SOFTENG 325 Software Architecture
Lab 4

Ian Warren

August 13, 2017

The purpose of this lab is to further prepare for the main assignment. You should review a minimal JPA ORM project, develop a stateless Web service that uses a database rather than memory for storing entity data, and develop an Amazon Web Services (AWS) client to fetch data from the AWS S3 cloud service.

Tasks

Task 1: Review the ORM Hello World application

Project `softeng325-lab4-orm-hello-world` is a simple JPA application that persists `Message` instances. `softeng325-lab4-orm-hello-world` was briefly presented in lectures. It's a complete project that includes the following necessary elements for a JPA project:

- Class `Message`. This is a simple entity class comprising a couple of primitive fields that is to be persisted.
- Configuration file `persistence.xml`. Stored in `META-INF` by convention, this file defines a JPA *persistence unit*.
- Class `HelloWorldJpaTest`. A JUnit test that demonstrates how to use JPA to persist and query `Message` instances.

In addition, the project includes:

- Class `HelloWorldJdbcTest`. This is similar to `HelloWorldJpaTest`, but shows how Java's more primitive JDBC API can be used to persist `Message` instances.
- Class `DatabaseUtility`. A utility class that provides methods for opening and closing a database connection, executing SQL requests, and outputting the contents of database tables. `HelloWorldJpaTest` and `HelloWorldJdbcTest` make use of this class. `DatabaseUtility` can also be used as an alternative to the H2 console for displaying the contents of tables.

(a) Import, build and run the project

Project `softeng325-lab4-orm-hello-world` is a basic Maven project. Simply import the project into your Eclipse workspace and run the `test` goal to run the two unit tests. Both tests use the `DatabaseUtility` class to output the state of the database at the end of their execution. The tests should run without error.

(b) Reflect on the project

Compare and contrast the JPA and JDBC solutions.

Task 2: Develop a stateless Concert Web service that uses ORM

Develop a JAX-RS Web service for managing Concerts. The service should be stateless and use JPA to persist domain objects to a database.

The service is to provide a basic REST interface as follows:

- `GET /concerts/{id}`. Retrieves a representation of a `Concert`, identified by its unique ID. The HTTP response message should have a status code of either 200 or 404, depending on whether the specified `Concert` is found.
- `POST /concerts`. Creates a `Concert`. The body of the HTTP request message contains a representation of the new `Concert` (less unique ID) to create. The service generates the `Concert`'s ID via the database, and returns a HTTP response of 201 with a `Location` header storing the URI for the newly created `Concert`.
- `PUT /concerts`. Updates an existing `Concert`. A representation of the modified `Concert` is stored in the body of the HTTP request message. Being an existing `Concert` that was earlier created by the Web service, it should include a unique ID value. The HTTP status code should be 204 on success, or 404 where the `Concert` isn't known to the Web service.
- `DELETE /concerts{id}`. Deletes a `Concert`, where the `Concert` to delete is specified by a unique ID. This operation returns either 204 or 404, depending on whether the `Concert` exists.
- `DELETE /concerts`. Deletes all `Concerts`, and returns a 204 status code.

The Web service is to store `Concert` and `Performer` entities. Similarly to Lab 3's DAO task, a `Concert` has a `Performer`, and a `Performer` may perform at many `Concerts`. For the REST interface, you don't need to use DTO classes – `Concerts` and `Performers` are sufficiently simple and involve little data, so when a `Concert` is being exchanged between a client and the Web service, it should include its `Performer`. Concerning representation format, the service need only support XML.

A partially complete project named `softeng325-lab4-concert` is available to download from Canvas. The project is a multi-module project with `softeng325-lab4-concert-domain-model` and `softeng325-lab4-concert-web-service`. The parent project's POM file declares dependencies common to the two modules. The `softeng325-lab4-concert-domain-model` project includes complete implementations of the `Concert` and `Performer` classes. It also contains converter classes that are useful during XML marshalling/unmrshalling and for persisting/retrieving data. Project `softeng325-lab4-concert-web-service` includes a JPA configuration file (`persistence.xml`), a class named `PersistenceManager` and an integration test class.

(a) Import the project

Import the project into your Eclipse workspace, as you would a multi-module project.

(b) Develop the project

To complete the project, you need to do the following:

- Add all necessary metadata to the domain classes `Concert` and `Performer`. You should use appropriate JAXB annotations to specify how instances are to be marshalled/unmarshalled to/from XML. Similarly, you should add JPA annotations to specify how `Concert` and `Performer` instances should be persisted.
- Introduce a subclass of `javax.ws.rs.core.Application`. This has two start-up responsibilities. First, it should return a *resource-per-request* handler for processing HTTP requests. Second, it should instantiate the supplied `PersistenceManager` class.

- Implement the `Resource` class. As with any JAX-RS application, you need a `Resource` class that performs the actual processing of requests.

A key difference between this Web service and the earlier ones that you've developed is that this one should be stateless – with all `Concert` and `Performer` data stored in a database. Rather than returning a singleton `Resource` handler ***object***, the `Application` subclass should return a `Resource` handler ***class***. Where a `Resource` handler class is returned, the JAX-RS run-time instantiates it every time it's needed to process an incoming request. This is known as *resource-per-request* handling. It differs to the *singleton-resource* handler that has been used until now, where one `Resource` object is created on start up and used to process all requests. Where a Web service stores state, obviously the same `Resource` object must be used to process all requests since the `Resource` object updates its in-memory state each time. Now that the Web service is to be stateless, the *resource-per-request* model is more appropriate; the `Application` subclass' `getClasses` method should return the `Resource` class.

Class `PersistenceManager` is a singleton class that provides the means to create an `EntityManager`. An `EntityManager` essentially provides the JPA interface for reading and writing persistent objects, and is required by the `Resource` class to retrieve and store data in the database. At construction time, the `PersistenceManager` creates a JPA `EntityManagerFactory` that reads the `persistence.xml` file, scans the entity classes for mapping metadata, and generates the relational schema. Theses action should be performed ***once***, on application startup (hence it's convenient for the `Application` subclass' `getSingletons()` method to instantiate the `PersistenceManager`).

In adding JPA metadata to the `Concert` and `Performer` classes, you should use both the `@Id` and `@GeneratedValue` annotations on the identity field of the `Concert` and `Performer` classes. The effect of `@GeneratedValue` is to ensure that the database is responsible for generating primary key values. You should also ensure that when a `Concert` is persisted, so too is its `Performer` – automatically. Similarly, when updating or deleting a `Concert`, the associated `Performer` should also be updated or deleted automatically. `Concert` has a `LocalDateTime`[1] field that JPA cannot readily persist. Since the JPA specification was released prior to Java 8, JPA implementations don't recognise Java's new date/time classes. Hence a converter has to be used when persisting and retrieving the new date/time objects. See the Javadoc for the `@Convert` annotation (`javax.persistence.Convert`) and the supplied `LocalDateTimeConverter` class in package `nz.ac.auckland.concert.domain.jpa`.

Similarly to converting `javax.LocalDateTime` objects for persistence, they must also be converted when being marshalled/unmarshalled to/from XML. The project includes a class named `nz.ac.auckland.concert.jaxb.LocalDateTimeAdapter` for this purpose. The `package.info` file in package `nz.ac.auckland.concert.domain` includes a `XmlJavaTypeAdapter` declaration that registers the `LocalDateTimeAdapter`. It will automatically be applied on all fields of classes defined in the `nz.ac.auckland.concert.domain` package – hence you don't need to annotate individual fields in `nz.ac.auckland.concert.domain` classes for the adapter to be used.

In implementing the `Resource` class' handler methods, you'll need to use the `EntityManager`. The typical usage pattern for `EntityManager` is shown overpage. Particular `EntityManager` methods that will be useful for this task include:

- `find(Class, primary-key)`. `find()` looks up an object in the database based on the type of object and primary-key value arguments. If a match is found, this method returns an object of the specified type and with the given primary key. If there's no match the method returns null.

- `persist(Object)`. This persists a new object in the database when the enclosing transaction commits.

- `merge(Object)`. A `merge()` call updates an object in the database. When the enclosing transaction commits, the database is updated based on the state of the in-memory object to which the `merge()` call applies.

[1]Previous projects have used the Joda date/time classes. As of Java 8, the JDK includes similar classes that make use of the Joda library redundant.

- `remove(Object)`. This deletes an object from the database when the transaction commits.

For this task, the above `EntityManager` methods are sufficient. The `EntityManager` interface will be discussed in more detail later; for more information in the meantime consult the Javadoc for `javax.persistence.EntityManager` (`https://docs.oracle.com/javaee/7/api/`).

A simple JPQL query to return all `Concerts` might be useful for this task, and this can be expressed simply as:

```
EntityManager em;
TypedQuery<Concert> concertQuery =
   em.createQuery("select c from Concert c", Concert.class);
List<Concert> concerts = concertQuery.getResultList();
```

JPQL will be introduced later. For further information, the Java Enterprise tutorial includes a chapter on JPQL:

```
https://docs.oracle.com/javaee/7/tutorial/persistence-querylanguage.htm
```

EntityManager usage scenario

```
// Acquire an EntityManager (creating a new persistence context).
EntityManager em = PersistenceManager()
   .instance()
   .createEntityManager();

// Start a new transaction.
em.getTransaction().begin();

// Use the EntityManager to retrieve, persist or delete object(s).
Object object;
em.persist(object);

// Commit the transaction.
em.getTransaction().commit();
```

(c) Run the project

Once you've added the metadata, `Application` and `Resource` classes, build and run the project. The supplied integration tests should pass with a correct Web service implementation.

You might want to use the H2 console so see the tables that have been generated and populated.

(d) Reflect on the project

Compare your JPA solution for persisting `Concert` and `Performer` objects with that of the DAO class you completed in Lab 2. How do they compare? What does the ORM solution provide that Lab 2's DAO class lacks? With regard to Web services, what are the benefits to having stateless services?

Task 3: Develop an Amazon Web Services client

Amazon Web Services (AWS) is a company that provides several Cloud-based services. At the core are elastic compute services (EC2) and data storage services (S3). With elastic compute services, applications run on leased computational resources. By being elastic, the compute services can dynamically scale up and down as demand for an application changes. Essentially, additional machines can be acquired when required to host service replicas (e.g. stateless Web services), and released when demand drops. The Amazon Web services infrastructure handles load balancing and routing requests through to different replicas. For data storage, Amazon's S3 service provides large-scale data storage. S3 is particularly well suited to storing immutable data, e.g. images, videos, and backup data. The S3 service also uses replication to ensure data availability.

For this task, you want to download concert performer images that are stored in S3. S3 exposes a Web service interface, but there's an AWS Java library that abstracts the interface. The POM for the supplied `softeng325-lab4-aws` project includes the necessary dependency.

Project `softeng325-lab4-aws` is largely complete – you just need to add a couple of methods to find the names of the stored images and then download them. The following on-line documentation describes how to use the S3 API to list the contents of a storage bucket and download its objects synchronously.

`http://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/examples-s3-object-s.html`

If you want to download images asynchronously, without blocking, use the `TransferManager` class, described at:

`http://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/examples-s3-transf-ermanager.html`

The preferred way to create a `TransferManager` is using a `TransferManagerBuilder`:

```
TransferManager mgr = TransferManagerBuilder
   .standard()
   .withS3Client(s3)
   .build();
```

where `s3` is an `AmazonS3` instance.

When you've completed the program, it should download 21 images.

Assessment and submission

Run Maven's `clean` goal on the Task 2 and 3 projects to clear all generated code. Zip up the projects and upload the archive to the Assignment Drop Box (https://adb.auckland.ac.nz).

The submission deadline is 18:00 on Friday 25 August. Participating in this lab is worth 1% of your SOFTENG 325 mark.