WA2511 Spring Boot

Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-877-812-8887
Web: http://www.webagesolutions.com

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Web Age Solutions Inc.
821A Bloor Street West
Toronto
Ontario, M6G 1M1

Table of Contents

Chapter 1 - Spring Framework Configuration

Objectives

Key objectives of this chapter

- Cover ways to configure Spring components in Spring Boot:
  - Java @Configuration classes
  - Spring Expression Language
  - External Resource Bundles
  - Spring Property Editors and Converters

1.1 Java @Configuration Classes

- Often using source code annotations and XML configuration are not sufficient to configure Spring applications
  - This is especially true when annotations can't be added to the classes being used and configuring only with XML is difficult
  - With Spring Boot, we don't typically use XML configuration ('spring-beans.xml')
- We can write Java code to initialize components
  - This can also be true if there is an existing code infrastructure for the "Factory" pattern
- To link this Java initialization code into the rest of Spring configuration the Spring **@Configuration** and **@Bean** annotation can be added to Java classes that provide this initialization logic
  - This initialization code can be used where needed and hooked into the rest of the Spring application which can be configured with the other mechanisms
- Spring components will also look for classes that implement a particular interface, and call initializer methods on those classes
  - e.g. WebApplicationInitializer for Spring-MVC apps

1.2 Defining @Configuration Classes

- Any Java class with the **@Configuration** annotation indicates to the Spring container that the class can be used as a source of bean definitions
- Methods of a **@Configuration** class are annotated with the **@Bean** annotation to indicate that the methods returns a Spring bean
  - The return type of the method is critical as this will be the type of bean that Spring will use to compare against injection into other components
- One way to understand how the @Bean method works is to compare how the equivalent XML configuration would be done

```
@Configuration
public class AppConfig {
	@Bean
	public MyService myService() {
		return new MyServiceImpl();
	}
}
```

  - Would be similar to this XML configuration:

```
<beans>
	<bean id="myService"
class="com.acme.services.MyServiceImpl"/>
</beans>
```

Defining @Configuration Classes

The simple example above does not really show much of the benefit of using @Configuration classes. The benefit is much greater when other Java code performs initialization of the object before the object is returned from the @Bean method.

1.3 Defining @Configuration Classes

- In order to use @Configuration classes you must have the 'cglib' code generation library as part of the project
  - This library dynamically adds the feature of turning @Configuration classes into Spring bean definitions at startup time
- Since 'cglib' will sub-class your @Configuration classes this places a few

minor restrictions on the class definition used for @Configuration classes:

- @Configuration classes may not be *final*
- @Configuration classes must have a no-argument constructor

1.4 Loading @Configuration Classes

- A @Configuration class does not contribute to the Spring configuration unless it is visible to Spring
- In a plain Spring application, you can define your @Configuration class as a bean in the xml config
  - Could also configure component scan in the xml config
  - Put @ComponentScan on a @Configuration class would do the same thing
- Spring Boot is typically configured to scan for components in the classpath
  - @SpringBootApplication on the main class is equivalent to @Configuration, @EnableAutoConfiguration and @ComponentScan

Loading @Configuration Classes

There is also a 'scan' method that can be used from Java code to replicate the behavior of the <context:component-scan> element from XML configuration.

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.scan("com.acme.config");
```

1.5 Modularizing @Configuration Classes

- Although it would be possible to provide all bean definitions in a single @Configuration class, multiple classes are often used to provide the modularity of configuration that resembles the rest of the application
- You could use the @Import annotation to import one configuration class into another

```
@Configuration
public class ConfigA {
	// @Bean methods
```

```
}
@Configuration
@Import(ConfigA.class)
public class ConfigB { ... }
```

- One weakness of this approach though is one @Configuration class has to refer directly to the other @Configuration class

- You can also use @ImportResource to import XML from Java @Configuration classes

```
@Configuration
@ImportResource("classpath:/com/acme/other-config.xml")
public class AppConfig { .. }
```

1.6 Modularizing @Configuration Classes

- Any class with a @Configuration annotation also has a @Component annotation added by Spring
  - This means you could even inject the configuration classes themselves into each other
  - These injected configuration classes could even implement interfaces

```
@Configuration
public class ServiceConfig {
  @Autowired private RepositoryConfig repositoryConfig;
  public @Bean TransferService transferService() {
    return new TransferServiceImpl(
      repositoryConfig.accountRepository());
  }
}
@Configuration
public interface RepositoryConfig {
  @Bean AccountRepository accountRepository();
}
@Configuration
public class DefaultRepositoryConfig implements
RepositoryConfig {
  @Bean public AccountRepository accountRepository() { .. }
}
```

Modularizing @Configuration Classes

One benefit of the approach above is that you can directly call the methods of the injected configuration class to return a bean of the desired type. Since you are writing Java code anyway this may be more intuitive than having Spring use the @Autowired annotation to inject a component as a field that is then used elsewhere in component initialization, as shown below.

```
@Configuration
public class ServiceConfig {
  private @Autowired AccountRepository accountRepository;

  public @Bean TransferService transferService() {
      return new TransferServiceImpl(accountRepository);
  }
}

@Configuration
public class RepositoryConfig {
  public @Bean AccountRepository accountRepository() { .. }
}
```

1.7 Qualifying @Bean Methods

- By default the name of a @Bean method is the name of the bean defined

```
@Bean public MyService myService() { // 'myService' bean
```

- You can also supply one or more bean names with the @Bean annotation to override the default bean name

```
@Bean(name={"myService","myOtherBeanName"})
```

- You can change the scope of the bean definition by also adding a @Scope annotation along with the @Bean annotation to the method
  - If no @Scope is present the Spring "singleton" default scope is used

```
@Bean
@Scope("prototype")
public MyService myService() { .. }
```

- You can also use @Primary to indicate this would be the primary bean to inject for dependency injection

```
@Bean  @Primary
public MyService myService() { .. }
```

- Finally, you can use any custom qualifier annotation or the Spring @Qualifier annotation on a @Bean method to indicate which qualifiers would match the bean definition

```
@Bean  @StrategyType(strategy=GenerationStrategy.RANDOM)
public IntGenerator randomGenerator() { .. }
```

Qualifying @Bean Methods

If a bean has more than one bean name they are typically referred to as bean "aliases".

1.8 Trouble with Prototype Scope

- If any Spring bean defined at "prototype" scope is used in a @Configuration class care must be taken to ensure proper behavior
- If you use @Autowired on a field of a @Configuration class that field will only be initialized once and using it more than once will use the same object
  - This is not what is desired with prototype scope

```
@Configuration
public class AppConfig {
       @Autowired
       private SomePrototypeBean prototypeBean;

       @Bean public OtherBean1 otherBean1 () {
              // this uses the injected instance
              return new OtherBean1(prototypeBean);
       }

       @Bean public OtherBean2 otherBean2 () {
              // this uses the SAME instance
              return new OtherBean2(prototypeBean);
       }
```

1.9 Trouble with Prototype Scope

- If using a prototype bean that is declared in a @Configuration class the best way is to inject the @Configuration class itself and call the @Bean method directly
  - This continues to work also if the bean is a singleton as Spring will return the proper bean from the @Bean method depending on the

behavior indicated by the scope

```
@Configuration
public class AppConfig {
  @Autowired private OtherConfig otherConfig;

  @Bean public OtherBean1 otherBean1 () {
       return new OtherBean1(
            otherConfig.producePrototypeBean());
```

- If the prototype bean you need to use has been defined in XML it is a little more tricky as you need to use the @Scope("prototype") annotation on the @Configuration class itself so that the @Configuration class is created every time and has the @Autowired injection performed again

```
@Configuration
@Scope("prototype")
public class AppConfig {
       @Autowired
       private SomePrototypeXMLBean prototypeXMLBean;
```

1.10 Configuration with Spring Expression Language

- One of the big changes in Spring 3.0 was the introduction of the 'Spring Expression Language' (SpEL)
- One of the more useful aspects of the Spring EL is using it in XML configuration files to make them more dynamic
  - You can set the value of one Spring component based on the property value of another

```
<bean id=".." class=".." >
       <property name="serverVendor"
       value="#{environmentBean.serverType}" />
</bean>
```

  - You can also use the predefined variable 'systemProperties' to pull in values of environment variables using Spring EL

```
<bean id=".." class="..">
       <property name="defaultLocale"
       value="#{ systemProperties['user.region'] }"/>
</bean>
```

- You can also use the @Value annotation within Java code for a field

```
@Value("#{ systemProperties['user.region'] }")
private String defaultLocale;
```

Configuration with Spring Expression Language

This allowed Spring to define and use it's own expression language instead of relying on an external implementation.

The Spring documentation focuses a lot on how to write Java code to use the Spring expression language to evaluate expressions. This will not be commonly done in Java code by end users

In the example above the 'environmentBean' could be a bean that retrieves various configuration parameters from some mechanism and then exposes them through bean properties. You could have several beans configured with the properties of this bean by using the Spring EL.

Using the @Value annotation makes the initialization of a component property more externally configurable with a dynamic value even though it is placed within the Java code.

The @Value annotation could be used on a component field or a 'setter' method.

1.11 Resolving Text Messages

- Sometimes configuration of an application might depend on the language or locale of the environment the application is run in
  - Java has a great Internationalization framework for message bundles but how to use this within a Spring application?
- The Spring ApplicationContext can resolve messages based on locale
- To use this you could first declare a 'ResourceBundleMessageSource' bean on a configuration bean

```
@Configuration
public class SpringContext {
	@Bean
	@Qualifier("myMessages")
	public MessageSource getMessageSource() {
		ResourceBundleMessageSource rbms=new
			ResourceBundleMessageSource();
		rbms.setBasename("Messages");
		return rbms;
	}
```

```
}
```

1.12 Resolving Text Messages

- You could then inject a 'MessageSource' into a @Configuration class and use the 'getMessage' method

```
@Configuration
public class ProductCodeConfiguration {
       @Autowired
       @Qualifier("myMessages")
       private MessageSource messageSource;

       @Bean
       public ProductCodeGenerator prodCodeGenerator() {
              String prefix = messageSource.
              getMessage("defaultPrefix", null, null);
              // use this to configure a bean
       .. }
```

1.13 Spring Property Conversion

- Spring has two main ways to convert property values from XML configuration into Java types
  - Spring Converter implementations
  - JavaBean PropertyEditors
- For configuration both mechanisms are primarily used to convert Strings read from XML configuration files into other Java types for Spring bean properties
- For the following examples we will show converting a String into a custom Java type like 'PersonalSSN'

```
convert "XXX-XX-XXXX" into:
public class PersonalSSN {
       private int area;
       private int group;
       private int serial;
... }
```

Spring Property Conversion

It has been observed that both property conversion mechanisms in Spring can have a problem with behavior when you would expect the conversion to happen more than once. One example of this is if a Spring bean is defined at prototype scope and a property of the bean is converted from a String. It appears that the Spring property conversion is not triggered for any but the creation of the first prototype bean. It has been observed that if the same String is encountered that Spring has already converted it will simply use the same instance from the first conversion instead of creating a new instance. This would cause problems with prototype beans since the properties of two DIFFERENT instances of the prototype bean could be sharing an object set as the property of the bean and therefore sharing state BETWEEN THE BEANS.

1.14 Spring Converter Interface

- The easiest way to implement conversion is with the new Spring 3 type conversion system
- To provide a custom Spring converter you implement the *Converter<S, T>* interface
  - This defines a way to convert from the 'Source' class 'S' to the 'Target' class 'T'
    - The use of Java generics makes the 'Converter' interface very self-documenting
  - This only requires the implementation of the 'convert' method

```
public class PersonalSSNConverter
       implements Converter<String, PersonalSSN> {
       public PersonalSSN convert(String number) {
              ...
       }
```

  - A runtime 'IllegalArgumentException' should be thrown to indicate conversion errors

1.15 Using Custom Converters

- To register a custom converter you would register it with the `DefaultConversionService` and provide it with a unique qualifier

```
@Bean
```

```
	@Qualifier("ssnConversion")
	public ConversionService conversionService() {
		DefaultConversionService service = new
DefaultConversionService();
		service.addConverter(new
PersonalSSNConverter());
		return service;
	}
```

- Besides Spring now knowing how to use your custom converter when needed, you can inject the conversion service and use it directly

```
public class MyServiceImpl implements MyService {
	@Autowired
	@Qualifier("ssnConversion")
	private ConversionService converter;

	public void calculateBenefits(String ssn) {
		PersonalSSN ssn = converter.convert(ssn,
PersonalSSN.class);
		...
	}
}
```

1.16 Using Custom Converters

- Alternatively in Spring Boot, instead of explicitly registering the converter with the `ConversionService`, you can annotate it with `@Component` and `@ConfigurationPropertiesBinding`

```
@Component
@ConfigurationPropertiesBinding
public class PersonalSSNConverter implements
Converter<String, PersonalSSN> { … }
```

- You would then remove the @Qualifier annotation from the ConversionService reference in your service, since you're no longer defining a conversionService bean

```
public class PersonalSSNServiceImpl implements
PersonalSSNService {
	@Autowired
```

```
        private ConversionService converter;
```

1.17 Spring PropertyEditors

- Another way to implement conversion is to provide a PropertyEditor which extends the JavaBean PropertyEditorSupport class
- You would then override the 'setAsText' method to indicate how to convert from a String to the custom property type
  - The key of this method is to call the 'setValue' method at the end from the base class passing in the object that was constructed by converting from the String

```
public class SSNEditor extends PropertyEditorSupport {
        public void setAsText(String toConvert) {
                int area = // parse area
                int group = // parse group
                int serial = // parse serial
                PersonalSSN ssn =
                        new PersonalSSN(area, group, serial);
                setValue(ssn);
        }
}
```

Spring PropertyEditors

Although the PropertyEditorSupport class also has a 'getAsText' method that you could override, the most common way a property editor is used is converting from the String in an XML configuration file to the custom property type of a bean class. This means the 'setAsText' method is used much more commonly.

1.18 Registering Custom PropertyEditors

- In order for a custom PropertyEditor to be used it must be registered to Spring
  - The best way to do this is to have another class that implements the 'PropertyEditorRegistrar' interface and will register the PropertyEditor with Java code

```
public class SSNEditorRegistrar implements
```

```
PropertyEditorRegistrar {
        public void registerCustomEditors(
                PropertyEditorRegistry registry) {
                registry.registerCustomEditor(
                        PersonalSSN.class, new SSNEditor());
        }
}
```

- You would then configure the Spring 'CustomEditorConfigurer' with this PropertyEditorRegistrar
  - Note: This is a case where it actually is a little more direct to use xml. Import the file with @ImportResource

```
<bean class="org.springframework.
        beans.factory.config.CustomEditorConfigurer">
        <property name="propertyEditorRegistrars">
                <list><bean
class="example.SSNEditorRegistrar"/>
                </list>
        </property>
</bean>
```

Registering Custom PropertyEditors

You can register more than one PropertyEditor in a PropertyEditorRegistrar.

Although it would also be possible to directly register the custom property editor with the following property on this bean, using the Spring Property Editor Registrar method is better as it is more thread safe.

```
<bean class="org.springframework.
        beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
                <map>
                        <entry key="com.webage.bean.ComboIntType"
                        value="com.webage.editors.ComboIntPropertyEditor" />
                </map>
        </property>
</bean>
```

1.19 Summary

- There are many alternate mechanisms that can be used in Spring configuration

- Spring Boot in particular favors Java-based configuration
- The use of @Configuration classes can help when it is easier to initialize Spring components with direct Java code
- Spring has a few mechanisms for property conversion

Chapter 2 - Introduction to Spring Boot

Objectives
Key objectives of this chapter ■ Overview of Spring Boot ■ Using Spring Boot for building microservices ■ Examples of using Spring Boot

2.1 What is Spring Boot?

- Spring Boot (*http://projects.spring.io/spring-boot/)* is a project within the Spring IO Platform (*https://spring.io/platform*)
- Developed in response to Spring Platform Request SPR-9888 "Improved support for 'containerless' web application architectures"
- Inspired by the DropWizard Java framework *(http://www.dropwizard.io/)*
- The main focus of Spring Boot is on facilitating a fast-path creation of stand-alone web applications packaged as executable JAR files with minimum configuration
- An excellent choice for creating microservices
- Released for general availability in April 2014
  - (*https://spring.io/blog/2014/04/01/spring-boot-1-0-ga-released*)
- Version 2 was released in Spring of 2017

Notes:

A JAR (Java ARchive) is a file format and deployment unit used to package a set of Java class files and associated metadata and resources.

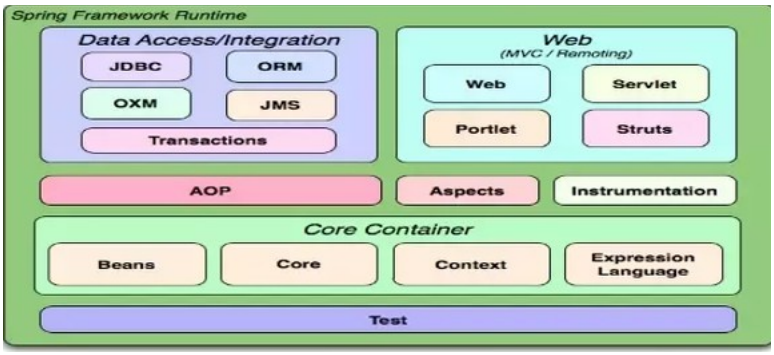
Microservices are a collection of loosely coupled services used by an application. The services are fine-grained and the protocols are lightweight (e.g., HTTP). Services can be developed and deployed independently. Each service runs in its own process.

2.2 Spring Framework

- Simply put, the Spring framework provides comprehensive support for

developing Java applications.

- It lets you utilize features like Dependency Injection and comes with out of the box modules like:
  - Spring JDBC
  - Spring MVC
  - Spring Security
  - Spring AOP
  - Spring ORM
  - Spring Test
- These modules can drastically reduce the development time of an application.



2.3 How is Spring Boot Related to Spring Framework?

- Unlike Spring, Spring Boot is not a framework. Instead, it's an extension of Spring Framework.
- It is a way to easily create stand-alone applications with minimal or zero configurations.
- It provides boilerplate code and annotation configuration to quick start new spring projects.
- It provides a set of starter pom or gradle build files which one can use to add required dependencies and also facilitate autoconfiguration.
- It provides 'starter' dependencies to simplify build and application configuration

- It provides Embedded HTTP servers like Tomcat, Jetty, etc. to develop and test our web applications very easily.
- Spring Boot provides lots of plugins to develop and test Spring Boot Applications very easily using build tools like Maven and Gradle

2.4 Spring Boot 2

- Spring Boot is currently at version 2.2.6
- Spring Boot 2 extends Spring Framework 5
- Spring Boot 2 requires Java 8 minimum
- Breaking changes in security configuration
  - No more default username and password
  - Everything is now secured by default, including static resources and endpoints
- Support for reactive programming
- Actuator endpoints are now independent of Spring MVC
  - Custom endpoints can be created
  - Predefined endpoints are no longer configurable
- Spring Boot DAO API are now in line with JPA APIs
  - .findOne is now called .findById
  - .findById returns an Optional<T> instead of T

2.5 Spring Boot Main Features

- Spring Boot offers web developers the following features:
  - Ability to create WAR-less stand-alone web applications that you can run from command line
  - Embedded web container that is bootstrapped from the **`public static void main`** method of your web application module:
    - Tomcat Servlet container is default; you have options to plug in Jetty

or Undertow containers instead

- No, or minimal configuration
  - Spring Boot relies on Spring MVC annotations (more on annotations later ...) for configuration
- Built-in production-ready features for run-time metrics collection, health checks, and externalized configuration

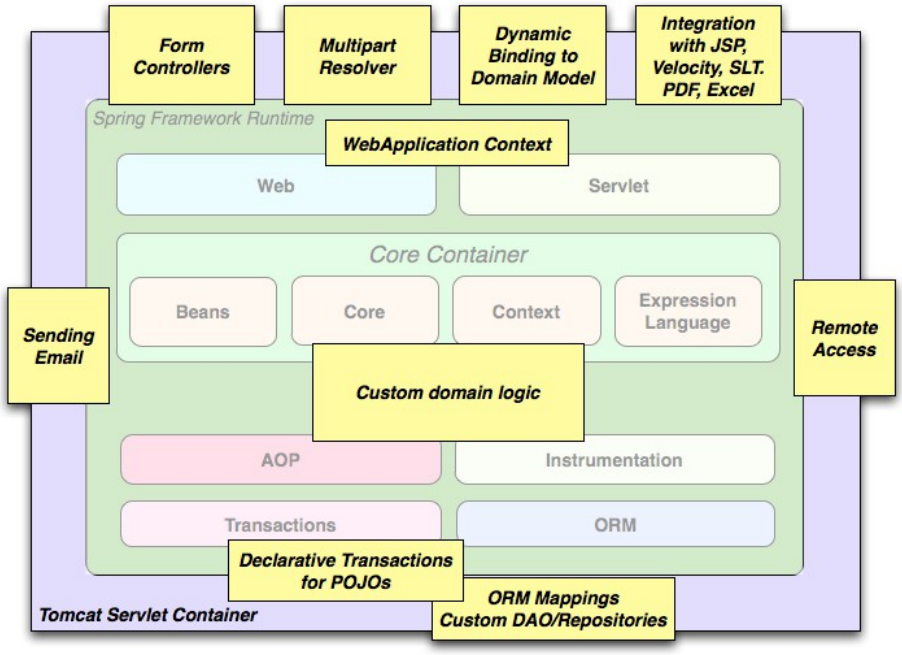
Notes:

A WAR (Web application ARchive) file is a format for and unit of packaging and distribution of Java-based web applications that are deployed in the web container of a Java-enabled web server.

For steps how to switch from Tomcat to Jetty or Undertow, visit *http://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html*

For an overview of the Spring framework, visit *http://docs.spring.io/spring/docs/current/spring-framework-reference/html/overview.html*

The following diagram, borrowed from the above Spring framework link, lays out the components of a full-fledged Spring web application:



2.6 Spring Boot on the PaaS

- Spring boot is designed to be conveniently run in a container. All major cloud provider can run Docker and Kubernetes
- Older projects might use PaaS (Platform as a Service)
- PaaS offer robust, scalable, and cost-efficient run-time environments that can be used with minimum operational involvement
- Spring Boot is being financed by Pivotal Software Inc. which also, in partnership with VMware, sponsors Cloud Foundry
  - While Cloud Foundry is available for free, Pivotal offers a commercial version of it called Pivotal Web Services

2.7 Understanding Java Annotations

- Annotations in Java are syntactic metadata that is baked right into Java source code; you can annotate Java classes, methods, variables and parameters
- Basically, an annotation is a kind of label that is processed during a compilation stage by annotation processors when the code or configuration associated with the annotation is injected in the resulting Java class file, or some additional operations associated with the annotation are performed
- An annotation name is prefixed with an '@' character
- **Note:** Should you require to change annotation-based configuration in your application, you would need to do it at source level and then re-build your application from scratch

2.8 Spring MVC Annotations

- Spring Boot leverages Spring MVC (Model/View/Controller) annotations instead of XML-based configuration
  - Additional REST annotations, like *@RestController* have been added with Spring 4.0

- The main Spring MVC annotations are:
  - **@Controller / @RestController** – annotate a Java class as an HTTP end-point
  - **@RequestMapping** – annotate a method to configure with URL path / HTTP verb the method responds to
  - **@RequestParam** - named request parameter extracted from client HTTP request

2.9 Example of Spring MVC-based RESTful Web Service

- The following code snippet shows some of the more important artifacts of Spring MVC annotations (with REST-related Spring 4.0 extensions) that you can apply to your Java class
- **Note:** Additional steps to provision and configure a web container to run this module on are required

```
// … required imports are omitted
@RestController
public class EchoController {
    @RequestMapping(path="/echoservice", method=RequestMethod.GET)
    public String echoback(@RequestParam(value="id") String echo) {
        return echo;
    }
}
```

- The above annotated Java class, when deployed as a Spring MVC module, will echo back any *echo* message send with an HTTP GET request to this URL:

```
http(s)://<Deployment-specific>/echoservice?id=hey
```

2.10 Spring Booting Your RESTful Web Service

- Spring Boot allows you to build production-grade web application without the hassle of provisioning, setting up, and configuring the web container
- A Spring Boot application is a regular executable JAR file that includes the required infrastructure components and your compiled class that must have the **`public static void main`** method which is called by the Java VM on application submission

- The **`public static void main`** method references the **`SpringApplication.run`** method that loads and activates Spring annotation processors, provisions the default Tomcat Servlet container and deploys the Spring Boot-annotated modules on it

2.11 Spring Boot Skeletal Application Example

- The following code is a complete Spring Boot application based on the Spring MVC REST controller from a couple of slides back

```
// … required imports are omitted
@SpringBootApplication
@RestController
public class EchoController {
    @RequestMapping(path="/echoservice", method=RequestMethod.GET)
    public String echoback(@RequestParam(value="id") String echo) {
        return echo;
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(EchoController.class, args);
    }
}
```

- You run Spring Boot applications from command line as a regular executable JAR file:

```
java -jar Your_Spring_Boot_App.jar
```

- The default port the Spring Boot embedded web container starts listening on is 8080
  - To change the default port, you need to pass a **server.port** System property or specify it in the Spring's **application.properties** file

Notes:

The **@SpringBootApplication** is functionally equivalent to three annotations applied sequentially:

@Configuration, @EnableAutoConfiguration, and **@ComponentScan**

2.12 Converting a Spring Boot Application to a WAR File

- In some scenarios, users want to have a Sprint Boot runnable JAR file

converted into a WAR file

- For those situations, Spring Boot provides two plug-ins:
  - **spring-boot-gradle-plugin** for the Gradle build system (*https://gradle.org/*)
  - **spring-boot-maven-plugin** for the Maven build system (*https://maven.apache.org/*)
- For more details on Spring Boot JAR to WAR conversion, visit *https://spring.io/guides/gs/convert-jar-to-war/*

2.13 Externalized Configuration

- Lets you deploy the same Spring Boot artifact (jar file) in different environments—configuration is drawn from environment variables
- Properties are pulled from (note - this is an incomplete list)
  - OS Environment variable SPRING_APPLICATION_JSON
  - Java System properties
  - OS Environment variables
  - Application property files - application.properties or application.yml
    - in a '/config' folder in current directory
    - in the current directory
    - in a classpath '/config' package
    - in the classpath root
    - In a folder pointed to by 'spring.config.location' on the command line.

2.14 Starters

- Spring Boot auto-configures based on what it finds on the classpath
- So, the set of modules is determined by what's in the 'pom.xml'
- The project provides a number of 'starter' artifacts that pull in the correct

dependencies for a given technology

- Some examples:
  - spring-boot-starter-web
  - spring-boot-starter-jdbc
  - spring-boot-starter-amqp
- You just need to include these artifacts in the 'pom.xml' to configure those technologies.

2.15 Maven - The 'pom.xml' File

- Assuming you're building with Apache Maven, the 'pom.xml' file describes all the artifacts and build tools that go into producing a delivered artifact.
- Generally, use the 'starter parent':

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.6.RELEASE</version>
</parent>
```

  - It defines all the dependency management and core dependencies for a Spring Boot application

2.16 Maven - The 'pom.xml' File

- Add dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
```

```
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
     …
    </dependency>
  </dependencies>
```

2.17 Spring Boot Maven Plugin

- The Maven plugin can package the project as an executable jar file, that includes all the dependencies

```
<build>
  <plugins>
    <plugin>
      <groupId>
        org.springframework.boot
      </groupId>
      <artifactId>
        spring-boot-maven-plugin
      </artifactId>
    </plugin>
  </plugins>
</build>
```

2.18 Gradle - The 'build.gradle' File

- To define the dependencies, use the dependencies block as shown below:

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-
starter-web'
  testImplementation('org.springframework.boot:spring-boot-
starter-test')
  ...
}
```

- It defines the dependency management and core dependencies for a

Spring Boot application

2.19 Spring Boot Maven Plugin

- Spring provides a standalone Spring Boot Gradle plugin which adds some tasks and configurations to ease the work with Spring Boot based projects.
- Assuming you're building with Gradle, the 'build.gradle' file describes all the artifacts and build tools that go into producing a delivered artifact.

```
plugins {
  id 'org.springframework.boot' version
'2.2.6.RELEASE'
  id 'io.spring.dependency-management' version
'1.0.9.RELEASE'
  id 'java'
}
```

2.20 HOWTO: Create a Spring Boot Application

- Create a new Maven project in the IDE of your choice
- In 'pom.xml',
  - add the Spring Boot parent project (see above)
  - Add dependencies to the 'starter' projects that describe the features you want (e.g. Spring MVC, JDBC, etc)
  - Add the 'spring-boot-maven-plugin' to the build plugins
- Add a default 'application.properties' or 'application.yml' file in 'src/main/resources'
  - List of properties is at:
    - https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html
    - At least include 'spring.application.name'

2.21 HOWTO: Create a Spring Boot Application

- Create a main class in src/main/java/<package>
- e.g. in 'com.mycom.app',

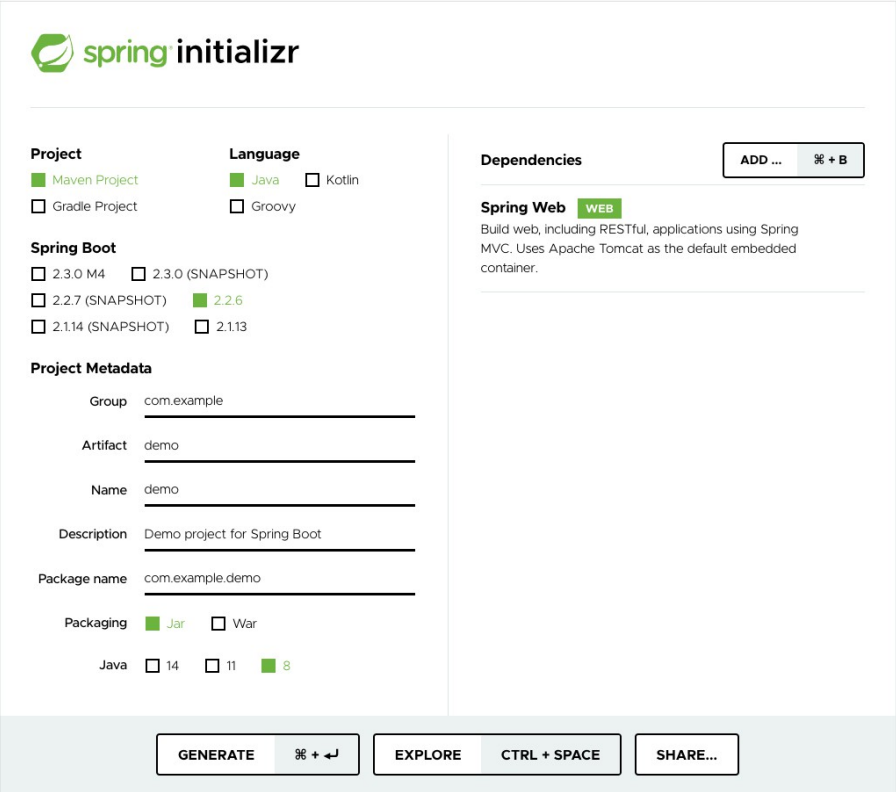
```
@SpringBootApplication
public class MyApp {
   public static void main(String[] args) {
     SpringApplication.run(MyApp.class, args);
   }
}
```

- Build with 'mvn install'
- From the IDE, run the main class
- From command line, run the jar file

2.22 Spring Initializr

- Spring also offers an Initializr website which can be used to create a new Spring Boot project
  - https://start.spring.io/
- Use it to configure whether you want to use Maven or Gradle, the Spring Boot version, language, project metadata, Java version, packaging format and dependencies.
- Preview the Gradle or Maven build file using the **EXPLORE** button.
- When done, click **GENERATE** button to download a zip file.
- Import the zip file into your favorite IDE.

2.23 Spring Initializr



2.24 Summary

- Spring Boot eliminates many of the headaches related to provisioning, setting up and configuring a web server by offering a framework for running WAR-less web applications using embedded web containers
- Spring Boot favors annotation-based configuration over XML-based one
  - Any change to such a configuration (e.g. a change to the path a REST-enabled method responds to), would require changes at source level and recompilation of the project
- Spring Boot leverages much of the work done in Spring MVC

Chapter 3 - Spring Web MVC

Objectives

Key objectives of this chapter:

- Overview of Spring Web MVC
- Mapping Web Requests
- Controller Handler Methods
- View Resolution

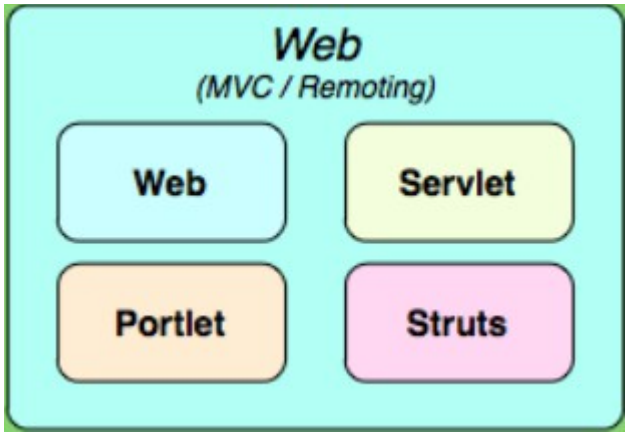
3.1 Spring Web MVC

- Spring provides an MVC (Model/View/Controller) web application framework
- Also known as Spring MVC
- Addresses state management, workflow, validation, etc.
- Spring Web MVC framework is modular, allowing various components to be changed easily

Spring Web MVC

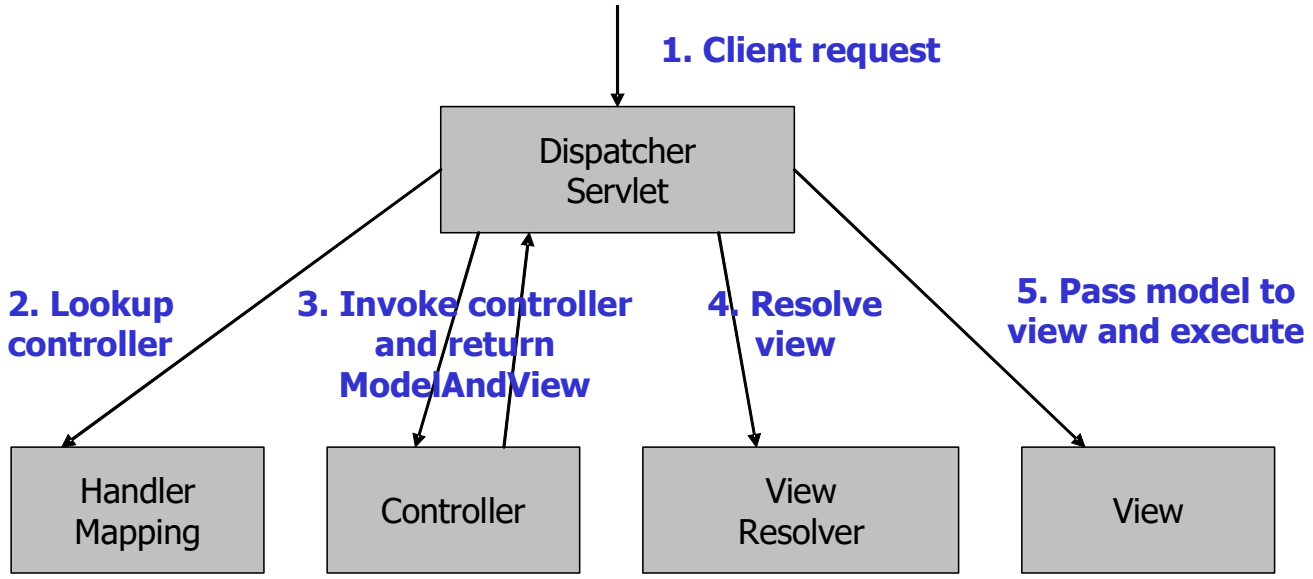
Spring provides an MVC web application framework known as both Spring Web MVC and Spring MVC. It is similar to other leading web application frameworks such as Struts. However, Spring Web MVC is more modular and less intrusive than other web frameworks.

3.2 Spring Web Modules



- The Spring 'Web' module has core web support classes
  - It is required by other modules
- The 'Servlet' module has Spring Web MVC and other Servlet support classes
- There are also 'Portlet' and 'Struts' modules for integrating those types of web applications with Spring

3.3 Spring Web MVC Components



Spring Web MVC Components

The diagram on the slide depicts the high-level workflow for handling a web request. It illustrates the major Spring Web MVC components involved in handling a request.

It starts with a client request, typically from a web browser. (1) The DispatcherServlet is first to receive this request. (2) The dispatcher consults one or more HandlerMappings to determine which

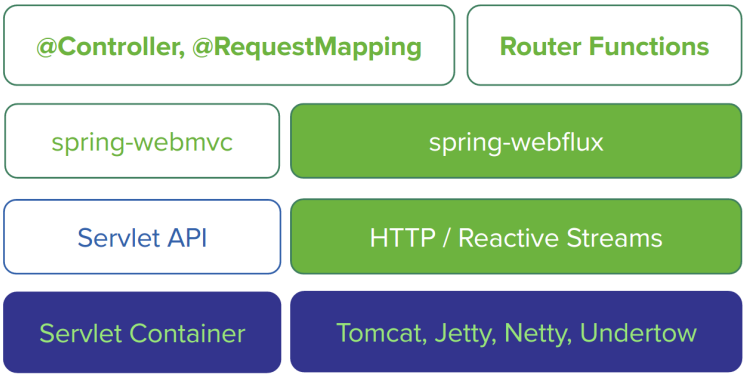
Controller should handle the request. (3) The dispatcher invokes the Controller to handle the request and the controller returns a ModelAndView object. A ModelAndView contains model information for the next view to display, as well as the logical name of the next view to display. (4) The dispatcher then uses a ViewResolver to map the logical view name to a View object. (5) Finally, the dispatcher passes the model of data to the View object. The view object uses the data to present its view back to the client.

3.4 DispatcherServlet

- Spring Boot automatically sets up a DispatcherServlet that:
  - recognizes the @Controller and @RequestMapping annotations
  - Automatically sets up a ContentNegotiatingViewResolver and a BeanNameViewResolver
  - includes HttpMessageConverters that can translate objects to JSON or XML
  - serves status content from the '/static', '/public', '/resources' or '/META-INF/resources' folders in the classpath
  - serves Webjars content from the /webjars/** folder in the classpath

3.5 Spring WebFlux Module

- Spring Framework 5 includes a new spring-webflux module.
- The module contains support for reactive HTTP and WebSocket clients as well as for reactive server web applications including REST, HTML browser, and WebSocket style interactions.



3.6 Spring WebFlux

- WebFlux includes a functional, reactive WebClient that offers a fully non-blocking and reactive alternative to the RestTemplate.
- It exposes network input and output as a reactive ClientHttpRequest and ClientHttpResponse where the body of the request and response is a Flux<DataBuffer> rather than an InputStream and OutputStream.
- In addition it supports the same reactive JSON and XML serialization mechanism as on the server side so you can work with typed objects.
- Example of WebClient

```
WebClient client = WebClient.create("http://example.com");

Mono<Account> account = client.get()
              .url("/accounts/{id}", 1L)
              .accept(APPLICATION_JSON)
              .exchange(request)
              .then(response ->
response.bodyToMono(Account.class));
```

Spring WebFlux

Reactive programming is a style of programming that's asynchronous, non-blocking, and event-driven. It involves modeling data and events as observable data streams and implementing observers to react to those changes in streams.

Spring WebFlux uses Project Reactor and its publisher implementations - Flux and Mono.

To use Spring WebFlux, you need to add the spring-boot-starter-webflux dependency to your project.

Spring WebFlux supports the same annotations as Spring MVC.

3.7 Template Engines

- In addition to REST services, you can use Spring Web MVC to serve dynamic HTML content using one of several template engines:
  - Thymeleaf

- FreeMarker
- Groovy
- Mustache

- Call out the 'starter' dependency for the template engine you want
  - e.g. spring-boot-starter-thymeleaf
- If you package your Spring Boot application as a war file, you can use JSP's but it's best to avoid them, so you can run Spring Boot applications in a stand-alone way.

Template Engines

If you use JSPs instead of one of the above templating technologies, there are several limitations:

1. JSPs are not supported when using an executable JAR.

Note: When you package your Spring Boot application as a WAR file with Jetty or Tomcat, it will work when deployed to a standard container or when launched with java -jar.

2. Undertow does not support JSPs.

3. Creating a custom error.jsp page does not override the default view for error handling. Custom error pages should be used instead.

3.8 Spring Boot MVC Example

- To demonstrate each part of a Spring Web MVC web application, we will develop an application that displays "Hello World"
- The steps are:
  - Call out the Spring Boot Web starter
  - Develop the controller
  - Configure Spring Web MVC controllers
  - Configure a template engine
  - Develop the view definition

Spring Web MVC Example

These steps assume that you have already configured your DispatcherServlet and context loader.

In the remainder of this section we will develop a simple web application to display "Hello World" or some other configurable message.

3.9 Spring Boot MVC Example

- Call out the Spring Boot Web Starter, by adding the following to the 'dependencies' in 'pom.xml'

```
            <dependency>

      <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-
web</artifactId>
            </dependency>
```

3.10 Spring Boot MVC Example

- Develop the controller

```
@Controller
public class HelloController  {
       @RequestMapping("/hello")
  public ModelAndView sayHello(
      @RequestParam("userMessage") String message) {
      return new ModelAndView(
        "helloView", "helloMessage", message);
  }
}
```

- The method above is invoked by the request:

```
localhost:8080/hello?userMessage=Howdy
```

- Configure Spring Web MVC to pick up controllers
  - This is already done in Spring Boot, if you used the '@SpringBootApplication' annotation

Spring Web MVC Example

First we start by implementing a controller. In Spring Web MVC 3 this is simply a Java class which has the @Controller Spring stereotype annotation applied to it.

The controller can have arbitrary methods with the @RequestMapping annotation. This will indicate the request that will be handled by the method. This is used by the DispatcherServlet to figure out which controller method to invoke.

This method also has a String parameter that is used as the message. If we wish the user to supply this message the @RequestParameter annotation is used to tell Spring how to use 'request.getParameter' from the Servlet API to extract the request parameter and initialize the method parameter with the value.

The ModelAndView indicates that the next view should be the "helloView" view (logical name).

The ModelAndView includes a bean to be used as the model for the view. This model is the message String that is taken as a request parameter. It also indicates to use the bean name of "helloMessage". A JSP can use this name to refer to the model.

The configuration consists of a <context:component-scan> to detect the Java classes annotated with @Controller that indicates they are Spring components and the <mvc:annotation-driven> to look for @RequestMapping annotations. There is still a "View Resolver" as shown next but gone is the need to register each controller as individual <bean> elements and there is no "Handler mapping" configuration anymore compared to prior Spring Web MVC versions.

3.11 Spring Boot MVC Example

- Add a templating engine, by adding the following to 'pom.xml' in the dependencies section

```
            <dependency>

       <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-
thymeleaf</artifactId>
            </dependency>
```

3.12 Spring Boot MVC Example

- Add a template under 'src/main/resources/templates'
- e.g. 'helloView.html'

```
<html xmlns:th="http://www.thymeleaf.org">
```

```
<body>
  <h1>Hello</h1>
  Hi there, system says: <span th:text="${helloMessage}">blah</span>
</body>
</html>
```

Spring Boot MVC Example

Thymeleaf is a Java template engine which acts as a substitute for JSPs inside Spring applications. Thymeleaf templates are stored inside of HTML files and consist of a combination of markup and Thymeleaf tags. For more information on Thymeleaf, refer to:

https://www.thymeleaf.org/

3.13 Spring Web MVC Mapping of Requests

- Prior to Spring 3, you would use various "Handler Mapping" beans in Spring configuration files to tell Spring how to determine how the incoming requests would map to controllers
- Since then, you simply need to use @RequestMapping annotations on methods on the controller classes
  - This is much simpler and easier to configure how individual methods will be invoked

3.14 Advanced @RequestMapping

- You can have the @RequestMapping annotation at the class level and the method level to combine how a common pattern is used for a set of requests
  - This can even include @PathVariable declarations or the type of HTTP request

```
@Controller
@ResponseBody
@RequestMapping("/appointments")
public class AppointmentController {
       // gets all appointments
```

```
@RequestMapping(method = RequestMethod.GET)
public List<Appointment> getAll() { .. }

// gets only appointments for a specific date
@RequestMapping(value="/{day}",
        method = RequestMethod.GET)
public List<Appointment> getForDay(
        @PathVariable Date day) { .. }
```

3.15 Composed Request Mappings

- Spring 4 added "composed" annotations that extend from @RequestMapping
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping
- They add in the appropriate 'method=...', so you don't need to type it.

3.16 Spring Web MVC Annotation Controllers

- Since Spring Web MVC 3, the "controller" classes do not need to extend or implement Spring-specific classes
  - All that is needed is the @Controller annotation
- This means that methods within these classes that are meant to handle requests have a number of options for the method signature
  - What is used depends on what the method needs to do to handle the request
- There are a number of options for "handler methods":
  - Method parameters
  - Method return types

Spring Web MVC Annotation Controllers

Rather than trying to detail every possible choice, which is done in the Spring documentation, the next few slides will focus on some of the most common options.

3.17 Controller Handler Method Parameters

- Some of the most common parameters to controller handler methods are:
  - Domain classes from the application
    - Spring can initialize this object matching properties of the object to request parameter names

```
@RequestMapping("/addCustomer")
public String addNewCustomer(Customer customerToAdd) { ..
```

  - Parameters annotated with @PathVariable, @RequestParam, or @CookieValue

```
@RequestMapping("/display/{purchaseId}")
public String displayPurchase(@PathVariable int purchaseId)
```

  - A Map, Spring 'Model', or Spring 'ModelMap' object which can have objects added for the view that will be rendered

```
@RequestMapping("/edit/{purchaseId}")
public String editPurchase(Model model,
       @PathVariable int purchaseId) {
       Purchase toEdit =
              purchaseService.findPurchaseById(purchaseId);
       model.addAttribute("purchase", toEdit);
```

Controller Handler Method Parameters

It is also possible to have various Servlet API objects, like HttpServletRequest and HttpSession as method parameters. It is suggested not to use this approach though as that is more "low level" than you typically need to be with Spring Web MVC.

You can also have method parameters annotated with @RequestHeader and @RequestBody or of type HttpEntity<?> for low level access to the HTTP request.

3.18 Controller Handler Method Return Types

- Some of the most common return types of controller handler methods are:

- A Spring 'ModelAndView' object constructed in the method

```
public ModelAndView listUsers() {
       List<User> allUsers = ...;
       ModelAndView mav = new ModelAndView("userList");
       mav.addObject("users", allUsers);
       return mav;
```

- A String with the view name

```
public String listUsers(Model model) {
       List<User> allUsers = ...;
       model.addObject("users", allUsers);
       return "userList";
```

- A Map, Spring 'Model' object, or application domain class

```
public Map<String, Object> listUsers() {
       List<User> allUsers = ...;
       ModelMap map = new ModelMap()
       map.addAttribute("users", allUsers);
       return map;
```

Controller Handler Method Return Types

Notice that returning a ModelAndView object or using a Model object as a parameter and returning a String for the view name are very similar patterns. Both of these have control over the view name to be rendered and share data with that view.

The third option shown for returning just a Map, Model or domain class depends on some default mapping to a view of the request that had come in.

3.19 View Resolution

- Spring uses a ViewResolver to map the logical view name returned by a controller to a View bean
- The View Resolver is automatically configured when you place a dependency on the template engine 'starter'
- Views will correspond to the template engine
  - e.g. Thymeleaf will resolve a view called 'index' to a template file called 'index.html' in 'src/main/resources/templates'
- You can configure additional view resolvers in your @Configuration

classes

- Spring Boot automatically configures a 'ContentNegotiatingViewResolver' and a 'BeanNameViewResolver', but these can be overridden by the template engine starter

View Resolution

A ViewResolver keeps the controller and view aspects of MVC nicely decoupled. Controllers specify a logical name of the view to dispatch to. A ViewResolver maps this logical name to a View object that will handle the presentation. If you want to use a different view for a particular controller, you can change the ViewResolver or the configuration of the ViewResolver to achieve this, without impacting the controller. For example, you may want to change a view from a JSP to a PDF.

3.20 Spring Boot Considerations

- Spring Boot uses an embedded web server
- Embedded web servers don't do well with JSP
- Spring Boot includes auto-configuration for
  - FreeMarker
  - Groovy
  - Thymeleaf
  - Mustache
- Put templates in 'src/main/resources/templates'

3.21 Summary

- Spring Web MVC provides a rich framework for web applications
- Spring Boot simplifies Spring Web MVC development by automatically setting up a DispatcherServlet for you when you add spring-boot-starter-web as a dependency
- Spring uses the @Controller and @RequestMapping annotations for defining controllers
- Spring Web MVC provides auto configuration support for several template

technologies, including Thymeleaf

Chapter 4 - Overview of Spring Boot Database Integration

Objectives
Key objectives of this chapter: ■ DAO Support in Spring ■ Various data access technologies supported by Spring

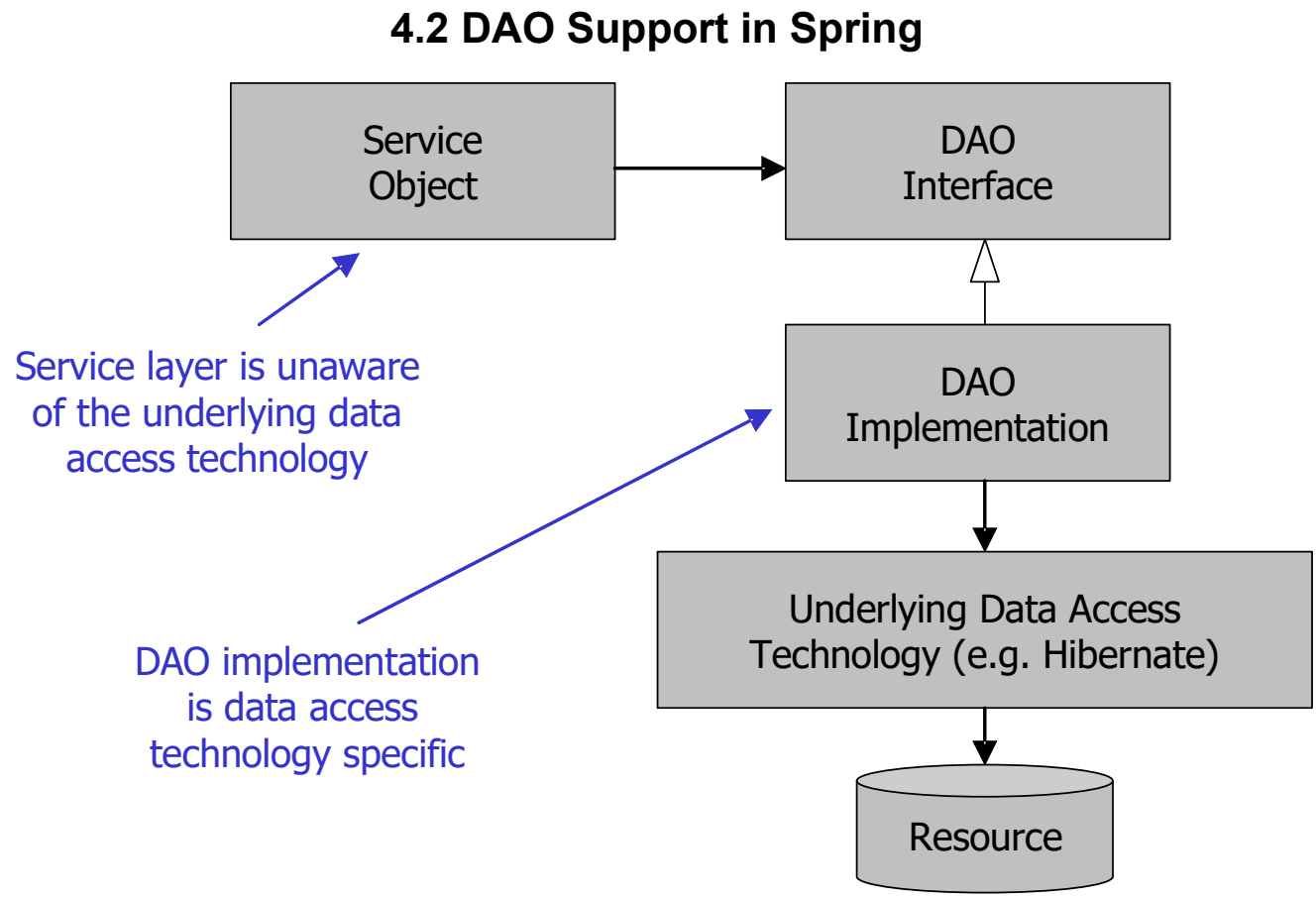
4.1 DAO Support in Spring

- A data access object (DAO) is a mechanism to read and write data from a database
- DAOs use underlying data access technologies such as JDBC or an ORM framework like Hibernate
- Spring DAO support is designed so it is easy to switch from one data access technology to another
  - E.g. JDBC, Hibernate, JDO, etc

DAO Support in Spring

The data access object design pattern is a proven pattern in J2EE application architecture. Its primary purpose is to provide a way of keeping upper software layers decoupled from lower level data access technologies and infrastructural details so upper layers can concern themselves with business logic and not have to worry about data persistence.

Spring recognizes the value of this pattern and provides a sophisticated framework based around it.

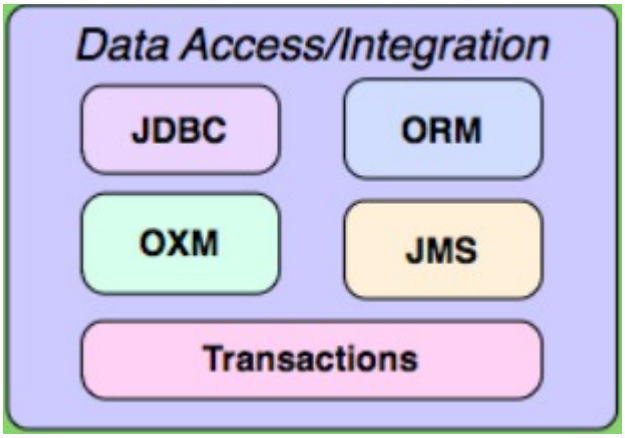


DAO Support in Spring

Spring promotes coding to interfaces and the area of DAOs is no exception. By coding service layers to DAO interfaces, you gain at least two advantages. First, the code is easier to test. Mock DAO implementations can more easily be swapped in during testing of service layers allowing you to fully test a service object without the need for the real DAO implementation and all of its dependencies.

Second, using an interface allows you the flexibility of swapping in a different DAO implementation at deployment time. For example, if you choose to change from JDBC to Hibernate as your data access technology, your service layers are oblivious to this fact and should not need to change.

4.3 Spring Data Access Modules



- Spring has 2 modules providing support for various data access options
  - JDBC module
  - ORM module
    - The ORM module depends on the JDBC module
- Both modules depend on the Spring Transaction module
  - Data access is transactional in nature but Spring supports this with a separate module

4.4 Spring JDBC Module

- The Spring JDBC module provides a framework and supporting classes for simplifying JDBC code
  - It is a good fit for applications that already use direct JDBC access and have not migrated to some Object Relational Mapping framework (ORM)
- The Spring JDBC module also has classes for DataSource access
  - This includes DataSource implementations that could be used in testing or when running an application outside a Java EE server
  - This also includes support for an embedded database which could be used in testing
  - The DataSource support from this module is used even when an application is using the Spring ORM module which is why the ORM module depends on the JDBC module
- Much of the classes from this module used directly in applications are some form of JdbcTemplate class

4.5 Spring ORM Module

- The Spring ORM module has supporting classes for the following ORM frameworks:
  - Hibernate
  - Java Persistence API (JPA)
    - This is the relatively new Java standard way to do persistence
  - Java Data Objects (JDO)
  - iBATIS SQL maps
- An application would typically use only one of these frameworks although the Spring ORM module contains support for all four
  - For applications not already using one of these frameworks the JPA standard is generally the default choice since it is a Java standard

4.6 Spring ORM Module

- Code written using one of the ORM frameworks supported by Spring typically uses the persistence framework directly
  - Spring provides supporting classes to integrate other Spring features like transactions with the ORM framework but these are often not seen in the code of the application itself
  - Spring configuration files is the area that largely contains the integration with the ORM framework
  - Spring has various XXXTemplate classes, like HibernateTemplate, but these generally should NOT be used as Spring 3 can properly integrate when the application uses the ORM API directly

4.7 DataAccessException

- The Spring DAO framework insulates higher layers from technology-specific exceptions
  - E.g. SQLException and HibernateException

- Instead, it throws DataAccessException
  - Root cause is still available using getCause()
- This "Exception translation" is one of the key benefits of Spring data access support
  - This keeps higher layers decoupled from lower level technologies
- DataAccessException is a RuntimeException
  - RuntimeExceptions are "unchecked" which means they do not need to be caught in order for the code to compile
  - This is important because it means that application code is not required to catch these Exceptions since nothing can often be done about them anyway
    - Why add a bunch of code to catch Exceptions when you aren't doing anything except ignoring the Exception?

DataAccessException

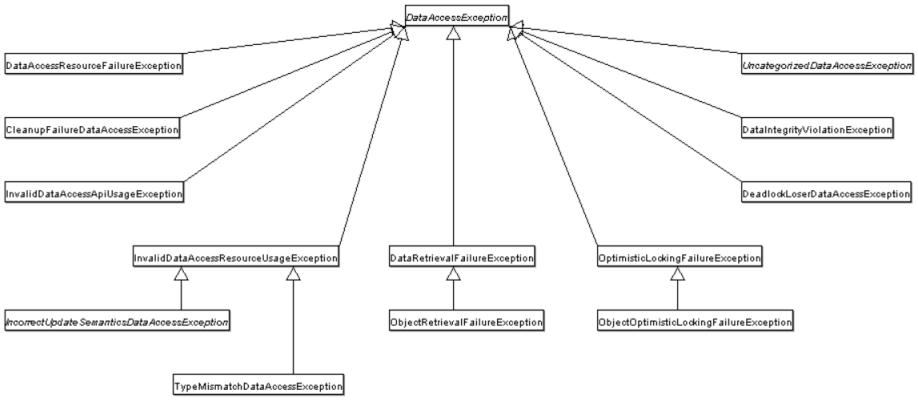
By throwing DataAccessException from your DAO layer instead of underlying data access specific exceptions like SQLException, you keep upper software layers decoupled from the underlying data access technology. For example, you can change from JDBC to Hibernate without upper layers even knowing about it because they don't deal with SQLExceptions generated by JDBC.

Since DataAccessException is a RuntimeException, you are not forced to catch exceptions that you probably cannot recover from. For example, if the database goes down, there probably isn't much that the application can do about it. Even if there is, you can still catch the exception, you just aren't forced to.

4.8 DataAccessException

- Spring provides a hierarchy of DataAccessException subclasses representing different types of exceptions
  - E.g. DataRetrievalFailureException – Data could not be retrieved
- Spring converts data access technology errors to subclasses in this hierarchy
  - Spring understands some database specific error codes and ORM specific exceptions

- Spring exceptions are more descriptive



DataAccessException

The Spring DataAccessException hierarchy allows higher layers to code to technology independent exception classes, further supporting the ability to change data access technologies without impacting higher layers. Springs exceptions can be more descriptive than the underlying database error codes/exceptions because the creators of Spring have gone to great lengths to understand and handle database errors from different vendors.

Some examples exceptions include:

TypeMismatchDataAccessException – Type mismatch between Java type and data type

DataIntegrityViolationException – A write operation resulted in a database integrity violation of some kind (e.g. foreign key violation)

DeadlockLoserDataAccessException – The current process entered a database deadlock and was chosen to be the loser

4.9 @Repository Annotation

- The DataAccessException translation described previously can easily be added to a DAO implementation by adding the @Repository annotation
- This annotation is one of the Spring "stereotype" annotations for declaring Spring components with an annotation
- This annotation is the only code that needs to be added to a class to enable the Exception translation functionality

```
@Repository
public class PurchaseDAOJDBCImpl implements PurchaseDAO {
```

4.10 Using DataSources

- Spring Boot will auto-configure a datasource based on the 'application.properties' file

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

- All you need to do then is annotate a DataSource property with '@Autowired'
- For extra convenience, Spring Boot also sets up a JdbcTemplate and NamedParameterJdbcTemplate instance
  - Again, just declare a reference to the JdbcTemplate type and annotate it '@AutoWired'

Using DataSources

A DataSource provides Connection objects to the underlying database and often provide a connection pool for resource management purposes.

The jndi:lookup element is new to Spring 2.x. It requires the use of the new XML schema-based configuration syntax. In the example on the slide, a DataSource object is looked up using JNDI and assigned to the id "dataSource".

DriverManagerDataSource is a simple data source that uses a DriverManager. It is useful for testing. It has properties for driverClassName, url, username, and password.

4.11 DAO Templates

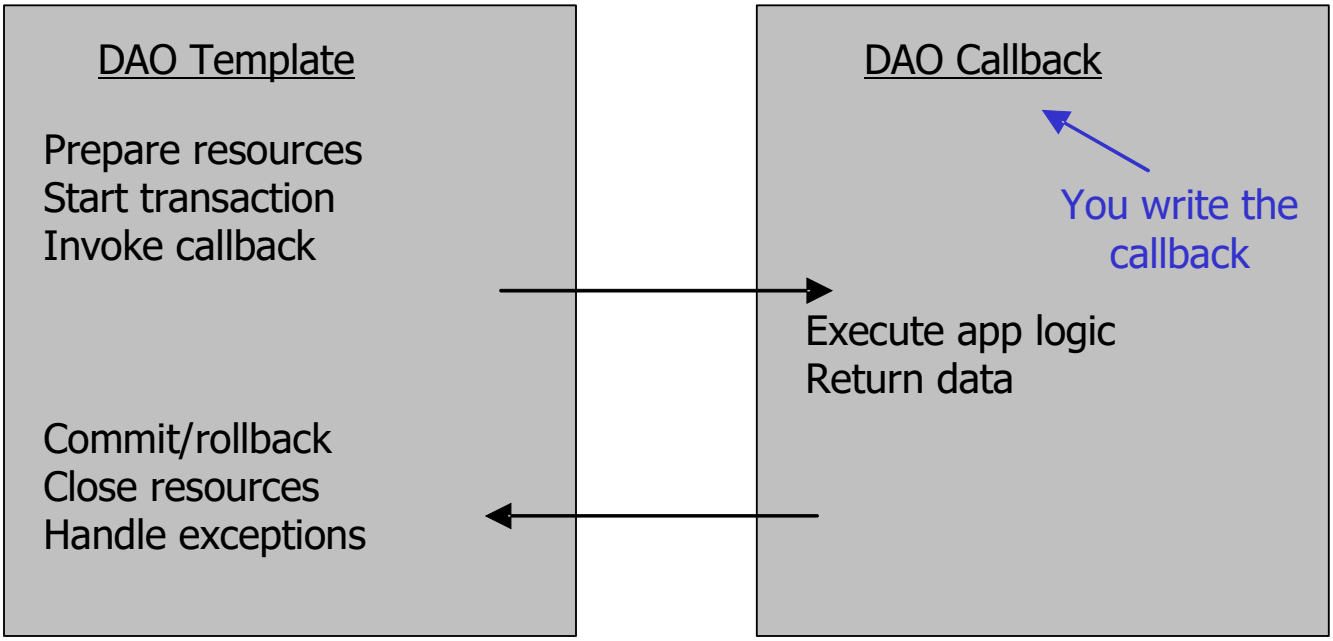
- Data access logic often consists mostly of infrastructural code
  - Resource management, exception handling, connection management, etc
- Spring uses the template method pattern to provide the infrastructural logic for you

- The developer focuses on creating the application-specific logic using a "DAO callback"

DAO Templates

The template method design pattern is illustrated in the classic book "Design Patterns – Elements of Reusable Object-Oriented Software" by Erich Gamma, et al (the "Gang of Four").

4.12 DAO Templates and Callbacks



DAO Templates and Callbacks

The diagram on the slide illustrates templates and callbacks. The DAO template is responsible for all of the data access "plumbing" logic such as resource management and exception handling. Spring provides this for you.

The DAO callback is where the application developer writes his/her application specific data access logic.

4.13 ORM Tool Support in Spring

- ORM (object/relational mapping) frameworks provide functionality over JDBC such as:

    - Lazy loading, caching, cascading updates, etc
- Spring provides integration support for several ORM frameworks including:
    - Hibernate, JDO, iBATIS SQL Maps, Apache OJB
- Spring also provides services on top of these frameworks including:
    - Transaction management
    - Exception handling
    - Template classes
    - Resource management

ORM Tool Support in Spring

Spring does not provide an ORM framework since several excellent frameworks already exist. However, it does provide sophisticated integration with many of these frameworks as well as adding additional services on top of them.

4.14 Summary

- Spring provides modules to simplify data access code
    - These modules are some of the most used features of Spring
- No matter what technology is used Spring provides a common architecture so it is easy to switch data access technologies with minimum impact on the rest of an application
- The Spring @Repository annotation should be used on DAO components to enable Spring's data Exception translation

Chapter 5 - Using Spring with JPA

Objectives
Key objectives of this chapter: ■ Using Spring with JPA ■ Spring Data JPA Repositories ■ Schema Management using Liquibase

5.1 Spring JPA

- Besides basic JDBC support, Spring also provides support for ORM, or Object Relational Mapping
  - ORM is the ability of mapping values of Java objects in memory to tables and columns in a database and having the environment automatically synchronize the two
- Spring Boot provides an JPA module that provides this support

```
<dependency>
  <groupId>
    org.springframework.boot
  </groupId>
  <artifactId>
    spring-boot-starter-data-jpa
  </artifactId>
</dependency>
```

- This brings in
  - Java Persistence (JPA)
  - Hibernate
  - Spring Data JPA
- Although Spring does support the "template" approach like with JDBC, you are often using the ORM framework natively and using Spring just to bootstrap it into the environment

5.2 Benefits of Using Spring with ORM

- Simply using an ORM framework can simplify an application considerably
  - You can simply indicate what is persisted and let the framework calculate the actual SQL code to accomplish that
- Using Spring's ORM support in addition to an ORM framework provides the following benefits:
  - Spring can provide the configuration for the ORM framework as Spring configuration which can make it easy to swap out different configurations for testing, production, etc
  - Spring can translate the data access exceptions of the ORM framework into a standardized set of Spring exceptions
  - Spring can provide access to the ORM managed resources avoiding many common issues when used without Spring
  - The ORM support of Spring also integrates with the transactional support so that transactional behavior of a Spring application is consistent and the ORM framework is integrated into this correctly

5.3 Spring @Repository

- One of the Spring configuration stereotype annotations is specifically designed for Spring components in an application that work with data access
  - This is the @Repository annotation
- By using this annotation (and the appropriate annotation scanning XML configuration) you can register the Java class as a Spring component and enable the Spring data access exception translation
  - Java class:

```
@Repository
public class ProductDaoImpl implements ProductDao {
```

  - Spring Boot normally includes @ComponentScan
  - Spring configuration:

```
<context:component-scan ... />
```

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

Spring @Repository

In the example above there are two <context:...> elements in the XML configuration. Both of these are required to get the detection of the @Repository annotation and the registration of the Spring data access exception translation. The <context:component-scan ..> element is what would scan the Java class to pick up the @Repository annotation while the <context:annotation-config /> registers the PersistenceExceptionTranslationPostProcessor Spring class as a bean to enable exception translation.

If you want to register the exception translation Spring bean manually in XML instead of using <context:annotation-config /> you could use the following although there is not much point to since you are using some kind of XML configuration either way:

<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

5.4 Using JPA with Spring

- The use of JPA is based on the JPA 'EntityManager'
  - This is the interface used to persist or query data managed by JPA
  - Java classes that are mapped by JPA to database tables are 'Entities'

```
@Entity
public class Product { ...
```

- The EntityManager is often injected into a Spring component so that code in the rest of the Spring component can use JPA directly

```
public class PurchaseDaoJpaImpl implements PurchaseDao {
       @PersistenceContext
       private EntityManager em;

  // ...
```

  - The @PersistenceContext annotation is a standard JPA annotation that is used for injection provided the Spring <context:annotation-config /> element is in your Spring configuration
- Spring 4 can be used with JPA 2.2 as long as the JPA provider being used also supports JPA 2.0
  - Hibernate 5+ supports JPA 2.2

Using JPA with Spring

Technically it is possible to register the appropriate Spring bean post processor directly instead of using <context:annotation-config /> to enable Spring injection of @PersistenceContext but again it is not really beneficial to do this manually.

<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

5.5 Configure Spring Boot JPA EntityManagerFactory

- Spring Boot automatically sets up an EntityManagerFactory, so you can just inject the EntityManager with @AutoWired or @PersistenceContext
- In Spring Boot, we don't need a 'persistence.xml' file
  - All Entity classes are picked up by the classpath scanner.

5.6 Application JPA Code

- Although Spring is providing the JPA configuration the actual code that works with persisted data can use only the JPA API

```
public class PurchaseDaoJpaImpl implements PurchaseDao {
	@PersistenceContext
	private EntityManager em;

	public void savePurchase(Purchase purchase) {
		if (getPurchase(purchase.getId()) == null) {
			em.persist(purchase);
		} else {
			em.merge(purchase);
		}
	}

	public List<Purchase> getAllPurchases() {
		Query q = em.createQuery("select p from
Purchase p");
		List<Purchase> results = (List<Purchase>)
			q.getResultList();

		return results;
```

```
}
```

Application JPA Code

The above code could also have Spring annotations at the class or method level such as @Repository or @Transactional but this could also be configured in Spring configuration files if the goal is to not have ANY Spring classes mentioned in the application source code.

You can have this Spring configuration in addition to the above JPA code:

```
<beans>
        <context:annotation-config/>
        <bean id="myPurchaseDao" class="com.mycom.PurchaseDaoJpaImpl"/>
        <!--  Various <aop:config> and <tx:advice> transaction configuration  -->
</beans>
```

instead of the following Spring annotations in the class:

@Repository

@Transactional

public class PurchaseDaoJpaImpl implements PurchaseDao {

5.7 Spring Boot Considerations

- Spring Boot will automatically configure an embedded database if there's one available on the classpath
  - H2
  - HSQL
  - Apache Derby
- At startup time, it runs the scripts 'schema.sql' and 'data.sql' from the resources root.
- Override by configuring 'spring.datasource.*' in 'application.properties'

5.8 Spring Data JPA Repositories

- Spring Data JPA repositories are interfaces that you define to access data
- Spring allows you to avoid writing boilerplate code by providing a set of

`Repository` interfaces

- To automatically generate CRUD (Create, Retrieve, Update, Delete) style methods for your entities, you can simply extend the `CrudRepository<T, ID>` interface
  - `T` corresponds with the entity type and `ID` corresponds with the primary key type
- For example:

```
public interface PurchaseRepository extends
CrudRepository<Purchase, Long> {}
```

- If you need pagination capabilities too, you can instead extend `PagingAndSortingRepository` instead
- If you need advanced JPA capabilities, you can extend `JpaRepository`

Spring Data JPA Repositories

The Repository interface is a marker interface which specifies the entity type as well as the primary key type for that entity.

```
public interface Repository<T, ID> {
}
```

CrudRepository extends Repository and provides basic CRUD operations:

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {
    <S extends T> S save(S entity);
    T findOne(ID primaryKey);
    Iterable<T> findAll();
    Long count();
    void delete(T entity);
    boolean exists(ID primaryKey);
    // … more functionality omitted.
}
```

PagingAndSortingRepository extends Repository and provides basic pagination operations:

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {
  Iterable<T> findAll(Sort sort);
  Page<T> findAll(Pageable pageable);
}
```

JpaRepository extends PagingAndSortingRepository and provides advanced JPA capabilities (e.g., flushing persistence context and deleting records in a batch) operations:

```
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID>,
QueryByExampleExecutor<T> {
```

```
    ...
    void flush();
    <S extends T> S saveAndFlush(S entity);
    void deleteInBatch(Iterable<T> entities);
    void deleteAllInBatch();
    ...
}
```

5.9 Spring Data JPA Repositories

- Often times, you need custom query methods too. Spring Data can automatically generate these for you based on the names of the methods in your interface.
- Spring Data reads the name of the methods on your interface and automatically creates queries to implement them

```
public interface CityRepository
        extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(
        String name,
        String country
    );

}
```

- e.g. 'findByNameAndCountryAllIgnoringCase' provides enough information (along with the type definitions) to synthesize a query

Spring Data JPA Repositories

See the following for more information on Spring Data JPA repositories:

https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html

5.10 Database Schema Migration

- Agile projects require frequent changes to the code and the database schema
- Just like the code is version-controlled, the database schema should also

be version-controlled

- When you have multiple database instances in dev, QA, production and you want to have a tool to automatically track the change history and apply changes intelligently, i.e. apply the diff of current schema and final schema.
- Tools, such as Liquibase and Flyway, are very useful when it comes to supporting database schema migration.
- Database schema migration tools can ensure the data isn't lost when you update a schema.

5.11 Database Schema Migration for CI/CD using Liquibase

- Liquibase is one of the most versatile tools for database migration.
- The alternate to Liquibase is Flyway.
- Liquibase works on nearly all relational database platforms, such as MySQL, SQL Server, and Oracle.
- It performs all the DDL operations and useful database refactorings.
- You can use Liquibase to deploy database changes.
- Since Liquibase runs from a shell or command line, you can run it manually or in any CI/CD pipeline
- Liquibase can also be integrated with Maven, Ant, and other build engines.
- Liquibase can also be integrated with Spring Boot.

Database Schema Migration for CI/CD using Liquibase

For information on integrating Liquibase with Spring Boot and Maven, see:

https://javadeveloperzone.com/spring-boot/spring-boot-liquibase-example/#22_POM_file_configuration

https://www.liquibase.org/documentation/spring.html

5.12 How Liquibase Works?

- Changes are defined in platform-agnostic language and translated to

database platform you use.

- You keep a running list of changes, and Liquibase applies those changes for you through its execution engine.
- It runs on Java that's why it needs the correct JDBC driver to perform the database update.
- Liquibase tracks changes using its own tables in your schema to ensure consistency and to avoid corruption due to incorrectly altered changelogs.
- It records a hash of each changeset. While it's running updates, it puts a "lock" on your database so you can't accidentally run two changelogs concurrently.

5.13 Changelogs in Liquibase

- Changelogs are written using domain-specific languages.
- You can write them in JSON, YAML, XML, or one of a few other supported formats, such as Groovy and Clojure.
- Changelogs consist of a series of changesets
- A changeset represents a single change to your database, e.g. creating a table, adding a column to a table, and adding an index.
- You can also include SQL statements in a changelog.
- The changelog should be kept under source control.
- You always add a new changeset to the end of the changelog and never alter an existing changeset.

5.14 Preconditions in Changelogs

- Preconditions can be attached to change logs or changesets to control the execution of an update based on the state of the database.
- There are several reasons to use preconditions, including:
  - Document what assumptions the writers of the changelog had when creating it.

- Enforce that those assumptions are not violated by users running the changelog
- Perform data checks before performing an unrecoverable change such as drop_Table
- Control what changesets are run and not run based on the state of the database

- Preconditions at the changelog level apply to all changesets, not just those listed in the current changelog or its child changelogs.

5.15 Sample Empty Changelog

- XML

```
<databaseChangeLog
  xmlns=
    "http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ext=
    "http://www.liquibase.org/xml/ns/dbchangelog-ext"
  xsi:schemaLocation=
    "http://www.liquibase.org/xml/ns/dbchangelog
   http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-
3.1.xsd
    http://www.liquibase.org/xml/ns/dbchangelog-ext
   http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-
ext.xsd">
</databaseChangeLog>
```

- JSON

```
{
    "databaseChangeLog": [
    ]
}
```

5.16 Sample Precondition in Changelog

```
{
  "databaseChangeLog": [
    {
      "preConditions": [
        {
          "runningAs": {
             "username": "bob"
           }
        }
      ]
    },
...
```

5.17 Sample Changeset in Changelog

- The JSON listed below creates a table named person and adds a column named id to the table.

```
{
"changeSet": {
  "id": "1",
  "author": "bob",
  "changes": [
    {
      "createTable": {
        "tableName": "person",
        "columns": [
          {
            "column": {
              "name": "id",
              "type": "int",
              "autoIncrement": true,
              "constraints": {
                "primaryKey": true,
                "nullable": false
            },
          }
        },
...
```

5.18 Running Liquibase

- Liquibase can be run from the command line by running:

```
liquibase [options] [command] [command parameters]
```

- Optionally, you can replace the liquibase command with:

```
java -jar liquibase.jar
```

- Standard migration command

```
java -jar liquibase.jar \
  --driver=oracle.jdbc.OracleDriver \
  --classpath=\path\to\classes:jdbcdriver.jar \
  --changeLogFile=com/example/db.changelog.xml \
  --url="jdbc:oracle:thin:@localhost:1521:oracle" \
  --username=scott \
  --password=tiger \
  update
```

5.19 Liquibase Commands

- There are several commands supported by Liquibase. Some of the common commands are listed below:
  - tag <tag> - Tags the current database for future rollback
  - update – Updates database to current version
  - updateSQL – Writes SQL to update database to the current version to STDOUT
  - rollback <tag> – Rolls back database to the state it was in
  - diff – Writes a description of differences to STDOUT
  - dbDoc – Generates Javadoc-like documentation based on the current database and changelog
  - …
- The complete command list is available on the official website: https://www.liquibase.org/documentation/command_line.html

5.20 Summary

- Spring provides exceptional support for various ORM frameworks
- The preferred approach in modern Spring applications is:
  - Use the JPA API to create standards-based persistence code
  - Use a JPA provider library (like Hibernate) to provide the JPA support
  - Use Spring to configure JPA and link to other Spring configuration or the environment in a Java EE server
- Spring Data JPA repositories help you avoid writing boilerplate code by automatically generating CRUD style methods and query methods for your entities
- Use Liquibase for database migration in your Spring Boot application

Chapter 6 - Spring REST Services

Objectives
Key objectives of this chapter ■ A Basic introduction to REST-style services ■ Using Spring MVC to implement REST services ■ Combining the JAX-RS standard with Spring

6.1 Many Flavors of Services

- Web Services come in all shapes and sizes
  - XML-based services (SOAP, XML-RPC, RSS / ATOM Feeds)
  - HTTP services (REST, JSON, standard GET / POST)
  - Other services (FTP, SMTP)
- While SOAP is the most common style of service, increasingly organizations are utilizing REST for certain scenarios
  - REpresentational State Transfer (REST), first introduced by Roy Fielding (co-founder of the Apache Software Foundation and co-author of HTTP and URI RFCs)
  - REST emphasizes the importance of resources, expressed as URIs
  - Used extensively by Amazon, Google, Yahoo, Flickr, and others

6.2 Understanding REST

- REST applies the traditional, well-known architecture of the Web to Web Services
  - Everything is a resource
  - Each URI is treated as a distinct resource and is addressable and accessible using an application or Web browser
  - URIs can be bookmarked and even cached
- Leverages HTTP for working with resources

- GET – Retrieve a representation of a resource. Does not modify the server state. A GET should have no side effects on the server side.
- DELETE – Remove a representation of a resource
- POST – Create or update a representation of a resource
- PUT – Update a representation of a resource

Understanding REST

The notion of a resource "representation" is very important within REST. Technically you should never have a direct pipeline to a resource, but rather access to a representation of a resource. For example, a resource which represents a circle may accept and return a representation which specifies a center point and radius, formatted in SVG, but may also accept and return a representation which specifies any three distinct points along the curve as a comma-separated list. This would be two distinct representations of the same resource.

6.3 RESTful Services

- A RESTful Web service services as the interface to one or more resource collections.
- There are three essential elements to any RESTful service
  - Resource Address – expressed as a URI
  - Representation Format – a known MIME type such as TXT or XML, common data formats include JSON, RSS / ATOM, and plain text
  - Resource Operations – a list of supported HTTP methods (GET, POST, PUT, DELETE)

6.4 REST Resource Examples

- GET /checkflightstatus/AA1215 HTTP/1.1
  - Check the flight status for American Airlines flight #1215
- POST /checkflightstatus HTTP/1.1
  - Upload a new flight status by sending an XML document that conforms to a previously defined XML Schema

- Response is a “201 Created” and a new URI

```
201 Created
Content-Location: /checkflightstatus/AA1215
```

- PUT /checkflightstatus/AA1215 HTTP/1.1
  - Update an existing resource representation
- DELETE /checkflightstatus/AA1215 HTTP/1.1
  - Delete the resource representation

6.5 @RestController Annotation

- Spring 4 (and by extension, Spring Boot) added the @RestController annotation
- It combines @Controller and @ResponseBody
- So now you just need one annotation on a REST controller class.

6.6 Implementing JAX-RS Services and Spring

- JAX-RS is the official Java specification for defining REST services in Java
  - Similar to Spring MVC this is done mainly with JAX-RS annotations although these annotations are different
- Spring MVC is NOT a JAX-RS implementation
- It is possible though to use standard JAX-RS code and implementation to define the REST service itself and simply inject Spring components into the JAX-RS service class
  - This would provide for more "standard" code for the REST service but allow the other features of Spring to be used as well
- The best way to do this would be to use the SpringBeanAutowiringSupport class from within a @PostConstruct method of the JAX-RS class which contains @Autowired Spring components

```
public class QuoteService {
@Autowired
```

```
private QuoteGenerator generator;
@PostConstruct
public void initSpringComponents() {
    SpringBeanAutowiringSupport.
        processInjectionBasedOnCurrentContext(this);
}
```

Implementing JAX-RS Services and Spring

Since Spring is not a JAX-RS implementation it will not recognize the JAX-RS annotations.

6.7 JAX-RS Annotations

- JAX-RS uses different annotations from Spring MVC
- @Path annotation for linking classes or methods to the request path that will map to them

```
@Path("/quotes")
public class QuoteService {
```

- Separate annotations for mapping to the various HTTP methods

```
@GET   @POST  @PUT   @DELETE
```

- You can use a @PathParam annotation with a method parameter to extract data from the URL

```
@Path("/thisResource/{resourceId}")
public String getResourceById(@PathParam("resourceId")
String id) { … }
```

JAX-RS Annotations

See the following article for a comparison chart of the JAX-RS and Spring annotations:

https://dzone.com/articles/lets-compare-jax-rs-vs-spring-for-rest-endpoints

6.8 Java Clients Using RestTemplate

- The most common types of clients for REST services are JavaScript and AJAX clients
- It is perhaps common also to need to have Java code communicate with

REST services as a client

- Although REST service requests and responses are fairly simple using basic Java APIs like the java.net package would be difficult and low-level
- JAX-RS 2.1 provides a client API, but before then did not
- Spring provides the RestTemplate class which has a number of convenience methods for sending requests to REST services
  - These REST services do not need to be implemented using Spring MVC or JAX-RS
- To use the RestTemplate class in a Java client you would still need to include the Spring MVC module in the application

6.9 RestTemplate Methods

- There are many different methods and these are generally provided based on the HTTP method that will be sent
  - The parameters and return type differ depending on if the request body will contain data from an object and if the response body will contain data coming back
  - All take a String for URL and a variable argument Object... parameter for parameters in the URL
- DELETE methods are probably the simplest

```
void delete(String url, Object... urlVariables)
```

- GET takes no request body but returns an object for the response body

```
<T> getForObject(String url, Class<T> responseType,
Object... urlVariables)
```

- PUT takes a request body but returns nothing

```
void put(String url, Object request, Object...
urlVariables)
```

- POST takes a request body and can return a response body

```
<T> postForObject(String url, Object request, Class<T>
responseType, Object... uriVariables)
```

6.10 Summary

- Spring MVC provides basic support for implementing REST services
- Spring could also be used behind standard JAX-RS REST services

Chapter 7 - Spring Security

Objectives

Key objectives of this chapter

- Overview of Spring Security
- Configuring Spring Security
- Defining security restrictions for an application
- Customizing Spring Security form login, logout, and HTTPS redirection
- Using various authentication sources for user information

7.1 Securing Web Applications with Spring Boot 2

- Spring Security is a framework extending the traditional JEE Java Authentication and Authorization Service (JAAS)
- It can work by itself on top of any Servlet-based technology
  - It does however continue to use Spring core to configure itself
- It can integrate with many back-end technologies
  - Support for OpenID, Kerberos, LDAP, SAML and database
- It uses a servlet-filter to control access to all Web requests
- It can also integrate with AOP to filter method access
  - This gives you method-level security without having to actually use EJB

7.2 Spring Security

- Because it is based on a servlet-filter, it can also work with SOAP based Web Services, RESTful Services, any kind of Web Remoting, and Portlets
- It can even be integrated with non-Spring web frameworks such

as Struts, Seam, and ColdFusion

- Single Sign On (SSO) can be integrated through CAS, the Central Authentication Service from JA-SIG
  - This gives us access to authenticate against X.509 Certificates, OpenID (supported by Google, Facebook, Yahoo, and many others), and LDAP
  - WS-Security and WS-Trust are built on top of these
- It can integrate into WebFlow
- There's support for it in SpringSource Tool Suite

Notes

Security is a many-headed problem. In this course we're mostly concerned with request-level authentication and authorization, but there's much more to deal with. Cross-Site Replay attacks, SQL injection, Man-In-The-Middle, Denial-Of-Service and Phishing attacks are just some of the concerns a comprehensive security offering needs to address.

The Spring Security site is:

http://static.springsource.org/spring-security/site/

7.3 Authentication and Authorization

- Authentication answers the question “Who are you?”
  - Includes a User Registry of known user credentials
  - Includes an Authentication Mechanism for comparing the user credentials with the User Registry
  - Spring Security can be configured to authenticate users using various means or to accept the authentication that has been done by an external mechanism
- Authorization answers the question “What can you do?”
  - Once a valid user has been identified, a decision can be made about allowing the user to perform the requested function
  - Spring Security can handle the authorization decision

- Sometimes this may be very fine-grained. For example, allowing a user to delete their own data but not the data of other users

7.4 Programmatic vs Declarative Security

- Programmatic security allows us to make fine grained security decisions but requires writing the security code within our application
  - The security rules being applied may be obscured by the code being used to enforce them
- Whenever possible, we would prefer to declare the rules for access and have a framework like Spring Security enforce those rules
  - This allows us to focus on the security rules themselves and not writing the code to implement them
- With Spring Security we have a DSL for security that enables us to declare the kinds of rules we would have had to code before
  - It also enables us to use EL in our declarations which gives us a lot of flexibility
  - This can include contextual information like time of access, number of items in a shopping cart, number of previous orders, etc.

7.5 Adding Spring Security to a Project

- To use Spring Security with Spring Boot, add the following dependency to your Maven file:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

7.6 Spring Security Configuration

- If Spring Security is on the classpath, then web applications will be setup with “basic” authentication on all HTTP endpoints.
- There is a default AuthenticationManager that has a single user called 'user' with a random password.
  - The password is printed out during application startup
  - Override the password with 'security.user.password' in 'application.properties'.
- To override security settings, define a bean of type 'WebSecurityConfigurerAdapter' and plug it into the configuration

7.7 Spring Security Configuration Example

```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class ApplicationSecurity
        extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http)
                throws Exception {

            http.authorizeRequests()
                .antMatchers("/css/**").permitAll().anyRequest()
                .fullyAuthenticated().and().formLogin()
                .loginPage("/login")
                .failureUrl("/login?error")
                .permitAll().and().logout().permitAll();
        }

        @Override
        public void configure(AuthenticationManagerBuilder auth)
            throws Exception {
            auth.inMemoryAuthentication()
              .withUser("user").password("user").roles("USER");
        }

}
```

7.8 Authentication Manager

- The AuthenticationManager class provides user information
  - You can use multiple <authentication-provider> elements and they will be checked in the declared order to authenticate the user
- In the example above, the WebSecurityConfigurerAdapter's 'configure()' method gets called with an AuthenticationManagerBuilder.
- We can use this to configure the AuthenticationManager using a “fluent API”
  - auth.jdbcAuthentication().dataSource(ds).withDefaultSchema()
  - See the Spring Security JavaDocs for more details.

7.9 Using Database User Authentication

- You can obtain user details from tables in a database with the jdbcAuthentication() method
  - This will need a reference to a Spring Data Source bean configuration

```
auth.jdbcAuthentication().dataSource(ds).withDefaultSchema()
```

- If you do not want to use the database schema expected by Spring Security you can customize the queries used and map the information in your own database to what Spring Security expects

```
auth.jdbcAuthentication()
  .dataSource(securityDatabase)
  .usersByUsernameQuery(
    "SELECT username, password, 'true' as enabled " +
    "FROM member " +
    "WHERE username=?")
  .authoritiesByUsernameQuery(
    "SELECT m.username, mr.role as authority " +
    "FROM MEMBER m, MEMBER_ROLE mr WHERE m.username=? " +
    "AND m.id=mr.member_id");
```

Using Database User Authentication

The configuration of the 'securityDatabase' Data Source above is not shown but it is just like Spring database configuration.

The queries that Spring Security uses by default are:

```
SELECT username, password, enabled FROM users WHERE username = ?
SELECT username, authority FROM authorities WHERE username = ?
```

The default statements above assume a database schema similar to:

```
CREATE TABLE USERS (
    USERNAME VARCHAR(20) NOT NULL,
    PASSWORD VARCHAR(20) NOT NULL,
    ENABLED SMALLINT,
    PRIMARY KEY (USERNAME)
);
CREATE TABLE AUTHORITIES (
    USERNAME VARCHAR(20) NOT NULL,
    AUTHORITY VARCHAR(20) NOT NULL,
    FOREIGN KEY (USERNAME) REFERENCES USERS
);
```

Notice in the custom queries defined in the slide the 'enabled' part of the query is mapped as 'true' since it is assumed the table referenced does not have this column but Spring Security expects it. If the table does have some column similar to 'enabled' it should map to a boolean type (like a '1' for enabled and '0' for disabled).

The custom queries above would work with a database schema of:

```
CREATE TABLE MEMBER (
    ID BIGINT NOT NULL,
    USERNAME VARCHAR(20) NOT NULL,
    PASSWORD VARCHAR(20) NOT NULL,
    PRIMARY KEY (ID)
);
CREATE TABLE MEMBER_ROLE (
    MEMBER_ID BIGINT NOT NULL,
    ROLE VARCHAR(20) NOT NULL,
    FOREIGN KEY (MEMBER_ID) REFERENCES MEMBER
);
```

7.10 LDAP Authentication

- It is common to have an LDAP server that stores user data for an entire organization
- The first step in using this with Spring Security is to configure how Spring Security will connect to the LDAP server with the ldapAuthentication builder

```
auth.ldapAuthentication()
    .contextSource()
    .url("ldap://localhost").port(389)
    .managerDn("cn=Directory Admin")
    .managerPassword("ldap");
```

- You can also use a "embedded" LDAP server in a test environment by not providing the 'url' attribute and instead providing ldif files to load

LDAP Authentication

The 'manager-dn' and 'manager-password' attributes of <ldap-server> are used for how to authenticate against the LDAP server to query user details.

If using the embedded LDAP server the default for the 'root' will be "dc=springframework,dc=org" if you do not supply a value.

In order to configure Spring Security there are a number of attributes related to LDAP that have various defaults that may affect how your LDAP configuration behaves. This slide is meant to simply introduce the feature. One step you should take when attempting to use Spring Security with LDAP is to avoid configuring everything at once. Start with an embedded list of users to test the other configuration settings and then switch to using LDAP. Also try using the embedded LDAP server with an ldif file exported from your LDAP server with a few sample users.

7.11 What is Security Assertion Markup Language (SAML)?

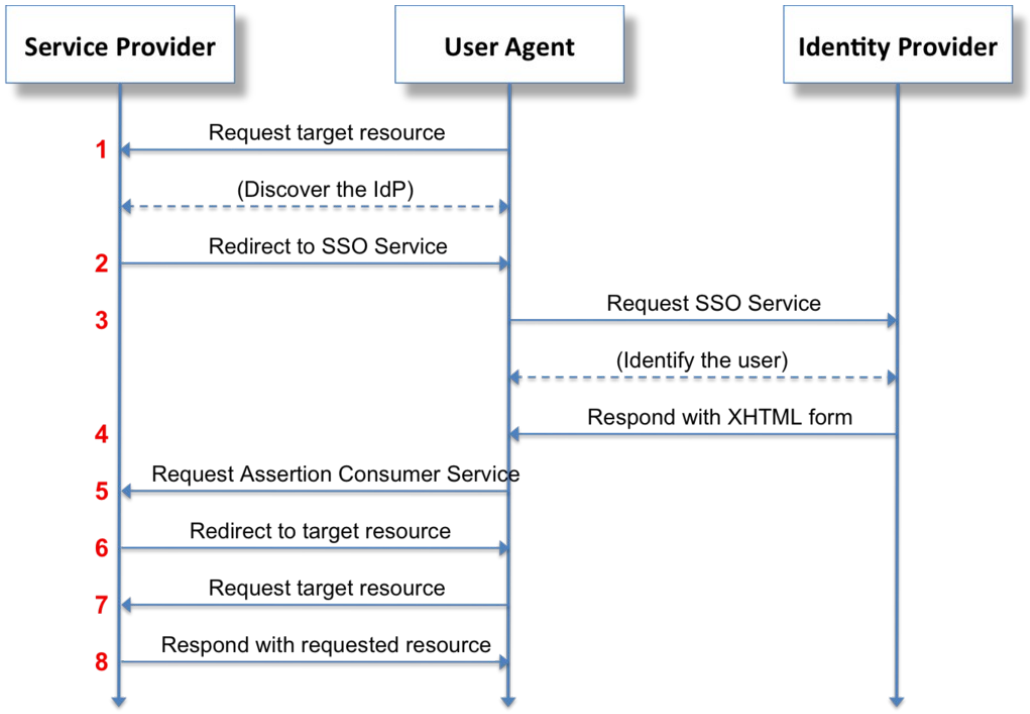
- Security Assertion Markup Language (SAML) is an open standard that allows identity providers (IdP) to pass authorization credentials to service providers (SP).
- It’s a security protocol similar to OpenId, OAuth, Kerberos etc
- SAML is the link between the authentication of a user’s identity and the authorization to use a service.
- SAML adoption allows IT shops to use software as a service (SaaS) solutions while maintaining a secure federated identity management system.
- SAML enables Single-Sign On (SSO), which means users can log in once, and those same credentials can be reused to log into other service providers.

7.12 What is a SAML Provider?

- A SAML provider is a system that helps a user access a service they need.
- There are two primary types of SAML providers, service provider, and identity provider.
  - A service provider needs the authentication from the identity provider to grant authorization to the user.
  - An identity provider performs the authentication that the end user is who they say they are and sends that data to the service provider along with the user's access rights for the service.
- Microsoft Active Directory or Azure are common identity providers. Salesforce and other CRM solutions are usually service providers, in that they depend on an identity provider for user authentication.

7.13 Spring SAML2.0 Web SSO Authentication

- This diagram from wikipedia explains how SAML works



1. User hits the Service Provider URL Service provider discovers the IDP to contact for authentication
2. Service provider redirects to the corresponding IDP
3. User hits the IDP and identifies the user
4. IDP redirects to the Login form
5. Redirect to Service provider Assertion consumer URL (the URL in Service provider that accepts SAML assertion)
6. SP initiates redirect to target resource
7. Browser requests for the target resource
8. Service provider responds with the requested resource

7.14 Setting Up an SSO Provider

- For SAML authentication to work we need an identity provider (IdP).
- There are various providers, such as Active Directory, Azure, AWS, Google, Microsoft, Facebook, Onelogin, etc
- Obtain the domain name and fully qualified domain name of the Active Directory server.
- To enable SSO on Active Directory, the following steps are typically performed:
  - Ensure that LDAP is configured on the Active Directory (AD) server:
  - From the AD Server, run **ldp**.
  - From the Connections menu, click Connect, and configure Server name, port, and select SSL option.
  - When the LDAP is properly configured, the external domain server details are displayed in the LDP window. Otherwise, an error message appears indicating that a connection cannot be made using this feature.
  - When the LDAP is properly configured, the external domain server details are displayed in the LDP window. Otherwise, an error message appears indicating that a connection cannot be made using this feature.

7.15 Adding SAML Dependencies to a Project

- To use SAML, add the following dependencies to your Maven file:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.security.extensions</groupId>
 <artifactId>spring-security-saml2-core</artifactId>
 <version>1.0.10.RELEASE</version>
 </dependency>
```

7.16 SAML vs. OAuth2

- OAuth is a slightly newer standard that was co-developed by Google and Twitter to enable streamlined internet logins.
- OAuth uses a similar methodology as SAML to share login information.
- SAML provides more control to enterprises to keep their SSO logins more secure, whereas OAuth is better on mobile and uses JSON.
- Facebook and Google are two OAuth providers that you might use to log into other internet sites.

7.17 OAuth2 Overview

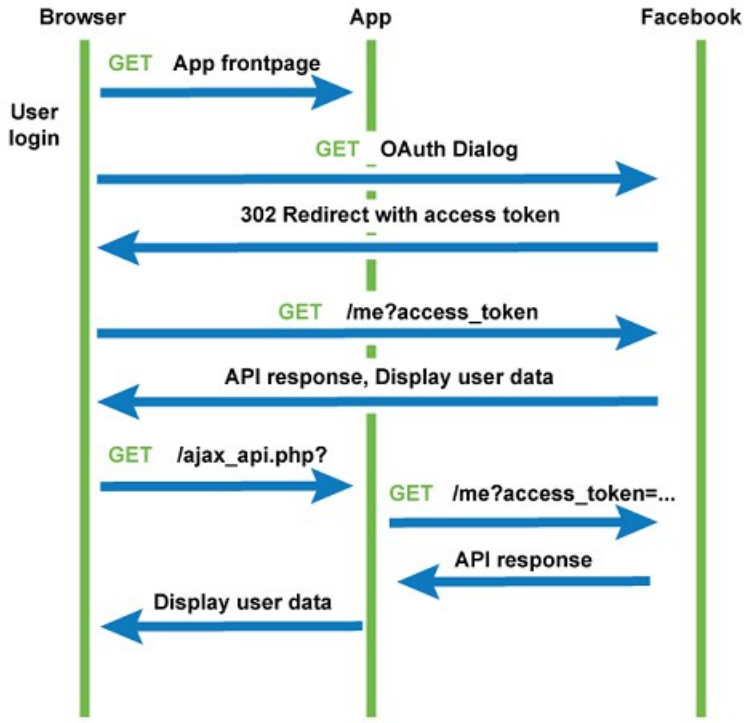
- OAuth is an authorization method to provide access to resources over the HTTP protocol.
- It can be used for authorization of various applications or manual user access
- It is commonly used as a way for internet users to grant websites or applications access to their information on other websites without giving them the passwords.
- This mechanism is used by companies, such as Google, Facebook, Microsoft, Twitter, and DropBox, to permit the users to share information about their accounts with third party applications or websites.
- It allows an application to have an access token
- Access token represents a user's permission for the client to access their

data

- The access token is used to authenticate a request to an API endpoint

7.18 OAuth – Facebook Sample Flow

- Although, the diagram below is for Facebook, but it's similar for any other provider.



7.19 OAuth Versions

- There are two versions of OAuth authorization
  - OAuth 1 – HMAC-SHA signature strings
  - OAuth 2 – tokens over HTTPS
- OAuth2 is not backwards compatible with OAuth 1.0.
- OAuth2 provides specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.

7.20 OAuth2 Components

- Resource server – the API server which contains the resources to be accessed
- Authorization server – provides access tokens. It can be the same as the API server
- Resource owner – access tokens are provided by the resource owner, i.e. the user, when resources are accessed
- Client / consumer – an application using the credentials

7.21 OAuth2 – End Points

- The token Endpoint is used by clients to get an access token from the authorization server
- It can also optionally refresh the token

7.22 OAuth2 – Tokens

- The two token types involved in OAuth2 authentication are:
  - Access Token – used for authentication an authorization to get access to the resources from the resource server
  - Refresh Token – sent together with the access token. It is used to get a new access token, when the old one expires. It allows for having a short expiration time for access tokens to the resource server and a long expiration time for access to the authorization server.
- Access tokens also have a type which defines how they are constructed
  - Bearer Tokens – uses HTTPS security and the request is not signed or encrypted. Possession of the bearer token is considered authentication
  - MAC Tokens – more secure than bearer tokens. MAC tokens are similar to signatures, in that they provide a way to have partial cryptographic verification of the request.

7.23 OAuth – Grants

- Methods to get access tokens from the authorization server are called grants
- The same method used to request a token is also used by the resource server to validate a token
- There are 4 basic grant types:
  - Authorization Code – when the resource owner allows access, an authorization code is then sent to the client via browser redirect, and the authorization code is used in the background to get an access token. Optionally, a refresh token is also sent. This grant flow is used when the client is a third-party server or web application, which performs the access to the protected resource.
  - Implicit – it's similar to authorization code, but instead of using the code as an intermediary, the access token is sent directly through a browser redirect. This grant flow is used when the user-agent will access the protected resource directly, such as in a rich web application or a mobile app.
  - Resource Owner Credentials – the password / resource owner credentials grant uses the resource owner password to obtain the access token. Optionally, a refresh token is also sent. The password is then authenticated.
  - Client Credentials – the client's credentials are used instead of the resource owner's. The access token is associated either with the client itself, or delegated authorization from a resource owner. This grant flow is used when the client is requesting access to protected resources under its control

7.24 Authenticating Against an OAuth2 API

- Most OAuth2 services use the /oauth/token URI endpoint for handling all OAuth2 requests
- The first step in authenticating against an OAuth2 protected API service is exchanging your API key for an Access Token.

- It can be done by performing these steps:
  - Create a POST request
  - Supply grant_type=client_credentials in the body of the request
- Let's say the API key has two components
  - ID:xxx
  - Secret: yyy
- cURL could be used to get an Access Token like this:

```
curl --user xxx:yyy --data grant_type=client_credentials -X
POST https://api.someapi.com/oauth/token
```

Authenticating Against an OAuth2 API

To be able to view sample OAuth2 requests and responses, try out:

https://developers.google.com/oauthplayground/

7.25 OAuth2 using Spring Boot – Dependencies

- Maven dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-oauth2-resource-
server</artifactId>
</dependency>
```

7.26 OAuth2 using Spring Boot – application.yml

- src/main/resources/application.yml requires security configuration
- Note: This example uses the Facebook provider.

```
security:
```

```
  oauth2:
    client:
      clientId: 233668646673605
      clientSecret: 33b17e044ee6a4fa383f46ec6e28ea1d
      accessTokenUri: https://graph.facebook.com/oauth/access_token
      userAuthorizationUri: https://www.facebook.com/dialog/oauth
      tokenName: oauth_token
      authenticationScheme: query
      clientAuthenticationScheme: form
    resource:
      userInfoUri: https://graph.facebook.com/me
```

7.27 OAuth2 using Spring Boot – Main Class

```
@SpringBootApplication
@EnableOAuth2Sso
@RestController
public class DemoApplication extends WebSecurityConfigurerAdapter {

    @RequestMapping("/user")
        public Principal user(Principal principal) {
            return principal;
        }

        @Override
        protected void configure(HttpSecurity http) throws Exception {
        http
        .antMatcher("/**")
        .authorizeRequests()
        .antMatchers("/", "/login**", "/webjars/**")
        .permitAll()
        .anyRequest()
        .authenticated()
        .and()
        .logout()
        .logoutSuccessUrl("/")
        .permitAll()
        .and()
        .csrf()
        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
    }
    // ...
```

7.28 OAuth2 using Spring Boot – Single Page Application Client

- The sample code below uses AngularJS, but you can use similar concepts with or without client-side framework

```
angular.module("app", [])
```

```
    .controller("home", function($http) {
        var self = this;

        self.logout = function() {
            $http.post('/logout', {}).success(function() {
                self.authenticated = false;
                $location.path("/");
        }).error(function(data) {
            console.log("Logout failed")
            self.authenticated = false;
        });
    };

    $http.get("/user").success(function(data) {
        self.user = data.userAuthentication.details.name;
        self.authenticated = true;
    }).error(function() {
        self.user = "N/A";
        self.authenticated = false;
    });
});
```

7.29 JSON Web Tokens

- A replacement for standard/traditional API keys
- It is an open standard
- Allow fine-grained access control via “claims”
  - A claim is any data a client “claims” to be true
  - Typically includes “who issued the request”, “when it was issued”, ...
- Cross-Domain capable (cookies are not)
- Compact (compared with XML based security)
- Encoded (URL-Safe)
- Signed (to prevent tampering)
- OAuth and JWT are not the same

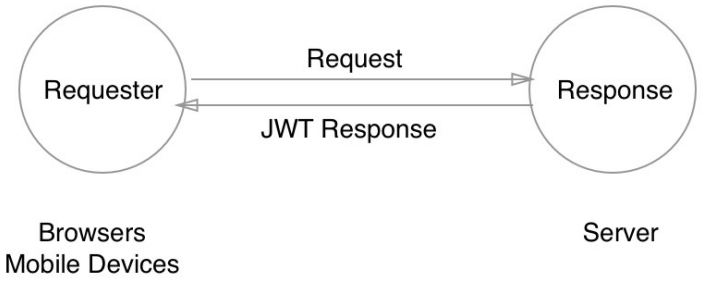
- JWT is a specific protocol for a security access token
- OAuth is a broader security framework for the interaction of different actors (end users, back-end APIs, authorization servers) for the generation and distribution of security access tokens

7.30 JSON Web Token Architecture

- Three sections
  - Header
  - Payload
  - Signature
- Header and Payload are base64 encoded
- Signature is calculated from the encoded header and payload
- Sections are separated by a period

7.31 How JWT Works

- JWT works as a two way protocol where a request is made and the response is generated from a server



- The browser makes the request for JWT encoded data
- The server generates the signed token and return to the client
- The token can be sent over the http request for every other request that

needs authentication on the server.

- The server then validates the token and, if it's valid, returns the secure resource to the client.

7.32 JWT Header

- Declares the signature algorithm and type

```
{
  "typ":"JWT",
  "alg":"HS256"
}
```

- The algorithm shown here (HMAC SHA-256) will be used to create the signature.
- The type “JWT” stands for JSON Web Token
- When base64 encoded it looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

7.33 JWT Payload

- The payload contains “Claims”
- Claims come in several types
  - Registered
  - Public
  - Private
- Examples of Registered claims include
  - iss: use to define who issued the token
  - sub: the subject of the token
  - exp: token expiration time
- Public claims
  - Use URI schemes to prevent name collision
  - i.e. https:/corpname.com/jwt_claims/is_user
- Private claims
  - For use inside organizations
  - Can use simple naming conventions
  - i.e. “department”

7.34 JWT Example Payload

- Example:

```
{
  "iss": "corpname.com",
  "aud": "corpname.com/rest/product",
  "sub": "jdoe",
  "Email": "jdoe@corpname.com"
}
```

- After base64 encoding:

```
eyJpc3MiOiJjb3JwbmFtZS5jb20iLCJhdWQiOiJjb3JwbmFtZS5jb20vcmV
zdC9wcm9kdWN0Iiwic3ViIjoiamRvZSIsIkVtYWlsIjoiamRvZUBjb3Jwbm
FtZS5jb20ifQ
```

7.35 JWT Example Signature

- The signature is created from the header and body like this:

```
content = base64UrlEncode(header)
        + "."
        +  base64UrlEncode(payload);
signature = HMACSHA256(content);
```

- Completed signature:

```
pEonrJLKkpSvAMk5dmBYoxP5hZ0ZhKcnkLJYNNlVxipSoZbCnDrhSq8Psda
5dPqyjnLasPY7pyxoRKx99HAVu8L9hwdO_h9GZ6K443Xvb6uDSMsyvqQp8v
65Rv0SjUenWQRK7INyZ2N8rkHdEaMOOiOPFp7yHLUo8Tq_AM2Q
```

7.36 How JWT Tokens are Used

- Client requests token
  - Sends credentials to Authentication server
  - Server returns a JWT token
- Client adds token to HTTP request via the Authentication header.
  - A JWT token can be cached on the browser and returned on every request to the server to ensure the user has access to the resources on every request without authentication on every request.

- The downside of this is that the user will have access for the duration of the token unless there is a blacklist each service checks against.

- Client sends the request
- API receives request:
  - Reads the JWT from the Authentication header
  - Unpacks the payload
  - Checks claims
  - Allows or denies access

7.37 Adding JWT to HTTP Header

- After obtaining a JWT token the client adds it to an HTTP request as an HTTP header
  - Header Name: Authorization
  - Type: Bearer
- Example:

```
 Authorization:Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJjb3JwbmFtZS
5jb20iLCJhdWQiOiJjb3JwbmFtZS5jb20vcmVzdC9wcm9kdWN0Iiwic3ViI
joiamRvZSIsIkVtYWlsIjoiamRvZUBjb3JwbmFtZS5jb20ifQ.pEonrJLKk
pSvAMk5dmBYoxP5hZ0ZhKcnkLJYNNlVxipSoZbCnDrhSq8Psda5dPqyjnLa
sPY7pyxoRKx99HAVu8L9hwdO_h9GZ6K443Xvb6uDSMsyvqQp8v65Rv0SjUe
nWQRK7INyZ2N8rkHdEaMOOiOPFp7yHLUo8Tq_AM2Q
```

7.38 How The Server Makes Use of JWT Tokens

- The RESTful web service needs to validate JWT tokens when it receives requests.
- Process
  - Unpack token
  - Validate that signature matches header and payload
  - Validates claims (has token expired?)

- Compares scopes
- If required it makes call to ACL (access control list) server.
- Grants or denies access

- This process can be coded into JEE Servlet filters or added directly to the web service code

7.39 What are “Scopes”?

- The payload area of a JSON web token contains a "claim" named "scope"
- The value for the “scope” field is an array.
- Example:

```
"scope": [ "app.feature" ]
"scope": [ "HR.review " ]
```

- Technically scope strings can include any text.
- In practice scope strings are limited to those defined by an organization.
- Scope strings refer to specific operations on a specific API endpoints.

7.40 JWT with Spring Boot – Dependencies

- Detailed JWT implementation with Spring Boot is covered in the course project.
- Add JWT dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.1</version>
</dependency>
```

7.41 JWT with Spring Boot – Main Class

```
@EnableWebSecurity
public class SecurityTokenConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf()
            .disable()
            // Add a filter to validate the tokens with every request
            .addFilterAfter(
                new JwtTokenAuthenticationFilter(jwtConfig),
                UsernamePasswordAuthenticationFilter.class)
            // authorization requests config
            .authorizeRequests()
            // allow all who are accessing "auth" service
            .antMatchers(HttpMethod.POST, jwtConfig.getUri()).permitAll()
            // must be an admin if trying to access admin area
            // (authentication is also required here)
            .antMatchers("/gallery" + "/admin/**").hasRole("ADMIN")
            // Any other request must be authenticated
            .anyRequest()
            .authenticated();
    }

    // ...
```

7.42 Summary

- Spring Security has many features that simplify securing web applications
- Making use of many of these features only requires configuration in a Spring configuration file
- Spring Security can work with many different sources of user and permission information

Chapter 8 - Spring JMS

Objectives
Key objectives of this chapter ■ Spring JMS ■ JmsTemplate ■ Async Message Receiver, Message-Driven POJOs(MDPs) ■ Transaction management ■ Producer Configuration ■ Consumer Configuration ■ Message-Driven POJO's Async receiver Configuration

8.1 Spring JMS

- Spring 5 JMS fully supports JMS 2.0
  - Supports JMS 1.1 conventions by default. However, you're encouraged to upgrade to using a JMS 2.0 compatible driver.
- JMS provides a domain-independent API, that means messages can be sent and received to/from different model Queue or Topic using the same Session object
- Spring provides a Template based solution (JmsTemplate class) to simplify the JMS API code development
- JmsTemplate class can be used for sending messages and synchronously receive messages
- For Asynchronous message receiving, just like JEE Message-Driven Beans (MDBs), Spring uses Message-Driven POJOs (MDPs)
- This template converts JMSException into org.springframework.jms.JmsException

Spring JMS

org.springframework.jms.support.converter provides a MessageConverter abstraction to convert between Java objects and JMS messages.

org.springframework.jms.support.destination provides various strategies for managing JMS destinations, such as providing a service locator for destinations stored in JNDI.

org.springframework.jms.connection provides an implementation of the ConnectionFactory suitable for use in standalone applications. It also contains an implementation of Spring's PlatformTransactionManager for JMS (the cunningly named JmsTransactionManager). This allows for seamless integration of JMS as a transactional resource into Spring's transaction management mechanisms.

8.2 JmsTemplate

- The JmsTemplate class helps to obtain and release the JMS resources
- To send messages, simply call send method which take two parameters, destination and MessageCreator
- MessageCreator object is usually implemented as an anonymous class
- MessageCreator interface has only one method createMessage() to be implemented.
- JmsTemplate objects are thread-safe and keeps the reference to ConnectionFactory
- Define template on the configuration file

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

8.3 Connection and Destination

- Spring provided SingleConnectionFactory which is a implementation of JMS ConnectionFactory
- Calls to createConnection() on ConnectionFactory return a Connection object
- JmsTemplate can be configured with a default destination via the property defaultDestination
    - The default destination can be used on JmsTemplate with send and receive operations that do not refer to a specific destination

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

Connection to Messaging Server

The CachingConnectionFactory extends the functionality of SingleConnectionFactory and adds the caching of Sessions, MessageProducers, and MessageConsumers.

8.4 JmsTemplate Configuration

- Declare JmsTemplate, ConnectionFactory, and Destinations
  - The ConnectionFactory and Destination may depend on the JMS implementation of the environment
  - Below is shown using ActiveMQ as a JMS implementation in Tomcat
    - In a Java EE Environment you would likely use <jee:jndi-lookup> to link Spring beans to JNDI lookups

```
<bean id="connectionFactory" class=
"org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
        value="tcp://localhost:61616" />
</bean>
<bean id="orderDestination"
     class="org.apache.activemq.command.ActiveMQQueue">
     <constructor-arg value="order.queue" />
</bean>
<bean id="jmsTemplate" class="org.sf.jms.core.JmsTemplate">
    <property name="connectionFactory"
        ref="connectionFactory" />
    <property name="defaultDestination"
       ref="orderDestination" />
</bean>
```

JmsTemplate Configuration

The declaration of ConnectionFactory and Destinations to link to JNDI lookups would be something like:

```
<jee:jndi-lookup id="connectionFactory"
         jndi-name="connectionFactory"/>

<jee:jndi-lookup id="orderDestination"
         jndi-name="jms/orderQueue"/>
```

8.5 Transaction Management

- JmsTransactionManager provides support for managing transactions on a single ConnectionFactory
- Add the followings to bean configuration file (see the example below)
  - <tx:annotation-driven/>
  - JmsTransactionManager
- JmsTemplate automatically detects transaction
- JmsTemplate can be used with JtaTransactionManager (XA) for distributed transactions

8.6 Example Transaction Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
    ...>
   <tx:annotation-driven/>
    <bean id="transactionManager" class=
"org.springframework.jms.connection.JmsTransactionManager">
        <property name="connectionFactory">
            <ref bean="connectionFactory" />
        </property>
    </bean>
</beans>
```

8.7 Producer Example

- The following example illustrates how to call the send method on the

JmsTemplate

```
@Component
public class OrderProducer {
   @Autowired private JmsTemplate jmsTemplate;

   public void addOrder(final Order order) {
     jmsTemplate.send(new MessageCreator() {
       public Message createMessage(Session session)
           throws JMSException {
         MapMessage m = session.createMapMessage();
         m.setLong("id", order.getId());
         m.setString("custName", order.getCustName());
         …
         return m;
       }
     });
   }
}
```

Producer Example

Now we can use JmsTemplate to send messages. We do this by calling the send method of JmsTemplate.

The send method takes a MessageCreator that implements the callback portion of the template method pattern. You place your application specific logic in the createMessage method. In the above code notice the use of the anonymous inner class when creating the MessageCreator in the parameters of the 'send' method. You don't need to worry about exception handling, resource management, etc. This is similar to other template techniques used in Spring (e.g. JDBC).

The createMessage method can use the JMS Session argument to perform JMS specific logic. In the example on the slide, we create a MapMessage and build the message using the properties of the order. We then return this message as the one to be sent.

Note that createMessage throws JMSException but we don't have to handle the exception. We leave that to JmsTemplate (which will catch it and re-throw it as a Spring unchecked JmsException).

MessageCreator is in the org.springframework.jms.core package.

8.8 Consumer Example

- Call receive method on the JmsTemplate

```
@Component
public class OrderConsumer {
   @Autowired private JmsTemplate jmsTemplate;

   public Order receiveOrder() {
     MapMessage m = (MapMessage)jmsTemplate.receive();
     try {
       Order order = new Order();
       order.setId(m.getLong("id"));
       order.setCustName(m.getString("custName"));
       …
       return order;
     }
     catch (JMSException e) {
       throw JmsUtils.convertJmsAccessException(e);
     }
   }
}
```

Consumer Configuration

However, when extracting information from the received MapMessage object, you still have to handle the JMS API's JMSException. This is in stark contrast to the default behavior of the framework, where it automatically maps exceptions for you when invoking methods on the JmsTemplate. To make the type of the exception thrown by this method consistent, you have to make a call to JmsUtils.convertJmsAccessException() to convert the JMS API's JMSException into Spring's JmsException.

By default, this template will wait for a JMS message at the destination forever, and the calling thread is blocked in the meantime.

To avoid waiting for a message so long, you should specify a receive timeout for this template. If there's no message available at the destination in the duration, the JMS template's receive() method will return a null message.

If you're expecting a response to something or want to check for messages at an interval, handling the messages and then spinning down until the next interval. If you intend to receive messages and respond to them as a service, you're likely going to want to use the message-driven POJO functionality.

8.9 Converting Messages

- You can use a MessageConverter to encapsulate the logic to convert

messages to domain objects

```
public class OrderConverter implements MessageConverter {
  public Object fromMessage(Message m) {
    MapMessage mm = (MapMessage)m;
    Order order = new Order();
    try {
      order.setId(mm.getLong("id"));
      …
    }
    catch (JMSException e) {
      throw new
        MessageConversionException(e.getMessage());
    }
    return order;
  }
```

Converting Messages

The service methods in the examples on the previous slides contain considerable logic to convert domain objects to JMS messages and vice versa. To increase the reusability of this logic, Spring provides the MessageConverter interface.

In the example on the slide, we implement this interface as the OrderConverter class. It defines the fromMessage and toMessage methods (toMessage is depicted on the next slide).

The fromMessage method takes a JMS Message and converts it to an Object. We have transplanted the logic from the MessageCreator createMessage method to fromMessage.

8.10 Converting Messages

- And convert domain objects to messages

```
…
public Message toMessage(Object object,
    Session session) throws JMSException {
  Order order = (Order)object;
  MapMessage m = session.createMapMessage();
  m.setLong("id", order.getId());
  m.setString("custName", order.getCustName());
  …
  return m;
```

```
    }
}
```

Converting Messages

The toMessage method takes an Object and converts it to a JMS Message. We have transplanted the logic from the receiveOrder method to toMessage.

8.11 Converting Messages

- Configure JmsTemplate and MessageConverter

```
<bean id="jmsTemplate" class="org…JmsTemplate">
  …
  <property name="messageConverter">
    <ref bean="messageConverter"/>
  </property>
</bean>
<bean id="messageConverter" class="com…OrderConverter"/>
```

- The addOrder method is much simpler now

```
public void addOrder(Order order) {
  jmsTemplate.convertAndSend(order);
}
```

- The receiveOrder method is also simpler

```
public Order receiveOrder() {
  return (Order)jmsTemplate.receiveAndConvert();
}
```

Converting Messages

Configure the JmsTemplate to use the OrderConverter by wiring it into the "messageConverter" property.

Now the service methods are much cleaner. Use the convertAndSend method to convert an object to a JMS message using the currently registered MessageConverter of the JmsTemplate. Use the receiveAndConvert method to receive a message and convert it to an object.

8.12 Message Listener Containers

- Spring provides message-driven POJOs (MDPs) similar to Message-

Driven Bean in EJB container, to receive messages

- A message listener container can be used to receive messages from a JMS message queue and drive the MessageListener that is injected into it.
  - A message listener container is the intermediary between a MDP and a messaging provider
  - This container is responsible for all threading of message reception and dispatches into the listener for processing.
- Spring provides two standard JMS message listener containers
  - SimpleMessageListenerContainer
    - Creates a fixed number of JMS sessions and consumers at startup, registers the listener using the standard JMS MessageConsumer.setMessageListener() method
    - JMS provider should perform listener callbacks
    - Does not support managed Transaction environment, like Java EE
  - DefaultMessageListenerContainer
    - It supports managed Transaction environment, like Java EE
    - Received messages are registered with XA transaction

8.13 Message-Driven POJO's Async Receiver Example

- Create a message listener class using MessageListener interface

```
public class OrderListener implements MessageListener {
    public void onMessage(Message message) {
        MapMessage msg = (MapMessage) message;
        try {
            OrderInfo info = new OrderInfo();
            info. setOrderID(msg.getString("orderID"));
            info.setDate(msg.getString("orderDate"));
            info.setDeliveryDate(msg.
               getDouble("deliveryDate"));
            displayOrderInfo(info);
        } catch (JMSException e) {
            throw JmsUtils.convertJmsAccessException(e);
```

```
        }
    }
    private void displayOrderInfo(Orderinfo info) {
        System. out. println("OrderID" + info.getOrderID()
              " received and to be shipped on "  +
              info.getDeliveryDate);
    }
}
```

8.14 Message-Driven POJO's Async Receiver Configuration

```
<bean id="connectionFactory" class=
      "org.apache.activemq.ActiveMQConnectionFactory">
   <property name="brokerURL"
      value="tcp://localhost:61616" />
</bean>
<bean id="orderListener"
      class="com.simple.OrderListener" />
<bean class=
"org.springframework.jms.listener.DefaultMessageListenerCon
tainer">
   <property name="connectionFactory"
      ref="connectionFactory" />
   <property name="destinationName" value="order.queue" />
   <property name="messageListener" ref="orderListener" />
</bean>
```

Message-Driven POJO's Async Receiver Configuration

There are two types of message listener containers:

- SimpleMessageListenerContainer, does not support transactions
- DefaultMessageListenerContainer, does support transactions

8.15 Spring Boot Considerations

- Spring JMS and hence Spring Boot support a number of different JMS providers including ActiveMQ, Rabbit MQ, Artemis and Kafka
- To use Spring Boot with Rabbit MQ, add the following dependency to your

build file:

- spring-boot-starter-amqp

- Spring Boot will automatically configure an 'AmqpTemplate' and 'AmqpAdmin' if you use the 'spring-boot-starter-amqp' starter
- Support for RabbitMQ is built-in
- Configure using application properties:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

- To setup a listener, simply annotate a method with '@RabbitListener'

```
@RabbitListener(queues = "someQueue")
public void processMessage(String content) {
    // ...
}
```

8.16 Summary

- JMS support in Spring, how to use Spring to build message-oriented architectures.
- You learned how to configure both producer and consumer messages using a message queue.
- You learned how to build message-driven POJOs.
- How to configure Spring Boot to use Rabbit MQ