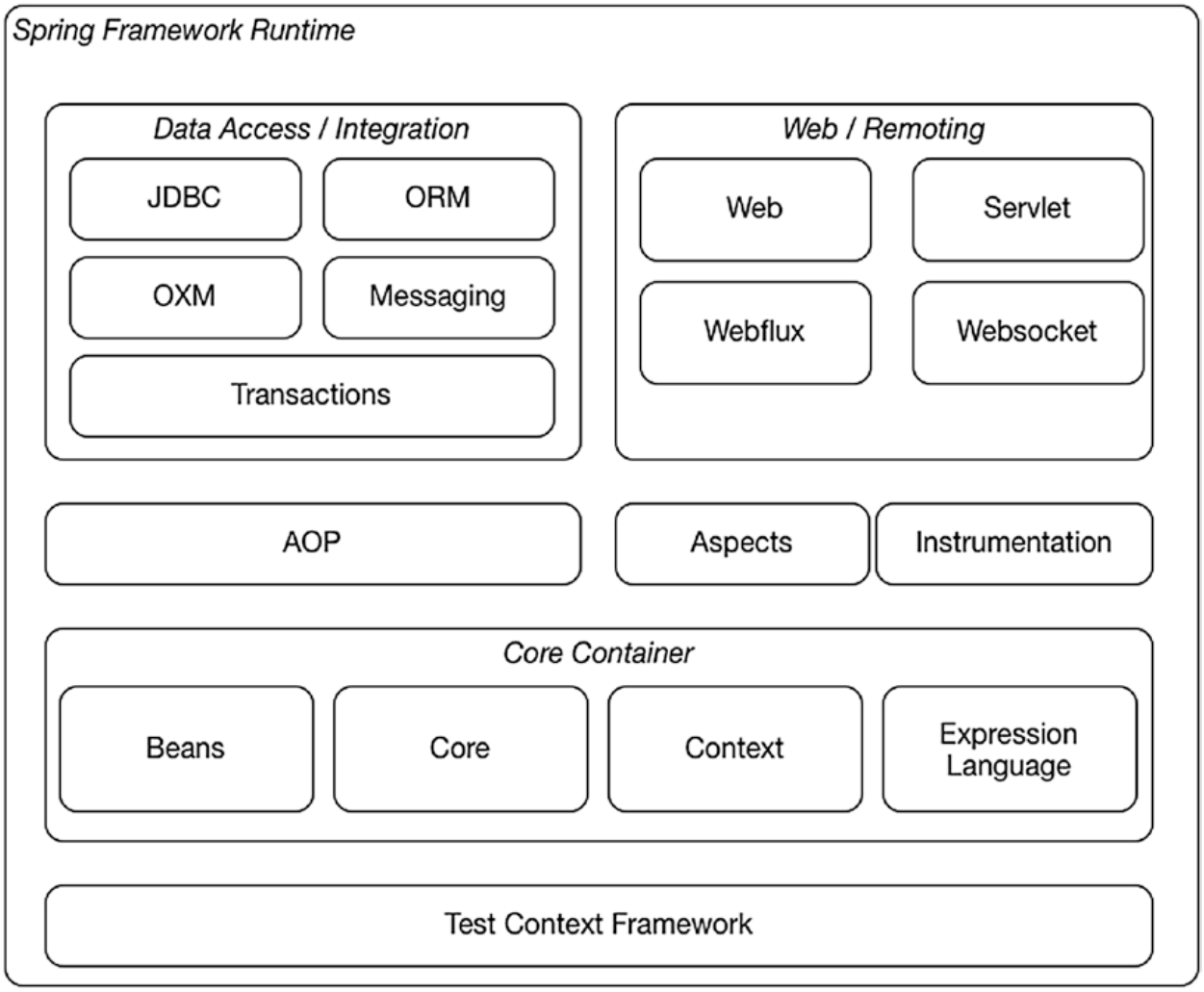
Spring MVC + WebFlux



Module Artifact Description
AOP spring-aop The proxy-based AOP framework for Spring
Aspects spring-aspects AspectJ-based aspects for Spring
Beans spring-beans Spring's core bean factory support
Context spring-context Application context runtime implementations; also contains scheduling and remoting support classes
Context spring-context-indexer
Support for providing a static index of beans used in the application; improves startup performance
Context spring-context-support
Support classes for integrating third-party libraries with Spring
Core spring-core Core utilities
Expression
Language

spring-expression
Classes for the Spring Expression Language (SpEL)
Instrumentation spring-instrument
Instrumentation classes to be used with a Java agent
JCL spring-jcl Spring specific replacement for commons-logging
JDBC spring-jdbc JDBC support package that includes datasource setup classes and JDBC access support
JMS spring-jms JMS support package that includes synchronous JMS access and message listener containers
ORM spring-orm ORM support package that includes support for Hibernate 5+ and JPA
Messaging spring-messaging
Spring messaging abstraction; used by JMS and WebSocket
OXM spring-oxm XML support package that includes support for object-to-XML mapping; also includes support for JAXB, JiBX, XStream, and Castor
Test spring-test Testing support classes
Transactions spring-tx Transaction infrastructure classes; includes JCA integration and DAO support classes
Web spring-web Core web package for use in any web environment
WebFlux spring-webflux Spring WebFlux support package Includes support for several reactive runtimes like Netty and Undertow
Servlet spring-webmvc Spring MVC support package for use in a Servlet environment Includes support for common view technologies
WebSocket spring-websocket
Spring WebSocket support package Includes support for communication over the WebSocket protocol

Dependency Injection
In dependency injection (DI), objects are given their dependencies at construction time. It is a Spring Framework foundation. You have probably heard of Inversion of Control (IoC).7 IoC is a broader, more general concept that can be addressed in different ways. IoC lets developers decouple and focus on what is important for a given part of an enterprise application, but without thinking about what other parts of the system do. Programming to interfaces is one way to think about decoupling.
Almost every enterprise application consists of multiple components that need to work together. In the early days of Java enterprise development, we simply put all the logic of constructing those objects (and the objects they needed) in the constructor (see Listing 2-1). At first sight, there is nothing wrong with that approach; however, as time progresses, object construction became slow, and objects had a lot of knowledge they shouldn’t have had (see the single-responsibility principle).8 Those classes became hard to maintain, and they were also hard to the unit and/or integration test.

ApplicationContexts
To do dependency injection in Spring, you need an application context. In Spring, this is an instance of the org.springframework.context.ApplicationContext interface. The

application context is responsible for managing the beans defined in it. It also enables more elaborate things like applying AOP to the beans defined in it.

An ApplicationContext | Overview Implementation | Location File type
ClassPathXmlApplicationContext Classpath XML
FileSystemXmlApplicationContext File system XML
AnnotationConfigApplicationContext Classpath Java
XmlWebApplicationContext Web Application Root XML
AnnotationConfigWebApplicationContext Web Application Classpath Java

Resource Loading
Prefix Location
classpath: The root of the classpath
file: File system
http: Web application root

Expression Description
classpath:/META-INF/
spring/*.xml
Loads all files with the XML file extensions from the classpath in the
META -INF/spring directory
file:/var/conf/*/.
properties
Loads all files with the properties file extension from the /var/conf
directory and all subdirectories

Component Scanning
Spring also has something called component scanning. In short, this feature enables Spring to scan your classpath for classes that are annotated with org.springframework.stereotype.Component (or one of the specialized annotations like @Service, @Repository, @Controller, or org.springframework.context.annotation.Configuration). If we want to enable component scanning, we need to instruct the application context to do so. The org.springframework.context.annotation.ComponentScan annotation enables us to accomplish that. This annotation needs to be put on our configuration class to enable component scanning.

```
@Configuration
@ComponentScan(basePackages = {
"com.apress.prospringmvc.moneytransfer.scanning",
"com.apress.prospringmvc.moneytransfer.repository" })
public class ApplicationContextConfiguration {}
```

Scopes
By default, all beans in a Spring application context are singletons. As the name implies,

there is a single instance of a bean, and it is used for the whole application. This doesn't typically present a problem because our services and repositories don't hold state; they simply execute a certain operation and (optionally) return a value.

An Overview of Scopes

Prefix	Description
singleton	The default scope. A single instance of a bean is created and shared throughout the application.
prototype	Each time a certain bean is needed, a fresh instance of the bean is returned.
thread	The bean is created when needed and bound to the currently executing thread. If the thread dies, the bean is destroyed.
request	The bean is created when needed and bound to the lifetime of the incoming javax.servlet.ServletRequest. If the request is over, the bean instance is destroyed.
session	The bean is created when needed and stored in javax.servlet. HttpSession. When the session is destroyed, so is the bean instance.
globalSession	The bean is created when needed and stored in the globally available session (which is available in Portlet environments). If no such session is available, the scope reverts to the session scope functionality.
application	This scope is very similar to the singleton scope; however, beans with this scope are also registered in javax.servlet.ServletContext.

Profiles

Spring introduced profiles in version 3.1. Profiles make it easy to create different configurations of our application for different environments. For instance, we can create separate profiles for our local environment, testing, and our deployment to CloudFoundry. Each of these environments requires some environment-specific configuration or beans. You can think of database configuration, messaging solutions, and testing environments, stubs of certain beans. To enable a profile, we need to tell the application context in which profiles are active.

```
@Configuration
public class ApplicationContextConfiguration {
@Bean
public AccountRepository accountRepository() {
return new MapBasedAccountRepository();
}
@Bean
public MoneyTransferService moneyTransferService() {
return new MoneyTransferServiceImpl();
}
@Configuration
@Profile(value = "test")
public static class TestContextConfiguration {
@Bean
```

```
public TransactionRepository transactionRepository() {
return new StubTransactionRepository();
}
}
@Configuration
@Profile(value = "local")
public static class LocalContextConfiguration {
@Bean
public TransactionRepository transactionRepository() {
return new MapBasedTransactionRepository();
}
}
}
```

Enabling Features

The Spring Framework offers a lot more flexibility than dependency injection; it also provides many different features we can enable. We can enable these features using annotations (see Table 2-6). Note that we won't use all the annotations in Table 2-6; however, our sample application uses transactions, and we use some AOP. The largest part of this book is about the features provided by the org.springframework.web.servlet.config.annotation.EnableWebMvc and org.springframework.web.reactive.config.EnableWebFlux annotations.

Spring Boot automatically enables some of these features; it depends on the classes detected on the classpath.

Annotation	Description	Detected by Spring Boot
org.springframework.context.annotation.EnableAspectJAutoProxy	Enables support for handling beans stereotyped as org.aspectj.lang.annotation.Aspect.	Yes
org.springframework.scheduling.annotation.EnableAsync	Enables support for handling bean methods with the org.springframework.scheduling.annotation.Async or javax.ejb.Asynchronous annotations.	No
org.springframework.cache.		

annotation.EnableCaching
Enables support for bean methods with the org.springframework.cache.annotation.Cacheable annotation.
Yes
org.springframework.context.annotation.EnableLoadTimeWeaving
Enables support for load-time weaving. By default, Spring uses a proxy-based approach to AOP; however, this annotation enables us to switch to load-time weaving. Some JPA providers require it.
No
org.springframework.scheduling.annotation.EnableScheduling
Enables support for annotation-driven scheduling, letting us parse bean methods annotated with the org.springframework.scheduling.annotation.Scheduled annotation.
No
org.springframework.beans.factory.aspectj.EnableSpringConfigured
Enables support for applying dependency injection to non-Spring managed beans. In general, such beans are annotated with the org.springframework.beans.factory.annotation.Configurable annotation. This feature requires load-time or compiletime weaving because it needs to modify class files.
No
org.springframework.transaction.annotation.EnableTransactionManagement
Enables annotation-driven transaction support, using org.springframework.transaction.annotation.Transactional or javax.ejb.TransactionAttribute to drive transactions.
Yes

org.springframework.web.
servlet.config.annotation.
EnableWebMvc
Enables support for the powerful and flexible
annotation-driven controllers with request
handling methods. This feature detects
beans with the org.springframework.
stereotype.Controller annotation
and binds methods with the org.
springframework.web.bind.
annotation.RequestMapping annotations
to URLs.
Yes
org.springframework.web.
reactive.config.EnableWebFlux
Enables support for the powerful and flexible
reactive web implementation using the well-known
concepts from Spring Web MVC and,
where possible, extend on them.
Yes
Aspect-Oriented Programming
To enable the features listed in Table 2-4, Spring uses aspect-oriented programming (AOP). AOP is another way of thinking about the structure of software. It enables you to modularize things like transaction management or performance logging, features that span multiple types and objects (cross-cutting concerns). In AOP, there are a few important concepts to keep in mind (see Table 2-7).
Now let's look at transaction management and how Spring uses AOP to apply transactions around methods. The transaction advice, or interceptor, is org.springframework.transaction.interceptor.TransactionInterceptor. This advice is placed around methods with the org.springframework.transaction.annotation.Transactional annotation. To do this, Spring creates a wrapper around the actual object, which is known as a proxy (see Figure 2-5). A proxy acts like an enclosing object, but it allows (dynamic) behavior to be added (in this case, the transactionality of the method).
Table 2-7. Core AOP Concepts
Concept Description
Aspect The modularization of a cross-cutting concern. In general, this is a Java class with the org.aspectj.lang.annotation.Aspect annotation.
Join Point A point during the execution of a program. This can be the execution of a method, the assignment of a field, or the handling of an exception. In Spring, a join point is always the execution of a method!
Advice The specific action taken by an aspect at a particular join point. There are several types of advice: before, after, after throwing, after returning, and around. In Spring, an advice is called an interceptor because we are intercepting method invocations.
Pointcut A predicate that matches join points. The advice is associated with a pointcut

expression and runs at any join point matching the pointcut. Spring uses the AspectJ expression language by default. Join points can be written using the org.aspectj.lang.annotation.Pointcut annotation.

Spring Boot

All the things mentioned previously in this chapter also apply to Spring Boot. Spring Boot builds upon and extends the features of the Spring Framework. It does make things a lot easier, however. Spring Boot automatically configures the features it finds on the classpath by default. When Spring Boot detects Spring MVC classes, it starts Spring MVC. When it finds a DataSource implementation, it bootstraps it.

MVC in Short

Component Description

Model The model is the data needed by the view so that it can be rendered. It might be an order placed or a list of books requested by a user.

View The view is the actual implementation, and it uses the model to render itself in a web application. This could be a JSP or JSF page, but it could also be a PDF, XML, or JSON representation of a resource.

Controller The controller is the component that is responsible for responding to the action the user takes, such as form submission or clicking a link. The controller updates the model and takes other actions needed, such as invoking a service method to place an order.

Application Layering

In the introduction, we mentioned that an application consists of several layers (see Figure 3-3). We like to think of a layer as an area of concern for the application. Therefore, we also use layering to achieve separation of concerns. For example, the view shouldn't be burdened with business or data access logic because these are all different concerns and typically located in different layers.

Layer Description

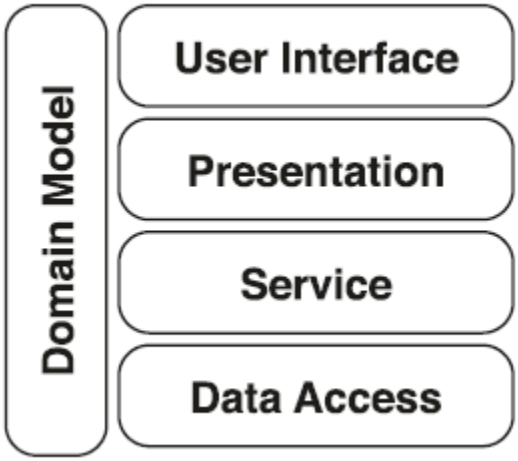
Presentation This is most likely to be a web-based solution. The presentation layer should be as thin as possible. It should also be possible to provide alternative presentation layers like a web frontend or a web service façade. This should all operate on a well-designed service layer.

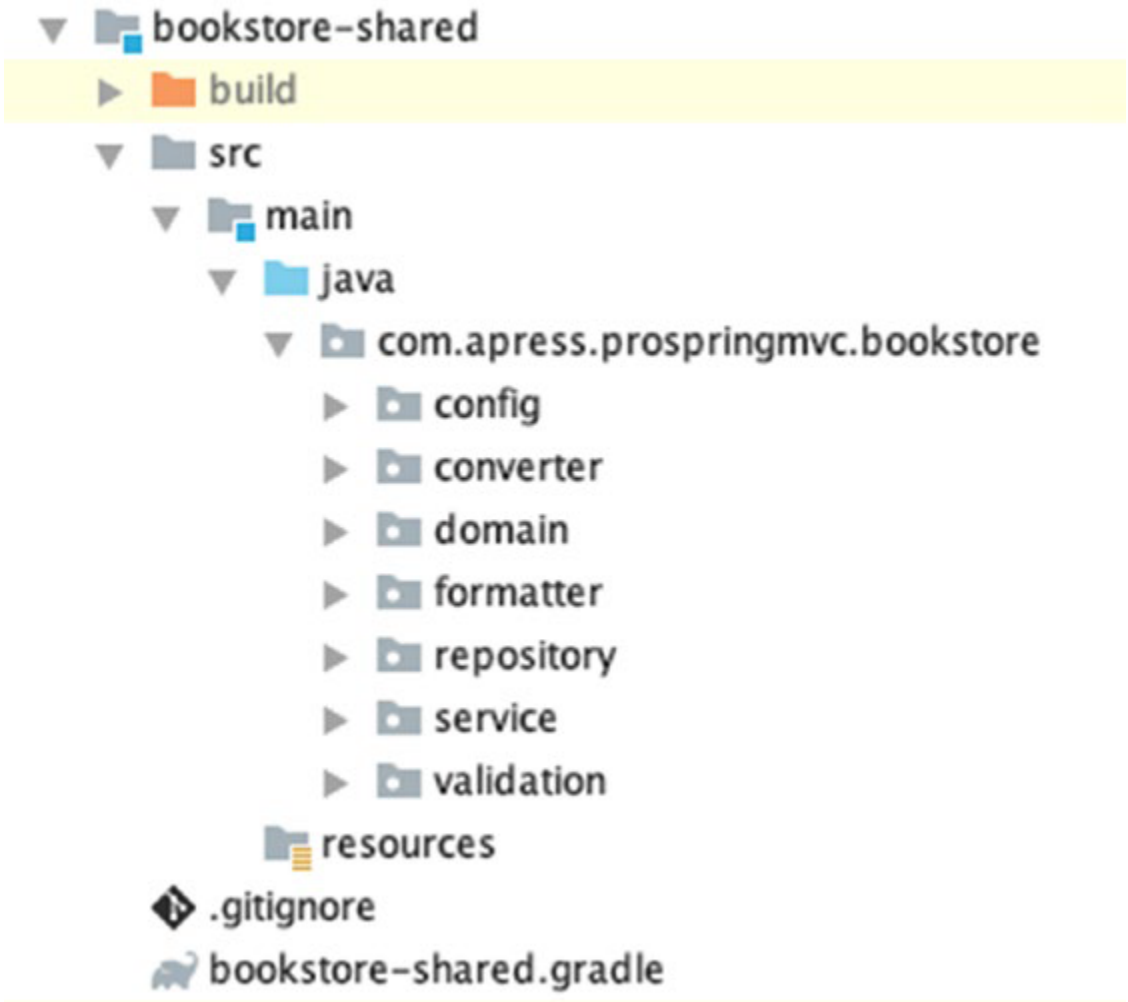
Service The entry point to the actual system containing the business logic. It provides a coarse-grained interface that enables the use of the system. It is also the layer that should be the system's transactional boundary (and probably security, too). This layer shouldn't know anything (or as little as possible) about persistence or the view technology used.

Data Access An interface-based layer provides access to the underlying data access technology

without exposing it to the upper layers. This layer abstracts the actual persistence framework (e.g., JDBC, JPA, or something like MongoDB). Note that this layer

should not contain business logic.





The Domain Layer

The domain is the most important layer in an application. It is the code representation

of the business problem we are solving, and it contains the business rules of our domain. These rules might check whether we have sufficient funds to transfer money from our account or ensure that fields are unique (e.g., usernames in our system).

The User Interface Layer

The user interface layer presents the application to the user. This layer renders the response generated by the server into the type requested by the user's client. For instance, a web browser will probably request an HTML document, a web service may want an XML document, and another client could request a PDF or Excel document.

The Web Layer

The web layer has two responsibilities. The first responsibility is to guide the user through the web application. The second is to be the integration layer between the service layer and HTTP.

Navigating the user through the website can be as simple as mapping a URL to views or a full-blown page flow solution like Spring Web Flow. The navigation is typically bound to the web layer only, and there isn't any navigation logic in the domain or service layer.

As an integration layer, the web layer should be as thin as possible. It should be the layer that converts the incoming HTTP request to something that can be handled by the service layer and then transforms the result (if any) from the server into a response for the user interface. The web layer should not contain any business logic—that is the sole purpose of the service layer.

The web layer also consists of cookies, HTTP headers, and possibly an HTTP session. It is the responsibility of the web layer to manage all these things consistently and transparently. The different HTTP elements should never creep into our service layer.

The Service Layer

The service layer is very important in the architecture of an application. It is considered the heart of our application because it exposes the system's functionality (the use cases) to the user. It does this by providing a coarse-grained API (as mentioned in Table 3-2). Listing 3-1 describes a coarse-grained service interface.

Listing 3-1. A Coarse-Grained Service Interface

```
package com.apress.prospringmvc.bookstore.service;
import com.apress.prospringmvc.bookstore.domain.Account;
public interface AccountService {
Account save(Account account);
Account login(String username, String password) throws
AuthenticationException;
Account getAccount(String username);
}
```

This listing is considered coarse-grained because it takes a simple method call from the client to complete a single use case. This contrasts with the code in Listing 3-2

(fine-grained
service methods), which requires a couple of calls to perform a use case.
Listing 3-2. A Fine-Grained Service Interface

```
package com.apress.prospringmvc.bookstore.service;
import com.apress.prospringmvc.bookstore.domain.Account;
public interface AccountService {
Account save(Account account);
Account getAccount(String username);
void checkPassword(Account account, String password);
void updateLastLogin(Account account);
}
```

The Data Access Layer

The data access layer is responsible for interfacing with the underlying persistence mechanism. This layer knows how to store and retrieve objects from the datastore. It does this in such a way that the service layer doesn't know which underlying datastore is used. (The datastore could be a database, but it could also consist of flat files on the file system.)

There are several reasons for creating a separate data access layer. First, we don't want to burden the service layer with knowledge of the kind of datastore (or datastores) we use; we want to transparently handle persistence. In our sample application, we use an in-memory database and JPA (Java Persistence API) to store our data. Now imagine that, instead of coming from the database, our com.apress.prospringmvc.bookstore.domain.Account comes from an Active Directory Service. We could simply create a new implementation of the interface that knows how to deal with Active Directory—all without changing our service layer. In theory, we could easily swap out implementations; for example, we could switch from JDBC to Hibernate without changing the service layer. It is unlikely that this will happen, but it is nice to have this ability.

DispatcherServlet Request Processing Workflow

In the previous chapter, you learned about the important role the front controller plays in a Model 2 MVC pattern. The front controller takes care of dispatching incoming requests to the correct handler and prepares the response to be rendered into something that the user would like to see. The role of the front controller in Spring MVC is played by org.springframework.web.servlet.DispatcherServlet. This servlet uses several components to fulfill its role. All these components are expressed as interfaces, for which one or more implementations are available. The next section explores the general role these components play in the request processing workflow. Another upcoming section covers the different implementations of the interfaces.

The DispatcherServlet Components Used in Request Processing Workflow

Component Type	Description
org.springframework.web.multipart.MultipartResolver	

Strategy interface to handle multipart form processing

org.springframework.web.servlet.LocaleResolver

Strategy for locale resolution and modification

org.springframework.web.servlet.ThemeResolver

Strategy for theming resolution and modification

org.springframework.web.servlet.HandlerMapping

Strategy to map incoming requests to handler objects

org.springframework.web.servlet.HandlerAdapter

Strategy for the handler object type to execute the handler

org.springframework.web.servlet.HandlerExceptionResolver

Strategy to handle exceptions thrown during handler execution

org.springframework.web.servlet.RequestToViewNameTranslator

Strategy to determine a view name when the handler returns none

org.springframework.web.servlet.ViewResolver

Strategy to translate the view name to an actual view implementation

org.springframework.web.servlet.FlashMapManager

Strategy to simulate flash scope

Bootstrapping DispatcherServlet

The servlet specification (as of version 3.0) has several options for configuring and registering a servlet.

- Option 1: Use a web.xml file (see Listing 4-1).
- Option 2: Use a web-fragment.xml file (see Listing 4-2).
- Option 3: Use javax.servlet.ServletContainerInitializer (see Listing 4-3).
- Option 4: The sample application uses Spring 5.2, so you can get a fourth option by implementing the org.springframework.web.WebApplicationInitializer interface.
- Option 5: Use Spring Boot to autoconfigure DispatcherServlet.

The web.xml Configuration (Servlet 4.0)

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
version="4.0" metadata-complete="true">
<servlet>
<servlet-name>bookstore</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>bookstore</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Using web-fragment.xml

The web-fragment.xml feature has been available since the 3.0 version of the servlet specification, and it allows a more modularized configuration of the web application. The web-fragment.xml has to be in the META-INF directory of a jar file. It isn't detected in the web application's META-INF; it must be in a jar file. web-fragment.xml can contain the same elements as web.xml (see Listing 4-2).

The benefit of this approach is that each module packaged as a jar file contributes to the configuration of the web application. This is also considered a drawback because now you have scattered your configuration over your code base, which could be troublesome in larger projects.

Listing 4-2. The web-fragment.xml Configuration (Servlet 4.0)

```
<web-fragment xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-fragment_4_0.xsd"
version="4.0" metadata-complete="true">
<servlet>
<servlet-name>bookstore</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>bookstore</servlet-name>
<url-pattern>/*</url-pattern>
```

```
</servlet-mapping>
</web-fragment>
```

Using ServletContainerInitializer

The 3.0 version of the servlet specification introduced the option to use a Java-based approach to configuring your web environment (see Listing 4-3). A Servlet 3.0+ compatible container scan the classpath for classes that implement the javax.servlet.ServletContainerInitializer interface, and it invokes the onStartup method on those classes. By adding a javax.servlet.annotation.HandlesTypes annotation on these classes, you can also be handed classes that you need to further configure your web application (this is the mechanism that allows the fourth option to use org.springframework.web.WebApplicationInitializer).

Like web fragments, ServletContainerInitializer allows a modularized configuration of your web application, but now in a Java-based way. Using Java gives you all the added benefits of using the Java language instead of XML. At this point, you have strong typing, can influence the construction of your servlet, and have an easier way of configuring your servlets (in an XML file, this is done by adding init-param and/or context-param elements in the XML file).

A Java-based Configuration

```
package com.apress.prospringmvc.bookstore.web;
import java.util.Set;
// javax.servlet imports omitted.
import org.springframework.web.servlet.DispatcherServlet;
public class BookstoreServletContainerInitializer
implements ServletContainerInitializer {
@Override
public void onStartup(Set<Class<?>> classes, ServletContext
servletContext)
throws ServletException {
ServletRegistration.Dynamic registration;
registration = servletContext.addServlet("ds", DispatcherServlet.
class);
registration.setLoadOnStartup(1);
registration.addMapping("/");
}
}
```

Using WebApplicationInitializer

Now it’s time to look at option 4 for configuring your application while using Spring. Spring provides a ServletContainerInitializer implementation (org.springframework.web.SpringServletContainerInitializer) that makes life a little easier (see Listing 4-4). The implementation provided by the Spring Framework detects and instantiates all instances of org.springframework.web.WebApplicationInitializer and calls the onStartup method of those

instances.

Listing 4-4. The WebApplicationInitializer Configuration

```
package com.apress.prospringmvc.bookstore.web;
// javax.servlet imports omitted
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.servlet.DispatcherServlet;
public class BookstoreWebApplicationInitializer
implements WebApplicationInitializer {
@Override
public void onStartup(ServletContext servletContext)
throws ServletException {
ServletRegistration.Dynamic registration
registration = servletContext.addServlet("dispatcher",
DispatcherServlet.class);
registration.addMapping("/");
registration.setLoadOnStartup(1);
}
}
```

Using Spring Boot

When using Spring Boot, you don't need to configure DispatcherServlet manually. Spring Boot automatically configure based on the detected configuration. The properties mentioned in Table 4-2 are mostly configurable through properties in the spring.mvc namespace. See Listing 4-5 for a basic sample.

Listing 4-5. BookstoreApplication Using Spring Boot

```
package com.apress.prospringmvc.bookstore;
@SpringBootApplication
public class BookstoreApplication {
public static void main(String[] args) {
SpringApplication.run(BookstoreApplication.class, args);
}
}
```

A specialized WebApplicationInitializer is needed when using Spring Boot in a classic WAR application. Spring Boot provides the SpringBootServletInitializer for this. See Listing 4-6 for a sample.

Listing 4-6. BookstoreApplication Using Spring Boot in a WAR

```
package com.apress.prospringmvc.bookstore;
@SpringBootApplication
public class BookstoreApplication extends SpringBootServletInitializer {
public static void main(String[] args) {
SpringApplication.run(BookstoreApplication.class, args);
}
@Override
protected SpringApplicationBuilder configure(SpringApplicationBuilder
```

```
builder) {
return builder.sources(BookstoreApplication.class);
}
}
```

Configuring DispatcherServlet

Configuring org.springframework.web.servlet.DispatcherServlet is a two-step process. The first step is to configure the servlet's behavior by setting properties directly on the dispatcher servlet (the declaration). The second step is to configure the components in the application context (initialization).

The dispatcher servlet comes with a lot of default settings for components. This saves you from doing a lot of configuration for basic behavior, and you can override and extend the configuration however you want. In addition to the default configuration for the dispatcher servlet, there is also a default for Spring MVC. This can be enabled by using an org.springframework.web.servlet.config.annotation.EnableWebMvc annotation (see the "Enabling Features" section in Chapter 2).

DispatcherServlet Properties

The dispatcher servlet has several properties that can be set. All these properties have a setter method, and all can be either set programmatically or by including a servlet initialization parameter. Table 4-2 lists and describes the properties available on the dispatcher servlet.

Property	Default	Description
`cleanupAfterInclude`	True	Indicates whether to clean up the request attributes after an include request. In general, the default suffices, and this property should only be set to false in special cases.
`contextAttribute`	Null	Stores the application context for this servlet. It is useful if the application context is created by some means other than the servlet itself.
`contextClass`	`org.springframework.web.context.support.XmlWebApplicationContext`	Configures the type of `org.springframework.web.context.WebApplicationContext` to be constructed by the servlet (it needs a default constructor). Configured using the given `contextConfigLocation`. It isn't needed if you pass in an application context by using the constructor.
`contextConfigLocation`	`[servlet-name]-servlet.xml`	Indicates the location of the configuration files for the specified application context class.
`contextId`	Null	Provides the application context ID. For example, this is used when the context is logged or sent to `System.out`.
`contextInitializers contextInitializerClasses`	Null	Use the optional `org.springframework.context.ApplicationContextInitializer` classes to do some initialization logic for the application context, such as activating a certain profile.

Property	Default	Description
detectAllHandlerAdapters	True	Detects all org.springframework.web.servlet.HandlerAdapter instances from the application context. When set to false, a single one is detected by using the special name, handlerAdapter.
detectAllHandlerExceptionResolvers	True	Detects all org.springframework.web.servlet.HandlerExceptionResolver instances from the application context. When set to false, a single one is detected by using the special name, handlerExceptionResolver.
detectAllHandlerMappings	True	Detects all org.springframework.web.servlet.HandlerMapping beans from the application context. When set to false, a single one is detected by using the special name, handlerMapping.
detectAllViewResolvers	True	Detects all org.springframework.web.servlet.ViewResolver beans from the application context. When set to false, a single one is detected by using the special name, viewResolver.
dispatchOptionsRequest	False	Indicates whether to handle HTTP OPTIONS requests. The default is false; when set to true, you can also handle HTTP OPTIONS requests.

Property	Default	Description
dispatchTraceRequest	False	Indicates whether to handle HTTP TRACE requests. The default is false; when set to true, you can also handle HTTP TRACE requests.
environment	org.springframework.web.context.support.StandardServletEnvironment	Configures org.springframework.core.env.Environment for this servlet. The environment specifies which profile is active and can hold properties specific to this environment.
namespace	[servletname]-servlet	Use this namespace to configure the application context.
publishContext	True	Indicates whether the servlet's application context is being published to the javax.servlet.ServletContext. For production, we recommend that you set this to false.
publishEvents	True	Indicates whether to fire after request processing org.springframework.web.context.support.ServletRequestHandledEvent. You can use org.springframework.context.ApplicationListener to receive these events.
threadContextInheritable	False	Indicates whether to expose the LocaleContext and RequestAttributes to child threads created from the request handling thread.

The Spring MVC Components

In the previous sections, you learned about the request processing workflow and the components used in it. You also learned how to configure org.springframework.web.servlet.DispatcherServlet. In this section, you take a closer look at all the components involved in the request processing workflow. For example, you explore the different components' APIs and see which implementations ship with the Spring

Framework.

HandlerMapping
Handler mapping determines which handler to dispatch the incoming request to. A criterion that you could use to map the incoming request is the URL; however, implementations (see Figure 4-10) are free to choose what criteria to use to determine the mapping.
The API for org.springframework.web.servlet.HandlerMapping consists of a single method (see Listing 4-9). This method is called by DispatcherServlet to determine org.springframework.web.servlet.HandlerExecutionChain. It is possible to have more than one handler mapping configured. The servlet calls the different handler mappings in sequence until one of them doesn't return null.

Listing 4-9. The HandlerMapping API
```
package org.springframework.web.servlet;
import javax.servlet.http.HttpServletRequest;
public interface HandlerMapping {
HandlerExecutionChain getHandler(HttpServletRequest request)
throws Exception;
}
```

BeanNameUrlHandlerMapping
The org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping implementation is one of the default strategies used by the dispatcher servlet. This implementation treats any bean with a name that starts with a / as a potential request handler. A bean can have multiple names, and names can also contain a wildcard, expressed as an *.

This implementation uses ant-style regular expressions to match the URL of the incoming request to the name of a bean. It follows this algorithm.
1. Attempt exact match; if found, exit.
2. Search all registered paths for a match; the most specific one wins.
3. If no matches are found, return the handler mapped to /* or to the default handler (if configured).

Listing 4-10 shows how to use a bean name and map it to the /index.htm URL. In the sample application, you could now use http://localhost:8080/chapter4-bookstore/index.htm to call this controller.
Listing 4-10. The BeanNameUrlHandlerMapping sample Configuration
```
package com.apress.prospringmvc.bookstore.web.config;
import java.util.Properties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.apress.prospringmvc.bookstore.web.IndexController;
```

```
@Configuration
public class WebMvcContextConfiguration {
@Bean(name = { "/index.htm" })
public IndexController indexController() {
return new IndexController();
}
}
```

SimpleUrlHandlerMapping

This implementation requires explicit configuration, as opposed to org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping, and it doesn't autodetect mappings. Listing 4-11 shows a sample configuration. Again, you map the controller to the /index.htm.

Listing 4-11. The SimpleUrlHandlerMapping Sample Configuration

```
package com.apress.prospringmvc.bookstore.web.config;
// Other imports omitted see Listing 4-10
import org.springframework.web.servlet.HandlerMapping;
import org.springframework.web.servlet.handler.SimpleUrlHandlerMapping;
@Configuration
public class WebMvcContextConfiguration {
@Bean
public IndexController indexController() {
return new IndexController();
}
@Bean
public HandlerMapping simpleUrlHandlerMapping() {
var mappings = new Properties();
mappings.put("/index.htm", "indexController");
var urlMapping = new SimpleUrlHandlerMapping();
urlMapping.setMappings(mappings);
return urlMapping;
}
}
```

RequestMappingHandlerMapping

The RequestMappingHandlerMapping implementation is more sophisticated. It uses annotations to configure mappings. The annotation can be on either the class and/or the method level. To map the com.apress.prospringmvc.bookstore.web.IndexController to /index.htm, you need to add the @RequestMapping annotation. Listing 4-12 is the controller, and Listing 4-13 shows the sample configuration.

Listing 4-12. The IndexController with RequestMapping

```
package com.apress.prospringmvc.bookstore.web;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.servlet.ModelAndView;
@Controller
public class IndexController {
@RequestMapping(value = "/index.htm")
public ModelAndView indexPage() {
return new ModelAndView("/WEB-INF/views/index.jsp");
}
}
```

Listing 4-13. An annotation-based sample Configuration

```
package com.apress.prospringmvc.bookstore.web.config;
// Other imports omitted see Listing 4-10
@Configuration
public class WebMvcContextConfiguration {
@Bean
public IndexController indexController() {
return new IndexController();
}
}
```

RouterFunctionMapping

The org.springframework.web.servlet.function.support.HandlerFunctionAdapter implementation is a functional way to define handlers. Listing 4-14 shows the functional style of writing a handler to render the index page.

Listing 4-14. A Functional-Style Sample Configuration

```
package com.apress.prospringmvc.bookstore.web.config;
// Other imports omitted see Listing 4-10
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerRequest;
import org.springframework.web.servlet.function.ServerResponse;
@Configuration
public class WebMvcContextConfiguration {
@Bean
public RouterFunction<ServerResponse> routes() {
return route()
.GET("/", response -> ok().render("index"))
.build();
}
}
```

HandlerAdapter

org.springframework.web.servlet.HandlerAdapter is the glue between the dispatcher servlet and the selected handler. It removes the actual execution logic from the dispatcher servlet, which makes the dispatcher servlet infinitely extensible. Consider this component the glue between the servlet and the actual handler implementation.

Listing 4-15 shows the HandlerAdapter API.

The HandlerAdapter API

```
package org.springframework.web.servlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public interface HandlerAdapter {
boolean supports(Object handler);
ModelAndView handle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception;
long getLastModified(HttpServletRequest request, Object handler);
}
```

HttpRequestHandlerAdapter

org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter knows how to execute org.springframework.web.HttpRequestHandler instances. This handler adapter is mostly used by Spring Remoting to support some of the HTTP remoting options. However, there are two implementations of the org.springframework.web.HttpRequestHandler interface that you also use. One serves static resources, and the other forwards incoming requests to the servlet container's default servlet (see Chapter 5 for more information).

SimpleControllerHandlerAdapter

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter knows how to execute org.springframework.web.servlet.mvc.Controller implementations. It returns org.springframework.web.servlet.ModelAndView from the handleRequest method of the controller instance.

SimpleServletHandlerAdapter

It can be convenient to configure javax.servlet.Servlet instances in the application context and put them behind the dispatcher servlet. To execute those servlets, you need org.springframework.web.servlet.handler.SimpleServletHandlerAdapter. It knows how to execute javax.servlet.Servlet, and it always returns null because it expects the servlet to handle the response itself.

MultipartResolver

The org.springframework.web.multipart.MultipartResolver strategy interface determines whether an incoming request is a multipart file request (for file uploads), and if so, it wraps the incoming request in org.springframework.web.multipart. MultipartHttpServletRequest. The wrapped request can then get easy access to the underlying multipart files from the form. File uploading is explained in Chapter 7.

Listing 4-16 shows the MultipartResolver API.

Listing 4-16. The MultipartResolver API

```
package org.springframework.web.multipart;
import javax.servlet.http.HttpServletRequest;
public interface MultipartResolver {
boolean isMultipart(HttpServletRequest request);
```

```
MultipartHttpServletRequest resolveMultipart(HttpServletRequest request)
throws MultipartException;
void cleanupMultipart(MultipartHttpServletRequest request);
}
```

LocaleResolver

The org.springframework.web.servlet.LocaleResolver strategy interface determines which java.util.Locale to render the page. In most cases, it resolves validation messages or labels in the application. The different implementations are shown in Figure 4-14 and described in the following subsections.

The LocaleResolver API

```
package org.springframework.web.servlet;
import java.util.Locale;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public interface LocaleResolver {
Locale resolveLocale(HttpServletRequest request);
void setLocale(HttpServletRequest request,HttpServletResponse response,
Locale locale);
}
```

AcceptHeaderLocaleResolver

The org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver implementation simply delegates to the getLocale method of the current javax.servlet.HttpServletRequest. It uses the Accept-Language HTTP header to determine the language. The client sets this header value; this resolver doesn't support changing the locale.

CookieLocaleResolver

The org.springframework.web.servlet.i18n.CookieLocaleResolver implementation uses javax.servlet.http.Cookie to store the locale to use. This is particularly useful in cases where you want an application to be as stateless as possible. The actual value is stored on the client side, and it is sent to you with each request. This resolver allows the locale to be changed (you can find more information on this in Chapter 6). This resolver also allows you to configure the name of the cookie and a default locale to use. If no value can be determined for the current request (i.e., there is neither a cookie nor a default locale), this resolver falls back to the request's locale (see AcceptHeaderLocaleResolver).

FixedLocaleResolver

org.springframework.web.servlet.i18n.FixedLocaleResolver is the most basic implementation of org.springframework.web.servlet.LocaleResolver. It allows you to configure a locale to use throughout the whole application. This configuration is fixed; as such, it cannot be changed.

SessionLocaleResolver

The org.springframework.web.servlet.i18n.SessionLocaleResolver implementation uses the javax.servlet.http.HttpSession to store the value of the locale. The name of the attribute, as well as a default locale, can be configured. If no value can be determined for the current request (i.e., there is neither a value stored in the session nor a default locale), then it falls back to the request's locale (see AcceptHeaderLocaleResolver). This resolver also lets you change the locale (see Chapter 6 for more information).

ThemeResolver

The org.springframework.web.servlet.ThemeResolver strategy interface determines which theme to render the page. There are several implementations; these are shown in Figure 4-15 and explained in the following subsections. How to apply theming is explained in Chapter 8. If no theme name can be resolved, then this resolver uses the hardcoded default theme.

Listing 4-18. The ThemeResolver API

```
package org.springframework.web.servlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public interface ThemeResolver {
String resolveThemeName(HttpServletRequest request);
void setThemeName(HttpServletRequest request, HttpServletResponse
response, String themeName);
}
```

CookieThemeResolver

org.springframework.web.servlet.theme.CookieThemeResolver uses javax.servlet.http.Cookie to store the theme to use. This is particularly useful where you want your application to be as stateless as possible. The actual value is stored on the client side and sent to you with each request. This resolver allows the theme to be changed; you can find more information on this in Chapters 6 and 8. This resolver also allows you to configure the name of the cookie and a theme locale to use.

FixedThemeResolver

org.springframework.web.servlet.theme.FixedThemeResolver is the most basic implementation of org.springframework.web.servlet.ThemeResolver. It allows you to configure a theme to use throughout the whole application. This configuration is fixed; as such, it cannot be changed.

SessionThemeResolver

org.springframework.web.servlet.theme.SessionThemeResolver uses javax.servlet.http.HttpSession to store the value of the theme. The name of the attribute, as well as a default theme, can be configured.

HandlerExceptionResolver

In most cases, you want to control how you handle an exception that occurs during the handling of a request. You can use a HandlerExceptionResolver for this. The API (see Listing 4-19) consists of a single method that is called on the org.springframework.web.

servlet.HandlerExceptionResolvers detected by the dispatcher servlet. The resolver can choose to handle the exception itself or to return an org.springframework.web.servlet.ModelAndView implementation that contains a view to render and a model (generally containing the exception thrown).

Listing 4-19. The HandlerExceptionResolver API

```
package org.springframework.web.servlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public interface HandlerExceptionResolver {
ModelAndView resolveException(HttpServletRequest request,
HttpServletResponse response, Object
handler, Exception ex);
}
```

Introducing Controllers

The controller is the component that is responsible for responding to the action the user takes. This action could be a form submission, clicking a link, or simply accessing a page. The controller selects or updates the data needed for the view. It also selects the name of the view to render or can render the view itself. With Spring MVC, we have two options when writing controllers. We can either implement an interface or put an annotation on the class.

Interface-based Controllers

To write an interface-based controller, we need to create a class that implements the org.springframework.web.servlet.mvc.Controller interface. Listing 5-1 shows the API for that interface. When implementing this interface, we must implement the handleRequest method. This method needs to return an org.springframework.web.servlet.ModelAndView object or null when the controller handles the response itself.

Listing 5-1. The Controller Interface

```
package org.springframework.web.servlet.mvc;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
@FunctionalInterface
public interface Controller {
ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception;
}
```

Annotation-based Controllers

To write an annotation-based controller, we need to write a class and put the org.springframework.stereotype.Controller annotation on that class. Also, we need to add an org.springframework.web.bind.annotation.RequestMapping annotation to the class, a method, or both. Listing 5-3 shows an annotation-based approach to our

IndexController.
An Annotation-based IndexController

```
package com.apress.prospringmvc.bookstore.web;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
@Controller
public class IndexController {
@RequestMapping(value = "/index.htm")
public ModelAndView indexPage() {
return new ModelAndView("index");
}
}
```

Configuring View Controllers

The two controller samples we have written so far are called view controllers. They don't select data; rather, they only select the view name to render. If we had a large application with more of these views, it would become cumbersome to maintain and write these. Spring MVC can help us here. Enabling us simply to add org.springframework.web. servlet.mvc.ParameterizableViewController to our configuration and to configure it accordingly. We would need to configure an instance to return index as a view name and map it to the /index.htm URL. Listing 5-4 shows what needs to be added to make this work.

A ParameterizableViewController Configuration

```
package com.apress.prospringmvc.bookstore.web.config;
import org.springframework.web.servlet.mvc.ParameterizableViewController;
// Other imports omitted
@Configuration
public class WebMvcContextConfiguration implements WebMvcConfigurer {
// Other methods omitted
@Bean(name = "/index.htm")
public Controller index() {
ParameterizableViewController index = new
ParameterizableViewController();
index.setViewName("index");
return index;
}
}
```

Request-Handling Methods

Class	Method	Description
	`@RequestMapping(value="/order.htm")`	Maps to all requests on the order.htm URL
`@RequestMapping("/order.htm")`	`@RequestMapping(method=RequestMethod.GET)`	Maps to all GET requests to the order.html URL
`@RequestMapping("/order.*")`	`@RequestMapping(method={RequestMethod.PUT, RequestMethod.POST})`	Maps to all PUT and POST requests to the order.* URL. * means any suffix or extension such as .htm, .doc, .xls, and so on
`@RequestMapping(value="/customer.htm", consumes="application/json")`	`@RequestMapping(produces="application/xml")`	Maps to all requests that post JSON and accept XML as a response
`@RequestMapping(value="/order.htm")`	`@RequestMapping(params="add-line", method=RequestMethod.POST)`	Maps to all POST requests to the order.htm URL that include an add-line parameter
`@RequestMapping(value="/order.htm")`	`@RequestMapping(headers="!VIA")`	Maps to all requests to the order.htm URL that don't include a VIA HTTP header

Supported Method Argument Annotations

In addition to explicitly supported types (as mentioned in the previous section), there are also a couple of annotations that we can use to annotate our method arguments (see Table 5-5). Some of these can also be used with the method argument types mentioned in Table 5-4. In that case, they specify the name of the attribute in the request, cookie, header, or response, as well as whether the parameter is required.

Table 5-5. The Supported Method Argument Annotations

Argument Type	Description
`RequestParam`	Binds the argument to a single request parameter or all request parameters.
`RequestHeader`	Binds the argument to a single request header or all request headers.[5]
`RequestBody`	Gets the request body for arguments with this annotation. The value is converted using org.springframework.http.converter.HttpMessageConverter.
`RequestPart`	Binds the argument to the part of a multipart form submission.
`ModelAttribute`	Binds and validates arguments with this annotation. The parameters from the incoming request are bound to the given object.
`PathVariable`	Binds the method parameter to a path variable specified in the URL mapping (the value attribute of the `RequestMapping` annotation).
`CookieValue`	Binds the method parameter to a `javax.servlet.http.Cookie`.
`SessionAttribute`	Binds the method parameter to a session attribute.
`RequestAttribute`	Binds the method parameter to a request attribute (not to be confused with a request parameter).
`MatrixVariable`	Binds the method parameter to a name-value pair within a path-segment.

The HandlerMethodArgumentResolver Interface

```
package org.springframework.web.method.support;
import org.springframework.core.MethodParameter;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebDataBinderFactory;
import org.springframework.web.context.request.NativeWebRequest;
public interface HandlerMethodArgumentResolver {
boolean supportsParameter(MethodParameter parameter);
Object resolveArgument(MethodParameter parameter,
ModelAndViewContainer mavContainer,
NativeWebRequest webRequest,
WebDataBinderFactory binderFactory)
throws Exception;
}
```