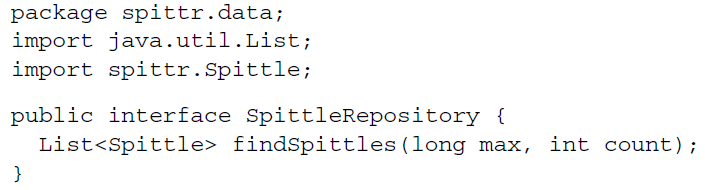
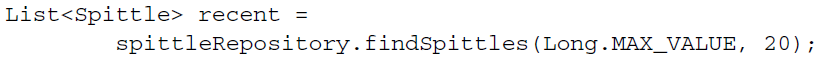
***Passing model data to the view***

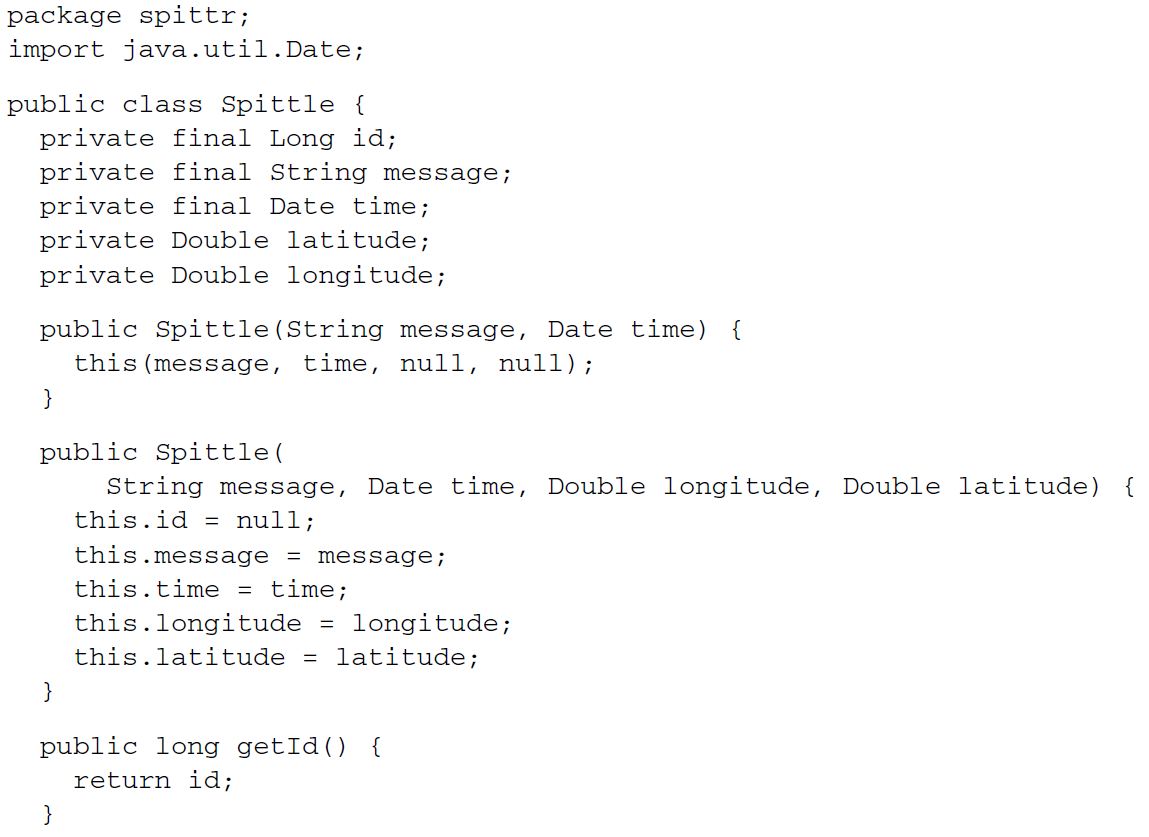
* As it stands now, HomeController is a great example of how to write an extremely simple controller. But most controllers aren’t this simple. In the Spittr application, you’ll need a page that displays a list of the most recent spittles that have been submitted. Therefore, you’ll need a new method to serve such a page.
* First you need to define a repository for data access. For decoupling purposes, and so you don’t get bogged down in database specifics, you’ll define the repository as an interface now and create an implementation of it later.
* At the moment, you only need a repository that can fetch a list of the spittles. SpittleRepository, as defined here, is a sufficient start:

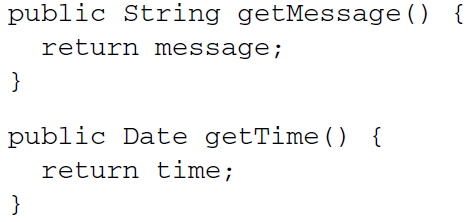


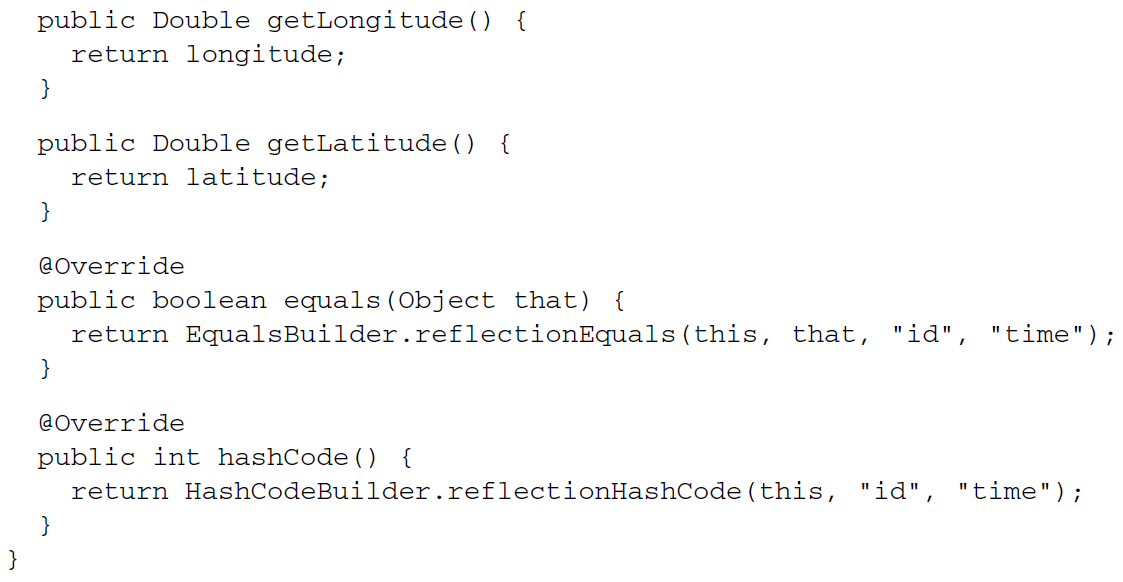
* The findSpittles() method takes two parameters. The max parameter is a Spittle ID that represents the maximum ID of any Spittle that should be returned. As for the count parameter, it indicates how many Spittle objects to return. In order to get the 20 most recent Spittle objects, you can call findSpittles() like this:



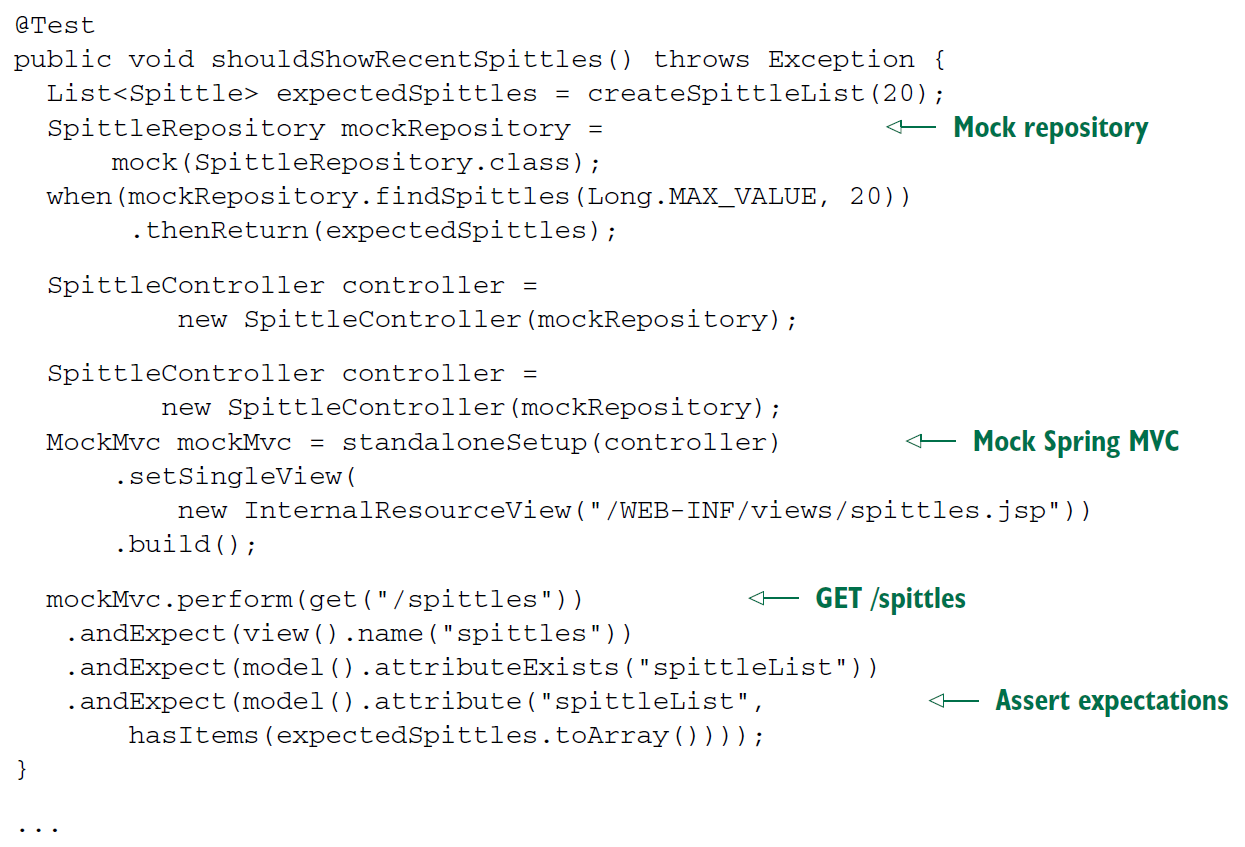
* You’ll keep the Spittle class fairly simple for now, as shown next. It will have properties to carry a message, a timestamp, and the latitude/longitude of the location from which the spittle was posted.

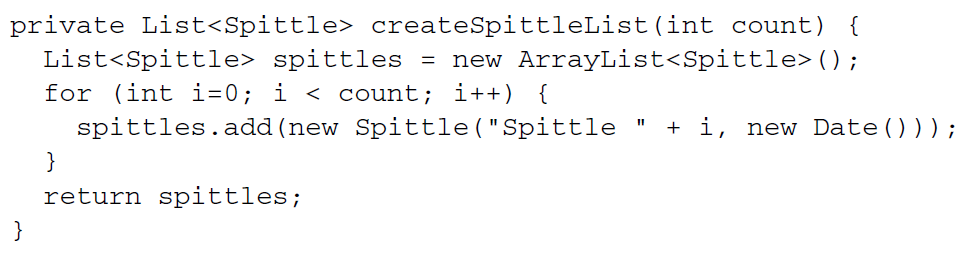




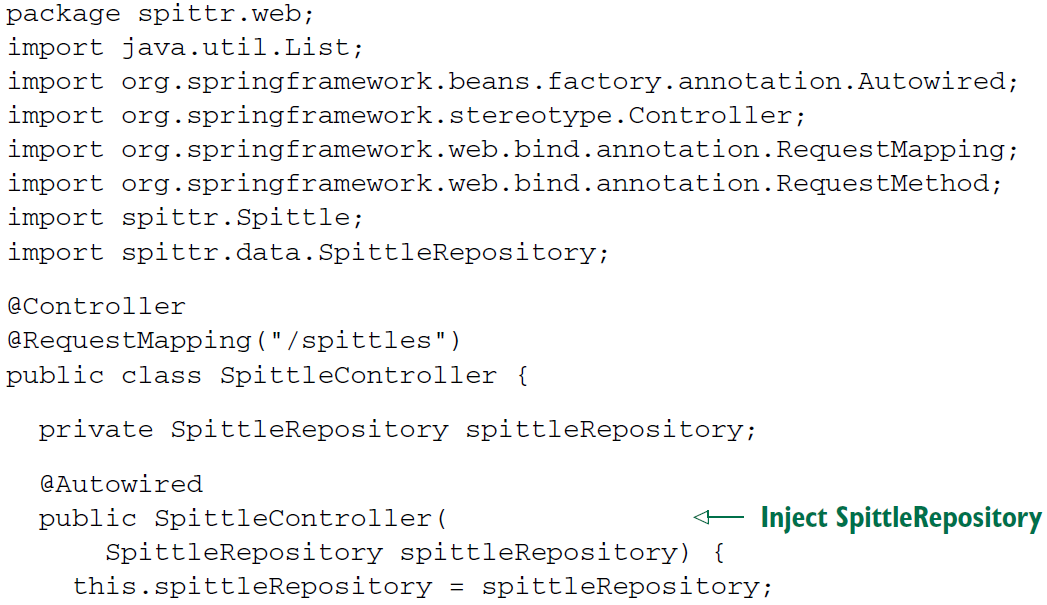


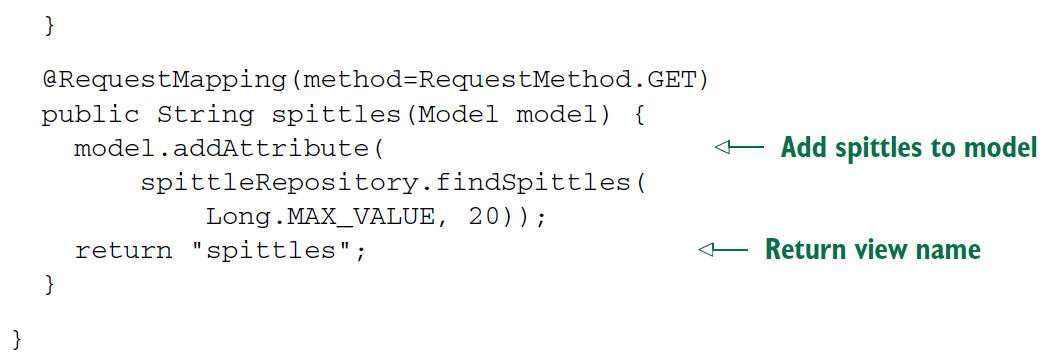
* For the most part, Spittle is a basic POJO data object—nothing complicated. The only thing to note is that you’re using Apache Commons Lang for easy implementation of the equals() and hashCode() methods. Aside from the general utility value of those methods, they’ll be valuable in writing a test for the controller handler method.



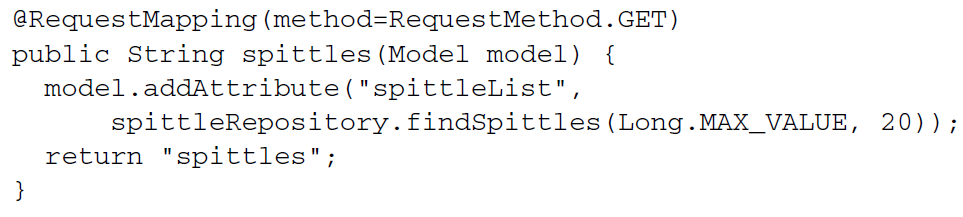


* This test starts by creating a mock implementation of the SpittleRepository interface that will return a list of 20 Spittle objects from its findSpittles() method. It then injects that repository into a new SpittleController instance and sets up MockMvc to use that controller.
* Notice that unlike HomeControllerTest, this test calls setSingleView() on the MockMvc builder. This is so the mock framework won’t try to resolve the view name coming from the controller on its own. In many cases, this is unnecessary. But for this controller method, the view name will be similar to the request’s path; left to its default view resolution, MockMvc will fail because the view path will be confused with the controller’s path. The actual path given when constructing the InternalResourceView is unimportant in this test, but you set it to be consistent with how you’ve configured InternalResourceViewResolver.
* The test wraps up by performing a GET request for /spittles and asserting that the view name is spittles and that the model has an attribute named spittleList with the expected contents.
* Of course, if you ran the test at this point, it would fail. It wouldn’t just fail to run; it would fail to compile. That’s because you haven’t yet written the SpittleController. Let’s create a SpittleController so that it satisfies the expectations of the test. Here’s an implementation of SpittleController that should satisfy the test:

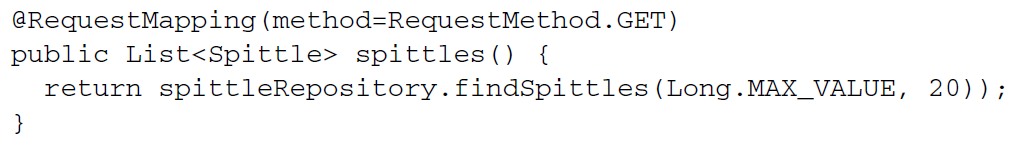




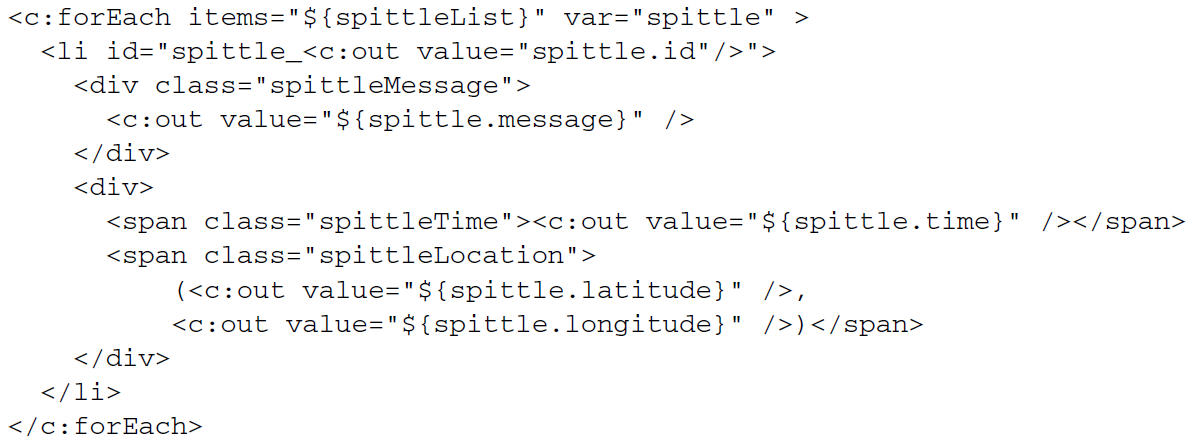
* As you can see, SpittleController has a constructor that’s annotated with @Autowired to be given a SpittleRepository. That SpittleRepository is then used in the spittles() method to fetch a list of recent spittles.
* Notice that the spittles() method is given a Model as a parameter. This is so that spittles() can populate the model with the Spittle list it retrieves from the repository. The Model is essentially a map (that is, a collection of key-value pairs) that will be handed off to the view so that the data can be rendered to the client.
* When addAttribute() is called without specifying a key, the key is inferred from the type of object being set as the value. In this case, because it’s a List<Spittle>, the key will be inferred as spittleList.
* The last thing spittles() does is return spittles as the name of the view that will render the model.
* If you’d prefer to be explicit about the model key, you’re welcome to specify it. For example, the following version of spittles() is equivalent to the one.



* And while we’re on the subject of alternate implementations, here’s another way to write the spittles() method:



* This version is quite a bit different from the others. Rather than return a logical view name and explicitly setting the model, this method returns the Spittle list. When a handler method returns an object or a collection like this, the value returned is put into the model, and the model key is inferred from its type (spittleList, as in the other examples).
* As for the logical view name, it’s inferred from the request path. Because this method handles GET requests for /spittles, the view name is spittles (chopping off the leading slash).
* No matter which way you choose to write the spittles() method, the result is the same. A list of Spittle objects is stored in the model with a key of spittleList and given to the view whose name is spittles.
* Given the way you’ve configured InternalResourceViewResolver, that view is a JSP at /WEB-INF/views/spittles.jsp.
* Now that there’s data in the model, how does the JSP access it? As it turns out, when the view is a JSP, the model data is copied into the request as request attributes. Therefore, the spittles.jsp file can use JavaServer Pages Standard Tag Library’s (JSTL) <c:forEach> tag to render the list of spittles:



* Although SpittleController is simple, it’s still a step up from what you wrote in HomeController. One thing that neither HomeController nor SpittleController does, however, is handle any form of input. Let’s expand on SpittleController to take some input from the client.