Creating REST APIs with Spring MVC

How Spring supports REST

Now, at version 4.0, Spring supports the creation 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`. Spring 3.2 and higher also supports the `PATCH` method.
- The `@PathVariable` annotation enables controllers to handle requests for parameterized URLs (URLs that have variable input as part of their path).
- Resources can be represented in a variety of ways using Spring views and view resolvers, including `View` implementations for rendering model data as XML, JSON, Atom, and RSS.
- The representation best suited for the client can be chosen using `ContentNegotiatingViewResolver`.

How Spring supports REST

- View-based rendering can be bypassed altogether using the `@ResponseBody` annotation and various `HttpMethodConverter` implementations.
- Similarly, the `@RequestBody` annotation, along with `HttpMethodConverter` implementations, can convert inbound HTTP data into Java objects passed in to a controller's handler methods.
- Spring applications can consume REST resources using `RestTemplate`.

Creating a REST endpoint

NOTE Although Spring supports a variety of formats for representing resources, you aren't obligated to use them all when defining your REST API. JSON and XML are often sufficient representations expected by most clients.

Creating a REST endpoint

Spring offers two options to transform a resource's Java representation into the representation that's shipped to the client:

- *Content negotiation*—A view is selected that can render the model into a representation to be served to the client.
- *Message conversion*—A message converter transforms an object returned from the controller into a representation to be served to the client.

Negotiating resource representation

Spring's `ContentNegotiatingViewResolver` is a special view resolver that takes the content type that the client wants into consideration. In it's simplest possible form, `ContentNegotiatingViewResolver` can be configured like this:

```
@Bean
public ViewResolver cnViewResolver() {
  return new ContentNegotiatingViewResolver();
}
```

A lot is going on in that simple bean declaration. 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).

ContentNegotiationManager

A few of the things you can do via a `ContentNegotiationManager` are as follows:

- Specify a default content type to fall back to if a content type can't be derived from the request.
- Specify a content type via a request parameter.
- Ignore the request's `Accept` header.
- Map request extensions to specific media types.
- Use the Java Activation Framework (JAF) as a fallback option for looking up media types from extensions.

ContentNegotiationManager

There are three ways to configure a `ContentNegotiationManager`:

- Directly declare a bean whose type is `ContentNegotiationManager`.
- Create the bean indirectly via `ContentNegotiationManagerFactoryBean`.
- Override the `configureContentNegotiation()` method of `WebMvcConfigurer-Adapter`.

Creating a `ContentNegotiationManager` directly is a bit involved and not something you'll want to do unless you have good reason to. The other two options exist to make the creation of a `ContentNegotiationManager` easier.

ContentNegotiationManager

Generally speaking, `ContentNegotiationManagerFactoryBean` is most useful when you're configuring the `ContentNegotiationManager` in XML. For example, you might configure a `ContentNegotiationManager` with a default content type of `application/json` in XML like this:

```
<bean id="contentNegotiationManager"
  class="org.springframework.http.ContentNegotiationManagerFactoryBean"
  p:defaultContentType="application/json">
```

ContentNegotiationManager

- In Java configuration:

```
@Override
public void configureContentNegotiation(
    ContentNegotiationConfigurer configurer) {
  configurer.defaultContentType(MediaType.APPLICATION_JSON);
}
```

ContentNegotiationManager

Now that you have a ContentNegotiationManager bean, all you need to do is inject it into the contentNegotiationManager property of ContentNegotiating-ViewResolver. That requires a small change to the @Bean method where you declare the ContentNegotiatingViewResolver:

```
@Bean
public ViewResolver cnViewResolver(ContentNegotiationManager cnm) {
  ContentNegotiatingViewResolver cnvr =
      new ContentNegotiatingViewResolver();
  cnvr.setContentNegotiationManager(cnm);
  return cnvr;
}
```

ContentNegotiationManager

Listing 16.2 Configuring a ContentNegotiationManager

```
@Bean
public ViewResolver cnViewResolver(ContentNegotiationManager cnm) {
  ContentNegotiatingViewResolver cnvr =
      new ContentNegotiatingViewResolver();
  cnvr.setContentNegotiationManager(cnm);
  return cnvr;
}

@Override
public void configureContentNegotiation(
    ContentNegotiationConfigurer configurer) {
  configurer.defaultContentType(MediaType.TEXT_HTML);        <— Default to HTML
 }

@Bean
public ViewResolver beanNameViewResolver() {        <— Look up views as beans
   return new BeanNameViewResolver();
}

@Bean
public View spittles() {
  return new MappingJackson2JsonView();        <— "spittles" JSON view
 }
```

ContentNegotiatingViewResolver

- Client expectation:

```
[
  {
    "id": 42,
    "latitude": 28.419489,
    "longitude": -81.581184,
    "message": "Hello World!",
    "time": 1400389200000
  },
  {
    "id": 43,
    "latitude": 28.419136,
    "longitude": -81.577225,
    "message": "Blast off!",
    "time": 1400475600000
  }
]
```

actual response:

```
{
    "spittleList": [
        {
            "id": 42,
            "latitude": 28.419489,
            "longitude": -81.581184,
            "message": "Hello World!",
            "time": 1400389200000
        },
        {
            "id": 43,
            "latitude": 28.419136,
            "longitude": -81.577225,
            "message": "Blast off!",
            "time": 1400475600000
        }
    ]
}
```

Working with HTTP message converters

Table 16.1 Spring provides several HTTP message converters that marshal resource representations to and from various Java types.

Message converter	Description
`AtomFeedHttpMessageConverter`	Converts Rome `Feed` objects to and from Atom feeds (media type `application/atom+xml`). *Registered if the Rome library is present on the classpath.*
`BufferedImageHttpMessageConverter`	Converts `BufferedImage` to and from image binary data.
`ByteArrayHttpMessageConverter`	Reads and writes byte arrays. Reads from all media types (`*/*`), and writes as `application/octet-stream`.

Message converter	Description
`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 (either `text/xml` or `application/xml`) to and from JAXB2-annotated objects. *Registered if JAXB v2 libraries are present on the classpath.*
`MappingJacksonHttpMessageConverter`	Reads and writes JSON to and from typed objects or untyped `HashMaps`. *Registered if the Jackson JSON library is present on the classpath.*
`MappingJackson2HttpMessageConverter`	Reads and writes JSON to and from typed objects or untyped `HashMaps`. *Registered if the Jackson 2 JSON library is present on the classpath.*

Message converter	Description
`MarshallingHttpMessageConverter`	Reads and writes XML using an injected marshaler and unmarshaler. Supported (un)marshalers include Castor, JAXB2, JIBX, XMLBeans, and XStream.
`ResourceHttpMessageConverter`	Reads and writes `org.springframework.core.io.Resource`.
`RssChannelHttpMessageConverter`	Reads and writes RSS feeds to and from Rome `Channel` objects. *Registered if the Rome library is present on the classpath.*
`SourceHttpMessageConverter`	Reads and writes XML to and from `javax.xml.transform.Source` objects.
`StringHttpMessageConverter`	Reads all media types (`*/*`) into a `String`. Writes `String` to `text/plain`.
`XmlAwareFormHttpMessageConverter`	An extension of `FormHttpMessageConverter` that adds support for XML-based parts using a `SourceHttpMessageConverter`.

Returning resource state in the response body

Revisiting the `spittles()` method from listing 16.1, you can add `@ResponseBody` to have Spring convert the returned `List<Spittle>` to the body of the response:

```
@RequestMapping(method=RequestMethod.GET,
                produces="application/json")
public @ResponseBody List<Spittle> spittles(
    @RequestParam(value="max",
                  defaultValue=MAX_LONG_AS_STRING) long max,
    @RequestParam(value="count", defaultValue="20") int count) {

  return spittleRepository.findSpittles(max, count);
}
```

The `@ResponseBody` annotation tells Spring that you want to send the returned object as a resource to the client, converted into some representational form that the client can accept. More specifically, `DispatcherServlet` considers the request's `Accept` header and looks for a message converter that can give the client the representation it wants.

Returning resource state in the response body

The message converter will convert the `Spittle` list returned from the controller into a JSON document that will be written to the body of the response. That response might look a little something like this:

```
[
  {
    "id": 42,
    "latitude": 28.419489,
    "longitude": -81.581184,
    "message": "Hello World!",
    "time": 1400389200000
  },
  {
    "id": 43,
    "latitude": 28.419136,
    "longitude": -81.577225,
    "message": "Blast off!",
    "time": 1400475600000
  }
]
```

Receiving resource state in the request body

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public @ResponseBody
    Spittle saveSpittle(@RequestBody Spittle spittle) {
  return spittleRepository.save(spittle);
}
```

Defaulting controllers for message conversion

The key thing to notice in listing 16.3 is what's not in the code. Neither of the handler methods are annotated with @ResponseBody.

But because the controller is annotated with @RestController, the objects returned from those methods will still go through message conversion to produce a resource representation for the client.

```
import spittr.data.SpittleRepository;

@RestController                                  ⟵ Default to message conversion
@RequestMapping("/spittles")
public class SpittleController {

  private static final String MAX_LONG_AS_STRING="9223372036854775807";

  private SpittleRepository spittleRepository;

  @Autowired
  public SpittleController(SpittleRepository spittleRepository) {
    this.spittleRepository = spittleRepository;
  }

  @RequestMapping(method=RequestMethod.GET)
  public List<Spittle> spittles(
      @RequestParam(value="max",
                    defaultValue=MAX_LONG_AS_STRING) long max,
      @RequestParam(value="count", defaultValue="20") int count) {

    return spittleRepository.findSpittles(max, count);
  }

  @RequestMapping(
      method=RequestMethod.POST
      consumes="application/json")
  public Spittle saveSpittle(@RequestBody Spittle spittle) {
    return spittleRepository.save(spittle);
  }
```

Communicating errors to the client

Spring offers a few options for dealing with such scenarios:

- Status codes can be specified with the `@ResponseStatus` annotation.
- Controller methods can return a `ResponseEntity` that carries more metadata concerning the response.
- An exception handler can deal with the error cases, leaving the handler methods to focus on the happy path.

Working with ResponseEntity

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  HttpStatus status = spittle != null ?
                      HttpStatus.OK : HttpStatus.NOT_FOUND;
  return new ResponseEntity<Spittle>(spittle, status);
}
```

Working with ResponseEntity

```
public class Error {
  private int code;
  private String message;

  public Error(int code, String message) {
    this.code = code;
    this.message = message;
  }

  public int getCode() {
    return code;
  }

  public String getMessage() {
    return message;
  }
}
```

Then you can change spittleById() to return the Error:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<?> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) {
    Error error = new Error(4, "Spittle [" + id + "] not found");
    return new ResponseEntity<Error>(error, HttpStatus.NOT_FOUND);
  }
  return new ResponseEntity<Spittle>(spittle, HttpStatus.OK);
}
```

Handling errors

Let's refactor some of the code to take advantage of an error handler. Begin by defining an error handler that reacts to a SpittleNotFoundException:

```
@ExceptionHandler(SpittleNotFoundException.class)
public ResponseEntity<Error> spittleNotFound(
                                        SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  Error error = new Error(4, "Spittle [" + spittleId + "] not found");
  return new ResponseEntity<Error>(error, HttpStatus.NOT_FOUND);
}
```

Handling errors

As for SpittleNotFoundException, it's a fairly basic exception class:

```
public class SpittleNotFoundException extends RuntimeException {
  private long spittleId;
  public SpittleNotFoundException(long spittleId) {
    this.spittleId = spittleId;
  }

  public long getSpittleId() {
    return spittleId;
  }
}
```

Now you can remove most of the error handling from the spittleById() method:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) { throw new SpittleNotFoundException(id); }
  return new ResponseEntity<Spittle>(spittle, HttpStatus.OK);
}
```

Handling errors

Knowing that the error handler method always returns an `Error` and always responds with an HTTP status code of 404 (Not Found), you can apply a similar cleanup process to `spittleNotFound()`:

```
@ExceptionHandler(SpittleNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public @ResponseBody Error spittleNotFound(SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  return new Error(4, "Spittle [" + spittleId + "] not found");
}
```

Handling errors

Again, if the controller class is annotated with `@RestController`, you can remove the `@ResponseBody` annotation and clean up the code a little more:

```
@ExceptionHandler(SpittleNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public Error spittleNotFound(SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  return new Error(4, "Spittle [" + spittleId + "] not found");
}
```

Setting headers in the response

Listing 16.4 Setting headers in the response when returning a ResponseEntity

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public ResponseEntity<Spittle> saveSpittle(
                        @RequestBody Spittle spittle) {

  Spittle spittle = spittleRepository.save(spittle);      ⟵ Fetch spittle

  HttpHeaders headers = new HttpHeaders();      ⟵ Set the location header
   URI locationUri = URI.create(
       "http://localhost:8080/spittr/spittles/" + spittle.getId());
  headers.setLocation(locationUri);

  ResponseEntity<Spittle> responseEntity =      ⟵ Create a ResponseEntity
       new ResponseEntity<Spittle>(
           spittle, headers, HttpStatus.CREATED)
  return responseEntity;
}
```

Setting headers in the response

Listing 16.5 Using a `UriComponentsBuilder` to construct the location URI

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public ResponseEntity<Spittle> saveSpittle(
      @RequestBody Spittle spittle,
      UriComponentsBuilder ucb) {          <— Given a UriComponentsBuilder ...

  Spittle spittle = spittleRepository.save(spittle);

  HttpHeaders headers = new HttpHeaders();     <— ... calculate the location URI
   URI locationUri =
      ucb.path("/spittles/")
         .path(String.valueOf(spittle.getId()))
         .build()
         .toUri();
  headers.setLocation(locationUri);

  ResponseEntity<Spittle> responseEntity =
      new ResponseEntity<Spittle>(
          spittle, headers, HttpStatus.CREATED)
  return responseEntity;
}
```

Consuming REST resources

Listing 16.6 Fetching a Facebook profile using Apache HTTP Client

```
public Profile fetchFacebookProfile(String id) {
  try {
    HttpClient client = HttpClients.createDefault();

    HttpGet request = new HttpGet("http://graph.facebook.com/" + id);
     request.setHeader("Accept", "application/json");

    HttpResponse response = client.execute(request);

    HttpEntity entity = response.getEntity();
    ObjectMapper mapper = new ObjectMapper();
     return mapper.readValue(entity.getContent(), Profile.class);
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
}
```

Create the client

Create the request

Execute the request

Map response to object

Exploring Rest Template's operations

RestTemplate defines 36 methods for interacting with REST resources

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

Method	Description
headForHeaders()	Sends an HTTP HEAD request, returning the HTTP headers for the specified resource URL
optionsForAllow()	Sends an HTTP OPTIONS request, returning the Allow header for the specified URL
postForEntity()	POSTs data to a URL, returning a ResponseEntity containing an object mapped from the response body
postForLocation()	POSTs data to a URL, returning the URL of the newly created resource
postForObject()	POSTs data to a URL, returning an object mapped from the response body
put()	PUTs resource data to the specified URL

Exploring Rest Template's operations

Most of the operations in table 16.2 are 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

GETting resources

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;
```

Retrieving resources

As a simple example of what `getForObject()` can do, let's take another stab at implementing `fetchFacebookProfile()`:

```
public Profile fetchFacebookProfile(String id) {
  RestTemplate rest = new RestTemplate();
  return rest.getForObject("http://graph.facebook.com/{spitter}",
      Profile.class, id);
}
```

Retrieving resources

Alternatively, you could place the `id` parameter into a `Map` with a key of `id` and pass in that `Map` as the last parameter to `getForObject()`:

```
public Spittle[] fetchFacebookProfile(String id) {
  Map<String, String> urlVariables = new HashMap<String, String();
  urlVariables.put("id", id);
  RestTemplate rest = new RestTemplate();
  return rest.getForObject("http://graph.facebook.com/{spitter}",
        Profile.class, urlVariables);
}
```

Extracting response metadata

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 List<String> getConnection() { ... }
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 getOrigin() { ... }
public String getPragma() { ... }
public String getUpgrade() { ... }
```

PUTting resources

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;
```

PUTting resources

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 {
  RestTemplate rest = new RestTemplate();
  String url = "http://localhost:8080/spittr-api/spittles/"
               + spittle.getId();
  rest.put(URI.create(url), spittle);
}
```

PUTting resources

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  RestTemplate rest = new RestTemplate();
  rest.put("http://localhost:8080/spittr-api/spittles/{id}",
           spittle,  spittle.getId());
}
```

PUTting resources

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

DELETEing resources

Much like the put() methods, the thods.
delete() methods have 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;
```

DELETEing resources

```
public void deleteSpittle(long id) {
  RestTemplate rest = new RestTemplate();
  rest.delete(
      URI.create("http://localhost:8080/spittr-api/spittles/" + id));
}
```

That's easy enough, but here again you rely on `String` concatenation to create a `URI` object. Let's turn to one of the simpler versions of `delete()` to avoid doing so:

```
public void deleteSpittle(long id) {
  RestTemplate rest = new RestTemplate();
  rest.delete("http://localhost:8080/spittr-api/spittles/{id}", id));
}
```

Receiving object responses from POST requests

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;
```

Receiving object responses from POST requests

When you POST new Spitter resources to the Spitter REST API, they should be posted to http://localhost:8080/spittr-api/spitters, where a POST-handling controller handler method is waiting to save the object. Because this URL requires no URL variables, you can use any version of postForObject(). But in the interest of keeping it simple, let's make the call like this:

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

Receiving object responses from POST requests

As with the `getForObject()` methods, you 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;
```

Receiving object responses from POST requests

Suppose 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/spittr-api/spitters",
    spitter, Spitter.class);
Spitter spitter = response.getBody();
URI url = response.getHeaders().getLocation();
```

Receiving a resource location after a POST request

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;
```

Receiving a resource location after a POST request

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

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

Exchanging resources

Like all the other methods in `RestTemplate`, `exchange()` is overloaded into three signature forms. One takes a `java.net.URI` to identify the target URL, whereas the other two take the URL in `String` form with URL variables, 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;
```

Exchanging resources

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/spittr-api/spitters/{spitter}",
          Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

As you can see in the next snippet of code, exchange() is also up to the task:

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

Exchanging resources

Without specifying the headers, `exchange()` sends 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
```

Exchanging resources

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);
```

Exchanging resources

Now you can call exchange(), passing in the HttpEntity:

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

Exchanging resources

On the surface, the results should be the same. You should receive the `Spitter` object that you asked for. Under the surface, the request is 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.

Creating REST APIs with Spring MVC