Università di Trento

Web Architectures

Assignment 5

Author:
Marcel Pascal Stolin
marcelpascal.stolin@studenti.unitn.it

January 30, 2022

1 Introduction

The fifth assignment is about creating an application where tourists can book and aprtmand or a hotel room, depending on its availability. On this application, the user can search for accommodtions by giving the start- and end-date, and the number of persons. Then, the user should see a list of all available results, where the user can select one accommodation and confirm the reservation.

The application is composed of the following parts:

- Business-logic
- Web-application
- Database

The business-logic is supposed to be implemented using EJB and EJB-patterns. All EJBs must be deployed on a wildfly application server. The web-application must be implement using Servlets. It should not be deployed using wildfly and must run outside of the wildfly application server. H2 must be used as the database. A routine must be implemented to write default data to the database.

2 Implementation

This section describes the most important parts of the implementation of the application. It highlihtes the databse structure, and the behaviour of the Beans.

2.1 Database

2.1.1 Entities

Figure 1 visualizes the UML diagram of the entities, used for the database. Overall, the database is used to save the following information:

- Accommodations which represent either an apartment or a hotel
- Occupancy, represents the availability of an accommodation entity for a specific day
- Reservation which represent a single reservation made by a user

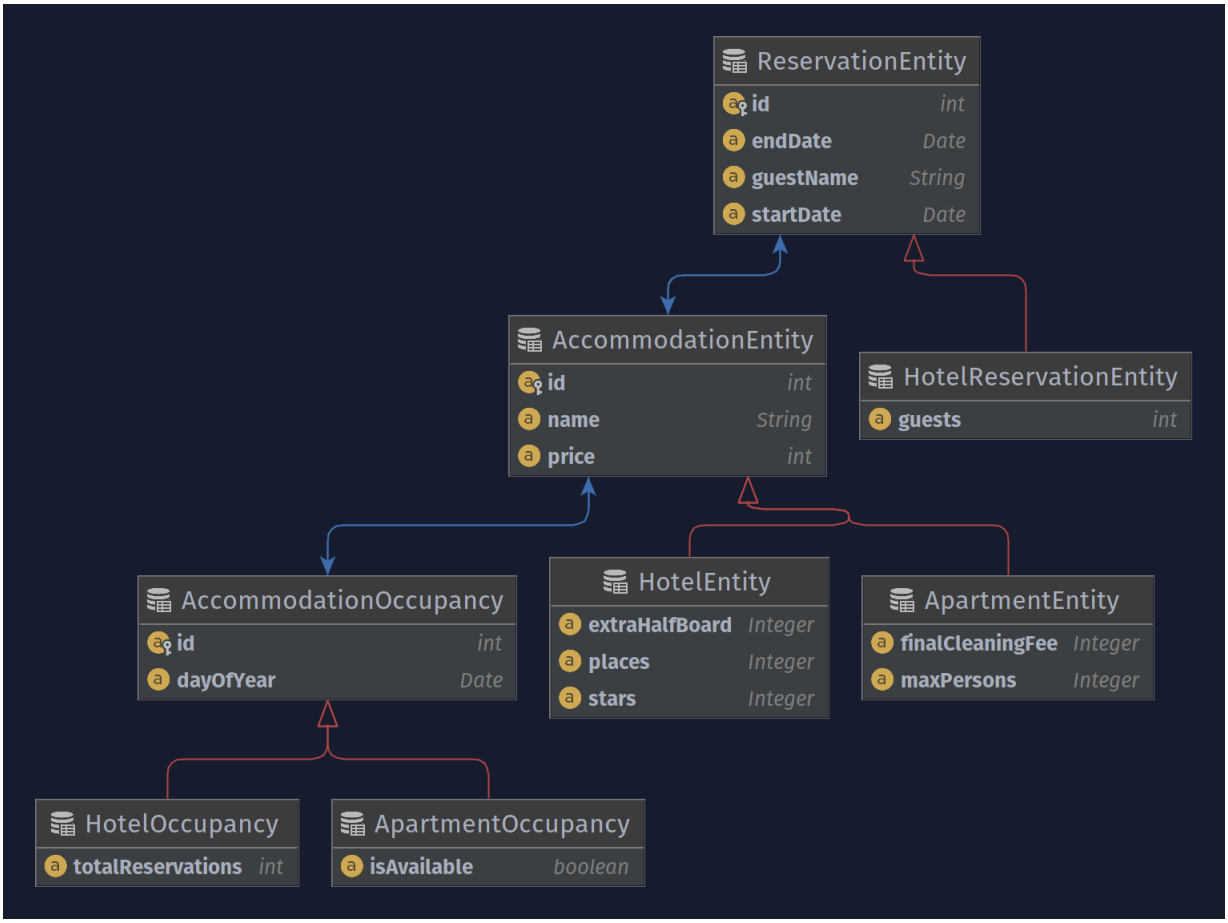

Figure 1: UML diagram of all database entities

Accommodations Two types of accommodations exists for this application: Apartments and Hotels. All accommadtions have a name, and a daily price. Additionally, an apartment has a final cleaning fee, an a number of maximum persons. A hotel has a rating of start, a number of free places, and a price for extra half-board.

To implement the entities, an ApartmentEntity, and a HotelEntity have been implemented. Both classes inherit from the abstract AccommodationEntity class.

Occupancies Each accommodation needs an occupancy information for a specific date. The occupancy information describes if an accommodation is available on a specific date. The difference between an apartment and a hotel is, that a hotel has a specific number of reservation, and an aprtment is either available or unavaible, independetly of the number of persons.

To implement the occupancies, a parent class called HotelOccupancy exists, that includes only the date. Additionally, a class called ApartmentOccupancy saves the occupancies information for a hotel, which includes a boolean value called isAvailable that specifies, if the aprtment is avaiable for the specific date or not. Furthemore, a class called HotelEntity represent the occupancy of a hotel, which is a number of reservation for a specific date.

Reservations A user can make reservation that are saved to the database. A reservation consists of a start date and an end date that represent the date interval of the stay, and the name of the guest. These informations are saved by the ReservationEntity. Additionally, a classes called HotelReservationEntity, which inherits the ReservationEntity, is used to save the number of guests for on reservation of a hotel. An apartment does not need extra information, because it is either avaible or not, which is represented by the existance of a ReservationEntity in the table for the specific apartment for a specific date interval.

2.1.2 Structure

The database consists of three tables:

- Accommodation
- Occupancy
- Reservation

Each table is associated with an entity that are described in SEC XY.

Accommodation FIG AB describes the structure of the Accommodation table. It is used to save the AccommodationEntity, as well as the child entities ApartmentEntity and HotelEntity. To accomplish inheritance, the Single Table per Class Hierarchy strategy is used.

Occupancy FIG BH describes the structure of the Occupancy table. It is used to save the OccupancyEntity and the HotelOccupancyEntity. The Single Table per Class Hierarchy strategy is used as well to enable inheritance.

Reservation FIG HJ describes the structure of the Reservations table. It saves the ApartmentEntity and the HotelEntity. For inheritance, it uses the Single Table per Class Hierarchy as well.

2.1.3 Database Routine

A database routine needs to be implemented to seed the database, that is supposed to be used in the application. This routine is not implemented as a EJB, instead, it will run as a standalone Java application. The accommodations, as well as the occupancies of accommodatios are given by the task description.

The following technologies are used to implement the database routine:

- Hibernate
- JPA

2.2 Enterprise Java Beans

The business logic of this application is implemented using EJB 3. It is composed of three parts:

- `LocalDatabaseBean`
- `AccommodationService`
- `ReservationService`

All services are part of a single Maven project called *WebServices*.

Each part (service) is an independent Baen. The client is able to lookup the Bean given an address and invoke methods on it. Only the Accommodation Service and the Reservation Service is available for the client. The Database Service is implemented using a local bean, because it is not intended, for the user, to interact with the database directly.

2.2.1 LocalDatabaseBean

The `LocalDatabaseBean` is used to interact with the H2 database. It initiates the `EntityManager`, given by JPA, bu establishing a connection to a JTA datasource. Other services can use the Database Service to send queries and receive results from the database.

As mentioned before, the Database Service is implemented as a `LocalBean`. Other Beans can interact with the Database Service using the Code-Injection pattern, which is shown in Listing 1.

```java
@Stateless
@Remote(ReservationService.class)
public class ReservationBean implements ReservationService  {

    @EJB()
    private LocalDatabaseBean databaseBean;

    ...

}
```

Listing 1: Usage of the `LocalDatabaseBean` using Code-Injection

2.2.2 AccommodationService

The Accommodation is used by the client to interact with the `AccommdationEntity` and its child classes `ApartmentEntity`, and `HotelEntity`. It uses the LocalDatabaseBean to send HQL queries to the H2 database.

Except from receiving accommodations based on specific properties, the most interesting part about the AccommodationService is how it receives available accommodations in a specific date range, given a number of guests. The idea is, that the number of occurencies of an accommadtion which is available in the specific date range, has to be qual the number of days of the given date range.

```
SELECT a.accommodation
FROM AccommodationOccupancyEntity a
WHERE (
  ((a.isAvailable IS TRUE AND a.accommodation.maxPersons >= 2)
  OR
  ((a.accommodation.places - a.totalReservations) >= 2))
)
AND a.dayOfYear BETWEEN '2022-02-01' AND '2022-02-09'
GROUP BY a.accommodation.id
HAVING COUNT(*) = 9
```

Listing 2: Example of a HQL query to receive all available accommodations

Listing 2 show an example HQL query implementation to get all available accommodations between 1. February 2022 and 10. February 2022 for 2 guests. As described before, the `AccommodationOccupancyEntity` saves the occupancies of each accommodation for specific dates. Therefore, it is necessary to check for apartments if it is available, and has enough places for the number of guests. For hotels, it is necessary to check if the existing number of reservations minus the available places of that hotel is lower or equal the number of guests. These constraints are checked for the range between the start date (1. February 2022) and the end date minus one day (9. February 2022). It is necessary to substract one day, because the guests will not stay for that day, and therefore the availability information for that day is unrelated. After that it is important, thte the count of numbers of the accommodation, in that specific date range, is euqal to the number of days between the start and end date.

2.2.3 ReservationService

To read from and write to the reservation table of the database, the ReservationService is used. It allows to persist new reservations to the database, and to get all reservations for a specific customer name.

When adding a new reservation to the database, the `ReservationService` is also responsible to update the occupancies (mentioned in XY) of the selected accommodation. Then, it is necessary to add the number of guests, of the new reservation, to each entry of the accommodation occupancy.

Additionally, the ReservationService provides an interface to calculate the price of a reservation for either a Hotel or an Apartment.

2.2.4 Build Process

The project is build using the `maven-ejb-plugin` maven plugin to create a EJB-Jar file. Then, the artifact can be built using the `mvn clean build` command.

Listing 3 shows the configuration of the `maven-ejb-plugin` plugin.

```
<packaging>ejb</packaging>
...
<build>
  <finalName>${artifactId}.${version}</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-ejb-plugin</artifactId>
      <version>3.1.0</version>
    </plugin>
  </plugins>
</build>
```

Listing 3: `maven-ejb-plugin` plugin configuration

2.3 Web Application

2.3.1 Search

2.3.2 Results

2.3.3 Reservation Summary

2.3.4 Reservation Confirm

2.3.5 My Reservations

3 Deployment

This section explains the deployment of the application. The requirements to run all applications are the following:

- H2
- Wildfly
- Java 11
- Tomcat
- Maven

3.1 Creating the Database

As mentioned before, this application uses a local H2 database, that is not integrated in the project itself, but is located somewhere on the users computer.

First, it is necessary to create a local database. Fig XY illustrates the process of creating a local database. First, it is necessary to change the directory to `H2_DIRECTORY/bin`. Then, execute the command `$ java -cp h2-*.jar org.h2.tools.Shell`. It is important, that the databse is called *accommodations*, and the username is *sa*, and the the password is *sa* as well.

After the database has been created, it is needed to start H2 via the terminal. To accomplish this, execute the command `$ java -jar h2*.jar` in the directory `H2_DIRECTORY/bin`. This process is shown in FIG AB. After that, the web browser should open the H2 console automatically. At the H2 console, it is possible to test the database connection to see if the database has been created successfully. First, it is necessary to put in the correct JDBC URL in the format `jdbc:h2:tcp://localhost/PATH_TO_DATABASE/accommodations`. The username is *sa*, and the password is *sa* as well. After that, by clicking on *Test Connection* it is possible to test the connection. Fig AB shows the message if the test was successful. After that, the database can be used in the application.

3.2 Seeding the Database

After the database has been created, it is important to write dummy data to it using the *DatabaseRoutine* application.

3.2.1 Set up the DatabaseRoutine application

It is important to set the path to the database in the persistance.xml of the *DatabaseRoutine* application. FIG AB shows a correct configuration. The property `hibernate.connection.url` has to be set to the URL used in SEC AB.

3.2.2 Execute the DatabaseRoutine application

To execute the DatabaseRoutine application, open the project in IntelliJ, right-click on the DataRoutine.java file, and select *Run 'DataRoutine.main()'*. Then, the file gets executes and write dummy data to the database.

After that, it is possible to check if the data has been written to the database by connecting to the database using the H2 console, introduced in SEC AB. FIG AB shows the newly created tables in the database.

3.3 Setting up Wildfly

At first, it is important to make sure, that Wildfly (mentioned in SEC AB) is available locally.

Second, to access the database, the path to the previously created H2 database need to be set as a datasource in the configuration file of Wildfly. The configuration file is located at `WILDFLY_DIRECTORY/standalone/configuration/standalone.xml`. LST AB shows how the in SEC AB created H2 database can be added as a datasource in Wildfly.

Third, the EJB's need to to be deployed to the Wildfly server. Therefore, it is necessary to run the command `$ mvn clean package` in the directory of the *WebServices* project. After that, a new directory called `/target` was created in the root folder of the *WebServices* project, where the EJB JAR `WebServices.1.0-SNAPSHOT.jar` is located, which is shown at FIG AB.

After the `.jar` artifact has been created, it need to be copied to the deployment directory of the Wildfly server. LST XY shows how to copy the artifact from the target directory to the deployment directory which is located at `WILDFLY_DIRECTORY/standalone/deployments`.

Finally, the Wildfly server can be started as shown in LST AB. Additionally, the log shows the lookup address of both the AccommodationService (mentioned in SEC XY) and the ReservationService (mentioned in SEC AB), which is illustrated in FIG AB.

3.4 Starting the Web Application

Before starting the web application it is important, that the following steps have been done in the given order:

1. Start H2 database
2. Add data to H2 database
3. Set up H2 database as a Wildfly datasource
4. Compile and deploy EJB JAR artifact to Wildfly
5. Start Wildfly application server

After those steps have been executed, the web application can be deployed and started using Tomcat.

First, it is necessary to create the `.war` artifact of the *WebApp* project, using the command `$ mvn clean package` which has to be executed in the *WebApp* projects root directory. After that, a new directory called `target/` has been created which contains the artifact `marcel-stolin-web-app.war`.

After the artifact has been built, it needs to be deployed to the `webapps/` directory of the Apache Tomcat directory using the command `$ cp target/marcel-stolin-web-app.wa APACHE_TOMCAT/webapps`.

One problem is, Apache Tomcat starts its service on port 8080 by default, which is also the default port of Wildfly. Therefore, the port of Apache Tomcat has to be changed to anaother port, e.g. 8000. The default port can be changed in the file `server.xml`, which can be seen in LST AB.

Finally, the Apache Tomcat server can be started using the command `$ APACHE_TOMCAT/bin/cat start`. After that, the web application is available at `http://localhost:8000/marcel-stolin-web-app/`, as shown in FIG XY.

4 Comments and Notes

This section describes the problems encountered during the development of the application.

4.1 Routing

A requirement was to deploy the *Angular* application on a *Tomcat* webserver. However, *Angular* redirects all requests to the `index.html`. For example, requesting the page `http://localhost:8080/marcel-stolin/detail/1735` will not work, because a directory `detail/1735` does not exist. Then, the *Tomcat* web server responses a 404 error page.

The solution to this problem, is to use the `HashLocationStrategy`[1]. Listing 4 shows how it is implemented in the application. Then, the previously mentioned URL can be accessed via `http://localhost:8080/marcel-stolin/#/detail/1735`.

```
const routes: Routes = [
  { path: '', component: MemberListComponent },
  { path: 'detail/:memberId', component: MemberDetailComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes, {useHash: true})],
  exports: [RouterModule]
})
```

Listing 4: Application routing configuration of `app-routing.module.ts`

4.2 Building the Angular Application

Another problem is how to build the *Angular* application. A *JS-Servlet* project requires static files (e.g.: HTML files, Javascript libraries) to be located at `PROJECT\_ROOT/src/main/webapp`. Therefore, whenever the angular application is built, the output has to be saved in this directory.

To solve this issue, *Angular* allows to define the `outputPath` for the application when using `ng build`. Listing 5 shows the `angular.json` configuration.

```
...
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
    "outputPath": "../../webapp",
...
```

Listing 5: `angular.json` configuration

[1] `https://angular.io/api/common/HashLocationStrategy`

Additionally, it is important to define the *base-href*, that defines the path where the application is located on the web server. This application is supposed to be available via the path `/marcel-stolin` (e.g.: `http://localhost:8080/marcel-stolin/`).

Therefore, the command `$ ng build --base-href /marcel-stolin/` has to be used to build the application.