REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Implementing and Publishing RESTful Web Services

1.1 The Java Options

Java offers more than way to implement and to publish RESTful web services. This chapter explores some options. On the publishing side, the choices range from very basic, development-oriented tools such as the Grizzly RESTful container and the core `Endpoint` class; through lightweight, Java-centric web servers such as Tomcat and Jetty; and up to full-blown Java application servers (JAS) such as Glassfish, JBoss, and WebSphere. There are also various APIs for implementing RESTful services, both standard and third-party. Here is a short list:

- The `HttpServlet` and JSP APIs, introduced briefly in Chapter 1 and examined more thoroughly in this chapter.
- The JAX-RS (Java API for XML-Restful Services) API.
- The JAX-WS (Java API for XML-Web Services) API, in particular the `WebServiceProvider` interface.
- The third-party *restlet* API.

For the most part, the API used to implement the web service does not constrain how this service can be published. The exception is the servlet API, as servlets need to be deployed in a servlet container such as Tomcat's Catalina or Jetty. (Jetty is the name of both the web server and its servlet container.) This chapter uses Tomcat and Jetty to publish servlet-based services. There are shotcuts for publishing JAX-RS and JAX-WS services but these, too, can be published with Tomcat or Jetty. Services based on the restlet API are meant to be published with a servlet container.

The decision of how to publish a service depends on many factors, of course. For example, if service deployment requires wire-level security in the form of HTTPS together with user authentication/authorization, then a web server such as Tomcat is the obvious starting point. If the published web services are to interact with EJBs, which are deployed in an EJB container, then a souped-up web server such as TomEE (Tomcat with EE support) or a full JAS is the better choice. In development, simpler options such as Grizzly or `Endpoint` are attractive. This chapter introduces various options for publication; and Chapter 6 covers web services deployed in a JAS.

1.2 A RESTful Service as an `HttPServlet`

Chapter 1 has a sample RESTful service implemented as a JSP script and a pair of back-end classes, `Prediction` and `Predictions`. The JSP-based service supported only GET requests. This section revises the example to provide an `HttpServlet` implementation with support for the four CRUD operations:

- A new `Prediction` can be created with a POST request whose body has two key/value pairs: a `who` key whose value is the name of the predictor and a `what` key whose value is the prediction.
- The `Prediction` objects can be read one at a time or all together with a GET request.

  If the GET request has a query string with an `id` key, then the corresponding `Prediction`, if any, is returned. If the GET request has no query string, then the list of `Predictions` is returned. On any GET request, the client can indicate a preference for JSON rather than the default XML format.
- A specified `Prediction` can be updated with a PUT request that provides the `Prediction` identifier and either a new `who` or a new `what`. (The reason for this restriction, explained in detail later, is that Tomcat has trouble with PUT requests.)
- A specified `Prediction` can be deleted.

The structure of a servlet-based service differs from that of the earlier JSP-based service. The obvious change is that an `HttpServlet` replaces the JSP script. There are also changes in the details of the `Prediction` and `Predictions` classes, which still provide back-end support. The details follow.

Deploying under Jetty instead of Tomcat

The Jetty web server is available at *http://jetty.codehaus.org* as a ZIP file. Assume that *JETTY_HOME* is the install directory. The subdirectories of *JETTY_HOME* are similar to those of *TOMCAT_HOME*. For example, *JETTY_HOME* has a *webapps* subdirectory into which WAR files are deployed; a *logs* subdirectory; a *lib* subdirectory with various JAR files, including a versioned counterpart of Tomcat's *servlet-api.jar*; and others. Jetty ships with an executable JAR file *start.jar*; hence, Jetty can be started at the command line with the command

```
% java -jar start.jar
```

A standard WAR file deployable under Tomcat is deployable under Jetty and vice-versa. (A *standard* WAR file contains only the regular deployment descriptor *web.xml* and not any product-specific configuration files.) The Jetty web server, like Tomcat, listens by default on port 8080. Jetty is a first-rate web server that has a lighter feel than does Tomcat; and Jetty's simplicity makes embedding this web server in other systems relatively straightforward. In the end, it is hard to make a bad choice between Tomcat and Jetty.

1.2.1 Implementation Details

There are small but important changes to the `Prediction` class (see Example 1), which now includes an `id` property, an an auto-incremented integer that the service sets when a new `Prediction` object is constructed. The `id` property is used to sort the `Prediction` objects, which explains why the `Prediction` class implements the `Comparable` interface used in sorting:

```
public class Prediction implements Serializable, Comparable<Prediction> {
```

Implementing the `Comparable` interface requires that the `compareTo` method be defined:

```
public int compareTo(Prediction other) {
    return this.id - other.id;
}
```

The `compareTo` method uses the comparison semantics of the age-old C function `qsort`. For illustration, suppose that `this.id` in the code above is 7 and `other.id` is 12, where `this` is the current object and `other` is another `Prediction` object against which the current `Prediction` object is to be compared. The difference *7 - 12* is the negative integer -5, which signals that the current `Prediction` *precedes* the other `Prediction` because 7 precedes 12. In general,

- a returned negative integer signals that the current object *precedes* the other object
- a returned positive integer signals that the current object *succeeds* the other object
- a returned zero signals that the two objects are to be treated as equals with respect to sorting

The implementation of the `compareTo` method means the sort is to be in ascending order. Were the `return` statement changed to

```
return other.id - this.id;
```

the sort would be in descending order. The `Prediction` objects are sorted for ease of confirming that the CRUD operations work correctly. For example, if a new `Prediction` object is created with the appropriate POST request, then the newly created `Prediction` occurs at the *end* of the `Prediction` list. In similar fashion, it is straightword to confirm that the other destructive CRUD opertions—PUT (update) and DELETE—work as intended by inspecing the resulting sorted list of `Prediction` objects.

Example 1.1 The back-end `Prediction` class

```
package cliches2;

import java.io.Serializable;

// An array of Predictions is to be serialized
// into an XML or JSON document, which is returned to
// the consumer on a request.
public class Prediction implements Serializable, Comparable<Prediction> {
    private String who;   // person
    private String what;  // his/her prediction
    private int    id;    // identifier used as lookup-key

    public Prediction() { }

    public void setWho(String who) {
        this.who = who;
    }
    public String getWho() {
        return this.who;
    }

    public void setWhat(String what) {
        this.what = what;
    }
    public String getWhat() {
        return this.what;
    }

    public void setId(int id) {
        this.id = id;
    }
    public int getId() {
        return this.id;
    }

    // implementation of Comparable interface
    public int compareTo(Prediction other) {
        return this.id - other.id;
    }
}
```

A `Prediction` is still `Serializable` so that a list of these can be serialized into XML. An added feature is that this list also can be formatted in JSON if the client so requests.

The utility class `Predictions` has changed as well (see Example 2). As explained in the sidebar about thread synchronization and servlets, the `Map` of the earlier JSP implementation gives way to a `ConcurrentMap` so that the code can avoid explicit locks in the form of `synchronized` blocks. The `Predictions` class now has an `addPrediction` method

```
public int addPrediction(Prediction p) {
   p.setId(mapKey);
   predictions.put(String.valueOf(mapKey), p);
   return mapKey++;
}
```

to support POST requests. The servlet's `doPost` method creates a new `Prediction`, sets the `who` and `what` properties with data from the POST message's body, and then invokes `addPrediction` to add the newly constructed `Prediction` to the map whose object reference is `predictions`. The `mapKey` is an integer that gets incremented with each new `Prediction`; this integer becomes the `id` newly constructed `Prediction`.

Example 1.2 The back-end `Predictions` class

```
package cliches2;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.ByteArrayOutputStream;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ConcurrentHashMap;
import java.util.Collections;
import java.beans.XMLEncoder; // simple and effective
import javax.servlet.ServletContext;

public class Predictions {
    private ConcurrentMap<String, Prediction> predictions;
    private ServletContext sctx;
    private static int mapKey = 1;

    public Predictions() {
        predictions = new ConcurrentHashMap<String, Prediction>();
    }

    //** properties

    // The ServletContext is required to read the data from
    // a text file packaged inside the WAR file
    public void setServletContext(ServletContext sctx) {
        this.sctx = sctx;
    }
    public ServletContext getServletContext() { return this.sctx; }

    public void setMap(ConcurrentMap<String, Prediction> predictions) {
        // no-op for now
    }
    public ConcurrentMap<String, Prediction> getMap() {
        // Has the ServletContext been set?
        if (getServletContext() == null) return null;

        // Have the data been read already?
        if (predictions.size() < 1) populate();

        return this.predictions;
    }

    public String toXML(Object obj) {
        String xml = null;

        try {
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            XMLEncoder encoder = new XMLEncoder(out);
            encoder.writeObject(obj); // serialize to XML
            encoder.close();
            xml = out.toString(); // stringify
        }
        catch(Exception e) { }
        return xml;
    }

    public int addPrediction(Prediction p) {
        p.setId(mapKey);
        predictions.put(String.valueOf(mapKey), p);
        return mapKey++;
    }

    //** utility
    private void populate() {
        String filename = "/WEB-INF/data/predictions.db";
        InputStream in = sctx.getResourceAsStream(filename);
```

The remaining `Predictions` code is slightly changed, if at all, from the earlier version. For example, the `populate` method is modified slightly to give each newly constructed `Prediction` an `id`; but the method's main job is still to read data from the text file encapsulated in the WAR, data that represent the `who` and the `what` of each `Prediction`.

The `PredictionServlet` (see Example 3) replaces the JSP script and differs from this script in supporting all of the CRUD operations. The servlet offers new functionality by allowing the client to request JSON format for the response of any GET request. Further, the JSP script interpreted GET to mean *read all* but the servlet allows the client to request a read for a single `Prediction` or for them all.

Example 1.3 The `PredictionsServlet` with full support for the CRUD operations

```
package cliches2;

import java.util.concurrent.ConcurrentMap;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.http.HTTPException;
import java.util.Arrays;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.OutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.beans.XMLEncoder;
import org.json.JSONObject;
import org.json.XML;

public class PredictionsServlet extends HttpServlet {
    private Predictions predictions; // back-end bean

    // Executed when servlet is first loaded into container.
    // Create a Predictions object and set its servletContext
    // property so that the object can do I/O.
    public void init() {
        predictions = new Predictions();
        predictions.setServletContext(this.getServletContext());
    }

    // GET /cliches2
    // GET /cliches2?id=1
    // If the HTTP Accept header is set to application/json (or an equivalent
    // such as text/x-json), the response is JSON and XML otherwise.
    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        String key = request.getParameter("id");

        // Check user preference for XML or JSON by inspecting
        // the HTTP headers for the Accept key.
        String accept = request.getHeader("accept");
        boolean json = accept.contains("json") ? true : false;

        // If no query string, assume client wants the full list.
        if (key == null) {
            ConcurrentMap<String, Prediction> map = predictions.getMap();

            // Sort the map's values for readability.
            Object[] list = map.values().toArray();
            Arrays.sort(list);

            String xml = predictions.toXML(list);
            sendResponse(response, xml, json);
        }
        // Otherwise, return the specified Prediction.
        else {
            Prediction pred = predictions.getMap().get(key);

            if (pred == null) { // no such Prediction
                String msg = key + " does not map to a prediction.\n";
                sendResponse(response, predictions.toXML(msg), false);
            }
            else { // requested Prediction found
                sendResponse(response, predictions.toXML(pred), json);
            }
        }
    }

    // POST /cliches2
    // HTTP body should contain two keys, one for the predictor ("who") and
```

The challenge of thread synchronization in servlets

A web server such as Tomcat can instantiate arbitrarily many instances of a servlet, although the number is typically small, for example, 1 through 4. Although the *web.xml* deployment file can recommend how many instances of a servlet should be loaded into the servlet container at startup, the web server itself makes the decision. For example, Tomcat by default loads one instance of a servlet to begin but may load more instances thereafter if simultaneous requests for the servlet are sufficient in number. Whatever the number of servlet instances, the number of client requests per instance is typically greater—and significantly so. For example, one servlet instance might handle tens of simultaneous requests. For reasons of efficiency, a web server such as Tomcat keeps the number of servlet instances as small as possible in supporting reasonable response time per request. A high-volume web server might have to handle hundreds or even thousands of requests per second, distributed across many servlets; and one servlet per request is out of the question in this real-world scenario. The upshot is that Java-based web servers rely on multithreading to handle simultaneous requests. The model is sometimes described as *one thread per request*.

For reasons of performance, a web server such as Tomcat creates a thead pool at startup; as requests come in, each is dispatched to a thread from the pool, which then handles the request and returns to the pool afterwards. The pooling amortizes the relatively expensive cost of thread creation across the web server's uptime. There are, of course, various ways to measure how well a web server is performing. One critical measure is response time. For example, a web site might require that the average response time per request be no more than, say, 10 *ms*.

The one-thread-per-request model poses challenges for the servlet/JSP programmer, in particular the challenge of thread coordination or *synchronization*. For example, if there are a dozen concurrent requests against the `PredictionsServlet` of the revised *sayings* service, then each of these requests is implemented as a thread that executes the appropriate *do*-method in the servlet. On a multi-core server (that is, a server with more than CPU), one thread could be executing the `doGet` method at exactly the same time as another is executing the `doPut` method: the result is a simultaneous *read* and *write* operation on the same resource. There are various other concurrency scenarios, any one of which requires proper thread synchronization; and the programmer rather than the servlet container must ensure that these scenarios maintain thread safety.

A servlet container such as Catalina or Jetty, in contrast to an EJB container, does *not* ensure thread safety; instead, the programmer is responsible for proper thread coordination. A servlet must be programmed so that, for example, two requests—each implemented as a separate thread—cannot simultaneously update the same resource such as a `Prediction`. In earlier Java versions, the mainstay of thread coordination was the `synchronized` block; later versions of Java have added higher-level constructs, many in the `java.util.concurrent` package, for managing thread-based concurrency. The revised *sayings* service uses a `ConcurrentMap` from this package to coordinate simultaneous thread access to the `Predictions`. A `ConcurrentMap` segments its entries; as a result, the map usually needs to lock only a portion of its total entries to enforce synchronization. In summary, the `ConcurrentMap` synchronizes request access to the `Predictions` collection and does so in an efficient manner.

Recall that each of the *do*-methods in an `HttpServlet` takes the same arguments: an `HttpServletRequest`, a map that contains the information encapsulated in the HTTP request, and an `HttpServletRespose`, which encapsulates an output stream for communicating back with the client. Here is the start of `doGet` method:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
```

The `HttpServletRequest` has a `getParameter` method that expects a string argument, a key into the request map, and returns either `null` if there is no such key or the key's value otherwise. The `getParameter` method is agnostic about whether the key/value pairs are in the body of, for example, a POST request or in the query string of, for example, a GET request. The method works the same in either case. There is also a `getParameters` method that returns the parameter collection as a whole.

In the case of `PredictionsServlet`, the `doGet` method needs to answer two questions about the incoming request:

- Does the body-less GET request include a key named `id` whose value identifies a particular `Prediction`?

  If the `id` is present, the `doGet` method uses this value as a lookup key in the `ConcurrentMap`, which holds references to all of the `Prediction` objects. If the lookup fails, then the `doGet` method method returns an XML message to that effect:

```
Prediction pred = predictions.getMap().get(key);
if (pred == null) { // no such Prediction
```

```
    String msg = key + " does not map to a prediction.\n";
    sendResponse(response, predictions.toXML(msg), false);
}
```

The last argument to the `sendResponse` method indicates whether JSON rather than XML should be sent back to the client. In this case, XML only is returned. If the `id` parameter is not present, the `doGet` method assumes that the client wants to read a list of all `Predictions` and returns this list in either JSON or XML format:

```
ConcurrentMap<String, Prediction> map = predictions.getMap();
// Sort the map's values for readability.
Object[] list = map.values().toArray();
Arrays.sort(list);
...
```

- Does the client prefer JSON over XML?

  In an HTTP request, the requester can express a preference for the MIME type of the returned representation. For example, the header element

```
Accept: text/html
```

expresses a preference for the MIME type/subtype `text/html`. Among the MIME combinations is `application/json` that, together with several variants, expresses a preference for JSON. The `doGet` method therefore checks the HTTP header element with `Accept` as its key

```
String accept = request.getHeader("accept");
boolean json = accept.contains("json") ? true : false;
```

to determine whether the client prefers JSON to XML. (Recall that HTTP is case insensitive; hence, the key could be `Accept`, `accept`, `ACCEPT`, and so on.) The `json` flag is the third argument to the `sendResponse` method:

```
private void sendResponse(HttpServletResponse res, String payload, boolean json) {
   try {
      if (json) {
        JSONObject jobt = XML.toJSONObject(payload);
        payload = jobt.toString(3); // 3 is indentation level for nice look
      }
      OutputStream out = res.getOutputStream();
      out.write(payload.getBytes());
      out.flush();
   }
   catch(Exception e) {
      throw new HTTPException(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
   }
}
```

The deployed WAR file *cliches2.war* includes a lightweight, third-party JSON library in the JAR file *json.jar*. (See URL) If the client prefers JSON over XML, then the response `payload` is converted to JSON. If anything goes awry in sending the response back to the client, the servlet throws an `HTTPException`, which in this case generates a resonse with HTTP status code 500 for *Internal Server Error*, a catch-all for server-side problems.

The `doPost` and `doPut` operations are similar in that the first creates a new `Prediction` using data in the body of a POST request and the second edits an existing `Prediction` from data in the body of a PUT request. The main difference is that a PUT request needs to include the `id` of the `Prediction` to be updated, whereas a POST request creates a new `Prediction` and then sets its `id` to an auto-incremented integer. In implementation, however, `doPost` and `doPut` differ significantly because neither Tomcat nor Jetty generates a usable parameter map, the `HttpServletRequest`, on a PUT request; both web servers do generate a usable parameter map on a POST request. As a result, the `doPut` implementation needs to get close to the HTTP metal.

To begin, here is the straightforward `doPost` implementation, without the comments:

```
public void doPost(HttpServletRequest request, HttpServletResponse response) {
   String who = request.getParameter("who");
   String what = request.getParameter("what");

   if (who == null || what == null)
      throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);

   Prediction p = new Prediction();
   p.setWho(who);
   p.setWhat(what);
   int id = predictions.addPrediction(p);

   String msg = "Prediction " + id + " created.\n";
   sendResponse(response, predictions.toXML(msg), false);
}
```

The two calls to the `getParameter` method extract the required data. A new `Prediction` is then constructed, its `who` and `what` properties are set, and a confirmation is generated for the client.

In the `doPut` method, the `getParameter` method does not work correctly because neither Tomcat nor Jetty builds an appropriate parameter map inside of the `HttpServletRequest` object. The workaround is to access directly the input stream encapsulated in the `HttpServletRequest`

```
BufferedReader br =
   new BufferedReader(new InputStreamReader(request.getInputStream()));
String data = br.readLine();
...
```

and then to the parse the `data` from this stream. The code, though not pretty, gets the job done. The point of interest is that the `HttpServletRequest` does provide access to the underlying input stream from which the PUT data can be extracted. Using the `getParameter` method is, of course, much easier.

The body of `doDelete` method has straightforward logic:

```
String key = request.getParameter("id");
if (key == null)
   throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);
try {
   predictions.getMap().remove(key);
   String msg = "Prediction " + key + " removed.\n";
   sendResponse(response, predictions.toXML(msg), false);
}
catch(Exception e) {
   throw new HTTPException(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
}
```

If `id` for the `Prediction` can be extracted from the parameter map, then the `Prediction` is effectively removed from the collection by removing the lookup key from the `ConcurrentMap`.

The `PredictionsServlet` also implements three other *do*-methods and all in the same way. Here, for example, is the implementation of `doHead`:

```
public void doHead(HttpServletRequest request, HttpServletResponse response) {
     throw new HTTPException(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
}
```

Throwing the `HTTPException` signals to the client that the underlying HTTP verb, in this case `HEAD`, is not supported. The numeric status code for `Method Not Allowed` is 405.

1.2.2 Sample Client Calls against the `sayings` Service

Example 4 is a list of *curl* calls against the service. These calls serve as a very preliminary test of the service. Two semi-colons introduce comments that explain the purpose of the *curl* call. Recall that the Ant script can be used to deploy the *sayings* service under Tomcat:

```
% ant -Dwar.name=cliches2 deploy
```

Example 1.4 A suite of *curl* calls against the *sayings* RESTful service

```
;; GET all sayings (XML response)
% curl localhost:8080/cliches2/   ;; shorthand for: curl --request GET localhost:8080/ ↩
    cliches2/

;; GET a specified saying (XML response)
% curl localhost:8080/cliches2?id=31

;; GET all sayings (JSON response)
% curl --header "Accept: application/json" localhost:8080/cliches2/

;; GET a specified saying (JSON response)
% curl --header "Accept: application/json" localhost:8080/cliches2?id=31

;; POST a new saying
% curl --request POST --data "who=TSEliot&what=This is the way the world will end"  ↩
    localhost:8080/cliches2/

;; GET all sayings to confirm the POST (new saying is at the end)
% curl localhost:8080/cliches2/

;; PUT new data into an existing saying
% curl --request PUT --data "id=33#what=This is an update" localhost:8080/cliches2/

;; GET all sayings to confirm the PUT (edited saying is at the end)
% curl localhost:8080/cliches2/

;; DELETE a specificed saying
% curl --request DELETE localhost:8080/cliches2?id=33

;; GET all sayings to confirm the DELETE
% curl localhost:8080/cliches2/
```

The XML responses from the revised *sayings* service are formatted exactly the same as in the original version. Here is a sample JSON response from a GET request on the `Prediction` with `id` 31:

```
{"java": { "class": "java.beans.XMLDecoder",
   "object": {
      "void": [
         {
            "int": 31,
            "property": "id"
         },
         {
            "string": "Balanced clear-thinking utilisation will expedite collaborative  ↩
               initiatives.",
            "property": "what"
         },
         {
            "string": "Deven Blanda",
            "property": "who"
```

```
         }
      ],
      "class": "cliches2.Prediction"
   }, "version": "1.6.0_21"
}}
```

The layout has been edited slightly for readability.

1.3 Providing an XML Schema for a RESTful Web Service

A web service client might prefer JSON over XML for various reasons but one obvious reason for this preference is that client is a JavaScript program ready to treat the JSON response document as a native JavaScript object. The JAX-B (Java API for XML-Binding) interfaces and classes offer a similiarly convenient way for a Java client of a RESTful service to treat an XML response document as a native Java object; but some preparatory work is required. Here is a summary of how things would work:

- The RESTful service offers client, in addition to its regular business operations, an XML Schema or equivalent. The XML Schema specifies a grammar for the XML payload that the client receives.
- A Java client can apply JAX-B utilities to the XML Schema in order to generate native Java data types. For example, the Java client of the *sayings* service could transform the XML list of predictions into a `List<Prediction>` and then process this `List` in some typical Java way. Languages other than Java have utilities comparable to JAX-B.

Figure X depicts this scenario.

The JAX-B utilities offer the client the chance to bypass XML parsing and, instead, to generate Java directly from XML. The next chapter goes into the JAX-B details; for now, the issue is how to generate an XML Schema for the *sayings* service.