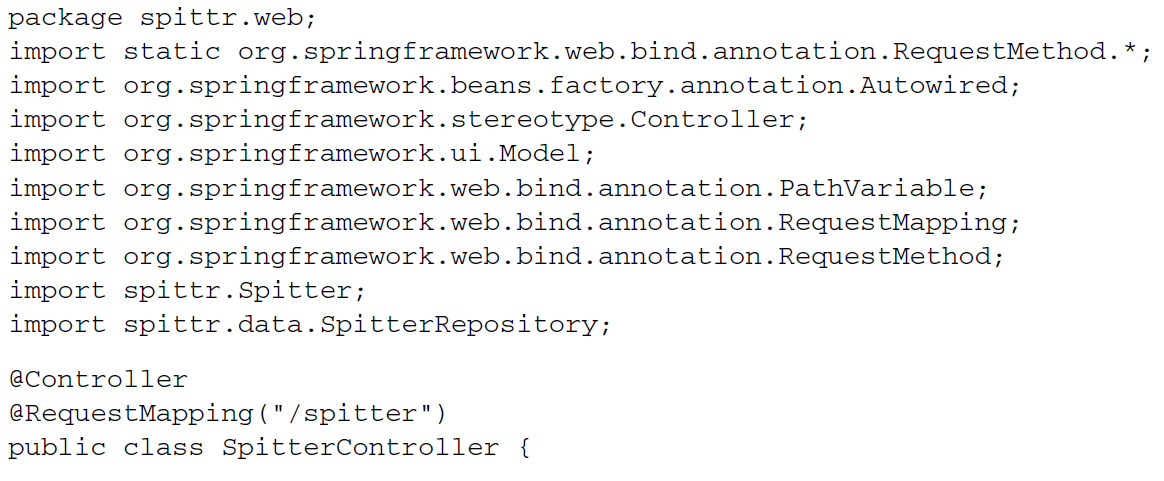
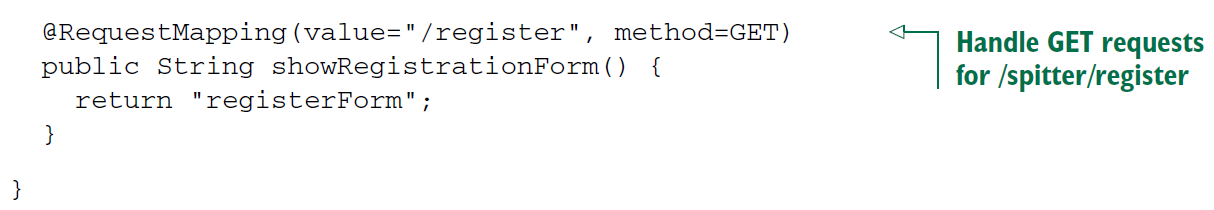
***Processing forms***

Query parameters and path parameters are fine for passing small amounts of data on a request. But often you need to pass a lot of data (perhaps data coming from a form submission), and query parameters are too awkward and limited for that. Let’s see how you can write controller methods that handle form submissions.

Web applications typically do more than just push content out to the user. Most also let users participate in the conversation by filling out forms and submitting data back into the application. Spring MVC controllers are well-suited for form processing as well as serving content.

* SpitterController is a new controller with a single request-handling method for displaying the registration form:

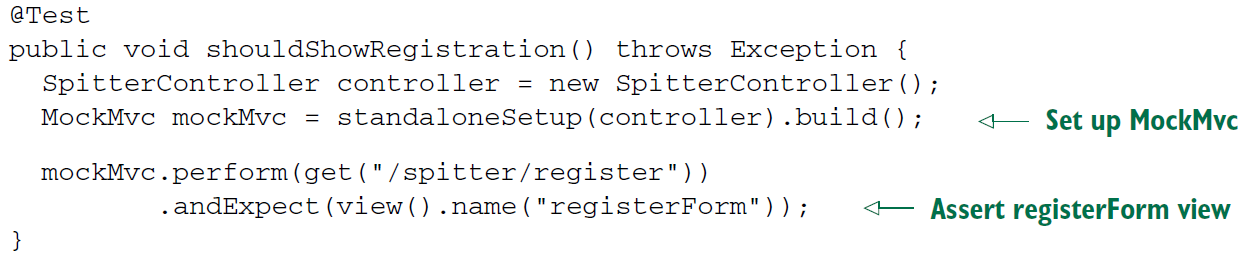




* The showRegistrationForm() method’s @RequestMapping annotation, along with the class-level @RequestMapping annotation, declares that it will handle HTTP GET requests for /spitter/register. It’s a simple method, taking no input and only returning a logical view named registerForm. Given how you’ve configured InternalResourceViewResolver, that means the JSP at

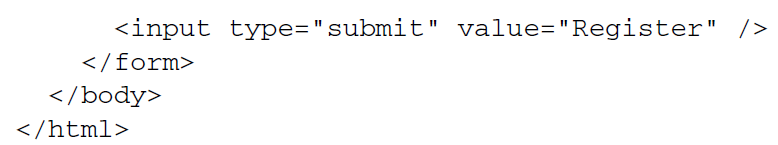
/WEB-INF/views/registerForm.jsp will be called on to render the registration form.

* As simple as showRegistrationForm() is, it still deserves to be covered by a test. Because it’s a simple method, its test will be equally simple.

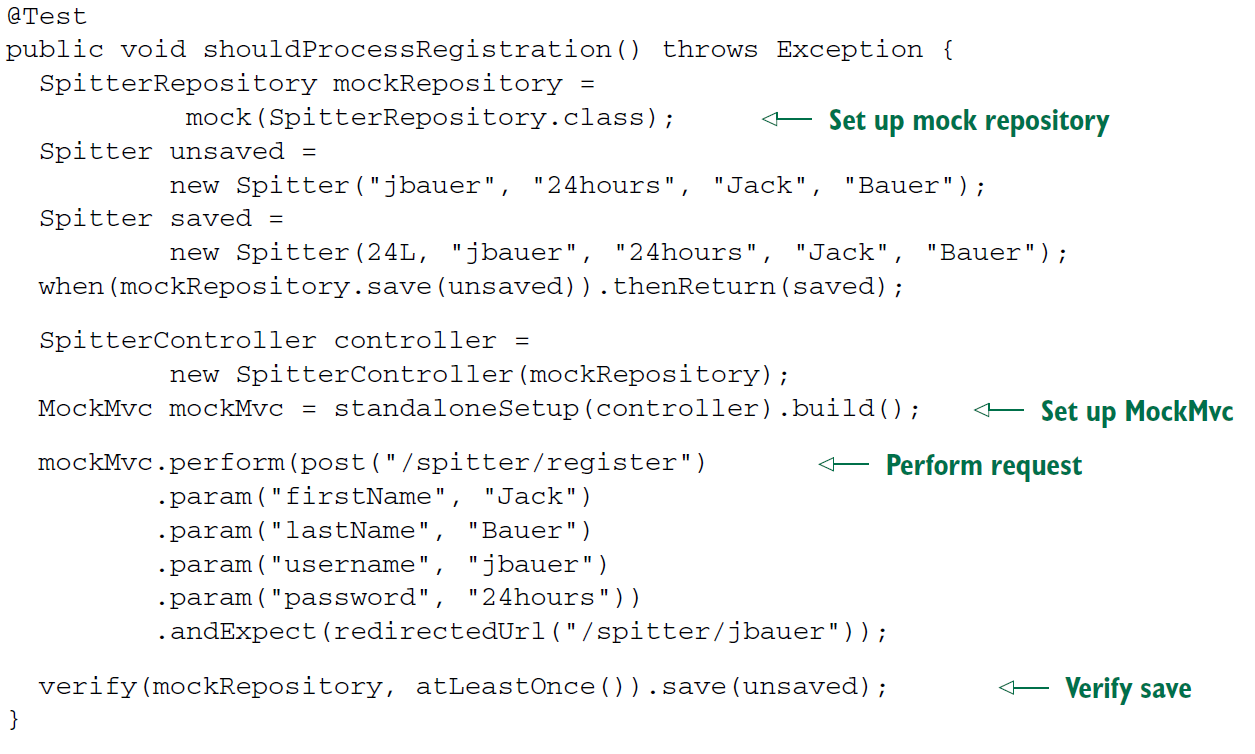


* This test method is very similar to the test for the home page controller method. It performs a GET request for /spitter/register and then asserts that the resulting view is named registerForm.
* Now let’s get back to the view. Because the view name is registerForm, you’ll need a JSP named registerForm.jsp. This JSP must include an HTML <form> where the user will enter information to sign up with the application. Here’s the JSP you’ll use for now:

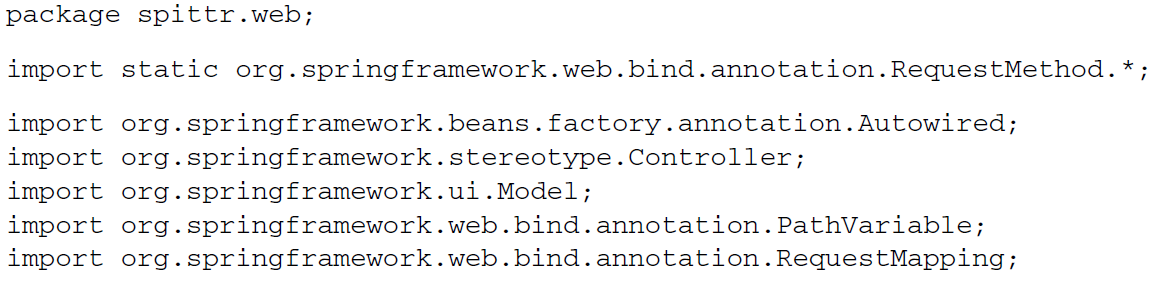


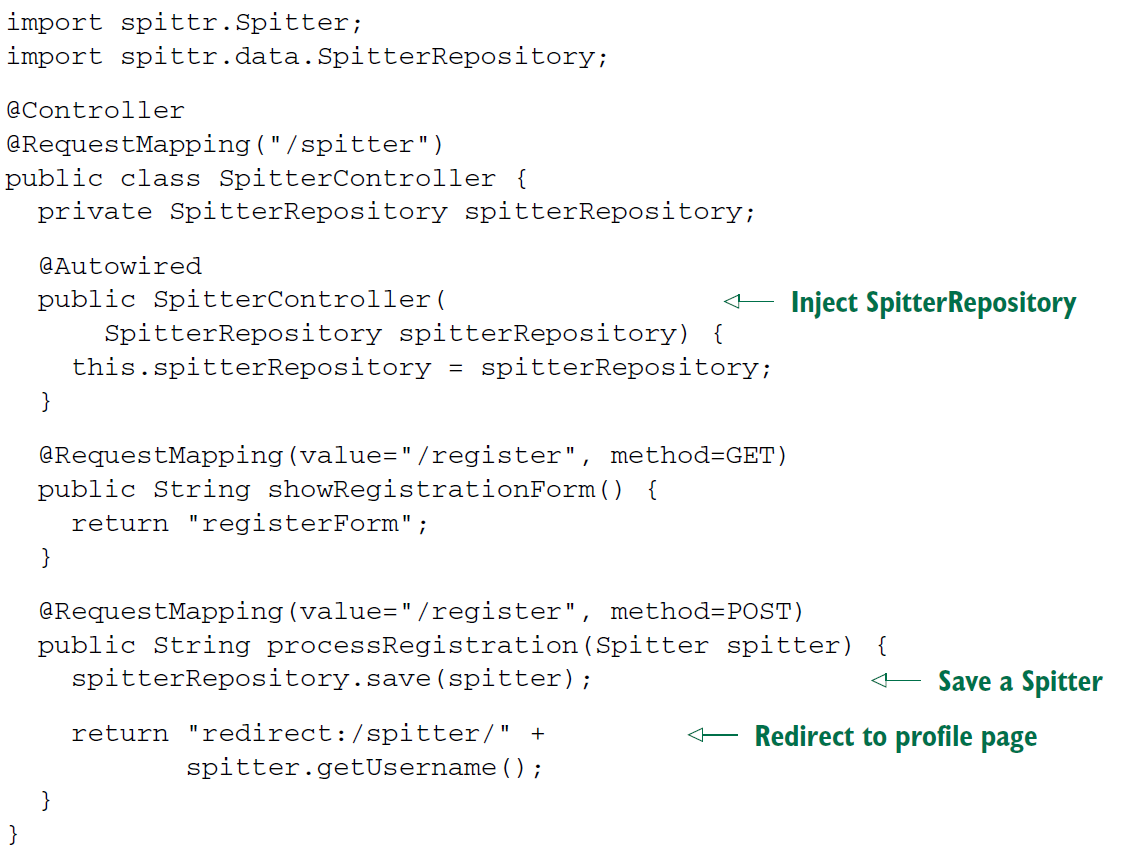


* Notice that the <form> tag doesn’t have an action parameter set. Because of that, when this form is submitted, it will be posted back to the same URL path that displayed it. That is, it will be posted back to /spitters/register.
* That means you’ll need something back on the server to handle the HTTP POST request. Let’s add another method to SpitterController to handle form submission.
* When processing the POST request from the registration form, the controller needs to accept the form data and save the form data as a Spitter object. Finally, in order to prevent a duplicate submission (such as might happen if the user clicked their browser’s Refresh button), it should redirect the browser to the newly created user’s profile page. This behavior is captured and tested in shouldProcessRegistration().



* Clearly, this test is more involved than the test for displaying the registration form. After setting up a mock implementation of SpitterRepository and creating a controller and MockMvc setup to execute against, shouldProcessRegistration() performs a POST request against /spitter/register. As part of that POST request, user information is passed as parameters on the request to simulate a form being submitted.
* When handling a POST request, it’s usually a good idea to send a redirect after the POST has completed processing so that a browser refresh won’t accidentally submit the form a second time. This test expects that the request will end in a redirect to /spitter/jbauer, the URL path of the new user’s profile page.
* Finally, the test verifies that the mocked SpitterRepository was actually used to save the data coming in on the form.
* Now let’s implement the controller method that will handle this form submission test. shouldProcessRegistration() may have left you with the impression that a chunk of work is required to satisfy the test. But as you can see in the new SpitterController in this listing, there’s not much to it:



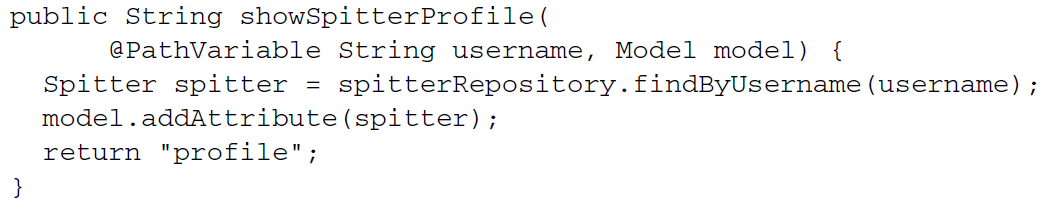


* The showRegistrationForm() method is still in place. **But notice the new processRegistration() method: it’s given a Spitter object as a parameter. This object has firstName, lastName, username, and password properties that will be populated from**

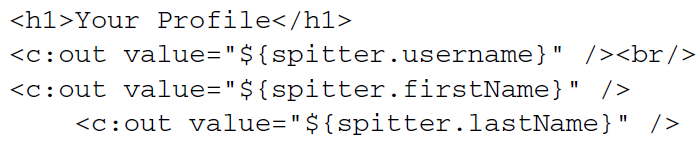
**the request parameters of the same name.**

* Once it’s called with the Spitter object, processRegistration() calls the save() method on the SpitterRepository that is now injected into SpitterController in the constructor.
* The last thing that processRegistration() does is return a String specifying the view. But this view specification is different from what you’ve seen before. Rather than just return a view name and let the view resolver sort it out, here you’re returning a redirect specification.
* When InternalResourceViewResolver sees the redirect: prefix on the view specification, it knows to interpret it as a redirect specification instead of as a view name. In this case, it will redirect to the path for a user’s profile page. For example, if the Spitter.username property is jbauer, then the view will redirect to /spitter/jbauer.
* It’s worth noting that in addition to redirect:, InternalResourceViewResolver also recognizes the forward: prefix. When it sees a view specification prefixed with forward:, the request is forwarded to the given URL path instead of redirected.
* Because you’re redirecting to the user’s profile page, you should probably add a handler method to SpitterController to handle requests for the profile page. Here’s a showSpitterProfile() method that will do the trick:

****

****

* showSpitterProfile() fetches a Spitter object from the SpitterRepository by the username. It adds the Spitter to the model and then returns profile, the logical view name for the profile view. Like all the other views presented in this chapter, you’ll keep the profile view simple for now:

****