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 then 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 does not constrain how the service is to 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 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 obvious 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 ends with 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 revisits 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 request that the response payload be in JSON instead of in 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 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. Listing 1 is the servlet; Listing 2 is the `Prediction` class; and Listing 3 is the `Predictions` class.

Listing 1: The PredictionsServlet with full support for the CRUD operations.

```java
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
    // another for the prediction ("what").
    public void doPost(HttpServletRequest request, HttpServletResponse response) {
        String who = request.getParameter("who");
        String what = request.getParameter("what");

        // Are the data to create a new prediction present?
        if (who == null || what == null)
            throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);

        // Create a Prediction.
        Prediction p = new Prediction();
        p.setWho(who);
        p.setWhat(what);

        // Save the ID of the newly created Prediction.
        int id = predictions.addPrediction(p);

        // Generate the confirmation message.
        String msg = "Prediction " + id + " created.\n";
        sendResponse(response, predictions.toXML(msg), false);
    }

    // PUT /cliches
    // HTTP body should contain at least two keys: the id (which prediction is
    // to be edited) must be present; the predictor or the prediction or both
    // should be present. See documentation below, however.
    public void doPut(HttpServletRequest req, HttpServletResponse res) {
        /* A workaround is necessary for a PUT request to Tomcat, which does
           not parse the request stream to generate the parameter map. A
           hack is thus required. */
        String key = null;
        String rest = null;
        boolean who = false;

        /* Let the hack begin. */
        try {
            BufferedReader br =
                new BufferedReader(new InputStreamReader(req.getInputStream()));
            String data = br.readLine();

            /* To simplify the hack, assume that the PUT request has exactly
               two parameters: the id and either who or what. Assume, further,
               that the id comes first. From the client side, a hash character
               # separates the id and the who/what, e.g.,

                  id=33#who=Homer Allision
             */
            String[] args = data.split("#");      // id in args[0], rest in args[1]
            String[] parts1 = args[0].split("="); // id = parts1[1]
            key = parts1[1];

            String[] parts2 = args[1].split("="); // parts2[0] is key
            if (parts2[0].contains("who")) who = true;
            rest = parts2[1];
        }
        catch(Exception e) {
            throw new HTTPException(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        }

        if (key == null)
            throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);

        Prediction p = predictions.getMap().get(key);
        if (p == null) {
            String msg = key + " does not map to a Prediction.\n";
            sendResponse(res, predictions.toXML(msg), false);
        }
        else {
            if (rest == null) {
                throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);
            }
            // Do the editing.
            else {
                if (who) p.setWho(rest);
                else p.setWhat(rest);

                String msg = "Prediction " + key + " has been edited.\n";
                sendResponse(res, predictions.toXML(msg), false);
            }
        }
    }

    // DELETE /cliches2?id=1
    public void doDelete(HttpServletRequest req, HttpServletResponse res) {
        String key = req.getParameter("id");
        // Only one Prediction can be deleted at a time.
        if (key == null)
            throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);
        try {
            predictions.getMap().remove(key);
            String msg = "Prediction " + key + " removed.\n";
            sendResponse(res, predictions.toXML(msg), false);
        }
        catch(Exception e) {
            throw new HTTPException(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        }
    }

    // Method Not Allowed
    public void doInfo(HttpServletRequest req, HttpServletResponse res) {
        throw new HTTPException(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
    }

    // Method Not Allowed
    public void doHead(HttpServletRequest req, HttpServletResponse res) {
        throw new HTTPException(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
    }

    // Method Not Allowed
    public void doOptions(HttpServletRequest req, HttpServletResponse res) {
        throw new HTTPException(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
    }

    // Send the response payload to the client.
    private void sendResponse(HttpServletResponse res, String payload, boolean json) {
        try {
            // Convert to JSON?
            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);
        }
    }
}
```

Listing 2: 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;
    }
}
```

Listing 3: 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);

        // Read the data into the array of Predictions.
        if (in != null) {
            try {
                InputStreamReader isr = new InputStreamReader(in);
                BufferedReader reader = new BufferedReader(isr);

                int i = 0;
                String record = null;
                while ((record = reader.readLine()) != null) {
                    String[] parts = record.split("!");
                    Prediction p = new Prediction();
                    p.setWho(parts[0]);
                    p.setWhat(parts[1]);
                    p.setId(mapKey);

                    predictions.put(String.valueOf(mapKey++), p);
                }
            }
            catch (IOException e) { }
        }
    }
}
```

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. In the end, it is hard to make a bad choice between Tomcat and Jetty.