return new SolveQuadraticImpl();
33       }
34   }
```

Using Thymeleaf Instead of JSP

- **Thymeleaf** is a template engine that many Spring developers prefer instead of JSPs for the view:
  - Supports HTML5 out of the box and lets you better preview pages in a browser (JSPs often need to run tag libraries)
  - There is a Spring/Thymeleaf integration library



Thymeleaf's home page is http://www.thymeleaf.org/index.html.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Thymeleaf/Spring Recipe

- Steps for using Thymeleaf:
  - Load the Thymeleaf libraries into the project
  - Write HTML5, XML-parseable Thymeleaf **templates** instead of JSPs
  - Configure Thymeleaf template-engine and view-resolver beans

formula.html

roots.html

QuadraticThymeleafConfig.java

Note that Thymeleaf templates must be well formed XML – that means that you need to close all tags, even those such as <img> that we don't normally explicitly close in HTML5.

Also note that with Thymeleaf, we generally map the Spring Dispatcher servlet to a URL pattern of '/'. That means that we need to do a bit more configuration so that the dispatcher servlet sends static resources such as images from the Web app's root directory.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    <!DOCTYPE html>
2    <html>
3      <head>
4        <title>Solve The Quadratic Formula</title>
5      </head>
6      <body>
7        <h3>Solve The Quadratic Formula</h3>
8        <p>
9          The Quadratic Formula solves equations of the form   
10         <img src="equation.png" /> using this formula:
11       </p>
12       <p>
13         <img src="QuadraticFormula.jpg" />
14       </p>
15       <form action="formula.html" method="post">
16         <table>
17           <tr>
18             <td>Enter <i>a</i> value:</td>
19             <td><input type="text" name="a" th:value="${a}"/></td>
20           </tr>
21           <tr>
22             <td>Enter <i>b</i> value:</td>
23             <td><input type="text" name="b"  th:value="${b}"/></td>
24           </tr>
25           <tr>
26             <td>Enter <i>c</i> value:</td>
27             <td><input type="text" name="c"  th:value="${c}" /></td>
28           </tr>
29           <tr>
30             <td>
31               <input type="submit" value="Solve equation" />
32             </td>
33           </tr>
34         </table>
35       </form>
36     </body>
37   </html>
```

```
1    <!DOCTYPE html>
2    <html>
3      <head>
4        <title>Equation Solved</title>
5      </head>
6      <body>
7         <h3>Equation Solved</h3>
8         <p>For input values:</p>
9
10         <div>
11            a=<span th:text="${a}"></span>
12         </div>
13         <div>
14            b=<span th:text="${b}"></span>
15         </div>
16         <div>
17            c=<span th:text="${c}"></span>
18         </div>
19
20        <p><b>Root 1:</b> <span th:text="${root1}"></span></p>
21        <p><b>Root 2:</b> <span th:text="${root2}"></span></p>
22     </body>
23   </html>
```

```
1    package config;
2
3    import org.springframework.context.annotation.Bean;
4    import org.springframework.context.annotation.ComponentScan;
5    import org.springframework.context.annotation.Configuration;
6    import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConf
7    import org.springframework.web.servlet.config.annotation.EnableWebMvc;
8    import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
9    import org.thymeleaf.spring4.SpringTemplateEngine;
10   import org.thymeleaf.spring4.view.ThymeleafViewResolver;
11   import org.thymeleaf.templateresolver.ServletContextTemplateResolver;
12
13   import equations.SolveQuadratic;
14   import equations.SolveQuadraticImpl;
15
16   /*
17    *   Note that we extend WebMvcConfigurerAdaptor
18    *   so we can override configureDefaultServletHandling().
19    *   That lets us configure the Spring MVC servlet so
20    *   it loads static resources (e.g. images) from the
21    *   WebContent folder (context root).
22    */
23
24   @Configuration
25   @EnableWebMvc
26   @ComponentScan("controllers")
27   public class QuadraticThymeleafConfig extends WebMvcConfigurerAdapter
28   {
29       @Bean
30       public ServletContextTemplateResolver templateResolver()
31       {
32           ServletContextTemplateResolver resolver =
33              new ServletContextTemplateResolver();
34
35           resolver.setPrefix("/WEB-INF/templates/");
36           resolver.setSuffix(".html");
37           resolver.setTemplateMode("HTML5");
38           resolver.setCacheable(false);
39
40           return resolver;
41       }
42
43       @Bean
44       SpringTemplateEngine templateEngine()
45       {
```

```
46          SpringTemplateEngine templateEngine = new SpringTemplateEngine();
47
48          templateEngine.setTemplateResolver(templateResolver());
49
50          return templateEngine;
51      }
52
53      @Bean
54      public ThymeleafViewResolver viewResolver()
55      {
56          ThymeleafViewResolver tvr = new ThymeleafViewResolver();
57          tvr.setTemplateEngine(templateEngine());
58
59          return tvr;
60      }
61
62      @Override
63      public void configureDefaultServletHandling(
64              DefaultServletHandlerConfigurer configurer)
65      {
66          configurer.enable();
67      }
68
69      @Bean
70      public SolveQuadratic solveQuadratic()
71      {
72          return new SolveQuadraticImpl();
73      }
74  }
```

Chapter Summary

In this chapter, you learned:

- Fundamental Model-View-Controller concepts
- The basics of the Spring MVC module

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring and JDBC

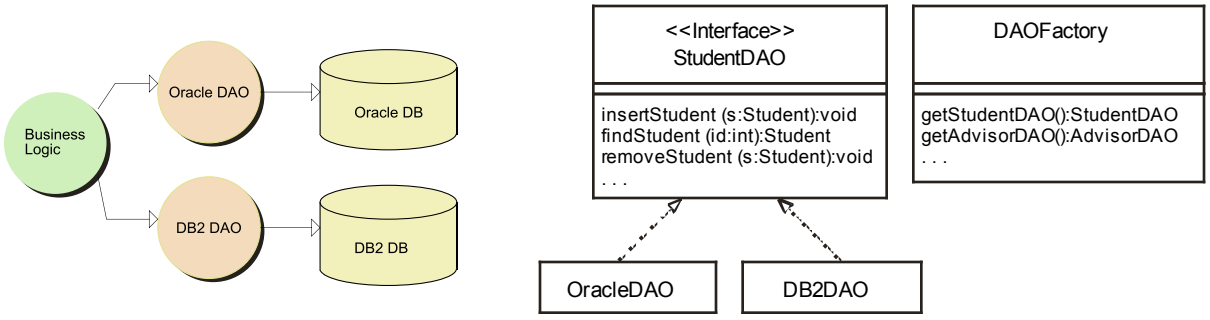
- The DAO Design Pattern
- JdbcTemplate and SimpleJdbcTemplate
- JdbcDaoSupport

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The DAO Design Pattern

- Spring supports and encourages the JEE DAO design pattern and simplifies coding, even if you you use basic JDBC



In a perfect world, or if JDBC was perfect, changing databases would not require any new code. However, in the real world, if you decide to change databases, at least some of your JDBC code probably will be affected. The goal of the DAO pattern is to minimize such changes if we do make a change.

By defining an interface for the DAO, you can minimize the impact of changes on the business logic. Here we show defining a StudentDAO interface that specifies the required behavior for any Student data-access object. The Oracle-specific and DB2-specific classes implement the interface with database-specific behavior.

The DAO Factory interface is an optional part of the pattern that is not really needed in Spring, since Spring programs generally use dependency injection to "hide" the creation of objects, including the DAO objects themselves.

You can read more about the DAO design pattern at:

`http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What's Wrong with JDBC?

- Even if you use DAOs, JDBC involves a lot of low-level, tedious and repetitive coding
- Managing JDBC connections is prone to errors
- OR/M solutions like Hibernate help with some of these issues

```
1    Connection con = null;
2    try {
3      con = dataSource.getConnection();
4      . . .
5    }
6    catch (SQLException exc) {
7    . . . }
8    finally {
9      try {
10       if (con != null) con.close();
11     catch (Exception e) {}
12   }
```

The snippet of code shown here should look familiar to anyone who's worked much with JDBC. Note the "try-catch" and "finally" blocks and how such programs need to manually manage connections.

While this code is not all that conceptually difficult, it is tedious and verbose.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

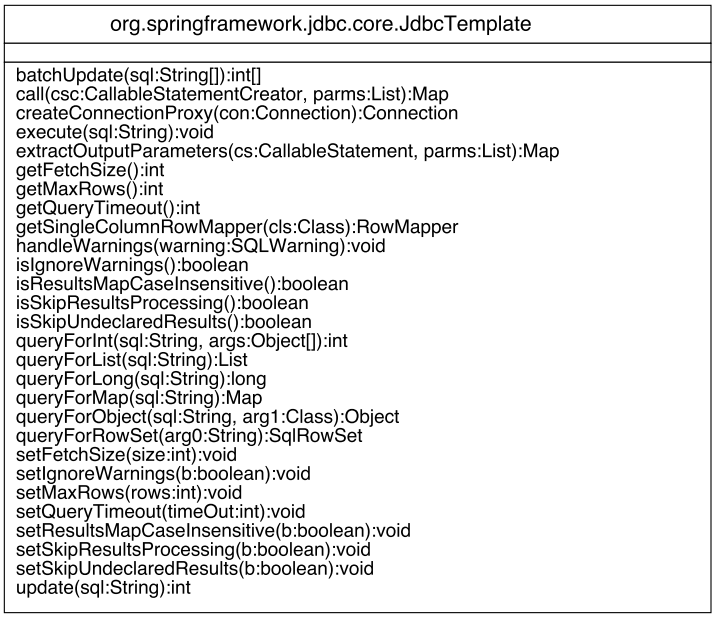
Introducing the Spring JDBC Module

- Spring provides the following support for database access:
    - Classes that retrieve an underlying JDBC **data source**
    - **Template** classes that simplify database coding, including templates for basic JDBC, Hibernate and JPA
    - **Support** classes that you can subclass when writing your DAOs
    - An improved exception hierarchy that's based on unchecked (runtime) exceptions

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JdbcTemplate Class

- This is the workhorse class for programs that use Spring's basic JDBC support
- It provides methods to let you query, update, insert and delete rows in a table

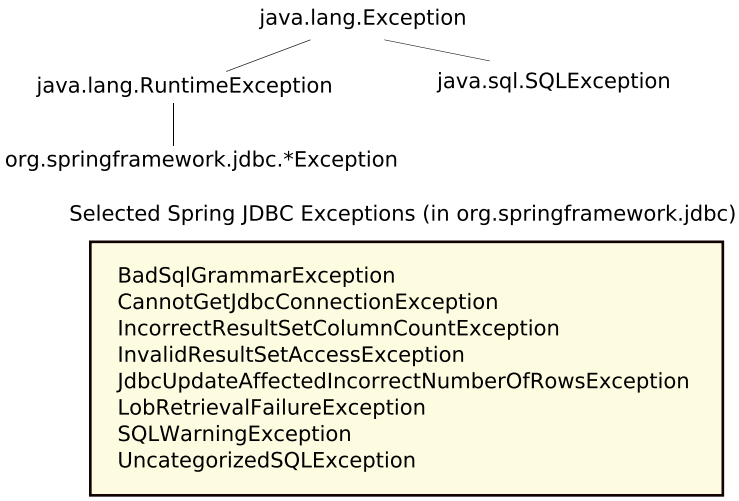


Note that there are many more methods in this class that we didn't show in the figure so that the figure fits on the page.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring JDBC Exceptions

- JDBC provides only the single SQLException and it's a **checked** exception, which can require messy nested **try-catch** blocks
- Spring provides many more precisely named **unchecked** exceptions -- exception handling is simpler and more intuitive



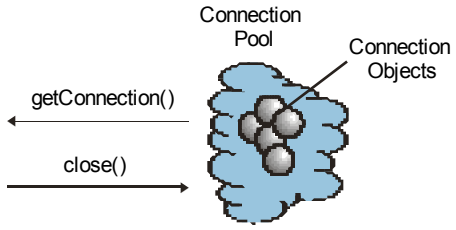
The Spring exceptions are unchecked, so you don't need try-catch blocks. This is a standard Spring philosophy. The idea is that you cannot recover from most JDBC exceptions, so why clutter the code with all of the try-catch blocks? You will need a try-catch block SOMEWHERE in the program to avoid uncaught exceptions, however.

To support this, Spring provides the SQLErrorCodeSQLExceptionTranslator class which consults a Spring-provided table of vendor error codes and their corresponding Spring exceptions.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JDBC Connection Review

- Non-Spring JDBC applications can obtain a connection in one of two ways:
  - Using the DriverManager interface
  - Using a DataSource
- The DriverManager approach is appropriate for standalone Java programs or for testing of enterprise applications
- The DataSource approach requires a JEE runtime that provides a JNDI service, but allows for **pooled** connections



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Working with Data Sources in Spring

- Spring lets you treat both JDBC approaches in a similar fashion
- Spring provides:
  - A DriverManagerDataSource class that uses the JDBC DriverManager approach
  - A JndiObjectFactoryBean class or a <jee: jndi-lookup> configuration element that locate JNDI-based DataSources
- Since you configure these in the Spring configuration file, you can easily switch between the two approaches

In Spring 1, we used the JndiFactoryBean, but in Spring 2 and later, it's easier to use the "jndi-lookup" element in the Spring configuration file to retrieve DataSource objects from JNDI.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a DriverManager Source (XML)

- The Spring DriverManagerDataSource wraps the JDBC DriverManager and is appropriate for standalone programs
- You can inject the data source bean into your DAOs using XML, annotations or Java configuration

```
1    <bean id="studentDataSource"
2      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
3      <property name="driverClassName"
4        value="org.apache.derby.jdbc.EmbeddedDriver"/>
5      <property name="url"
6        value="jdbc:derby:c:\springclass\db\StudentDB"/>
7      <property name="username"
8        value="SA"/>
9      <property name="password"
10       value="SA"/>
11   </bean>
```

The DriverManagerDataSource encapsulates a non-pooled JDBC DataSource. Note that we must provide connection properties.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a DriverManager (Java Config)

- With Java configuration, you can use the **DataSourceBuilder** to create and initialize a driver-manager DataSource

```
1    @Bean
2    public DataSource studentDataSource()
3    {
4      DataSourceBuilder dsb =
5          DataSourceBuilder.create();
6      dsb.url("jdbc:derby:c:\springclass\db\StudentDB");
7      dsb.driverClassName(
8         "org.apache.derby.jdbc.EmbeddedDriver");
9      dsb.username("SA");
10     dsb.password("SA");
11     return dsb.build();
12   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a JNDI-Based DataSource (XML)

- You can use <jee:jndi-lookup> to create a data source via a JNDI lookup
- If your "client" defines a **resource-reference**, then you should specify **resource-ref="true"** - Spring will prepend *java:comp/env* to the provided JNDI name

```
<jee:jndi-lookup id="studentDataSource"
  jndi-name="studentDS" resource-ref="true">
```


With this technique, the programmer does not need to know connection details, only the JNDI name provided by an administrator.

Spring 1.x provided the JndiObjectFactoryBean, but newer applications should use jee:jndi-lookup.

If your client is a servlet, you can define a resource reference in the servlet's Web application deployment descriptor. This is an extra level of indirection that's considered a best practice in JEE since it insulates clients from changes in the actual JNDI name.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring JNDI (Java Config)

- With Java configuration, you can use the **JndiDataSourceLookup** to create and initialize a JNDI-based DataSource

```
1    @Bean
2    public DataSource studentDataSource()
3    {
4      JndiDataSourceLookup dsLookup =
5          new JndiDataSourceLookup();
6      dsLookup.setResourceRef(true);
7      DataSource dataSource = dsLookup.getDataSource(
8          "StudentDS");
9      return dataSource;
10   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Writing a DAO

- DAO classes normally implement a DAO interface, and use injection to obtain the data source

```
1    @Repository
2    public class StudentDAOJdbcImpl implements StudentDAO
3    {
4        private DataSource studentDataSource;
5        private JdbcTemplate template;
6
7        @Autowired
8        public void setDataSource(DataSource ds)
9        {
10          studentDataSource = ds;
11          template = new JdbcTemplate(ds);
12       }
13   . . .
14   }
```

The @Repository annotation is a special case of the @Component annotation designed for use in the data tier. You have to configure the <context:component-scan> element in the XML to include the DAO's package.

Note how we use autowiring to obtain the data source reference on a "set" method, then create the JDBC template that we can use for queries, etc.

Not shown is the interface, StudentDAO, which defines all of the methods that the DAO provides.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JdbcTemplate Query Methods

- The JdbcTemplate provides numerous ways to query for various types

```
<T> List<T> query(String sql, RowMapper<T> rm, Map<String,?> args)
<T> List<T> query(String sql, RowMapper<T> rm, Object... args)
<T> List<T> query(String sql, RowMapper<T> rm, SqlParameterSource args)
int queryForInt(String sql, Map<String,?> args)
int queryForInt(String sql, Object... args)
int queryForInt(String sql, SqlParameterSource args)
List<Map<String,Object>> queryForList(String sql, Map<String,?> args)
List<Map<String,Object>> queryForList(String sql, Object... args)
List<Map<String,Object>> queryForList(String sql, SqlParameterSource args)
long queryForLong(String sql, Map<String,?> args)
long queryForLong(String sql, Object... args)
long queryForLong(String sql, SqlParameterSource args)
Map<String,Object> queryForMap(String sql, Map<String,?> args)
Map<String,Object> queryForMap(String sql, Object... args)
Map<String,Object> queryForMap(String sql, SqlParameterSource args)
<T> T queryForObject(String sql, Class<T> requiredType, Map<String,?> args)
<T> T queryForObject(String sql, Class<T> requiredType, Object... args)
<T> T queryForObject(String sql, Class<T> requiredType, SqlParameterSource args)
<T> T queryForObject(String sql, RowMapper<T> rm, Map<String,?> args)
<T> T queryForObject(String sql, RowMapper<T> rm, Object... args)
<T> T queryForObject(String sql, RowMapper<T> rm, SqlParameterSource args)
```


SimpleJdbcTemplate has quite a few query methods, and its wrapped JdbcTemplate defines even more (recall that you can retrieve the wrapped JdbcTemplate via getJdbcOperations()).

Many of these query methods use Java 5 generics and varargs so that using them is relatively easy.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Example: Querying Row Count

- A common DAO operation is to return the count of rows in a table

```
1    @Repository
2    public class StudentDAOJdbcImpl implements StudentDAO
3    {
4        // initialized in setDataSource()
5        private JdbcTemplate template;
6
7    . . .
8
9      public int getStudentCount()
10     {
11       return template.queryForObject(
12          "SELECT COUNT(*) FROM student", Integer.class);
13     }
14   }
```

Note how we provide the actual SQL for the query, but we don't have to connect to the database, create any statements or handle exceptions (if we don't want to). Spring JDBC handles all of that for us.

NOTE: In Spring prior to 3.2.2, we used queryForInt(), but that is now deprecated.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Rows to Objects

- Spring defines **row mappers**, which are interfaces you can implement so a template can convert a result-set row to an object

```
1    private class StudentRowMapper implements
2      RowMapper<Student>  {
3      public Student mapRow(ResultSet rs, int rowNum)
4       throws SQLException  {
5          Student s = new Student();
6          s.setGpa(rs.getDouble("gpa"));
7          s.setName(rs.getString("name"));
8          s.setStudentID(rs.getInt("studentID"));
9          return s;
10      }
11   }
```

Several of the query methods shown on the last page accept a row-mapper object - the template calls the row-mapper after a query to convert the result(s) to object(s).

Most programmers implement these row-mapper classes as Java "inner classes" within the DAO.

NOTE: Prior to Spring 3.2.2 or later, we used ParameterizedRowMapper instead of RowMapper, which has the same syntax and is functionally equivalent. ParameterizedRowMapper is now deprecated.

In Java 8 or later, instead of writing a separate row mapper class containing only a single method, you could instead use a lambda expression.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Example: Query All

```
1    public class StudentDAOJdbcImpl
2      implements StudentDAO
3    {
4        // initialized in setDataSource()
5        private JdbcTemplate template;
6    . . .
7        private class StudentRowMapper
8           implements RowMapper<Student>
9        { . . . // shown earlier }
10
11       public Collection<Student> findAllStudents()
12       {
13           List<Student> students = template
14              .query("SELECT * FROM student", mapper);
15           return students;
16       }
17   }
```

Here we are using the query method:

```
<T> List<T> query(String sql, RowMapper<T> rm, Object... args)
```

Note that we are passing zero arguments for the last Object... parameter.

The equivalent Java 8 lambda might look like:

```
List<Student> students = jdbcTemplate.query(
  "SELECT * FROM student", (rs, rowNum) ->
   {
      Student s = new Student();
      s.setGpa(rs.getDouble("gpa"));
      s.setName(rs.getString("name"));
      s.setStudentID(rs.getInt("studentID"));
      return s;
   });
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Example: Find Student By ID

```
1    public class StudentDAOJdbcImpl
2      implements StudentDAO
3    {
4
5    . . .
6
7        public Student findStudentByID(int studentID)
8        {
9            return template.queryForObject(
10             "SELECT * FROM student WHERE studentID=?",
11              mapper, studentID);
12       }
13   }
```


Here we are using the query method:

```
<T> List<T> query(String sql, RowMapper<T> rm, Object... args)
```

Note that we are passing a single integer argument for the last Object... parameter. The template uses that integer to supply the value for the '?' in the SQL. If the SQL had more than one '?', we could simply pass as many arguments to the "query()" as was needed.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JdbcTemplate Insert, Update and Delete

```
int update(String sql, Map<String,?> args)
int update(String sql, Object... args)
int update(String sql, SqlParameterSource args)

int[] batchUpdate(String sql, List<Object[]> batchArgs)
int[] batchUpdate(String sql, List<Object[]> batchArgs, int[] argTypes)
int[] batchUpdate(String sql, Map<String,?>[] batchValues)
int[] batchUpdate(String sql, SqlParameterSource[] batchArgs)
```


Note that the non-batch versions of these return an integer, which is the count of rows affected. The batch versions return an array of integers, which are the count of rows affected for each of the batch operations.

According to the Spring docs:

Most JDBC drivers provide improved performance if you batch multiple calls to the same prepared statement. By grouping updates into batches you limit the number of round trips to the database.

There is an example of batching in the Spring docs at:

http://static.springsource.org/spring/docs/2.5.6/reference/jdbc.html

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Example: Insert

```
1    public class StudentDAOJdbcImpl
2      implements StudentDAO
3    {
4        // initialized in setDataSource()
5        private JdbcTemplate template;
6
7    . . .
8
9        public int insertStudent(Student s)
10       {
11         return template.update(
12           "INSERT INTO student
13           (studentId,name,gpa) VALUES(?,?,?)",
14           s.getStudentID(), s.getName(), s.getGpa());
15       }
16   }
```

Here are using the update method of the form:

```
int update(String sql, Object... args)
```

We supply three values for the Object... varargs parameter that correspond to the three '?' in the SQL. Note that we are expecting the caller to provide the primary key (studentID).

The update() method returns the count of rows affected. In this case, we'd expect the count to be one, since we inserted a single row.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Complete DAO Example

Student.java

StudentDAO.java

StudentDAOJdbcImpl.java

DisplayStudents.java

FindStudent.java

InsertStudent.java

spring.xml

web.xml

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package university;
2
3    public class Student
4    {
5        private int studentID;
6        private String name;
7        private double gpa;
8
9        public double getGpa()
10       {
11           return gpa;
12       }
13
14       public void setGpa(double gpa)
15       {
16           this.gpa = gpa;
17       }
18
19       public String getName()
20       {
21           return name;
22       }
23
24       public void setName(String name)
25       {
26           this.name = name;
27       }
28
29       public int getStudentID()
30       {
31           return studentID;
32       }
33
34       public void setStudentID(int studentID)
35       {
36           this.studentID = studentID;
37       }
38   }
```

```
1    package dao;
2
3    import java.util.Collection;
4
5    import university.Student;
6
7    public interface StudentDAO
8    {
9        public int getStudentCount();
10       public Collection<Student> findAllStudents();
11       public Student findStudentByID(int studentID);
12       public int insertStudent(Student s);
13       public int insertStudentReturnId(Student s);
14   }
```

```
1    package dao;
2
3    import java.sql.ResultSet;
4    import java.sql.SQLException;
5    import java.util.Collection;
6    import java.util.HashMap;
7    import java.util.List;
8    import java.util.Map;
9
10   import javax.sql.DataSource;
11
12   import org.springframework.beans.factory.annotation.Autowired;
13   import org.springframework.jdbc.core.JdbcTemplate;
14   import org.springframework.jdbc.core.RowMapper;
15   import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
16   import org.springframework.stereotype.Repository;
17
18   import university.Student;
19
20   @Repository("studentDAO")
21   public class StudentDAOJdbcImpl implements StudentDAO
22   {
23       private DataSource studentDataSource;
24       private JdbcTemplate template;
25       private StudentRowMapper mapper;
26
27       @Autowired
28       public void setDataSource(DataSource ds)
29       {
30           studentDataSource = ds;
31           template = new JdbcTemplate(ds);
32           mapper = new StudentRowMapper();
33       }
34
35       private class StudentRowMapper implements RowMapper<Student>
36       {
37           public Student mapRow(ResultSet rs, int rowNum) throws SQLException
38           {
39               Student s = new Student();
40               s.setGpa(rs.getDouble("gpa"));
41               s.setName(rs.getString("name"));
42               s.setStudentID(rs.getInt("studentID"));
43               return s;
44           }
45       }
```

```
46
47      public int getStudentCount()
48      {
49          return template.queryForObject("SELECT COUNT(*) FROM student",
50                  Integer.class);
51      }
52
53      public Collection<Student> findAllStudents()
54      {
55          List<Student> students = template.query("SELECT * FROM student",
56                  mapper);
57          return students;
58      }
59
60      public Student findStudentByID(int studentID)
61      {
62          Student s = template.queryForObject(
63                  "SELECT * FROM student WHERE studentID=?",
64                  mapper, studentID);
65          return s;
66      }
67
68      public int insertStudent(Student s)
69      {
70          return template.update(
71                 "INSERT INTO student (name,gpa) VALUES(?,?)",
72                  s.getName(), s.getGpa());
73      }
74
75      // This insertStudent version uses the SimpleJdbcInsert
76      // class to retrieve an autogenerated key
77      public int insertStudentReturnId(Student s)
78      {
79          SimpleJdbcInsert inserter = new SimpleJdbcInsert(studentDataSource);
80          inserter.setTableName("student");
81          inserter.usingGeneratedKeyColumns("StudentID");
82          Map<String, Object> parms = new HashMap<String, Object>();
83          parms.put("name", s.getName());
84          parms.put("gpa", s.getGpa());
85
86          Number id = inserter.executeAndReturnKey(parms);
87          return id.intValue();
88      }
89  }
```

```
1    package servlets;
2
3    import java.io.IOException;
4    import java.io.PrintWriter;
5    import java.util.Collection;
6
7    import javax.servlet.ServletContext;
8    import javax.servlet.ServletException;
9    import javax.servlet.annotation.WebServlet;
10   import javax.servlet.http.HttpServletRequest;
11   import javax.servlet.http.HttpServletResponse;
12
13   import org.springframework.web.context.WebApplicationContext;
14   import org.springframework.web.context.support.WebApplicationContextUtils;
15
16   import university.Student;
17   import dao.StudentDAO;
18
19
20   @SuppressWarnings("serial")
21   @WebServlet("/displayStudents")
22   public class DisplayStudents extends javax.servlet.http.HttpServlet
23   {
24       @Override
25       protected void doGet(HttpServletRequest request, HttpServletResponse response)
26       {
27           response.setContentType("text/html");
28           PrintWriter out = response.getWriter();
29
30           ServletContext servletContext = getServletContext();
31           WebApplicationContext ctx =
32               WebApplicationContextUtils.getRequiredWebApplicationContext(
33                   servletContext);
34
35           try
36           {
37               StudentDAO dao = (StudentDAO)ctx.getBean("studentDAO");
38               out.println("There are " + dao.getStudentCount() + " students");
39
40               Collection<Student> students = dao.findAllStudents();
41               out.println("<table border='1'>");
42               for(Student s : students)
43               {
44                   out.println("<tr>");
45                   out.println("<td>" + s.getStudentID() + "</td>");
```

```
46                  out.println("<td>" + s.getName() + "</td>");
47                  out.println("<td>" + s.getGpa() + "</td>");
48                  out.println("</tr>");
49              }
50              out.println("</table>");
51          }
52          catch (Exception e)
53          {
54              out.println("Something went wrong: " + e);
55          }
56
57  //        Connection con = null;
58  //        try
59  //        {
60  //            Context ctx = new InitialContext();
61  //            DataSource ds =
62  //                (DataSource)ctx.lookup("java:comp/env/studentDS");
63  //            con = ds.getConnection();
64  //            Statement st = con.createStatement();
65  //            ResultSet rs = st.executeQuery("SELECT COUNT(*) FROM student");
66  //            rs.next();
67  //            out.println("There are " + rs.getInt(1) + " students");
68  //        }
69  //        catch (NamingException e)
70  //        {
71  //            e.printStackTrace();
72  //        }
73  //        catch (SQLException e)
74  //        {
75  //            e.printStackTrace();
76  //        }
77  //        finally
78  //        {
79  //            try
80  //            {
81  //               if (con != null) con.close();
82  //            } catch (Exception e1) {}
83  //        }
84
85
86      }
87  }
```

```
1    package servlets;
2
3    import java.io.IOException;
4    import java.io.PrintWriter;
5
6    import javax.servlet.ServletContext;
7    import javax.servlet.ServletException;
8    import javax.servlet.annotation.WebServlet;
9    import javax.servlet.http.HttpServletRequest;
10   import javax.servlet.http.HttpServletResponse;
11
12   import org.springframework.dao.EmptyResultDataAccessException;
13   import org.springframework.web.context.WebApplicationContext;
14   import org.springframework.web.context.support.WebApplicationContextUtils;
15
16   import university.Student;
17   import dao.StudentDAO;
18
19   @SuppressWarnings("serial")
20   @WebServlet("/findStudent")
21   public class FindStudent extends javax.servlet.http.HttpServlet
22   {
23       protected void doGet(HttpServletRequest request,
24               HttpServletResponse response) throws ServletException, IOException
25       {
26           response.setContentType("text/html");
27           PrintWriter out = response.getWriter();
28
29           ServletContext servletContext = getServletContext();
30           WebApplicationContext ctx = WebApplicationContextUtils
31                   .getRequiredWebApplicationContext(servletContext);
32
33           try
34           {
35               int studentID = Integer.parseInt(request.getParameter("studentID"));
36               StudentDAO dao = (StudentDAO) ctx.getBean("studentDAO");
37               Student s = dao.findStudentByID(studentID);
38               if (s != null)
39               {
40                   out.println("<p>Student ID: " + s.getStudentID() + "</p>");
41                   out.println("<p>Name: " + s.getName() + "</p>");
42                   out.println("<p>GPA: " + s.getGpa() + "</p>");
43               }
44           }
45           catch (NumberFormatException e)
```

```
46          {
47              out.println("Invalid number entered. Press 'Back' and try again.");
48          }
49          catch (EmptyResultDataAccessException e)
50          {
51              out.println("Student not found");
52          }
53          catch (Exception e)
54          {
55              out.println("Something went wrong: " + e);
56          }
57      }
58  }
```

```
1    package servlets;
2
3    import java.io.IOException;
4    import java.io.PrintWriter;
5
6    import javax.servlet.ServletContext;
7    import javax.servlet.ServletException;
8    import javax.servlet.annotation.WebServlet;
9    import javax.servlet.http.HttpServletRequest;
10   import javax.servlet.http.HttpServletResponse;
11
12   import org.springframework.web.context.WebApplicationContext;
13   import org.springframework.web.context.support.WebApplicationContextUtils;
14
15   import university.Student;
16   import dao.StudentDAO;
17
18   @SuppressWarnings("serial")
19   @WebServlet("/insertStudent")
20   public class InsertStudent extends javax.servlet.http.HttpServlet
21   {
22       protected void doPost(HttpServletRequest request,
23               HttpServletResponse response) throws ServletException, IOException
24       {
25           response.setContentType("text/html");
26           PrintWriter out = response.getWriter();
27
28           ServletContext servletContext = getServletContext();
29           WebApplicationContext ctx = WebApplicationContextUtils
30                   .getRequiredWebApplicationContext(servletContext);
31
32           String name = request.getParameter("name");
33           String s = request.getParameter("gpa");
34
35           double gpa = Double.parseDouble(s);
36
37           try
38           {
39               StudentDAO dao = (StudentDAO) ctx.getBean("studentDAO");
40
41               Student student = new Student();
42               student.setGpa(gpa);
43               student.setName(name);
44               int rows = dao.insertStudent(student);
45               out.println("Insert successful, " + rows + " inserted.");
```

```
46            }
47            catch (Exception e)
48            {
49                out.println("Something went wrong: " + e);
50            }
51        }
52    }
```

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:aop="http://www.springframework.org/schema/aop"
5       xmlns:context="http://www.springframework.org/schema/context"
6       xmlns:jee="http://www.springframework.org/schema/jee"
7       xmlns:jms="http://www.springframework.org/schema/jms"
8       xmlns:lang="http://www.springframework.org/schema/lang"
9       xmlns:tx="http://www.springframework.org/schema/tx"
10      xmlns:util="http://www.springframework.org/schema/util"
11      xsi:schemaLocation="http://www.springframework.org/schema/aop
12          http://www.springframework.org/schema/aop/spring-aop-4.1.xsd
13          http://www.springframework.org/schema/beans
14          http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
15          http://www.springframework.org/schema/context
16          http://www.springframework.org/schema/context/spring-context-4.1.xsd
17          http://www.springframework.org/schema/jee
18          http://www.springframework.org/schema/jee/spring-jee-4.1.xsd
19          http://www.springframework.org/schema/lang
20          http://www.springframework.org/schema/lang/spring-lang-4.1.xsd
21          http://www.springframework.org/schema/tx
22          http://www.springframework.org/schema/tx/spring-tx-4.1.xsd
23          http://www.springframework.org/schema/util
24          http://www.springframework.org/schema/util/spring-util-4.1.xsd">
25
26      <context:component-scan base-package="dao" />
27
28      <bean id="studentDataSource"
29          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
30          <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
31          <property name="url" value="jdbc:hsqldb:hsql://localhost/" />
32          <property name="username" value="sa" />
33          <property name="password" value="" />
34      </bean>
35
36  </beans>
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.
3      <display-name>SpringJdbcTemplateWeb</display-name>
4        <context-param>
5            <param-name>contextConfigLocation</param-name>
6            <param-value>/WEB-INF/spring.xml</param-value>
7        </context-param>
8        <listener>
9            <display-name>ContextLoaderListener</display-name>
10           <listener-class>
11               org.springframework.web.context.ContextLoaderListener
12           </listener-class>
13       </listener>
14     <welcome-file-list>
15       <welcome-file>index.html</welcome-file>
16       <welcome-file>index.htm</welcome-file>
17       <welcome-file>index.jsp</welcome-file>
18       <welcome-file>default.html</welcome-file>
19       <welcome-file>default.htm</welcome-file>
20       <welcome-file>default.jsp</welcome-file>
21     </welcome-file-list>
22   </web-app>
```

Chapter Summary

In this chapter, you learned:

- The basics of Spring's JDBC support
- How to use SimpleJdbcTemplate and SimpleJdbcDaoSupport

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to AOP

- What is AOP?
- AOP Terminology
- AOP Hello, World

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Aspect-Oriented Programming?

- **Aspect-oriented programming** (AOP) is a software discipline that lets you separate **cross-cutting concerns** from domain logic
- Cross-cutting concerns include:
  - Logging
  - Transactions
  - Security
  - Persistence

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

AOP Terminology

- **Advice** is the code that the aspect performs
- A **joinpoint** is a place during execution where an aspect can kick in
- A **pointcut** describes which join points should actually be advised
- An **aspect** combines advice and pointcuts
- **Weaving** is the act of applying aspects to the target

Advice also includes the timing of the advice, for example is it applied before the target method runs or after?

Joinpoint examples include method calls, constructors, field access and so forth. Note that not all AOP systems support all of these types of joinpoints: for example, Spring AOP only supports method calls.

A join point is a predicate, or filter of all of the possible joinpoints. It could be just a list of method names or perhaps a regular expression.

Weaving is often by using a proxy that wraps the target object. Weaving can occur at different times, such as load-time or runtime.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Support for AOP

- Spring provides first-class support for aspect-oriented programming, either using annotations or XML configuration
- We will cover the annotation approach here, using:
  - **@Aspect** to declare an aspect
  - **@Pointcut** to specify to specify pointcuts
  - **@Before** and **@After** to specify when advice should be applied
- Spring also uses AOP internally to implement other services such as transactions

Spring uses the third-party AspectJ framework for syntax and runtime support of AOP.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Steps for Using Spring AOP

- Define one or more aspect classes using @Aspect
- Define one or more pointcuts using @Pointcut
- Apply the advice using @Before and/or @After
- Configure Spring

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Defining a Pointcut

- Defining a pointcut is a bit odd: you write a method with no code and annotate the method with **@Pointcut**
- The method's name acts as the pointcut name

```
1    @Aspect
2    public class MyAspect
3    {
4        @Pointcut("execution(void myMethod())")
5        public void myPointcut() {}
6
7       . . .
8    }
```


Note that this method is just a placeholder - we will see actual advice-providing method(s) in a moment.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Basic Pointcut Syntax

- A pointcut defines which of all possible joinpoints should have advice and thus acts like a predicate, or filter
- You can specify method options, return types, method names and parameters
- You can use the '*' character as a wildcard

Pointcut	Explanation
`execution(void myMethod())`	Matches exact signature
`execution(String myMethod(String))`	Matches exact signature
`execution(* myMethod(..))`	Matches any myMethod
`execution(String *(..))`	Matches any method that returns String
`execution(public * *(..))`	Matches any public method

Spring only supports pointcuts based on method names. AspectJ itself supports more options.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Applying Advice

- In the aspect class, in addition to defining pointcuts, you also write the methods that actually provide advice and annotate them

```
1    @Aspect
2    public class MyAspect
3    {
4      @Pointcut("execution(void myMethod())")
5      public void myPointcut() {}
6
7      @Before("myPointcut()")
8      public void doAdvice()
9      {
10       System.out.println("My advice, sir: get de-icer.");
11     }
12     . . .
13   }
```

When you apply and specify the advice-providing method, you specify the name of the pointcut. Note that the pointcut name includes the parenthesis.

There are additional annotations, including @Around, @AfterReturning (normally) and @AfterThrowing (an exception). The @Around annotation is especially interesting, because it lets the advice method call a proceed() method to continue running the advised method. If you omit that call, only the advice runs, not the target.

It's possible to specify multiple pointcuts, for example the following would require the intersection of two pointcut predicates:

```
@Before(myAdvice() && yourAdvice())
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Spring AOP

- To enable AOP support, your Spring configuration file must:
  - Define all beans containing aspects
  - Define all beans that are subject to advice
  - Enable **autoproxy**

```
1    <beans . . . >
2
3      <bean class="aspects.MyAspect" />
4      <bean id="myBean" class="domain.MyDomainClass" />
5
6      <aop:aspectj-autoproxy />
7
8    </beans>
```

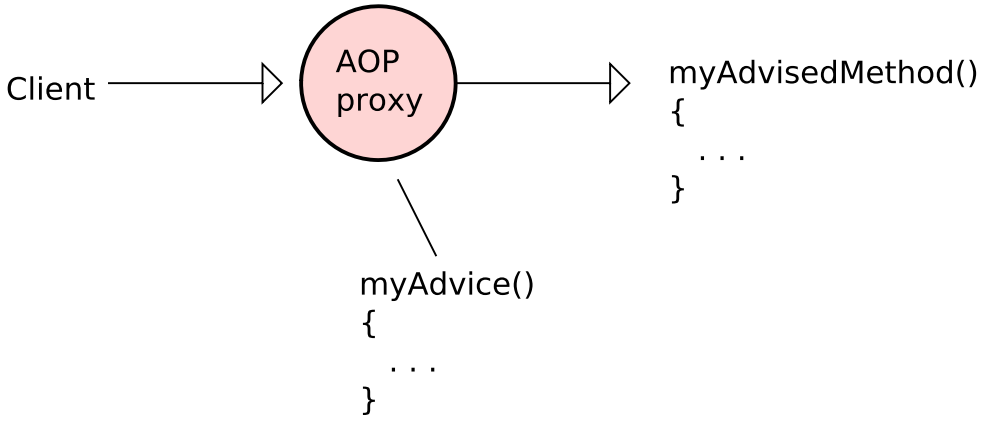
The aspectj-autoproxy element instructs Spring to install a bean post processor that scans for the AOP annotations.

If you are using Java configuration, you can accomplish the same by adding the @EnableAspectJAutoProxy annotation to your configuration class.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring AOP Behind the Scenes

- Spring applies advice using **proxy objects** that intercept method requests
- Due to using proxies, Spring AOP only supports **runtime** weaving on method joinpoints
- Proxying works best if the proxied bean has an interface



If the proxied bean does not have an interface, Spring uses the CGLIB project to create the proxy. That requires the appropriate JAR file in the application's class path. If there is an interface, Spring uses the Java 1.4 dyna-proxy API which requires no other dependencies.

AspectJ also supports "load-time" weaving which allows for more joinpoints besides just method invocations. For example, in AspectJ, you can write advice that "fires" when a field is modified.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Complete Hello, World Example

MyDomainInterface.java

MyDomainClass.java

MyAspect.java

spring.xml

TestAOP.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package domain;
2
3    public interface MyDomainInterface
4    {
5        public void myMethod();
6    }
```

```
1    package domain;
2
3    public class MyDomainClass implements MyDomainInterface
4    {
5        public void myMethod()
6        {
7            System.out.println("Thank you!");
8        }
9    }
```

```
1    package aspects;
2
3    import org.aspectj.lang.annotation.After;
4    import org.aspectj.lang.annotation.Aspect;
5    import org.aspectj.lang.annotation.Before;
6    import org.aspectj.lang.annotation.Pointcut;
7
8    @Aspect
9    public class MyAspect
10   {
11       @Pointcut("execution(void myMethod())")
12       public void myPointcut() {}
13
14       @Before("myPointcut()")
15       public void doAdvice()
16       {
17           System.out.println("My advice, sir: get de-icer.");
18       }
19
20       @After("myPointcut()")
21       public void doMoreAdvice()
22       {
23           System.out.println("You are welcome!");
24       }
25   }
```

```
1    <beans xmlns="http://www.springframework.org/schema/beans"
2        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.sp
3        xmlns:tx="http://www.springframework.org/schema/tx"
4        xsi:schemaLocation="http://www.springframework.org/schema/beans
5          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
6          http://www.springframework.org/schema/aop
7          http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
8          http://www.springframework.org/schema/tx
9          http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
10
11       <bean class="aspects.MyAspect" />
12       <bean id="myBean" class="domain.MyDomainClass" />
13
14       <aop:aspectj-autoproxy />
15
16   </beans>
```

```
1    package main;
2
3    import org.springframework.context.support.AbstractApplicationContext;
4    import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6    import domain.MyDomainInterface;
7
8    public class TestAOP
9    {
10       public static void main(String[] args)
11       {
12           String[] ctxPaths = {"spring.xml"};
13           AbstractApplicationContext ctx = new ClassPathXmlApplicationContext(
14                   ctxPaths);
15
16           MyDomainInterface mdi = (MyDomainInterface)ctx.getBean("myBean");
17           mdi.myMethod();
18
19           ctx.close();
20       }
21   }
```

Chapter Summary

In this chapter, you learned:

- About fundamental AOP concepts
- How to write a simple Spring AOP application

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transactions in Spring

- What is a Transaction?
- Transaction Scope and Propagation

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Transactions

- Transactions let programs define units of work consisting of steps that must all complete or none complete
- If you use transactions properly, your programs will be more robust and reliable

The notion of transactions has been around for quite awhile and is well known in enterprise systems. Many databases and other back-end resource managers implement transactions and can be "enlisted" by an Spring container.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Program with No Transactions

- This program does not use transactions and is subject to integrity problems if something goes wrong

```
1    // transfer $100 from savings to checking
2
3    long balance = savings.getBalance();
4    balance -= 100;
5    savings.setBalance ( balance );
6
7    balance = checking.getBalance();
8    balance += 100;
9    checking.setBalance()
```


This is the classic program that illustrates the need for transactions. Suppose the system crashes in line 6 -- will the customer be happy that the program "lost" $100?

Note that in the interest of illustrating the point, we are doing the balance transfer using primitive "get" and "set" methods. A real program would probably not work exactly like this, but the concept is still relevant.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Program with Transactions

- By adding calls to a *transaction service*, the system will not be corrupted if something goes wrong
- This is an example of program-demarcated transactions

```
1    // transfer $100 from savings to checking
2
3    ts.begin();
4    long balance = savings.getBalance();
5    balance -= 100;
6    savings.setBalance ( balance );
7
8    balance = checking.getBalance();
9    balance += 100;
10   checking.setBalance()
11   ts.commit();
```

This is the same program, now with calls to some "transaction service", which manages the transaction. Now if the program crashes in line 7, the transaction service can rollback the system so that no money is lost.

Note that this is the traditional transaction model -- EJBs support this, but also provide a technique in which the container manages transactions for you.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

ACID Transactions

Atomic Transactions should complete entirely or not at all

Consistent The system state after the transaction should meet "invariants", for example `balance > 0` or `customerID is unique`

Isolated Other transactions cannot "see" intermediate results. Implies locking

Durable Updates to data should survive a system crash, perhaps use transaction logs to recover after a restart

The literature on transactions defines these four properties as attributes of a good transaction, sometimes referred to by the acronym ACID</i>

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

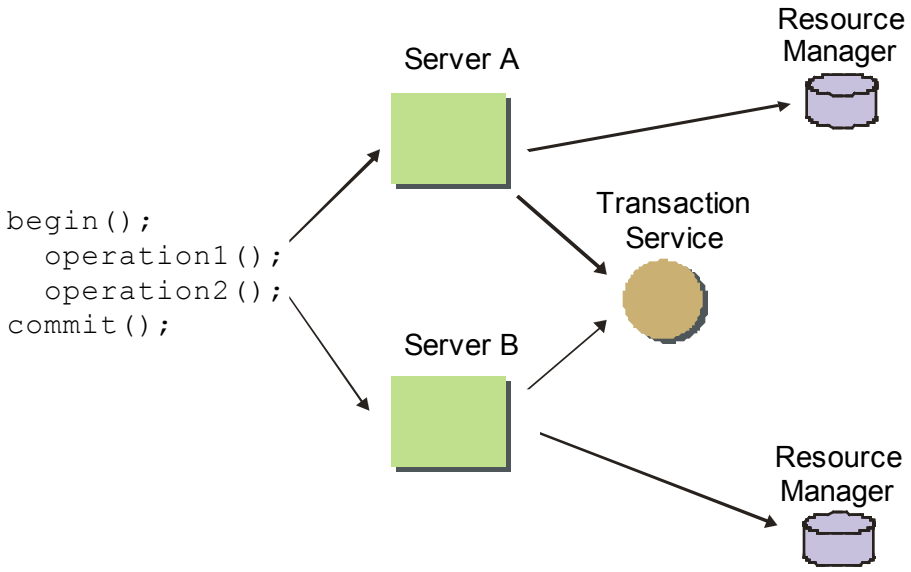
Global and Local Transactions

- **Local** transactions are those associated with a single resource manager, e.g. a relational database
- **Global** transactions are under the control of a **transaction manager**, typically provided an application server
- Global transactions allow for a distributed, two-phase commit across multiple transaction managers
- Unfortunately, local and global transactions have different programing models in Java, thus making it difficult to switch

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Distributed Transactions

- Some transaction managers allow for transactions to be work across different resource managers that adhere to industry standards for transactions



Distributed transactions are more complex than simple transactions and typically require a separate transaction manager that coordinates the individual resource managers' transaction services.

To prevent data corruption, distributed transactions require a process referred to as a two-phase commit</i>. In a two-phase commit, the transaction manager coordinates commits to ensure that all resource managers can successfully commit before committing the transaction. If they can't all commit, the transaction manager rolls back the transaction.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring and Transactions

- Spring provides the following support for transactions:
  - **Transaction managers** that abstract the underlying transaction architecture (i.e. local vs global)
  - An exception hierarchy based on **unchecked** exceptions
  - Either **programmatic** or **declarative** transaction demarcation

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Transaction Managers

- Spring provides **transaction managers** that you can configure via dependency injection, thus insulating applications from the actual transaction implementation and programing model
- The Spring transaction managers delegate to the underlying resource manager
- Spring provides several transaction managers, including:
  - DataSourceTransactionManager (basic JDBC)
  - HibernateTransactionManager
  - JpaTransactionManager
  - JtaTransactionManager
  - WebLogicJtaTransactionManger
  - WebSphereUowTransactionManager

Since you configure the transaction manager in configuration, it's easy to switch between them without changing your application code.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a DataSource Transaction Manager (XML)

- This is a local transaction manager that works with a database

```
1    <jee:jndi-lookup id="acctDataSource"
2      jndi-name="accountDS"
3      resource-ref="true" />
4
5    <bean id="transactionManager"
6      class=
7    "org.springframework.jdbc.datasource.DataSourceTransactionManager">
8      <property name="dataSource" ref="acctDataSource" />
9    </bean>
```


You can use this transaction manager in applications where all of the transactional data resides in a single database -- it doesn't support distributed transactions. This transaction manager basically just delegates to the database manager's own JDBC-based transaction service.

Note that you configure the transaction manager with the data source it needs to begin, commit and rollback transactions.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a DataSource Transaction Manager (Java Config)

```
1    @Configuration
2    @EnableTransactionManagement
3    public class MyConfig
4    {
5      @Bean
6      public DataSource acctDataSource()
7      {
8        JndiDataSourceLookup dsLookup = new JndiDataSourceLookup();
9        dsLookup.setResourceRef(true);
10       return  dsLookup.getDataSource("AccountDS");
11     }
12
13     @Bean
14     public DataSourceTransactionManager transactionManager()
15     {
16       return new DataSourceTransactionManager(acctDataSource());
17     }
18   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a JTA Transaction Manager (XML)

- This transaction manager uses a application server's JTA service and thus typically supports global transactions (e.g. two-phase commit)
- This transaction manager can **enlist** multiple underlying resource managers (e.g. DataSource, Hibernate or JPA transaction managers)

```
<bean id="transactionManager" class=
"org.springframework.transaction.jta.JtaTransactionManager" />
```


This only works if the Spring application itself is deployed onto a JEE application server. The JTA transaction manager looks up the app server's JTA reference using a well known JNDI name, and then delegates to it. Since JTA can enlist resource managers, no other configuration is required.

Not all application servers support distributed transactions, but most do.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

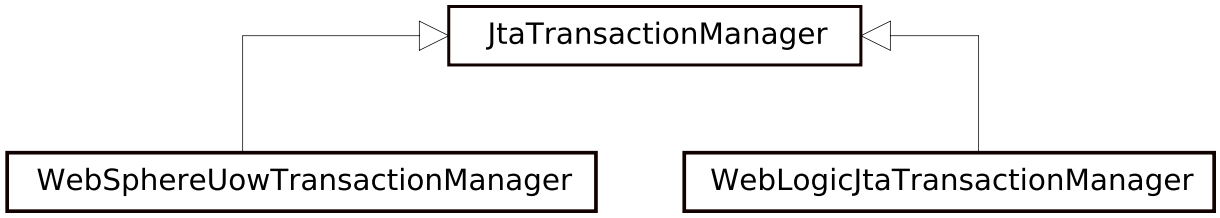
Configuring a JTA Transaction Manager (Java Config)

```
1    @Configuration
2    @EnableTransactionManagement
3    public class MyConfig
4    {
5      . . .
6
7      @Bean
8      public JtaTransactionManager transactionManager()
9      {
10       return new JtaTransactionManager();
11     }
12   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a Vendor-Specific Transaction Manager

- If your Spring application is running on BEA (Oracle) WebLogic or IBM WebSphere, Spring provides specialized transaction managers tailored for those application servers
- The tailored transaction managers support the full range of Spring transactional capability including transaction suspension



If you are using WebSphere, be sure to install the latest fixpacks.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Demarcation and Management

- Applications that use transactions must define where a transaction should start and where it should commit or rollback
- We refer to that process as **transaction demarcation**
- Spring lets you demarcate and configure transactions via configuration (declarative) or in the code itself (programmatic)
- This chapter covers only **declarative** transaction management

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Declarative Transaction Management

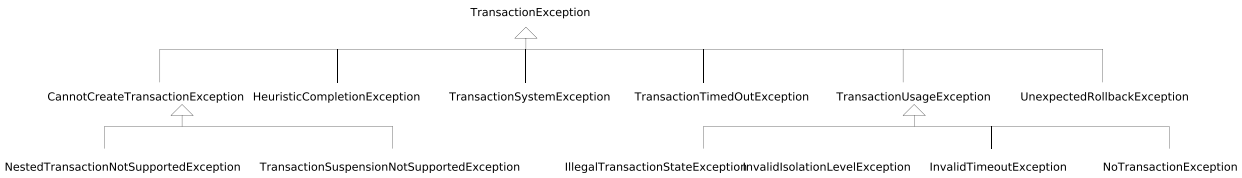
- According to the Spring docs, most Spring applications use declarative transactions since it affects code the least
- Spring lets you configure transactions either via XML configuration or via annotations in the code
- Behind the scenes, Spring uses **aspect oriented programming (AOP) proxies** to implement declarative transactions

AOP works behind the scenes to install "proxies" in front of transactional objects – the AOP proxies automatically begin and commit transactions so that your application code doesn't need to.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Transaction Exception Handling

- Since transaction exceptions are generally not recoverable, Spring provides **unchecked** exceptions that you can catch at a global level in your program and report the problem
- This saves you from writing messy try-catch blocks when using programmatic transaction management

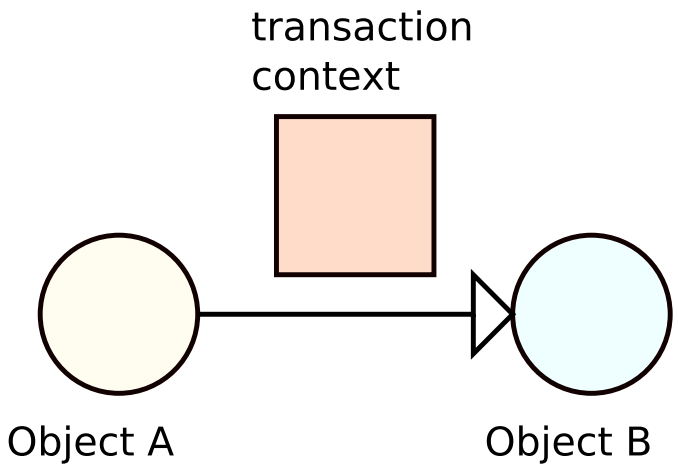


Using unchecked exceptions for unrecoverable errors is a general philosphy in Spring. It makes for more readable, maintainable code that's not cluttered with a lot of try-catch blocks.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Propagation

- Transaction **propagation** governs on how transactions flow from object to object
- Spring provides several **propagation attributes**, based on a similar notion in Enterprise JavaBeans



Objects typically call methods in other objects. The question is: when that happens, does the called method run in the same transaction as the caller? Spring provides "propagation attributes" so you can configure how called methods participate in transactions.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Propagation Attributes

Required If there is no existing transaction, start one. If transaction already exists, use existing

RequiresNew If there is no existing transaction, start one. If transaction exists, suspend it and start new

Mandatory If there is no existing transaction, throw exception. If transaction exists, use it

Supports If there is no existing transaction, run method without transaction. If transaction exists, use it

NotSupported If there is no existing transaction, run method without a transaction. If transaction exists, suspend existing transaction then run method with no transaction

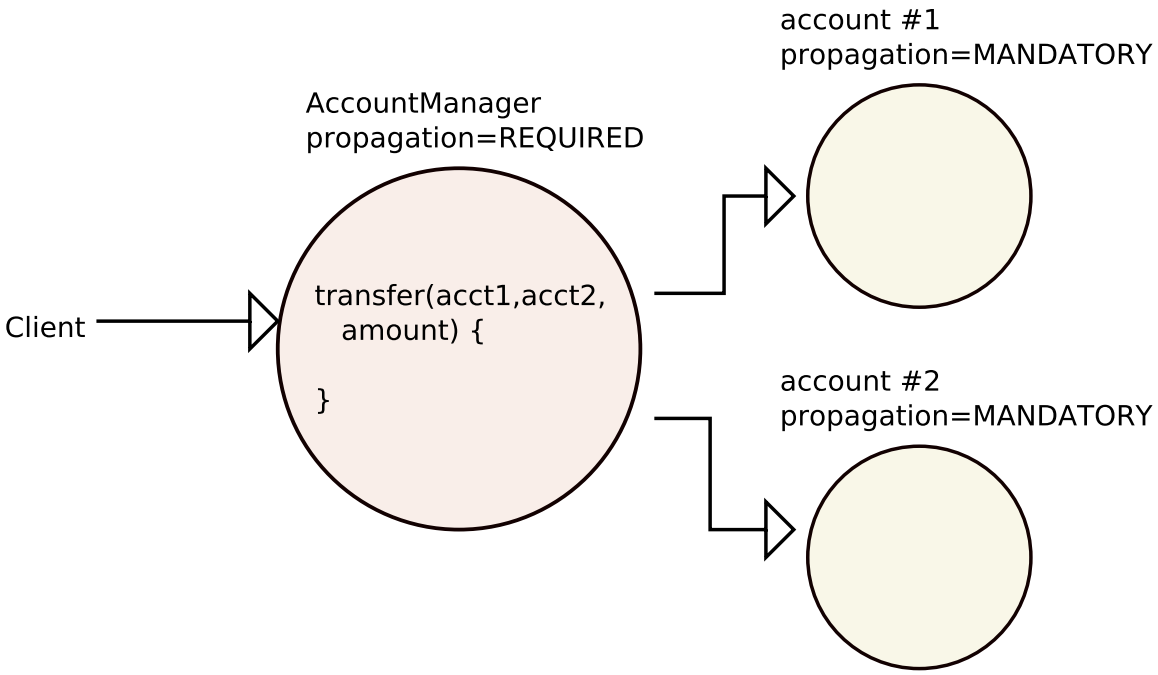
Never If there is no existing transaction, run method without a transaction. If transaction exists, throw RemoteException

---

Spring also defines a "Nested" transactional propagation attribute, but it's not well supported by resource managers, so it's not often used.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Propagation Example

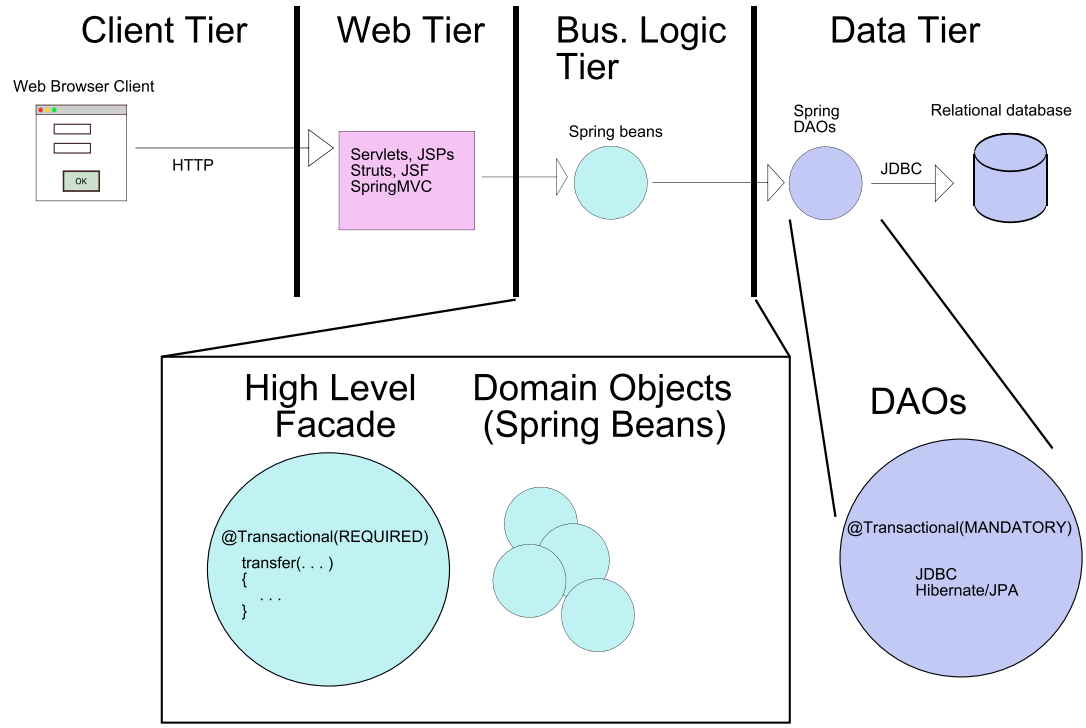


In this example, the "front" bean, AccountManager's transfer() method is configured with the REQUIRED attribute, so when a client calls it, the Spring transaction manager starts a new transaction. Then when the transfer method calls methods in the Account beans that are configured with MANDATORY, Spring enlists those objects in the running transaction.

The transaction commits when the transfer() method returns or rolls back automatically if an unchecked exception occurs.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Typical Architecture



When the Web tier calls methods in the facade, since it's annotated as REQUIRED, Spring starts a transaction and commits it when the method returns.

The facade delegates to Spring domain beans and DAOs which are annotated with MANDATORY, thus propogating the transaction.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Exceptions and Rolling Back

- Spring automatically rolls back a transaction if an **unchecked** exception occurs, but not for **checked** exceptions
- You can change this behavior on a method-by-method basis during configuration

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Timeouts

- Most resource managers define a **timeout** after which a running transaction is automatically aborted (rolled back)
- Spring lets you configure this timeout value
- The default is whatever default the resource manager has, or zero (no timeout) for resource managers that don't support timeouts

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Isolation

- Completely isolated transactions are robust but complete isolation can impact performance
- Spring lets you set an **isolation level** that relaxes isolation at the risk of allowing transactions to "see" intermediate results of other uncommitted transactions

The problems that can occur are:

```
Dirty reads
Phantom reads
Non-repeatable reads
```

The Wikipedia has a nice discussion of these problems at:

```
http://en.wikipedia.org/wiki/Transaction_isolation
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Isolation Levels

- The TransactionDefinition interface defines several constants for isolation:

- **ISOLATION_DEFAULT** -- Use resource manager default
- **ISOLATION_READ_COMMITED** -- *Phantom Read* and *Non-repeatable Read* inconsistencies may occur
- **ISOLATION_READ_UNCOMMITED** -- *Dirty Read*, *Phantom Read* and *Non-repeatable Read* inconsistencies may occur
- **ISOLATION_REPEATABLE_READ** -- *Phantom Read* inconsistencies may occur
- **ISOLATION_SERIALIZABLE** -- Transaction is completed isolated

This interface is in the org.springframework.transaction package.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Read Only Transactions

- In some cases, a resource manager can optimize access if the transaction will not modify data
- Spring lets you configure transactions as **read-only** as a hint to the resource manager

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Specifying Transaction Attributes

- Spring lets you specify transaction propagation attributes either via:
  - Annotations (Spring 2.x or later)
  - XML configuration
- It's generally simpler to use annotations, so that's what we cover in this chapter

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Annotations

- Instead of, or in addition to using XML configuration, Spring 2.x supports configuring transactions by annotations in the code itself
- You must use the **annotation-driven** element in the configuration file for transactional annotations to work

```
1    Spring configuration file:
2    -------------------------
3    <beans ...
4      xmlns:tx="http://www.springframework.org/schema/tx"
5       ...>
6
7      <tx:annotation-driven
8        transaction-manager="transactionManager" />
9
10   </beans>
10 - 28
```

The "annotation-driven" element instructs Spring to process annotations in your source files. If you forget this line in the configuration file, your annotations will have no effect.

If you are using Java configuration, you can accomplish the same goal using the EnableTransactionManagement annotation on your configuration class.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The @Transactional Annotation

```
1    @Transactional(
2      propagation=Propagation.MANDATORY)
3    public class MyBean
4    {
5      @Transactional(readOnly=true,
6        rollbackFor=MyException.class)
7      public void myMethod() throws MyException
8      {}
9    }
```

Element	Type	Default
propagation	enum on Propagation type	PROPAGATION_REQUIRED
isolation	enum on Isolation type	ISOLATION_DEFAULT
readOnly	boolean	false
timeout	int (in seconds)	resource manager default
rollbackFor	array of Class objects	none
noRollbackFor	array of Class objects	none

@Transactional is the workhorse annotation for transactions in Spring. You can place on a class definition, in which case it applies to all of the public methods in the class, or individually on public methods to configure each method separately. Note that you can annotation on both a class and a method - if there's a conflict, the method takes precedence.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Complete Declarative Transaction Example

Account.java

AccountManager.java

AccountManagerImpl.java

AccountDAO.java

JdbcAccountDAO.java

NoMoneyException.java

TestTransactions.java

web.xml

AccountConfig.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package acct;
2
3    public class Account
4    {
5        private Long accountNumber;
6        private double balance;
7        private String holderName;
8
9        public Long getAccountNumber()
10       {
11           return accountNumber;
12       }
13       public void setAccountNumber(Long accountNumber)
14       {
15           this.accountNumber = accountNumber;
16       }
17       public double getBalance()
18       {
19           return balance;
20       }
21       public void setBalance(double balance)
22       {
23           this.balance = balance;
24       }
25       public String getHolderName()
26       {
27           return holderName;
28       }
29       public void setHolderName(String holderName)
30       {
31           this.holderName = holderName;
32       }
33
34       @Override
35       public String toString()
36       {
37           return "Acct#: " + accountNumber + ", holder: " + holderName + ", balance:
38       }
39   }
```

```
1    package acct;
2
3    public interface AccountManager
4    {
5        public void transfer(int acct1, int acct2, double amount)
6            throws NoMoneyException;
7        public String getAccountInfo(int accountNumber);
8    }
```

```
1    package acct;
2
3    import org.springframework.beans.factory.annotation.Autowired;
4    import org.springframework.stereotype.Component;
5    import org.springframework.transaction.annotation.Propagation;
6    import org.springframework.transaction.annotation.Transactional;
7
8    @Component("accountMgr")
9    public class AccountManagerImpl
10       implements AccountManager
11   {
12       private AccountDAO accountDAO;
13
14       @Autowired
15       public void setAccountDAO(AccountDAO accountDAO)
16       {
17           this.accountDAO = accountDAO;
18       }
19
20       @Transactional(rollbackFor=NoMoneyException.class, timeout=10,
21               propagation=Propagation.REQUIRED)
22       public void transfer(int acct1, int acct2, double amount)
23           throws NoMoneyException
24       {
25           Account a1 = accountDAO.getAccountByID(acct1);
26           Account a2 = accountDAO.getAccountByID(acct2);
27
28           double a1Balance = a1.getBalance();
29           if (a1Balance - amount < 0)
30               throw new NoMoneyException();
31
32           double a2Balance = a2.getBalance();
33           a1Balance -= amount;
34           a2Balance += amount;
35           a1.setBalance(a1Balance);
36           a2.setBalance(a2Balance);
37           accountDAO.updateAccount(a1);
38           accountDAO.updateAccount(a2);
39       }
40
41       @Transactional(readOnly=true)
42       public String getAccountInfo(int accountNumber)
43       {
44           Account a = accountDAO.getAccountByID(accountNumber);
45           return a.toString();
```

```
46          }
47      }
```

```
1    package acct;
2
3    public interface AccountDAO
4    {
5        public Account getAccountByID(int accountNumber);
6        public void updateAccount(Account a);
7    }
```

```
1    package acct;
2
3    import java.sql.ResultSet;
4    import java.sql.SQLException;
5
6    import javax.sql.DataSource;
7
8    import org.springframework.beans.factory.annotation.Autowired;
9    import org.springframework.jdbc.core.JdbcTemplate;
10   import org.springframework.jdbc.core.RowMapper;
11   import org.springframework.stereotype.Repository;
12   import org.springframework.transaction.annotation.Propagation;
13   import org.springframework.transaction.annotation.Transactional;
14
15   @Repository("accountDao")
16   public class JdbcAccountDAO implements AccountDAO
17   {
18       private static class AccountRowMapper implements
19               RowMapper<Account>
20       {
21           public Account mapRow(ResultSet rs, int index) throws SQLException
22           {
23               Account acct = new Account();
24               acct.setAccountNumber(rs.getLong("ACCT_NUM"));
25               acct.setBalance(rs.getDouble("BALANCE"));
26               acct.setHolderName(rs.getString("HOLDERNAME"));
27               return acct;
28           }
29       }
30
31       private JdbcTemplate template;
32       private AccountRowMapper mapper;
33
34       @Autowired
35       public void setDataSource(DataSource ds)
36       {
37           template = new JdbcTemplate(ds);
38           mapper = new AccountRowMapper();
39       }
40
41       @Transactional(propagation=Propagation.MANDATORY)
42       public void updateAccount(Account a)
43       {
44           String SQL = "UPDATE account SET balance=?,holdername=? WHERE acct_num=?";
45           template.update(SQL, a.getBalance(), a.getHolderName(),
```

```
46                      a.getAccountNumber());
47          }
48
49          @Transactional(propagation=Propagation.MANDATORY)
50          public Account getAccountByID(int accountNumber)
51          {
52              String SQL = "SELECT * FROM account WHERE ACCT_NUM=?";
53              return template.queryForObject(SQL,
54                      mapper, accountNumber);
55          }
56      }
```

```
1    package acct;
2
3    @SuppressWarnings("serial")
4    public class NoMoneyException extends Exception
5    {
6        public NoMoneyException()
7        {
8            super("Insufficient funds for transfer");
9        }
10   }
```

```
1    package servlets;
2
3    import java.io.IOException;
4    import java.io.PrintWriter;
5
6    import javax.servlet.ServletContext;
7    import javax.servlet.ServletException;
8    import javax.servlet.annotation.WebServlet;
9    import javax.servlet.http.HttpServletRequest;
10   import javax.servlet.http.HttpServletResponse;
11
12   import org.springframework.web.context.WebApplicationContext;
13   import org.springframework.web.context.support.WebApplicationContextUtils;
14
15   import acct.AccountManager;
16
17   @SuppressWarnings("serial")
18   @WebServlet("/TestTransactions")
19   public class TestTransactions extends javax.servlet.http.HttpServlet
20   {
21       protected void doGet(HttpServletRequest request,
22               HttpServletResponse response) throws ServletException, IOException
23       {
24           response.setContentType("text/html");
25           PrintWriter out = response.getWriter();
26
27           ServletContext servletContext = getServletContext();
28           WebApplicationContext ctx = WebApplicationContextUtils
29                   .getRequiredWebApplicationContext(servletContext);
30           try
31           {
32               AccountManager mgr = (AccountManager)ctx.getBean("accountMgr");
33
34               out.println("Before transfer: <br>");
35               out.println(mgr.getAccountInfo(1) + "<br>");
36               out.println(mgr.getAccountInfo(2) + "<br>");
37
38               mgr.transfer(1, 2, 1);
39
40               out.println("After transfer: <br>");
41               out.println(mgr.getAccountInfo(1) + "<br>");
42               out.println(mgr.getAccountInfo(2) + "<br>");
43           }
44           catch (Exception e)
45           {
```

```
46                out.println("Something went wrong: " + e);
47            }
48        }
49    }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns
3      <display-name>TestTransactionsTomcatWeb</display-name>
4      <resource-ref>
5        <res-ref-name>jdbc/account</res-ref-name>
6        <res-type>javax.sql.DataSource</res-type>
7        <res-auth>Container</res-auth>
8        <res-sharing-scope>Shareable</res-sharing-scope>
9      </resource-ref>
10     <listener>
11       <listener-class>org.springframework.web.context.ContextLoaderListener</listene
12     </listener>
13     <context-param>
14       <param-name>contextClass</param-name>
15       <param-value>org.springframework.web.context.support.AnnotationConfigWebApplic
16     </context-param>
17     <context-param>
18       <param-name>contextConfigLocation</param-name>
19       <param-value>config.AccountConfig</param-value>
20     </context-param>
21     <welcome-file-list>
22       <welcome-file>index.html</welcome-file>
23       <welcome-file>index.htm</welcome-file>
24       <welcome-file>index.jsp</welcome-file>
25       <welcome-file>default.html</welcome-file>
26       <welcome-file>default.htm</welcome-file>
27       <welcome-file>default.jsp</welcome-file>
28     </welcome-file-list>
29   </web-app>
```

```
1    package config;
2
3    import javax.sql.DataSource;
4
5    import org.springframework.context.annotation.Bean;
6    import org.springframework.context.annotation.ComponentScan;
7    import org.springframework.context.annotation.Configuration;
8    import org.springframework.jdbc.datasource.DataSourceTransactionManager;
9    import org.springframework.jdbc.datasource.lookup.JndiDataSourceLookup;
10   import org.springframework.transaction.annotation.EnableTransactionManagement;
11
12   @Configuration
13   @ComponentScan("acct")
14   @EnableTransactionManagement
15   public class AccountConfig
16   {
17       @Bean
18       public DataSource acctDataSource()
19       {
20         JndiDataSourceLookup dsLookup =
21             new JndiDataSourceLookup();
22         dsLookup.setResourceRef(true);
23         DataSource dataSource = dsLookup.getDataSource(
24             "jdbc/account");
25         return dataSource;
26       }
27
28         @Bean
29         public DataSourceTransactionManager transactionManager()
30         {
31           return new DataSourceTransactionManager(acctDataSource());
32         }
33   }
```

Chapter Summary

In this chapter, you learned:

- The basics of how transactions work
- About Spring transaction managers
- How to configure and control transactions in Spring

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Testing Spring Applications

- Unit Testing Spring Beans
- Integration For Spring Beans
- Databases and Testing

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Testing Issues

- Which testing framework: JUnit, TestNG? Which version?
- Test using Spring alone or Spring and Hibernate/JPA?
- What versions of Spring and Hibernate/JPA are you using?
- Use a dedicated framework: Unitils?
- Test against a real database or in-memory substitute?
- Use **mock** objects for collaborators?

Unitils is an interesting framework that ties together a unit-testing framework such as JUnit, mocking frameworks like EasyMock, database testing with DBUnit and Spring and Hibernate. You can read more at:

`http://unitils.sourceforge.net/spring_article.html`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Unit Testing Spring Beans

- Since Spring beans are POJOs, they should be easy to unit test outside of the container
- You can easily use JUnit or TestNG to test a bean's basic logic
- If you wish, you can use a mocking framework such as JMock to test how the bean works with collaborators

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Example: Simple Unit Test with JUnit4

- Here we show using JUnit4 to test a simple POJO outside of the Spring container

SolveQuadratic.java

SolveQuadraticImpl.java

TestQuadratic.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package equations;
2
3    public interface SolveQuadratic
4    {
5        public double getRoot1(double a, double b, double c);
6        public double getRoot2(double a, double b, double c);
7    }
```

```
1    package equations;
2
3    public class SolveQuadraticImpl implements SolveQuadratic
4    {
5
6        public double getRoot1(double a, double b, double c)
7        {
8            return (-b + Math.sqrt(b * b - 4 * a * c)) / 2 * a;
9        }
10
11       public double getRoot2(double a, double b, double c)
12       {
13           return (-b - Math.sqrt(b * b - 4 * a * c)) / 2 * a;
14       }
15   }
```

```
1    package equations;
2
3    import static org.junit.Assert.*;
4
5    import org.junit.Test;
6
7    public class TestQuadratic
8    {
9        @Test
10       public void testGetRoot1()
11       {
12           SolveQuadratic sq = new SolveQuadraticImpl();
13           assertEquals("First root should be 1.0",
14                   1.0, sq.getRoot1(1, 3, -4));
15       }
16
17       @Test
18       public void testGetRoot2()
19       {
20           SolveQuadratic sq = new SolveQuadraticImpl();
21           assertEquals("Second root should be -4.0",
22                   -4.0, sq.getRoot2(1, 3, -4));
23       }
24   }
```

Integration Testing Spring Beans

- In an **integration** test, you want to test how a given Spring bean interacts with other Spring beans
- Spring provides superclasses that help you manage the application context in your tests:
  - AbstractJUnit38SpringContextTests
  - AbstractJUnit4SpringContextTests
  - AbstractTestNGSpringContextTests

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Integration Test Example

- Here we show using JUnit4 to test a Spring bean that references other beans, retrieving the initial reference from a Spring application context

Student.java

StudentManager.java

SimpleStudentManager.java

AthleticDepartment.java

TestAthleticDepartment.java

university.xml

This example uses XML configuration, but if you're using Java configuration, in TestAthleticDepartment.java, the annotation would look like:

```
@ContextConfiguration(classes = MyConfig.class)
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package university;
2
3    public class Student
4    {
5        private int id;
6        private String name;
7        private double gpa;
8
9        public Student(int id, String name, double gpa)
10       {
11           super();
12           this.id = id;
13           this.name = name;
14           this.gpa = gpa;
15       }
16
17       public int getId()
18       {
19           return id;
20       }
21
22       public void setId(int id)
23       {
24           this.id = id;
25       }
26
27       public String getName()
28       {
29           return name;
30       }
31
32       public void setName(String name)
33       {
34           this.name = name;
35       }
36
37       public double getGpa()
38       {
39           return gpa;
40       }
41
42       public void setGpa(double gpa)
43       {
44           this.gpa = gpa;
45       }
```

```
46    }
```

```
1    package university;
2
3    public interface StudentManager
4    {
5        public Student findStudent(int id);
6    }
```

```
1    package university;
2
3    import java.util.ArrayList;
4
5    public class SimpleStudentManager implements StudentManager
6    {
7        private static ArrayList<Student> students =
8            new ArrayList<Student>();
9
10       public Student findStudent(int id)
11       {
12           Student found = null;
13
14           for(Student s : students)
15           {
16               if (s.getId() == id)
17               {
18                   found = s;
19                   break;
20               }
21           }
22
23           return found;
24       }
25
26       private static void addStudent(int id, String name, double gpa)
27       {
28           students.add(new Student(id, name, gpa));
29       }
30
31       static
32       {
33           addStudent(13, "Paul Westerberg", 3.44);
34           addStudent(44, "Joan Jett", 3.65);
35           addStudent(193, "David Byrne", 3.90);
36           addStudent(483, "Jonny Rotten", 1.45);
37       }
38
39
40   }
```

```
1    package university;
2
3
4    public class AthleticDepartment
5    {
6        private StudentManager studentManager;
7
8        public void setStudentManager(StudentManager studentManager)
9        {
10           this.studentManager = studentManager;
11       }
12
13       public boolean isEligible(int id)
14       {
15           Student s = studentManager.findStudent(id);
16           if (s.getGpa() > 3.00)
17               return true;
18           else
19               return false;
20       }
21   }
```

```
1    package university;
2
3    import static org.junit.Assert.assertFalse;
4    import static org.junit.Assert.assertTrue;
5
6    import org.junit.Test;
7    import org.springframework.beans.factory.annotation.Autowired;
8    import org.springframework.test.context.ContextConfiguration;
9    import org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;
10
11   @ContextConfiguration(locations = "/university.xml")
12   public class TestAthleticDepartment
13     extends AbstractJUnit4SpringContextTests
14   {
15       @Test
16       public void testPaulEligibility()
17       {
18           AthleticDepartment dept =
19               (AthleticDepartment) applicationContext
20                   .getBean("athleticDepartment");
21           assertTrue("Paul should be eligible",
22                   dept.isEligible(13));
23       }
24
25       @Test
26       public void testJonnyEligibility()
27       {
28           AthleticDepartment dept =
29               (AthleticDepartment) applicationContext
30                   .getBean("athleticDepartment");
31           assertFalse("Jonny Rotten should not be eligible",
32                   dept.isEligible(483));
33       }
34   }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xsi:schemaLocation="http://www.springframework.org/schema/beans
5            http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6
7        <bean id="cowpieTechU"
8          class="university.SimpleStudentManager"/>
9        <bean id="athleticDepartment"
10         class="university.AthleticDepartment">
11         <property name="studentManager" ref="cowpieTechU"/>
12       </bean>
13   </beans>
```

Injecting Spring Beans

- Instead of requiring the test method(s) to use the Spring application context to look up beans, you can use dependency injection and the **@Autowired** annotation

TestAthleticDepartmentWiring.java

This example uses the same Spring object model as the last example.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package university;
2
3    import static org.junit.Assert.assertFalse;
4    import static org.junit.Assert.assertTrue;
5
6    import org.junit.Test;
7    import org.springframework.beans.factory.annotation.Autowired;
8    import org.springframework.test.context.ContextConfiguration;
9    import org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;
10
11   @ContextConfiguration(locations = "/university.xml")
12   public class TestAthleticDepartmentWiring
13      extends AbstractJUnit4SpringContextTests
14   {
15       @Autowired
16       private AthleticDepartment dept;
17
18       @Test
19       public void testPaulEligibility()
20       {
21           // AthleticDepartment dept =
22           // (AthleticDepartment)applicationContext.getBean(
23           //    "athleticDepartment");
24           assertTrue("Paul should be eligible",
25                   dept.isEligible(13));
26       }
27
28       @Test
29       public void testJonnyEligibility()
30       {
31           // AthleticDepartment dept =
32           // (AthleticDepartment)applicationContext.getBean(
33           //   "athleticDepartment");
34           assertFalse("Jonny Rotten should not be eligible",
35                   dept.isEligible(483));
36       }
37   }
```

Using Tests with Transactions and Database

- Spring provides abstract classes that you can subclass that can automatically rollback a transaction so each test runs independently:
  - AbstractTransactionalJUnit38SpringContextTests
  - AbstractTransactionalJUnit4SpringContextTests
  - AbstractTransactionalTestNGSpringContextTests
- If you don't want your test class to extend one of these classes, you can use the **@RunWith(SpringJUnit4ClassRunner.class**) annotation instead

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transaction Test Example

- Here we show doing a unit test that inserts a row during the test
- However, since we used the SpringJUnit4ClassRunner. class annotation, the transaction will be rolled back and no new row will actually be inserted

StudentDAO.java

StudentDAOHibernateImpl3.java

TestStudentDAO.java

spring.xml

Student.hbm.xml

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package dao;
2
3    import java.util.Collection;
4
5    import university.Student;
6
7    public interface StudentDAO
8    {
9        public int getStudentCount();
10       public Collection<Student> findAllStudents();
11       public void insertStudent(Student s);
12   }
```

```
1    package dao;
2
3    import java.util.Collection;
4
5    import org.hibernate.Query;
6    import org.hibernate.Session;
7    import org.hibernate.SessionFactory;
8    import org.springframework.transaction.annotation.Propagation;
9    import org.springframework.transaction.annotation.Transactional;
10
11   import university.Student;
12
13   /*
14    * Method 3: Use Hibernate maximally - do not extend
15    * HibernateDaoSupport
16    */
17   public class StudentDAOHibernateImpl3
18       implements StudentDAO
19   {
20       private SessionFactory sessionFactory;
21
22       @Transactional(propagation = Propagation.REQUIRED)
23       public Collection<Student> findAllStudents()
24       {
25           Session session = sessionFactory.getCurrentSession();
26           Query q = session.createQuery("from Student");
27           return q.list();
28       }
29
30       @Transactional(propagation = Propagation.REQUIRED)
31       public int getStudentCount()
32       {
33           Session session = sessionFactory.getCurrentSession();
34           Query q = session.createQuery("select count(s) from Student s");
35           Long count = (Long) q.uniqueResult();
36
37           return count.intValue();
38       }
39
40       @Transactional(propagation = Propagation.REQUIRED)
41       public void insertStudent(Student s)
42       {
43           Session session = sessionFactory.getCurrentSession();
44           session.save(s);
45       }
```

```
46
47      public SessionFactory getSessionFactory()
48      {
49          return sessionFactory;
50      }
51
52      public void setSessionFactory(SessionFactory sessionFactory)
53      {
54          this.sessionFactory = sessionFactory;
55      }
56  }
```

```
1    package university;
2
3    import static org.junit.Assert.assertEquals;
4
5    import org.junit.Test;
6    import org.junit.runner.RunWith;
7    import org.springframework.beans.factory.annotation.Autowired;
8    import org.springframework.test.context.ContextConfiguration;
9    import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
10   import org.springframework.transaction.annotation.Transactional;
11
12   import dao.StudentDAO;
13
14   @RunWith(SpringJUnit4ClassRunner.class)
15   @ContextConfiguration(locations = "/spring.xml")
16   @Transactional
17   public class TestStudentDAO
18   {
19       @Autowired
20       private StudentDAO dao;
21
22       @Test
23       public void testStudentCount()
24       {
25           int countBefore = dao.getStudentCount();
26
27           Student s = new Student();
28           s.setGpa(3.42);
29           s.setName("Horace Greeley");
30           dao.insertStudent(s);
31
32           int countAfter = dao.getStudentCount();
33           assertEquals("Count should increment", 1,
34               countAfter - countBefore);
35       }
36   }
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <tx:annotation-driven/>

    <bean id="studentDataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.EmbeddedDriver" />
        <property name="url"
            value="jdbc:derby:c:\springclass\db\StudentDB" />
    </bean>

    <bean id="studentSessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="studentDataSource" />
        <property name="mappingResources">
            <list>
                <value>university/Student.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.DerbyDialect
                </prop>
            </props>
        </property>
    </bean>

    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="studentSessionFactory" />
```

```
46        </bean>
47
48        <bean id="studentDAO3" class="dao.StudentDAOHibernateImpl3">
49            <property name="sessionFactory" ref="studentSessionFactory" />
50        </bean>
51
52    </beans>
```

```
1    <?xml version="1.0"?>
2    <!DOCTYPE hibernate-mapping PUBLIC
3        "-//Hibernate/Hibernate Mapping DTD//EN"
4        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5    <hibernate-mapping>
6        <class name="university.Student" table="STUDENT">
7            <id name="studentID" column="STUDENTID">
8                <generator class="identity" />
9            </id>
10           <property name="name" column="NAME" />
11           <property name="gpa" column="gpa" />
12       </class>
13   </hibernate-mapping>
14
```

Chapter Summary

In this chapter, you learned:

- How to write unit tests that test Spring beans inside or outside of the Spring container
- How to write integration tests for Spring beans, including those that access a database

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Remoting with RMI, HttpInvoker and JMS

- Spring and RMI
- Spring HTTP Invoker
- Spring and JMS

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Remote Objects

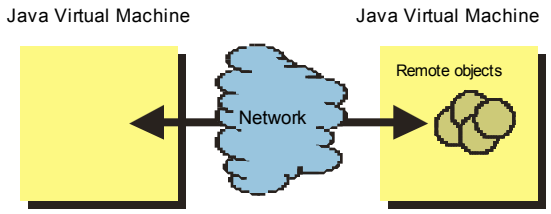
- Distributed computing is a architecture in which programs run on more than one computer distributed across a network, often organized into **client/server**
- Historically, architects have used technologies such as CORBA, DCOM and SOAP for language-neutral client/server applications
- Java provides **Remote Method Invocation** for Java-based client/server applications
- Spring provides "bridges" to integrate with various client/server technologies

A more modern client/server architecture might use JavaScript in a browser and remote services organized according to the REST model.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to RMI

- Remote Method Invocation is a Java-only distributed-object technology
- RMI is an enabling technology for EJBs



Sample Client Code

```
MyRemoteObject o = . . .;

o.someMethod();
```


RMI's purpose is to make objects in separate virtual machines look and act like local objects. The virtual machine that calls the remote object is sometimes referred to as a client. Similarly, we refer to the VM that contains the remote as a server.

Obtaining a reference for a remote object is a bit different than for local objects, but once you have the reference, you call the remote object just as if it was local as shown in the code snippet. The RMI infrastructure will automatically intercept the request, find the remote object, and dispatch the request remotely.

This location transparency even includes garbage collection. In other words, the client doesn't have to do anything special to release the remote object -- the RMI infrastructure and the remote VM handle the garbage collection for you.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

RMI Shortcomings

- RMI is a standard Java API, but it's cumbersome to work with:
  - You need to define an interface that extends java.rmi.Remote in which all methods throw java.rmi.RemoteException
  - You need to create a *main* program that exposes objects in the RMI **registry**

You also need to run the RMI registry as a daemon process. Clients then look up remote objects by name and the registry returns a "stub", also known as a proxy.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring RMI Support

- Spring provides the following "bridge" beans:
  - **RmiServiceExporter** - exports a Spring bean as a remote RMI object
  - **RmiProxyFactoryBean** - client-side factory to create objects that represent remote objects
- These beans free the developer from having to know much about RMI

The RmiServiceExporter bean creates a server instance and deploys it as an RMI service, including giving the bean an RMI name in the registry. It also automatically checks to see if there's an instance of the RMI registry running, and if not, starts one.

The RmiProxyFactoryBean does the RMI lookup for the client and returns a Spring bean that represents the remote object.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring RMI Example

HelloService.java

HelloServiceRmiImpl.java

HelloRmiServer.java

server.xml

HelloClient.java

ClientMain.java

client.xml

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package testrmi.server;
2
3    public interface HelloService
4    {
5        public String sayHello();
6    }
```

```
1    package testrmi.server;
2
3    import org.springframework.stereotype.Component;
4
5    @Component
6    public class HelloServiceRmiImpl implements HelloService
7    {
8
9        @Override
10       public String sayHello()
11       {
12           return "Hello, world";
13       }
14
15   }
```

```
1    package testrmi.server;
2
3    import org.springframework.context.support.AbstractApplicationContext;
4    import org.springframework.context.support.FileSystemXmlApplicationContext;
5
6    public class HelloRmiServer
7    {
8
9        public static void main(String[] args)
10       {
11           AbstractApplicationContext ctx = new FileSystemXmlApplicationContext(
12                   "src/server.xml");
13       }
14
15   }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.sp
4        xmlns:context="http://www.springframework.org/schema/context"
5        xmlns:jee="http://www.springframework.org/schema/jee" xmlns:jms="http://www.sp
6        xmlns:lang="http://www.springframework.org/schema/lang" xmlns:tx="http://www.s
7        xmlns:util="http://www.springframework.org/schema/util"
8        xsi:schemaLocation="http://www.springframework.org/schema/aop
9            http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
10           http://www.springframework.org/schema/beans
11           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
12           http://www.springframework.org/schema/context
13           http://www.springframework.org/schema/context/spring-context-3.1.xsd
14           http://www.springframework.org/schema/jee
15           http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
16           http://www.springframework.org/schema/lang
17           http://www.springframework.org/schema/lang/spring-lang-3.1.xsd
18           http://www.springframework.org/schema/tx
19           http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
20           http://www.springframework.org/schema/util
21           http://www.springframework.org/schema/util/spring-util-3.1.xsd">
22
23       <context:component-scan base-package="testrmi" />
24
25       <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
26           <property name="serviceName" value="HelloService" />
27           <property name="serviceInterface"
28               value="testrmi.server.HelloService" />
29           <property name="service" ref="helloServiceRmiImpl" />
30       </bean>
31   </beans>
```

```
1    package testrmi.client;
2
3    import org.springframework.beans.factory.annotation.Autowired;
4    import org.springframework.stereotype.Component;
5
6    import testrmi.server.HelloService;
7
8    @Component
9    public class HelloClient
10   {
11       @Autowired
12       private HelloService helloService;
13
14       public String getHelloMsg()
15       {
16           return helloService.sayHello();
17       }
18
19   }
```

```
1    package testrmi.client;
2
3    import org.springframework.context.support.AbstractApplicationContext;
4    import org.springframework.context.support.FileSystemXmlApplicationContext;
5
6    public class ClientMain
7    {
8
9        public static void main(String[] args)
10       {
11           AbstractApplicationContext ctx = new FileSystemXmlApplicationContext(
12                   "src/client.xml");
13
14           HelloClient client = (HelloClient)ctx.getBean("helloClient");
15
16           System.out.println(client.getHelloMsg());
17
18           ctx.close();
19       }
20
21   }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.sp
4        xmlns:context="http://www.springframework.org/schema/context"
5        xmlns:jee="http://www.springframework.org/schema/jee" xmlns:jms="http://www.sp
6        xmlns:lang="http://www.springframework.org/schema/lang" xmlns:tx="http://www.s
7        xmlns:util="http://www.springframework.org/schema/util"
8        xsi:schemaLocation="http://www.springframework.org/schema/aop
9            http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
10           http://www.springframework.org/schema/beans
11           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
12           http://www.springframework.org/schema/context
13           http://www.springframework.org/schema/context/spring-context-3.1.xsd
14           http://www.springframework.org/schema/jee
15           http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
16           http://www.springframework.org/schema/lang
17           http://www.springframework.org/schema/lang/spring-lang-3.1.xsd
18           http://www.springframework.org/schema/tx
19           http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
20           http://www.springframework.org/schema/util
21           http://www.springframework.org/schema/util/spring-util-3.1.xsd">
22
23       <context:component-scan base-package="testrmi" />
24
25       <bean id="helloService" class="org.springframework.remoting.rmi.RmiProxyFactor
26           <property name="serviceUrl" value="rmi://localhost:1099/HelloService" />
27           <property name="serviceInterface"
28               value="testrmi.server.HelloService" />
29       </bean>
30
31
32   </beans>
```

Client/Server Firewall Issues

- RMI requires the use of a special **daemon** process that listens on a non-standard port 1099
- Many organizations prefer not to open that port on firewalls, due to security concerns
- Spring provides the **HTTP Invoker** to allow RMI-like applications to use the firewall-friendly HTTP protocol

It is possible to configure basic RMI to "tunnel" over HTTP, but it's a complex process and has performance implications.

There are other third-party approaches to this problem, including Caucho Hessian and Burlap. Spring does provide integration with these, but we will not cover that in this chapter, instead focusing on Spring's own HTTP Invoker.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring HTTP Invoker Support

- Spring provides the following "bridge" beans:
  - **HttpInvokerServiceExporter** - exports a Spring bean as a remote object over HTTP
  - **HttpInvokerProxyFactoryBean** - client-side factory to create objects that represent remote objects
- These beans free the developer from having to know much about HTTP remote invocation

This support is quite similar in concept to the Spring RMI bridge covered earlier, but doesn't require the RMI registry.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring HttpInvoker Example

HelloService.java

HelloServiceHttpInvokerImpl.java

web.xml

hello-servlet.xml

HelloClient.java

ClientMainHttpInvoker.java

client-httpinvoker.xml

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package testrmi.server;
2
3    public interface HelloService
4    {
5        public String sayHello();
6    }
```

```
1    package testhttpinvoker.server;
2
3    import org.springframework.stereotype.Component;
4
5    @Component
6    public class HelloServiceHttpInvokerImpl implements HelloService
7    {
8
9        @Override
10       public String sayHello()
11       {
12           System.out.println("Received a request");
13           return "Hello, world";
14       }
15
16   }
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xmlns="http://java.sun.com/xml/ns/javaee"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
6      id="WebApp_ID" version="3.0">
7      <display-name>SpringHttpInvokerWeb</display-name>
8
9        <servlet>
10           <servlet-name>hello</servlet-name>
11           <servlet-class>
12               org.springframework.web.servlet.DispatcherServlet
13           </servlet-class>
14           <load-on-startup>1</load-on-startup>
15       </servlet>
16       <servlet-mapping>
17         <servlet-name>hello</servlet-name>
18         <url-pattern>/services/*</url-pattern>
19       </servlet-mapping>
20
21   </web-app>
```

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.sp
4        xmlns:context="http://www.springframework.org/schema/context"
5        xmlns:jee="http://www.springframework.org/schema/jee" xmlns:jms="http://www.sp
6        xmlns:lang="http://www.springframework.org/schema/lang" xmlns:tx="http://www.s
7        xmlns:util="http://www.springframework.org/schema/util"
8        xsi:schemaLocation="http://www.springframework.org/schema/aop
9            http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
10           http://www.springframework.org/schema/beans
11           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
12           http://www.springframework.org/schema/context
13           http://www.springframework.org/schema/context/spring-context-3.1.xsd
14           http://www.springframework.org/schema/jee
15           http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
16           http://www.springframework.org/schema/lang
17           http://www.springframework.org/schema/lang/spring-lang-3.1.xsd
18           http://www.springframework.org/schema/tx
19           http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
20           http://www.springframework.org/schema/util
21           http://www.springframework.org/schema/util/spring-util-3.1.xsd">
22
23       <context:component-scan base-package="testhttpinvoker" />
24
25       <bean name="/HelloService"
26           class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter
27           <property name="service" ref="helloServiceHttpInvokerImpl" />
28           <property name="serviceInterface" value="testhttpinvoker.server.HelloServi
29       </bean>
30
31   </beans>
```

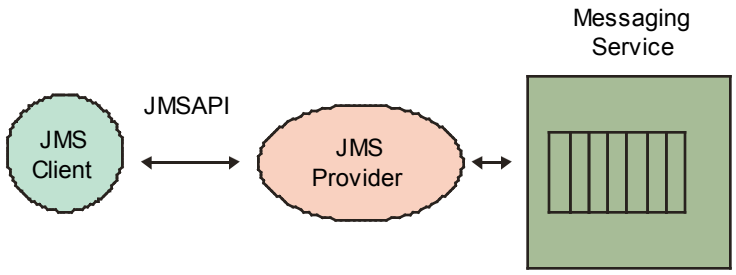
```
1    package testrmi.client;
2
3    import org.springframework.beans.factory.annotation.Autowired;
4    import org.springframework.stereotype.Component;
5
6    import testrmi.server.HelloService;
7
8    @Component
9    public class HelloClient
10   {
11       @Autowired
12       private HelloService helloService;
13
14       public String getHelloMsg()
15       {
16           return helloService.sayHello();
17       }
18
19   }
```

```
1    package testhttpinvoker.client;
2
3    import org.springframework.context.support.AbstractApplicationContext;
4    import org.springframework.context.support.FileSystemXmlApplicationContext;
5
6    public class ClientMainHttpInvoker
7    {
8
9        public static void main(String[] args)
10       {
11           AbstractApplicationContext ctx = new FileSystemXmlApplicationContext(
12                   "src/client-httpinvoker.xml");
13
14           HelloClient client = (HelloClient)ctx.getBean("helloClient");
15
16           System.out.println(client.getHelloMsg());
17
18           ctx.close();
19       }
20
21   }
```

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.sp
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:jee="http://www.springframework.org/schema/jee" xmlns:jms="http://www.sp
6       xmlns:lang="http://www.springframework.org/schema/lang" xmlns:tx="http://www.s
7       xmlns:util="http://www.springframework.org/schema/util"
8       xsi:schemaLocation="http://www.springframework.org/schema/aop
9           http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
10          http://www.springframework.org/schema/beans
11          http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
12          http://www.springframework.org/schema/context
13          http://www.springframework.org/schema/context/spring-context-3.1.xsd
14          http://www.springframework.org/schema/jee
15          http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
16          http://www.springframework.org/schema/lang
17          http://www.springframework.org/schema/lang/spring-lang-3.1.xsd
18          http://www.springframework.org/schema/tx
19          http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
20          http://www.springframework.org/schema/util
21          http://www.springframework.org/schema/util/spring-util-3.1.xsd">
22
23      <context:component-scan base-package="testhttpinvoker" />
24
25      <bean id="helloService"
26          class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBea
27          <property name="serviceUrl"
28              value="http://localhost:8080/SpringHttpInvokerWeb/services/HelloServic
29          <property name="serviceInterface"
30              value="testhttpinvoker.server.HelloService" />
31      </bean>
32
33  </beans>
```

What is the Java Message Service?

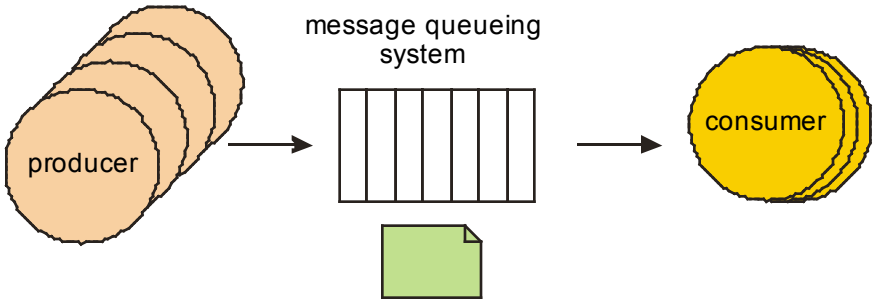
- The Java Message Service (JMS) provides a system-neutral interface so that Java programs can interact with messaging systems
- JMS is not itself a messaging service -- like JDBC, it simply provides a standard way to communicate with any messaging service that can act as a JMS provider
- All JEE-compliant platforms must support JMS, including message-driven EJBs



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Point to Point Messaging

- In a point-to-point setup, one or more producers send messages to a queue, which delivers each message to a single consumer
- Point-to-point messaging is simple and easy



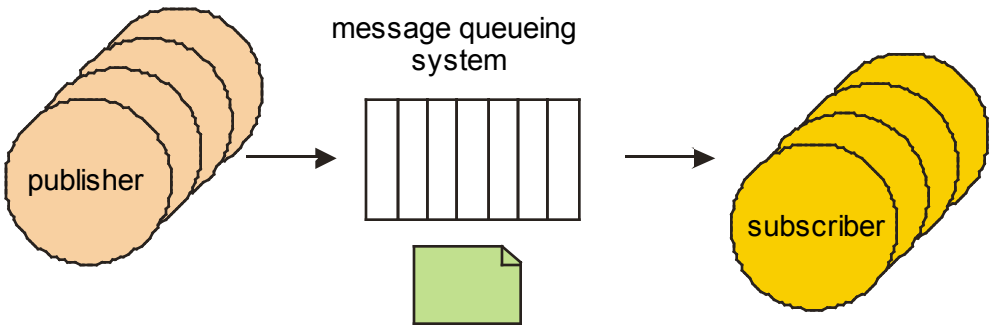
With point-to-point messaging, each message goes to exactly one consumer.

This form of messaging is like sending an email with any CCs-- the sender doesn't have to wait for recipient to get it, but the email goes to only one recipient.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Publish and Subscribe Messaging

- In a publish/subscribe setup, the messaging system acts as an intermediary between producers and consumers (subscribers)
- Each message goes to all subscribers



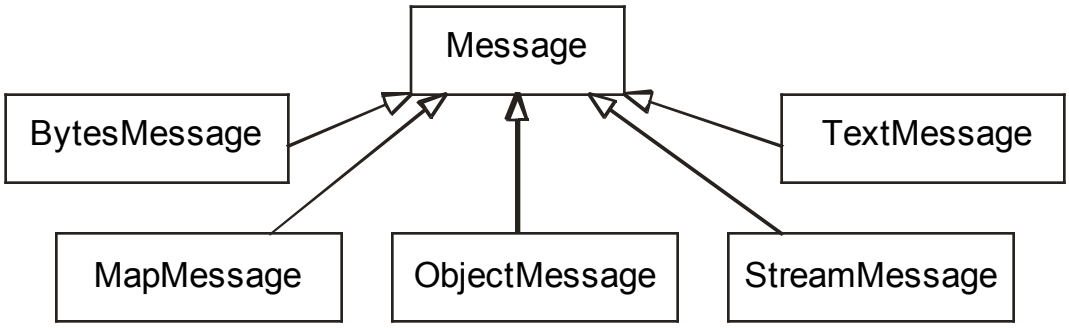
In a pub/sub setup, the subscribers typically must register (subscribe) to receive messages.

This type of messaging is like a mailing list (listserve) where whenever anyone posts a message to the list, the list propogates it to all subscribers.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JMS Message Types

- To provide flexibility, JMS supports various message types



A BytesMessage object is used to send a message containing a stream of uninterpreted bytes.

A MapMessage object is used to send a set of name-value pairs.

An ObjectMessage object is used to send a message that contains a serializable Java object.

A StreamMessage object is used to send a stream of primitive types in the Java programming language.

A TextMessage object is used to send a message containing a java.lang.String.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JMS Shortcomings

- JMS is a standard JEE API, but it's cumbersome to work with and requires lots of "boilerplate" coding:
    - You need to work with JNDI to define messaging destinations and factories
    - Receiving JMS messages using the JMS API involves a lot of code (EJB **message-driven beans** make that easier)

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

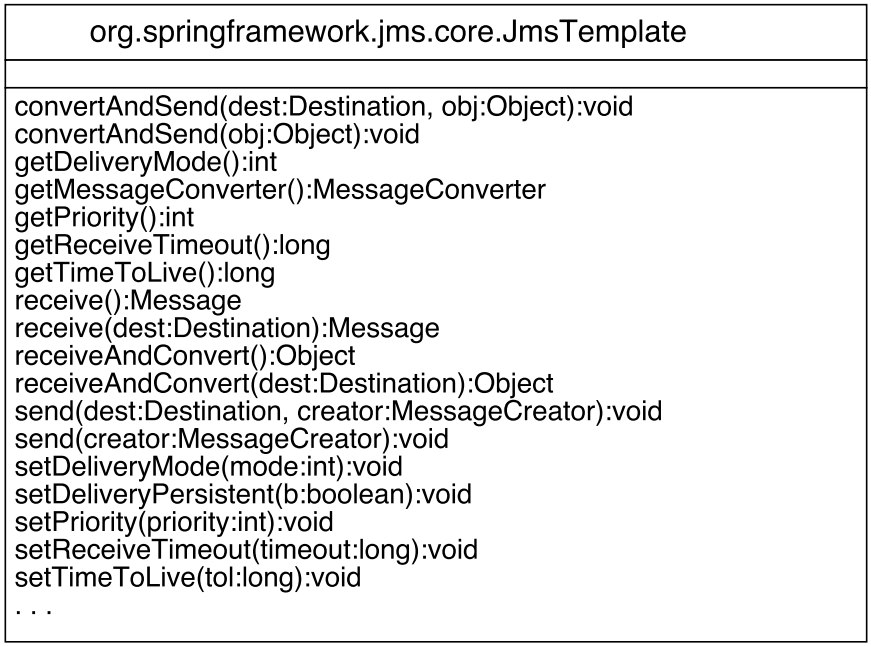
Spring and JMS

- Spring provides support for integrating JMS into Spring applications:
  - **JmsTemplate** type that handles boilerplate JMS coding
  - **Unchecked exceptions** for JMS exception handling
  - **MessageConverters** so you can centralize converting domain objects to/from JMS messages
  - **Gateway** classes that you can subclass in a similar fashion to JDBC DAO support classes
  - **POJO-based** alternative to message-driven beans for receiving JMS messages

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JmsTemplate Type

- This is the workhorse type for Spring programs that work with JMS



This notion of a "template" is a common Spring pattern that's also used with JDBC, Hibernate, iBATIS and other types of backends. The template essentially automates repetetive coding, making the Spring application simpler.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Accessing JNDI Resources

- JMS applications use **connection factories** and **destinations** (queues or topics) that are defined administratively and have JNDI names
- Spring applications can use the **jee:jndi-lookup** element to lookup the JMS artifacts so that Spring beans can reference them

```
1    <jee:jndi-lookup id="jmsConnectionFactory"
2      resource-ref="false"
3      jndi-name="jms/purchaseQCF" />
4
5    <jee:jndi-lookup id="purchaseQueue"
6       resource-ref="false"
7      jndi-name="jms/purchaseQ" />
```

If you create a local resource reference, then you specify resource-ref="true" - that would be the case if you created the resource reference in the web.xml for a Web application.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring the JmsTemplate

- You can inject a JmsTemplate into a Spring bean
- You must configure the template with the JMS connection factory

```
1    <bean id="jmsTemplate"
2      class="org.springframework.jms.core.JmsTemplate">
3      <property name="connectionFactory"
4       ref="jmsConnectionFactory" />
5    </bean>
6
7    <bean id="myJmsSender" class="beans.MyJmsSender">
8      <property name="template" ref="jmsTemplate" />
9      <property name="destination" ref="purchaseQueue" />
10   </bean>
```


Here we show configuring a Spring bean that uses an injected template. Note how we also inject the JMS destination that we looked up previously.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JmsTemplate Examples

MyJmsSender.java

MyJmsReceiver.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

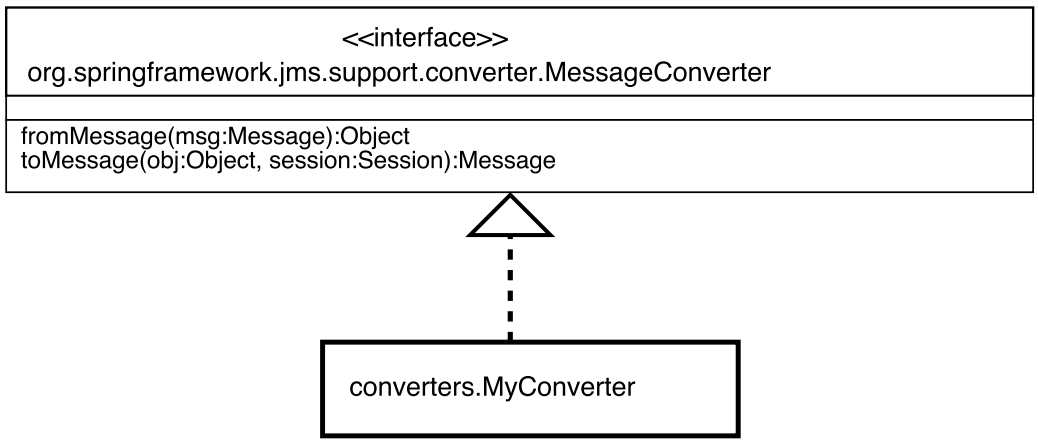
```
1    package beans;
2
3    import javax.jms.Destination;
4    import javax.jms.JMSException;
5    import javax.jms.Message;
6    import javax.jms.ObjectMessage;
7    import javax.jms.Session;
8
9    import org.springframework.jms.core.JmsTemplate;
10   import org.springframework.jms.core.MessageCreator;
11
12   import domain.Purchase;
13
14   public class MyJmsSender
15   {
16       private JmsTemplate template;
17       private Destination destination;
18
19       public Destination getDestination()
20       {
21           return destination;
22       }
23
24       public void setDestination(Destination destination)
25       {
26           this.destination = destination;
27       }
28
29       public JmsTemplate getTemplate()
30       {
31           return template;
32       }
33
34       public void setTemplate(JmsTemplate template)
35       {
36           this.template = template;
37       }
38
39       public void sendPurchase(final Purchase p)
40       {
41           class MyMessageCreator implements MessageCreator
42           {
43               public Message createMessage(Session session)
44                   throws JMSException
45               {
```

```
46                  ObjectMessage msg = session.createObjectMessage();
47                  msg.setObject(p);
48
49                  return msg;
50              }
51          }
52
53          MyMessageCreator mmc = new MyMessageCreator();
54          template.send(destination, mmc);
55      }
56
57
58
59  }
```

```
1    package beans;
2
3    import javax.jms.Destination;
4    import javax.jms.JMSException;
5    import javax.jms.ObjectMessage;
6
7    import org.springframework.jms.core.JmsTemplate;
8
9    import domain.Purchase;
10
11   public class MyJmsReceiver
12   {
13       private JmsTemplate template;
14       private Destination destination;
15
16       public Destination getDestination()
17       {
18           return destination;
19       }
20
21       public void setDestination(Destination destination)
22       {
23           this.destination = destination;
24       }
25
26       public JmsTemplate getTemplate()
27       {
28           return template;
29       }
30
31       public void setTemplate(JmsTemplate template)
32       {
33           this.template = template;
34       }
35
36       public Purchase receivePurchase()
37           throws JMSException
38       {
39           ObjectMessage msg = (ObjectMessage) template.receive(destination);
40           return (Purchase)msg.getObject();
41       }
42
43   }
```

Converting Messages

- Often, programs need to convert domain objects to/from JMS messages (e.g. MapMessage)
- You can centralize such processing by writing a class that implements the **MessageConverter** interface and then call the template's **convertAndSend** and **receiveAndConvert** methods



In the example in this chapter, a converter isn't very helpful, since the example uses the JMS ObjectMessage type - that's easy to convert to/from a domain class like Purchase. But if you use other JMS message types, then writing a converter can save you from having to duplicate the code in several places.

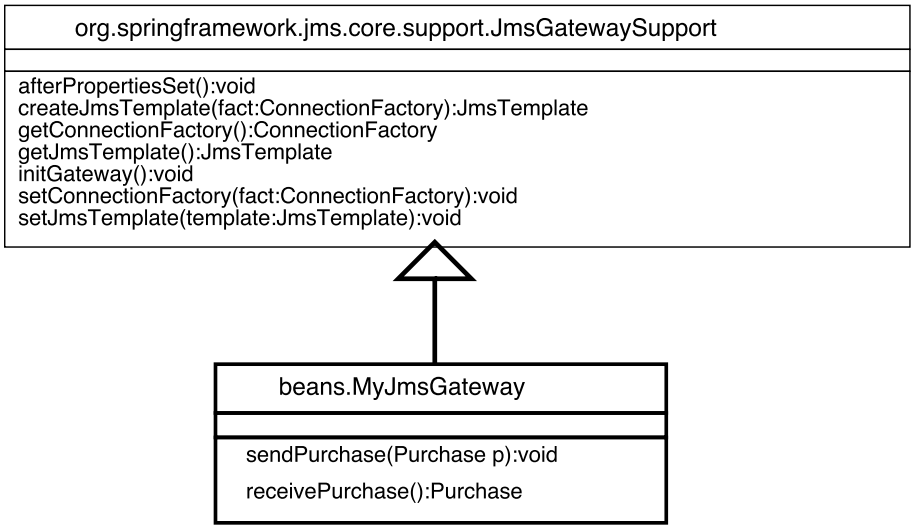
You must configure the converter in the Spring configuration file:

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

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Subclassing JmsGatewaySupport

- In a fashion similar to DAO support classes, Spring provides the **JmsGatewaySupport** superclass that composes a JmsTemplate
- Using this superclass makes configuration a bit easier since it provides the template



You could configure the "gateway" in Spring.xml:

```
<bean id="myGateway"
  class="beans.MyJmsGateway" />
<property name="connectionFactory"
  ref="jmsConnectionFactory" / >
</bean>
```

Note that you don't need to explicitly configure the template!

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

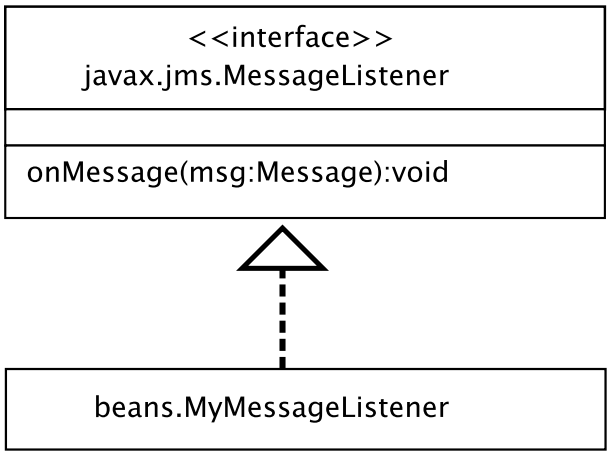
Receiving Messages

- There are three basic ways to act as a JMS receiver:
    - Use native JMS API with or without Spring
    - Use a message-driven Enterprise JavaBean
    - Use a Spring message-driven POJO
- In this chapter, we've already covered the first option and will now cover the third

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The MessageListener Interface

- Spring beans that implement this interface can act as the receiver of JMS messages
- You must configure a Spring **message listener container** that actually receives the JMS messages and then calls the listener's **onMessage** method



This is very similar in concept to JEE's "message driven bean".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a Message Listener Container

- Spring provides three types of message-listener containers
- Note that some application servers may require using one of the types exclusively

```
SimpleMessageListenerContainer

DefaultMessageListenerContainer

ServerSessionMessageListenerContainer
```


SimpleMessageListenerContainer provides the basic support to listen for JMS messages and then call your listener.

DefaultMessageListenerContainer does everything that SimpleMessageListenerContainer does, plus supports JMS transactions.

ServerSessionMessageListenerContainer does everything that DefaultMessageListenerContainer does, plus supports dynamic management of JMS sessions.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

POJO Listener Example

MyMessageListener.java

```
<!-- note that class name continues on two lines
   in this listing so it fits on the printed page -->
<bean
 class="org.springframework.jms.listener.
    DefaultMessageListenerContainer">
  <property name="connectionFactory"
     ref="jmsConnectionFactory" />
  <property name="destination" ref="purchaseQueue" />
  <property name="messageListener"
     ref="myMessageListener" />
</bean>

<bean id="myMessageListener"
  class="beans.MyMessageListener" />
```


This page shows a fragment of the spring.xml file to configure the message-driven POJO.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package beans;
2
3    import java.io.FileWriter;
4    import java.io.IOException;
5    import java.io.PrintWriter;
6
7    import javax.jms.JMSException;
8    import javax.jms.Message;
9    import javax.jms.MessageListener;
10   import javax.jms.ObjectMessage;
11
12   import domain.Purchase;
13
14   public class MyMessageListener implements MessageListener
15   {
16
17       public void onMessage(Message m)
18       {
19           System.out.println("In onMessage");
20           ObjectMessage msg = (ObjectMessage)m;
21           try
22           {
23               Purchase p = (Purchase)msg.getObject();
24
25               PrintWriter out = new PrintWriter(
26                   new FileWriter("c:/temp/purchase.log"));
27
28               out.println("Date: " + p.getDate());
29               out.println("Amount: " + p.getAmount());
30               out.println("Reason: " + p.getReason());
31               out.close();
32           }
33           catch (JMSException e)
34           {
35               e.printStackTrace();
36           }
37           catch (IOException e)
38           {
39               e.printStackTrace();
40           }
41       }
42
43   }
```

Chapter Summary

In this chapter, you learned about various ways to create distributed Spring applications, including:

- Spring RMI
- HTTP Invoker
- Spring JMS

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Spring Boot

- What is Spring Boot?
- Starter Projects
- Autoconfiguration
- Building and Running

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Spring Boot?

- Spring Boot is an add-on project to Spring that provides:
  - A generator for starter projects that defines common configurations (e.g SpringMVC with JPA)
  - Easier configuration with no XML
  - Ability to run Web applications without deploying to an application server
  - An optional **command-line** interface (CLI)

The home page of Spring Boot is at:

`http://projects.spring.io/spring-boot`

Spring Boot provides several other features, including runtime instrumentation to check the health of an application (the Actuator).

The CLI lets you generate generate starter projects and build them. The Actuator provides a Web-based console or SSH access to the application so you examine many aspects of the app, including getting a list of beans, HTTP trace and so forth.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Building Spring Boot Projects

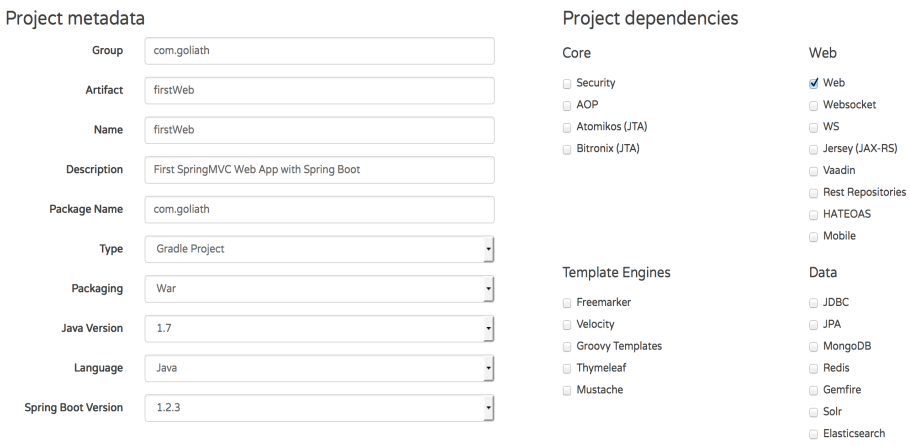
- Spring Boot supports both Maven and Gradle and can generate starter build scripts for both tools
- You can read more about Maven at: https://maven.apache.org/
- You can read more about Gradle at: https://gradle.org/

In this chapter, we'll use Gradle.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Generating a Starter Project

- Spring Boot provides the **Initializr** to generate projects with commonly used features
- You can access the Initializr on the Web at: http://start.spring.io or via the Spring Boot Command-Line Interface



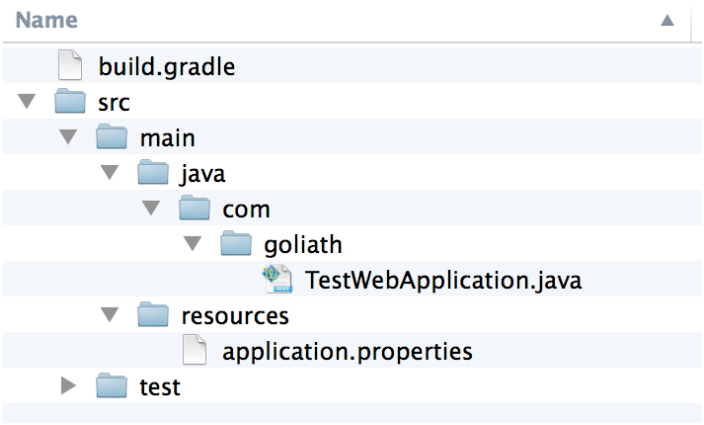
The Intitializr generates a ZIP file that you can download and unzip.

You can also run the Initializr from within the Spring Tool Suite Eclipse environment.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Gradle Project Structure

- Once you unzip the project generated by the Initializr, you will have a directory structure:



The Initializr generates a basic harness for unit testing. We will not cover it in this overview.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Generated Build Script

- The Initializr generated a Gradle build script that specifies project metadata, **plugins**, **repositories** and **dependencies**

```
1    ...
2    apply plugin: 'java'
3    apply plugin: 'eclipse'
4    apply plugin: 'idea'
5    apply plugin: 'spring-boot'
6    apply plugin: 'io.spring.dependency-management'
7
8    repositories {
9        mavenCentral()
10   }
11
12   dependencies {
13       compile(
14         "org.springframework.boot:spring-boot-starter-web")
15   }
```

Plugins add "tasks" that you can execute. For example, the java plugin knows how to compile the Java code in your project.

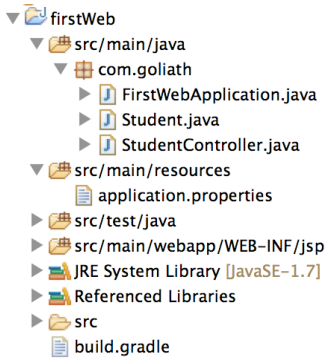
Repositories are sources of dependencies, i.e. libraries. You can use Maven, Ivy or locally defined repositories.

Dependencies are libraries that your code needs. Note that you can specify when the dependency should be satisfied, i.e. at compile time. Gradle fetches the dependencies from the repositories. Note that Gradle supports "transitive dependencies" - if a specified dependency itself requires other code, Gradle will that that too.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Generating an Eclipse Project

- The project generated by Initialzr does not contain support for an IDE out of the box, but the build script contains **eclipse** and **idea** plugins
- For Eclipse, run **gradle eclipse**, then import the project into your workspace
- You can then create the Java classes, Web pages and so forth that comprise your application



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The 'main' Program

- The Initializr generated a **main()** method class so you can run the application, even if it's a Web application
- The **@SpringBootApplication** annotation configures the class to use Spring Boot's autoconfiguration facility so we don't need any XML configuration

```
1    @SpringBootApplication
2    public class FirstWebApplication {
3
4        public static void main(String[] args) {
5            SpringApplication.run(
6              FirstWebApplication.class, args);
7        }
8    }
```

You shouldn't need to modify the class under normal circumstances.

According to the Spring Boot docs, the @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Boot Autoconfiguration

- Without Spring Boot, applications need to explicitly configure beans such as data sources, Spring MVC view mappings and so forth
- Spring Boot automatically configures default beans based on what it finds on the application's CLASSPATH
- You can override the defaults if needed, but if the defaults are OK, you can run an application with little or no Spring configuration

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring SpringMVC JSP Views

- It's traditional in Spring MVC to place JSPs in the WEB-INF folder
- For this to work with Spring Boot, you can update the **application.properties** to specify the exact location

```
1    spring.mvc.view.prefix: /WEB-INF/jsp/
2    spring.mvc.view.suffix: .jsp
```


Of course, you'll also need to create this directory in your project. You should also ensure that there's no trailing spaces at the end of any of the lines in application.properties.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Running the Project

- The Spring Boot plugin provides the **bootRun** task to run Web applications on an embedded Tomcat instance
- This is an easy and fast way to run your application without needing to load it onto an application server



It's also possible to build a standard WAR file for later deployment. See http://docs.spring.io/spring-boot/docs/current/reference/html/howto-traditional-deployment.html

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Complete Example

notes.txt

build.gradle

application.properties

FirstWebApplication.java

Student.java

StudentController.java

studentList.jsp

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
- Download, install gradle and put its "bin" directory
  in PATH

- Create the project using Web-based Spring Boot Initializr:
    Metadata:
        Group:  com.goliath
        Artifact: firstWeb
        Name: firstWeb
        Description: First SpringMVC Web App with Spring Boot
        Package Name: com.goliath
        Type: Gradle Project
        Packaging: JAR
        Java Version: 1.7
        Language: Java
        Spring Boot Version: 1.2.3

    Project dependencies:
        Web

- Generate Project, then unzip

- In terminal (command prompt), change to project directory

- Generate Eclipse project:

    gradle eclipse

- Import "existing project" into Eclipse

- In com.goliath package, create class Student.

- In com.goliath package, create StudentController

- Add to application.properties:

spring.view.prefix: /WEB-INF/jsp/
spring.view.suffix: .jsp

- Create source folder: src/main/webapp/WEB-INF/jsp

- In the new folder, create studentList.jsp

- In gradle.build, add plugin:

    apply plugin: 'war'
```

```
46
47   - In gradle.build, add dependencies:
48
49       providedRuntime("org.apache.tomcat.embed:tomcat-embed-jasper")
50       providedRuntime("javax.servlet:jstl")
51
52   - In terminal:
53
54      gradle bootRun
55
56   - In browser, navigate to http://localhost:8080/studentList
57
58
```

```
1   buildscript {
2       ext {
3           springBootVersion = '1.2.3.RELEASE'
4       }
5       repositories {
6           mavenCentral()
7       }
8       dependencies {
9           classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBoot
10          classpath("io.spring.gradle:dependency-management-plugin:0.5.0.RELEASE")
11      }
12  }
13
14  apply plugin: 'java'
15  apply plugin: 'eclipse'
16  apply plugin: 'idea'
17  apply plugin: 'spring-boot'
18  apply plugin: 'io.spring.dependency-management'
19  apply plugin: 'war'
20
21
22  jar {
23      baseName = 'firstWeb'
24      version = '0.0.1-SNAPSHOT'
25  }
26  sourceCompatibility = 1.7
27  targetCompatibility = 1.7
28
29  repositories {
30      mavenCentral()
31  }
32
33
34  dependencies {
35      compile("org.springframework.boot:spring-boot-starter-web")
36      testCompile("org.springframework.boot:spring-boot-starter-test")
37      providedRuntime("org.apache.tomcat.embed:tomcat-embed-jasper")
38      providedRuntime("javax.servlet:jstl")
39  }
40
41
42  eclipse {
43      classpath {
44           containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')
45           containers 'org.eclipse.jdt.launching.JRE_CONTAINER/org.eclipse.jdt.inter
```

```
46      }
47  }
48
49  task wrapper(type: Wrapper) {
50      gradleVersion = '2.3'
51  }
```

```
1    spring.view.prefix: /WEB-INF/jsp/
2    spring.view.suffix: .jsp
```

```
1    package com.goliath;
2
3    import org.springframework.boot.SpringApplication;
4    import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6    @SpringBootApplication
7    public class FirstWebApplication {
8
9        public static void main(String[] args) {
10           SpringApplication.run(FirstWebApplication.class, args);
11       }
12   }
```

```
1    package com.goliath;
2
3    public class Student
4    {
5        private int studentID;
6        private String name;
7        private double gpa;
8
9        public Student()
10       {
11       }
12
13       public Student(int studentID, String name, double gpa)
14       {
15           this.studentID = studentID;
16           this.name = name;
17           this.gpa = gpa;
18       }
19
20       public int getStudentID()
21       {
22           return studentID;
23       }
24
25       public void setStudentID(int studentID)
26       {
27           this.studentID = studentID;
28       }
29
30       public String getName()
31       {
32           return name;
33       }
34
35       public void setName(String name)
36       {
37           this.name = name;
38       }
39
40       public double getGpa()
41       {
42           return gpa;
43       }
44
45       public void setGpa(double gpa)
```

```
46        {
47            this.gpa = gpa;
48        }
49
50    }
```

```
1    package com.goliath;
2
3    import java.util.ArrayList;
4    import java.util.List;
5
6    import org.springframework.stereotype.Controller;
7    import org.springframework.ui.Model;
8    import org.springframework.web.bind.annotation.RequestMapping;
9    import org.springframework.web.bind.annotation.RequestMethod;
10
11   @Controller
12   public class StudentController
13   {
14
15       @RequestMapping(value = "/studentList", method = RequestMethod.GET)
16       public String getStudents(Model model)
17       {
18           List<Student> students = new ArrayList<Student>();
19
20           students.add(new Student(12, "Sue Smith", 3.44));
21           students.add(new Student(87, "Harry Wolfe", 2.68));
22           students.add(new Student(947, "Jerry Olde", 3.12));
23
24           model.addAttribute("students", students);
25
26           return "studentList";
27       }
28   }
```

```
1    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
2    <DOCTYPE html>
3    <html lang="en">
4      <head>
5        <title>Student List</title>
6      </head>
7      <body>
8        <h1>Student List</h1>
9          <ul>
10           <c:forEach items="${students}" var="aStudent">
11             <li>${aStudent.name}</li>
12           </c:forEach>
13         </ul>
14     </body>
15   </html>
```

Chapter Summary

In this chapter, you learned:

- What Spring Boot is all about
- How to configure, write and run a Spring Boot Web application

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to REST

- What is Representational State Transfer?
- Principles of REST
- Introduction to Java REST frameworks

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is REST?

- **Representational State Transfer** is an architecture for distributed systems that emphasizes statelessness
- REST was introduced by Roy Fielding, who was one of the authors of HTTP

The World Wide Web itself exhibits many REST principles.

Developers that use the REST style are sometimes humorously referred to as "RESTafarians".

Fielding's REST dissertation is available at:

`http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.html`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

REST vs SOAP

- REST provides a straightforward, simple architecture for providing highly scalable services (note that the Web itself is RESTful)
- SOAP provides a more complex but perhaps more enterprise-class architecture that includes additional standards such as WSDL, WS-Security and so forth

This is a highly contested argument -- both approaches have advantages.

Note that there is a research project from Sun for describing RESTful services in a similar fashion to WSDL: Web Application Description Language (WADL).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Principles of REST

- Every resource should have a unique ID
- Resources can have multiple representations
- Use links to connect resources
- Use standard HTTP methods
- Clients and servers communicate statelessly

These principles are adapted from an article by Stefan Tilkov, which can be found at:

http://www.infoq.com/articles/rest-introduction

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Resource IDs

- The Internet provides a resource-identification technique known as a **Uniform Resource Identifier** (URI), a.k.a URL
- RESTful applications use URIs to assign unique IDs to application resources
- Examples:
    - http://www.mydomain.com/students/5847
    - http://www.mydomain.com/students?id=44
    - http://www.mydomain.com/id=17

A URL is actually a type of URI, the other category being Uniform Resource Names (URN).

Note that the resources exposed by URIs are generally fairly abstract and don't necessarily exactly map to a database entry.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Resources With Multiple Representations

- For flexibility, applications that provide data can support different data formats, e.g. HTML, XML, CSV, JSON
- Clients can then request the resource in the format they like best, perhaps by specifying an HTTP **accept** header

```
//XML representation
<student>
  <ID>17</ID>
  <name>Sue Smith</student>
  <gpa>3.45</gpa>
</student>

//JSON representation
{"student":{"id":17, "name":"Sue Smith, "gpa":3.45}}
```

JSON is the JavaScript Object Notation. JSON is a lightweight data-interchange format that's easy for humans and programs to process.

You can find out more about JSON at:

http://www.json.org

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Use Links to Connect Resources

- HTML supports **hyperlinking** so that resources can reference other resources
- You can also perform linking in other data formats such as XML

```
<student xmlns:xlink="http://www.w3.org/1999/xlink">
  <ID>17</ID>
  <name>Sue Smith</student>
  <gpa>3.45</gpa>
  <advisor
    xlink:href="http://www.mydomain.com/advisors/156"/>
</student>
```


XML supports the Xlink standard for linking between documents.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Use Standard HTTP Methods

- HTTP supports **methods** that map to the four standard database Create, Retrieve, Update and Delete (CRUD) actions
- Note that only GET and POST can be sent directly from a browser so PUT, DELETE actions must be done programmatically (e.g. via AJAX)

HTTP Method	Database CRUD Operation
POST	CREATE
GET	SELECT (Retrieve)
PUT	UPDATE or CREATE
DELETE	DELETE

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Stateless Communication

- RESTful applications do **not** require the server to maintain communication state based on client requests
- This makes servers more scalable and allows the server to efficiently use techniques like caching, clustering and instance pooling

Most SOAP-based services are stateless, too.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

REST in Java

- JEE developers can use two basic approaches for writing RESTful services:
  - Write basic servlets
  - Use a REST framework
- Java RESTful frameworks include:
  - Restlets
  - JAX-WS REST
  - JAX-RS
  - Spring MVC REST

It's actually straightforward to write basic servlets that act as RESTful services, since servlets can override doGet(), doPost() and so forth. However, many developers prefer to use a framework that automates some of the low-level details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Restlet?

- Restlet is a lightweight, open-source REST framework for Java
- Restlet applications can run within a servlet container or as standalone Java applications
- Restlet was created by Jerome Louvel in 2005

The homepage for the Restlet framework is:

`http://www.restlet.org`

Restlets also have been ported to the Google Web Toolkit (GWT).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Simple Restlet Resource

```
1    public class HelloServerResource extends ServerResource
2    {
3        @Get
4        public String represent()
5        {
6            return "hello, world";
7        }
8    }
```


Imports not shown for brevity.

By default, Restlet provides a plain text representation.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JAX-WS and REST

- JAX-WS is the standard Web services API for Java Enterprise Edition 5 and later
- JAX-WS provides support for creating Web services with no SOAP envelope, i.e. RESTful services

```
1    @WebServiceProvider(serviceName = "HWService")
2    @BindingType(value = HTTPBinding.HTTP_BINDING)
3    public class MyProvider implements Provider<Source>
4    {
5      public Source invoke(Source source)
6      {
7         String replyElement =
8            new String("<p>hello world</p>");
9         StreamSource reply = new StreamSource(
10            new StringReader(replyElement));
11        return reply;
12     }
13   }
```


Some developers believe that the JAX-WS support for REST is not all it could be.

Here we show a JAX-WS endpoint that acts as a RESTful resource, returning the proverbial Hello, world string in HTML. Note the @BindingType annotation, which tells the JAX-WS runtime that this service uses HTTP binding rather than a SOAP envelope.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JAX-RS

- JAX-RS is an API for providing RESTful services in Java
- JAX-RS went through the Java Community Process as JSR-311 and is a part of JEE Version 6



The primary developer of Restlets was heavily involved in JSR-311 so there are similarities between the two frameworks.

If you don't have a JEE V6 container, you can use the reference implementation of JAX-RS, Jersey, which you can download and read about at:

`https://jersey.dev.java.net/`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A JAX-RS Service

- To use JAX-RS, you create **resource** classes, using annotations to configure them

```
1    @Path("/helloworld")
2    public class HelloWorldResource
3    {
4        @GET
5        @Produces("text/plain")
6        public String getMessage(
7                @DefaultValue("")
8                @QueryParam("name") String name)
9        {
10           return "Hello World, " + name;
11       }
12   }
```

This service would be available at:

`http://mydomain.com/myWebApp/helloworld?name=bill`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring MVC REST

- The popular Spring Framework's MVC module includes support for RESTful services

```
1    @Controller
2    @RequestMapping("rest")
3    public class StudentController
4    {
5      @RequestMapping(value = "/student/{id}",
6        method = RequestMethod.GET)
7        public @ResponseBody
8         Student getStudent(@PathVariable("id") int id)
9      {
10       Student s = findStudentByID(id);
11
12       return s;
13     }
14   }
14 - 16
```

This Spring class returns a JSON or XML representation of a Java "Student" object (source for Student not shown).

The service responds to URLs that have a student ID at the end of the URL:

```
http:// . . . /student/12
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the fundamentals of representational state transfer
- The basics of Java REST frameworks

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

RESTful CRUD Services

- Create
- Retrieve
- Update
- Delete

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Richardson Maturity Model

- According to the **Richardson Maturity Model**, which evaluates how RESTful an application is, apps that correctly implement CRUD are at **Level 2**

```
Level 0  Use a single verb, typically POST, and a single URI
         for all interactions. Most SOAP based services are
         like this

Level 1  Use a single verb, typically GET, but multiple URIs.
         These types of services often violate the idempotency
         of GET

Level 2  Use multiple verbs and multiple URIs. This includes
         CRUD RESTful apps

Level 3  Use Hypermedia as the Engine of Application State
```

You can read more about the RMM at:

`http://martinfowler.com/articles/richardsonMaturityModel.html`

According to the Wikipedia entry on "idempotence":

In computer science, the term idempotent is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

RESTful CRUD Service Overview

- Many applications use the four **CRUD** operations: **Create**, **Retrieve**, **Update** and **Delete**
- The REST community has come up with guidelines for how to implement CRUD using standard HTTP facilities

The Rest in Practice book by Leonard Richardson is a very good reference for these guidelines.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

HTTP Review

- HTTP provides several **methods** as well as standard **return codes** and **headers**

```
HTTP Methods
------------
GET, POST,
PUT, DELETE,
HEAD, PATCH,
OPTIONS, TRACE,
CONNECT

HTTP Headers
------------
Accept
Content-Type
Status
...
```

```
HTTP Status Codes
-----------------
200 OK
201 Created
204 No Content
400 Bad Request
404 Not Found
409 Conflict
500 Internal Server Error
...
```

For more information on HTTP, see the specification:

`http://www.w3.org/Protocols/rfc2616/rfc2616.html`

Note that this is not a complete listing of status codes and headers, only those commonly used by RESTful services.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Media Types

- The IANA defines **media types (MIME)** that services can use to specify data types

```
Common Media Types
------------------

application/json
text/plain
text/xml
application/vnd.*     -- vendor specific
```


For more information on the Internet Assigned Numbers Authority (IANA) and media types, see:

https://www.iana.org/assignments/media-types

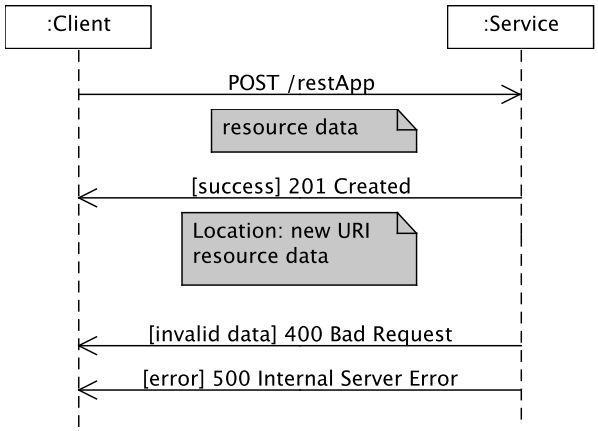
There is a lot of discussion in the REST community on whether to use standard media types like text/xml or application/json or to define custom media types for the application. See this article for a discussion:

http://www.infoq.com/news/2010/01/subbu-custom-media-types

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Implementing Create

- Client should **POST** a resource representation and will receive a possibly modified resource representation
- Service returns response code **201 Created** on success or **400 Bad Request** or **500 Internal Server Error**



The response message also contains an HTTP Location header that specifies the newly created resource's RESTful URI.

The body of the response contains a possibly different resource representation than was passed, since it might contain auto-generated data like a database primary key.

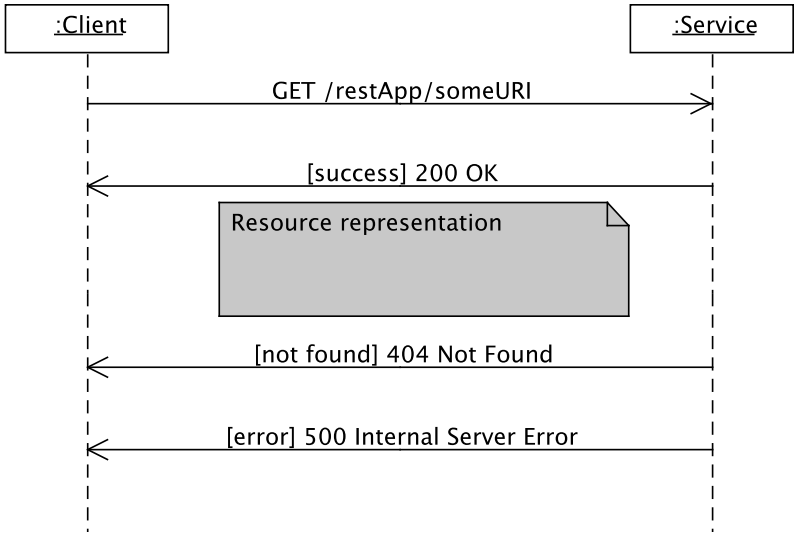
This page discusses using HTTP 201 vs 202:

`http://benramsey.com/blog/2008/04/http-status-201-created-vs-202-accepted/`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Implementing Retrieve

- Client should **GET** to a URI for the requested resource
- Service returns requested resource and **200 OK** or **404 Not Found** or **500 Internal Server Error**

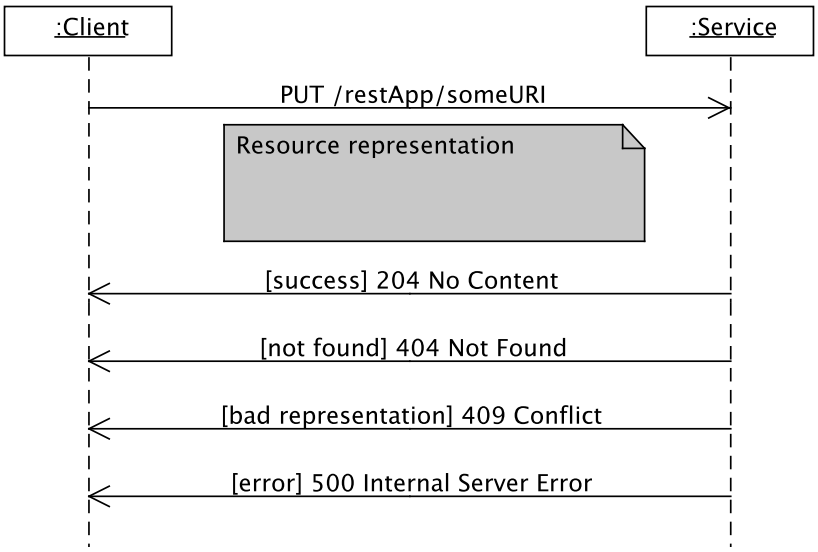


Even though the client can expect that the GET request itself doesn't change state on the server, it's not necessarily true that multiple GETs return the same data. For example, consider an Order-status request - the status might initially be "in progress" and eventually "complete".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Implementing Update

- Client should **PUT** a complete resource representation to the resource's URI
- Service updates the requested resource and returns **204 No Content**, **404 Not Found**, **409 Conflict** or **500 Internal Server Error**



Many RESTful applications follow this convention (established by Richardson and Ruby): <pre>- Use POST to create a resource identified by a service-generated URI. - Use POST to append a resource to a collection identified by a service-generated URI. - Use PUT to create or overwrite a resource identified by a URI computed by the client. </pre> The service could also return 200 OK, but the HTTP specification requires that such responses contain a response body, while 204 does not.

The service can return 409 Conflict if the supplied resource representation is incorrect, for example ill formed XML.

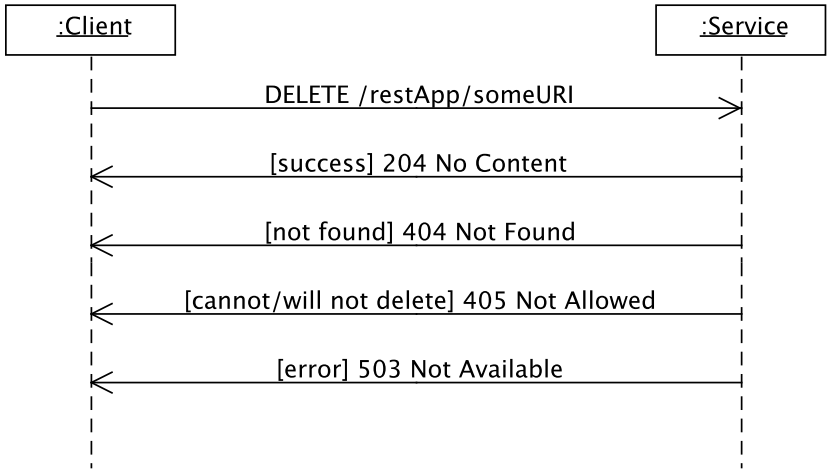
Note that the client must send complete resource representation. There is an emerging concept of using HTTP PATCH to provide just a delta to update.

Note that PUT should be idempotent since the client provides the entire resource. So if the service returns 500, it should be OK for the client to resend the request.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Implementing Delete

- Client uses HTTP **DELETE** specifying the resource's URI
- Service returns **204 No Content** on success or **404 Not Found**, **405 Method Not Allowed** or **503 Service Unavailable**



One reason the service might return 405 is if it refuses to delete for some reason. For example, the service might have a policy that once an order is fully complete, it cannot be deleted.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Complete JAX-RS CRUD Example

Student.java

MyContextListener.java

StudentApplication.java

StudentService.java

GetClient.java

PostClient.java

PutClient.java

DeleteClient.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package edu.bigcollege;
2
3    import java.io.Serializable;
4
5    import javax.xml.bind.annotation.XmlRootElement;
6
7    @SuppressWarnings("serial")
8    @XmlRootElement
9    public class Student implements Serializable
10   {
11       private int studentId;
12       private String name;
13       private double gpa;
14
15       public Student()
16       {
17       }
18
19       public Student(int studentId, String name, double gpa)
20       {
21           super();
22           this.studentId = studentId;
23           this.name = name;
24           this.gpa = gpa;
25       }
26
27       public int getStudentId()
28       {
29           return studentId;
30       }
31
32       public void setStudentId(int studentId)
33       {
34           this.studentId = studentId;
35       }
36
37       public String getName()
38       {
39           return name;
40       }
41
42       public void setName(String name)
43       {
44           this.name = name;
45       }
```

```
46
47      public double getGpa()
48      {
49          return gpa;
50      }
51
52      public void setGpa(double gpa)
53      {
54          this.gpa = gpa;
55      }
56
57      @Override
58      public int hashCode()
59      {
60          final int prime = 31;
61          int result = 1;
62          result = prime * result + studentId;
63          return result;
64      }
65
66      @Override
67      public boolean equals(Object obj)
68      {
69          if (this == obj)
70              return true;
71          if (obj == null)
72              return false;
73          if (getClass() != obj.getClass())
74              return false;
75          Student other = (Student) obj;
76          if (studentId != other.studentId)
77              return false;
78          return true;
79      }
80
81      @Override
82      public String toString()
83      {
84          StringBuilder builder = new StringBuilder();
85          builder.append("Student [studentId=").append(studentId)
86                  .append(", name=").append(name).append(", gpa=").append(gpa)
87                  .append("]");
88          return builder.toString();
89      }
90  }
```

```
1    package edu.bigcollege.listeners;
2
3    import java.util.ArrayList;
4    import java.util.List;
5
6    import javax.servlet.ServletContextEvent;
7    import javax.servlet.ServletContextListener;
8
9    import edu.bigcollege.Student;
10
11   public class MyContextListener implements ServletContextListener
12   {
13
14       @Override
15       public void contextDestroyed(ServletContextEvent ctx)
16       {
17       }
18
19       @Override
20       public void contextInitialized(ServletContextEvent ctx)
21       {
22           System.out.println("Context initialized");
23           List<Student> students = new ArrayList<Student>();
24
25           Student c = new Student();
26           c.setGpa(3.44);
27           c.setName("Sue Smith");
28           c.setStudentId(12);
29           students.add(c);
30
31           c = new Student();
32           c.setGpa(3.87);
33           c.setName("Bill Jones");
34           c.setStudentId(44);
35           students.add(c);
36
37           ctx.getServletContext().setAttribute("students", students);
38       }
39
40   }
```

```
1    package edu.bigcollege.service;
2
3    import java.util.HashSet;
4    import java.util.Set;
5
6    import javax.ws.rs.ApplicationPath;
7    import javax.ws.rs.core.Application;
8
9    @ApplicationPath("/rest/")
10   public class StudentApplication extends Application
11   {
12       @Override
13       public Set<Class<?>> getClasses()
14       {
15           Set<Class<?>> s = new HashSet<Class<?>>();
16           s.add(StudentService.class);
17
18           return s;
19       }
20   }
```

```
1    package edu.bigcollege.service;
2
3    import java.net.URI;
4    import java.net.URISyntaxException;
5    import java.util.List;
6    import java.util.Random;
7
8    import javax.servlet.ServletContext;
9    import javax.ws.rs.Consumes;
10   import javax.ws.rs.DELETE;
11   import javax.ws.rs.GET;
12   import javax.ws.rs.POST;
13   import javax.ws.rs.PUT;
14   import javax.ws.rs.Path;
15   import javax.ws.rs.PathParam;
16   import javax.ws.rs.Produces;
17   import javax.ws.rs.core.Context;
18   import javax.ws.rs.core.Response;
19   import javax.ws.rs.core.Response.ResponseBuilder;
20   import javax.ws.rs.core.Response.Status;
21
22   import edu.bigcollege.Student;
23
24   @Path("students")
25   public class StudentService
26   {
27       @Context
28       ServletContext servletContext;
29
30       @SuppressWarnings("unchecked")
31       @DELETE
32       @Path("{id}")
33       public Response delete(@PathParam("id") int id)
34       {
35           ResponseBuilder rb = null;
36
37           System.out.println("In delete: " + id);
38
39           Student found = findStudent(id);
40           if(found != null)
41           {
42               List<Student> students = (List<Student>) servletContext
43                       .getAttribute("students");
44
45               students.remove(found);
```

```
46              rb = Response.noContent();
47          }
48          else
49          {
50              rb = Response.status(Status.NOT_FOUND);
51          }
52
53          return rb.build();
54      }
55
56      @PUT
57      @Consumes({ "text/xml", "application/json" })
58      public Response update(Student s)
59      {
60          System.out.println("In update: " + s);
61          ResponseBuilder rb = null;
62
63          if(s != null)
64          {
65              Student found = findStudent(s.getStudentId());
66
67              if(found != null)
68              {
69                  found.setGpa(s.getGpa());
70                  found.setName(s.getName());
71                  rb = Response.noContent();
72              }
73              else
74              {
75                  rb = Response.status(Status.NOT_FOUND);
76              }
77          }
78          else
79          {
80              rb = Response.status(Status.CONFLICT);
81          }
82
83          return rb.build();
84      }
85
86
87      @SuppressWarnings("unchecked")
88      @POST
89      @Consumes({ "text/xml", "application/json" })
90      // need @Produces since we send back updated Student
```

```
91      // representation in response body
92      @Produces({ "text/xml", "application/json" })
93      public Response insert(Student s)
94      {
95          System.out.println("In insert: " + s);
96          ResponseBuilder rb = null;
97
98          if(s != null)
99          {
100             List<Student> students = (List<Student>) servletContext
101                     .getAttribute("students");
102
103             Random r = new Random();
104             s.setStudentId(Math.abs(r.nextInt()));
105
106             students.add(s);
107
108             try
109             {
110                 rb = Response.created(new URI("/students/" +
111                     s.getStudentId()));
112                 // marshal the student into the response body
113                 rb.entity(s);
114             }
115             catch (URISyntaxException e)
116             {
117                 System.out.println("ERROR inserting: " +
118                   e.getMessage());
119                 rb = Response.serverError();
120             }
121         }
122         else
123         {
124             rb = Response.status(Status.BAD_REQUEST);
125         }
126
127         return rb.build();
128     }
129
130     @SuppressWarnings("unchecked")
131     @GET
132     @Path("{id}")
133     @Produces({ "text/xml", "application/json" })
134     public Response getStudent(@PathParam("id") int id)
135     {
```

```
136         Response ret = null;
137
138         List<Student> students = (List<Student>) servletContext
139                 .getAttribute("students");
140
141         Student theStudent = null;
142         for (Student c : students)
143         {
144             if (c.getStudentId() == id)
145             {
146                 theStudent = c;
147                 break;
148             }
149         }
150
151         if (theStudent != null)
152         {
153             ret = Response.ok(theStudent).build();
154         }
155         else
156         {
157             ret = Response.status(Response.Status.NOT_FOUND).build();
158         }
159
160         return ret;
161     }
162
163     @SuppressWarnings("unchecked")
164     private Student findStudent(int id)
165     {
166         List<Student> students = (List<Student>) servletContext
167                 .getAttribute("students");
168
169         Student found = null;
170         for(Student temp : students)
171         {
172             if(temp.getStudentId() == id)
173             {
174                 found = temp;
175                 break;
176             }
177         }
178         return found;
179     }
180
```

```
181   }
```

```
1    package client;
2
3    import java.io.BufferedReader;
4    import java.io.InputStreamReader;
5    import java.net.HttpURLConnection;
6    import java.net.URL;
7
8    public class GetClient
9    {
10
11       public static void main(String[] args) throws Exception
12       {
13           URL url = new URL(
14             "http://localhost:9081/TestRestCRUDWeb/rest/students/12");
15           HttpURLConnection conn =
16             (HttpURLConnection) url.openConnection();
17           conn.setRequestMethod("GET");
18           conn.setRequestProperty("Accept", "application/json");
19
20           if (conn.getResponseCode() != 200)
21           {
22               throw new RuntimeException("Failed : HTTP error code : "
23                       + conn.getResponseCode());
24           }
25
26           BufferedReader br = new BufferedReader(new InputStreamReader(
27                   (conn.getInputStream())));
28
29           String output;
30           System.out.println("Output from Server .... \n");
31           while ((output = br.readLine()) != null)
32           {
33               System.out.println(output);
34           }
35
36           conn.disconnect();
37
38       }
39   }
```

```
1    package client;
2
3    import java.io.BufferedReader;
4    import java.io.InputStreamReader;
5    import java.io.OutputStream;
6    import java.net.HttpURLConnection;
7    import java.net.URL;
8
9    public class PostClient
10   {
11       public static void main(String[] args) throws Exception
12       {
13           URL url =
14             new URL(
15              "http://localhost:9081/TestRestCRUDWeb/rest/students");
16           HttpURLConnection conn =
17             (HttpURLConnection) url.openConnection();
18           conn.setDoOutput(true);
19           conn.setRequestMethod("POST");
20           conn.setRequestProperty("Content-Type", "application/json");
21
22           String input = "{\"name\":\"Harry Wolfe\",\"gpa\":2.22}";
23
24           OutputStream os = conn.getOutputStream();
25           os.write(input.getBytes());
26           os.flush();
27
28           if (conn.getResponseCode() != HttpURLConnection.HTTP_CREATED)
29           {
30               throw new RuntimeException("Failed : HTTP error code : "
31                       + conn.getResponseCode());
32           }
33
34           BufferedReader br = new BufferedReader(new InputStreamReader(
35                   (conn.getInputStream())));
36
37           String output;
38           System.out.println("Output from Server .... \n");
39           while ((output = br.readLine()) != null)
40           {
41               System.out.println(output);
42           }
43
44           System.out.println("Location header: "
45                   + conn.getHeaderField("Location"));
```

```
46
47              conn.disconnect();
48          }
49      }
```

```
1    package client;
2
3    import java.io.OutputStream;
4    import java.net.HttpURLConnection;
5    import java.net.URL;
6
7    public class PutClient
8    {
9        public static void main(String[] args) throws Exception
10       {
11           URL url =
12             new URL(
13               "http://localhost:9081/TestRestCRUDWeb/rest/students");
14           HttpURLConnection conn =
15             (HttpURLConnection) url.openConnection();
16           conn.setDoOutput(true);
17           conn.setRequestMethod("PUT");
18           conn.setRequestProperty("Content-Type", "application/json");
19
20           String input =
21             "{\"name\":\"Joe Blow\",\"gpa\":4.56, \"studentId\":12}";
22
23           OutputStream os = conn.getOutputStream();
24           os.write(input.getBytes());
25           os.flush();
26
27           if (conn.getResponseCode() !=
28              HttpURLConnection.HTTP_NO_CONTENT)
29           {
30               throw new RuntimeException("Failed : HTTP error code : "
31                       + conn.getResponseCode());
32           }
33
34           conn.disconnect();
35       }
36   }
```

```
1    package client;
2
3    import java.net.HttpURLConnection;
4    import java.net.URL;
5
6    public class DeleteClient
7    {
8        public static void main(String[] args) throws Exception
9        {
10           URL url =
11             new URL(
12               "http://localhost:9081/TestRestCRUDWeb/rest/students/44");
13           HttpURLConnection conn =
14             (HttpURLConnection) url.openConnection();
15           conn.setRequestMethod("DELETE");
16
17           conn.connect();
18
19           if (conn.getResponseCode() != HttpURLConnection.HTTP_NO_CONTENT)
20           {
21               throw new RuntimeException("Failed : HTTP error code : "
22                       + conn.getResponseCode());
23           }
24
25           conn.disconnect();
26       }
27   }
```

Chapter Summary

In this chapter, you learned:

- The fundamentals of REST applications that create, retrieve, update and delete resources
- How to implement a RESTful CRUD application using JAX-RS

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring and REST

- Spring Support for REST
- Message Converters
- Annotations

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring Support for REST

- Starting in Spring 3, Spring makes it easy to write SpringMVC **controllers** that work with REST resources
- If you are writing your application with Spring, than using Spring REST is easier and more natural than using JAX-RS

JAX-RS is the JEE standard for working with RESTful resources. Spring REST is similar in many ways, but integrates with the Spring infrastructure better than JAX-RS.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Message Converters

- Spring uses **message converters** to convert Java objects to/from a **resource representation**, including the following:
  - **SpringHttpMessageConverter** - transforms String objects to/from *text/plain*
  - **Jaxb2RootElementHttpMessageConverter** - transforms JAXB-annotated objects to/from *application/xml*
  - **MappingJacksonHttpMessageConverter** - transforms object to/from *application/json*

There are several more message converters available - see:

http://docs.spring.io/spring/docs/3.1.x/spring-framework-reference/html/remoting.html#rest-message-c

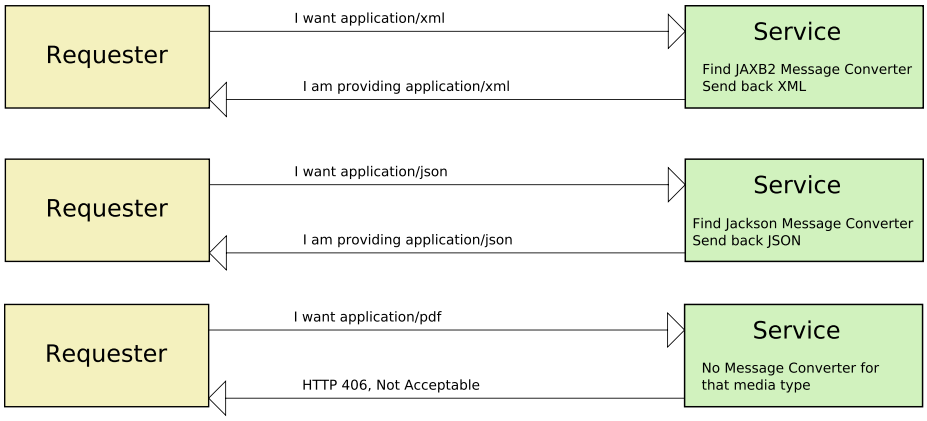
You must configure the JARs for JAXB2 and Jackson on the application's CLASSPATH to use those converters. Note that full JEE application servers like WebSphere include JAXB2 and might also include Jackson. If you're not running on a full JEE server, you can obtain Jackson from http://jackson.codehaus.org and JAXB2 from https://jaxb.java.net/.

Spring REST doesn't require you to use message converters. You could instead use the normal Spring MVC view-based infrastructure to send back JSON or XML to the client. Spring provides the ContentNegotiatingViewResolver for that purpose - see the Spring documentation for details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Content Negotiation

- The client and server must agree on a data **representation**, which is usually a standard format such as text, XML or JSON using a **MediaType**
- Ultimately, the service decides on which representations it will provide



Media types are also known as MIME types. When the client requests a particular media type, Spring looks for a registered message converter for that type and uses it to convert the service's data.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Negotiation via HTTP Accept Header

- The HTTP specification allows for an **Accept** header, in which the user agent client provide a list of media types
- Spring parses the Accept header and then finds a message converter that can work with that media type

```
 curl -H "Accept: application/xml, application/json"
  http://localhost:9080/TestSpringRESTWeb/rest/advisor/12

<!-- response -->
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<advisor>
  <id>12</id>
  <name>Sue Joens</name>
  <salary>50000.0</salary>
</advisor>
```


---

cURL is a popular and powerful tool for sending network requests - it's very useful for testing RESTful services. cURL is included with many operating systems, but not Windows. You can download it from;

`http://curl.haxx.se/download.html`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Spring REST Annotations

- Spring provides several annotations to configure controllers to work with REST:
  - **@RequestMapping** - assigns a URL and an HTTP method to a processing method
  - **@ResponsStatus** - sets the HTTP status code
  - **@RequestBody** - transforms REST request into Java object(s)
  - **@ResponseBody** - transforms Java object into a REST response
  - **@PathVariable** - initializes a method parameter from part of the request URL
  - **@RequestParam** - initializes a method parameter from an HTTP *request parameter*

Controllers can use other standard Spring MVC annotations such as @Valid, @Autowire and so forth.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Simple RESTful Controller

- This controller method returns either XML or JSON depending on the client's **Accept** header

```
1    @Controller
2    @RequestMapping("rest")
3    public class MyService
4    {
5      // respond to URLs like ..../rest/advisor/12
6      @RequestMapping(value = "/advisor/{id}",
7        method = RequestMethod.GET)
8      public @ResponseBody
9        Advisor getAdvisor(@PathVariable("id") int id)
10     {
11       Advisor a = findAdvisorByID(id);
12       if(a == null) throw new ResourceNotFoundException():
13       return a;
14     }
15   }
```

The ResourceNotFoundException class looks like:

```
@ResponseStatus( value = HttpStatus.NOT_FOUND )
public class ResourceNotFoundException extends RuntimeException{}
```

Note that for XML conversion to work, the class of the returned object must be annotated with JAXB:

```
@XmlRootElement
public class Advisor
{
. . .
}
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Responding to POST Requests

- POST requests cause the REST service to create the resource and assign its RESTful URL

```
1    @RequestMapping(value = "/advisor",
2      method=RequestMethod.POST)
3    @ResponseStatus(HttpStatus.CREATED)
4    public @ResponseBody Advisor createAdvisor(
5      @RequestBody Advisor a, HttpServletResponse response)
6    {
7        int newID = insertAdvisor(a);
8        a.id = newID;
9
10       response.setHeader("Location", "/advisor/" + newID);
11       return a;
12   }
```


curl -v -X POST -H "Content-type: application/json" -H "Accept: application/json" -d "{\"id\":0,\"name\":\"Sue Joens\",\"salary\":150000.0}" http://localhost:9080/TestSpringRESTWeb/rest/advisor

Return might look like:

```
* upload completely sent off: 45 out of 45 bytes
< HTTP/1.1 201 Created
< X-Powered-By: Servlet/3.0
< Location: /advisor/56
< Content-Type: application/json;charset=UTF-8
< Content-Language: en-US
< Transfer-Encoding: chunked
< Date: Thu, 12 Jun 2014 16:12:23 GMT
< Server: WebSphere Application Server/8.0
<
{"id":56,"name":"Sue Joens","salary":150000.0}*
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Responding to PUT Requests

- PUT requests cause the REST service to update the RESTful resource

```
1    @RequestMapping(value = "/advisor/{id}",
2      method = RequestMethod.PUT)
3    public ResponseEntity<String> putAdvisor(
4      @PathVariable("id") long id, @RequestBody Advisor a)
5    {
6      ResponseEntity<String> retVal =
7        new ResponseEntity<String>(HttpStatus.NO_CONTENT);
8
9        boolean ok = updateAdvisor(a);
10
11       if(!ok)
12         retVal = new
13           ResponseEntity<String>(HttpStatus.NOT_FOUND);
14
15       return retVal;
16   }
16 - 9
```

```
curl -v -X PUT -H "Content-Type: application/json" -H "Accept:application/json" -d
"{\"id\":12,\"name\":\"Sue\",\"salary\":345}"
http://localhost:9081/TestSpringRESTWeb/rest/advisor/12

Response:

* upload completely sent off: 35 out of 35 bytes
< HTTP/1.1 204 No Content
< X-Powered-By: Servlet/3.0
< Content-Language: en-US
< Content-Length: 0
< Date: Thu, 12 Jun 2014 16:23:17 GMT
< Server: WebSphere Application Server/8.0
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a Spring REST Application

- You configure a RESTful Spring application in the same fashion as for a "normal" Spring MVC application

```
1    <!-- springmvc-servlet.xml -->
2    <beans . . .>
3
4      <context:component-scan
5         base-package="com.goliath" />
6
7      <mvc:annotation-driven />
8    . . .
9    </beans>
```

MyConfig.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package com.goliath.config;
2
3    import org.springframework.context.annotation.ComponentScan;
4    import org.springframework.context.annotation.Configuration;
5    import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConf
6    import org.springframework.web.servlet.config.annotation.EnableWebMvc;
7    import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
8
9    @Configuration
10   @EnableWebMvc
11   @ComponentScan("com.goliath.services")
12   public class MyConfig extends WebMvcConfigurerAdapter
13   {
14       @Override
15       public void configureDefaultServletHandling(
16               DefaultServletHandlerConfigurer configurer)
17       {
18           configurer.enable();
19       }
20   }
```

Chapter Summary

In this chapter, you learned:

- How to use Spring MVC controllers, annotations and message converters to write RESTful CRUD services in Spring.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum