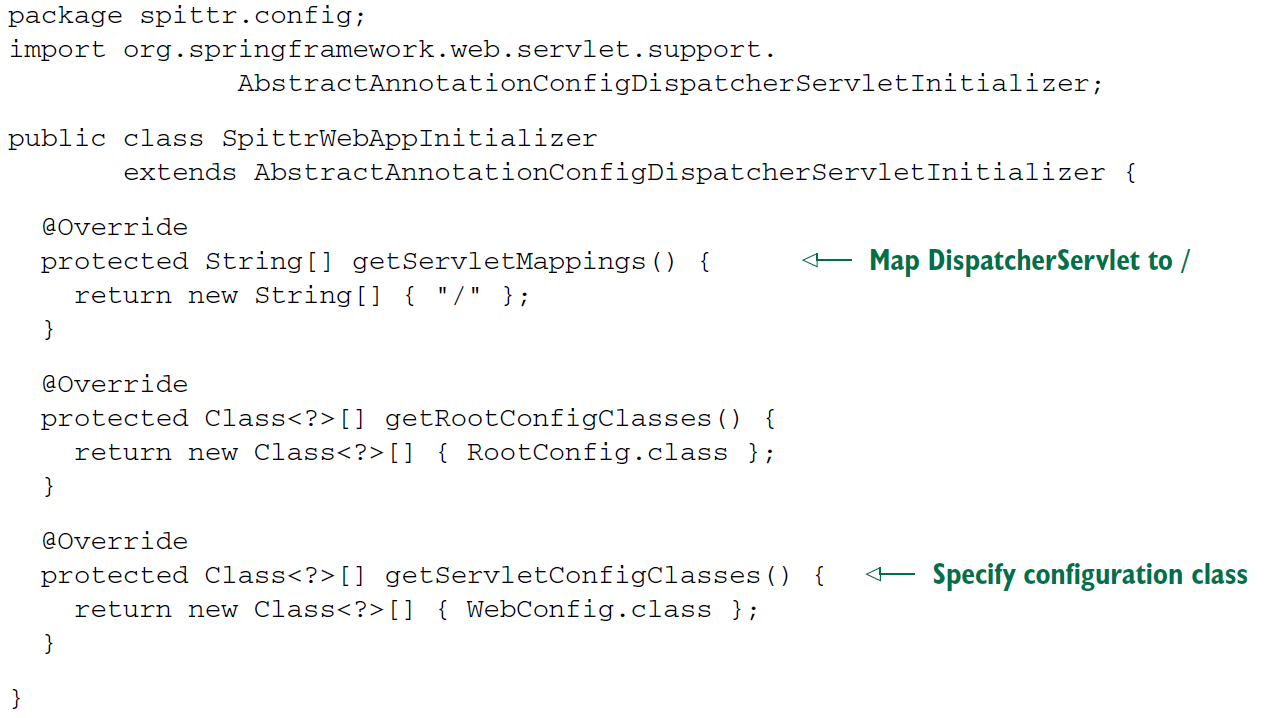
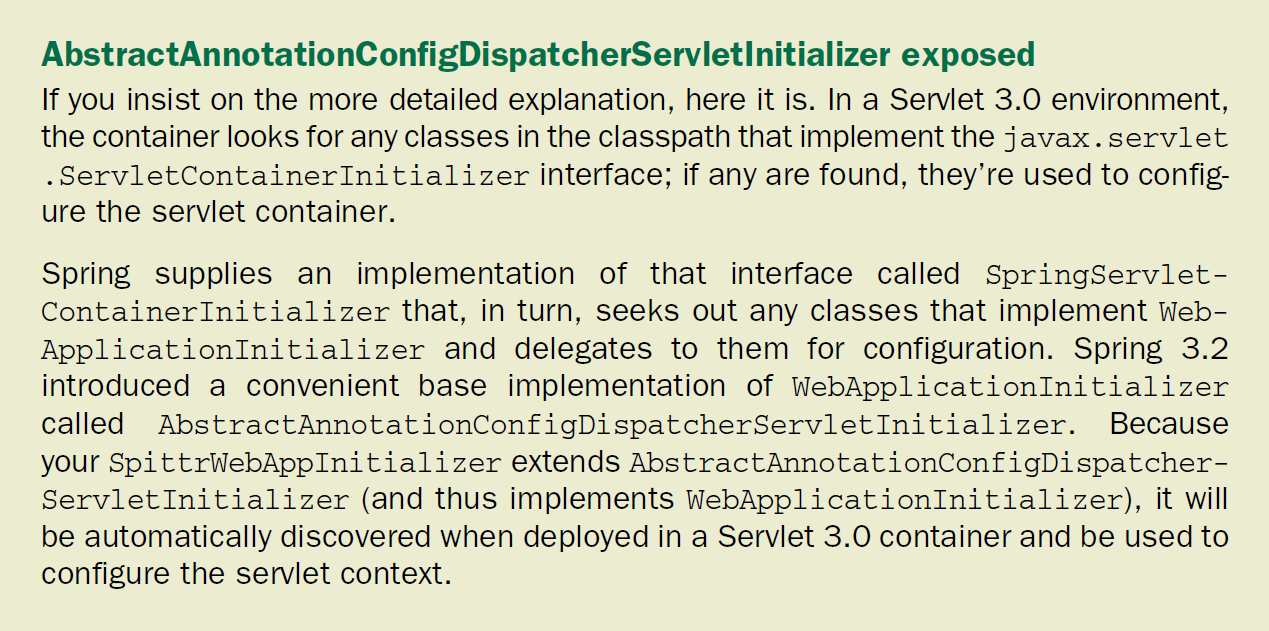
***Setting up Spring MVC***

**CONFIGURING DISPATCHERSERVLET**

* DispatcherServlet is the centerpiece of Spring MVC. It’s where the request first hits the framework, and it’s responsible for routing the request through all the other components.
* Historically, servlets like DispatcherServlet have been configured in a web.xml file that’s carried in the web application’s WAR file. Certainly that’s one option for configuring DispatcherServlet. But thanks to recent advances in the Servlet 3 specification and in Spring 3.1, it’s not the only option.
* Instead of a web.xml file, you’re going to use Java to configure DispatcherServlet in the servlet container. The following listing shows the Java class you’ll need.

****

* To understand how listing 5.1 works, it’s probably sufficient to know that any class that extends **AbstractAnnotationConfigDispatcherServletInitializer** will automatically be used to configure DispatcherServlet and the Spring application context in the application’s servlet context.



* The first method, getServletMappings(), identifies one or more paths that DispatcherServlet will be mapped to. In this case, it’s mapped to /, indicating that it will be the application’s default servlet. It will handle all requests coming into the application.
* In order to understand the other two methods, you must first understand the relationship between DispatcherServlet and a servlet listener known as ContextLoader-Listener.

**A TALE OF TWO APPLICATION CONTEXTS**

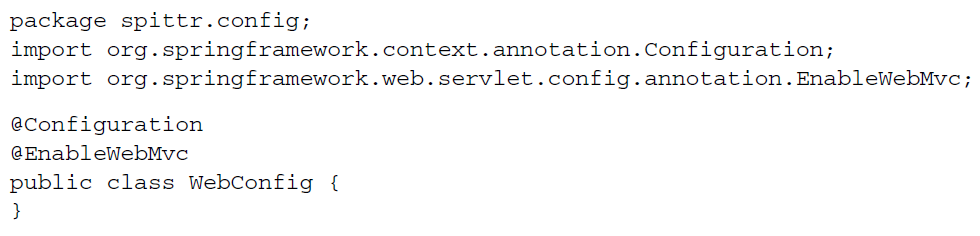
* When DispatcherServlet starts up, it creates a Spring application context and starts loading it with beans declared in the configuration files or classes that it’s given. With the getServletConfigClasses() method, you’ve asked that Dispatcher-Servlet load its application context with beans defined in the WebConfig configuration class (using Java configuration).
* But in Spring web applications, there’s often another application context. This other application context is created by ContextLoaderListener.
* Whereas DispatcherServlet is expected to load beans containing web components such as controllers, view resolvers, and handler mappings, ContextLoaderListener is expected to load the other beans in your application. These beans are typically the middle-tier and data-tier components that drive the back end of the application.
* Under the covers, AbstractAnnotationConfigDispatcherServletInitializer creates both a DispatcherServlet and a ContextLoaderListener.
* The @Configuration classes returned from getServletConfigClasses() will define beans for Dispatcher-Servlet’s application context. Meanwhile, the @Configuration class’s returned get-RootConfigClasses() will be used to configure the application context created by ContextLoaderListener.
* In this case, your root configuration is defined in RootConfig, whereas Dispatcher-Servlet’s configuration is declared in WebConfig. You’ll see what those two configuration classes look like in a moment.
* It’s important to realize that configuring DispatcherServlet via Abstract-AnnotationConfigDispatcherServletInitializer is an alternative to the traditional web.xml file. Although you can include a web.xml file alongside a subclass of AbstractAnnotationConfigDispatcherServletInitializer if you like, it’s not necessary.
* The only gotcha with configuring DispatcherServlet in this way, as opposed to in a web.xml file, is that it will only work when deploying to a server that supports Servlet 3.0, such as Apache Tomcat 7 or higher. The Servlet 3.0 specification has been final since December 2009, and the odds are good that you’ll be deploying your applications to a servlet container that supports Servlet 3.0.
* If you’re not yet working with a Servlet 3.0-capable server, then configuring DispatcherServlet in a subclass of AbstractAnnotationConfigDispatcherServletInitializer won’t work for you. You’ll have no choice but to configure Dispatcher-Servlet in web.xml.
* If you’re not yet working with a Servlet 3.0-capable server, then configuring DispatcherServlet in a subclass of AbstractAnnotationConfigDispatcherServletInitializer won’t work for you. You’ll have no choice but to configure Dispatcher-Servlet in web.xml.

**ENABLING SPRING MVC**

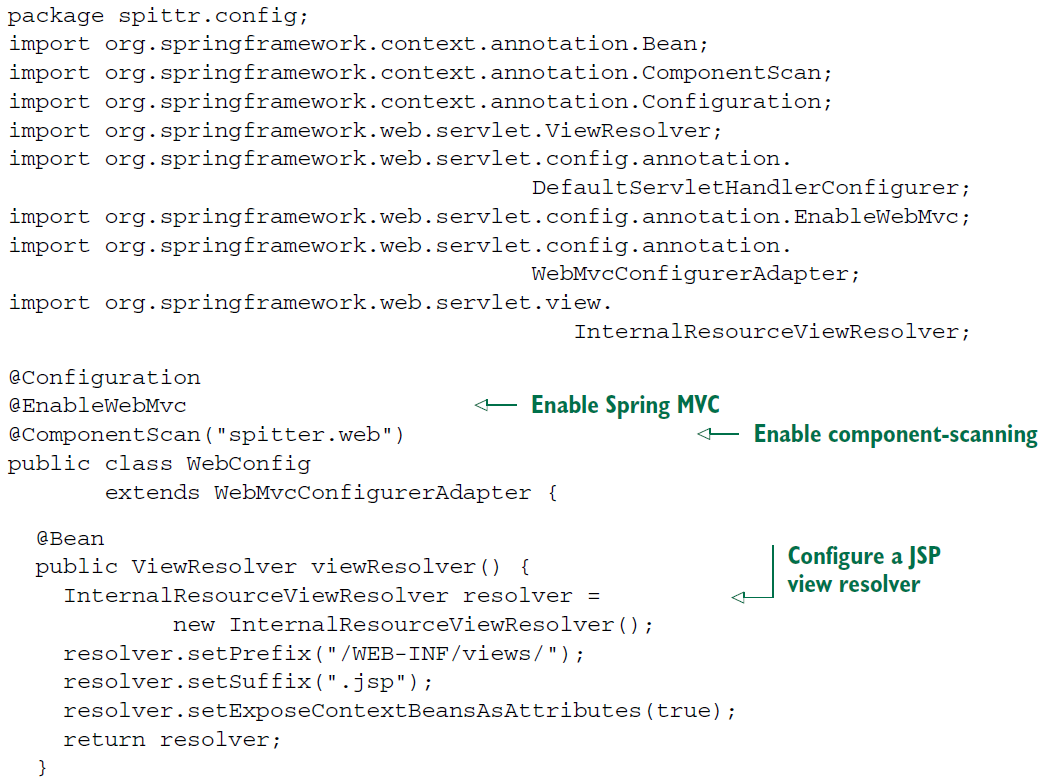
* Just as there are several ways of configuring DispatcherServlet, there’s more than one way to enable Spring MVC components. Historically, Spring has been configured using XML, and there’s an <mvc:annotation-driven> element that you can use to enable annotation-driven Spring MVC.

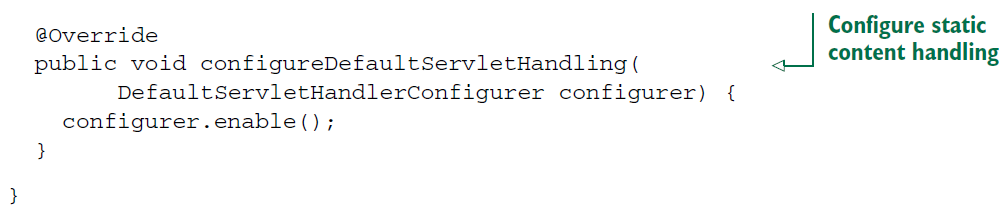
We’ll talk about <mvc:annotation-driven>, among other Spring MVC configuration options, in chapter 7. But for now, you’ll keep your Spring MVC setup simple and Java-based.

* The *very* simplest Spring MVC configuration you can create is a class annotated with @EnableWebMvc:

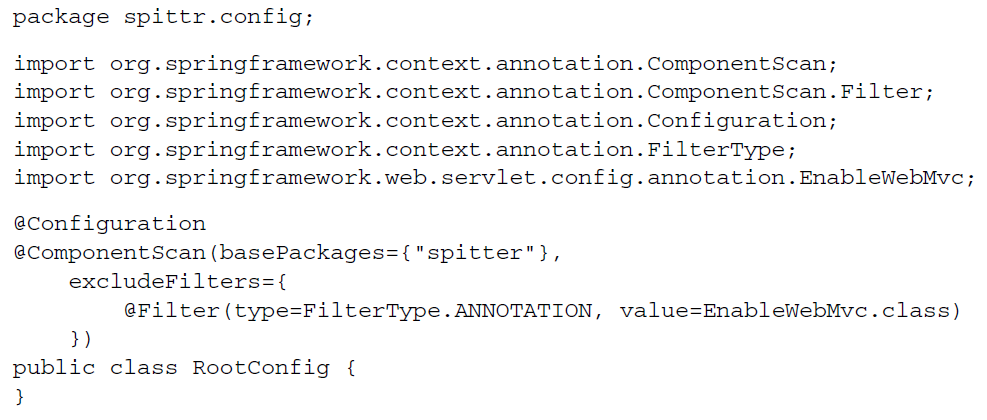


* This will work, and it will enable Spring MVC. But it leaves a lot to be desired:
* No view resolver is configured. As such, Spring will default to using BeanNameViewResolver, a view resolver that resolves views by looking for beans whose ID matches the view name and whose class implements the View interface.
* Component-scanning isn’t enabled. Consequently, the only way Spring will find any controllers is if you declare them explicitly in the configuration.
* As it is, DispatcherServlet is mapped as the default servlet for the application and will handle *all* requests, including requests for static resources, such as images and stylesheets (which is probably not what you want in most cases).
* Therefore, you need to add a bit more configuration in WebConfig on top of this bare minimum Spring MVC configuration to make it useful. The new WebConfig in the next listing addresses these concerns.





* The first thing to notice above is that WebConfig is now annotated with @ComponentScan so that the spitter.web package will be scanned for components. As you’ll soon see, the controllers you write will be annotated with @Controller, which will make them candidates for component-scanning. Consequently, you won’t have to explicitly declare any controllers in the configuration class.
* Next, you add a ViewResolver bean. More specifically, it’s an Internal-ResourceViewResolver. We’ll talk more about view resolvers in chapter 6. For now, just know that it’s configured to look for JSP files by wrapping view names with a specific prefix and suffix (for example, a view name of home will be resolved as /WEB-INF/ views/home.jsp).
* Finally, this new WebConfig class extends WebMvcConfigurerAdapter and overrides its configureDefaultServletHandling() method. By calling enable() on the given DefaultServletHandlerConfigurer, you’re asking DispatcherServlet to forward requests for static resources to the servlet container’s default servlet and not to try to handle them itself.
* Because this chapter is focused on web development, and web configuration is done in the application context created by DispatcherServlet, you’ll keep RootConfig relatively simple for now:



* The only significant thing to note in RootConfig is that it’s annotated with @Component-Scan. There will be plenty of opportunities throughout this book to flesh out Root-Config with non-web components. You’re almost ready to start building a web application with Spring MVC.