



*Università di Trento*

*Web Architectures*

---

## Assignment 5

---

*Author:*

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

February 3, 2022

# 1 Introduction

The fifth assignment is about creating an application where tourists can book apartments or a hotel rooms, given a 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 suppose to contain the business logic of the application. Additionally, a web application, and a database is 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 2.1), the purpose and behaviour of the EJB's (Section 2.2), and the behaviour about the servlets (Section 2.3).

### 2.1 Database

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

#### 2.1.1 Structure

The overall database structure, visualized in Figure 1, 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 is reasonable to use this strategy and have a time efficiency advantage.

**Accommodation** The *Accommodation* table is supposed to save all kinds of accommodation, apartments and hotels. Both have a name and price in. 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 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 name, the guests 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.



Figure 1: Diagram of the database structure

### 2.1.2 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 2 visualizes the UML diagram of the entities used to map the database structure (introduced in Section 2.1.1). Overall, the following entities are used in the implementation:

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

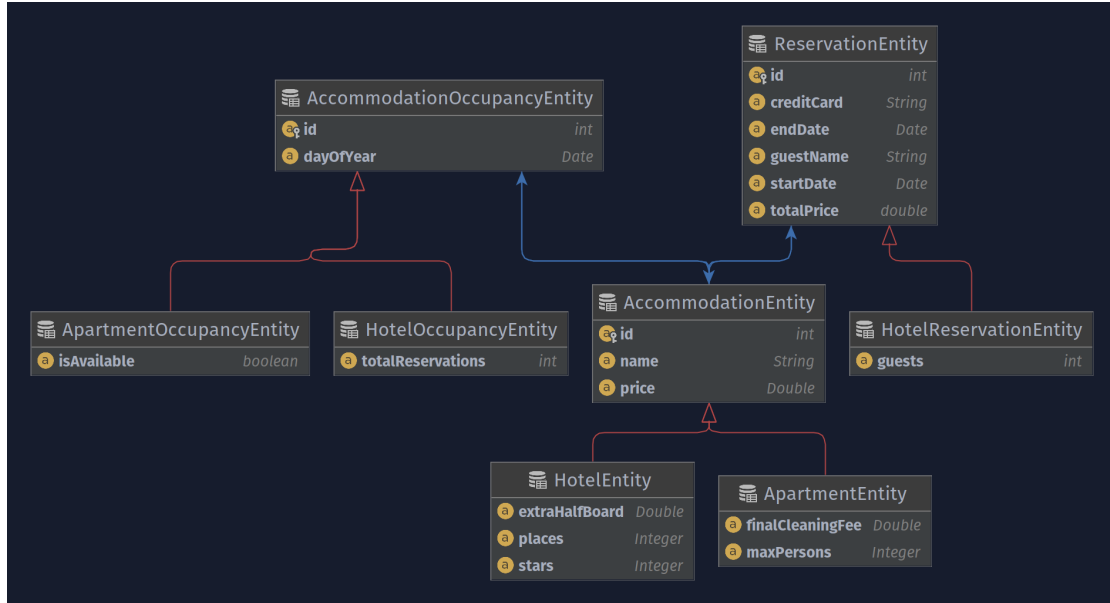


Figure 2: Database entities UML diagram

**AccommodationEntity** The *AccommodationEntity* represents the *Accommodation* table (introduced in Section 2.1.1). 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* represent the *Reservation* table. If a reservation is made for an aptartment, 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.

### 2.1.3 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.

## 2.2 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 is able to lookup the EJB given an address and invoke methods on it. Only the *AccommodationService* and the *ReservationService* are available for the client. The *DatabaseService* is implemented as 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, to establish a connection to a JTA datasource. 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

### 2.2.2 AccommodationService

The *AccommodationService* is used by the client to interact with the *AccommodationEntity* and its child classes *ApartmentEntity*, and *HotelEntity* (introduced in Section 2.1.2). It uses the *LocalDatabaseBean* to send 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 occurrences of an accommodation which is available in the specific date range, has to be equal the number of days of the given date range.

```

1 SELECT a.accommodation
2 FROM AccommodationOccupancyEntity a
3 WHERE (
4     ((a.isAvailable IS TRUE AND a.accommodation.maxPersons >= 2)
5     OR
6     ((a.accommodation.places - a.totalReservations) >= 2))
7 )
8 AND a.dayOfYear BETWEEN '2022-02-01' AND '2022-02-09'
9 GROUP BY a.accommodation.id
10 HAVING COUNT(*) = 9

```

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

Listing 2 show 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 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.

### 2.2.3 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 Section 2.1.1) 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.

### 2.2.4 Build Process

The project is build 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.

```
1 <packaging>ejb</packaging>
2 ...
3 <build>
4   <finalName>${artifactId}.${version}</finalName>
5   <plugins>
6     <plugin>
7       <groupId>org.apache.maven.plugins</groupId>
8       <artifactId>maven-ejb-plugin</artifactId>
9       <version>3.1.0</version>
10    </plugin>
11  </plugins>
12</build>
```

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

## 2.3 Web Application

The *WebApp* project implements the presentation layer of the application. Additionally, it constructs the EJB's (introduced in Section 2.2) for the business logic, and presents the results to the client.

It consists of the following servlets:

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

### 2.3.1 EJB Patterns

To utilize the EJB's (introduced in Section 2.2) 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 to 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*.

```
1 String serviceAddress = getAccommodationbeanAddress();
2 AccommodationService service = ServiceLocator.getInstance().
  getService(serviceAddress);
3 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 on how to use the *ServiceFactory* to construct the *AccommodationService*.

```
1 AccommodationService service = ServiceFactory.initializeService(
2   "AccommodationService",
3   AccommodationService.class.getName()
4 );
```

Listing 5: Example usage of the *ServiceFactory*

**Business Delegates** The *AccommodationDelegate* and *ReservationDelegate* both uses the *Business Delegate* pattern, to abstract the usage of the *AccommodationService* and *ReservationService* respectively. The servlets controller (previously mentioned in Section 2.3) uses these *Business Delegates*, instead of constructing, and invoking methods on the remote Beans directly. To construct the Beans, the *Business Delegates* uses the *ServiceFactory* internally.

### 2.3.2 AccommodationSearchServlet

The *AccommodationSearchServlet* is the start page of the *WebApp*, and provides the search to search for available accommodations, as shown in Figure 3.

The user is able to 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*.

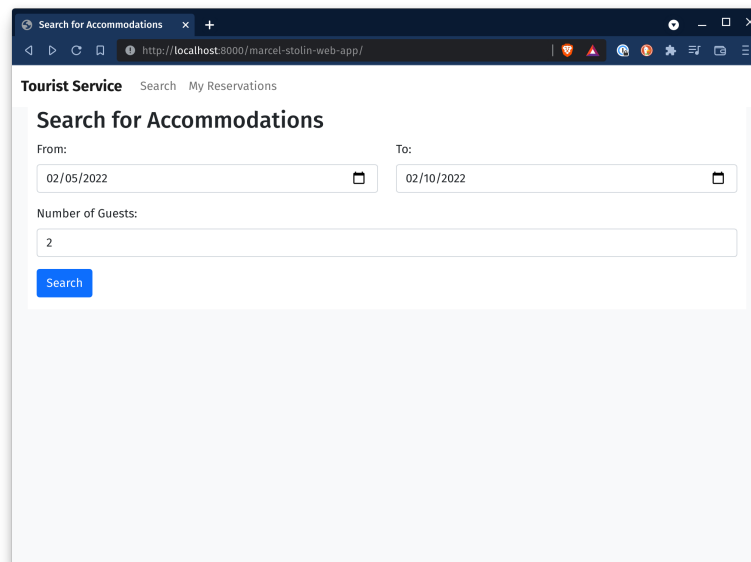
A screenshot of a web browser window showing the 'Search for Accommodations' page of a 'Tourist Service'. The browser's address bar shows 'http://localhost:8000/marcel-stolin-web-app/'. The page has a navigation bar with 'Tourist Service', 'Search', and 'My Reservations'. The main heading is 'Search for Accommodations'. Below this, there are three input fields: 'From:' with the date '02/05/2022', 'To:' with the date '02/10/2022', and 'Number of Guests:' with the value '2'. Each date field has a calendar icon to its right. A blue 'Search' button is positioned below the 'Number of Guests' field. The rest of the page is a light blue-grey color.

Figure 3: UI of the *AccommodationSearchServlet*

### 2.3.3 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 4. There, the user can click on the *Book* button to open the *ReservationSummaryServlet*. If the accommodation is a *Hotel* entity, two button are shown, one for the total price without half-board, and on including half-board.

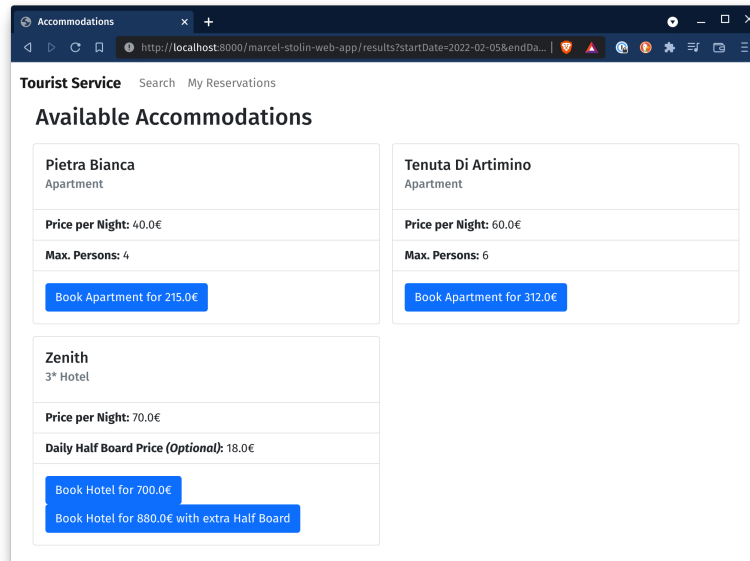


Figure 4: UI of the *AccommodationResultServlet*

### 2.3.4 ReservationSummaryServlet

In the *ReservationSummaryServlet* the user is able to see a summary after selecting an accommodation at the *AccommodationResultServlet*, as shown in Figure 5. 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, a last name, and a credit card number. Then, a POST request is sent to the *ReservationConfirmServlet*.

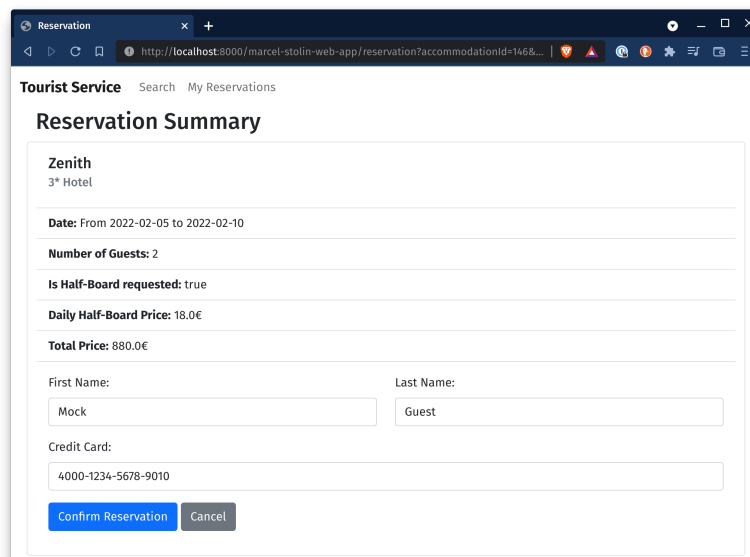


Figure 5: UI of the *ReservationSummaryServlet*

### 2.3.5 ReservationConfirmServlet

The *ReservationConfirmServlet* is responsible to save a reservation, and show the user a success message, as illustrated in Figure 6.

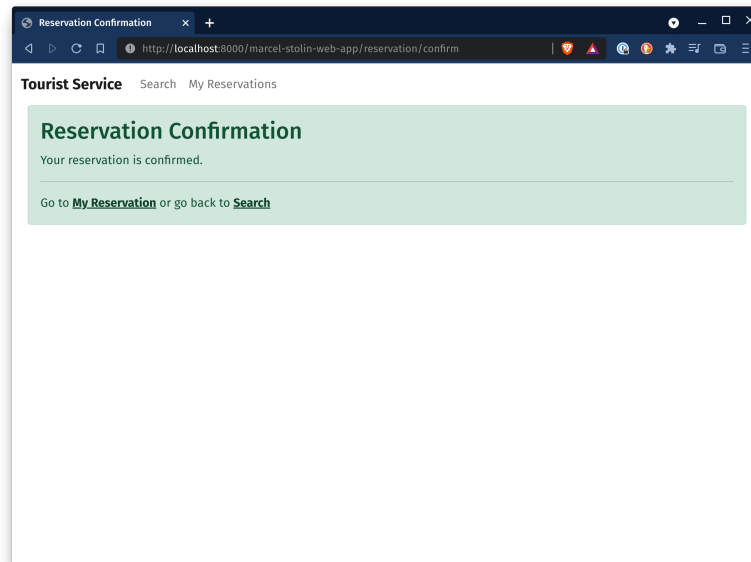


Figure 6: UI of the *ReservationConfirmServlet*

### 2.3.6 ReservationListServlet

After a reservation has been confirmed, the user is able to lookup 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 2.3.4), shown in Figure 7. After submitting, the *ReservationListServlet* sends a POST request to itself to present all reservation made by the given user, shown in Figure 8.

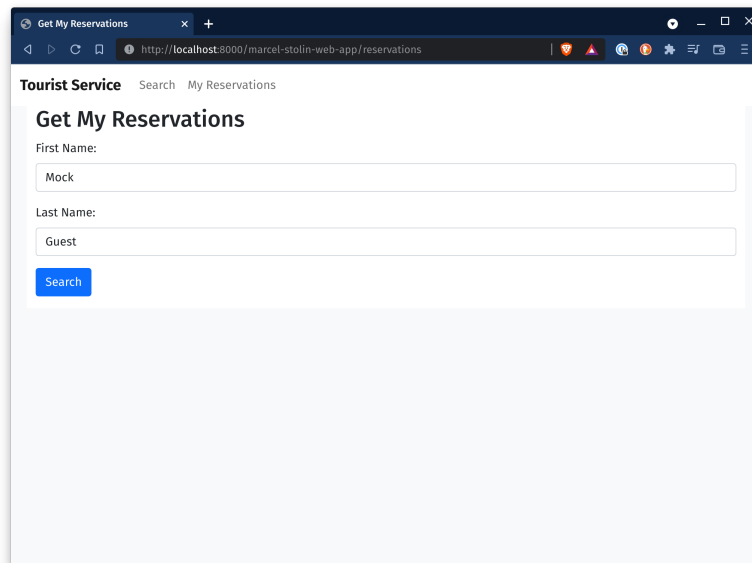


Figure 7: UI of the *ReservationListServlet*

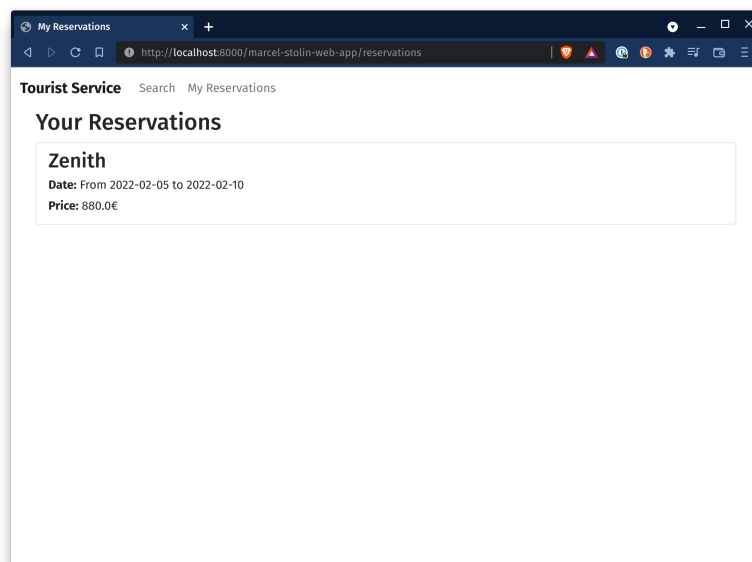


Figure 8: UI of the *ReservationListServlet*

## 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<sup>1</sup> 2.0.202
- WildFly<sup>2</sup> 20.0.1.Final
- Java<sup>3</sup> AdoptOpenJDK-11.0.11+9
- Apache Tomcat<sup>4</sup> 9.0.56
- Apache Maven<sup>5</sup> 3.8.4

### 3.1 H2 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.

#### 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 9. 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 9: 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 10, where the command `$ java -cp h2-*.jar org.h2.tools.Shell` needs to get executed in the `\$H2/bin`

---

<sup>1</sup>H2 Database - <http://www.h2database.com/>

<sup>2</sup>WildFly - <https://www.wildfly.org/>

<sup>3</sup>OpenJDK - <https://openjdk.java.net/>

<sup>4</sup>Apache Tomcat - <https://tomcat.apache.org/>

<sup>5</sup>Apache Maven - <https://maven.apache.org/>

directory. It is important, that the database is called *accommodations*, and the username is *sa*, and the 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 10: 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 informations: 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 11. If the connection is successful, the database can be used in the application.

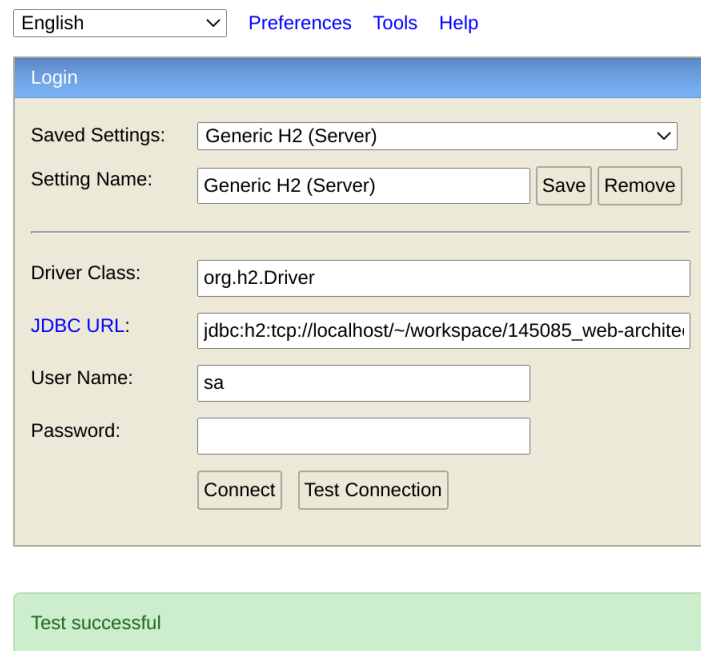


Figure 11: Successfull connection test for the H2 database

## 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 `persistence.xml` of the *DatabaseRoutine* project. The `persistence.xml` file is located at `DatabaseRoutine/src/main/resources/META-INF/persistence.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.

```

1 ...
2 <persistence-unit name="default">
3   <properties>
4     <property name="hibernate.connection.url"
5       value="jdbc:h2:tcp://localhost/~workspace/145085
6       _web-architectures/assignment_5/accommodations"/>
7   </properties>
8 </persistence-unit>
9 ...

```

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 12 shows the newly created tables in the database.

DISC	ID	NAME	PRICE	EXTRA_HALF_BOARD	PLACES	STARS	FINAL_CLEANING_FEE
APARTMENT	1	Pietra Bianca	40.0	null	null	15.0	4
APARTMENT	30	Tenuta Di Artimino	60.0	null	null	12.0	6
APARTMENT	59	Sapore Di Sale	80.0	null	null	20.0	8
HOTEL	88	Palace	200.0	30.0	25	5	null
HOTEL	117	Artemide	100.0	20.0	60	4	null
HOTEL	146	Zenith	70.0	18.0	40	3	null
HOTEL	175	Majestic	65.0	15.0	50	4	null

Figure 12: 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 datasource

To give the EJB's the ability to access the *accommodations* database, it has to be defined as a datasource in the configuration file of WildFly. The configuration file is located at `\$JBOSSE\HOME\standalone\configuration\standalone.xml`, where `\$JBOSSE_HOME` is the path to WildFly. Listing 7 shows how the *accommodations* database, created in Section 3.1, has been added as a datasource 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"
5       pool-name="AccommodationsDS" enabled="true" use-java-context=
6       "true" statistics-enabled="${wildfly.datasources.statistics-
7       enabled:${wildfly.statistics-enabled:false}}">
8       <connection-url>jdbc:h2:tcp://localhost/~workspace/145085
9       _web-architectures/assignment_5/accommodations;DB_CLOSE_DELAY
        =-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
        <driver>h2</driver>
        <security>
          <user-name>sa</user-name>
          <password>sa</password>

```

```

10     </security>
11     </datasource>
12     ...
13 </datasources>
14 </subsystem>
15 ...

```

Listing 7: WildFly datasource configuration

Additionally, this datasource has to be added to the `persistence.xml` configuration file of the `WebServices` project as well, which is located at `WebServices/src/main/resources/META-INF/persistence.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.

```

1 ...
2 <persistence-unit name="default">
3   <jta-data-source>java:jboss/datasources/AccommodationsDS</jta-
   data-source>
4   <properties>
5     <property name="hibernate.show_sql" value="true"/>
6     <property name="hibernate.format_sql" value="true"/>
7     <property name="hibernate.use_sql_comments" value="true"/>
8   </properties>
9 </persistence-unit>
10 ...
11 %

```

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 at Figure 13.

```

~/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 13: 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 14.

```
1 $ 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

```
~/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

~/workspace/145085_web-architectures/assignment_5/Assignment-5-Marcel-Stolin/WebServices/target main*
) cp marcel-stolin-web-services.jar $JBOSS_HOME/standalone/deployments
```

Figure 14: 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 ??) and the *ReservationService* (mentioned in Section ??), which is illustrated in Figure 15.

```
09:30:23,130 INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-1) WFLYEJB0473: JNDI bindings for session bean named 'AccommodationBean' in deployment unit 'deployment "marcel-stolin-web-services.jar"' are as follows:
    java:global/marcel-stolin-web-services/AccommodationBean!it.unitn.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    java:app/marcel-stolin-web-services/AccommodationBean!it.unitn.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    java:module/AccommodationBean!it.unitn.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    java:jboss/exported/marcel-stolin-web-services/AccommodationBean!it.unitn.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    ejb:/marcel-stolin-web-services/AccommodationBean!it.unitn.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    java:global/marcel-stolin-web-services/AccommodationBean
    java:app/marcel-stolin-web-services/AccommodationBean
    java:module/AccommodationBean

09:30:23,130 INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-1) WFLYEJB0473: JNDI bindings for session bean named 'ReservationBean' in deployment unit 'deployment "marcel-stolin-web-services.jar"' are as follows:
    java:global/marcel-stolin-web-services/ReservationBean!it.unitn.disi.webarch.mstolin.webservices.reservations.ReservationService
    java:app/marcel-stolin-web-services/ReservationBean!it.unitn.disi.webarch.mstolin.webservices.reservations.ReservationService
    java:module/ReservationBean!it.unitn.disi.webarch.mstolin.webservices.reservations.ReservationService
    java:jboss/exported/marcel-stolin-web-services/ReservationBean!it.unitn.disi.webarch.mstolin.webservices.reservations.ReservationService
    ejb:/marcel-stolin-web-services/ReservationBean!it.unitn.disi.webarch.mstolin.webservices.reservations.ReservationService
    java:global/marcel-stolin-web-services/ReservationBean
    java:app/marcel-stolin-web-services/ReservationBean
    java:module/ReservationBean

09:30:23,130 INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-1) WFLYEJB0473: JNDI bindings for session bean named 'LocalDatabaseBean' in deployment unit 'deployment "marcel-stolin-web-services.jar"' are as follows:
    java:global/marcel-stolin-web-services/LocalDatabaseBean!it.unitn.disi.webarch.mstolin.webservices.database.LocalDatabaseBean
    java:app/marcel-stolin-web-services/LocalDatabaseBean!it.unitn.disi.webarch.mstolin.webservices.database.LocalDatabaseBean
    java:module/LocalDatabaseBean!it.unitn.disi.webarch.mstolin.webservices.database.LocalDatabaseBean
    java:global/marcel-stolin-web-services/LocalDatabaseBean
    java:app/marcel-stolin-web-services/LocalDatabaseBean
    java:module/LocalDatabaseBean
```

Figure 15: 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 16. 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 import to build the *WebApp:war exploded* artifact.

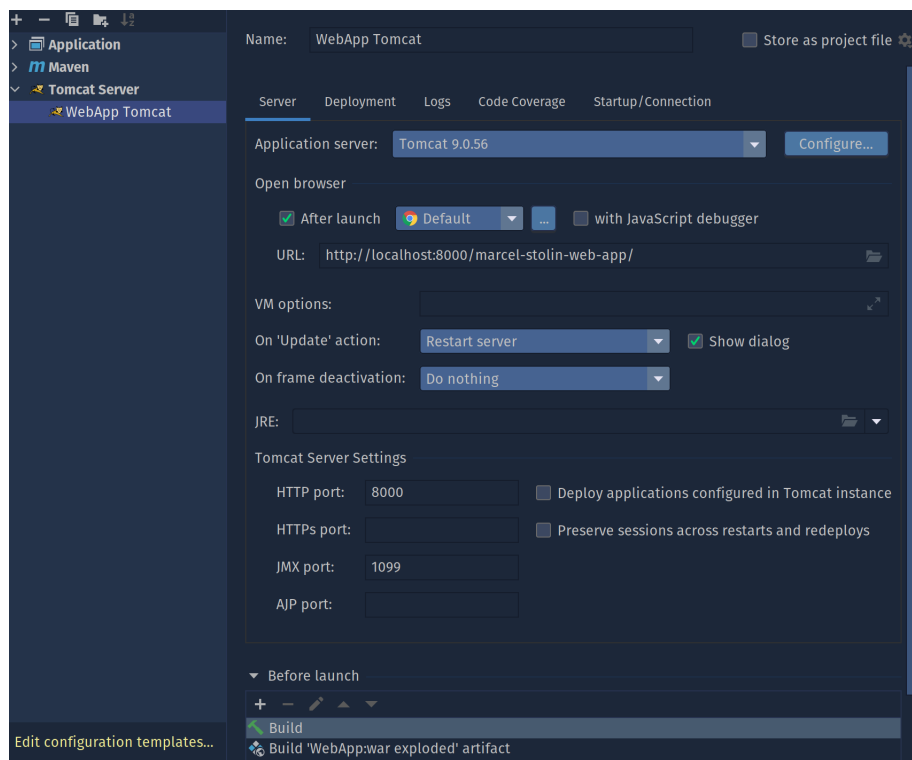


Figure 16: IntelliJ Tomcat Configuration Server Settings

#### 3.4.2 Deployment Settings

Figure 17 shows the settings for the *Deployment* section. It is necessary to deploy the *WebApp:war exploded* artifact on server startup. Additionally, the *Application Context* need to be set to `marcel-stolin-web-app`.

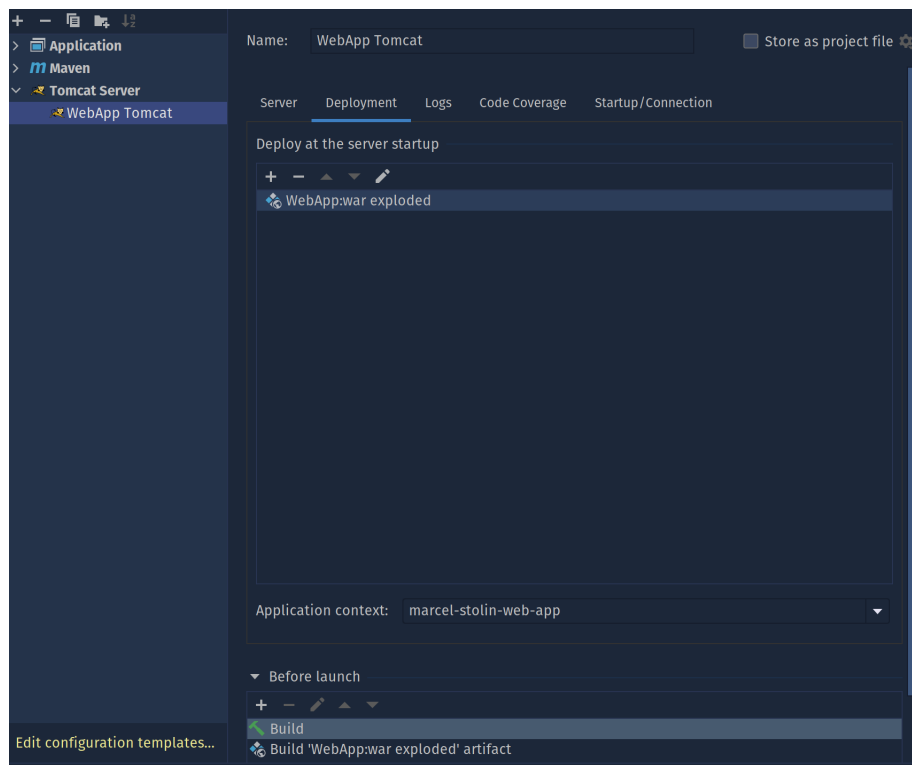


Figure 17: IntelliJ Tomcat Configuration Deployment Settings

### 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/`.

## 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`<sup>6</sup>. Listing 10 shows how it is implemented in the application. Then, the previously mentioned URL can be accessed via `http://localhost:8080/marcel-stolin/#/detail/1735`.

```
1 const routes: Routes = [  
2   { path: '', component: MemberListComponent },  
3   { path: 'detail/:memberId', component: MemberDetailComponent }  
4 ];  
5  
6 @NgModule({  
7   imports: [RouterModule.forRoot(routes, {useHash: true})],  
8   exports: [RouterModule]  
9 })
```

Listing 10: 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 11 shows the `angular.json` configuration.

```
1 ...  
2 "build": {  
3   "builder": "@angular-devkit/build-angular:browser",  
4   "options": {  
5     "outputPath": "../../webapp",  
6   }  
}
```

Listing 11: `angular.json` configuration

---

<sup>6</sup><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.