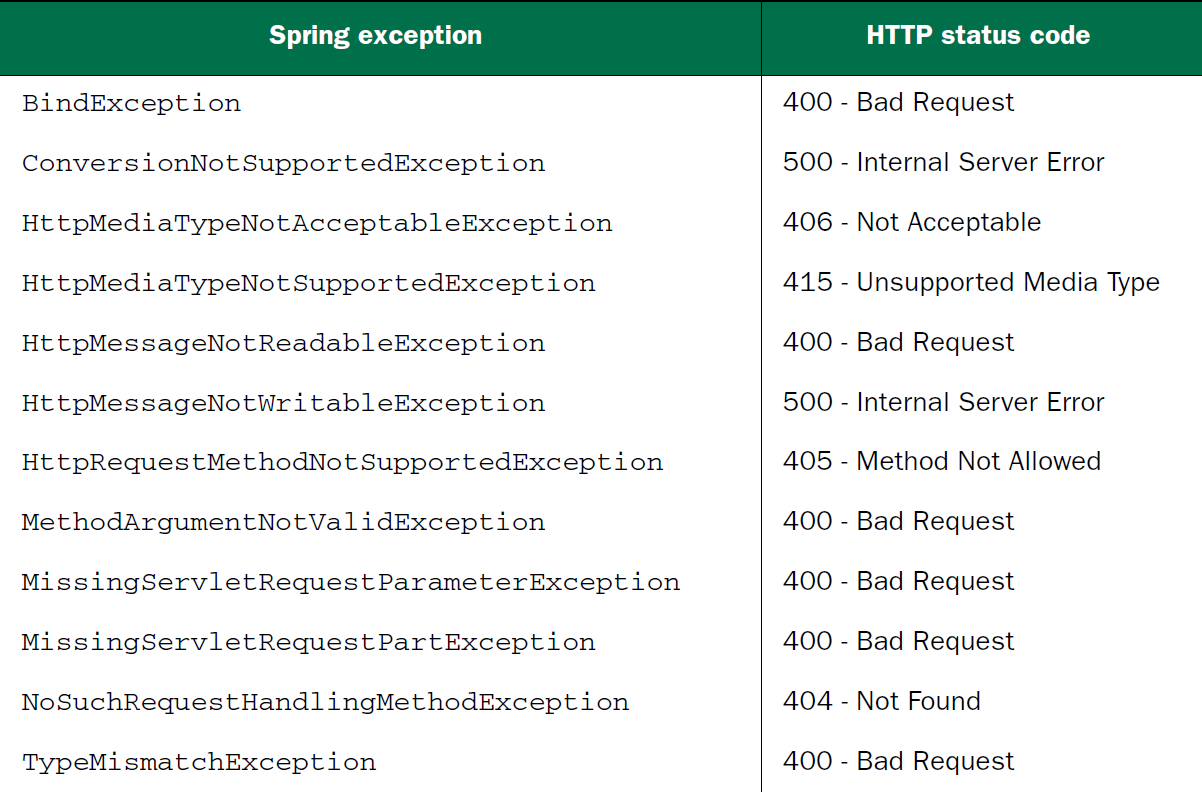
***Handling exceptions***

Up to this point, we’ve been assuming that everything will always work in the Spittr application. But what if something goes wrong? What if, while handling a request, an exception is thrown? What response will be sent to the client when thing go awry?

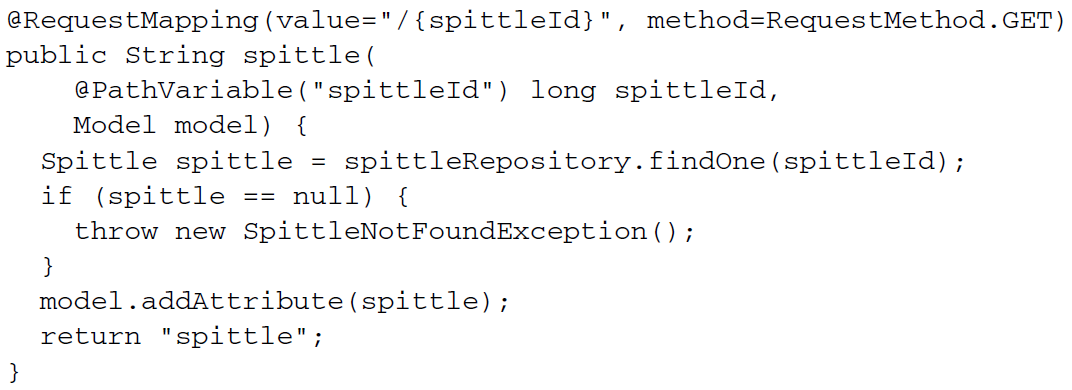
* No matter what happens, good or bad, the outcome of a servlet request is a servlet response. If an exception occurs during request processing, the outcome is still a servlet response. Somehow, the exception must be translated into a response.
* Spring offers a handful of ways to translate exceptions to responses:
* Certain Spring exceptions are automatically mapped to specific HTTP status codes.
* An exception can be annotated with @ResponseStatus to map it to an HTTP status code.
* A method can be annotated with @ExceptionHandler to handle the exception.
* The simplest way to handle an exception is to map it to the HTTP status code to be placed on the response. Let’s see how to map exceptions to HTTP status codes.

***Mapping exceptions to HTTP status codes***

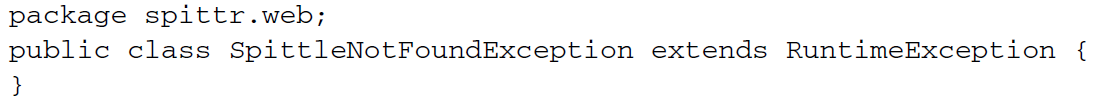
Out of the box, Spring automatically maps a dozen of its own exceptions to appropriate status codes.



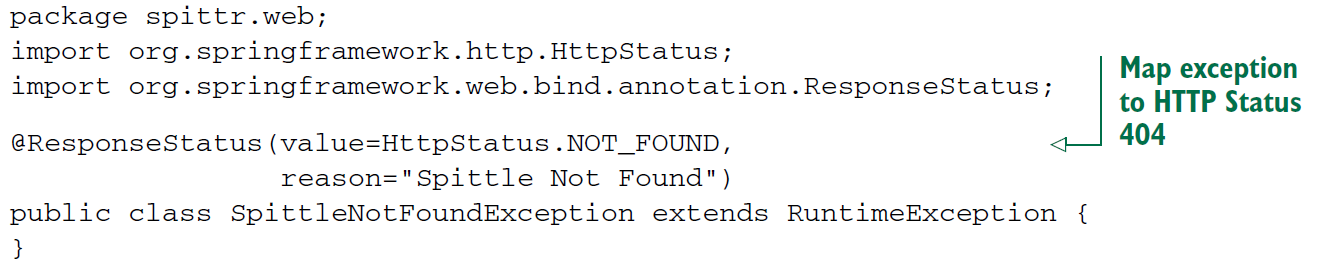
* The exceptions in above table are usually thrown by Spring itself as the result of something going wrong in DispatcherServlet or while performing validation. For example, if DispatcherServlet can’t find a controller method suitable to handle a request, a NoSuchRequestHandlingMethodException will be thrown, resulting in a response with a status code of 404 (Not Found).
* Although these built-in mappings are helpful, they do no good for any application exceptions that may be thrown. Fortunately, Spring offers a way to map exceptions to HTTP status codes via the @ResponseStatus annotation.
* To demonstrate, consider the following request-handling method from SpittleController that could result in an HTTP 404 status (but doesn’t):



* Here, a Spittle is retrieved by its ID from the SpittleRepository. If findOne()returns a Spittle object, that Spittle is put into the model, and the view whose name is spittle is tasked with rendering it in the response. But if findOne() returns null, then a SpittleNotFoundException is thrown. For now, SpittleNotFoundException is a simple unchecked exception that looks like this:



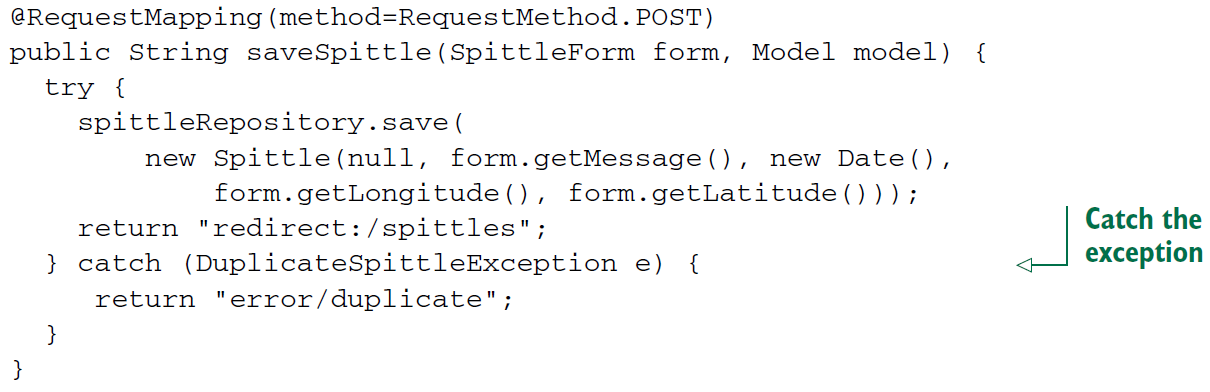
* If the spittle() method is called on to handle a request, and the given ID comes up empty, the SpittleNotFoundException will (by default) result in a response with a 500 (Internal Server Error) status code. In fact, in the event of any exception that isn’t otherwise mapped, the response will always have a 500 status code. But you can change that by mapping SpittleNotFoundException otherwise.
* When SpittleNotFoundException is thrown, it’s a situation where a requested resource isn’t found. The HTTP status code of 404 is precisely the appropriate response status code when a resource isn’t found. So, let’s use @ResponseStatus to map SpittleNotFoundException to HTTP status code 404.



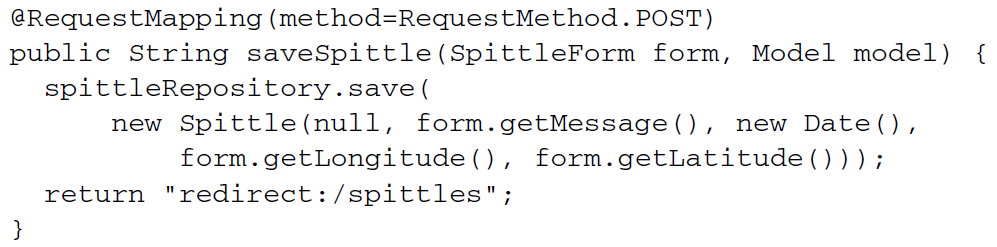
* After introducing this @ResponseStatus annotation, if a SpittleNotFoundException were to be thrown from a controller method, the response would have a status code of 404 and a reason of Spittle Not Found.

***Writing exception-handling methods***

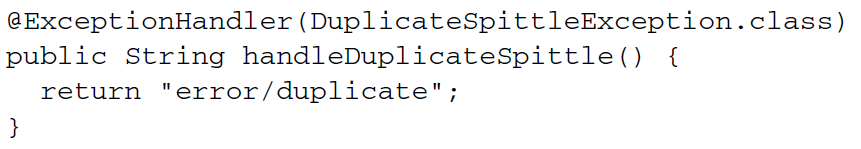
* Mapping exceptions to status codes is simple and sufficient for many cases. But what if you want the response to carry more than just a status code that represents the error that occurred? Rather than treat the exception generically as some HTTP error, maybe you’d like to handle the exception the same way you might handle the request itself.
* As an example, suppose that SpittleRepository’s save() method throws a DuplicateSpittleException if a user attempts to create a Spittle with text identical to one they’ve already created. That means the saveSpittle() method of SpittleController might need to deal with that exception. As shown in the following listing, saveSpittle() could directly handle the exception.



* It’s a basic example of Java exception handling. Nothing more. It works fine, but the method is a bit complex. Two paths can be taken, each with a different outcome. It’d be simpler if saveSpittle() could focus on the happy path and let some other method deal with the exception.
* First, let’s rip the exception-handling code out of saveSpittle():



* As you can see, saveSpittle() is now much simpler. Because it’s written to only be concerned with the successful saving of a Spittle, it has only one path and is easy to follow (and test). Now let’s add a new method to SpittleController that will handle the case where DuplicateSpittleException is thrown:



* The @ExceptionHandler annotation has been applied to the handleDuplicateSpittle() method, designating it as the go-to method when a DuplicateSpittleException is thrown. It returns a String, which, just as with the request-handling method, specifies the logical name of the view to render, telling the user that they attempted to create a duplicate entry.
* What’s especially interesting about @ExceptionHandler methods is that they handle their exceptions from *any* handler method in the same controller. So although you created the handleDuplicateSpittle() method from the code extracted from saveSpittle(), it will handle a DuplicateSpittleException thrown from any method in SpittleController. Rather than duplicate exception-handling code in every method that has the potential for throwing a DuplicateSpittleException, this one method covers them all.
* If @ExceptionHandler methods can handle exceptions thrown from any handler method in the same controller class, you might be wondering if there’s a way they can handle exceptions thrown from handler methods in *any* controller. As of Spring 3.2 they certainly can, but only if they’re defined in a controller advice class.

***Advising controllers***

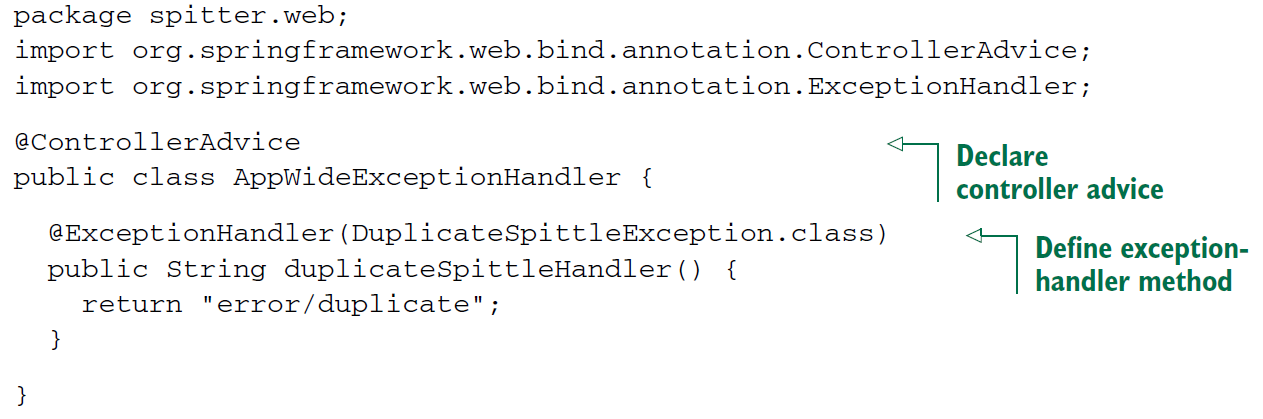
* Certain aspects of controller classes might be handier if they could be applied broadly across all controllers in a given application. @ExceptionHandler methods, for instance, could prove useful in handling exceptions across multiple controllers.
* If a particular exception is thrown from multiple controller classes, you might find yourself duplicating the same @ExceptionHandler method in all of those controllers. Or, to avoid the duplication, you might create a base controller class that all of your controllers could extend to inherit the common @ExceptionHandler method.
* Spring 3.2 brings another option to the table: controller advice. A *controller advice* is any class that’s annotated with @ControllerAdvice and has one or more of the following kinds of methods:

- @ExceptionHandler-annotated

- @InitBinder-annotated

- @ModelAttribute-annotated

* Those methods in an @ControllerAdvice-annotated class are applied globally across all @RequestMapping-annotated methods on all controllers in an application.
* The @ControllerAdvice annotation is itself annotated with @Component. Therefore, an @ControllerAdvice-annotated class will be picked up by component-scanning, just like an @Controller-annotated class.
* One of the most practical uses for @ControllerAdvice is to gather all @ExceptionHandler methods in a single class so that exceptions from all controllers are handled consistently in one place. For example, suppose you want to apply the DuplicateSpittleException-handling method across all controllers in your application. The next listing shows AppWideExceptionHandler, a @ControllerAdvice-annotated class that does just that.



* Now, if a DuplicateSpittleException is thrown from *any* controller method, no matter which controller it’s in, this duplicateSpittleHandler() method will be called to handle the exception. The ExceptionHandler-annotated method can be written much like an @RequestMapping-annotated method. As shown, it returns error/duplicate as the logical view name so that a friendly error page is displayed to the user.