***Introducing the Spittr application***

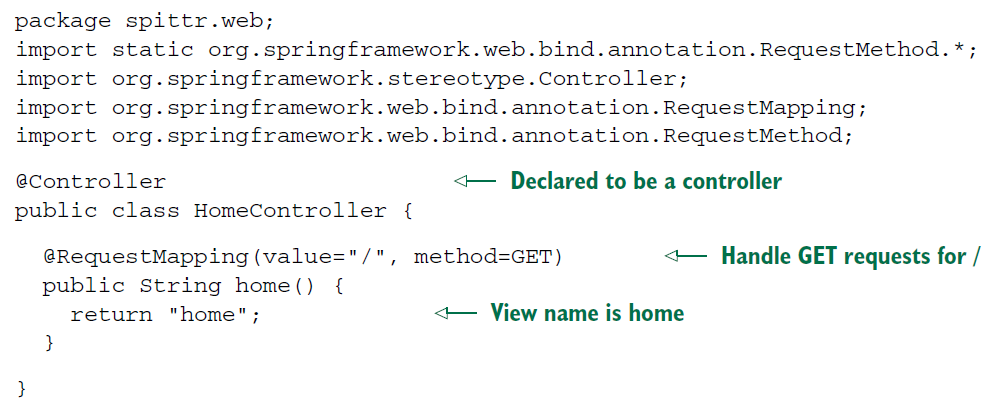
In an attempt to get in on the online social networking game, you’re going to develop a simple microblogging application. In many ways, your application will be much like the original microblogging application, Twitter. You’ll add some little twists on the idea along the way. And, of course, you’ll develop it using Spring.

Borrowing some ideas from Twitter and implementing them in Spring gives the application a working title: Spitter. Taking it a step further and applying a naming pattern hat’s popular with sites like Flickr, let’s drop the *e* and call the app Spittr. This name will also be helpful in differentiating the application name from a domain type you’ll create called Spitter. The Spittr application has two essential domain concepts: *spitters* (the users of the application) and *spittles* (the brief status updates that users publish). We’ll draw primarily on these two domain concepts throughout this book as we flesh out the functionality of the Spittr application. Initially, in this chapter, you’ll build out the web layer of the application, create controllers that display spittles, and process forms where users register as spitters. The stage is now set. You’ve configured DispatcherServlet, enabled essential Spring MVC components, and established a target application. Let’s turn to the meat of the chapter: handling web requests with Spring MVC controllers.

***Writing a simple controller***

* In Spring MVC, controllers are just classes with methods that are annotated with @RequestMapping to declare the kind of requests they’ll handle.

Starting simple, let’s imagine a controller class that handles requests for / and renders the application’s home page. HomeController, shown in the following listing, is an example of what might be the simplest possible Spring MVC controller class.



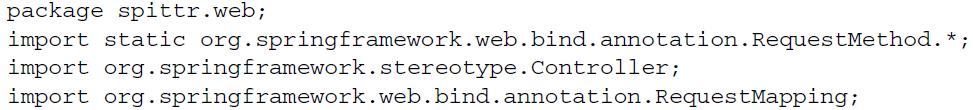
* The first thing you’ll notice about HomeController is that it’s annotated with @Controller. Although it’s clear that this annotation declares a controller, the annotation has little to do with Spring MVC.
* @Controller is a stereotype annotation, based on the @Component annotation. Its purpose here is entirely for the benefit of component-scanning. Because HomeController is annotated with @Controller, the component scanner will automatically pick up HomeController and declare it as a bean in the Spring application context.
* You could have annotated HomeController with @Component, and it would have had the same effect, but it would have been less expressive about what type of component HomeController is.
* HomeController’s only method, the home() method, is annotated with @RequestMapping. The value attribute specifies the request path that this method will handle, and the method attribute details the HTTP method that it can handle.
* In this case, whenever an HTTP GET request comes in for /, the home() method will be called. As you can see, the home() method doesn’t do much: it returns a String value of “home”. This String will be interpreted by Spring MVC as the name of the view that will be rendered. DispatcherServlet will ask the view resolver to resolve this logical view name into an actual view.
* Given the way you configured InternalResourceViewResolver, the view name “home” will be resolved as a JSP at /WEB-INF/views/home.jsp. For now, you’ll keep the Spittr application’s home page rather basic, as shown next.

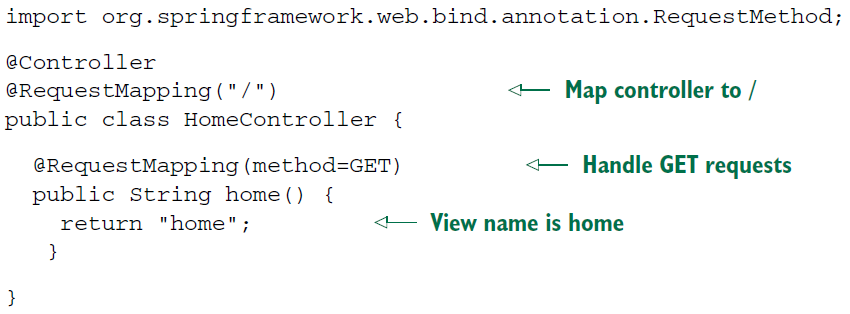


* There’s nothing noteworthy about this JSP. It merely welcomes the user to the application and offers two links: one to view a Spittle list and another to register with the application. But for now, let’s throw some requests at this controller and see if it works. The obvious way to test a controller may be to build and deploy the application and poke at it with a web browser, but an automated test will give you quicker feedback and more consistent hands-off results. So, let’s cover HomeController with a test.

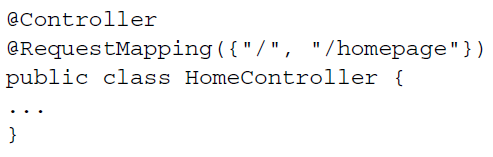
***Defining class-level request handling***

* You can do a bit of refactoring to be certain that nothing breaks. One thing you can do is split up @RequestMapping by placing the path-mapping portion of it at the class level. The next listing shows how this is done.





* In this new version of HomeController, the path has been moved up to a new classlevel @RequestMapping, whereas the HTTP method is still mapped at the method level. Any time there’s a class-level @RequestMapping on a controller class, it applies to all handler methods in the controller. Then any @RequestMapping annotations on handler methods will complement the class-level @RequestMapping.
* In the case of HomeController, there’s only one handler method. Its @RequestMapping, when taken together with the class-level @RequestMapping, indicates that the home() method will handle GET requests for /.
* In other words, you really haven’t changed anything. You’ve moved a few things around, but HomeController still does the same thing as before.
* While you’re tinkering with the @RequestMapping annotations, you can make another tweak to HomeController. The value attribute of @RequestMapping accepts an array of String. So far, you’ve only given it a single String value of “/”. But you can also map it to requests whose path is /homepage by changing the class-level @RequestMapping to look like this:



* Now HomeController’s home() method is mapped to handle GET requests for both / and /homepage requests.