# SOFTENG 325 Software Architecture
# Lab 2

Ian Warren

August 1, 2017

The purpose of this lab is to reinforce week 2's lecture material and to give you some practical experience with configuring and developing simple REST Web services. You will work with a servlet container, first directly with servlets and then with JAX-RS.

## Tasks

### Task 1: Build the supplied Fibonacci project

`softeng325-lab2-servlet-fibonacci` contains a Web application that is implemented using a servlet. The project includes:

- `RabbitCounterServlet`, a servlet that implements a REST Web service for manipulating numbers in the Fibonacci sequence. The servlet responds to HTTP GET requests to retrieve either a value at a particular position within the sequence, or all Fibonacci values known to the Web service. It responds to HTTP POST requests to generate and store values for specified positions within the sequence, and, for HTTP DELETE requests, it deletes the value at a specified sequence position.
- `RabbitCounterCounsumerIT`, a JUnit integration test for testing the functionality of the Web service.
- `web.xml`, the mandatory configuration file for a WAR (Web application ARchive) application.
- `pom.xml`, a minimal POM file that defines the output of the build process to be a WAR file (note the `packaging` element). In addition, the POM file specifies dependencies on the servlet library, necessary to compile and run `RabbitCounterServlet`, slf4j logging and JUnit.

The project is structured as a Maven single-module project, and includes the directory structure for storing the web.xml file: `src/main/webapp/WEB-INF/web.xml`. Where the POM's `packaging` element's value is `war`, the `web.xml` file is expected to be stored in the `WEB-INF` directory, and Maven's `package` phase generates a `war` file as opposed to a `jar` (which is the default `packaging` value). The structure of a `war` file is well specified; it includes a directory named `WEB-INF` that contains the `web.xml` file and two subdirectories: `classes` and `lib`. `classes` contains the Web application's compiled code and any other resources, and `lib` contains any third-party libraries (`jar` files). Since a `war` file has a well-defined structure, any servlet container, e.g. Jetty or Tomcat, can be used to host the Web application contained in the `war` file. You might want to view the contents of the Fibonacci `war` file once you've built the application – you can view `war` files using utilities like 7-Zip.

The purpose of the `web.xml` file is to specify the servlet class and url pattern for which the servlet will process HTTP requests. The supplied `web.xml` file names the servlet `RabbitCounter` and declares that the servlet class is `nz.ac.auckland.fibonacci.RabbitCounterServlet`. It also

specifies that the servlet will process HTTP requests where the domain name has a `rabbit` suffix. Where the domain name prefix is `localhost`, requests of the form `http://localhost/rabbit` would be routed through to the `RabbitCounter` servlet.

### (a) Import the project

Import the project into your Eclipse workspace.

### (b) Flesh out the POM file

As part of the Maven build process, Maven can arrange for the packaged Web service to be deployed in an *embedded* servlet container. An embedded container is one that runs in the same process as the client (integration tests in this case) that will invoke the Web application hosted by the container.

Just before the `integration-test` phase, you want Maven to start up the embedded servlet container and host the Web application. Once started, you want Maven to run the integration test. When the tests have finished, Maven should shutdown the servlet container. The use of an embedded servlet container in this way is very convenient for development and testing – you don't need to install and run a full standalone servlet container or repeatedly redeploy the WAR file every time you recompile the code. Using build tools like Maven in this way thus promotes *testability* of distributed applications.

To effect the above approach to integration testing, you need to configure two plugins:

- `Failsafe`. This is Maven's plugin for running integration tests. It runs any JUnit tests in the `src/test` directory with an "IT" (integration test) suffix.
- `Jetty`. The Jetty plugin runs the embedded Jetty servlet container.

A description of how to configure the two plugins so that Jetty starts up and deploys the Web application prior to running the integration tests, and shuts down afterwards, is available at:

```
http://maven.apache.org/surefire/maven-failsafe-plugin/usage.html
```

Other than what's described in the above web page, you need additional declarations in the `configuration` element of the Jetty plugin:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.2.2.v20140723</version>
  <configuration>
    <httpConnector>
      <port>10000</port>
    </httpConnector>
  <webApp>
    <contextPath>/</contextPath>
  </webApp>
    ...
</configuration>
...
</plugin>
```

The `httpConnector` specifies the port that the servlet container will listen on for incoming connection requests, while the `webApp contextPath` specifies the base URI for serving requests.

In addition, you should specify that the `goal` to execute during the `pre-integration-test` phase should be `run-war` rather than `start`. `run-war` will deploy the `war` file when the servlet container starts, whereas `start` merely starts the servlet container.

```
<execution>
  <id>start-jetty</id>
  <phase>pre-integration-test</phase>
  <goals>
    <goal>start</goal>
  </goals>
  ...
</execution>
```

If you get really stuck with configuring the plugins, look ahead to the project for task 2, `softeng325-lab2-parolee`. The POM for this configures the plugins as required for a different Web application project.

### (c) Build and run the project

To build and run the project, run Maven with the `verify` goal. This will compile, package (generating the WAR file), and run the integration tests on the deployed Web service. The integration tests should run without error.

### (d) Reflect on the project

Study the code to ensure you understand it, and think about the following questions:

- How does use of the servlet container and programming model simplify application development? Consider what would be involved if you had to write your own HTTP server.
- In what way is servlet programming still "low-level" programming?
- How does the servlet leverage the HTTP protocol?
- Why is the servlet's state (`_cache`) a threadsafe `Map`?

## Task 2: Build the supplied Parolee project

Project `softeng325-lab2-parolee` is a simple JAX-RS project. It provides a REST Web service that allows clients (consumers) to retrieve, create, update and delete parolees.

From version 3.0 of the Servlet specification, servlet containers are JAX-RS aware. This means that they automatically load the JAX-RS servlet class (`HttpServletDispatcher`) at startup time. Since you are packaging a WAR project, you still need a `web.xml` file – but it can be empty as the servlet container knows which `Servlet` class to instantiate. Hence, the `web.xml` file for this project is empty.

JBoss' RESTEasy implementation of JAX-RS is used, and several necessary dependencies are specified in the POM.

### (a) Import, build and run the project

The project is complete and ready to build. Import the project into your Eclipse workspace. Build and run the project, similarly to how you did for task 1.

### (b) Experiment with logging

The project's POM file includes additional dependencies for logging, including the log4j implementation. The behaviour of log4j is configured using the `log4j.properties` file, a sample file is included with the project in `src/resources`.

When developing HTTP applications, it's often useful to examine the contents of HTTP messages. The JAX-RS implementation uses a HTTP library that uses logging. This project's `log4j.properties` file can be used to control what the HTTP library outputs. By default, the

`log4j.properties` file causes only INFO-level logging to be output. However, you can specify the logging level for particular namespaces, e.g. DEBUG for `org.apache.http`:

```
log4j.logger.org.apache.http=DEBUG
```

This causes DEBUG-level messages (in addition to INFO-level messages) to be output by any code in package `org.apache.http` or its subpackages. When set to DEBUG, a lot of useful information about HTTP messages, including headers, payloads and response codes is logged, and, in the case of the supplied `log4j.properties` file, displayed to stdout.

After editing the `log4j.properties` file, rebuild and run the project so that you can examine the contents of the HTTP messages being generated by JAX-RS and exchanged between the client and Web service.

### (c) Reflect on the project

Study how the JAX-RS API is used in this simple project. Consider:

- The way in which the REST interface offers CRUD functionality for parolee resources.
- The abstraction offered by the the JAX-RS framework over basic servlet programming.
- Aspect(s) of the `softeng325-lab2-parolee` project that you think could be better addressed by the JAX-RS framework.

## Task 3: Develop a JAX-RS Concert service

Project `softeng325-lab2-concert` is a partially complete JAX-RS project for a simple REST Web service that manages concerts. The project includes a skeleton Web service implementation (`ConcertResource`), an integration test (`ConcertResourceIT`), a class to represent concerts (`Concert`) and a complete POM.

The service is to provide the following REST interface:

- `GET /concerts/id`, retrieves a `Concert` based on its unique id. The HTTP response code is either 200 or 404, depending on the existence of the required `concert`.
- `GET /concerts ? start & size`, retrieves a `List` of `Concerts` where the `start` query parameter specifies the `id` of the first `Concert` to retrieve, and the `size` parameter is the maximum number of `Concert`s to return (with increasing `id`s).
- `POST /concerts`, creates a new `Concert`. The HTTP request message should contain a `Concert` for which to generate an `id` and store in the Web service. The response message should have a status code of 201 and a `Location` header giving the URI of the new `Concert`.
- `DELETE /concerts`, deletes all `Concert`s known to the Web service. The HTTP response message's status code should be 204.

Instead of using XML (as with the Parolee service), this service is to use Java serialization as the format for exchanging `Concert` data in HTTP request and response message bodies. In addition, the service is to track a client session using a HTTP cookie.

### Complete the application

Begin by importing the `softeng325-lab2-concert` project into your workspace. Complete the application service, implementing the necessary `Application` subclass and fleshing out the `ConcertResource` class. For `ConcertResource`, you need to add instance variables, method bodies and all metadata (through JAX-RS annotations).

*Use of Java serialization*
The JAX-RS framework is extensible with respect to data formats (more on this later). For this project, a class named `SerializedMessageBodyReaderAndWriter` is provided that plugs into the JAX-RS framework and which manages the conversion of serialized data in HTTP message bodies to `Serializable` Java objects. You don't need to be familiar with the details of this class' implementation – suffice it implements two JAX-RS interfaces `MessageBodyReader` and `MessageBodyWriter`. Any implementation of these interfaces can be registered with the JAX-RS framework to manage conversion between some format and Java objects.

The `SerializedMessageBodyReaderAndWriter` is already registered on the client-side in class `ConcertResourceIT`. For the server-side, you'll need to handle registration in your `Application` subclass. To do this you should override `Application`'s `getClasses()` method to return a `Set` that contains the `SerializedMessageBodyReaderAndWriter` class.

```
private Set<Class<?>> _classes = new HashSet<Class<?>>();

public ConcertApplication() {
  ...
  _classes.add(SerializationMessageBodyReaderAndWriter.class);
}

@Override
public Set<Class<?>> getClasses() {
  return _classes;
}
```

When the JAX-RS runtime loads your `Application` subclass on startup, it calls both the `getSingletons()` and `getClasses()` methods. It instantiates classes obtained from `getClasses()` as necessary, and in the case of `SerializedMessageBodyReaderAndWriter`, it uses an instance to convert Java objects to and from their bytecode representations when reading and writing HTTP messages. Note that your `Application` subclass must also override `getSingletons()` to return an instance of `ConcertResource` – as discussed in lectures.

Using the registered `MessageBodyReader` and `MessageBodyWriter` support for the Java serialization format means that you don't need to manually parse and generate the payload data for the HTTP request and response messages (as with project `softeng325-lab2-parolee`). The `Concert` objects being exchanged between consumers and the service will automatically be converted to their byte-representation. Despite being a text-based protocol, HTTP can store binary data in HTTP message bodies.

`SerializedMessageBodyReaderAndWriter` defines a new MIME type:

```
application/java-serialization
```

The client (`ConcertResourceIT`), when preparing a Web service request, specifies this as its preferred content type. Similarly, in `ConcertResource`, you should use appropriate metadata annotations to specify that methods that produce or consume content do so using this data format. As you can see in `ConcertResource`, the method signatures are written to use class `Concert` directly – unlike the `softeng325-lab2-parolee` project there's no need to use stream I/O.

*Stateless communication protocol*
As discussed in class, HTTP is a stateless protocol. A JAX-RS Web service thus processes each incoming request without knowledge of earlier requests. The Concert Web service is required to track each client that is making requests of the service, and should recognise when requests are being made by the same client. Hence, each client is to be assigned a unique identity by the service, and should include this identity as part of each HTTP request.

In developing a client-tracking mechanism, you should use HTTP cookies. The first time a client sends a request to the Web service, it won't have a cookie – in which case the service should

generate one and return it in the HTTP response corresponding to the request. On all subsequent invocations, the HTTP request message should contain the cookie, thereby identifying the client that made the request. In a Web service that stores data for clients in a database, the cookie value could be used to retrieve data associated with the requesting client (more on the use of databases for persistence later).

*Suggestions and resources*
In developing class `ConcertResource`, you should read the JAX-RS API Javadoc, in particular for:

- `javax.ws.rs.core.Response`,
- `javax.ws.rs.core.Response.Builder`
- `javax.ws.rs.core.Cookie`
- `javax.ws.rs.core.NewCookie`

Studying the `softeng325-lab2-parolee` project will be helpful too. In addition, have a look at the `ConcertResourceIT` class, and be sure to read the comments in the `ConcertResource` class.

Javadoc documentation for JAX-RS (which is part of Java Enterprise Edition) is available at:

`https://docs.oracle.com/javaee/7/api/`

## Assessment and submission

Run Maven's `clean` goal on the project to clear all generated code. Zip up the project and upload it to the Assignment Drop Box (https://adb.auckland.ac.nz).

The submission deadline is 18:00 on Friday 11 August. Participating in this lab is worth 1% of your SOFTENG 325 mark.