



*Università di Trento*

*Web Architectures*

---

## Assignment 5

---

*Author:*

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

February 2, 2022

# 1 Introduction

The fifth assignment is about creating an application where tourists can book and apartment or a hotel room, depending on its availability. On this application, the user can search for accommodations 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 implemented 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 Design

This section describes the most important parts of the implementation of the application. It highlights the database 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

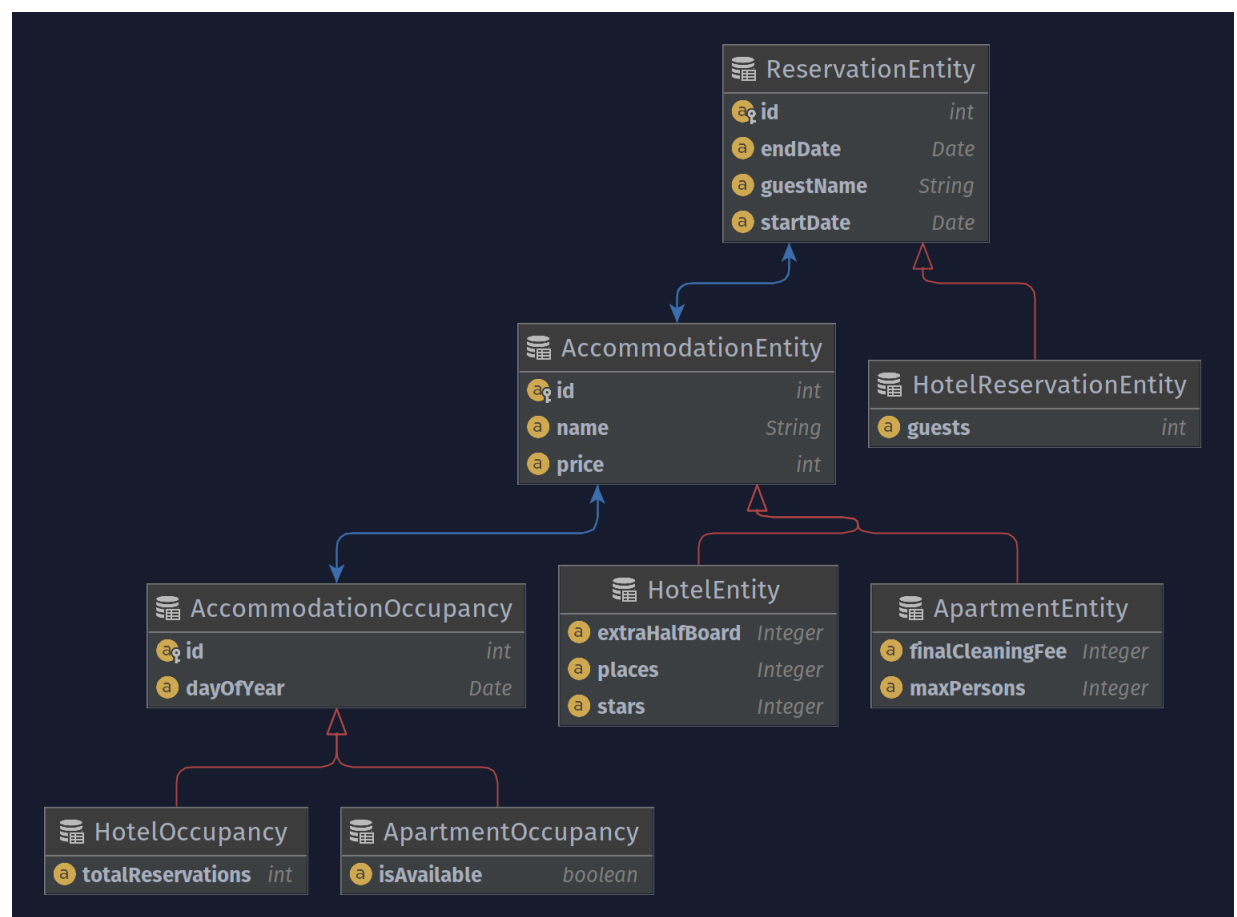


Figure 1: UML diagram of all database entities

**Accommodations** Two types of accommodations exist for this application: Apartments and Hotels. All accommodations have a name, and a daily price. Additionally, an apartment has a final cleaning fee, and a number of maximum persons. A hotel has a rating of stars, 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 reservations, and an apartment is either available or unavailable, independently 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 occupancy information for a hotel, which includes a boolean value called `isAvailable` that specifies, if the apartment is available for the specific date or not. Furthermore, a class called `HotelEntity` represents the occupancy of a hotel, which is a number of reservations for a specific date.

**Reservations** A user can make reservations 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 class 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 available or not, which is represented by the existence 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 accommodations 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 Bean. 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, by 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.

```

1 @Stateless
2 @Remote(ReservationService.class)
3 public class ReservationBean implements ReservationService {
4
5     @EJB()
6     private LocalDatabaseBean databaseBean;
7
8     ...
9
10 }

```

Listing 1: Usage of the LocalDatabaseBean using Code-Injection

### 2.2.2 AccommodationService

The Accommodation is used by the client to interact with the `AccommodationEntity` 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 occurrences of an accommodation which is available in the specific date range, has to be equal to 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 shows 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 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 that day, and therefore the availability information for that 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 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.

```
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 executes the EJB's (introduced in SEC AB) 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 Patterns

### 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 FIG XY.

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*.

### 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 as illustrated in FIG AB. Otherwise, a grid of all available results, order by the daily price, is presented to the user, as shown in FIG AB. 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 one including half-board.

To receive a list of all available accommodations, the *AccommodationResultServlet* delegates the request details to the *AccommodationService* (introduced in Section 2.2.2) to call the `getAvailableAccommodations` method of the Bean. The total prices are calculated by delegating the details to the `calculateApartmentReservationPrice` and `calculateHotelReservationPrice` methods of the *ReservationService* (introduced in Section 2.2.3) respectively.

### 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 FIG AB. 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*.

To get the information about the accommodation, the servlet uses the *AccommodationService* bean, and to get the total price of the reservation it uses the *ReservationService* bean respectively.

### 2.3.5 ReservationConfirmServlet

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



To persist a new *ReservationEntity* to the database, the *ReservationConfirmServlet* delegates the details to the *ReservationService* to call the `persistReservation` method.

### 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 2.3.3), shown in FIG XY. After submitting, the *ReservationListServlet* sends a POST request to itself to present all reservation made by the given user, shown in FIG AB.

To present all needed informations, the *ReservationListServlet* uses the *AccommodationDelegate* and *ReservationDelegate* respectively.

## 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 2. 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 2: 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 3, 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 3: 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 4. If the connection is successful, the database can be used in the application.

## 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 4 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"
```

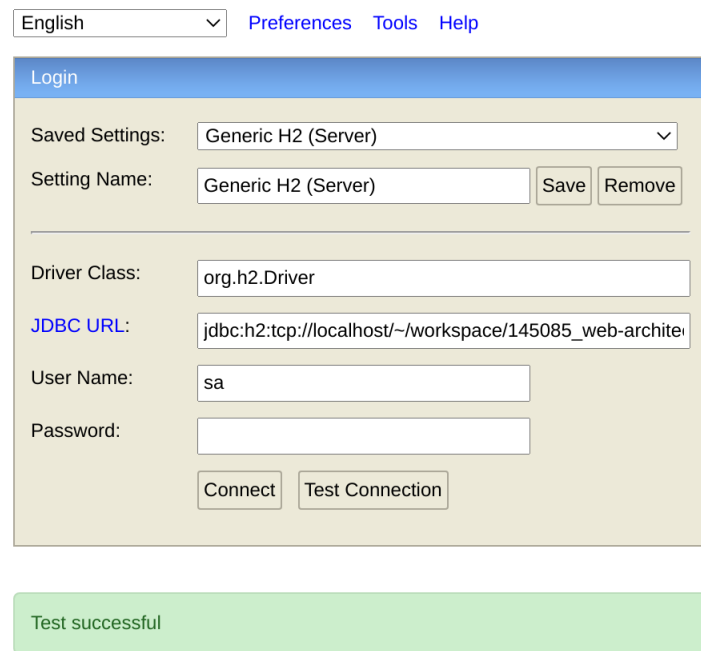


Figure 4: Successfull connection test for the H2 database

```

5         value="jdbc:h2:tcp://localhost/~workspace/145085
        _web-architectures/assignment_5/accommodations"/>
6     </properties>
7 </persistence-unit>
8 ...

```

Listing 4: 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 5 shows the newly created tables in the 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.

DISC	ID	NAME	PRICE	EXTRA_HALF_BOARD	PLACES	STARS	FINAL_CLEANING_FEE	MAX_PERSONS
APARTMENT	1	Pietra Bianca	40.0	null	null	15.0		4
APARTMENT	30	Tenuta Di Arimino	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	null
HOTEL	117	Artemide	100.0	20.0	60	4	null	null
HOTEL	146	Zenith	70.0	18.0	40	3	null	null
HOTEL	175	Majestic	65.0	15.0	50	4	null	null

Figure 5: Successfull connection test for the H2 database

### 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 `\$JBOSS\_HOME/standalone/configuration/standalone.xml`, where `$JBOSS_HOME` is the path to WildFly. Listing 5 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
10      =-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
11      <driver>h2</driver>
12      <security>
13        <user-name>sa</user-name>
14        <password>sa</password>
15      </security>
16    </datasource>
17    ...
18  </datasources>
19 </subsystem>
20 ...

```

Listing 5: 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 6.

```

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 6: 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 6.

```

~/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 6: 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 7, 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 7.

```

1 $ cp target/marcel-stolin-web-services.jar $JBOSS_HOME/standalone/
   deployments

```

Listing 7: Command to copy the EJB JAR artifact to the WildFly deployment directory

### 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 *AccommodationSer-*

```
~/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 7: Process to deploy the EJB JAR artifact to the WildFly application server

vice (mentioned in Section 2.2.2) and the *ReservationService* (mentioned in Section 2.2.3), which is illustrated in Figure 8.

```
09:38:23,130 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.univ.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    java:app/marcel-stolin-web-services/AccommodationBean!it.univ.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    java:module/AccommodationBean!it.univ.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    java:jboss/exported/marcel-stolin-web-services/AccommodationBean!it.univ.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    ejb:/marcel-stolin-web-services/AccommodationBean!it.univ.disi.webarch.mstolin.webservices.accommodations.AccommodationService
    java:global/marcel-stolin-web-services/AccommodationBean
    java:app/marcel-stolin-web-services/AccommodationBean
    java:module/AccommodationBean

09:38:23,130 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.univ.disi.webarch.mstolin.webservices.reservations.ReservationService
    java:app/marcel-stolin-web-services/ReservationBean!it.univ.disi.webarch.mstolin.webservices.reservations.ReservationService
    java:module/ReservationBean!it.univ.disi.webarch.mstolin.webservices.reservations.ReservationService
    java:jboss/exported/marcel-stolin-web-services/ReservationBean!it.univ.disi.webarch.mstolin.webservices.reservations.ReservationService
    ejb:/marcel-stolin-web-services/ReservationBean!it.univ.disi.webarch.mstolin.webservices.reservations.ReservationService
    java:global/marcel-stolin-web-services/ReservationBean
    java:app/marcel-stolin-web-services/ReservationBean
    java:module/ReservationBean

09:38:23,130 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.univ.disi.webarch.mstolin.webservices.database.LocalDatabaseBean
    java:app/marcel-stolin-web-services/LocalDatabaseBean!it.univ.disi.webarch.mstolin.webservices.database.LocalDatabaseBean
    java:module/LocalDatabaseBean!it.univ.disi.webarch.mstolin.webservices.database.LocalDatabaseBean
    java:global/marcel-stolin-web-services/LocalDatabaseBean
    java:app/marcel-stolin-web-services/LocalDatabaseBean
    java:module/LocalDatabaseBean
```

Figure 8: 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 9. 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.

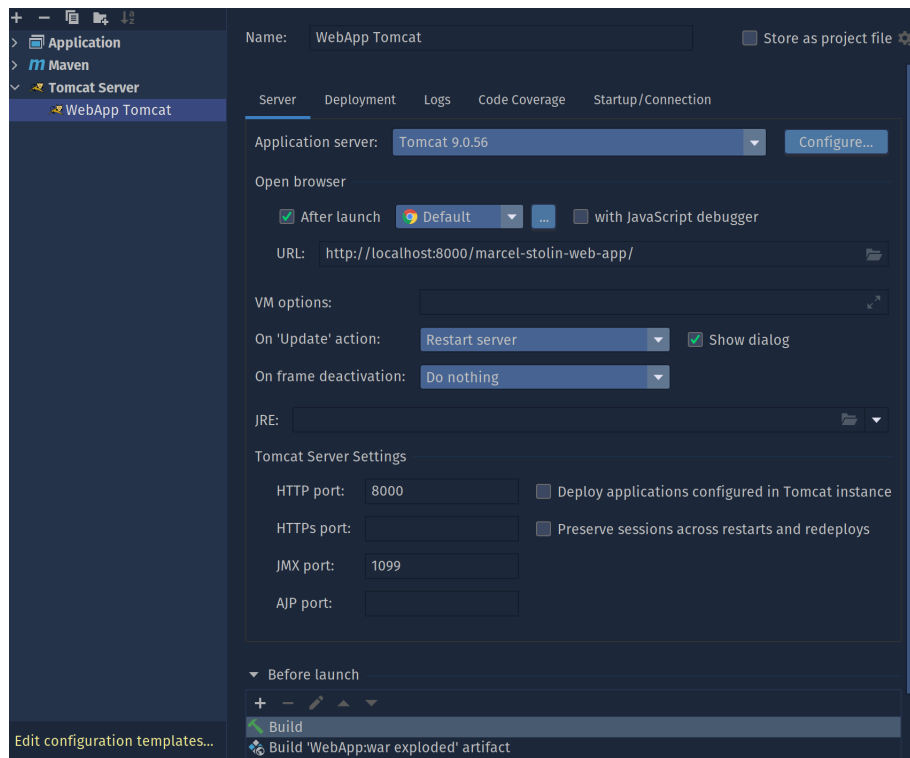


Figure 9: IntelliJ Tomcat Configuration Server Settings

### 3.4.2 Deployment Settings

Figure 10 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`.

### 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/`.



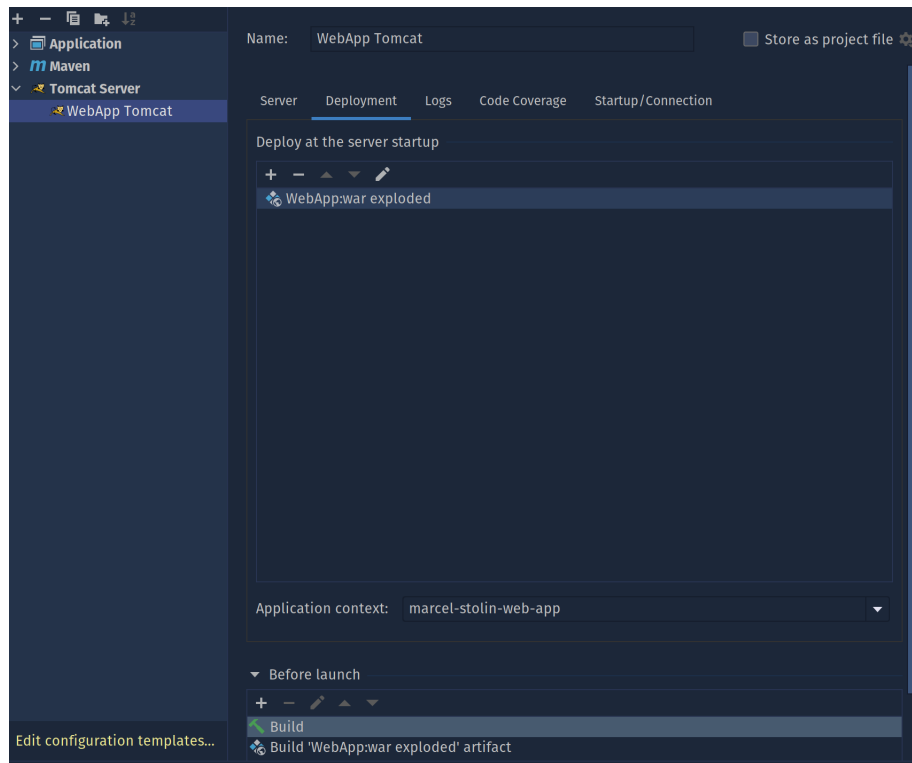


Figure 10: IntelliJ Tomcat Configuration Deployment Settings

## 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 8 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})],

```

<sup>6</sup><https://angular.io/api/common/HashLocationStrategy>

```
8   exports: [RouterModule]
9 })
```

Listing 8: 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 9 shows the `angular.json` configuration.

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

Listing 9: `angular.json` configuration

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.