Unit 5 RESTful Web Services

Unit Outcomes. Here 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

Contents

Introduction to Web Services
Motivation

- anything wrong with Java RMI?
  - very flexible middleware . . . too flexible?
    - allows mobile code
    - allows tightly coupled designs
  - all nodes must run on a JVM — not sufficiently heterogeneous
  - insufficient machine-readable remote interface documentation
    - parameter data structures?
      (~~cannot~~ share whole class definitions)
      need to

Middleware for open systems

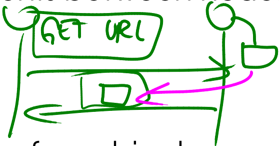
- ideally we need:
  - detailed guidelines for PL-independent remote communication
  - standards demanding precise and detailed remote interface specifications
  - limitations on remote interfaces to keep them simple and easy to explain
- Web Service (WS) middleware aims:
  - support for openness
  - + some of the power of distributed objects

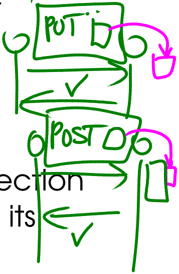
HTTP as a middleware
RESTful WSs motivation

- Web is the most successful DS ever
  - massively scalable
  - totally open
- main idea of RESTful WSs:
  - Web's good for people, Web's good for computers! (+ HTTP can do remote publishing)
  - coined in Fielding 2002
- some RESTful success stories:
  - Atom blog publishing interface
  - Amazon Simple Storage Service (S3)
  - Ruby on Rails

Remote objects via HTTP

- remote reference = URL
- can transfer objects and their components between nodes
- cannot call methods
- more precisely we can:
  - *GET*: ask a server for a representation of an object
  - *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 (often create a new object in a factory and fetch its canonical URI)
  - *DELETE*: tell a server to no longer publish this resource





CRUD

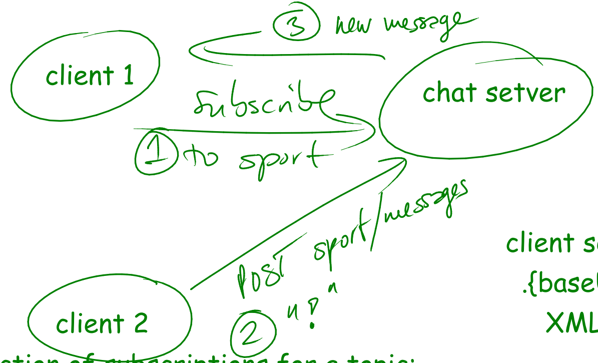
RESTful chat system
Server's resources (1/2)

- *collection of topics*
  - `http://localhost:8080/chat-server/topics`
  - ~~POST: adds a topic~~ ~~topic name as ASCII string~~
  - GET: returns a list of topics

+ topic resource on URL .../topics/{topicName}; PUT to add new topic

better way to add a topic - client decides last URL segment

- *collection of messages* for a topic
  - eg `.../topics/sport/messages`
  - POST: adds a new message
  - GET: returns the serial number of the last message



client service:

.{baseURL}/notifications

XML including URL of new message

collection of subscriptions for a topic:

.../topics/{topicName}/subscriptions

POST subscription info including URL of client's listener service

individual subscription

.../topics/{topicName}/subscriptions/{subscriptionID}

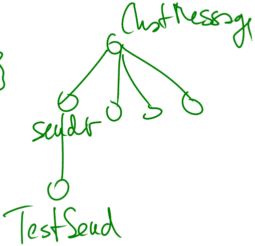
DELETE

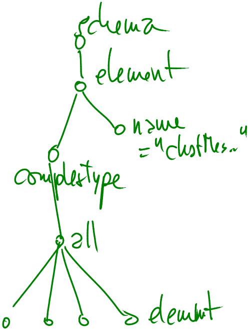
PUT - XML containing listener URL and status (paused/active)

Server's resources (2/2)

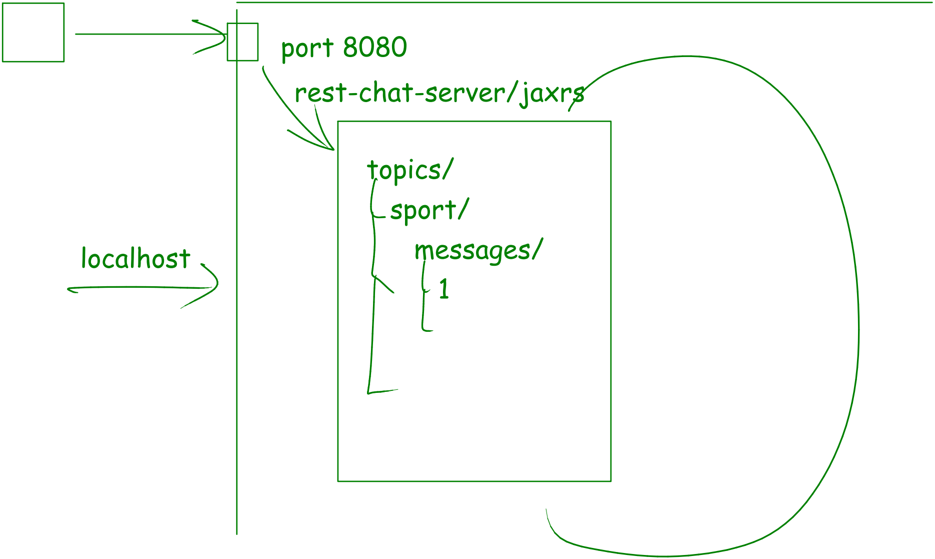
- *messages* identified by serial numbers
  - eg `.../topics/sport/messages/177` {msgId}
  - GET: retrieve the message
  - PUT: modify the message (only sender)
- *message threads* (exercise!)
  - eg /{topic}/threads/{threadID}/messages/
  - GET:
  - PUT:
  - POST:

/{topic}/messages/{msgID}/replies





http://localhost:8080/rest-chat-server/jaxrs/...



RESTful principles
REST outside HTTP

- can use other addressing than URI
- can use other protocol than HTTP
    - SMTP
        - messaging via email accounts
        - fully asynchronous but slow
    - Java RMI
        - lose interoperability and openness
        - keep maintainability and scalability

Overloading HTTP POST

- why not pure HTTP:
  - HTTP PUT and DELETE sometimes firewalled
  - URIs can get long — HTTP imposes limit
- common solution: overload HTTP POST
  - request data =
    method name + full URI + actual request data
  - disadvantages:
    - maybe tempted to invent new methods
    - POST is not usually cached

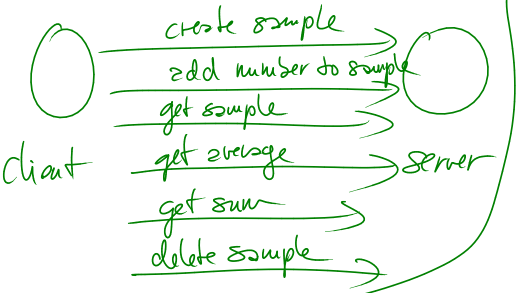
The essence of REST

- exposes *resources*
- each resource on *unique address*
- resources can be *related*, often hierarchically
  - *addresses related* too, eg:
    - A/B is a member of a collection A
    - A/B is derived from A — example?
- each resource has one or more *transferable representation*
- each *representation addressable* (extending resource addr)
- resources are accessed using *uniform interface* (fetch, overwrite, delete, append)

Example design methodology

1. Describe high-level data flows and distributed processes.
2. Split the data set into resources.
3. Name the resources with URIs.
4. For each resource, expose a subset of the uniform interface and specify security restrictions.
5. Design the representations accepted from and served to clients.
6. Integrate resources with one another using hypermedia links.
7. Specify the side-effects of creating and modifying resources.
8. Check that all required processes are well supported.
9. Specify error conditions, ie what could go wrong and how to respond.

1. describe high level data flows and distributed processes



notification:

.../samples/{sampleID}/subscribers POST

.../samples/{sampleID}/subscribers/{id

2. resources:	3	4	5
collection of samples:	.../samples	POST	none
sample	.../samples/{sampleID}	POST, DELETE, GET	XML
average	.../samples/{sampleID}/average	GET	decimal string
sum	.../samples/{sampleID}/sum	GET	-||-
~~numbers coll.~~	~~.../samples/{sampleID}/numbers~~	POST	-||-

6 extend the scenario temporarily to also serve "union samples"

resources:

- collection of union samples ..../unionsamples POST
  - union sample ..../unionsamples/{sampleID} GET XML
    - average - - -
    - sum - - -

(list of
URLs
to samples

Exercise

- design a simple client-server RESTful system
- client can:
  - register a user account with the server
  - upload text to the server
  - retrieve a translated version from server

Exercise — model answer

Evaluation of REST

- the good:
  - interfaces easy to specify and understand
    - formalisable using
      Web Application Description Language (*WADL*)
  - tend to be scalable and reliable
  - both clients and servers are relatively easy to develop
  - good support libraries exists
    (mainly Ruby on Rails, Java Jersey, Python Django)
- the negative:
  - unusual model, far from common OO methodology
    (but close to functional programming)
  - not many tools to automate development yet
  - harder to hide private resources

JAX-RS standard
Overview

- RESTful remote resource types — classes annotated with `@Path`
- uniform interface — method annotated with `@GET`, `@POST` etc.
- resource representation — as parameter and/or return value
- XML serialisation performed automatically for JAXB classes
- parameters in URI paths — method parameters annotated with `@PathParam`
- subresources — methods that return a subresource object

Basic annotations example

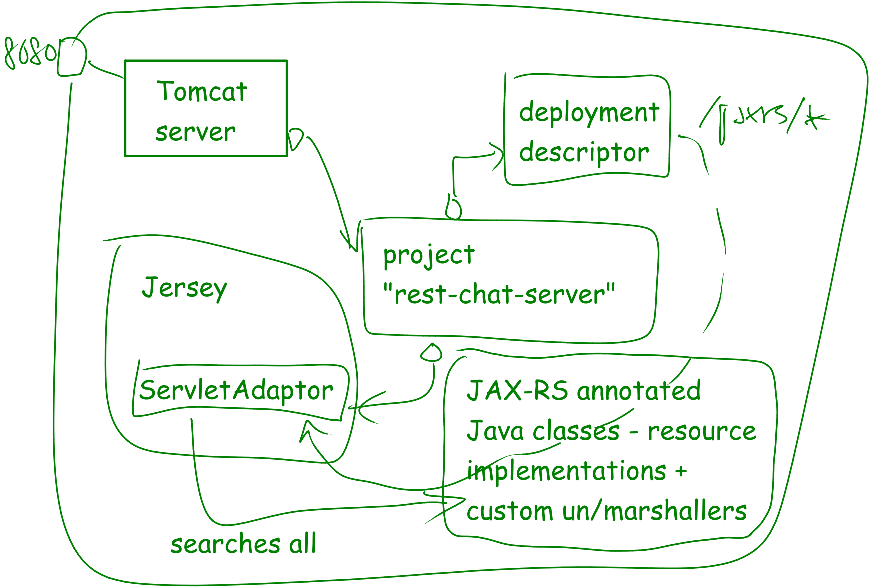
- part of a class `Topics`:

```
@Path("topics")
public class Topics
{   ...
     @GET
     public String getTopics()
     { ... }

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

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

Deployment overview



Tomcat server

deployment descriptor

project "rest-chat-server"

Jersey

ServletAdaptor

JAX-RS annotated Java classes - resource implementations + custom un/marshallers

searches all

GET method with XML serialisation

- simplified part of a class `Messages`:

```
@GET
@Path("{msgNo}")
@Produces("application/xml")
public ChatMessage
   getMessage(@PathParam("msgNo") String msgNoS)
{
   int msgNo = Integer.parseInt(msgNoS);
   return messages.get(msgNo - 1)
}
```

- Class `ChatMessage` is a JAXB class
  (usually created automatically from XML schema)
- serialisation performed automatically

Dealing with erroneous requests

- improved version of the method in previous slide:

```
@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);
    }
}
```

Typical PUT method

```
@Path("{topicName}") // topic sub-resource
@PUT
public void addTopic(@PathParam("topicName") String topicName)
{
    synchronized (topics)
    {
        if (topics.containsKey(topicName))
        {
            // report conflict:
            Response response =
                Response.status(Status.CONFLICT).build();
            throw new WebApplicationException(response);
        }
        else
        {
            topics.put(topicName, new Topic(topicName));
        }
    }
}
```

Typical POST method

```
@POST
public Response addTopicRespondCreated(String topicName)
{
    addTopic(topicName); // in the previous slide
    URI topicURI =
        UriBuilder.fromPath("{topicName}").build(topicName);
    return Response.created(topicURI).build();
}
```

Stateful resources

- by default: each request — new resource instance
  - advantage: no problems with concurrent access
  - problem: resource object state is lost
- solutions: use static fields OR special annotation:

```
@Path("topics")
@Singleton
public class Topics
{
    private Map<String, Messages> topics =
        new HashMap<String, Messages>();
    ...
    public void addTopic(@PathParam("topicName") String topicName)
    {
        synchronized(topics) { ... }
    }
}
```

Routing to sub-resources

```
@Path("topics")
@Singleton
public class Topics
{
    ...
    @Path("{topicName}/messages")
    public Messages
        getMessages(@PathParam("topicName") String topicName)
    {
        synchronized(topics){ return topics.get(topicName); }
    }
}

public class Messages
{
    ...
    @GET
    public String getMessageCount() { ... }
    ...
}
```

Learning Outcomes

Learning Outcomes. You should now be able to

- state the defining principles of RESTful systems
- design a RESTful remote interface for a simple client-server system
- argue for and against using RESTful Web Services for DS development comparing it to Java RMI
- extend an existing simple RESTful application in Java using JAX-RS and the Jersey library
- make good use of XML and XML schemata in RESTful applications