SE325.K21 – Chuyên đề J2EE

Giving Spring Some Rest

(Spring in Action 3rd Edition)

Vương Thịnh Đạt – 17530343

Đinh Hoàng Nhi – 17520853

Đinh Hoàng Luôn – 17520727

Giving Spring Some Rest

A. Getting Rest

Fundamental & How Spring Support Rest

B. Writing Resource-oriented Controllers

Restless - Restful Controllers & Rest verbs

C. Representing Resources

Negotiating resource representation

HTTP message converters

D. Writing Rest Clients

Rest templates operations

GET – PUT – DELETE – POST resources

Exchanging resources

E. Submitting RESTful forms

Rendering hidden method fields in JSP

Unmasking the real requests

A. Getting Rest

Fundamental & How Spring Support Rest

A. Getting Rest

REST: Representational State Transfer

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

REST is about transferring the state of resources from a server to a client

A. Getting Rest

How Spring supports development of REST resources

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.

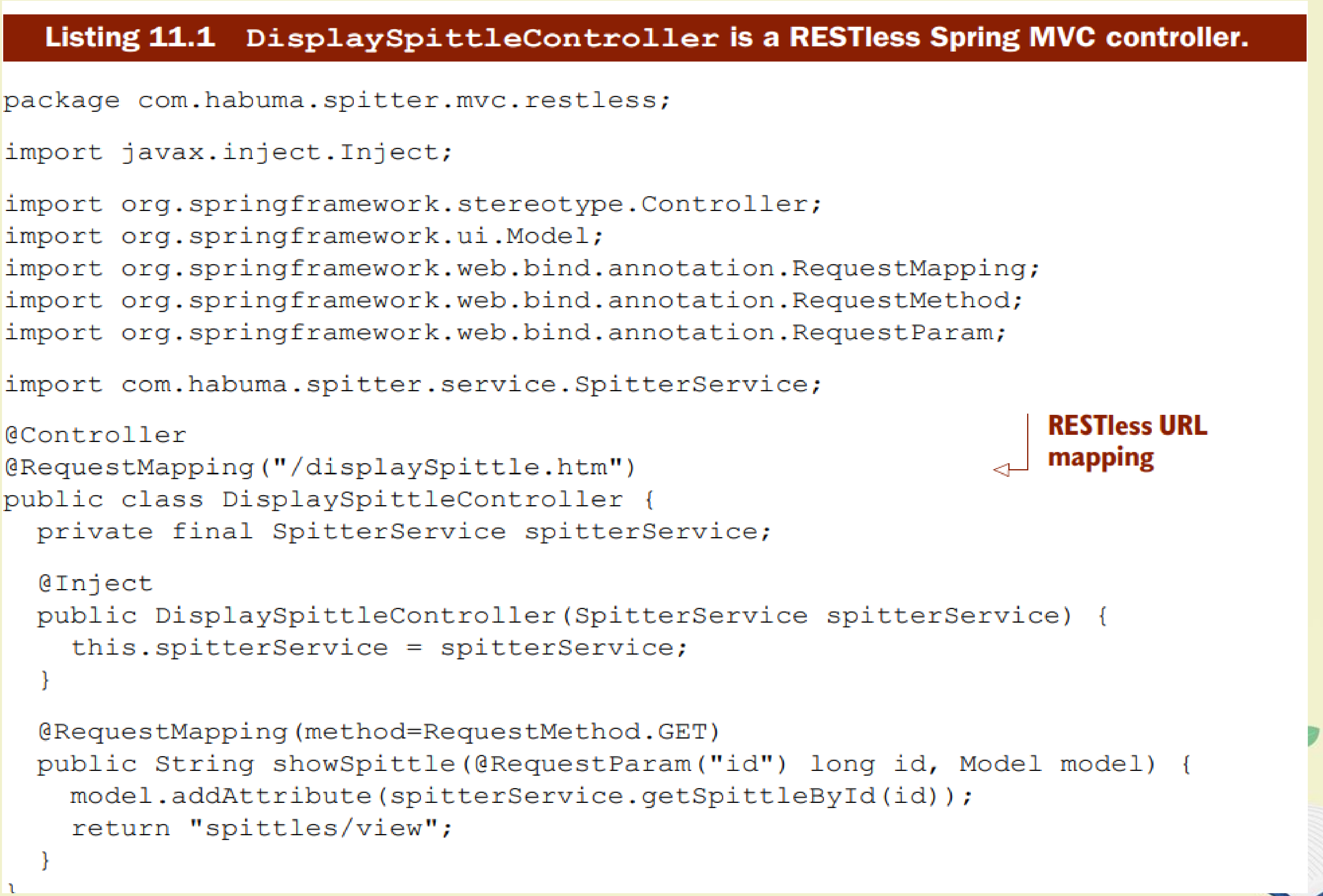
The JSP tag from Spring's form-binding JSP tag library, along with the new HiddenHttpMethodFilter.

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.

B. Writing Resource
- Oriented Controllers

Restless – Restful Controllers & Rest verbs

B. Writing Resource-oriented Controllers



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

B. Writing Resource-oriented Controllers

RESTless & RESTful URL

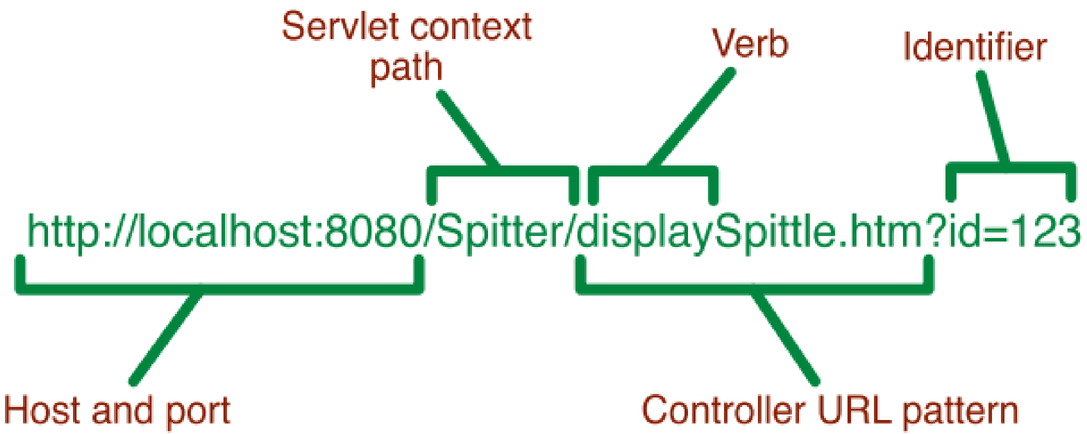


Figure 11.1 **A RESTless URL is action-oriented and doesn't identify or locate a resource.**

Many RESTless URLs don't locate or identify anything — they make demands. Rather than identify a thing, they insist that some action be taken.

B. Writing Resource-oriented Controllers

RESTless & RESTful URL

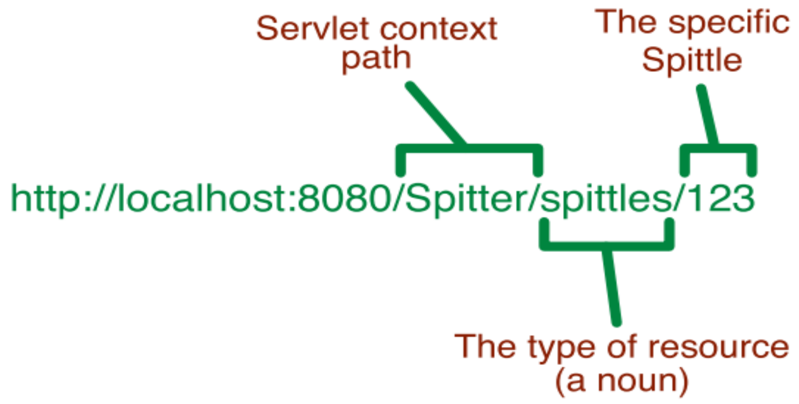

Servlet context path
The specific Spittle
http://localhost:8080/Spitter/spittles/123
The type of resource (a noun)

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

RESTful URLs fully acknowledge that HTTP is all about resources. What it does will be depended on the HTTP methods.

B. Writing resource-oriented controllers

RESTful URL observation

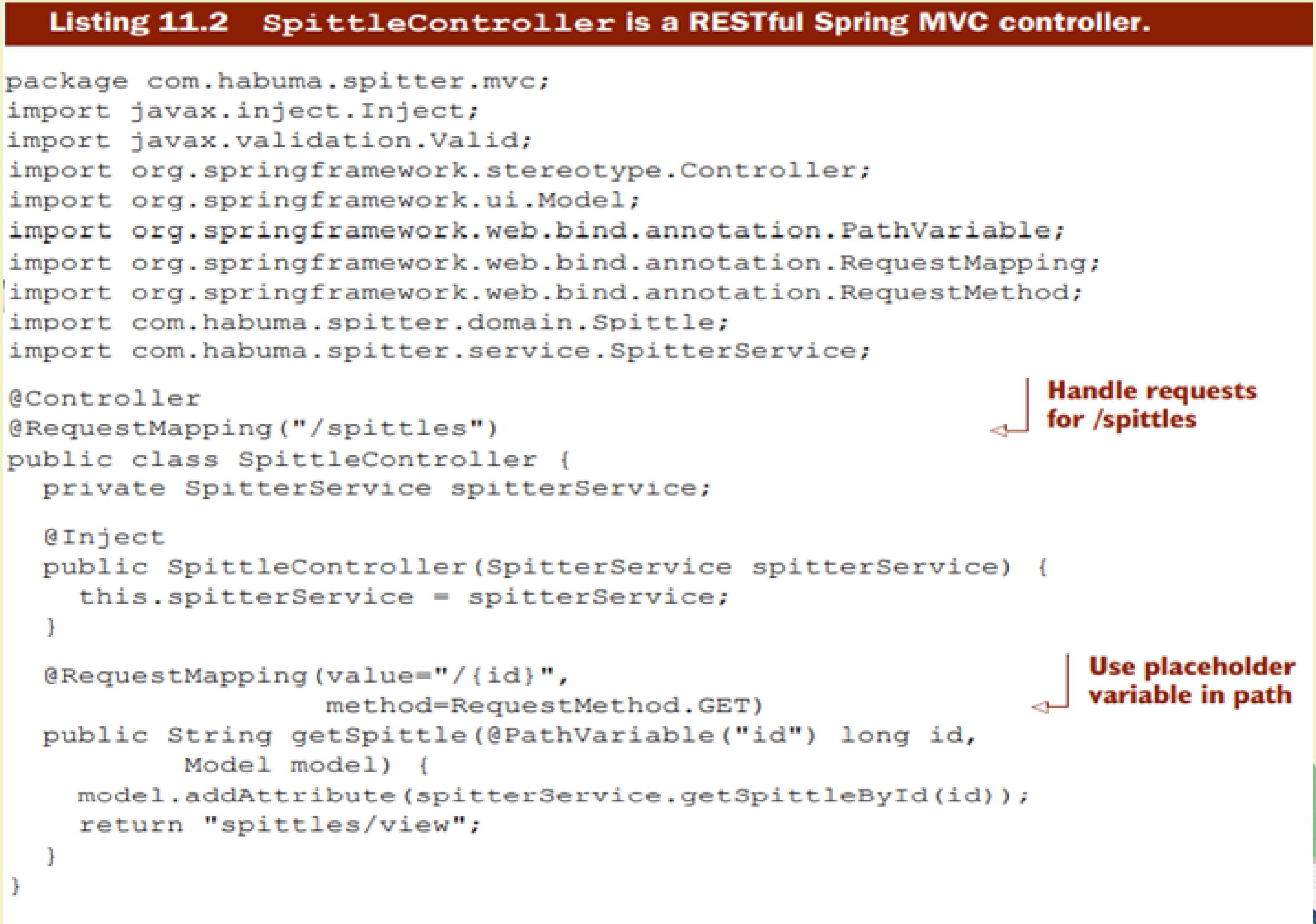
http://localhost:8080 identifies a domain and port. Although our application won’t associate a resource with this URL.

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.

B. Writing resource-oriented controllers



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

B. Writing resource-oriented controllers

REST verbs

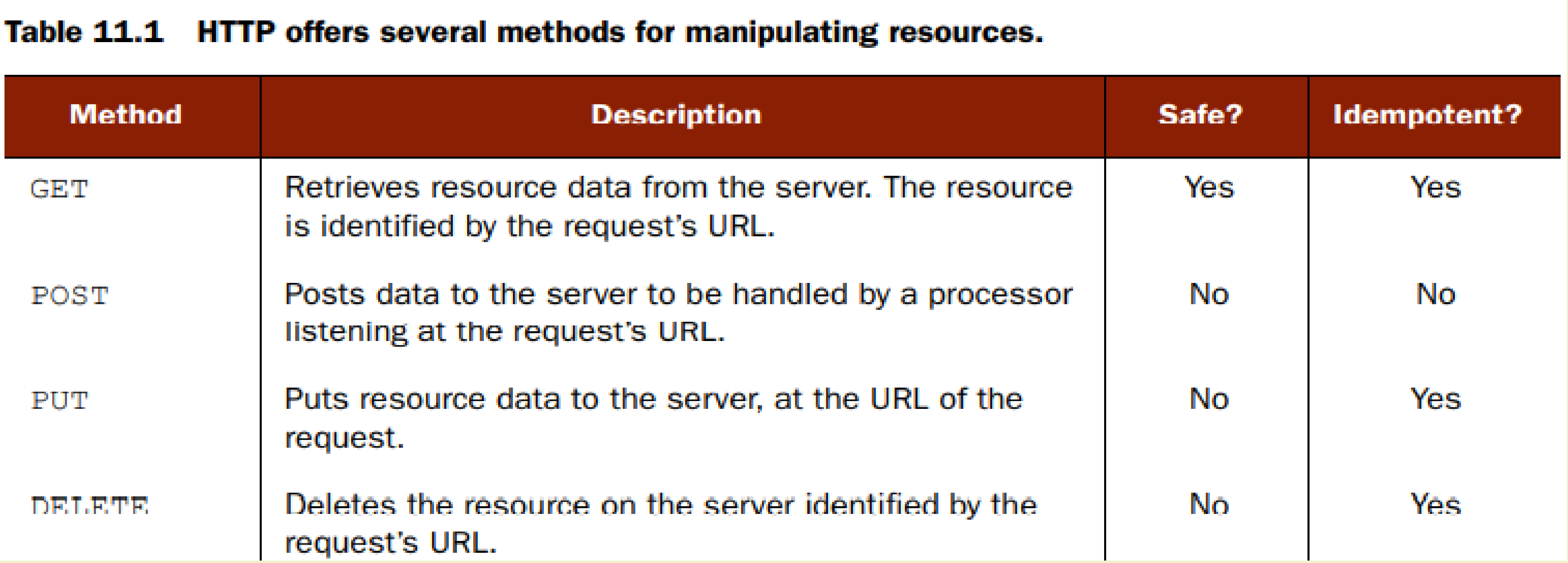


Table 11.1 HTTP offers several methods for manipulating resources.

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

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.**

C. Representing Resources

Negotiating resource representation

HTTP message converters

C. Representing resources

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.**

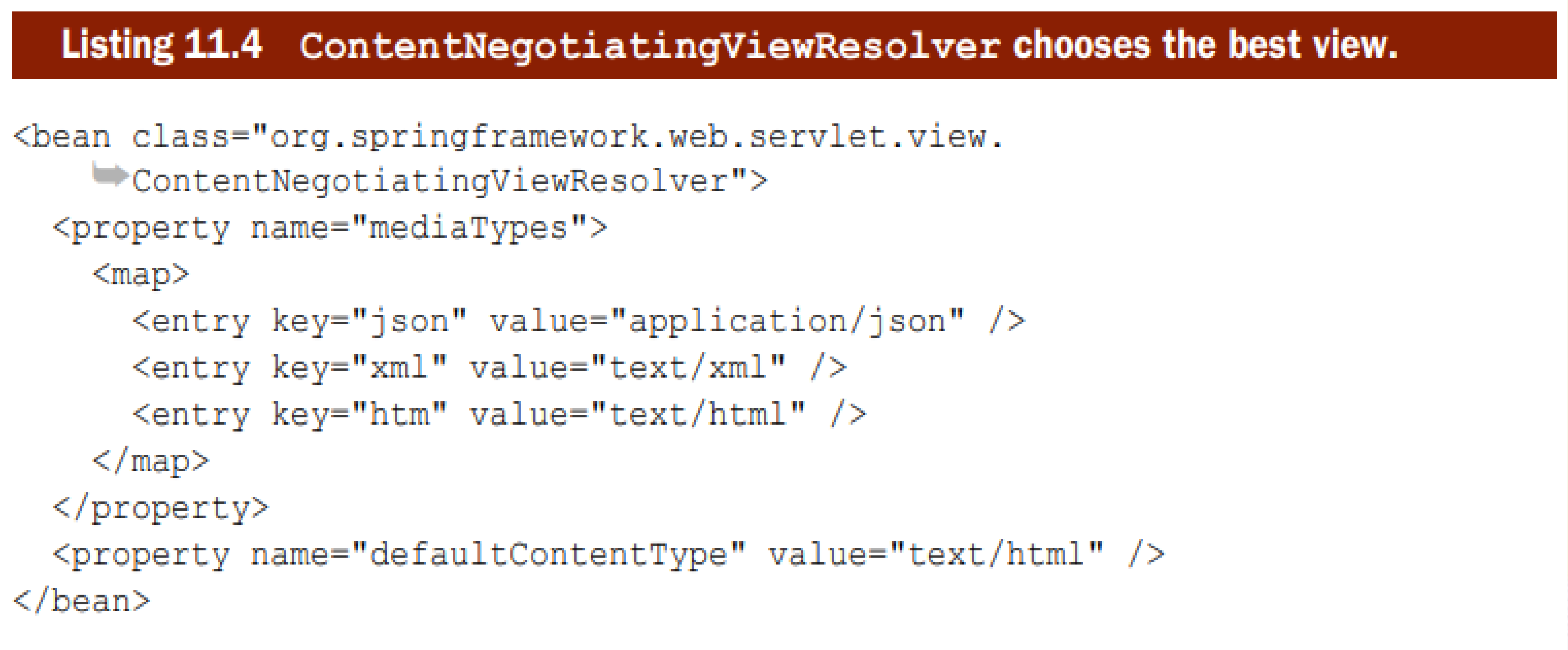
Spring provides two ways to **transform a resource’s Java representation into the representation that will be shipped to the client:**

- Negotiated view-based rendering
- HTTP message converters

C. Representing resources

Negotiatied view base rendering

Spring's ContentNegotiatingViewResolver is a special view resolver that takes **the content type** that the client wants into consideration.

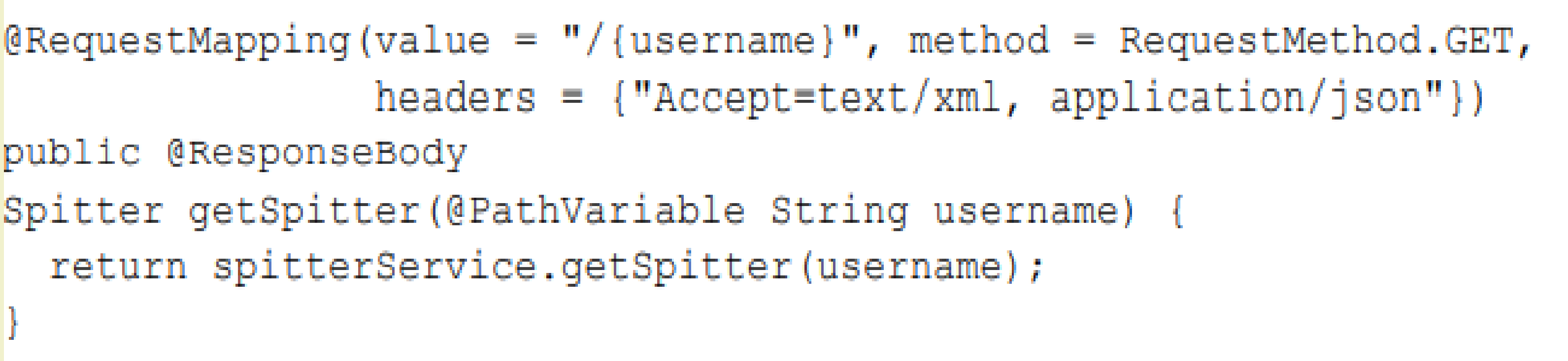


Listing 11.4 ContentNegotiatingViewResolver **chooses the best view.**

```
<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>
```

C. Representing resources

HTTP message converters



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

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.

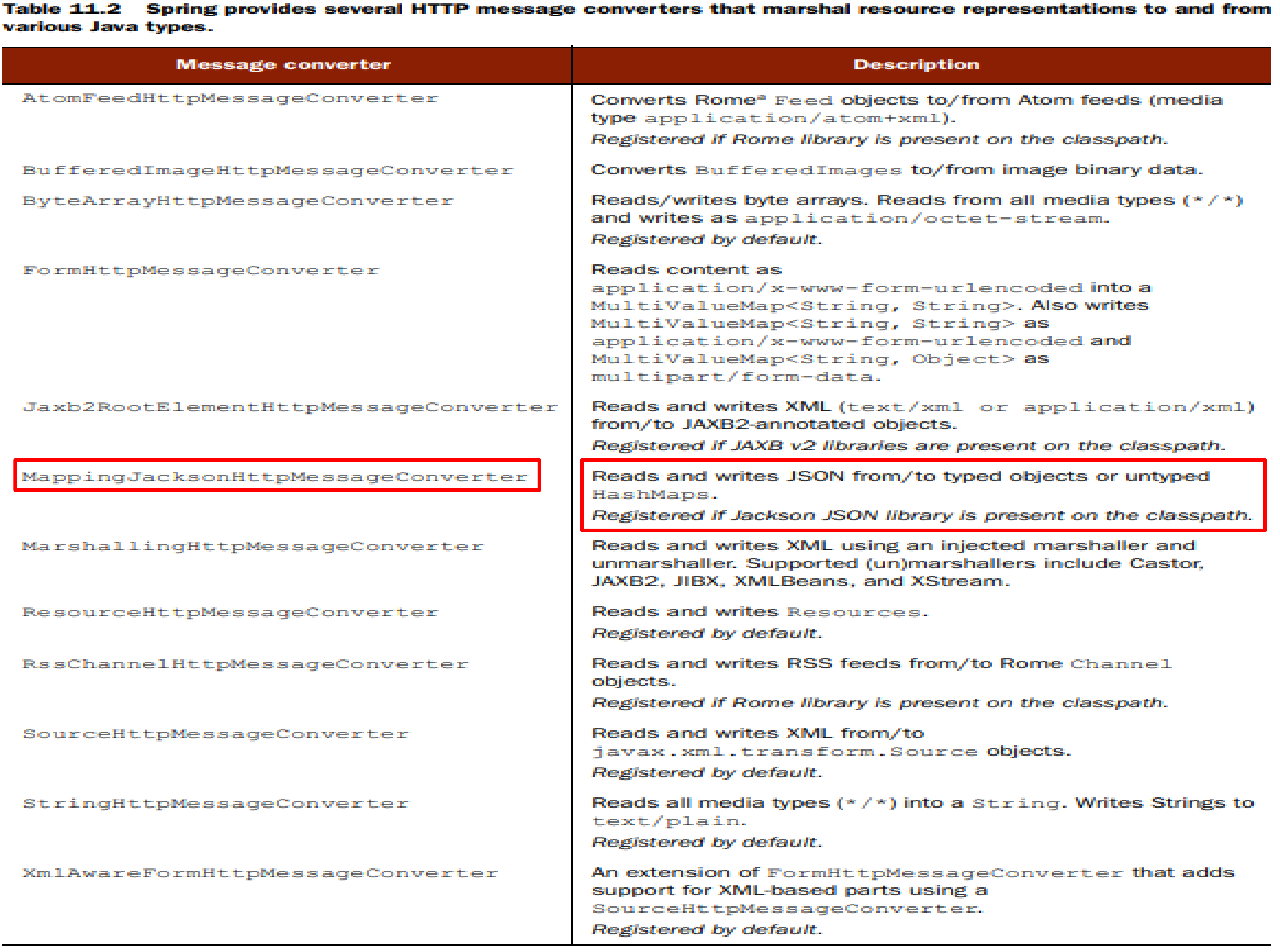
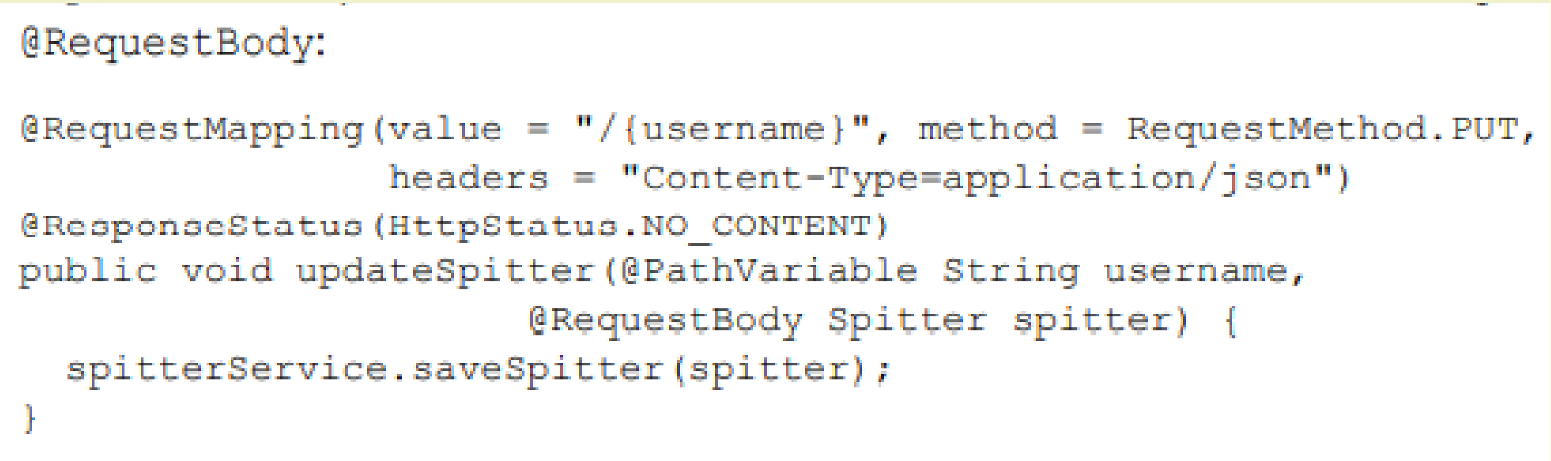


Table 11.2 Spring provides several HTTP message converters that marshal resource representations to and from various Java types.

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.*

C. Representing resources

Receiving resources from request body



```
@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);
}
```

The request’s **Content-Type** header must be set to **application/json**

The Jackson JSON library must be available on the application’s classpath.

D. Writing Rest Clients

Rest templates operations

GET – PUT – DELETE – POST resources

Exchanging resources

D. Writing Rest Clients

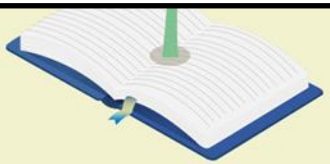
RestTemplate's operations

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.

D. Writing Rest Clients

RestTemplate's operations

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, 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.



D. Writing Rest Clients

GETting resources - Defination

```
<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;
```

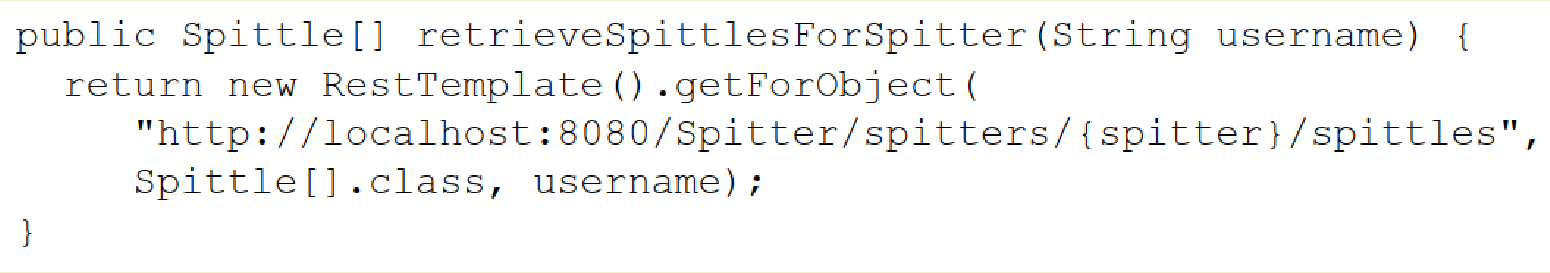
getForObject: Returns an object of the type requested

```
<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;
```

getForEntity: Returns that object along with extra information about the response.

D. Writing Rest Clients

GETting resources – retrieve resources



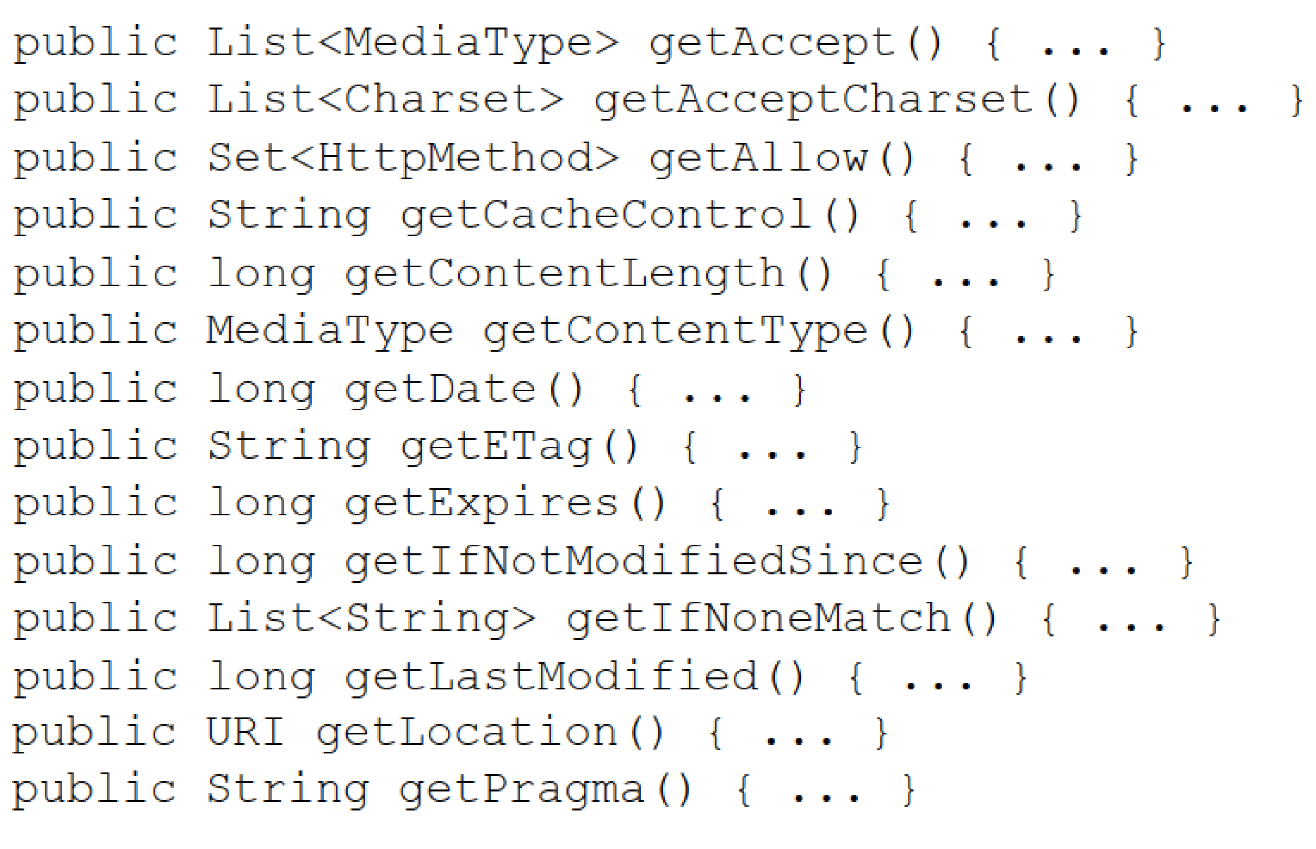
```
public Spittle[] retrieveSpittlesForSpitter(String username) {
  return new RestTemplate().getForObject(
     "http://localhost:8080/Spitter/spitters/{spitter}/spittles",
     Spittle[].class, username);
}
```

Cannot put username parameter into a new map and replace username variable by the map!!!

D. Writing Rest Clients

GETting resources – Extracting response metadata (getForEntity)

Headers

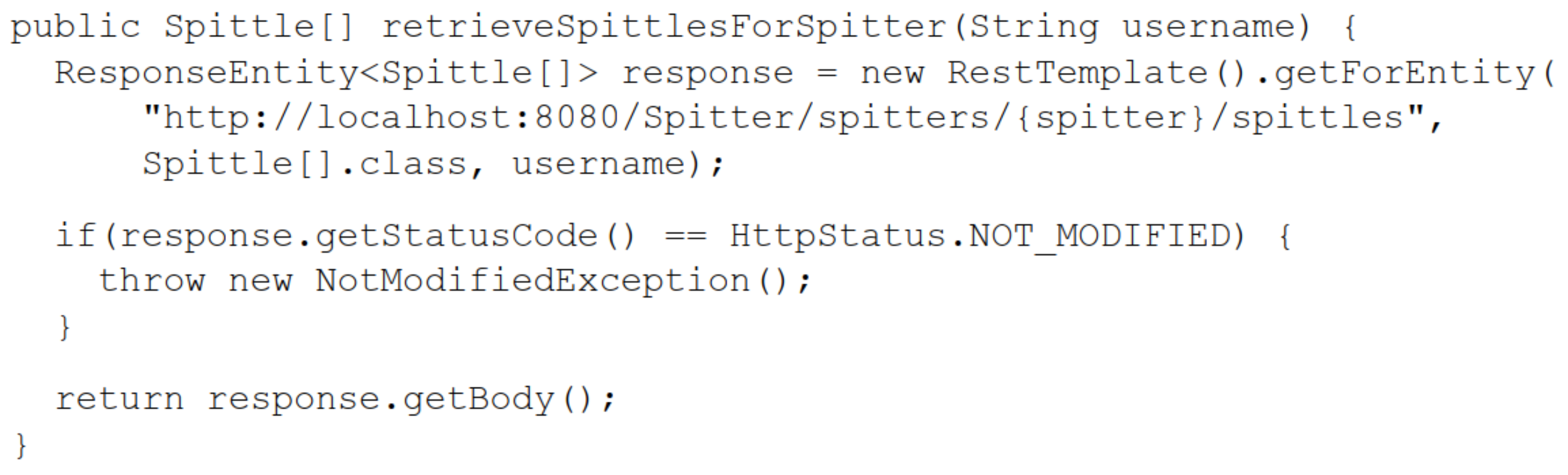


```
public List<MediaType> getAccept() { ... }
public List<Charset> getAcceptCharset() { ... }
public Set<HttpMethod> getAllow() { ... }
public String getCacheControl() { ... }
public long getContentLength() { ... }
public MediaType getContentType() { ... }
public long getDate() { ... }
public String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

D. Writing Rest Clients

GETting resources – Extracting response metadata (getForEntity)

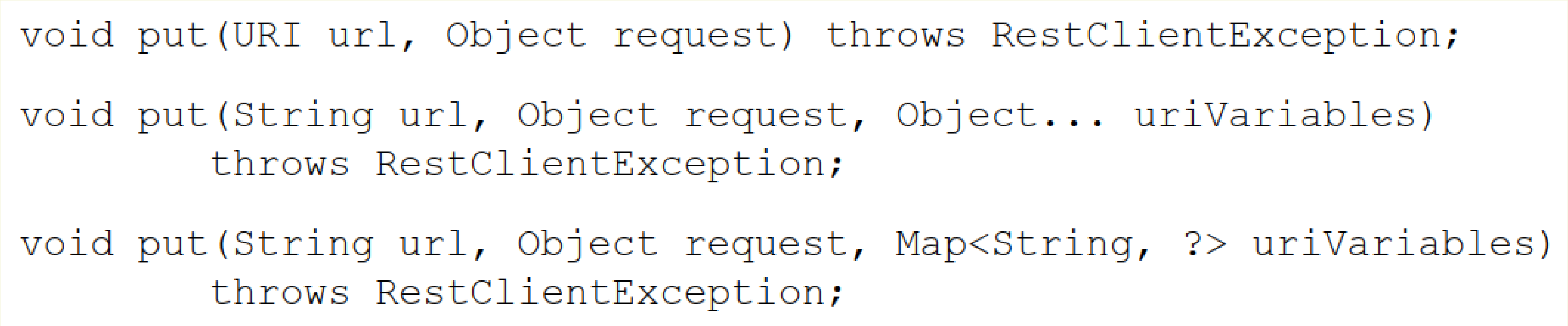
Response status



```
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();
}
```

D. Writing Rest Clients

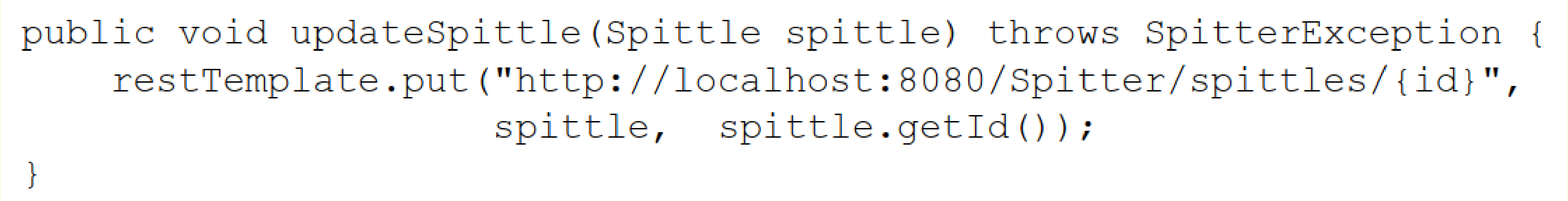
PUTting resources - Defination



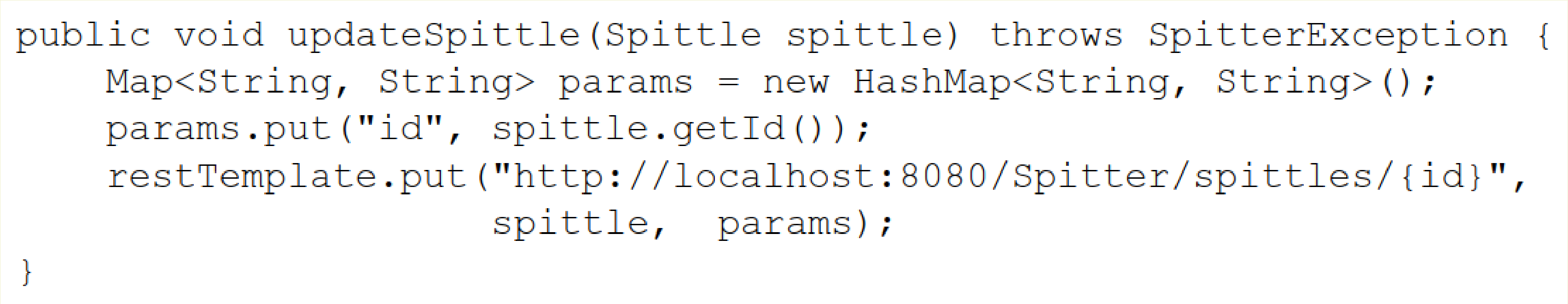
```
void put(URI url, Object request) throws RestClientException;

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

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

D. Writing Rest Clients

PUTting resources – How to use



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

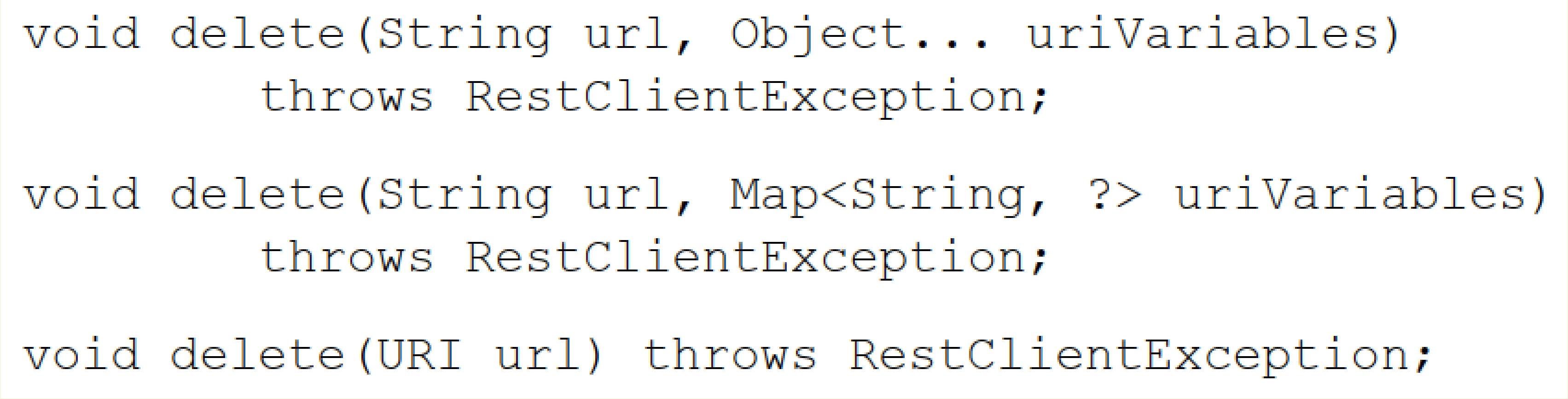


```
public void updateSpittle(Spittle spittle) throws SpitterException {
    Map<String, String> params = new HashMap<String, String>();
    params.put("id", spittle.getId());
    restTemplate.put("http://localhost:8080/Spitter/spittles/{id}",
                     spittle,  params);
}
```

Cannot put username parameter into a new map and replace username variable by the map!!!

D. Writing Rest Clients

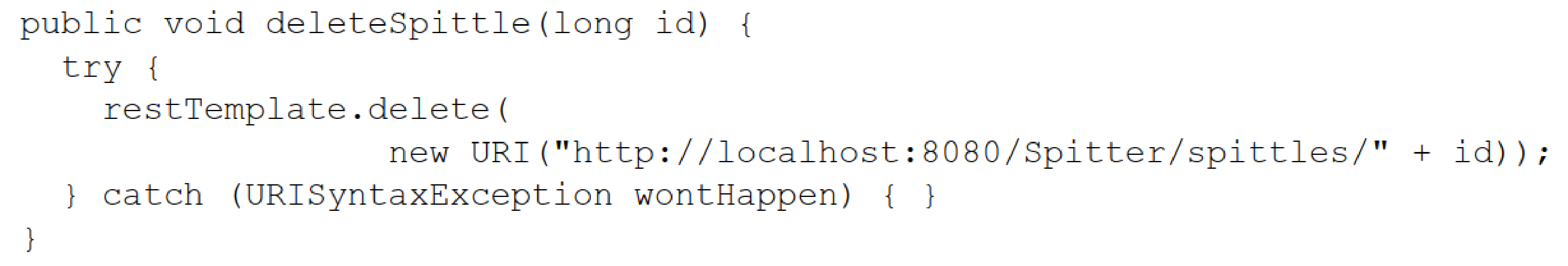
DELETE-ing resources - Defination



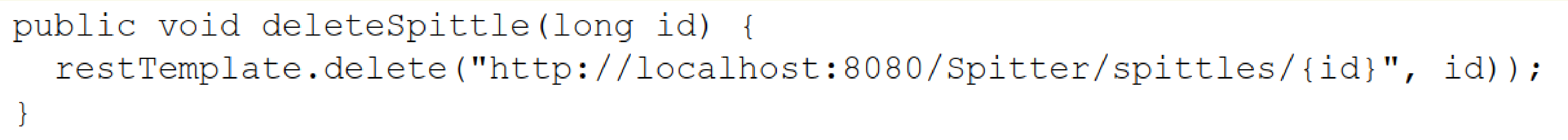
```
void delete(String url, Object... uriVariables)
        throws RestClientException;

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

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

D. Writing Rest Clients

DELETE-ing resources – How to use



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



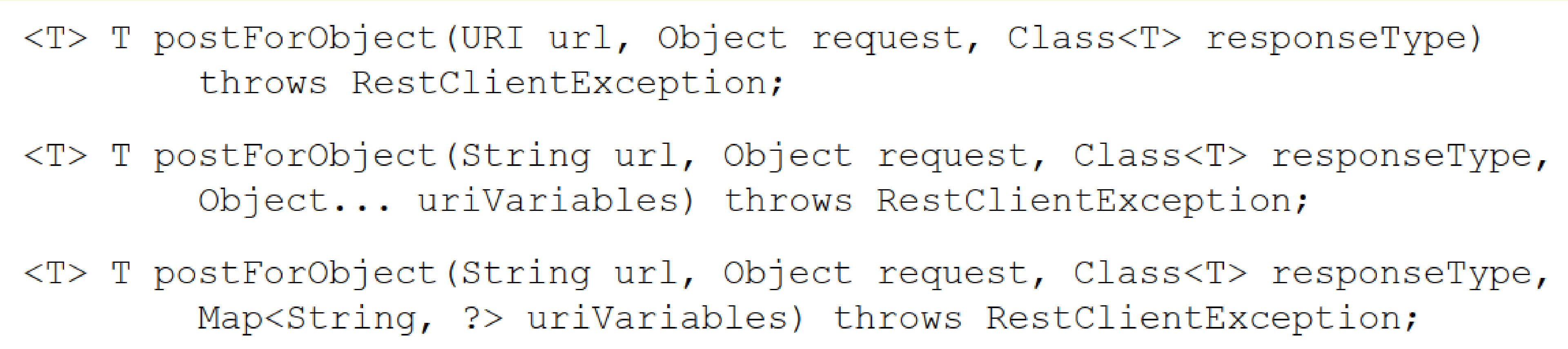
```
public void deleteSpittle(long id) {
  restTemplate.delete("http://localhost:8080/Spitter/spittles/{id}", id));
}
```

Cannot put username parameter into a new map and replace username variable by the map!!!

D. Writing Rest Clients

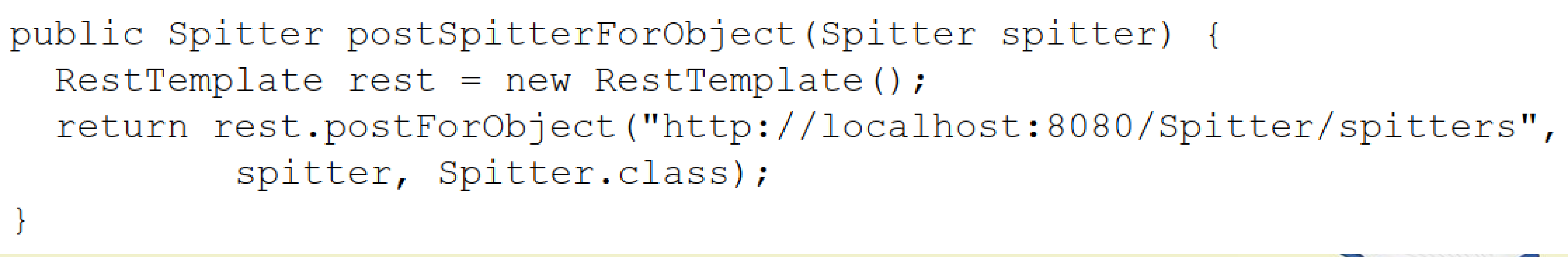
POSTing resource data – Defination & How to use (postForObject)

Defination



```
<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;
```

How to use



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

D. Writing Rest Clients

POSTing resource data – Defination & How to use (postForEntity)

Defination

```
<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;
```

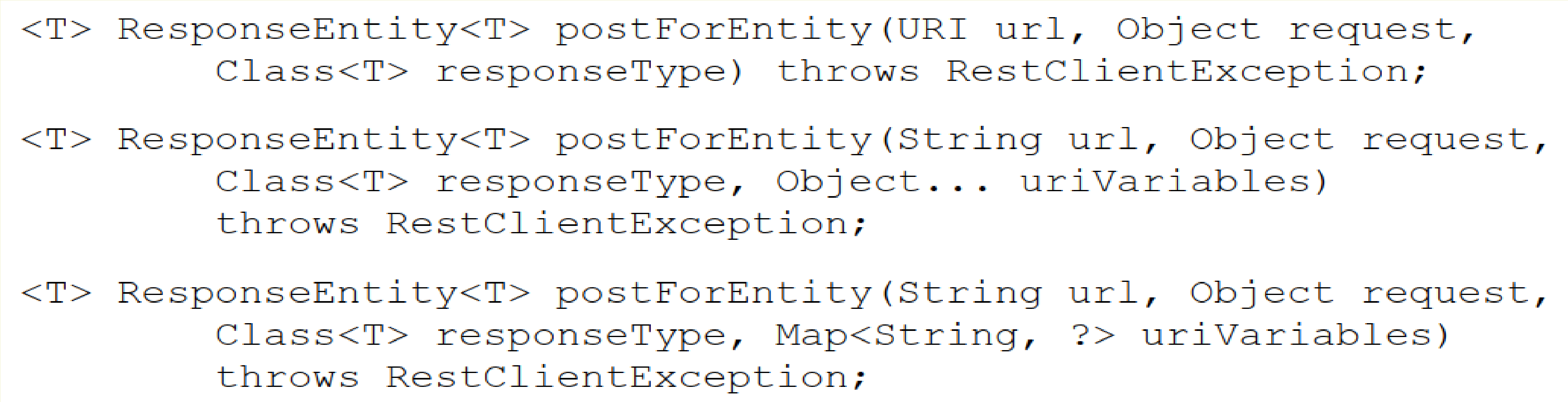
How to use

```
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();
```

D. Writing Rest Clients

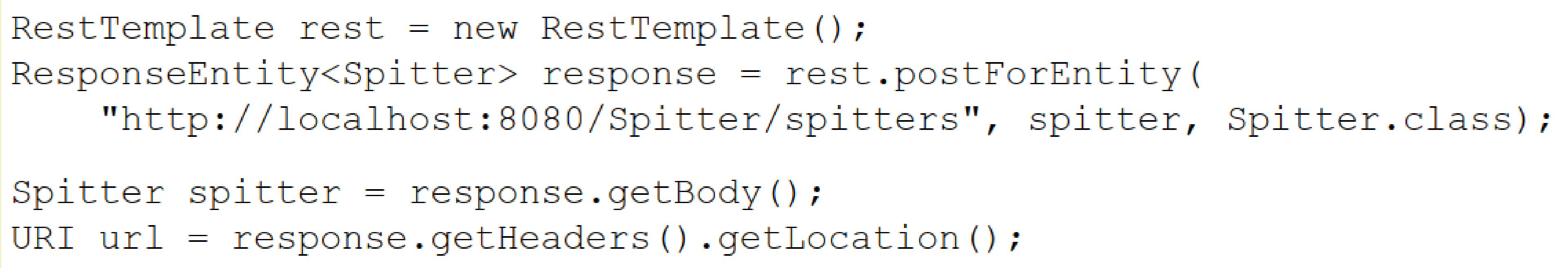
POSTing resource data – Defination & How to use (postForEntity)

Defination



```
<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;
```

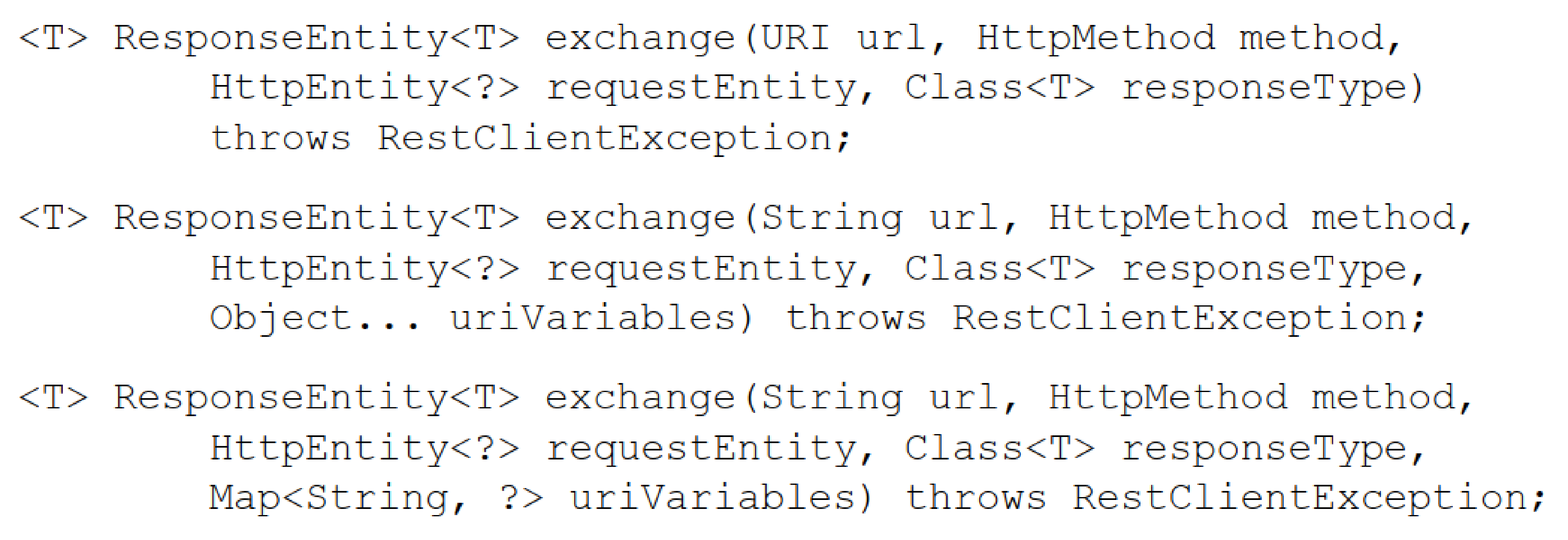
How to use



```
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();
```

D. Writing Rest Clients

Exchanging resources – Defination

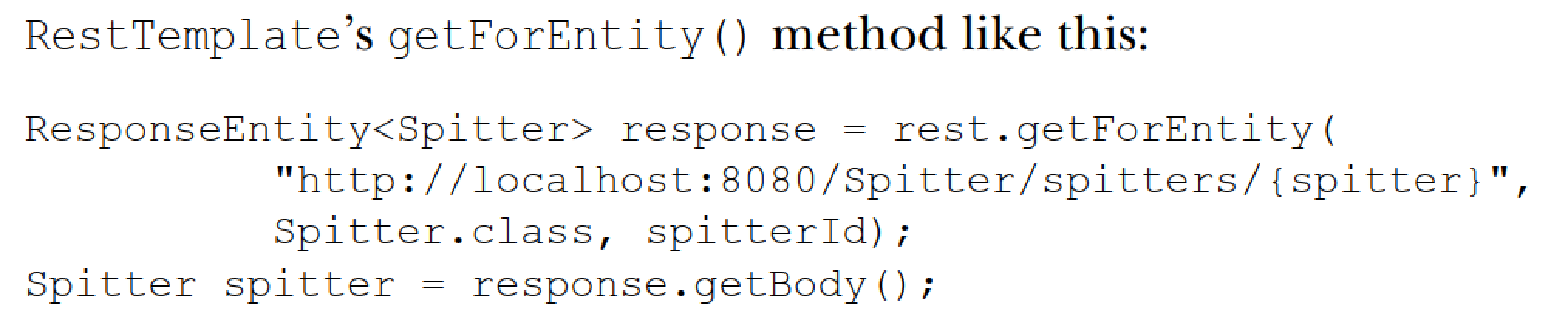


```
<T> ResponseEntity<T> exchange(URI url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType)
        throws RestClientException;

<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;
```

D. Writing Rest Clients

Exchanging resources – How to use

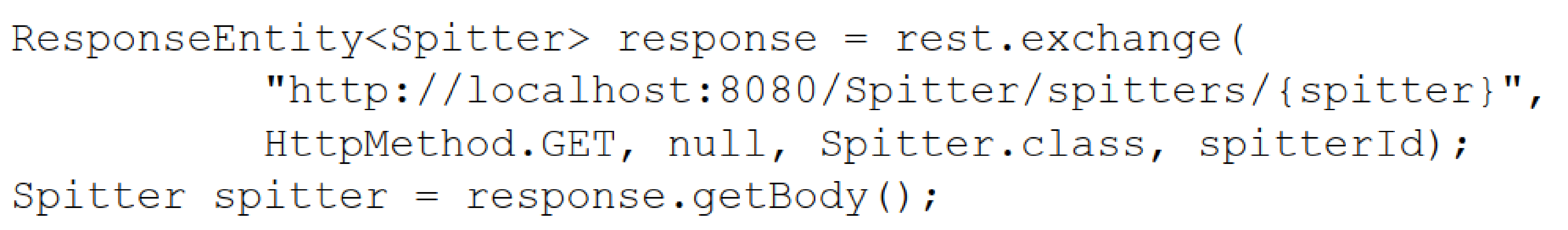
Get



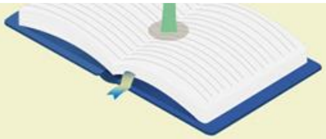
RestTemplate's getForEntity() method like this:

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

Exchange

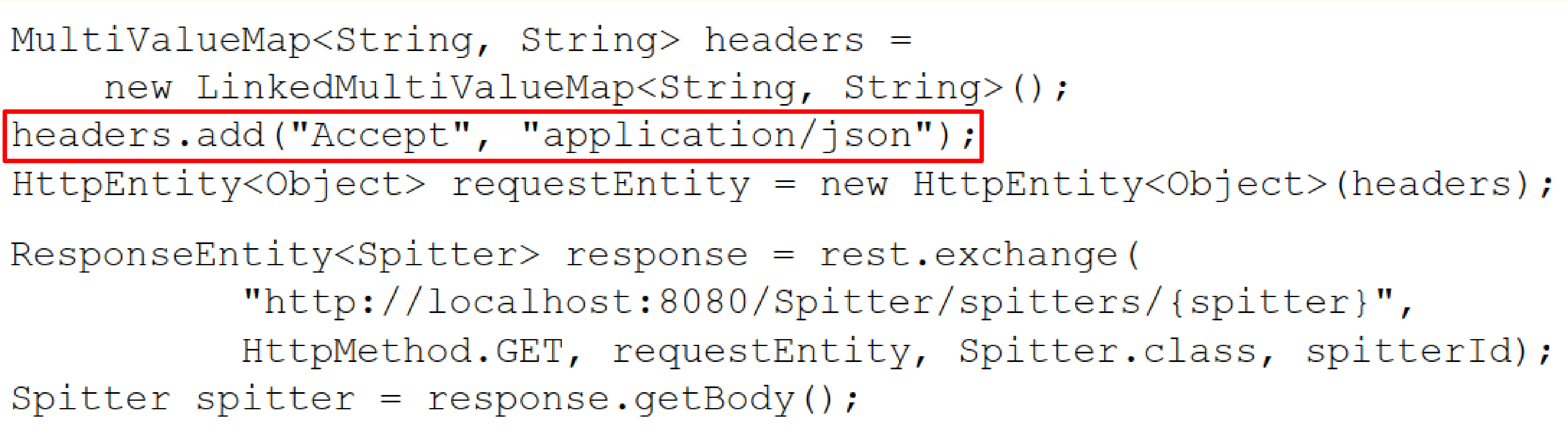


```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/Spitter/spitters/{spitter}",
          HttpMethod.GET, null, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```



D. Writing Rest Clients

Exchanging resources – Difference with other methods



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

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

The request will be sent with specific Accept in above

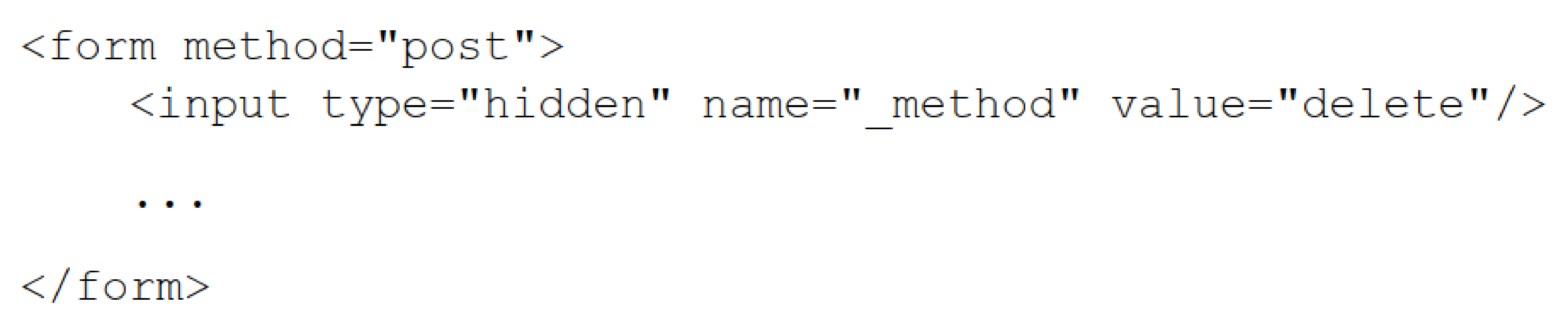
E. Submitting RESTful forms

Rendering hidden method fields in JSP

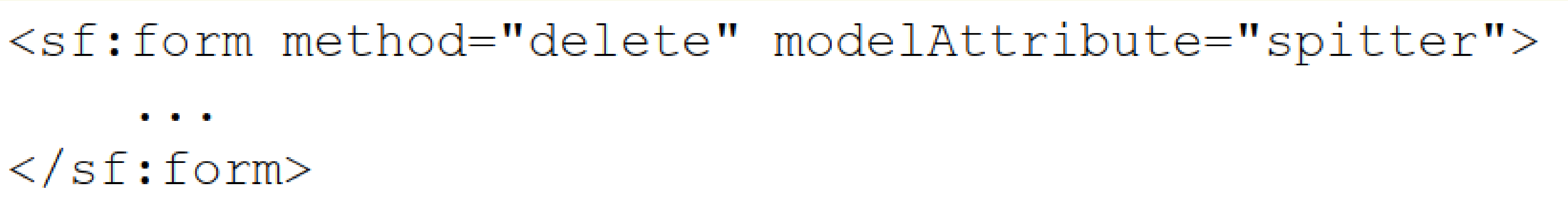
Unmasking the real request

E. Submitting RESTful forms

Rendering hidden method fields in JSP



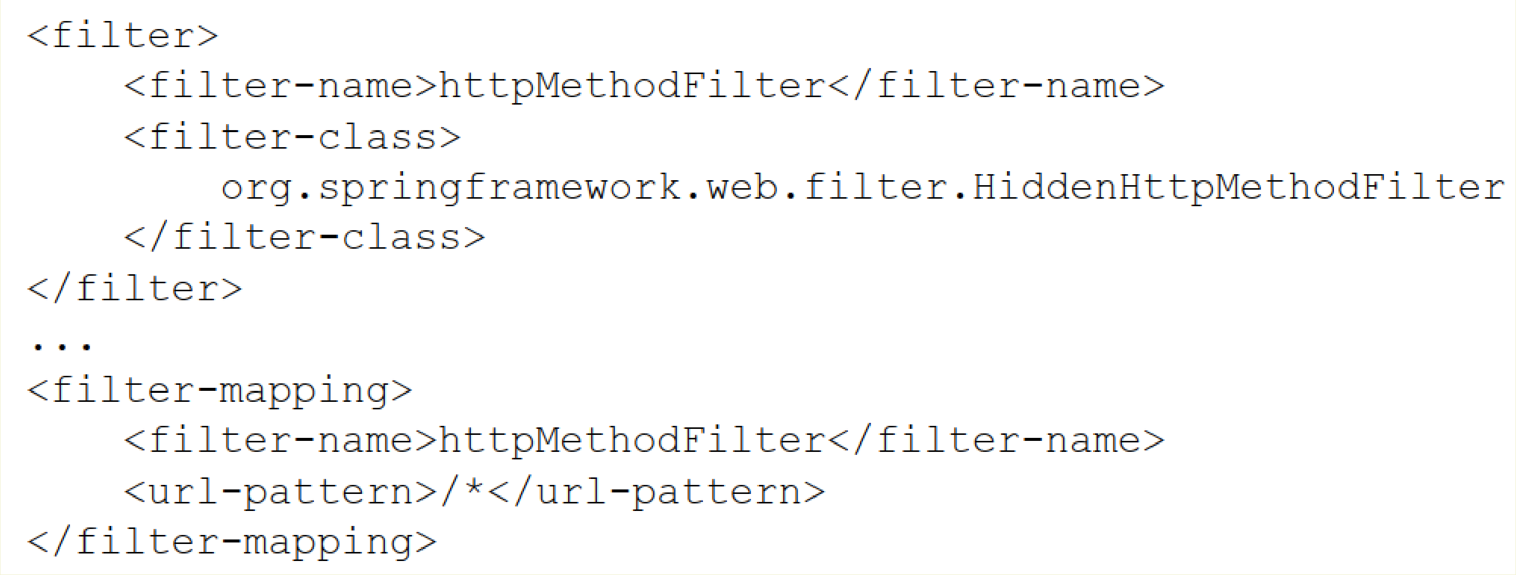
```
<form method="post">
    <input type="hidden" name="_method" value="delete"/>

    ...

</form>
```



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

E. Submitting RESTful forms

Unmasking the real request



```
<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>
```

E. Submitting RESTful forms

Unmasking the real request

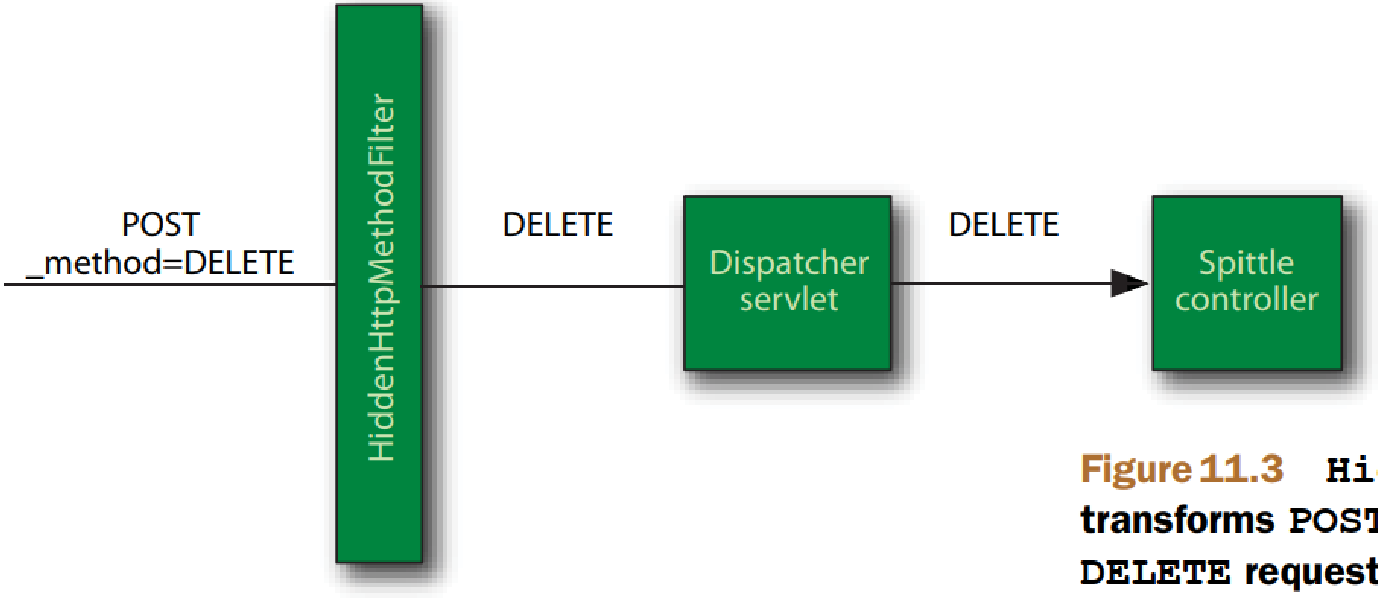


Figure 11.3 **`HiddenHttpMethodFilter` transforms `POST`-masquerading `PUT` and `DELETE` requests into their true form.**

F. Demo

Restful Controllers
Restful Forms
New annotations in Spring 5.x.x