Unit 5. RESTful Web Services

Outcomes Summary. You will learn

- how to design simple DSs using the Web and XML as a middleware
- why this approach is called RESTful
- how this approach compares with Java RMI
- implement simple RESTful DSs using the JAX-RS standard and the Jersey library

Further Reading: Richardson & Ruby 2007, web resources.

We have observed that Java RMI is a fairly simple-to-use yet powerful middleware for the development of object-oriented DSs. Although our examples were all client-server, one could adequately develop DSs based on peer-to-peer or other architectures in Java RMI. We may legitimately ask whether we really need anything else as far as DSs are concerned.

The shortcomings of Java RMI become obvious mainly in the context of creating DSs that aspire to be open and future-proof. Recall that an open DS must allow a programmer who has little connection to the original developers to alter or extend the system in a PL of their choice. Even if their PL of choice remains Java, the chances are high that the components of the DS are not as loosely coupled and their interfaces not entirely well documented. The power and flexibility of Java RMI could turn against its users unless they are very forward looking and spend enough effort developing a meticulous design and documentation for all remote interfaces. The power of Java RMI can also turn dangerous because it can transfer compiled classes behind the scenes together with serialisable parameters, effectively transferring executable code[1].

This is the first in a series of units introducing various Web Services (WS) middleware. All of these have in common that they build on top of the well-developed Web layer (URL, HTTP etc) and that they support the development of heterogeneous and open DSs via:

- detailed guidelines for PL-independent remote communication;
- standards demanding precise and detailed remote interface specifications;
- limitations on remote interfaces to keep them simple and easy to understand as well as PL-independent.

Here we start with the simplest WS middleware and later we introduce WS standards that almost re-create the power of distributed objects as provided by Java RMI yet with the benefits of interoperability and openness.

[1] We have seen an example of such hidden code mobility when a Java RMI chat client sent a chat message to a server. Despite the fact that the server was compiled without the class for chat messages, it could format the messages correctly as defined in the `toString()` method as overridden inside the chat message class. This was possible only because the compiled version of this chat message class was transferred from the client to the server during the first RMI that sent such a message to the server.

5.1 The Web as a middleware

The Web is the most successful DS ever developed and is proved to be massively scalable and open. Its original purpose is to share information among people via documents that link with each other in unlimited ways.

The main thesis of RESTful WS is that what worked well for people will also work well for computers when they, like people, need to share information. Indeed, for most DSs it makes sense to let the nodes publish their resources on Web servers and use computer-friendly Web "browsers" to help them communicate with each other by downloading and uploading representations of resources such as objects and files. A representation of an object is usually its PL-independent serialisation typically using XML or JSON syntax.

This way the communication in a DS is kept very simple and well specified. Each node holds a number of resources, some of which are accessible remotely via the HTTP protocol on specific URLs. This means that while we can have remote objects, the way they are accessed differs immensely from Java RMI. The only remote methods are the 8 HTTP methods, of which usually only the following ones are used:

- *GET*: ask a server for a representation of an object (or other resource)
- *PUT*: tell a server to create or update an object (usually according to its supplied representation)
- *POST*: tell a server to add a new element to a collection object (often create a new element object by a factory and fetch its canonical URL)
- *DELETE*: tell a server to no longer publish this object

Thus a whole object can be transferred between nodes using its representation but one has no way of invoking any interesting remote methods on them. This restriction seems to be rather severe, nevertheless it can buy worthwhile benefits.

5.2 RESTful chat system

Before we explore the RESTful principles further, let us work through a simple RESTful system: its design, interface specifications and a Java implementation.

5.2.1 The server's remote interface

Let us consider a single-server chat system with multiple topics. The objects/resources the server should export are:

- a unique collection of all topics
  URL: `<server base url>/topics/`, eg:
  `http://localhost:8080/rest-chat-server/topics/`
- individual topics, identified by their topics
  URL: `<server base url>/topics/<topic>/`, eg:
  `http://localhost:8080/rest-chat-server/topics/sport`
- the collection of all chat messages in a particular topic
  URL: `<server base url>/topics/<topic>/messages`, eg:
  `http://localhost:8080/rest-chat-server/topics/sport/messages`
- individual chat messages, identified by their serial numbers.
  URL: `<server base url>/topics/<topic>/messages/<msg num>`, eg:
  `http://localhost:8080/rest-chat-server/topics/sport/messages/3`

All of these resources except the individual topics support the HTTP GET method. To a GET request, the collection of topics returns a string containing a space-separted list of topic names while a collection of topic messages returns the number n of the latest message, which represents the range 1–n of numbers of all the messages. A GET request for a chat message returns an XML representation of the message.

Both types of collection resources accept POST requests to add a new element into the collection. Adding a topic to the topic collection requires the proposed topic name to be included in the request body. The new name must be distinct from all topics that already exist on the server. If successful, the request returns the full (ie absolute) URL of the new topic.

Posting a new chat message needs an XML representation of the message to be sent in the request body. The message must contain at least the sender and content components. Again, the server replies with the full URL of the new message.

The server will signal errors using the standard HTTP error codes. For example, any DELETE request will earn a 405 "bad request" response and a request to add an existing topic will be met with a 409 "conflict" response. Correct requests should be processed by the server and get a 200 "OK" response. A prototype of the server that does not implement some of the correct requests could respond with a 501 "not implemented" HTTP message for such requests.

To complete the specification, we should also say how message objects will be represented in HTTP requests. It is customary to use XML for such purposes and, at this stage, specify an *XML schema* for chat messages. We will defer this specification until

we have seen some of the server and client code for working with chat topics, which are communicated by plain strings.

5.2.2 Client for creating and listing chat topics

Let us first discuss how to program a *client* for the service, starting with the portions that are responsible for creating a new chat topic on the server and for listing existing chat topics. It is a good design to develop a proxy class for the service whose instances will play a similar role to the proxy objects used in Java RMI. A proxy object to the RESTful chat service will represent the chat server to the local JVM, ideally hiding the fact that the service is RESTful and make it look like a local Java implementation of the same functionality. The proxy is given the base URL of the server when created. Its methods such as `createTopic` and `listTopics` hide all the HTTP communication from the rest of the client system. Figure 5.1 shows a part of a proxy class for accessing the above server, focusing on the method `createTopic`. The proxy is making use of some classes from the Jersey library for programming RESTful systems.

Note that the Jersey library introduces the concept of a proxy for individual remote resources. Each instance of class `WebResource` acts as a proxy for one remote resource. These resource proxies are created using an instance of class `Client`, which thus serves as a factory for such proxies.

An instance of `WebResource` has methods `get`, `put`, `post`, `delete` to perform the remote RESTful remote requests. When these requests need to pass a resource representation as a parameter or as a return value, there is some flexibility in the Java types used for these representations. When using plain text representations, it is easiest to use String instances such as the parameter `topicName` on line 36 in Fig. 5.1. Notice that to indicate the return type of method `post` on line 36 we need to pass it as first parameter the `.class` field of the return type.

Some errors are automatically translated by the Jersey library into exceptions and others are not — for example, when the server does not recognise the topics collection URL, responding with the 404 error code, an exception is thrown automatically by the resource proxy. Nevertheless, when the problem is that a topic already exists, the server responds with a HTTP conflict error and the proxy has to detect it by inspecting the code and throw an exception itself. The name of the exception class `UniformInterfaceException` refers to the essential RESTful concept of *uniform interface*, which is explained in sub-section 5.3.1.

5.2.3 Server code for creating and listing topics

A node of a RESTful DS that holds resources must be attached to a web server or it has to have a web server integrated into it. Each HTTP request to a resource's URL has to be routed according to the URL to a Java object or class that defines how the request should affect the resource. Resources themselves can be internally embodied

```java
package rest.chat.client;

import java.net.URI;
import java.util.ArrayList;
import java.util.List;

import rest.chat.messages.ChatMessage;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.UniformInterfaceException;
import com.sun.jersey.api.client.WebResource;

public class ChatServerProxy
{
    private String serverURLTopics;
    private Client restClient;
    private WebResource topicsResource;

    public ChatServerProxy(String serverURLBase)
    {
        // initialise the base URL for topics:
        serverURLTopics = serverURLBase + "chatServer/topics/";

        // create an object that can create resource proxies:
        restClient = new Client();

        // create a proxy for the server's topics collection
        topicsResource = restClient.resource(serverURLTopics);
    }

    public URI createTopic(String topicName)
    throws UniformInterfaceException
    {
        ClientResponse response =
            topicsResource.post(ClientResponse.class, topicName);
        // check whether the server indicates an error:
        if(response.getStatus() < 400)
        {
            // return the reported resource URI:
            return response.getLocation();
        }
        else
        {
            // translate the HTTP error into Java exception:
            throw new UniformInterfaceException(response);
        }
    }

    public List<String> getTopics(){...}
    public URI newMessage(String topic, String sender, String content){...}
    public int getMsgCount(String topic){...}
    public ChatMessage getMessage(String topic, int msgNo){...}
    ...
}
```

Figure 5.1: Extract of code for a chat server proxy on a client using Jersey

```java
@Path("topics")
@Singleton
public class Topics
{
    private static class Topic
    {
        private Messages messages;
        public Topic(String topicName)
        {
            messages = new Messages(topicName);
        }
        public Messages getMessages(){ return messages; }
    }

    private Map<String, Topic> topics = new HashMap<String, Topic>();

    @GET
    public String getTopics()
    {
        String response = "";
        for (String topicName : topics.keySet()){ response += topicName + " "; }
        return response;
    }

    @POST
    public Response addTopicRespondCreated(String topicName)
    {
        addTopic(topicName);
        URL topicURL = UriBuilder.fromPath("{topicName}").build(topicName);
        return Response.created(topicURL).entity(topicName + "\n").build();
    }

    @Path("{topicName}") // topic sub-resource
    @PUT
    public void addTopic(@PathParam("topicName") String topicName)
    {
        synchronized (topics)
        {
            if (topics.containsKey(topicName))
            {
                String msg = String.format("topic %s already exists", topicName);
                Response response = Response.status(Status.CONFLICT)
                                            .entity(msg).build();
                throw new WebApplicationException(response);
            }
            else
            {
                topics.put(topicName, new Topic(topicName));
            }
        }
    }

    @Path("{topicName}") // topic sub-resource
    @DELETE
    public void deleteTopic(@PathParam("topicName") String topicName) { ... }

    @Path("{topicName}/messages") // messages sub-resource locator
    public Messages getMessages(@PathParam("topicName") String topicName)
    {
        return getTopic(topicName).getMessages();
    }

    private Topic getTopic(String topicName) throws WebApplicationException ...
}
```

Figure 5.2: Definition of a “topics” resource and its “topic” sub-resource.

```java
public class Messages
{
    private String topicName;
    private List<ChatMessage> messages;
    private int nextMsgSerialNo;

    public Messages(String topicName)
    {
        this.topicName = topicName;
        messages = new ArrayList<ChatMessage>();
        nextMsgSerialNo = 1;
    }

    @POST
    @Consumes("application/xml")
    @Produces("application/xml")
    public Response newMessage(ChatMessage msg)
    {
        int msgNo;

        synchronized(messages)
        {
            msgNo = nextMsgSerialNo;
            nextMsgSerialNo ++;
            // add the message to the collection:
            messages.add(msg);
        }

        // complete the message data:
        msg.setTopic(topicName);
        msg.setSerialNumber(msgNo);
        URL msgURL = UriBuilder.fromPath("/messages/{messageID}").build(msgNo);
        return Response.created(msgURL).entity(msg).build();
    }

    @GET
    public String getMessageCount()
    {
        int size;
        synchronized (messages){ size = messages.size(); }
        return "" + size;
    }

    @GET
    @Path("{msgNo:[1-9][0-9]*}")
    @Produces("application/xml")
    public ChatMessage getMessage(@PathParam("msgNo") String msgNoS)
    {
        int msgNo = Integer.parseInt(msgNoS);
        try
        {
            ChatMessage msg;
            synchronized (messages){ msg = messages.get(msgNo - 1); }
            return msg;
        }
        catch(IndexOutOfBoundsException e)
        {
            throw new WebApplicationException(Status.NOT_FOUND);
        }
    }
}
```

Figure 5.3: Definition of a "messages" resource and its "message" sub-resource.

eg by Java objects, database records or files. In our example each resource is enacted by a Java object.

Only a few years ago the primary way to write a RESTful server in Java was by writing a Java servlet that handled requests for all resources and had a long sequence of `if-then-else` statements decoding the URL path in order to identify the resource it refers to before calling a method of an object corresponding to the resource. All has changed with the arrival of dedicated Java libraries for RESTful programming such as Restlet and Jersey and the JAX-RS standard for annotating Java code with its role in a RESTful service. The current primary way to develop a Java RESTful application is to use a "very clever" universal servlet that:

1. scans all classes available to the JVM it runs on and looks-up all classes that contain some JAX-RS annotation;
2. finds among these classes the ones that correspond to the URL specified in the request it is dealing with;
3. creates or locates an instance of the appropriate class and calls the appropriate method(s) to perform the action required by the request.

The chat server discussed here is implemented using JAX-RS and has been deployed on a Tomcat web server using a universal servlet provided by the Jersey library.

Figures 5.2 and 5.3 show from our two server classes `Topics` and `Messages`. A unique instance of the former embodies the collection of topics and one instance of the latter is created for each topic to embody the collection of messages created for that topic.

The `Topics` class is annotated `@Path("topics")`, which indicates that this class represents the resource with the "root" URL appended by the string "topics". In this case the root URL means the URL that leads to the server such as Apache Tomcat and then to the deployed project containing this class. Eg in our case the URL is something like `http://localhost:8080/rest-chat-server/chatServer/`.

A method in such a class can be annotated `@GET`, `@POST`, etc to specify that it is a *resource method*, ie it is RESTfully exposed via the server for instances of this class. An additional `@Path` annotation can be placed on a resource method to indicate that the method refers to a *sub-resource* of the current instance of the class.

5.2.4 Supplying values to method parameters

A sub-resource method can apply generically to many sub-resources of the same kind. For example, the `deleteTopic` method in Fig. 5.2 should be applicable to any of the existing topics. As each resource has a unique URL, it is impossible to specify the exact path for such a method. Instead, the path annotation for such methods contains so-called path parameters. These can be recognised by the fact that they are enclosed in curly braces (`{topicName}` in our case). Path parameters should be connected to actual parameters for the sub-resource methods by annotating their formal parameters as shown in Fig. 5.2 on lines 35, 55 and 58.

Path parameters provide one way to give values to the parameters of resource methods. There are many other sources of values for method parameters. For example, if

a parameter is annotated with `@Context`, it tells the service to lookup a value for this parameter from its metadata related to the request. If the `@Context` parameter type is `Request`, then a complete description of the current HTTP request is made available to the method from which the method can obtain all the headers, authentication data etc.

There can be only one non-annotated parameter in each resource method — the value of this parameter is provided by the HTTP request "entity", ie the part of the request that carries the representation of the resource. By default, also the return value of a resource method is the representation of the resource to be sent back to the client in the response's "entity". XML serialisation is performed automatically whenever the declared type for the representation parameter or return value is a JAXB class (eg type `ChatMessage` used with a parameter on line 17 and as return type on line 47 in Fig. 5.3).

5.2.5 Error checking and reporting

By default path parameters will match any whole segment of the URL. In Fig. 5.3 on line 45 the path variable `msgNo` is associated with a regular expression ensuring that only sequences of digits that can be interpreted as an integer will match the path pattern. When a client does a GET request using an URL that leads to this resource but does not match this regular expression, the client will be given a "not found" 404 HTTP response.

Regular expressions sometimes cannot fully charcterise valid request URLs. Therefore, a resource method sometimes has to generate an error response. Eg the method `getMessage` should return a "not found" response if the number in the URL is out of range. The correct way to report such errors is to throw a `WebApplicationException` which has a response description embedded in it. The service will translate this exception into a corresponding HTTP response. If we were to let an exception such as `IndexOutOfBoundsException` propagate from a resource method, the client would be told that there was an internal server error, which would be misleading.

5.2.6 POST methods and response building

A POST request typically leads to the addition of a new element in a collection resource. It is common to require that the response to such a request contains the URL of the created element resource. Moreover, the URL is not placed in the "entity" element but in a location header. To achieve this effect, one has to specify as the return type the class `Response`, which gives the largest freedom to the method to specify various response details.

The methods `addTopicRespondCreated` on line 26 and`addTopic` in Fig. 5.2 are a typical example of programming a POST and PUT resource methods as two alternative ways of adding a new element to the same collection. In these two methods, we see two different ways to construct and use a response object. The variant in the PUT method constructs an error response for use in an `WebApplicationException`. (The same kind of response could returned directly instead of throwing an exception if the

return type of the resource method is `Response`.) Note that the entity part of the error response can be used to specify a custom error message. The variant in the POST method constucts a success response that includes the URL of the new topic.

The methods `status` and `created` are two of the many static "factory" methods in class `Response`. These methods create a `ResponseBuilder` instance which itself has similarly named methods (including the method `entity` used in our example) that allow one to alter some aspects of the response being built and return the modified `ResponseBuilder` so that further alterations can be chained in one expression. When the response builder alterations are complete, the method `build` is used to construct and return the specified `Response` instance.

A similar pattern is used to construct the URL using a `UriBuilder` class that allows one to first specify the URL with some patterns and then in the `build` method supply the values that the patterns should be replaced by.

5.2.7 Sub-resource locators

A method annotated with `@Path` but lacking any REST method annotation delegates all handling to the object that it returns. For example, the method `getMessages` in our scenario returns a `Messages` instance, which has its own resource methods that will be used to handle any requests to this topic or its sub-resources.

5.2.8 Persistent data

When a JAX-RS annotated class is deployed on a server such as Apache Tomcat, by default each request for a resource is handled by instantiating the class that implements that resource and then calling the appropriate method for that new instance. This behaviour is tailored for stateless services because concurrent requests have separate instances and there is less danger of conflicts between them. Nevertheless, our chat server is not stateless, as eg the list of topics is shared among all requests, whether concurrent or separated by some time of inactivity.

One solution is to have shared data, such as the field `topics` in class `ChatServer`, declared static. The Jersey implementation of JAX-RS offers a better solution: the annotation `@Singleton` before the class `Topics` changes the behaviour of the server to create only one instance of this class and keep it in its memory and use it for all requests instead of creating a new instance for each request. No such annotation is required for class `Messages` because its instantiation is not performed by the server automatically, instead is the responsibility of the parent resource, ie the `Topics` class and its messages resource locator method.

One has to take measures to protect such shared data because it is very likely that it will be accessed concurrently. In the example code, any access to the field `topics` is synchronised, locking the topics object for other threads while reading or writing its contents. Similarly, all access to the `messages` together with the `nextMsgSerialNo` variable, which is closely associated with this collection, is synchronised.

5.2.9 Specifying the format of chat messages

Let us now explore the design of chat message representations. According to the specification, a message sent by a client has to have two string components: sender and content. The server stores a richer record, including also serial number and topic. There is no problem expressing these records in XML, eg as shown in Figure 5.4.

Why is XML the representation vehicle of choice in Web Services? Mainly because there are tools for virtually any PL that make it very easy to parse and format XML data to and from the PL's native structures (ie objects in the case of Java). In our Java client and server, we shall use the JAXB library that is included in Java 6 SE and Java 5 EE.

JAXB helps us not only with XML parsing and formatting, it even automatically generates the class `ChatMessage` whose instances are the counterparts for XML in the parsing and formatting processes. The only thing that JAXB needs is an XML schema that defines the generic structure of the messages (the XML equivalent of a Java class).

The schema used here is shown in Figure 5.5. Recall that a schema defines a number of elements and for each element it specifies the possible content of the element. Our schema defines only one top-level element that may contain four other elements. Two of them are compulsory because their "occurs" are bounded by 1 from both sides. The other two are optional because its "occurs" are bounded between 0 and 1. These 2–4 elements can appear in any order because they are enclosed within the `all` element (to fix the order, use the `sequence` element instead).

The `complexType` element must be present to make the schema valid — it turns the description into a formal type. This kind of type definition embedded inside an element definition is anonymous and thus cannot be reused in other element definitions without copy and paste. In more involved schemas, there are separate type definitions and these are referenced inside element definitions. Nevertheless, the JAXB tools included in Java 6 SE at the time of writing have a slight problem with such schemas and require manual editing of the generated Java classes.

5.2.10 Sending and receiving of chat messages

We need to know how to code instructions for:

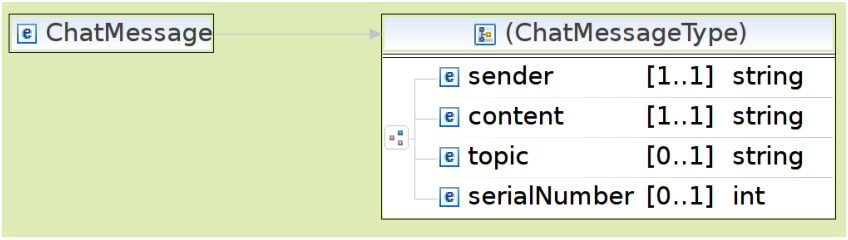
- *XML marshalling*: formatting of `ChatMessage` objects to their corresponding XML and at the same time send this XML to an output stream provided by the HTTP connection;
- *XML unmarshalling*: parsing of XML coming from an input stream provided by the HTTP connection and constructing a `ChatMessage` object that mirrors the XML structure.

Both are provided by JAXB as promised. Moreover, the Jersey servlet and client classes take care of when and how these translations are invoked. A JAX-RS resource method that requires XML marshalling and/or unmarshalling of a parameter or return value

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ChatMessage xmlns="rest/chatserver/messages">
    <sender>TestSend</sender>
    <content>got smashing gaming pad!</content>
</ChatMessage>
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ChatMessage xmlns="rest/chatserver/messages">
    <sender>TestSend</sender>
    <content>got smashing gaming pad!</content>
    <topic>hw</topic>
    <serialNumber>2</serialNumber>
</ChatMessage>
```

Figure 5.4: Example messages represented in XML




```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="rest/chatserver/messages"
    elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="rest/chatserver/messages">

    <element name="ChatMessage">
        <complexType>
            <all>
                <element name="sender" type="string" maxOccurs="1"
                    minOccurs="1">
                </element>
                <element name="content" type="string" maxOccurs="1"
                    minOccurs="1">
                </element>
                <element name="topic" type="string" maxOccurs="1"
                    minOccurs="0">
                </element>
                <element name="serialNumber" type="int" maxOccurs="1"
                    minOccurs="0">
                </element>
            </all>
        </complexType>
    </element>
</schema>
```

Figure 5.5: An XML schema for chat messages.

only needs to specify a JAXB-generated class as the type of the parameter or return value. When executing this method, Jersey notices that this is an XML serialisable class and takes care of the rest.

The marshaller and unmarshaller used by Jersey are not guaranteed to format and parse messages as valid XML with respect to the schema that was used to generate class `ChatMessage`. The unmarshaller as created in the code above will be quite tolerant when parsing XML, accepting even invalid ones. For example, if the XML contains multiple or no sender elements, it will still parse it and produce an instance of `ChatMessage`, either ignoring some of the sender elements or setting the sender field to null. Similarly, the marshaller will ignore fields whose value is null and they will not appear in the XML even if they are compulsory according to the schema.

To ensure strict checking of the schema, one has to write two special classes that replace the default Jersey marshaller and unmarshaller for a specific JAXB class. An example of a validating unmarshaller can be found using this link

5.2.11 Notification of new messages

With the current design, the chat clients have to contact the server to find out whether there are any new messages. If they need to stay up-to-date, they need to poll the server. It is also possible to set-up notification so that the server would tell clients about new messages. Nevertheless, this has several consequences:

- The clients that require notification will have to run their own web servers so that the chat server can contact them using HTTP. They will give the server their message collection URL such as:

  ```
  http://localhost:8080/chat-client1/messages
  ```

  The server will then HTTP POST each new message to this URL.
- The chat server will have to provide a "subscribers collection" resource via which others can subscribe. It could be exposed via URLs such as:

  ```
  http://localhost:8080/rest-chat-server/topics/sport/subscribers
  ```

  - HTTP POST together with the client's base URL for its own collection of messages will add a subscriber to the collection. The HTTP POST request will respond with a subscriber ID number.
  - HTTP PUT to the above URL extended with the subscriber ID number (eg `.../sport/subscribers/13`) will be used to update the client's messages URL when the client relocates.
  - HTTP DELETE to the subscriber's URL (eg `.../sport/subscribers/13`) will cancel the subscription.
  - The server will cancel a subscription if it repeatedly fails to deliver HTTP POST request to the client.
  - The client can use HTTP GET on its subscription URL to check that its subscription has not been cancelled due to earlier network errors.

It is clear that notification is harder to implement as a RESTful service than as a Java RMI service. Nevertheless, as we said, it has the advantage of being an open and PL-independent solution. With advanced libraries for RESTful programming such as Jersey, implementing the above features becomes fairly straightforward.

5.2.12 Security

As presented so far, the chat system is open to all kinds of attacks. To mention just a few of the simplest ones:

- chat messages could be read and modified by a malicious node on the IP route underlying the HTTP channel;
- a rogue server could take over the real server's domain name and then modify messages, withhold messages from clients, etc;
- a client could modify other client's earlier chat messages using HTTP PUT;
- a client could post a message with a false sender indicator;
- a client could subscribe another client to a topic they do not want.

Most of these attacks can be prevented fairly simply using HTTPS encryption and authentication. Both client and server would be modified to use HTTPS and setup a certificate for their authentication.

The server would be also modified to make appropriate use of the knowledge of the client's identity, eg by checking that a message update request comes from the same client that posted it in the first place. Similar but weaker identity checks could be performed using HTTP cookies or basic HTTP username/password authentication.

5.3 RESTful principles

In this section, we reflect on what we have seen in the example system in order to separate essential general principles behind a RESTful system from unimportant implementation details. We will express and evaluate these principles and provide the background necessary for the exploration of other kinds of Web Services.

5.3.1 The essence of RESTful services

We have motivated RESTful WSs as an extension of the exiting web infrastructure. Nevertheless, it turns out that the principles that we derived from the web infrastructure can be applied to other middleware than HTTP. Actually, with a bit of discipline we can program RESTful services without HTTP eg in Java RMI or JMS.

While pure HTTP is the most common middleware choice for RESTful services, sometimes it is obstructed by strict firewalls that allow only HTTP GET and POST or certain limitations of HTTP such as the limit on the length of a URL. The usual trick to overcome these obstructions and limitations is to use HTTP POST to the base URL for

all requests and code the actual method, full URL and request details in the body of the HTTP POST request. Again, this requires discipline because HTTP POST can be misused in this way to encode any kind of request, not necessarily conforming to the RESTful principles outlined below.

The characteristics that make a service RESTful are:

- It exposes a set of *resources* (some of which may be hypothetical — they do not actually exist on the server but the server is capable of creating them).
- Each of its resources has its *unique address* (eg a URL).
- Resources can be *logically related* and this should be expressed via addresses that are similar to each other.

  The addresses of resources are typically *hierarchically structured* according to resource *aggregation*.
  Ie the address of a resource A/B that belongs to resource A extends the address of resource A. The most common aggregations are:

  - A/B is a member of a collection A (eg a message in a topic);
  - A/B is derived from A (eg set of replies to a chat message).

- Each of its resources has one or more *transferable representations* (usually XML conforming to a schema) and each representation has its unique address that extends the address of the resource. Usually there is a default representation whose address coincides with the address of the resource.
  (Eg A/B can stand for both a chat message as well as its XML representation and A/B.pdf can stand for its Portable Document Format representation.)
- All of the resources are accessed using a subset of the same *uniform interface* consisting of the following operations:

  - *fetching* a representation from the server
  - *overwriting* the resource according to a given representation
  - *deleting* the resource from the server
  - *appending* to the resource according to a given representation

  (These correspond to the intended effect of the HTTP methods GET, PUT, DELETE and POST but can be implemented any other way.)

The simple nature of the uniform interface is the main ingredient of the success of the RESTful approach. It encourages loose coupling between the clients and the server and makes it much easier to provide a clear and unambiguous interface specification. Moreover, the first three operations can all be duplicated with no bad side effect (fetching, overwriting or deleting a resource twice has the same effect as doing it once). Such operations are called *idempotent*. Idempotency makes it easier to achieve reliability despite network delays and failures. Extra care to eliminate duplicates has to be taken only with appending (ie POST) operations. There are standard ways of doing this, eg so-called POST Once Exactly (POE).

The acronym **REST** stands for *Representational State Transfer*. The name stems from the requirement to communicate a representation of the full resource when overwriting it or of the new part of the resource when appending to it. The only way a client can change the state of the server is by transferring a representation of one of the server's resources. The client must assume that the server will not remember anything about its previous requests except that the resources have changed according to those previous requests. In particular, a request cannot directly mention other requests. Each request makes sense to the server in its own right, not as a part of a sequence of requests. A technical name for this property is that the service is *stateless* with respect to the access protocol. On the other hand, RESTful services are *stateful* with respect to the resources they expose.

5.3.2 Example design methodology

Resource-oriented architecture is a practical DS development methodology based on a particular interpretation of the RESTful principles. In essence, it encourages one to proceed as follows:

1. Describe high-level data flows and distributed processes.

   Eg: clients create topics on server, send chat messages to server's topics, read messages from topics, subscribe for notification; server notifies clients of new messages.

2. Split the data set into resources.

   Eg, set of topics, message list for topic, message, message thread, set of subscribers for topic, client's message list.

3. Name the resources with URLs.

4. For each resource, expose a subset of the uniform interface and specify security restrictions.

5. Design the representations accepted from clients and served to clients.

6. Integrate resources with one another using hyperlinks.

7. Specify the side-effects of creating and modifying resources.

   Eg, posting a message will cause the server to re-post the message to all subscribers, posting a message that is a reply to another message will add one item to its set of replies.

8. Check that all required processes are well supported by the resources and their methods.

9. Specify error conditions, ie what could go wrong and how to respond.

5.3.3 Evaluation of the RESTful approach

The main advantages of RESTful services are:

- their interfaces tend to be easy to specify and easy to understand;
- they tend to be scalable and reliable;
- both clients and servers are relatively easy to develop.

On the other hand, the RESTful approach constrains the designer to an unusual way of thinking and may force them to expose more of the service than they would wish. For example, it is complicated to rename a resource once it has been created.

Currently there are not many tools to automate the development of RESTful services. For example, there seem to be no tool to obtain a machine-readable description of a RESTful service and generate most of the server or client code automatically. There is some effort in this direction but some of the main advocates of RESTful approach argue that the benefits of such tools are not worth the effort. There is a language called **WADL**, in which one can precisely describe the structure of resources, their URLs and access methods and even the way resources can be hyper-linked with each other. The argument against using these descriptions to automatically generate code is that the most important aspects such as the processes and effects of resource creation are best formalised by an actual implementation of the system. Thus instead of designing new languages such as WADL, the RESTful advocates encourage the use of advanced RESTful libraries in mainstream PLs (such as Jersey in Java) for concise and readable implementations of services.