covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

11 Giving Spring some REST

This chapter covers

- Writing controllers that serve REST resources
- Representing resources in XML, JSON, and other formats
- Writing REST clients
- Submitting RESTful forms

Data is king.

As developers, we're often focused on building great software to solve business problems. Data is just the raw materials that our software processes need to get their job done. But if you were to ask most business people which is most valuable to them, data or software, they're likely to choose data. Data is the life's blood of many businesses. Software is often replaceable. But the data gathered over the years can never be replaced.[1]

[1] That's not to say that software has no value. Most businesses would be severely handicapped without their software. But they'd be dead without their data.

Don't you think it's odd that, given the importance of data, the way we develop software often treats data as an afterthought? Take the remote services from the previous chapter as an example. Those services were centered on actions and processes, not information and resources.

In recent years, *Representational State Transfer (REST)* has emerged as a popular information-centric alternative to traditional SOAP-based web services. To help Spring developers take advantage of the REST architectural model, Spring 3.0 came packed with first-class support for working with REST.

The good news is that Spring's REST support builds upon Spring MVC, so we've already covered much of what we'll need for working with REST in Spring. In this chapter, we'll build upon what we already know about Spring MVC to develop controllers that handle requests for RESTful resources. We'll also see what Spring has to offer on the client side of a REST conversation.

But before we get too carried away, let's examine what working with REST is all about.

11.1 Getting REST

I'll wager that this isn't the first time you've heard or read about REST. There's been a lot of talk about REST in recent years and you'll find that it's fashionable in software development to speak ill of SOAP-based web services while promoting REST as an alternative.

Certainly, SOAP can be overkill for many applications and REST brings a simpler alternative. The problem is that not everybody has a solid grasp of what REST really is. As a result, a lot of misinformation is floating about. Before we can talk about how Spring supports REST, we need to establish a common understanding of what REST is all about.

11.1.1 The fundamentals of REST

A mistake that's often made when approaching REST is to think of it as "web services with URLs"—to think of REST as another remote procedure call (RPC) mechanism, like SOAP, but invoked through plain HTTP URLs and without SOAP's hefty XML namespaces.

On the contrary, REST has little to do with RPC. Whereas RPC is service-oriented and focused on actions and verbs, REST is resource-oriented, emphasizing the things and nouns that describe an application.

Also, although URLs play a key role in REST, they're only a part of the story.

To understand what REST is all about, it helps to break down the acronym into its constituent parts:

- *Representational*—REST resources can be represented in virtually any form, including XML, JavaScript Object Notation (JSON), or even HTML—whatever form best suits the consumer of those resources.

- *State*—When working with REST, we're more concerned with the state of a resource than with the actions we can take against resources.
- *Transfer*—REST involves transferring resource data, in some representational form, from one application to another.

Put more succinctly, REST is about transferring the state of resources—in whatever form is most appropriate—from a server to a client (or vice versa).

Given this view of REST, I try to avoid terms such as *REST service*, or *RESTful web service*, or any similar term that incorrectly gives prominence to actions. Instead, I prefer to emphasize the resource-oriented nature of REST and speak of *RESTful resources*.

11.1.2 How Spring supports REST

Spring has long had some of the ingredients needed for exposing REST resources. But with Spring 3 came several enhancements to Spring MVC providing first-class REST support. Now Spring supports development of REST resources in the following ways:

- Controllers can handle requests for all HTTP methods, including the four primary REST methods: `GET`, `PUT`, `DELETE`, and `POST`.
- The new `@PathVariable` annotation enables controllers to handle requests for parameterized URLs (URLs that have variable input as part of their path).
- The `<form:form>` JSP tag from Spring's form-binding JSP tag library, along with the new `HiddenHttpMethodFilter`, make it possible to submit `PUT` and `DELETE` requests from HTML forms, even in browsers that don't support those HTTP methods.
- Resources can be represented in a variety of ways using Spring's view and view resolvers, including new view implementations for rendering model data as XML, JSON, Atom, and RSS.
- The representation best suited for the client can be chosen using the new `ContentNegotiatingViewResolver`.
- View-based rendering can be bypassed altogether using the new `@ResponseBody` annotation and various `HttpMethodConverter` implementations.
- Similarly, the new `@RequestBody` annotation, along with `HttpMethodConverter` implementations, can convert inbound HTTP data into Java objects passed into a controller's handler methods.
- `RestTemplate` simplifies client-side consumption of REST resources.

Throughout this chapter we're going to explore all of these features that make Spring more RESTful and see how to both produce and consume REST resources. We'll start by looking at what goes into a resource-oriented Spring MVC controller.

11.2 Writing resource-oriented controllers

As we saw in chapter 7, Spring MVC's model for writing controller classes is extremely flexible. Almost any method with almost any signature can be annotated to handle a web request. But a side effect of such flexibility is that Spring MVC allows you to

develop controllers that aren't ideal in terms of RESTful resources. It's too easy to write *RESTless* controllers.

11.2.1 Dissecting a RESTless controller

To help understand what a RESTful controller looks like, it helps to first know what a RESTless controller looks like. `DisplaySpittleController` is an example of a RESTless controller.

Listing 11.1 `DisplaySpittleController` is a RESTless Spring MVC controller.

```
package com.habuma.spitter.mvc.restless;

import javax.inject.Inject;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

import com.habuma.spitter.service.SpitterService;

@Controller
@RequestMapping("/displaySpittle.htm")
public class DisplaySpittleController {
  private final SpitterService spitterService;

  @Inject
  public DisplaySpittleController(SpitterService spitterService) {
    this.spitterService = spitterService;
  }

  @RequestMapping(method=RequestMethod.GET)
  public String showSpittle(@RequestParam("id") long id, Model model) {
    model.addAttribute(spitterService.getSpittleById(id));
    return "spittles/view";
  }
}
```

RESTless URL mapping

The first thing to notice about the controller in listing 11.1 is its name. Sure, it's just a name. But it accurately describes what the controller does. The first word is *Display*—a verb. This is indicative of the fact that this controller is action-oriented, not resource-oriented.

Take note of the `@RequestMapping` annotation at the class level. It says that this controller will handle requests for /displaySpittle.htm. That seems to imply that this controller is focused on the specific use case of displaying spittles (which is corroborated by the name of the class). What's more, the extension implies that it's only capable of displaying that list in HTML form.

Nothing is terribly wrong with how `DisplaySpittleController` is written. But it isn't a RESTful controller. It's action-oriented and focused on a specific use case: displaying a `Spittle` object's details in HTML form. Even the controller's class name agrees.

Now that you know what a RESTless controller looks like, let's see what goes into writing a RESTful controller. We'll start by looking at how to handle requests for resource-oriented URLs.

11.2.2 Handling RESTful URLs

URLs are one of the first things that most people think about when starting to work with REST. After all, everything that's done in REST is done through a URL. The funny thing about many URLs is that they usually don't do what a URL is supposed to do.

URL is an acronym that stands for *uniform resource locator*. Given that name, it seems that a URL is intended to locate a resource. What's more, all URLs are also URIs, or *uniform resource identifiers*. If that's true, then we should expect that any given URL would not only locate a resource, but should also serve to identify a resource.

The fact that a URL locates a resource should seem natural. After all, for years we've been typing URLs into our web browser's address field to find content on the internet. But it's not a stretch to think of that URL as a means of uniquely identifying a resource. No two resources could share the same URL, so the URL could also be a means of identifying a resource.[2]

Many URLs don't locate or identify anything—they make demands. Rather than identify a thing, they insist that some action be taken. For instance, figure 11.1 illustrates the kind of URL handled by the `DisplaySpittleController`'s `displaySpittle()` method.

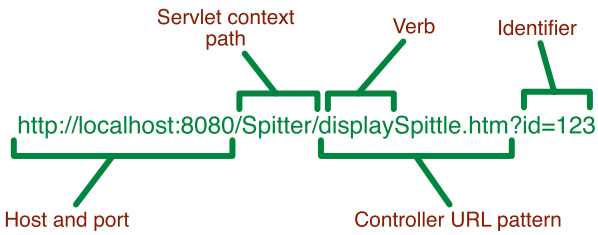

Figure 11.1 A RESTless URL is action-oriented and doesn't identify or locate a resource.

As you can see, this URL doesn't locate or identify a resource. It demands that the server display a `Spittle`. The only part of the URL that identifies anything is the `id` query parameter. The base portion of the URL is verb-oriented. That is to say that it's a RESTless URL.

If we're going to write controllers that properly handle RESTful URLs, we should first get to know what a RESTful URL looks like.

CHARACTERISTICS OF A RESTFUL URL

In contrast to their RESTless cousins, RESTful URLs fully acknowledge that HTTP is all about resources. For example, figure 11.2 shows how we might restructure the RESTless URL to be more resource-oriented.

[2] Although outside of the scope of this book, the semantic web takes advantage of the identifying nature of URLs in creating a linked web of resources.

One thing that's not clear about this URL is what it does. That's because the URL doesn't *do* anything. Rather, it identifies a resource. Specifically, it locates the resource that represents a `Spittle` object. What will be done with that resource is a separate matter—one for HTTP methods to decide (which we'll look at in section 11.2.3).

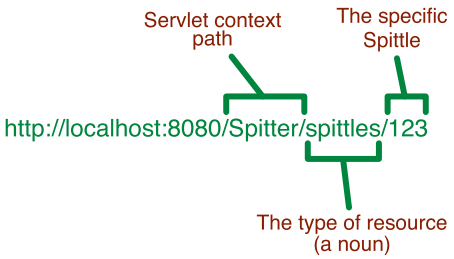

Figure 11.2 A RESTful URL is resource-oriented, both identifying and locating a resource.

This URL not only locates a resource, but it also uniquely identifies that resource—it serves equally well as a URI as it does as a URL. Instead of using a query parameter to identify the resource, the entire base URL identifies the resource.

In fact, the new URL has no query parameters at all. Although query parameters are still a legitimate way to send information to the server, they're intended to provide guidance to the server in producing the resource. They shouldn't be used to help identify a resource.

One final observation should be made about RESTful URLs: they tend to be hierarchical. As you read them from left to right, you move from a broad concept to something more precise. In our example, the URL has several levels, any of which could identify a resource:

- *http://localhost:8080* identifies a domain and port. Although our application won't associate a resource with this URL, there's no reason why it couldn't.
- *http://localhost:8080/Spitter* identifies the application's servlet context. This URL is more specific in that it has identified an application running on the server.
- *http://localhost:8080/Spitter/spittles* identifies a resource that represents a list of `Spittle` objects within the Spitter application.
- *http://localhost:8080/Spitter/spittles/123* is the most precise URL, identifying a specific `Spittle` resource.

What makes the RESTful URL interesting is that its path is parameterized. Whereas the RESTless URL took its input from query parameters, the RESTful URL's input is part of the URL's path. To be able to handle requests for that kind of URL, we'll need a way to write a controller's handler method so that it can take input from the URL's path.

EMBEDDING PARAMETERS IN URLS

To enable parameterized URL paths, Spring 3 introduced a new `@PathVariable` annotation. To see how this works, look at `SpittleController`, a new Spring MVC controller that takes a resource-oriented approach to handling requests for `Spittles`.

Listing 11.2 `SpittleController` is a RESTful Spring MVC controller.

```
package com.habuma.spitter.mvc;
import javax.inject.Inject;
import javax.validation.Valid;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

@Controller
@RequestMapping("/spittles")
public class SpittleController {
  private SpitterService spitterService;

  @Inject
  public SpittleController(SpitterService spitterService) {
    this.spitterService = spitterService;
  }

  @RequestMapping(value="/{id}",
                  method=RequestMethod.GET)
  public String getSpittle(@PathVariable("id") long id,
          Model model) {
    model.addAttribute(spitterService.getSpittleById(id));
    return "spittles/view";
  }
}
```

Handle requests for /spittles

Use placeholder variable in path

We've annotated `SpittleController` with `@RequestMapping` at the class level to indicate that this controller will handle requests for `Spittle` resources—requests whose URLs start with */spittles*.

For now, there's only one handler method—the `getSpittle()` method. When this method's `@RequestMapping` annotation is coupled with the class-level `@RequestMapping`, this method is set to handle `GET` requests for URLs that take the form */spittles/{id}*.

You're probably wondering about those weird curly-braces in the URL pattern. The part that says *{id}* is a placeholder through which variable data will be pass into the method. It corresponds to the `@PathVariable` annotation on the `id` method parameter.

So, if a `GET` request comes in for http://localhost:8080/Spitter/spittles/123, then the `getSpittle()` method will be called with `123` passed in for the `id` parameter. The method then uses that value to look up the requested `Spittle` object and place it into the model.

At this point you may have noticed that the phrase `id` is used three times in this method's signature. Not only is it used as the URL path placeholder and as the value of the `@PathVariable` annotation, it's also used as the actual name of the method parameter. That's only a coincidence in this case. But if the method parameter name happens to be the same as the path variable name (and I can think of no reason why it shouldn't be), then you can take advantage of a simple convention and leave out `@PathVariable`'s value. For example:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public String getSpittle(@PathVariable long id, Model model) {
```

```
  model.addAttribute(spitterService.getSpittleById(id));
  return "spittles/view";
}
```

With no value given to `@PathVariable`, the method parameter name serves as the name of the path variable.[3]

Regardless of whether you explicitly identify the path variable by name, `@PathVariable` makes it possible to write controller handler methods that handle requests for URLs that identify a resource instead of describing some action to be taken. The other side of RESTful requests are the HTTP methods that will be applied to the URLs. Let's see how HTTP methods provide the verbs in a REST request.

11.2.3 Performing the REST verbs

As I mentioned before, REST is about the transfer of resource state. Therefore, we really only need a handful of verbs to be able to act upon those resources—verbs to transfer the state of a resource. For any given resource, the most common operations will be to create a resource on the server, retrieve it from the server, update it on the server, or delete it from the server.

The verbs we're interested in (*post*, *get*, *put*, and *delete*) correspond directly to four of the methods as defined by the HTTP specification and as summarized in table 11.1.[4]

Each of the HTTP methods is characterized by two traits: safety and idempotency. A method is considered *safe* if it doesn't change the state of the resource. *Idempotent* methods may or may not change state, but repeated requests should have no further side effects after the first request. By definition, all safe methods are also idempotent, but not all idempotent methods are safe.

Table 11.1 HTTP offers several methods for manipulating resources.

Method	Description	Safe?	Idempotent?
`GET`	Retrieves resource data from the server. The resource is identified by the request's URL.	Yes	Yes
`POST`	Posts data to the server to be handled by a processor listening at the request's URL.	No	No
`PUT`	Puts resource data to the server, at the URL of the request.	No	Yes
`DELETE`	Deletes the resource on the server identified by the request's URL.	No	Yes
`OPTIONS`	Requests available options for communication with the server.	Yes	Yes

[3] This assumes that you've compiled your controller classes with debugging information compiled into the class files. Otherwise, the method parameter names won't be available at runtime to match against path variable names.

[4] The HTTP specification defines four other methods: TRACE, OPTIONS, HEAD, and CONNECT. But we'll focus on the four core methods.

Table 11.1 HTTP offers several methods for manipulating resources. *(continued)*

Method	Description	Safe?	Idempotent?
HEAD	Like GET, except that only the headers should be returned—no content should be returned in the response body.	Yes	Yes
TRACE	Echoes the request body back to the client.	Yes	Yes

It's important to realize that although Spring supports all of HTTP's methods, it's still up to you, the developer, to be sure that the implementation of those methods follows the semantics of those methods. In other words, a GET-handling method should only return a resource—it shouldn't update or delete a resource.

The four HTTP methods described in table 11.1 are often mapped to CRUD (create/read/update/delete) operations. Certainly the GET method performs a read operation and the DELETE method performs a delete operation. And, even though PUT and POST can be used in ways other than update and create operations, that's commonly how they're used.

We've already seen an example of how to handle GET requests. The SpittleController's getSpittle() method is annotated with @RequestMapping, with the method attribute set to handle GET requests. The method attribute is the key to detailing the HTTP method that will be handled by a controller method.

UPDATING RESOURCES WITH PUT

When it comes to understanding the PUT method's purpose, it helps to know that it's the semantic opposite of GET. Whereas a GET request transfers the state of a resource from the server to the client, PUT transfers the resource state from the client to the server.

For example, the following putSpittle() method is annotated to receive a Spittle object from a PUT request:

```
@RequestMapping(value="/{id}", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void putSpittle(@PathVariable("id") long id,
        @Valid Spittle spittle) {
  spitterService.saveSpittle(spittle);
}
```

The putSpittle() method is annotated with @RequestMapping, like any other handler method. In fact, the @RequestMapping annotation here is almost the same as the one used with the getSpittle() method. The only difference is that the method attribute is set to handle HTTP PUT requests instead of GET requests.

If that's the only difference, then that must mean that the putSpittle() method will handle requests with URLs that take the form */spittles/{id}*—the same URLs that are handled by the getSpittle() method. Again, the URL identifies a resource, not what'll be done with it. So the URL that identifies a Spittle will be the same whether we're GETting it or PUTting it.

The `putSpittle()` method is also tagged with an annotation we haven't seen before. The `@ResponseStatus` annotation defines the HTTP status that should be set on the response to the client. In this case, `HttpStatus.NO_CONTENT` indicates that the response status should be set to the HTTP status code 204. That status code means that the request was processed successfully, but nothing is returned in the body of the response.

HANDLING DELETE REQUESTS

Rather than simply update a resource, we may want to get rid of it altogether. In the case of the Spitter application, for example, we may want to enable clients to delete an embarrassing `Spittle` that was written in haste or while the user was impaired. When you don't want a resource around anymore, that's what the HTTP `DELETE` method is for.

As a sample of handling a `DELETE` request in Spring MVC, let's add a new handler method to `SpittleController` that answers `DELETE` requests to remove a `Spittle` resource:

```
@RequestMapping(value="/{id}", method=RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteSpittle(@PathVariable("id") long id) {
  spitterService.deleteSpittle(id);
}
```

Once again, the `@RequestMapping` annotation looks a lot like the ones we used on `getSpittle()` and `putSpittle()`. It only varies in that this method's `@RequestMapping` has its `method` attribute set to handle `DELETE` requests. The URL pattern that identifies a `Spittle` resource remains the same.

Just like `putSpittle()`, `deleteSpittle()` is also annotated with `@ResponseStatus` to let the client know that the request was processed successfully, but that no content will be returned in the response.

CREATING RESOURCES WITH POST

There's one in every bunch: a free spirit… a dissident… a rebel. Among the HTTP methods, `POST` is that rebel. It doesn't obey the rules. It's unsafe and it is certainly *not* idempotent. This nonconformist HTTP method seems to break all of the rules, but in doing so it can handle the jobs that other HTTP methods can't.

To see this rebel in action, watch `POST` as it performs a job that it is often called on to do—creating new resources. The `createSpittle()` method is a `POST`-handling controller method that creates new `Spittle` resources.

Listing 11.3 Creating new `Spittles` with `POST`

```
@RequestMapping(method=RequestMethod.POST)          <-- Handle POST

@ResponseStatus(HttpStatus.CREATED)                 <-- Response with HTTP 201
public @ResponseBody Spittle createSpittle(@Valid Spittle spittle,
                    BindingResult result, HttpServletResponse response)
                    throws BindException {
  if(result.hasErrors()) {
```

```
    throw new BindException(result);
  }

  spitterService.saveSpittle(spittle);

  response.setHeader("Location", "/spittles/" + spittle.getId());
  return spittle;
}
```

Set resource location

Return Spittle resource

The first thing you may notice is that this method's `@RequestMapping` is different from the ones we've seen so far. Unlike the others, this one doesn't have its `value` attribute set. That means that the controller's class-level `@RequestMapping` is solely responsible for determining the URL pattern handled by `createSpittle()`. More specifically, `createSpittle()` will handle requests whose URL pattern matches */spittles*.

Typically, the server determines a resource's identity. Since we're creating a new resource here, there's no way that we could know the URL for that resource. So, whereas `GET`, `PUT`, and `DELETE` requests operate directly on the resource identified by their URL, `POST` has to operate against a URL that isn't the same as the resource it's creating (because that URL won't exist until the resource is created).

Once again, this method is annotated with `@ResponseStatus` to set the HTTP status code in the request. This time, the status will be set to 201 (Created) to indicate that a resource was successfully created. When an HTTP 201 response is returned to the client, the URL of the new resource should be sent along with it. So one of the last things that `createSpittle()` does is set the `Location` header to contain the resource's URL.

Although it's not strictly required with an HTTP 201 response, it's possible to return the full entity representation in the body of the response. So, much like the `GET`-handling `getSpittle()` method from before, this method concludes by returning the new `Spittle` object. This object will be transformed into some representation that the client can use.

What's not clear yet is how that transformation will take place. Or what the representation will look like. Let's have a look at the *R* in the REST acronym: representation.

11.3 Representing resources

Representation is an important facet of REST. It's how a client and a server communicate about a resource. Any given resource could be represented in virtually any form. If the consumer of the resource prefers JSON, then the resource could be presented in JSON format. Or if the consumer has a fondness for angle brackets, then the same resource could be presented in XML. Meanwhile, a human user viewing the resource in a web browser will likely prefer seeing it in HTML (or possibly PDF, Excel, or some other human-readable form). The resource doesn't change—only how it's represented.

It's important to know that controllers usually don't concern themselves with how resources will be represented. Controllers will deal with resources in terms of the Java objects that define them. But it's not until after the controller has finished its works that the resource will be transformed into a form that best suits the client.

Spring provides two ways to transform a resource's Java representation into the representation that will be shipped to the client:

- Negotiated view-based rendering
- HTTP message converters

Since we discussed view resolvers in chapter 7 and are already familiar with view-based rendering (also from chapter 7), we'll start by looking at how to use content negotiation to select a view or view resolver that can render a resource into a form that's acceptable to the client.

11.3.1 Negotiating resource representation

As you'll recall from chapter 7, when a controller's handler method finishes, a logical view name is usually returned. Even if the method doesn't directly return a logical view name (if the method returns `void`, for example), then the logical view name is derived from the request's URL. `DispatcherServlet` then passes the view name to a view resolver, asking it to help determine which view should render the results of the request.

In a human-facing web application, the view chosen is almost always rendered as HTML. View resolution is a one-dimensional activity. If the view name matches a view, then that's the view we'll go with.

When it comes to resolving view names into views that can produce resource representations, there's an additional dimension to consider. Not only does the view need to match the view name, but also the view needs to be chosen to suit the client. If the client wants XML, then an HTML-rendering view won't do—even if the view name matches.

Spring's `ContentNegotiatingViewResolver` is a special view resolver that takes the content type that the client wants into consideration. Just like any other view resolver, it's configured as a `<bean>` in the Spring application context, as shown next.

Listing 11.4 `ContentNegotiatingViewResolver` chooses the best view.

```
<bean class="org.springframework.web.servlet.view.
    ➥ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="json" value="application/json" />
      <entry key="xml" value="text/xml" />
      <entry key="htm" value="text/html" />
    </map>
  </property>
  <property name="defaultContentType" value="text/html" />
</bean>
```

Understanding how `ContentNegotiatingViewResolver` works involves getting to know the content-negotiation two-step:

1. Determine the requested media type(s)
2. Find the best view for the requested media type(s)

Let's dig deeper into each of these steps to see what makes `ContentNegotiatingViewResolver` tick. We'll start by figuring out what kind of content the client wants.

DETERMINING THE REQUESTED MEDIA TYPES

The first step in the content-negotiation two-step is determining what kind of resource representation the client wants. On the surface, that seems like a simple job. Shouldn't the request's `Accept` header give a clear indication of what representation should be sent to the client?

Unfortunately, the `Accept` header can't always be deemed reliable. If the client in question is a web browser, there's no guarantee that what the client wants is what the browser sends in the `Accept` header. Web browsers typically only accept human-friendly content types (such as `text/html`) and there's no way (short of developer-oriented browser plug-ins) to specify a different content type.

`ContentNegotiatingViewResolver` will consider the `Accept` header and use whatever media types it asks for; but only after it first looks at the URL's file extension. If the URL has a file extension on the end, it'll match that extension against the entries in the `mediaTypes` property. `mediaTypes` is a `Map` whose keys are file extensions and whose values are media types. If a match is found, then the media type will be used. In this way, the file extension can override any media types in the `Accept` header.

If the file extension doesn't produce any media types to work with, then the `Accept` header in the request will be considered. But if the request header doesn't have an `Accept` header, then it'll fall back to the media type set in the `defaultContentType` property.

As an example of how this might play out, suppose that `ContentNegotiatingViewResolver` configured in listing 11.4 is asked to figure out what the desired media types are for a request whose extension is .json. In that case, the file extension matches the `json` entry in the `mediaTypes` property. Therefore, the chosen media type will be `application/json`.

But suppose that a request comes along whose extension is .huh. That extension doesn't match any of the entries in the `mediaTypes` property. In the absence of a matching extension in the `mediaTypes` property, `ContentNegotiatingViewResolver` will look to the request's `Accept` header for the media types. If the request came from Firefox, then the media types are `text/html`, `application/xhtml+xml`, `application/xml`, and `*/*`. If the request doesn't have an `Accept` header, then `text/html` will be chosen from `defaultContentType`.

INFLUENCING HOW MEDIA TYPES ARE CHOSEN

The media type selection process, as described so far, outlines the default strategy for determining the requested media types. But there are several options that can influence that behavior:

- Setting the `favorPathExtension` property to `false` will cause `ContentNegotiatingViewResolver` to ignore the URL's path extension.
- Adding the Java Activation Framework (JAF) to the classpath will cause `ContentNegotiatingViewResolver` to ask JAF for help in determining the

media type for the path extension in addition to entries in the `mediaTypes` property.

- If you set the `favorParameter` property to `true` and if the request has a `format` parameter, then the value of the `format` parameter will be matched against the `mediaTypes` property. (Additionally, the name of the parameter can be chosen by setting the `parameterName` property.)
- Setting the `ignoreAcceptHeader` to `true` will remove the `Accept` from consideration.

For example, suppose that you set the `favorParameter` property to `true`:

```
<property name="favorParameter" value="true" />
```

Now a request whose URL doesn't have a file extension could still be matched up with the `application/json` media type as long as the request's `format` parameter is set to `json`.

Once `ContentNegotiatingViewResolver` knows what media types the client wants, it's time to find a view that can render that kind of content.

FINDING A VIEW

Unlike other view resolvers, `ContentNegotiatingViewResolver` doesn't directly resolve views. Instead, it delegates to other view resolvers to find a view that best suits the client. Unless otherwise specified, it'll use any view resolver in the application context. But you can explicitly list the view resolvers it should delegate to by setting the `viewResolvers` property.

`ContentNegotiatingViewResolver` asks all of its view resolvers to resolve the logical view name into a view. Every view that's resolved is added to a list of candidate views. In addition, if a view is specified in the `defaultView` property, it'll be added to the end of the candidate view list.

With the candidate view list assembled, `ContentNegotiatingViewResolver` cycles through all of the requested media types, trying to find a view from among the candidate views that produces a matching content type. The first match found is the one that's used.

In the end, if `ContentNegotiatingViewResolver` fails to find a suitable view, then it returns a `null` view. Or, if `useNotAcceptableStatusCode` is set to `true`, then a view with an HTTP status code of 406 (Not Acceptable) will be returned.

Content negotiation is a way of rendering resource representations to a client that fits right in with how we developed the web front end of our application in chapter 7. It's perfect for adding additional representations on top of the HTML representations that a Spring MVC web application already provides.

When defining machine-consumed RESTful resources, it may make more sense to develop the controller in a way that acknowledges that the data it produces will be represented as a resource consumed by another application. That's where Spring's HTTP message converters and the `@ResponseBody` annotation come into play.

11.3.2 Working with HTTP message converters

As we've seen in chapter 7 and in the previous section, a typical Spring MVC controller method ends by placing one or more pieces of information into the model and designating a view to render that data to the user. Although there are several ways of populating the model with data and many ways of identifying the view, every controller handler method we've seen up until now has followed that basic pattern.

But when a controller's job is to produce a representation of some resource, another more direct option is available that bypasses the model and view. In this style of handler method, the object returned from the controller is automatically converted into a representation appropriate for the client.

Employing this new technique starts with applying the `@ResponseBody` annotation to a controller's handler method.

RETURNING RESOURCE STATE IN THE RESPONSE BODY

Normally when a handler method returns a Java object (anything other than `String`), that object ends up in the model for rendering in the view. But if that handler method is annotated with `@ResponseBody`, then it indicates that the HTTP message converter mechanism should take over and transform the returned object into whatever form the client needs.

For example, consider the following `getSpitter()` method from `SpitterController`:

```
@RequestMapping(value = "/{username}", method = RequestMethod.GET,
                headers = {"Accept=text/xml, application/json"})
public @ResponseBody
Spitter getSpitter(@PathVariable String username) {
  return spitterService.getSpitter(username);
}
```

The `@ResponseBody` annotation tells Spring that we want to send the returned object as a resource to the client, converted into some representational form that the client can accept. More specifically, the resource should take a form that satisfies the request's `Accept` header. If the request has no `Accept` header, then it's assumed that the client can accept any representation form.

Speaking of the `Accept` header, take note of `getSpitter()`'s `@RequestMapping`. The `headers` attribute indicates that this method will only handle requests whose `Accept` header includes `text/xml` or `application/json`. Any other kind of request, even if it's a `GET` request whose URL matches the path specified, won't be handled by this method. It'll either be handled by some other handler method (if an appropriate one exists) or the client will be sent an HTTP 406 (Not Acceptable) response.

Taking an arbitrary Java object returned from a handler method and converting it into a client-pleasing representation is a job for one of Spring's HTTP message converters. Spring comes with a variety of message converters, as listed in table 11.2, to handle the most common object-to-representation conversion needs.

Table 11.2 Spring provides several HTTP message converters that marshal resource representations to and from various Java types.

Message converter	Description
`AtomFeedHttpMessageConverter`	Converts Rome[a] `Feed` objects to/from Atom feeds (media type `application/atom+xml`). *Registered if Rome library is present on the classpath.*
`BufferedImageHttpMessageConverter`	Converts `BufferedImages` to/from image binary data.
`ByteArrayHttpMessageConverter`	Reads/writes byte arrays. Reads from all media types (`*/*`) and writes as `application/octet-stream`. *Registered by default.*
`FormHttpMessageConverter`	Reads content as `application/x-www-form-urlencoded` into a `MultiValueMap<String, String>`. Also writes `MultiValueMap<String, String>` as `application/x-www-form-urlencoded` and `MultiValueMap<String, Object>` as `multipart/form-data`.
`Jaxb2RootElementHttpMessageConverter`	Reads and writes XML (`text/xml or application/xml`) from/to JAXB2-annotated objects. *Registered if JAXB v2 libraries are present on the classpath.*
`MappingJacksonHttpMessageConverter`	Reads and writes JSON from/to typed objects or untyped `HashMaps`. *Registered if Jackson JSON library is present on the classpath.*
`MarshallingHttpMessageConverter`	Reads and writes XML using an injected marshaller and unmarshaller. Supported (un)marshallers include Castor, JAXB2, JIBX, XMLBeans, and XStream.
`ResourceHttpMessageConverter`	Reads and writes `Resources`. *Registered by default.*
`RssChannelHttpMessageConverter`	Reads and writes RSS feeds from/to Rome `Channel` objects. *Registered if Rome library is present on the classpath.*
`SourceHttpMessageConverter`	Reads and writes XML from/to `javax.xml.transform.Source` objects. *Registered by default.*
`StringHttpMessageConverter`	Reads all media types (`*/*`) into a `String`. Writes Strings to `text/plain`. *Registered by default.*
`XmlAwareFormHttpMessageConverter`	An extension of `FormHttpMessageConverter` that adds support for XML-based parts using a `SourceHttpMessageConverter`. *Registered by default.*

a. https://rome.dev.java.net

For example, suppose the client has indicated via the request's `Accept` header that it can accept `application/json`. Assuming that the Jackson JSON library is in the application's classpath, the object returned from the handler method will be given to the `MappingJacksonHttpMessageConverter` for conversion into a JSON representation to be returned to the client. On the other hand, if the request header indicates that the client prefers `text/xml`, then `Jaxb2RootElementHttpMessageConverter` will be tasked with producing an XML response to the client.

Note that all but three of the HTTP message converters in table 11.2 are registered by default, so no Spring configuration is required to use them. But you may need to add additional libraries to your application's classpath to support them. For instance, if you want to use the `MappingJacksonHttpMessageConverter` to convert JSON messages to and from Java objects, you'll need to add the Jackson JSON Processor[5] library to the classpath.

RECEIVING RESOURCE STATE IN THE REQUEST BODY

On the other side of a RESTful conversation, a client may send us an object in the form of JSON, XML, or some other content type. It'd be inconvenient for our controller's handler methods to receive those objects in their raw form and convert them ourselves. Fortunately, the `@RequestBody` annotation does the same thing for objects sent from the client as `@ResponseBody` does for objects returned to the client.

Let's say that the client submits a `PUT` request with the data for a `Spitter` object represented as JSON in the request's body. To receive that message as a `Spitter` object, we only need to annotate a handler method's `Spitter` parameter with `@RequestBody`:

```
@RequestMapping(value = "/{username}", method = RequestMethod.PUT,
                headers = "Content-Type=application/json")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void updateSpitter(@PathVariable String username,
                      @RequestBody Spitter spitter) {
  spitterService.saveSpitter(spitter);
}
```

When the request arrives, Spring MVC will see that the `updateSpitter()` is able to handle the request. But the message arrives as an XML document, and this method asks for a `Spitter` object. In this case, the `MappingJacksonHttpMessageConverter` may be chosen to convert the JSON message into a `Spitter` object. For that to work, the following criteria must be met:

- The request's `Content-Type` header must be set to `application/json`.
- The Jackson JSON library must be available on the application's classpath.

You may have also noticed that the `updateSpitter()` method is annotated with `@ResponseStatus`. After a PUT request, there's not much to do and no need to return

[5] http://jackson.codehaus.org

anything to the client. By annotating `updateSpitter()` this way, we're saying that HTTP response to the client should have a status code of 204, also known as No Content.

At this point, we've written some Spring MVC controllers with handler methods to handle requests for resources. There are a few more things to talk about with regard to defining a RESTful API using Spring MVC—and we'll get back to that part of the discussion in section 11.5. But first, let's switch gears and see how to use Spring's `RestTemplate` to write client code that consumes those resources.

11.4 Writing REST clients

When we build web applications, we often think of them as having a user interface that resides in a web browser. But with web applications that are made up of RESTful resources, there's no reason that has to be the case. Just because a resource's data is transmitted across the web, that doesn't mean that it'll necessarily be rendered in a web browser. You may even find yourself writing a web application that interacts with another web application through a RESTful API.

Writing code that interacts with a REST resource as a client can involve some tedium and boilerplate. For example, let's say that we need to write some client-side code to consume the Spittles-for-Spitter REST API we developed earlier. The following listing shows one way of getting the job done.

Listing 11.5 REST clients can involve boilerplate code and exception handling.

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
  try {
    HttpClient httpClient = new DefaultHttpClient();

    String spittleUrl = "http://localhost:8080/Spitter/spitters/" +
                        username + "/spittles";

    HttpGet getRequest = new HttpGet(spittleUrl);

    getRequest.setHeader(
            new BasicHeader("Accept", "application/json"));

    HttpResponse response = httpClient.execute(getRequest);

    HttpEntity entity = response.getEntity();
    ObjectMapper mapper = new ObjectMapper();
    return mapper.readValue(entity.getContent(), Spittle[].class);
  } catch (IOException e) {
    throw new SpitterClientException("Unable to retrieve Spittles", e);
  }
}
```

Create HttpClient

Assemble URL

Create request from URL

Execute request

Parse result

As you can see, a lot goes into consuming a REST resource. And I'm even cheating by using Jakarta Commons HTTP Client[6] to make the request and the Jackson JSON processor[7] to parse the response.

[6] http://hc.apache.org/httpcomponents-client/index.html
[7] http://jackson.codehaus.org/

Looking closely at the `retrieveSpittlesForSpitter()` method, you'll realize that little in that method is directly associated with this specific bit of functionality. If you were to write another method that consumed some other REST resource, it'd probably look a lot like this one, with only a few minor differences.

What's more, there are a few places along the way where an `IOException` could've been thrown. Since `IOException` is a checked exception, I'm forced to either catch it or throw it. In this case, I've chosen to catch it and throw an unchecked `SpitterClientException` in its place.

With so much boilerplate involved in resource consumption, you'd think it'd be wise to encapsulate the common code and parameterize the variations. That's precisely what Spring's `RestTemplate` does. Just as `JdbcTemplate` handles the ugly parts of working with JDBC data access, `RestTemplate` frees us from the tedium of consuming RESTful resources.

In a moment, we'll see how we can rewrite the `retrieveSpittlesForSpitter()` method, using `RestTemplate` to dramatically simplify it and eliminate the boilerplate. But first, let's take a high-level survey of all of the REST operations that `RestTemplate` offers.

11.4.1 Exploring RestTemplate's operations

You'll recall from table 11.1 that the HTTP specification defines seven method types for interacting with RESTful resources. These method types provide the verbs in a RESTful conversation.

`RestTemplate` defines 33 methods for interacting with REST resources using all of HTTP's verbs in a variety of ways. Unfortunately, I don't have enough space to go over all 33 methods in this chapter. As it turns out, there are really only 11 unique operations, each of which is overloaded into three method variants. Table 11.3 describes the 11 unique operations provided by `RestTemplate`.

Table 11.3 `RestTemplate` defines 11 unique operations, each of which is overloaded to a total of 33 methods.

Method	Description
`delete()`	Performs an HTTP DELETE on a resource at a specified URL.
`exchange()`	Executes a specified HTTP method against the URL, returning a `ResponseEntity` containing an object mapped from the response body.
`execute()`	Executes a specified HTTP method against the URL, returning an object mapped from the response body.
`getForEntity()`	Sends an HTTP GET request, returning a `ResponseEntity` containing the response body as mapped to an object.
`getForObject()`	GETs a resource, returning the response body as mapped to an object.
`headForHeaders()`	Sends an HTTP HEAD request, returning the HTTP headers for the specified resource URL.

Table 11.3 `RestTemplate` defines 11 unique operations, each of which is overloaded to a total of 33 methods. *(continued)*

Method	Description
`optionsForAllow()`	Sends an HTTP OPTIONS request, returning the `Allow` header for the specified URL.
`postForEntity()`	POSTs data, returning a `ResponseEntity` that contains an object mapped from the response body.
`postForLocation()`	POSTs data, returning the URL of the new resource.
`postForObject()`	POSTs data, returning the response body as mapped to an object.
`put()`	PUTs a resource to the specified URL.

With the exception of `TRACE`, `RestTemplate` covers all of the HTTP verbs. In addition, `execute()` and `exchange()` offer lower-level general-purpose methods for using any of the HTTP methods.

Each of the operations in table 11.3 is overloaded into three method forms:

- One that takes a `java.net.URI` as the URL specification with no support for parameterized URLs
- One that takes a `String` URL specification with URL parameters specified as a `Map`
- One that takes a `String` URL specification with URL parameters specified as a variable argument list

Once you get to know the 11 operations provided by `RestTemplate` and how each of the variant forms work, you'll be well on your way to writing resource-consuming REST clients. Let's survey `RestTemplate`'s operations by looking at those that support the four primary HTTP methods: `GET`, `PUT`, `DELETE`, and `POST`. We'll start with `getForObject()` and `getForEntity()`, the `GET` methods.

11.4.2 GETting resources

You may have noticed that table 11.3 lists two kinds of methods for performing `GET` requests: `getForObject()` and `getForEntity()`. As described earlier, each of these methods are overloaded into three forms. The signatures of the three `getForObject()` methods look like this:

```
<T> T getForObject(URI url, Class<T> responseType)
                                                  throws RestClientException;

<T> T getForObject(String url, Class<T> responseType,
                   Object... uriVariables)  throws RestClientException;

<T> T getForObject(String url, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

Similarly, the signatures of the `getForEntity()` methods are as follows:

```
<T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType)
        throws RestClientException;

<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
        Object... uriVariables) throws RestClientException;

<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

Except for the return type, the `getForObject()` methods are mirror images of the `getForEntity()` methods. And, in fact, they work much the same way. They both perform a `GET` request, retrieving a resource given a URL. And they both map that resource to some type specified by the `responseType` parameter. The only difference is that `getForObject()` simply returns an object of the type requested, whereas `getForEntity()` returns that object along with extra information about the response.

Let's first have a look at the simpler `getForObject()` method. Then we'll see how to get more information from a `GET` response by using the `getForEntity()` method.

RETRIEVING RESOURCES

The `getForObject()` method is a no-nonsense option for retrieving a resource. You ask for a resource and you shall receive that resource mapped to a Java type of your choosing. As a simple example of what `getForObject()` can do, let's take another stab at implementing the `retrieveSpittlesForSpitter()`:

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
  return new RestTemplate().getForObject(
     "http://localhost:8080/Spitter/spitters/{spitter}/spittles",
     Spittle[].class, username);
}
```

Back in listing 11.5, `retrieveSpittlesForSpitter()` involved more than a dozen lines of code. Using `RestTemplate`, it's now reduced to a handful of lines (and could be even less if I didn't have to wrap lines to fit within the margins of this book).

`retrieveSpittlesForSpitter()` starts by constructing an instance of `RestTemplate` (an alternate implementation might've used an injected instance instead). Then it invokes the `getForObject()` method to retrieve the list of `Spittle`s. In doing so, it asks for the results as an array of `Spittle` objects. Upon receiving that array, it returns it to the caller.

Note that in this new version of `retrieveSpittlesForSpitter()` we don't use `String` concatenation to produce the URL. Instead, we take advantage of the fact that `RestTemplate` accepts parameterized URLs. The *{spitter}* placeholder in the URL will ultimately be filled by the `username` parameter of the method. The last argument of `getForObject()` is a variable-sized list of arguments, where each argument is inserted into a placeholder in the specified URL in the order it appears.

Alternatively, we could've placed the `username` parameter into a `Map` with a key of `spitter` and passed that `Map` in as the last parameter to `getForObject()`:

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
    Map<String, String> urlVariables = new HashMap<String, String();
```

```
    urlVariables.put("spitter", username);
    return new RestTemplate().getForObject(
        "http://localhost:8080/Spitter/spitters/{spitter}/spittles",
        Spittle[].class, urlVariables);
}
```

One thing that's absent here is any sort of JSON parsing or object mapping. Under the covers, `getForObject()` converts the response body into an object for us. It does this by relying on the same set of HTTP message converters from table 11.2 that Spring MVC uses for handler methods that are annotated with `@ResponseBody`.

What's also missing from this method is any sort of exception handling. That's not because `getForObject()` couldn't throw an exception, but because any exception it throws is unchecked. If anything goes wrong in `getForObject()`, an unchecked `RestClientException` will be thrown. You can catch it if you'd like—but you're not forced by the compiler to catch it.

EXTRACTING RESPONSE METADATA

As an alternative to `getForObject()`, `RestTemplate` also offers `getForEntity()`. The `getForEntity()` methods work much the same as the `getForObject()` methods. But where `getForObject()` returns only the resource (converted into a Java object by an HTTP message converter), `getForEntity()` returns that same object carried within a `ResponseEntity`. The `ResponseEntity` also carries extra information about the response, such as the HTTP status code and response headers.

One thing you might want to do with a `ResponseEntity` is to retrieve the value of one of the response headers. For example, suppose that in addition to retrieving the resource, you want to know when that resource was last modified. Assuming that the server provides that information in the `Last-Modified` header, you can use the `getHeaders()` method like this:

```
Date lastModified = new Date(response.getHeaders().getLastModified());
```

The `getHeaders()` method returns an `HttpHeaders` object that provides several convenience methods for retrieving response headers, including `getLastModified()`, which returns the number of milliseconds since January 1, 1970.

In addition to `getLastModified()`, `HttpHeaders` includes the following methods for retrieving header information:

```
public List<MediaType> getAccept() { ... }
public List<Charset> getAcceptCharset() { ... }
public Set<HttpMethod> getAllow() { ... }
public String getCacheControl() { ... }
public long getContentLength() { ... }
public MediaType getContentType() { ... }
public long getDate() { ... }
public String getETag() { ... }
public long getExpires() { ... }
public long getIfNotModifiedSince() { ... }
public List<String> getIfNoneMatch() { ... }
public long getLastModified() { ... }
```

```
public URI getLocation() { ... }
public String getPragma() { ... }
```

For more general-purpose HTTP header access, `HttpHeaders` includes a `get()` method and the `getFirst()` method. Both take a `String` argument that identifies the header. The `get()` method returns a list of `String` values, one for each value assigned to the header. The `getFirst()` method returns only the first header value.

If you're interested in the response's HTTP status code, then you'll want to call the `getStatusCode()` method. For example, look at the implementation of `retrieveSpittlesForSpitter()`.

Listing 11.6 A `ResponseEntity` includes the HTTP status code.

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
  ResponseEntity<Spittle[]> response = new RestTemplate().getForEntity(
      "http://localhost:8080/Spitter/spitters/{spitter}/spittles",
      Spittle[].class, username);

  if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {
    throw new NotModifiedException();
  }

  return response.getBody();
}
```

Here, if the server responds with a status of 304, it indicates that the content on the server hasn't been modified since the client previously requested it. In that event, a custom `NotModifiedException` is thrown to indicate that the client should check its cache for the resource data.

11.4.3 PUTting resources

For performing `PUT` operations on a resource, `RestTemplate` offers a simple set of three `put()` methods. As with all of `RestTemplate`'s methods, the `put()` method comes in three forms:

```
void put(URI url, Object request) throws RestClientException;

void put(String url, Object request, Object... uriVariables)
        throws RestClientException;

void put(String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;
```

In its simplest form, the `put()` method takes a `java.net.URI` that identifies (and locates) the resource being sent to the server and an object that's the Java representation of that resource.

For example, here's how you might use the `URI`-based version of `put()` to update a `Spittle` resource on the server:

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  try {
    String url = "http://localhost:8080/Spitter/spittles/" + spittle.getId();
    new RestTemplate().put(new URI(url), spittle);
```

```
  } catch (URISyntaxException e) {
    throw new SpitterUpdateException("Unable to update Spittle", e);
  }
}
```

Here, although the method signature was simple, the implication of using a `java.net.URI` argument is evident. First, in order to create the URL for the `Spittle` object to be updated, we had to do `String` concatenation. Then, because it's possible for a non-URI to be given to the constructor of `URI`, we're forced to catch a `URISyntaxException` (even if we're pretty sure that the given URI is legitimate).

Using one of the other `String`-based `put()` methods alleviates most of the discomfort associated with creating a `URI`, including the need to handle any exceptions. What's more, these methods enable us to specify the URI as a template, plugging in values for the variable parts. Here's a new `updateSpittle()` method rewritten to use one of the `String`-based `put()` methods:

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    restTemplate.put("http://localhost:8080/Spitter/spittles/{id}",
                     spittle,  spittle.getId());
}
```

The URI is now expressed as a simple `String` template. When `RestTemplate` sends the `PUT` request, the URI template will be expanded to replace the *{id}* portion with the value returned from `spittle.getId()`. Just like `getForObject()` and `getForEntity()`, the last argument to this version of `put()` is a variable-sized list of arguments, each of which is assigned to the placeholder variables in the order they appear.

Optionally, you could've passed in the template variables as a `Map`:

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    Map<String, String> params = new HashMap<String, String>();
    params.put("id", spittle.getId());
    restTemplate.put("http://localhost:8080/Spitter/spittles/{id}",
                     spittle,  params);
}
```

When using a `Map` to send the template variables, the key of each entry in the `Map` corresponds to the placeholder variable of the same name in the URI template.

In all versions of `put()`, the second argument is the Java object that represents the resource being `PUT` to the server at the given URI. In this case, it's a `Spittle` object. `RestTemplate` will use one of the message converters from table 11.2 to convert the `Spittle` into a representation to send to the server in the request body.

The content type that the object will be converted into depends largely on the type being passed into `put()`. If given a `String` value, the `StringHttpMessageConverter` kicks in: the value is written directly to the body of the request and the content type is set to `text/plain`. When given a `MultiValueMap<String,String>`, the values in the map will be written to the request body in `application/x-www-form-urlencoded` form by the `FormHttpMessageConverter`.

Since we're passing in a `Spittle` object, we'll need a message converter that can work with arbitrary objects. If the Jackson JSON library is in the classpath, then the `MappingJacksonHttpMessageConverter` will write the `Spittle` to the request as `application/json`. Optionally, if the `Spittle` class were annotated for JAXB serialization and if a JAXB library were on the classpath, then the `Spittle` would be sent as `application/xml` and be written to the request body in XML format.

11.4.4 DELETE-ing resources

When you don't want a resource to be kept around on the server anymore, then you'll want to call `RestTemplate`'s `delete()` methods. Much like the `put()` methods, the `delete()` methods keep it simple with only three versions, whose signatures are as follows:

```
void delete(String url, Object... uriVariables)
        throws RestClientException;

void delete(String url, Map<String, ?> uriVariables)
        throws RestClientException;

void delete(URI url) throws RestClientException;
```

Hands down, the `delete()` methods are the simplest of all of the `RestTemplate` methods. The only thing you need to supply them with is the URI of the resource to be deleted. For example, to get rid of a `Spittle` whose ID is given, you might call `delete()` like this:

```
public void deleteSpittle(long id) {
  try {
    restTemplate.delete(
                  new URI("http://localhost:8080/Spitter/spittles/" + id));
  } catch (URISyntaxException wontHappen) { }
}
```

That's simple enough, but here again we've relied on `String` concatenation to create a `URI` object that defensively throws a checked `URISyntaxException`, which we're forced to catch. So let's turn to one of the simpler versions of `delete()` to get out of that mess:

```
public void deleteSpittle(long id) {
  restTemplate.delete("http://localhost:8080/Spitter/spittles/{id}", id));
}
```

There. I feel better about that. Don't you?

Now that I've shown you the simplest set of `RestTemplate` methods, let's look at `RestTemplate`'s most diverse set of methods—those that support HTTP `POST` requests.

11.4.5 POSTing resource data

Looking back at table 11.3, you see that `RestTemplate` comes with three different kinds of methods for sending `POST` requests. When you multiply that by the three

variants that each is overridden into, that's a total of nine methods for POSTing data to the server.

Two of those methods have names that look familiar. The `postForObject()` and `postForEntity()` methods work with `POST` requests in a way that's similar to how `getForObject()` and `getForEntity()` work for sending `GET` requests. The other method, `getForLocation()`, is unique for `POST` requests.

RECEIVING OBJECT RESPONSES FROM POST REQUESTS

Let's say that you're using `RestTemplate` to `POST` a new `Spitter` object to the Spitter application's REST API. Since it's a brand-new `Spitter`, the server doesn't know about it (yet). Therefore, it's not officially a REST resource yet and doesn't have a URL. Also, the client won't know the ID of the `Spitter` until it's created on the server.

One way of POSTing a resource to the server is to use `RestTemplate`'s `postForObject()` method. The three varieties of `postForObject()` have the following signatures:

```
<T> T postForObject(URI url, Object request, Class<T> responseType)
        throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType,
        Object... uriVariables) throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

In all cases, the first parameter is the URL to which the resource should be POSTed, the second parameter is the object to post, and the third parameter is the Java type expected to be given in return. In the case of the two versions that take the URL as a `String`, a fourth parameter identifies the URL variables (as either a variable arguments list or a `Map`).

When POSTing new `Spitter` resources to the Spitter REST API, they should be posted to http://localhost:8080/Spitter/spitters, where a `POST`-handling controller handler method is waiting to save the object. Since this URL requires no URL variables, we could use any version of `postForObject()`. But, in the interest of keeping it simple and to avoid catching any exceptions that may be thrown while constructing a new `URI`, we'll make the call like this:

```
public Spitter postSpitterForObject(Spitter spitter) {
  RestTemplate rest = new RestTemplate();
  return rest.postForObject("http://localhost:8080/Spitter/spitters",
          spitter, Spitter.class);
}
```

The `postSpitterForObject()` method is given a newly created `Spitter` object and uses `postForObject()` to send it to the server. In response, it receives a `Spitter` object and returns it to the caller.

As with the `getForObject()` methods, we may want to examine some of the metadata that comes back with the request. In that case, `postForEntity()` is the preferred method. `postForEntity()` comes with a set of signatures that mirror those of `postForObject()`:

```
<T> ResponseEntity<T> postForEntity(URI url, Object request,
        Class<T> responseType) throws RestClientException;

<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Object... uriVariables)
        throws RestClientException;

<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Map<String, ?> uriVariables)
        throws RestClientException;
```

So let's say that, in addition to receiving the `Spitter` resource in return, you'd also like to see the value of the `Location` header in the response. In that case you can call `postForEntity()` like this:

```
RestTemplate rest = new RestTemplate();
ResponseEntity<Spitter> response = rest.postForEntity(
    "http://localhost:8080/Spitter/spitters", spitter, Spitter.class);

Spitter spitter = response.getBody();
URI url = response.getHeaders().getLocation();
```

Just like the `getForEntity()` method, `postForEntity()` returns a `ResponseEntity<T>` object. From that object you can call `getBody()` to get the resource object (a `Spitter` in this case). And the `getHeaders()` method gives you an `HttpHeaders` from which you can access the various HTTP headers returned in the response. Here, we're calling the `getLocation()` to retrieve the `Location` header as a `java.net.URI`.

RECEIVING A RESOURCE LOCATION AFTER A POST REQUEST

The `postForEntity()` method is handy for receiving both the resource posted and any response headers. But often you don't need the resource to be sent back to you (after all, you sent it to the server in the first place). If the value of the `Location` header is all you really need to know, then it's even easier to use `RestTemplate`'s `postForLocation()` method.

Like the other `POST` methods, `postForLocation()` sends a resource to the server in the body of a `POST` request. But, instead of responding with that same resource object, `postForLocation()` responds with the location of the newly created resource. It has the following three method signatures:

```
URI postForLocation(String url, Object request, Object... uriVariables)
        throws RestClientException;

URI postForLocation(
        String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;

URI postForLocation(URI url, Object request) throws RestClientException;
```

To demonstrate `postForLocation()`, let's try `POST`ing a `Spitter` again. This time, we want the resource's URL in return:

```
public String postSpitter(Spitter spitter) {
  RestTemplate rest = new RestTemplate();
  return rest.postForLocation("http://localhost:8080/Spitter/spitters",
            spitter).toString();
}
```

Here, we're passing in the target URL as a `String`, along with the `Spitter` object to be `POST`ed (there are no URL variables in this case). If, after creating the resource, the server responds with the new resource URL in the response's `Location` header, then `postForLocation()` will return that URL as a `String`.

11.4.6 Exchanging resources

Up to this point, we've seen all manner of `RestTemplate` methods for `GET`ting, `PUT`-ting, `DELETE`-ing, and `POST`ing resources. Among those we saw two special methods, `getForEntity()` and `postForEntity()`, that gave us the resulting resource wrapped in a `RequestEntity` from which we could retrieve response headers and status codes.

Being able to read headers from the response is useful. But what if we want to set headers on the request sent to the server? That's what `RestTemplate`'s `exchange()` methods are good for.

Like all of the other methods in `RestTemplate`, `exchange()` is overloaded into three signature forms as shown here:

```
<T> ResponseEntity<T> exchange(URI url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType)
        throws RestClientException;

<T> ResponseEntity<T> exchange(String url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType,
        Object... uriVariables) throws RestClientException;

<T> ResponseEntity<T> exchange(String url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

As you can see, the three `exchange()` signatures are overloaded to match the same pattern as the other `RestTemplate` methods. One takes a `java.net.URI` to identify the target URL, whereas the other two take the URL in `String` form with URL variables.

The `exchange()` method also takes an `HttpMethod` parameter to indicate the HTTP verb that should be used. Depending on the value given to this parameter, the `exchange()` method can perform the same jobs as any of the other `RestTemplate` methods.

For example, one way to retrieve a `Spitter` resource from the server is to use `RestTemplate`'s `getForEntity()` method like this:

```
ResponseEntity<Spitter> response = rest.getForEntity(
          "http://localhost:8080/Spitter/spitters/{spitter}",
          Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

As you can see here, `exchange()` is also up to the task:

```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/Spitter/spitters/{spitter}",
          HttpMethod.GET, null, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

By passing in `HttpMethod.GET` as the HTTP verb, we're asking `exchange()` to send a `GET` request. The third argument is for sending a resource on the request, but since this is a `GET` request, it can be `null`. The next argument indicates that we want the response converted into a `Spitter` object. And the final argument is the value to place into the {spitter} placeholder in the specified URL template.

Used this way, the `exchange()` method is virtually identical to the previously used `getForEntity()`. But unlike `getForEntity()`—or `getForObject()`—`exchange()` will let us set headers on the request sent. Instead of passing `null` to `exchange()`, we'll pass in an `HttpEntity` created with the request headers we want.

Without specifying the headers, `exchange()` will send the `GET` request for a `Spitter` with the following headers:

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/xml, text/xml, application/*+xml, application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

Take a look at the `Accept` header. It says that it can accept several different XML content types as well as `application/json`. The leaves a lot of room for the server to decide which format to send the resource back as. Suppose that we want to demand that the server send the response back as JSON. In that case, we need to specify `application/json` as the only value in the `Accept` header.

Setting request headers is a simple matter of constructing the `HttpEntity` sent to `exchange()` with a `MultiValueMap` loaded with the desired headers:

```
MultiValueMap<String, String> headers =
    new LinkedMultiValueMap<String, String>();
headers.add("Accept", "application/json");
HttpEntity<Object> requestEntity = new HttpEntity<Object>(headers);
```

Here, we create a `LinkedMultiValueMap` and add an `Accept` header set to `application/json`. Then we construct an `HttpEntity` (with a generic type of `Object`), passing the `MultiValueMap` as a constructor argument. If this were a `PUT` or a `POST` request, we would've also given the `HttpEntity` an object to send in the body of the request—for a `GET` request, this isn't necessary.

Now we can call `exchange()` passing in the `HttpEntity`:

```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/Spitter/spitters/{spitter}",
          HttpMethod.GET, requestEntity, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

On the surface, the results should be the same. We should receive the `Spitter` object that we asked for. Under the surface, the request will be sent with the following headers:

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/json
Content-Length: 0
```

```
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

And, assuming that the server can serialize the `Spitter` response into JSON, the response body should be represented in JSON format.

In this section we've seen how using the various methods that `RestTemplate` provides, you can write Java-based clients that interact with RESTful resources on the server. But what if the client is browser-based? When a web browser is calling on REST resources, there are some limitations to be accounted for—specifically the range of HTTP methods supported in the browser. To wrap up this chapter, let's see how Spring can help overcome those limitations.

11.5 Submitting RESTful forms

We've seen how the four primary HTTP methods—`GET`, `POST`, `PUT`, and `DELETE`—define the basic operations that can be performed on a resource. And by setting the `method` attribute of the `@RequestMapping` annotation appropriately, we can cause `DispatcherServlet` to direct requests for those HTTP verbs to specific controller methods. Spring MVC can handle requests for any of the HTTP methods—assuming the client sends the requests in the form of the desired HTTP method.

The gotchas in that plan are HTML and the web browser. Non-browser clients, such as those that use `RestTemplate`, should have no trouble sending requests to perform any of the HTTP verbs. But HTML 4 only officially supports `GET` and `POST` in forms, leaving `PUT`, `DELETE`, and all other HTTP methods in the cold. Even though HTML 5 and newer browsers will support all of the HTTP methods, you probably can't count on the users of your application to be using a modern browser.

A common trick used to get around the shortcomings of HTML 4 and older browsers is to masquerade a `PUT` or `DELETE` request in the form of a `POST` request. The way it works is to submit a browser-pleasing `POST` request with a hidden field that carries the name of the actual HTTP method. When the request arrives at the server, it's rewritten to be whatever type of request was specified in the hidden field.

Spring supports `POST` masquerading through two features:

- Request transformation with `HiddenHttpMethodFilter`
- Hidden field rendering with the `<sf:form>` JSP tag

Let's first look at how Spring's `<sf:form>` tag can help render a hidden field for `POST` masquerading.

11.5.1 Rendering hidden method fields in JSP

In section 7.4.1, we saw how to use Spring's form-binding library to render HTML forms. The core element of that JSP tag library is the `<sf:form>` tag. As you'll recall, that tag sets the content for the other form-binding tags, associating the rendered form and its fields with a model attribute.

At that time, we used `<sf:form>` to define a form that was used to create a new `Spitter` object. In that case, a `POST` request was appropriate, as `POST` is often used to create new resources. But what if we wanted to update or delete a resource? In those situations, a `PUT` or `DELETE` request seems more fitting.

But as I've already mentioned, HTML's `<form>` tag can't be trusted to send anything other than a `GET` or `POST` request. Though some newer browsers won't have any trouble with a `<form>` tag whose `method` attribute is set to `PUT` or `DELETE`, accounting for older browsers will require sneaking the request to the server as a `POST`.

Within an HTML form, the key to masquerading a `PUT` or `DELETE` request as a `POST` is to create a form whose method is `POST`, along with a hidden field. For example, the following snippet of HTML shows how you might create a form that submits a `DELETE` request:

```
<form method="post">
    <input type="hidden" name="_method" value="delete"/>

    ...
</form>
```

As you can see, it's not a big deal to create a form with the hidden field that specifies the real HTTP method. All you need to do is to add a hidden field with a name that the form and the server can agree upon, and set that field to the desired HTTP method name. When this form is submitted, a `POST` request will be sent to the server. Presumably, the server will interpret the `_method` field to be the actual type of method to process (we'll see how to configure the server to do that in a moment).

When using Spring's form-binding library the `<sf:form>` can make this even easier. You set the `method` attribute to the desired HTTP method and `<sf:form>` will take care of the hidden field for you:

```
<sf:form method="delete" modelAttribute="spitter">
    ...
</sf:form>
```

When `<sf:form>` is rendered into HTML, the result will be quite similar to the HTML `<form>` shown before. Using `<sf:form>` frees you from having to deal with the hidden field, letting you work with `PUT` and `DELETE` forms in a more natural way, as if they were supported by the browser.

The `<sf:form>` tag only tells the browser's side of the `POST`-masquerade story. How does the server know what to do with those `POST` requests that should be handled as `PUT` and `DELETE` requests?

11.5.2 Unmasking the real request

When the browser submits a `PUT` or `DELETE` request from a form rendered by `<sf:form>`, it's in every way a `POST` request. It travels across the network as a `POST` request, arrives at the server as a `POST` request, and unless something on the server bothers to look at the hidden `_method` field, it'll be processed as a `POST` request.

Meanwhile, our controller's handler methods are annotated with `@RequestMapping`, ready to process `PUT` and `DELETE` requests. Somehow, this HTTP method mismatch must be resolved before `DispatcherServlet` tries to find a controller handler method to route them to. That's the job that Spring's `HiddenHttpMethodFilter` was born to do.

`HiddenHttpMethodFilter` is a servlet filter and is configured in web.xml:

```
<filter>
    <filter-name>httpMethodFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.HiddenHttpMethodFilter
    </filter-class>
</filter>
...
<filter-mapping>
    <filter-name>httpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Here, we've mapped `HiddenHttpMethodFilter` to the `/*` URL pattern. This is so requests for all URLs can pass through `HiddenHttpMethodFilter` on their way to `DispatcherServlet`.

As illustrated in figure 11.3, `HiddenHttpMethodFilter` transforms `PUT` and `DELETE` requests that are masquerading as `POST` requests into their true form. When a `POST` request arrives at the server, `HiddenHttpMethodFilter` sees that the `_hidden` field prescribes a different request type and rewrites it in its destined HTTP method type.

By the time `DispatcherServlet` and your controller methods see the request, it'll have been transformed. Nobody will be the wiser that the request actually started its life as a `POST` request. Between `<sf:form>`'s automatic rendering of the hidden field and `HiddenHttpMethodFilter`'s ability to transform requests based on the value of that hidden field, your JSP forms and your Spring MVC controllers needn't concern themselves with how the browser-unsupported HTTP methods are handled.

Before we leave the topic of `POST`-masquerading requests, I should remind you that this technique exists only as a workaround for lack of support for `PUT` and `DELETE` requests in HTML 4 and older browsers. Requests sent from non-browser clients,

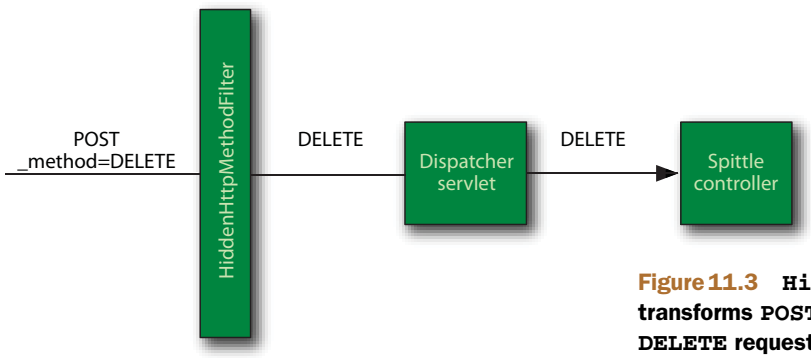

Figure 11.3 `HiddenHttpMethodFilter` transforms `POST`-masquerading `PUT` and `DELETE` requests into their true form.

including those sent from `RestTemplate`, can be sent as whatever HTTP verb is appropriate and don't need to be carried along in a `POST` request. Thus, if you won't be handling any `PUT` and `DELETE` requests from a browser form, then you won't need the services of `HiddenHttpMethodFilter`.

11.6 Summary

RESTful architecture leverages web standards to integrate applications, keeping the interactions simple and natural. Resources in a system are identified by URLs, manipulated with HTTP methods, and represented in one or more forms suitable for the client.

In this chapter, we've seen how to write Spring MVC controllers that respond to requests to manipulate RESTful resources. By utilizing parameterized URL patterns and associating controller handler methods with specific HTTP methods, controllers can respond to `GET`, `POST`, `PUT`, and `DELETE` requests for the resources in an application.

In response to those requests, Spring can represent the data behind those resources in a format that's best for the client. For view-based responses, `ContentNegotiatingViewResolver` can select the best view produced from several view resolvers to satisfy the client's desired content type. Or a controller handler method can be annotated with `@ResponseBody` to completely bypass view resolution and have one of several message converters convert the returned value into a response for the client.

On the client side of REST conversations, Spring provides `RestTemplate`, a template-based approach to consuming RESTful resources from Java. And when the client is browser-based, Spring's `HiddenHttpMethodFilter` can make up for the lack of support for `PUT` and `DELETE` methods in web browsers.

Although the RESTful interactions we've seen in this chapter and the RPC conversations that we covered in the previous chapter are quite different, they share a common trait: they're synchronous in nature. When the client sends a message, the server is expected to be ready to answer immediately. In contrast, asynchronous communication allows the server to react to a message as opportunity allows and not necessarily right away. In the next chapter, we're going to see how to use Spring to integrate applications asynchronously.