SOFTENG 325 Software Architecture
Assignment 1

Andrew Meads, Ian Warren

August 15, 2018

The aim of this assignment is to build a Web service for concert booking.

1 System description

The booking service is required for a small venue, with a capacity of 374 seats. The venue's seats are classified into three price bands, A, B and C. A concert may run on multiple dates, and each concert has at least one performer. Additionally, a performer may feature in multiple concerts.

The service is to allow clients to retrieve information about concerts and performers, to make and enquire about reservations, to register and authenticate users, and to download performer images. On making a reservation, the service randomly selects seats in the required price band (if available) and holds them for a short period. If the reservation is confirmed within this period, the seats are booked. Without any confirmation, the service frees the seats so that they can be reserved and booked by others. To make and confirm reservations, clients must be authenticated and the associated user must have a credit card registered with the service.

In addition, the service should allow clients to subscribe to news reports concerning concerts and performers. Once subscribed, clients will be notified of relevant news items, e.g. scheduling of particular concerts, notifications for when tickets are going on sale, news stories about performers etc.

A particular quality attribute that the service must satisfy is scalability. It is expected that the service will experience high load when concert tickets go on sale.

2 Tasks

2.1 Construct a domain model

Develop an object-oriented domain model for the concert service. The model should include appropriate classes and relationships that allow concerts, performers, reservations and users to be represented. The classes should be suitably annotated with JPA annotations to define their mapping to a relational schema.

2.2 Develop a REST-based Web service

Develop a JAX-RS Web service that exposes the required functionality. In developing the service, you should define an appropriate REST interface in terms of resource URIs and HTTP methods.

2.3 Implement a client interface

A regular Java interface named `ConcertService` (see Section 3) has been defined for use by clients. You should implement the interface according to its Javadoc contract. For full marks, the optional methods need to be implemented.

3 Resources

A multi-module Maven project, `softeng325-concert`, is supplied and comprises 3 modules:

- `softeng325-concert-client`. A module that contains resources for building a concert client application.
- `softeng325-concert-common`. This module defines several entities that are shared by the `client` and `service` modules.
- `softeng325-concert-service`. The `service` module is responsible for implementing the Web service and persistence aspects of the concert application.

The parent project's POM includes common dependencies and properties that are inherited by the 3 modules.

3.1 Client module

The `client` module contains 2 packages. Package `nz.ac.auckland.concert.client.service` contains the `ConcertService` interface, a related exception class (`ServiceException`) and a skeleton implementation of the service interface named `DefaultService`. You need to flesh out and fully implement class `DefaultService`. Package `nz.ac.auckland.concert.client.clientApp` is a placeholder for a Swing client – that will be made available shortly.

Class `nz.ac.auckland.concert.client.service.ConcertServiceTest` provides several tests that check conformance of a `DefaultService` instance with the `ConcertService` contract. You should study the Javadoc for `ConcertService` prior to implementing `DefaultService`.

The `client` module uses the embedded servlet container (Jetty) used in the earlier labs, but in a different way. Instead of having Maven startup the servlet container prior to running integration tests and shutting it down afterwards, class `ConcertServiceTest` launches the servlet container directly. Specifically, `ConcertServiceTest` starts the container immediately before running each test and shuts it down after each test. This ensures that there are no side effects from running tests, and gives the Web service application an opportunity to reinitialise itself when being loaded by the container.

Other than a POM file that declares client-specific dependencies, the `Client` module includes a `log4j.properties` file. You'll likely want to change the logging levels for different namespaces during development and debugging.

3.2 Common module

This module includes a number of packages:

- `nz.ac.auckland.concert.common.dto`. This packages contains complete implementations for 9 DTO classes. The `ConcertService` interface from the `client` module is defined in terms of the DTO classes. In addition, this package includes a `package-info` file that registers JAXB adapters on all occurrences of Java 8's `LocalDate` and `LocalDateTime` classes used by the DTO classes.
- `nz.ac.auckland.concert.common.jaxb`. This package includes the JAXB adapter implementations for Java 8's `LocalDate` and `LocalDateTime` classes.
- `nz.ac.auckland.concert.common.message`. Package `message` includes one class that defines status and error strings. Class `DefaultService` is expected to throw `ServiceException` objects whose message values are defined by class `Message`.
- `nz.ac.auckland.concert.common.types`. This package defines basic data types that are common to the `client` and `service` modules. The types comprise 3 enumerations: `Genre` (for performers), `PriceBand` (for ticket pricing) and `SeatRow` (for concert venue rows). It also includes a `SeatNumber` type for representing seat numbers at the venue.

- `nz.ac.auckland.concert.common.util`. Package `util` contains class `TheatreLayout`, which models the layout of the concert venue. It includes methods to find how many seats are in a given row, and which rows fall into which price bands.

3.3 Service module

The `Service` module is organised into 3 packages. `nz.ac.auckland.concert.service.services` is intended to contain the Web service implementation, in terms of a `javax.ws.rs.core.Application` subclass, a resource class that implements the HTTP request handlers, and a `PersistenceManager`. Currently, an empty `Application` subclass named `ConcertApplication` is supplied along with the `PersistenceManager` singleton, introduced in Lab 4.

Package `nz.ac.auckland.concert.service.domain` is intended to store the concert domain-model classes. Package `nz.ac.auckland.concert.service.domain.jpa` contains a class named `SeatNumberConverter`. This is useful if you define a domain class that has a field of type `SeatNumber`. The converter allows `SeatNumber` instances to be written to and read from a database as regular Integer data.

The `Service` module also contains package `nz.ac.auckland.concert.service.util`, in which class `TheatreUtility` is located. You can use this class to help find available seats when making a reservation.

Beyond packages and source files, the `Service` module contains the JPA `persistence.xml` file, a database initialise script (`db-init.sql`) and a `log4j.properties` file. This module's POM file includes the necessary configuration to run any integration tests after starting up an embedded servlet container hosting the Web service (similarly to the Web service modules developed in Labs 1 to 4). However, there are no integration tests supplied for the Web service. You don't need to write any tests within the `Service` module – because you can use the supplied `ConcertServiceTest` class from the `Client` module to drive testing of the Web service. That said, if you want to test your Web service as you have done for the Web services from Labs 1 to 4, you simply need to add integration tests to the `Service` module.

You'll need to edit the `persistence.xml` file to specify the names of your domain model classes that should be mapped to a relational schema (refer back to Lab 4). The `persistence.xml` file includes a `javax.persistence.sql-load-script-source` element whose value is the supplied `db-init.sql` script. The effect of this element is to run the script when the `EntityManagerFactory` is created. The `db-init.sql` file populates the database with concert and performer data. Depending on the relational schema you generate from your domain model, you may be able to use the supplied `db-init.sql` file, or you may need to edit it to fit with your schema.

4 Constraints

You are not permitted to change the `ConcertService` interface. Since this interface relies on the DTO classes in package `nz.ac.auckland.concert.common.dto`, and the data types in package `nz.ac.auckland.concert.common.types`, you cannot change these entities either – except to add any metadata annotations.

Your solution must load the concert and performer data as described by the `db-init.sql` file (i.e. you must have the 25 performers, 25 concerts, all concerts/performer pairings specified and all 75 concert dates). Your solution may use a different relational schema to what's expected by `db-init.sql`, but it should work with the dataset supplied (if your schema is different you'll need to edit the `db-init.sql` file for the data to load into your tables).

While your Web service doesn't need to be tested with concurrent clients, it **does** need to be implemented in such a way that it could be used by concurrent clients without comprising data integrity.

Finally, your solution must function, without modification, with the supplied `ConcertServiceTest` class.

5 Hints and suggestions

1. *Domain model design.* In designing the domain model, you should consider its impact on scalability. When the Web service is under heavy load by users wanting to book concerts, the domain model shouldn't constrain throughput because of coarse-grained concurrency control schemes.

2. *Domain modelling of concerts and performers.* The supplied `db-init.sql` file populates relational tables with data about concerts information (including dates and prices), performers, and which concerts feature which performers. To use the script without needing to modify it, you might want to work backwards from the tables and define `Concert` and `Performer` entity classes, annotated appropriately, such that the JPA provider generates the schema expected by the `db-init.sql` script. Your domain model would then need additional classes to fully meet the system's requirements.

3. *Initialising the database.* As discussed in Section 3, the `service` module includes the JPA configuration file `persistence.xml` that arranges for the `db-init.sql` script to run during creation of the `EntityManagerFactory`. Since the `EntityManagerFactory` is created by Singleton class `PersistenceManager`, the script is executed only *once*, when the `PersistenceManager` is instantiated. When `ConcertServiceTest` runs, restarting the servlet container between each test, it does **not** re-create the `PersistenceManager` – and so the `db-init.sql` isn't executed between tests. This is fine because the script loads only concert and performer data that isn't modified when booking tickets. To clear the effects on the database of running tests (e.g. tests that make reservations, create users, register credit cards etc.), the `Application` subclass (`ConcertApplication` in the `service` module) can be implemented to acquire a reference to the `PersistenceManager`, and remove all entities of a given type, e.g. reservations, upon start up. See Appendix B for a code fragment.

4. *Managing memory for the persistence context.* Depending on how you tackle the assignment, your solution might involve creating and persisting many (thousands of) entities. In this case you should periodically clear the persistence context during the entity creation/persistence operation. Using `EntityManager`'s `flush()` method causes all pending `persist()` requests to be effected, and `clear()` removes the entities from the persistence context – freeing memory. The code fragment in Appendix B illustrates use of `flush()` and `clear()` for this purpose.

5. *Token-based authentication.* Some of the `ConcertService` methods require an authenticated user. Authentication should be implemented using *token-based authentication*, which is where the client exchanges credentials (username and password) for a token. Once authenticated, the client sends the token with each subsequent request to the Web service. For any requests requiring authentication, the Web service checks for the presence of the token. If the token's missing, the request fails; if the token doesn't identify a user (based upon accessing storage to lookup any user to who the token is associated), the request fails. Where the token is bound to a particular user, the user is then authenticated, and the Web service may then determine whether the user is authorized to make the request. For the concert booking service, there are no authorization levels – any authenticated user can legitimately make any request that requires authentication. For this assignment, you don't need to use HTTP's specialist headers for authentication – refer back to Lab 3 to see how you can implement token-based authentication.

6. *Returning appropriate HTTP status codes.* There are several HTTP status codes that are defined by the HTTP specification. While there isn't complete agreement on which codes to use in REST-based Web services, the following conventions are widely accepted.

    - `200 OK`. This is appropriate where a request has been successfully processed, and the HTTP response message's body contains entity data.

- `201 Created`. A `201 Created` response is appropriate where a request to create a new resource has been successful. The response message should contain a `Location` header storing the URI of the newly created resource.
- `204`. Similar to `200 OK`, but where the corresponding HTTP response message doesn't contain entity data.
- `400 Bad Request`. This is suitable where a client makes a bad request, e.g. where a DTO argument is expected to have particular fields filled in but doesn't, or where the combination of field values don't make sense, e.g. a concert reservation request specifies a date when the concert isn't scheduled.
- `401 Unauthorized`. In practice, a `401 Unauthorized` status code is used to indicate that the requesting client/user was not *authenticated* (as opposed to not *authorized*, e.g. based on having insufficient access rights). Where a Web service request that requires an authentication token lacks the token, a `401` code is thus appropriate.
- `404 Not Found`. A `404 Not Found` is appropriate where a named resource doesn't exist in the Web service.
- `500 Internal Server Error`. Where an error other than an application condition has occurred in processing a Web service request, a `500` status code is appropriate.

7. *Returning error messages.* The `ConcertService` contract requires that any implementation returns particular error messages (defined in class `Messages` from the `common` module) in `ServiceExceptions`, depending on the specific outcome of processing a `ConcertService` request. The tests in `ConcertServiceTest` check for the specified error messages. When implementing your Web service, you'll find it useful to be able to return `Messages` messages in `Response` objects. Appendix C includes a code fragment showing how you can use the JAX-RS API to throw a `WebApplicationException` (that is automatically caught and handled by the JAX-RS runtime) and which sets a given error message as the entity data to be communicated back to the client. Appendix C also shows how the error message can be extracted from within a client when processing a `Response` object.

8. *Fetching images.* Performer images named in `db-init.sql` are hosted by AWS S3. You can fetch the images using the same credentials and bucket identifier as used in Lab 4.

9. *Building the project and running the supplied tests.* The easiest way to build the project is to run the Maven `compile` goal on the parent project (`softeng325-concert`). You can then run the `ConcertServiceTest` class from within Eclipse.

6 Submission

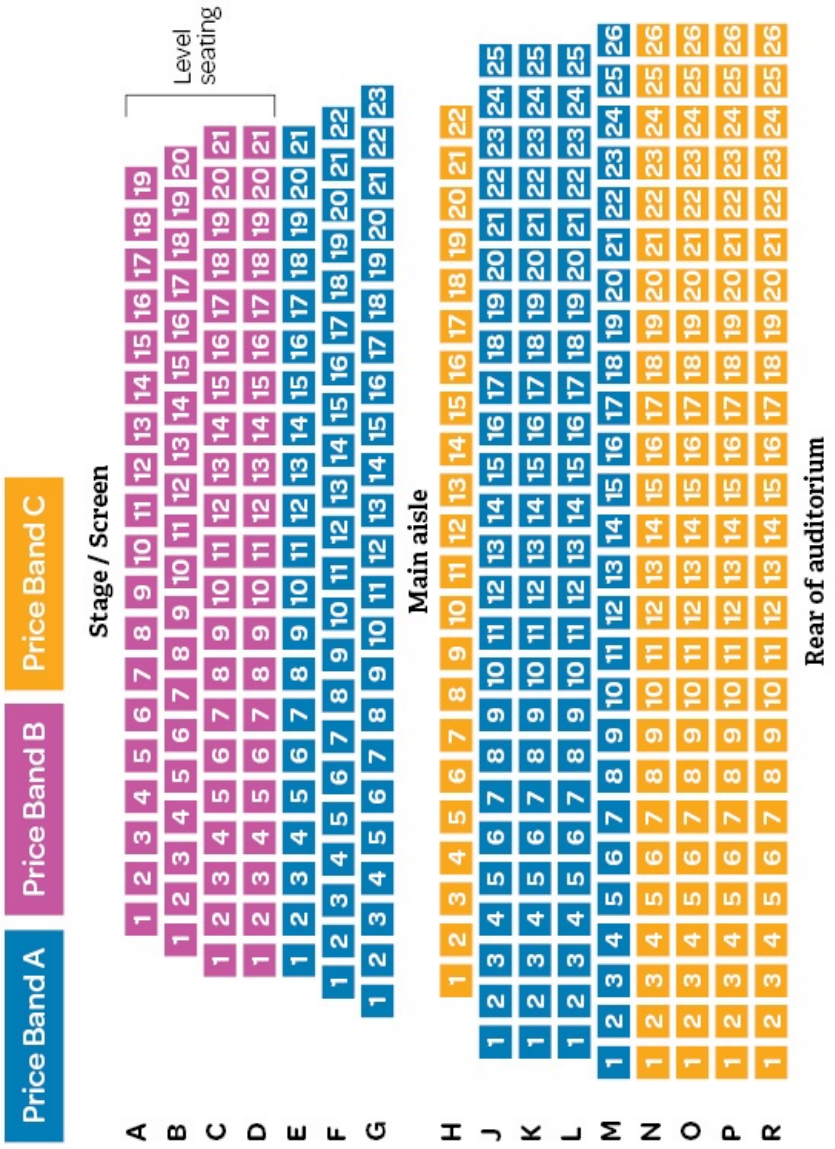
You should submit the following in a single archive (.zip or .7z) to the ADB:

1. *Report.* You should write a report, not exceeding **one** page, that describes the extent to which your solution meets the system's *scalability* requirement. You should highlight aspects of your Web service's design that contribute towards meeting this quality attribute.

2. *Source code.* You should include the full multi-module `softeng325-concert` project. You should prepare the project by running Maven `clean` on the project prior to adding the code to the archive.

The submission deadline is 18:00 on Monday 17th September. Assignment 1, including Labs 1 to 4, is worth 20% of your SOFTENG 325 mark (1% for each lab, 16% for the main assignment). Submission is via Canvas upload, as with the lab exercises.

Appendix A: Venue layout



Appendix B: Initialising the database

```
1  public class ConcertApplication extends Application {
2
3    // Constructor called by JAX-RS.
4    public ConcertApplication() {
5      EntityManager em = null;
6
7      try {
8        em = PersistenceManager.instance().createEntityManager();
9        em.getTransaction().begin();
10
11       // Delete all existing entities of some type, e.g. MyEntity.
12       em.createQuery("delete from MyEntity").executeUpdate();
13
14       // Make many entities of some type.
15       for (...) {
16         for (...) {
17           MyEntity e = new MyEntity(...);
18           em.persist(e);
19         }
20         // Periodically flush and clear the persistence context.
21         em.flush();
22         em.clear();
23       }
24       em.getTransaction().commit();
25
26     } catch(Exception e) {
27       // Process and log the exception.
28     } finally {
29       if (em != null && em.isOpen()) {
30         em.close();
31       }
32     }
33   }
34 }
```

Appendix C: Communicating error messages as entity data

Throwing a WebApplicationException with a custom error message

```
/*
 * From a WebService HTTP handler method, in detecting that a request described
 * by a ReservationRequestDTO to reserve seats specifies a date on which the
 * concert isn't scheduled, throw a BadRequestException. In creating the
 * BadRequestException, supply a Response object whose status code is 400 Bad
 * Request and whose entity data is a message string, defined by Messages.
 * The Response object contained within the exception is returned to the client.
 */
throw new BadRequestException(Response
      .status(Status.BAD_REQUEST)
      .entity(Messages.CONCERT_NOT_SCHEDULED_ON_RESERVATION_DATE)
      .build());
```

Processing a Response containing a custom error message

```
// Use a Builder to make a Web service request and return its Response.
Builder builder = ...;
Response response = builder ...;

// Get the response code from the Response object.
int responseCode = response.getStatus();

/*
 * Process the response. If the response code is 400/401, expect the
 * Response's entity to be a string error message. For other
 * response codes, the entity might be a DTO that could be retrieved.
 */
switch (responseCode) {
  case 400: // BAD_REQUEST
  case 401: // UNAUTHORIZED
    String errorMessage = response.readEntity(String.class);
  case 201: // CREATED
    ReservationDTO reservation = response.readEntity(ReservationDTO.class);
}
```