Università di Trento

Web Architectures

Assignment 5

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

February 8, 2022

1 Introduction

The fifth assignment is about creating an application where tourists can book apartments or hotel rooms, given a start and end date, and a number of guests. This application is intended to be implemented using Enterprise Java Beans (EJB) which are supposed to contain the business logic of the application. Additionally, a web application and a database are part of this application. Another requirement of this application is, that the implementation should be done using common EJB patterns. To populate test data to the database, it is required to implement a routine that writes given accommodations, and random accommodation occupancies to the database.

2 Design

This section describes the most important parts of the design of the application. It highlights the database structure and its entities (Section 2.2), the purpose and behavior of the EJB's (Section 2.3), and the behavior of the servlets (Section 2.4).

2.1 Multimodule Project Design

This application is delivered as a multimodule project. It contains the *DatabaseRoutine*, *WebServices*, and *WebApp* projects, as shown in Figure 1. The *DatabaseRoutine* (introduced in Section 2.2.3) implements the routine to seed the database with test data, *WebServices* contains the business logic, and the *WebApp* project contains the servlets.

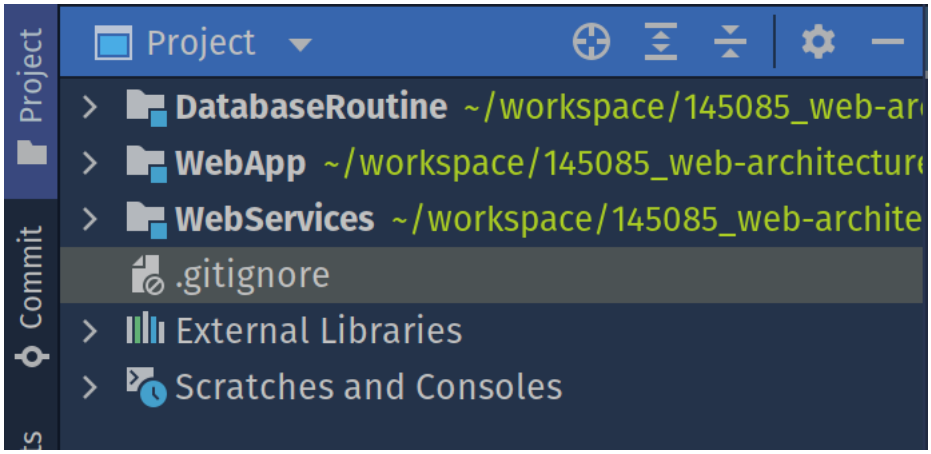

Figure 1: Multimodule project structure

2.2 Database

To persist all accommodations, the occupancy of accommodation, and reservations made by the users a database is needed. This application uses an H2 database.

2.2.1 Structure

The overall database structure, visualized in Figure 2, consists of the following three tables:

- *Accommodation*
- *Occupancy*
- *Reservation*

To enable polymorphism, the *Single Table per Class Hierarchy* strategy is used. This comes with a space inefficiency disadvantage, however, this application is not

intended to save a heavy amount of data. Therefore, it reasonable to use this strategy and have a time efficiency advantage.

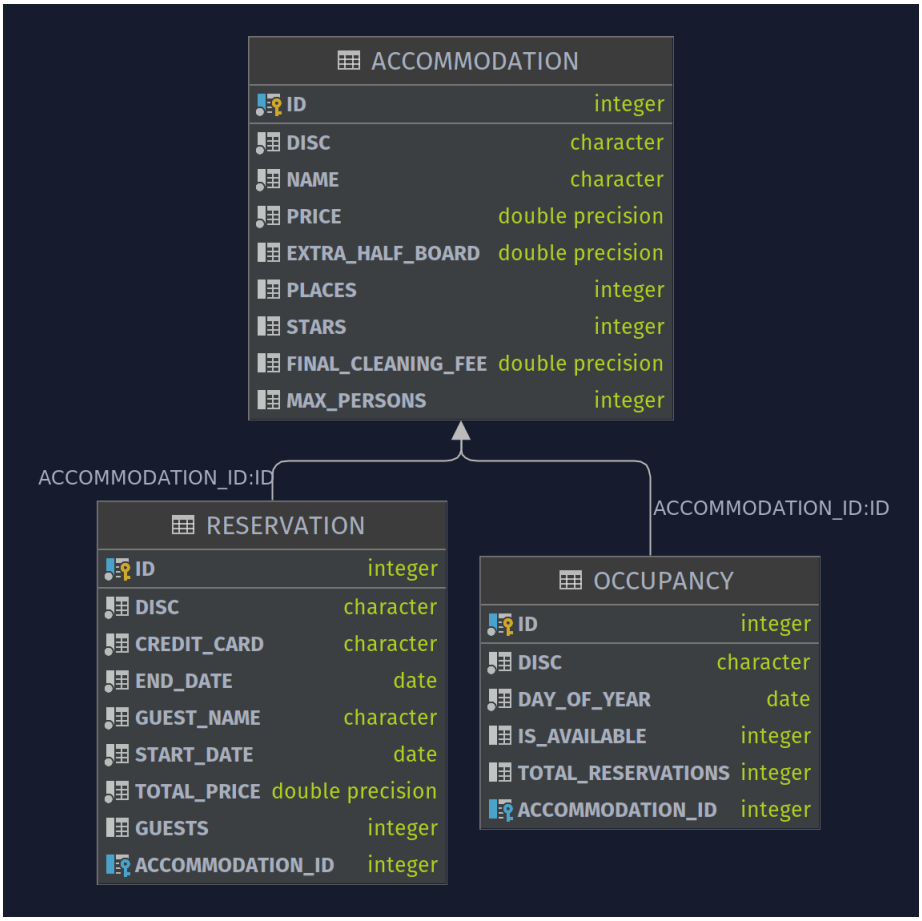

Figure 2: Diagram of the database strcture

Accommodation The *Accommodation* table is supposed to save all kinds of accommodation, apartments, and hotels. Both have a name and price. However, an apartment has an additional final cleaning fee, and a number of maximum persons allowed, whereas, a hotel has a rating of stars, a maximum number of places, and a price for an extra half-board.

Occupancy The *Occupancy* table saves the occupancy information for an accommodation for a specific date. An occupancy saves a date, and the specific accommodation, where one accommodation can have many occupancies. For an apartment, only a *is available* flag is important, and for a hotel, the number of total reservations for that date is needed.

Reservation The *Reservation* table saves all reservations made by all users. It saves the guest's name, the guest's credit card number, the start and end date, and the total price for the reservation. Additionally, for a hotel reservation, it is needed to save the number of guests as well. Accommodations and reservations are in a one-to-many relationship, where one accommodation can have many reservations.

2.2.2 Entities

To use the database in code, the application uses the Hibernate Object Relationship Mapping (ORM), in conjunction with the Java Persistence API (JPA). Figure 3 visualizes the UML diagram of the entities used to map the database structure (introduced in Section 2.2.1). Overall, the following entities are used in

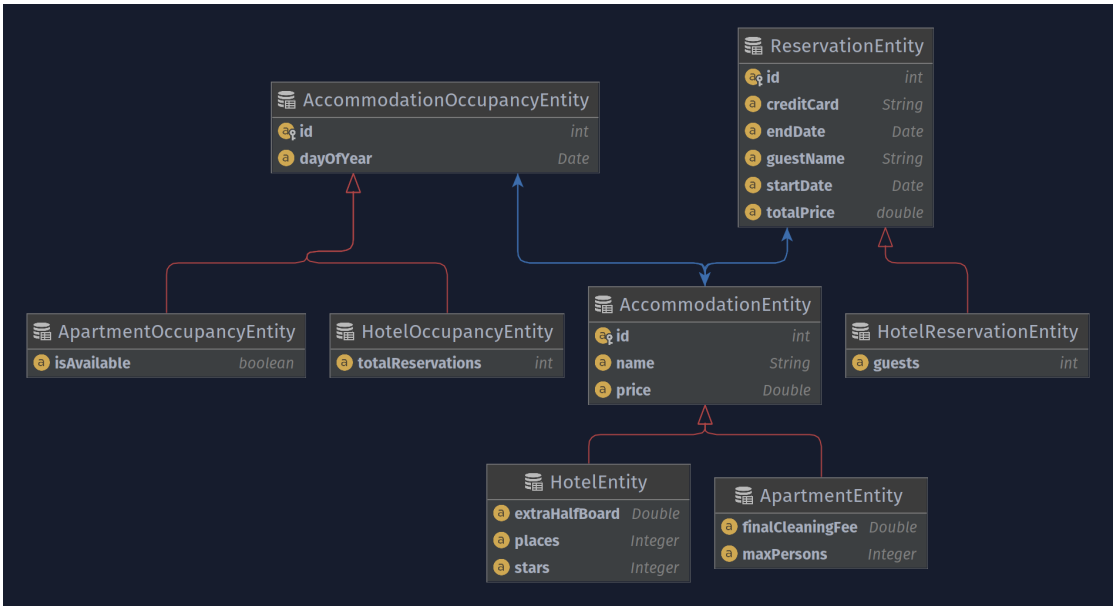

Figure 3: Database entities UML diagram

the implementation:

- *AccommodationEntity*
  - *ApartmentEntity*
  - *HotelEntity*
- *AccommodationOccupancyEntity*
  - *ApartmentOccupancyEntity*
  - *HotelOccupancyEntity*
- *ReservationEntity*
  - *HotelReservationEntity*

AccommodationEntity The *AccommodationEntity* represents the *Accommodation* table (introduced in Section 2.2.1). It is an abstract class and has two child classes, the *ApartmentEntity*, and the *HotelEntity*, which represent an apartment and a Hotel respectively.

AccommodationOccupancyEntity The *AccommodationOccupancyEntity* represents the *Occupancy* table. It is an abstract class as well and has two child classes, the *ApartmentOccupancyEntity*, and the *HotelOccupancyEntity*. Both child classes, save the occupancy information of either an apartment or a hotel.

ReservationEntity The *ReservationEntity* represents the *Reservation* table. If a reservation is made for an apartment, it saves a plain *ReservationEntity* to the database. Otherwise, if a reservation has been made for a hotel, a *HotelReservationEntity* is saved to the database.

2.2.3 Database Routine

To seed the database with test data, the *DatabaseRoutine* project has been implemented. This project is not implemented as an EJB, instead, it will run as a standalone Java application. To utilize the database, this project uses Hibernate and JPA.

2.3 Enterprise Java Beans

The business logic of this application is implemented using three EJB's:

- *LocalDatabaseBean*
- *AccommodationService*
- *ReservationService*

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

Each part (service) is an independent EJB. The client can look up the EJB given an address and invoke methods on it. Only the *AccommodationService* and the *ReservationService* are available for the client. Both utilize the *Facade* pattern, to encapsulate the complexity to work with entities (introduced in Section 2.2.2) directly. The *DatabaseService* is implemented as a local bean because it is not intended for the user to interact with the database directly.

2.3.1 LocalDatabaseBean

The *LocalDatabaseBean* is used to interact with the H2 database. It initiates the *EntityManager*, given by JPA, to establish a connection to a JTA data source. Other services can use the *DatabaseService* to send queries and receive results from the database.

As mentioned before, the *DatabaseService* is implemented as a `LocalBean`. Other EJB's can interact with the *DatabaseService* using the code-injection pattern, which is shown in Listing 1.

```
1 @Stateless
2 @Remote(ReservationService.class)
3 public class ReservationBean implements ReservationService  {
4     @EJB()
5     private LocalDatabaseBean databaseBean;
6     ...
7 }
```

Listing 1: Usage of the *LocalDatabaseBean* using code-injection

2.3.2 AccommodationService

The *AccommodationService* is used by the client to interact with the *AccommdationEntity* and its child classes *ApartmentEntity*, and *HotelEntity* (introduced in Section 2.2.2). It uses the *LocalDatabaseBean* to send queries to the H2 database.

Except for receiving accommodations based on specific properties, the most interesting part about *AccommodationService* is how it receives available accommodations in a specific date range, given the number of guests. The idea is, that the number of occurrences of an accommodation that is available in the specific date range, has to be equal to 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 shows an example HQL (Hibernate Query Language) query, 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 to 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 subtract one day, because the guests will not stay for the last day, and therefore the availability information for this day is unrelated. After that, it is important, that the count of numbers of the accommodation, in that specific date range, is equal to the number of days between the start and end date.

2.3.3 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 Section 2.2.1) 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.

2.3.4 Build Process

The project is built using the `maven-ejb-plugin` maven plugin to create an EJB JAR artifact. 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

2.4 Web Application

The *WebApp* project implements the presentation layer of the application. Additionally, it constructs the EJB's (introduced in Section 2.3) for the business logic and presents the results to the client.

It consists of the following servlets:

- *AccommodationSearchServlet*
- *AccommodationResultServlet*
- *ReservationSummaryServlet*
- *ReservationConfirmServlet*
- *ReservationListServlet*

2.4.1 EJB Patterns

To utilize the EJB's (introduced in Section 2.3) on the client side the *WebApp* project uses the *Singleton*, *Simple Factory*, and *Business Delegate* patterns.

The helper classes, which uses these patterns are:

- *ServiceLocator*
- *ServiceFactory*
- *AccommodationServiceDelegate*
- *ReservationServiceDelegate*

ServiceLocator The *ServiceLocator* is responsible for lookup for a remote Bean given an address. After that, it can be used recreated and used on the client. Internally, it uses the *Singleton* pattern. Listing 4 shows an example of how the AccommodationServe can be constructed using the *ServiceLocator*.

```
String serviceAddress = getAccommodationbeanAddress();
AccommodationService service = ServiceLocator.getInstance().
    getService(serviceAddress);
service.getAccommodations();
```

Listing 4: Example usage of the *ServiceLocator*

ServiceFactory The usage of the *ServiceLocator* still requires some boilerplate code, each time an EJB Bean needs to be constructed, for example, the generator of the Bean address. To simplify this process, the *ServiceFactory* uses the *Simple Factory* pattern to construct EJB Beans based on its name. Listing 5 shows an example of how to use the *ServiceFactory* to construct the *AccommodationService*.

```
AccommodationService service = ServiceFactory.initializeService(
  "AccommodationService",
  AccommodationService.class.getName()
);
```

Listing 5: Example usage of the *ServiceFactory*

Business Delegates The *AccommodationDelegate* and *ReservationDelegate* both use the *Business Delegate* pattern, to abstract the usage of the *Accommodation-Service* and *ReservationService* respectively. The servlets controller (previously mentioned in Section 2.4) uses these *Business Delegates*, instead of constructing and invoking methods on the remote Beans directly. To construct the Beans, the *Business Delegates* use the *ServiceFactory* internally.

2.4.2 AccommodationSearchServlet

The *AccommodationSearchServlet* is the start page of the *WebApp* and provides the search to search for available accommodations, as shown in Figure 4.

The user can set the start, and end date of the duration, as well as the number of persons who intend to be in the accommodation during that time. After submitting the form, the servlet sends a GET request with the given data to the *AccommodationResultServlet*.

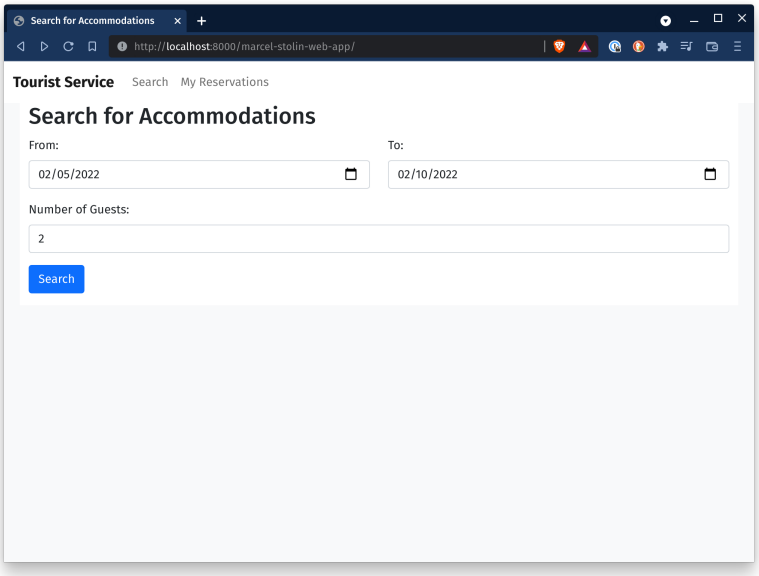

Figure 4: UI of the *AccommodationSearchServlet*

2.4.3 AccommodationResultServlet

After submitting the search form, the *AccommodationResultServlet* is responsible to present the results. If no accommodations are available for the given specifications, a message is shown. Otherwise, a grid of all available results, order by the daily price, is presented to the user, as shown in Figure 5. There, the user can click on the *Book* button to open the *ReservationSummaryServlet*. If the accommodation is a *Hotel* entity, two buttons are shown, one for the total price without half-board, and on including half-board.

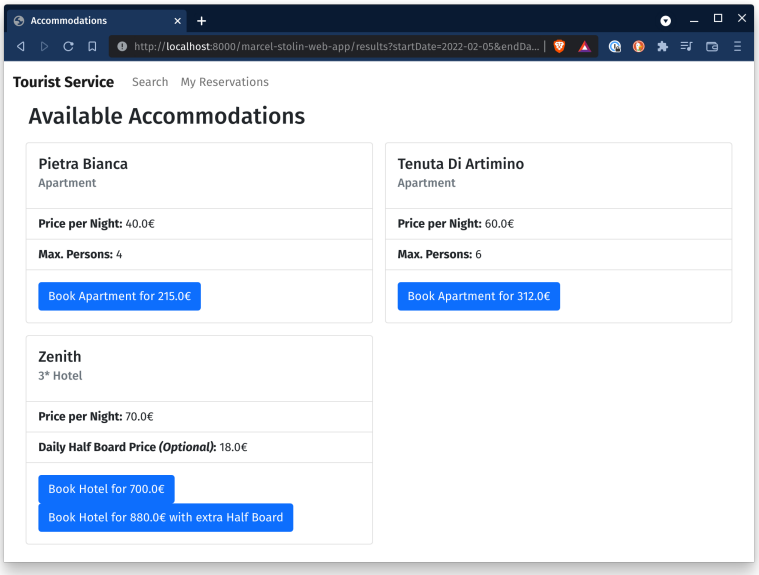

Figure 5: UI of the *AccommodationResultServlet*

2.4.4 ReservationSummaryServlet

In the *ReservationSummaryServlet* the user can see a summary after selecting an accommodation at the *AccommodationResultServlet*, as shown in Figure 6. Additionally, at this point, the user has the opportunity to confirm the reservation by clicking the *Confirm* button or canceling the reservation by clicking the *Cancel* button. If the user decides to confirm the reservation, the user has to provide a first name, last name, and a credit card number. Then, a POST request is sent to the *ReservationConfirmServlet*.

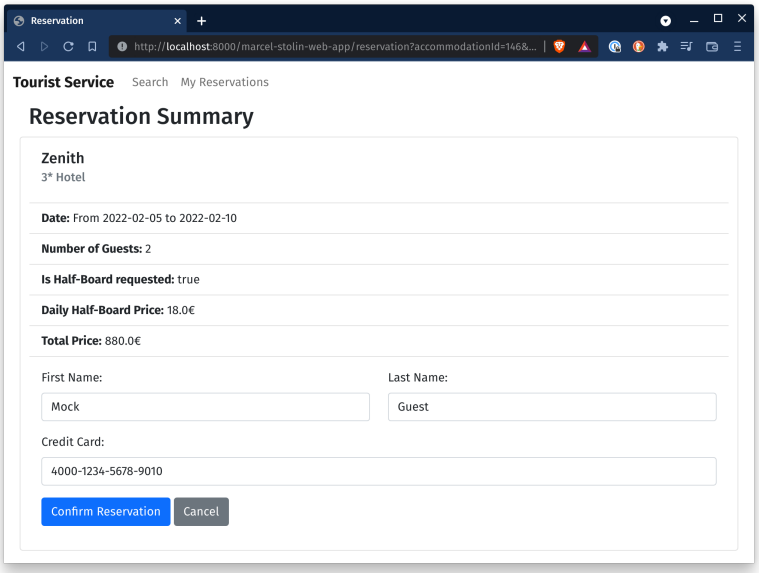

Figure 6: UI of the *ReservationSummaryServlet*

2.4.5 ReservationConfirmServlet

The *ReservationConfirmServlet* is responsible to save a reservation, and showing the user a success message, as illustrated in Figure 7.

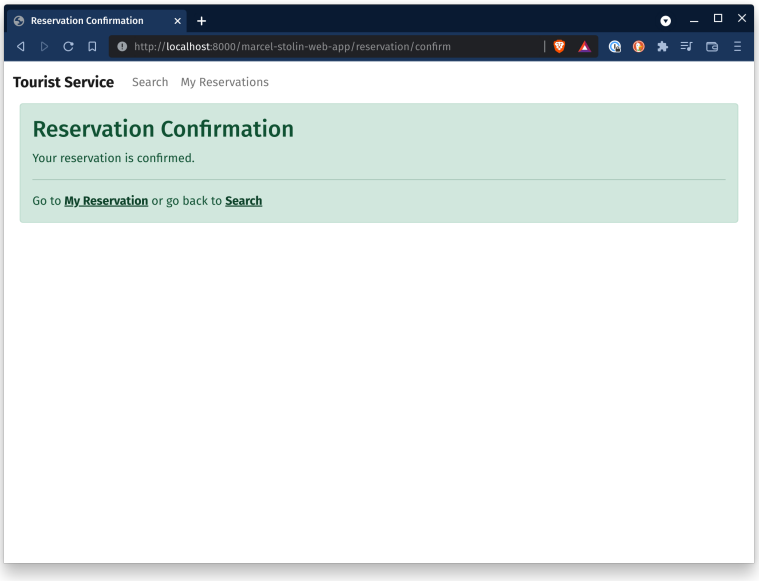

Figure 7: UI of the *ReservationConfirmServlet*

2.4.6 ReservationListServlet

After a reservation has been confirmed, the user can look up all reservations at the *ReservationListServlet*. There, the user has to provide the first name, and last name, same as given at the *ReservationSummaryServlet* (introduced in Section 2.4.4), shown in Figure 8. After submitting, the *ReservationListServlet* sends a POST request to itself to present all reservations made by the given user, shown in Figure 9.

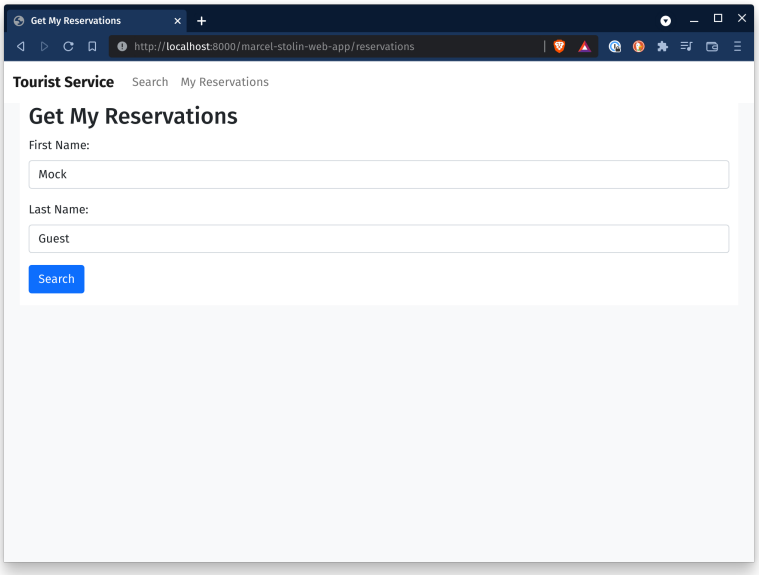

Figure 8: UI of the *ReservationListServlet*

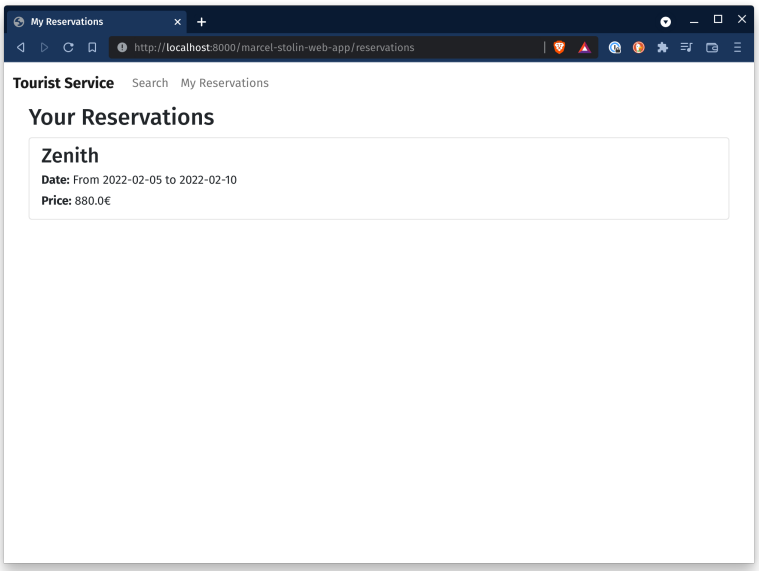

Figure 9: UI of the *ReservationListServlet*

3 Deployment

This section explains the deployment of the application. The requirements to run all applications are the following (the versions correspond to the local system of the author):

- H2[1] 2.0.202
- WildFly[2] 20.0.1.Final
- Java[3] AdoptOpenJDK-11.0.11+9
- Apache Tomcat[4] 9.0.56
- Apache Maven[5] 3.8.4

3.1 H2 Database

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

3.1.1 Start H2

At first, it is important to start the H2 service via the terminal. This can be accomplished by executing the command `$ java -jar h2*.jar` in the directory `\$H2/bin`, where `$H2` is the path of the H2 database. This process is shown in Figure 10. After executing the command, the *H2* console is automatically started in the default web browser, which is needed later.

```
~/h2/bin
❯ java -jar h2*.jar
Opening in existing browser session.
libva error: /usr/lib/x86_64-linux-gnu/dri/i965_drv_video.so init failed
[190690:190690:0100/000000.977293:ERROR:sandbox_linux.cc(376)] InitializeSandbox
() called with multiple threads in process gpu-process.
```

Figure 10: Process to start the H2 service

3.1.2 Create a Database

After the H2 service has been started, it is needed to create a new local database using the shell tool of H2. This process is shown in Figure 11, where the command `$ java -cp h2-*.jar org.h2.tools.Shell` needs to get executed in the `\$H2/`

[1] H2 Database - `http://www.h2database.com/`
[2] WildFly - `https://www.wildfly.org/`
[3] OpenJDK - `https://openjdk.java.net/`
[4] Apache Tomcat - `https://tomcat.apache.org/`
[5] Apache Maven - `https://maven.apache.org/`

`bin` directory. It is important, that the database is called *accommodations*, the username is *sa*, and the password is *sa* as well.

```
~/h2/bin
❯ java -cp h2-*.jar org.h2.tools.Shell

Welcome to H2 Shell 2.0.202 (2021-11-25)
Exit with Ctrl+C
[Enter]   jdbc:h2:tcp://localhost/~/workspace/145085_web-architectures/assignment_5/accommodations
URL       jdbc:h2:/home/marcel/workspace/145085_web-architectures/assignment_5/accommodations
[Enter]   org.h2.Driver
Driver
[Enter]   sa
User
Password
Type the same password again to confirm database creation.
Password
Connected
Commands are case insensitive; SQL statements end with ';'
help or ?      Display this help
list           Toggle result list / stack trace mode
maxwidth       Set maximum column width (default is 100)
autocommit     Enable or disable autocommit
history        Show the last 20 statements
quit or exit   Close the connection and exit

sql>
```

Figure 11: Process to create a H2 database

3.1.3 Test Database Connection

After creating the *accommodations* database, it is possible to test the connection using the H2 console mentioned in Section 3.1.1. First, it is necessary to fill in the correct information: The JDBC URL in the format `jdbc:h2:tcp://localhost/PATH_TO_DATABASE/` and the previously used username and password from Section 3.1.2 (username: `sa`, password: `sa`). Then, by clicking on *Test Connection*, a status message is shown as illustrated in Figure 12. If the connection is successful, the database can be used in the application.

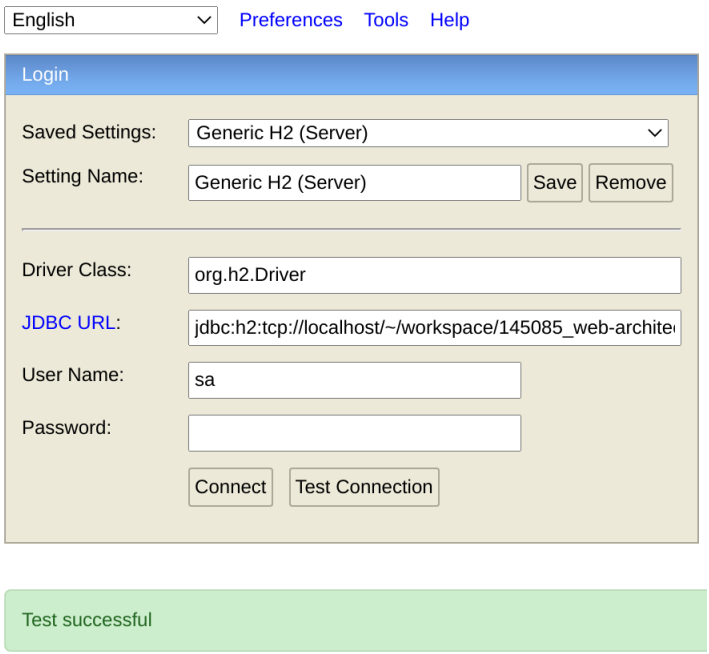

Figure 12: Successfull connection test for the H2 database

3.2 Seeding the Database

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

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* project. The `persistance.xml` file is located at `DatabaseRoutine/src/main/resources/META-INF/persistance.xml`. Listing 6 shows a correct configuration. The property `hibernate.connection.url` has to be set to the URL used in Section 3.1.2.

```
...
<persistence-unit name="default">
  <properties>
    <property name="hibernate.connection.url"
              value="jdbc:h2:tcp://localhost/~/workspace/145085
    _web-architectures/assignment_5/accommodations"/>
  </properties>
</persistence-unit>
...
```

Listing 6: Default data source configuration

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 executed and writes 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 Section 3.1.2. Figure 13 shows the newly created tables in the database.

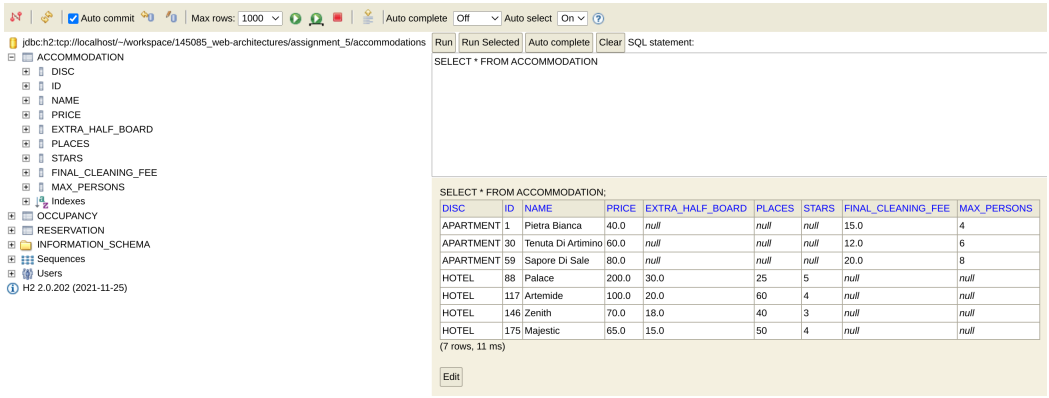

Figure 13: Successfull connection test for the H2 database

3.3 Setting up WildFly

After the H2 database is created, the EJB's can be deployed to the WildFly application server.

It is important to mention, that at this point the H2 database (explained in Section 3.1) has to be running.

3.3.1 Add H2 database data source

To give the EJB's the ability to access the *accommodations* database, it has to be defined as a data source in the configuration file of WildFly. The configuration file is located at `\$JBOSS\_HOME/standalone/configuration/standalone.xml`, where `$JBOSS_HOME` is the path to WildFly. Listing 7 shows how the *accommodations* database, created in Section 3.1, has been added as a data source in WildFly given the name `AccommodationsDS`.

```
1 ...
2 <subsystem xmlns="urn:jboss:domain:datasources:6.0">
3   <datasources>
4     <datasource jndi-name="java:jboss/datasources/AccommodationsDS
    " pool-name="AccommodationsDS" enabled="true" use-java-context=
    "true" statistics-enabled="${wildfly.datasources.statistics-
    enabled:${wildfly.statistics-enabled:false}}">
5       <connection-url>jdbc:h2:tcp://localhost/~/workspace/145085
    _web-architectures/assignment_5/accommodations;DB_CLOSE_DELAY
    =-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
6       <driver>h2</driver>
7       <security>
8         <user-name>sa</user-name>
9         <password>sa</password>
```

```
      </security>
    </datasource>
    ...
  </datasources>
</subsystem>
...
```

Listing 7: WildFly datasource configuration

Additionally, this data source has to be added to the `persistance.xml` configuration file of the *WebServices* project as well, which is located at `WebServices/src/main/resources/META-INF/persistance.xml`. Therefore, a new *jta-data-source* with the value `java:jboss/datasources/AccommodationsDS` is added to the default *persistence unit*, as shown in Listing 8.

```
...
<persistence-unit name="default">
  <jta-data-source>java:jboss/datasources/AccommodationsDS</jta-
   data-source>
  <properties>
    <property name="hibernate.show_sql" value="true"/>
    <property name="hibernate.format_sql" value="true"/>
    <property name="hibernate.use_sql_comments" value="true"/>
  </properties>
</persistence-unit>
...
```

Listing 8: Default data source configuration

3.3.2 Compile EJB Jar

Next, the EJB's need 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, which builds the `.jar` artifact containing all EJB's. After that, a new directory called `/target` was created in the root folder of the *WebServices* project, where the EJB JAR `marcel-stolin-web-services.jar` is located, which is shown in Figure 14.

```
~/workspace/145085_web-architectures/assignment_5/Assignment-5-Marcel-Stolin/WebServices/target main*
❯ l
Permissions Size User   Date Modified Name
drwxrwxr-x     - marcel  1 Feb 09:24  classes
drwxrwxr-x     - marcel  1 Feb 09:24  generated-sources
.rw-rw-r--   21k marcel  1 Feb 09:24  marcel-stolin-web-services.jar
drwxrwxr-x     - marcel  1 Feb 09:24  maven-archiver
drwxrwxr-x     - marcel  1 Feb 09:24  maven-status
```

Figure 14: Compiled EJB JAR of the *WebServices* project

3.3.3 EJB JAR Deployment

After the `.jar` artifact has been created, it needs to be copied to the deployment directory of the WildFly server. The artifact can be deployed using the command

shown in Listing 9, where `$JBOSS_HOME` is the path to the local WildFly directory. This command needs to be executed in the project folder of the *WebServices* project, which is shown in Figure 15.

```
$ cp target/marcel-stolin-web-services.jar $JBOSS_HOME/standalone/
    deployments
```

Listing 9: Command to copy the EJB JAR artifact to the WildFly deployment directory

Figure 15: Process to deploy the EJB JAR artifact to the WildFly application server

3.3.4 Start Wildfly Application Server

Finally, the WildFly server can be started using the command `$ bin/standalone.sh`. Additionally, the log prints the lookup addresses of both the *AccommodationService* (mentioned in Section 2.3.2) and the *ReservationService* (mentioned in Section 2.3.3), which is illustrated in Figure 16.

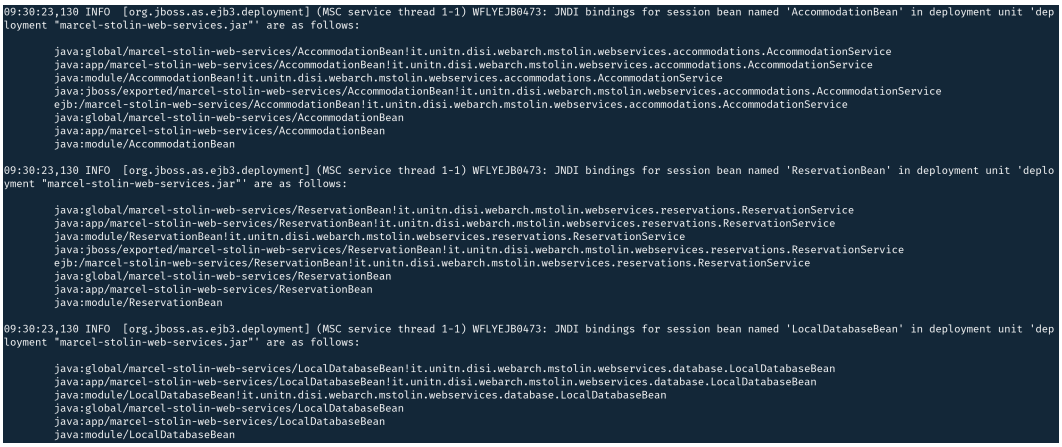

Figure 16: WildFly application server Log

3.4 Starting the Web Application

After the database has been created successfully, and the EJB‘s have been deployed successfully, the web application can be deployed to Apache Tomcat.

One problem is, that WildFly already uses port 8080, and Apache Tomcat uses port 8080 by default as well. Therefore, Apache Tomcat has to be configured to use port 8000 instead.

For this project, the IntelliJ IDEA is used to start and deploy the *WebApp* project on Apache Tomcat. Therefore, it is necessary to create a *Run Configuration* for Apache Tomcat for the *WebApp* project.

3.4.1 Server Settings

The *Server* settings for IntelliJ are shown in Figure 17. It is important, that the URL is set to `http://localhost:8000/marcel-stolin-web-app/` and the HTTP port, at *Tomcat Server Settings*, is set to 8000. In addition, under *Before launch* it is important to build the *WebApp:war exploded* artifact.

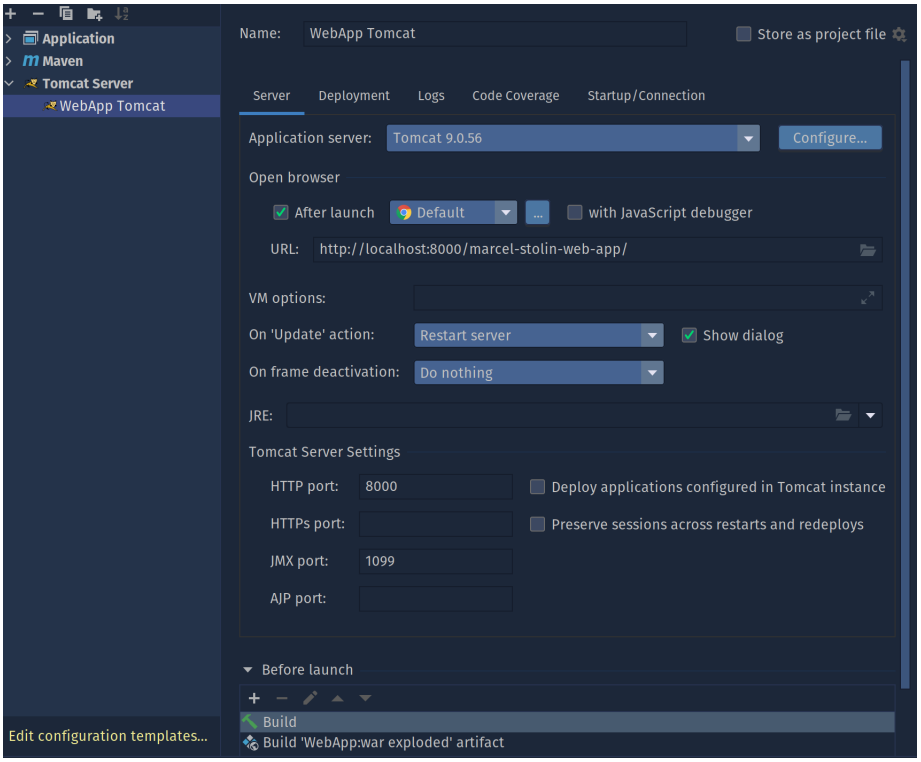

Figure 17: IntelliJ Tomcat Configuration Server Settings

3.4.2 Deployment Settings

Figure 18 shows the settings for the *Deployment* section. It is necessary to deploy the *WebApp:war exploded* artifact on server startup. Additionally, the *Application Context* needs to be set to `marcel-stolin-web-app`.

3.4.3 Starting the Application

By running the Apache Tomcat *Run Configuration*, the web application is available at `http://localhost:8000/marcel-stolin-web-app/`.

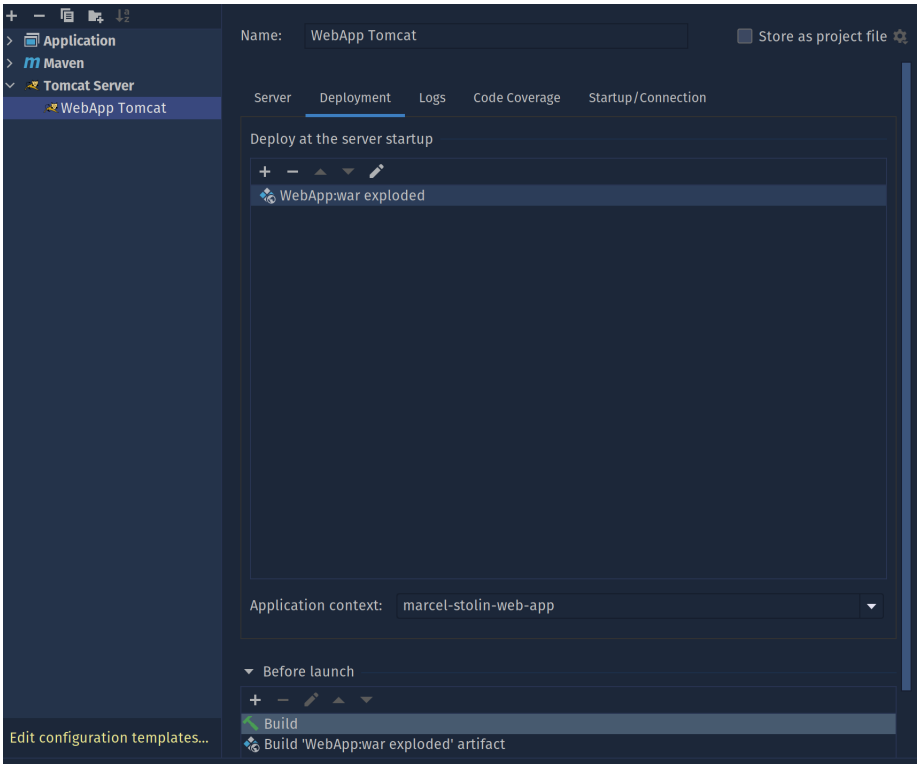

Figure 18: IntelliJ Tomcat Configuration Deployment Settings

4 Comments and Notes

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

4.1 Deploying the WebApp manually

Section 3.4 describes how to start the *WebApp* project using IntelliJ. Another solution would be, to create the `.war` artifact, and deploy that artifact to Apache Tomcat manually.

4.1.1 Build and Deploy the WebApp Artifact

To create the `.war` artifact, the command `$ mvn clean build` has to be executed in the root of the *WebApp* project directory. After the artifact has been built, it is located at `WebApp/target/marcel-stolin-web-app.war`, as shown in Figure 19. Next, to deploy the artifact to Apache Tomcat, it needs to be copied to the path `\$TOMCAT_HOME\webapps`, where `$TOMCAT_HOME` is the directory of Apache Tomcat. This process is shown in Listing 10.

```
1 $ cp target/marcel-stolin-web-app.war $TOMCAT_HOME/webapps
```

Listing 10: Artifact deployment command

```
~/workspace/145085_web-architectures/assignment_5/Assignment-5-Marcel-Stolin/WebApp/target main*
❯ l
Permissions Size User   Date Modified Name
drwxrwxr-x     - marcel 30 Jan 15:04  classes
drwxrwxr-x     - marcel 30 Jan 15:04  generated-sources
drwxrwxr-x     - marcel 30 Jan 15:04  marcel-stolin-web-app
.rw-rw-r--   31M marcel 30 Jan 15:04  marcel-stolin-web-app.war
drwxrwxr-x     - marcel 30 Jan 15:04  maven-archiver
drwxrwxr-x     - marcel 30 Jan 15:04  maven-status
drwxrwxr-x     - marcel 30 Jan 15:04  test-classes
```

Figure 19: *WebApp* `.war` artifact

4.1.2 Configure Apache Tomcat

As described in Section 3.4, the port of Apache Tomcat needs to be changed to port 8000. Therefore, the file `\$TOMCAT_HOME/conf/server.xml` has to be changed as given in Listing 11. This will change the default port to 8000.

```
<Service name="Catalina">
  <Connector port="8000" protocol="HTTP/1.1"
             connectionTimeout="20000"
             redirectPort="8443" />
</Service>
```

Listing 11: Apache Tomcat configuration

After the artifact has been deployed, and the default port is updated, the Apache Tomcat server can be started using the command `$TOMCAT_HOME/bin/catalina.sh start`. Then, the *WebApp* is available at `http://localhost:8000/marcel-stolin-web-app`.