Jakarta EE Tutorial

All sample using in the classroom will be added to the multi-project `~/development/ffhs/jea/jea-classroom`.

Build first Hello App:

Setup Environment

Enable Java Faces

Implement first Java Facelet

Add Database Access

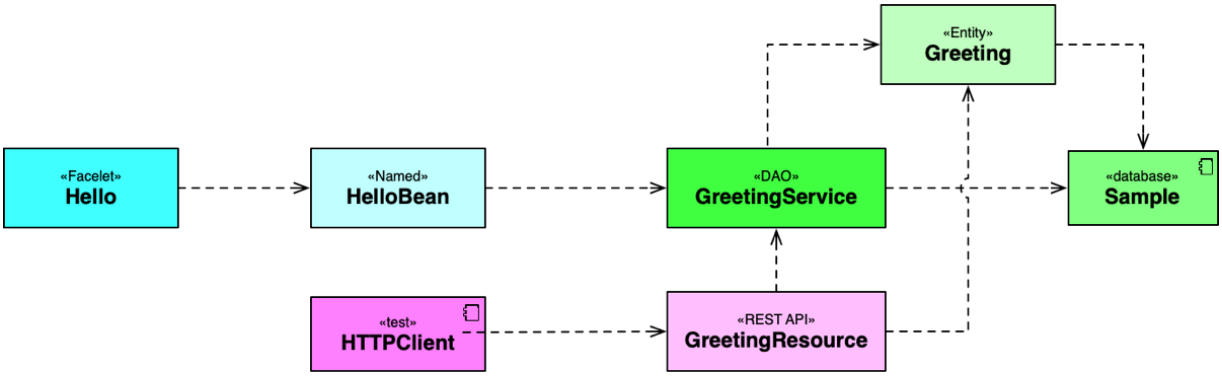
Test Data Access

Using HTTP Client

Extend Web Form

Overview

Our sample looks as follow



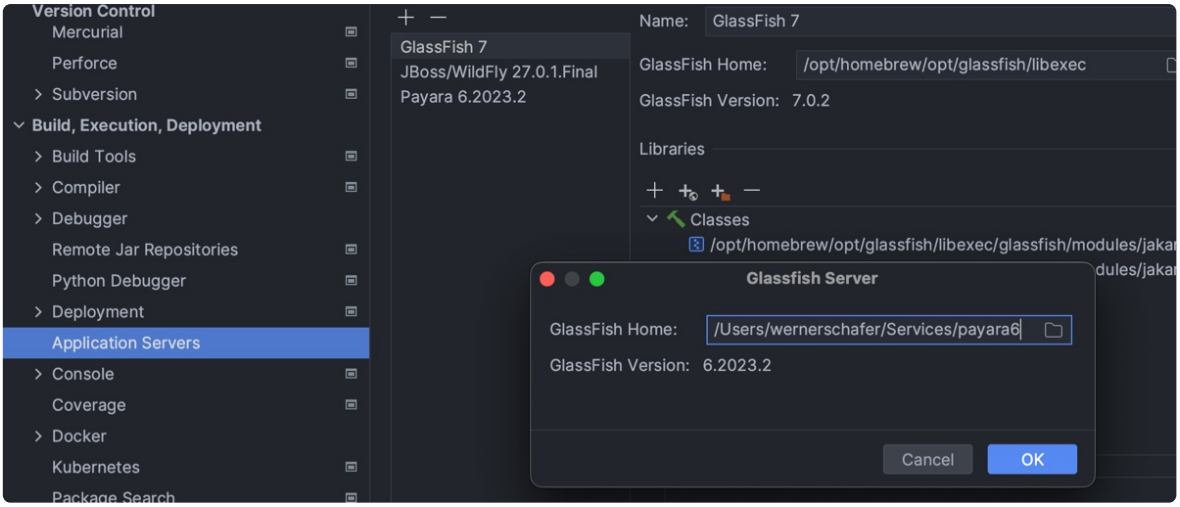
Setup Environment

- Install IntelliJ Ultimate (version 2023.1 or newer) and Gradle
- Install Glassfish app server 7 (download). You can use Homebrew on macOS to install and keep it up-to-date or download the zip file and copy the content into a specific folder. Add the following environment variable to your login script `export GLASSFISH_HOME=<glassfish-home-dir>`.

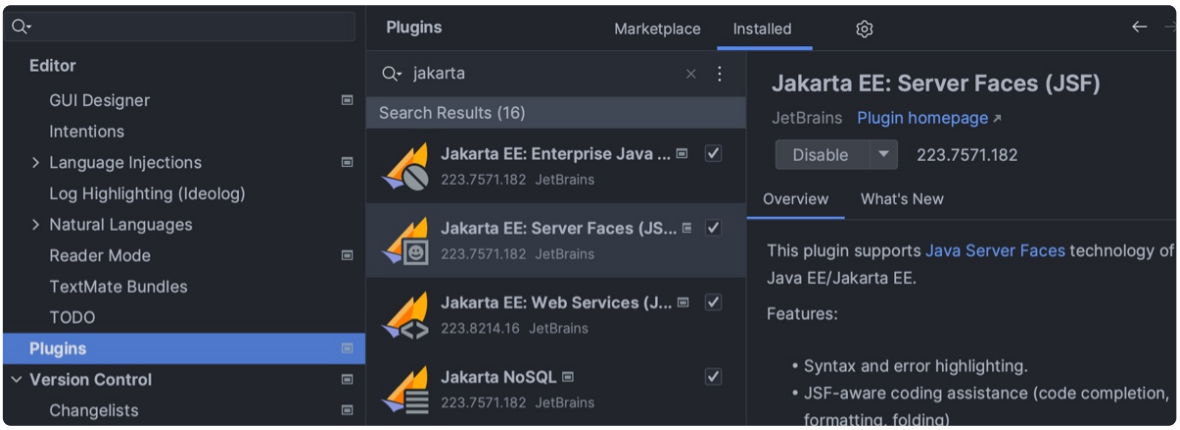
- Recreate the default created domain1 to change the default port 8080 to 8090. This gives you less conflicts with other tools using the same *standard* port. With the command `asadmin list-domains` we can list the available domains. With the command `asadmin version` examine what version and application server is running.

```
asadmin delete-domain domain1
asadmin create-domain --instanceport 8090 --adminport 4850 domain1
# You may start your domain already in the background
asadmin start-domain domain1
```

- Add this Glassfish application server to IntelliJ in the settings properties.



- Make sure the following plugins are installed (using settings properties)



- Install your favour database for example PostgreSQL. In this sample we are using MariaDB (because the current version 15.2 gives trouble starting it). Install MariaDB: under macOS you can use Homebrew to install and start the MariaDB daemon. Setup the database samples as follow with the command `mariadb`.

```
create database samples;
create user admin identified by 'admin';
use samples
grant usage on *.* to 'admin'@'localhost' identified by 'admin';
grant all privileges on *.* to 'admin'@'localhost';
```

- Download and copy the corresponding JDBC driver into Glassfish. For PostgreSQL and for MariaDB.

```
# Glassfish
cp postgresql-42.5.4.jar $GLASSFISH_HOME/glassfish/domains/domain1/lib
cp mariadb-java-client-3.1.2-sources.jar $GLASSFISH_HOME/glassfish/
domains/domain1/lib
# Payara
cp postgresql-42.5.4.jar $PAYARA_HOME/glassfish/domains/domain1/lib
cp mariadb-java-client-3.1.2-sources.jar $PAYARA_HOME/glassfish/
domains/domain1/lib
# Restart server if running
asadmin restart-domain domain1
```

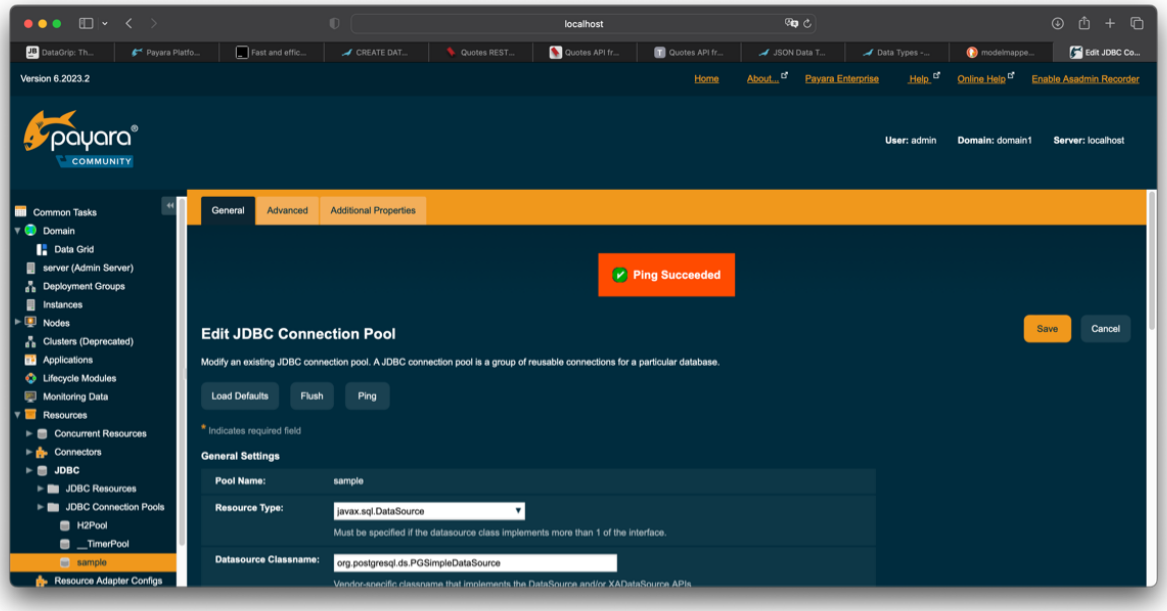
- Create the JDBC connection pool and resources inside the application server using either the admin web console or the `asadmin` tool. Using either you have to start the domain server so that the admin server for that domain is running.

  You may access the web console via http://localhost:4850/common/index.jsf · Image · Or in Glassfi...

```
# Overwrite default asadmin (found by PATH)
export PATH=$PAYARA_HOME/bin:$PATH
# check if the domain is running by list its application
asadmin list-applications --port 4850
# Start the domain if necessary
asadmin start-domain domain1
# Add JDBC connection pool and resource for our sample database (we
will create later)
asadmin --port 4850 create-jdbc-connection-pool --datasourceclassname
org.postgresql.ds.PGSimpleDataSource \
  --restype javax.sql.DataSource --property
```

```
portNumber=5432:password=admin:user=admin:serverName=localhost:databas
eName=sample \
  sample
asadmin --port 4850 create-jdbc-resource --connectionpoolid sample
jdbc/sample
```

- With the admin console running under the port 4850 you can check the connection by sending a Ping

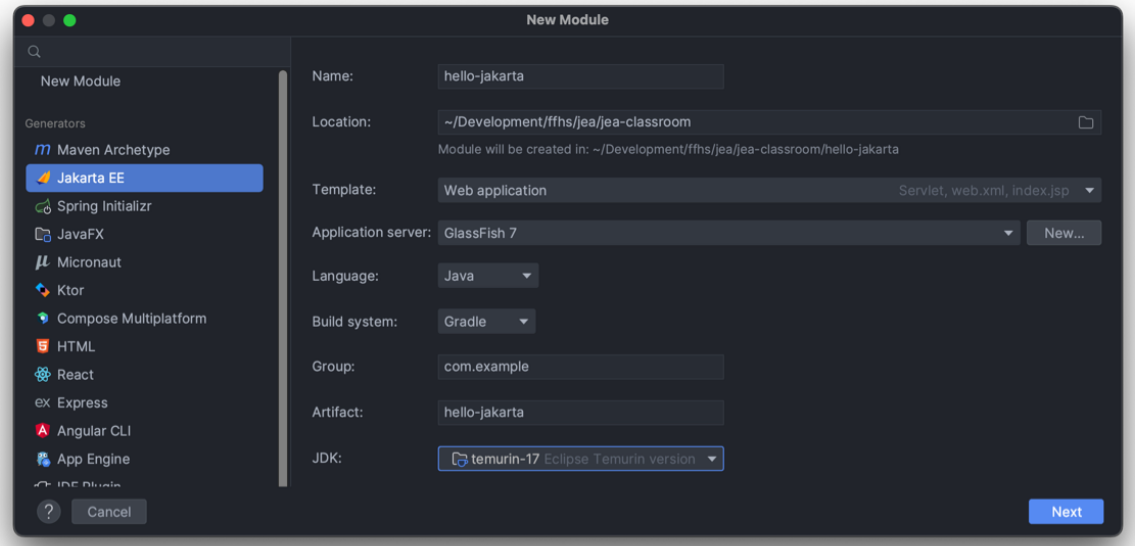


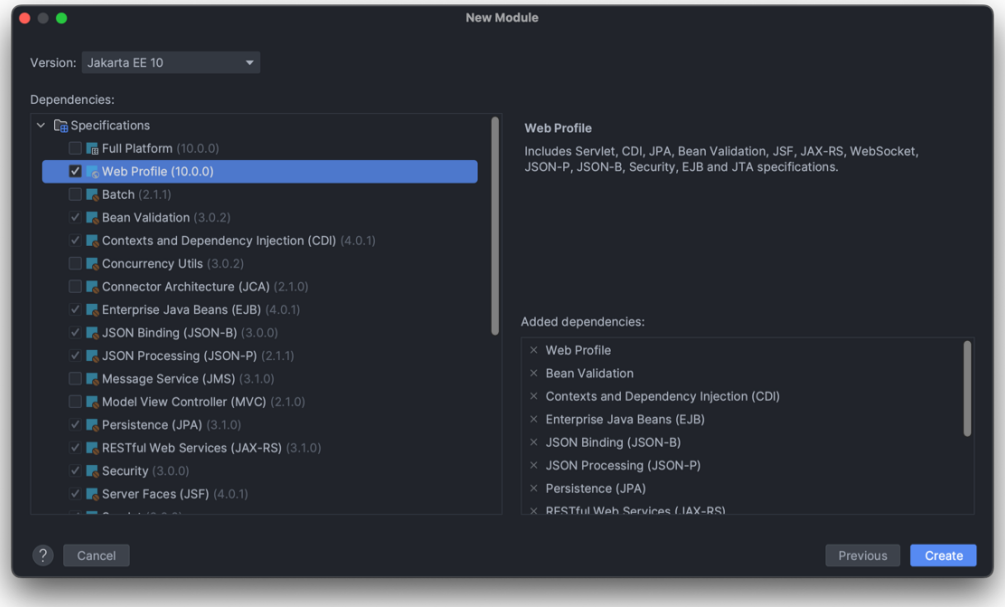
- With Glassfish define the `GLASSFISH_HOME` to the Glassfish root directory and add the `bin` folder to the `PATH`.

Create Web application

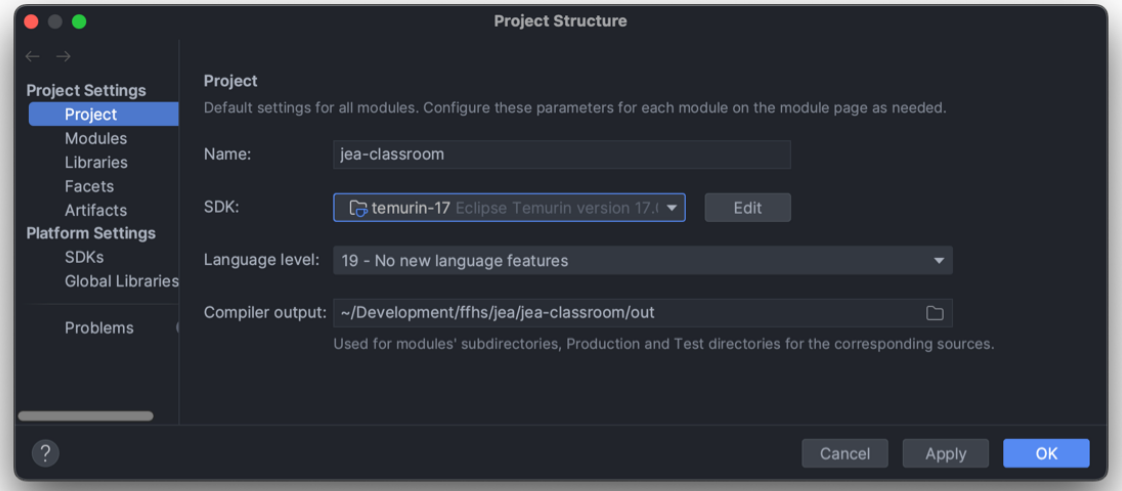
Setup new project

In IntelliJ create a new project and select *Jakarta EE* generator. In the **first window** name your application and choose the Web-application template and Gradle as build-system.

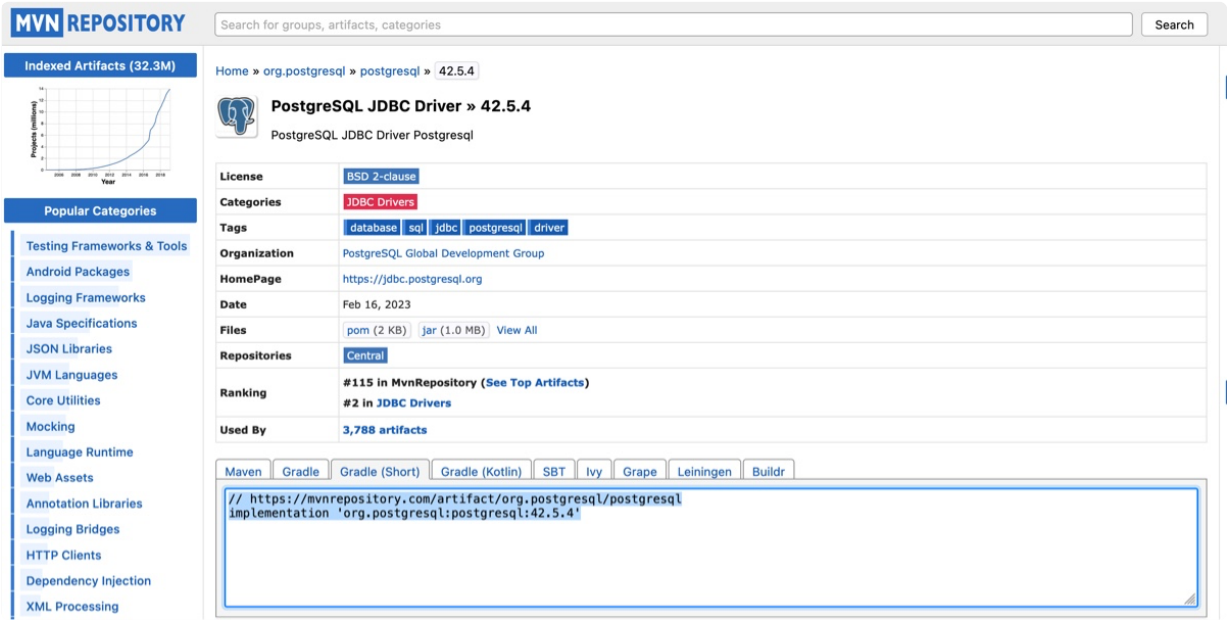




In the **second window** make sure you choose Jakarta EE 10 and select the Web Profile to include all necessary dependencies for a web application with a UI and database access. After the new project is created, make sure that the right **JDK** is selected in the Project Structure under the section Project.



Add the right JDBC driver to your Gradle build script. The actual version can be found in the Maven Central repository. Search there for example for the PostgreSQL JDBC driver. From there you can select the right dependency format.



And add it to your dependency clause in your `gradle.build` script.

```
dependencies {
    compileOnly('jakarta.platform:jakarta.jakartaee-web-api:10.0.0')
    implementation 'org.postgresql:postgresql:42.5.4'
    //..
}
```

Run/Debug Configuration

By adding configuration in IntelliJ we specify how the program is executed using a specific runtime environment. But first we overwrite the default target name of the War file in our `build.gradle` script by adding the following lines:

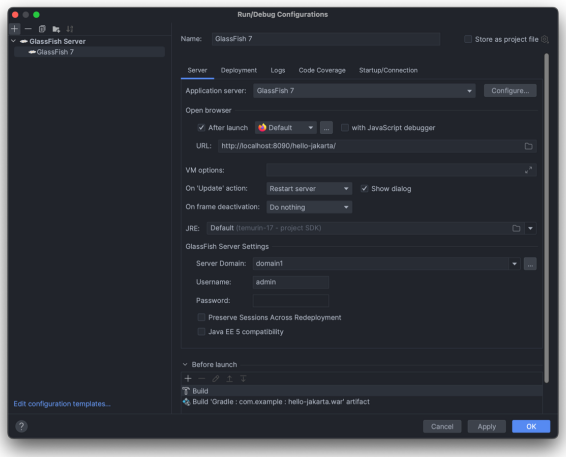
```
war {
    archiveFileName.set("${project.name}.war")
}
```

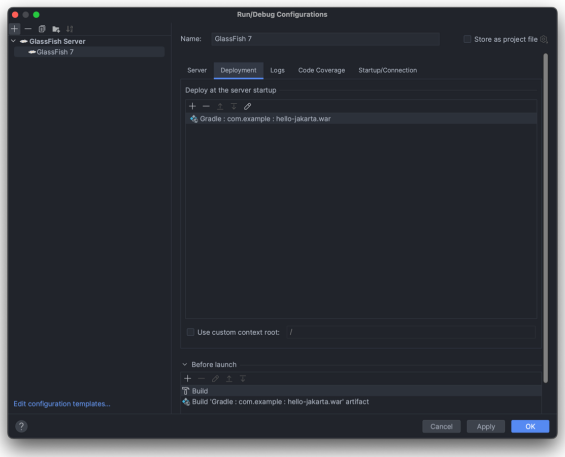
By default the target name is a combination of the project-filename and its version. By overwriting the war filename we simplify the target name to the project-name. After chancing the gradle script to have to refresh gradle in IntelliJ.

Now we have to possibilities to execute a Jakarta web application: a) we can start the application server (domain) each time from IntelliJ or b) we only deploy the target war file into the auto-deploy folder of the application server.

a) Run the application server from IntelliJ

For that add either a Glassfish/local or a Payara/local configuration. In the first tab *Server* select the right domain and in the second tab *Deployment* select the right war to be deployed.





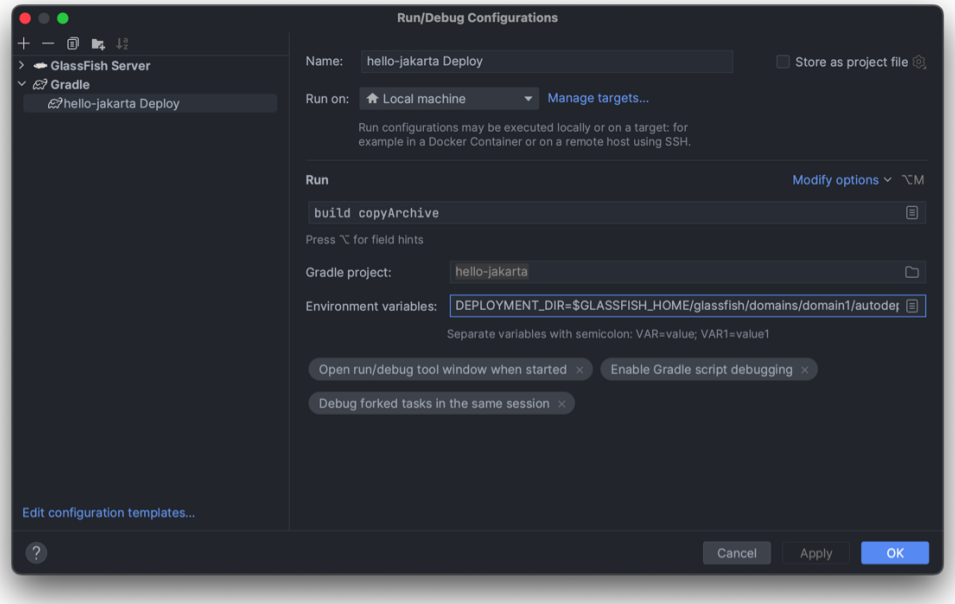
To start and run the application server from IntelliJ make sure the application server is not already running. Now you can start the app from IntelliJ. Depending of your configuration the app server is always restarted or an update of the war is only redeployed without restarting the server.

b) Deploy war to running app server

In this option we only rebuild the target War file and copy it then to the auto-deploy folder of the running application server. For that first add the following task to your `build.gradle` script.

```
tasks.register("copyArchive", Copy) {
    dependsOn(war)
    def archiveFile = war.archiveFileName.get()
    from ("$buildDir/libs") {
        include "**/${archiveFile}"
    }
    into "$System.env.DEPLOYMENT_DIR"
    doNotTrackState("Installation directory contains unrelated files")
    println("Copyed file $buildDir/libs/${archiveFile}")
}
```

Now we add a run/debug configuration for a Gradle task. In the gradle script we expected that the deployment folder is defined by the DEVELOPMENT_DIR environment variable.



This configuration expect that the application server (domain) is already running and the Gradle build script update and deploy the War file to the auto-deploy folder. Using Glassfish domain1 app server the DEPLOYMENT_DIR will be defined as follow

```
export DEPLOYMENT_DIR=$GLASSFISH_HOME/glassfish/domains/domain1/
autodeploy
```

It may be easier to add the plugin for an embedded browser to test the web UI instead of open an external browser each time the app is started.

Enable Java Faces

To enable Java Faces you can either create any Java class and annotate it as follow. We also add the Producer for a Logger to log information. This informs the framework what Logger class is to inject when the Logger is referenced.

```
@FacesConfig
public class AppContext {
    @Produces
    public Logger getLogger() {
        return Logger.getLogger("hello");
    }
}
```

Another way is to add the the `faces-config.xml` to `webapp/WEB-INF` folder.
You can do it by use the new context menu and there select to create a JSF Config file · Image

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="4.0" xmlns="https://jakarta.ee/xml/ns/jakartaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                                  https://jakarta.ee/xml/ns/jakartaee/
web-facesconfig_4_0.xsd" >

</faces-config>
```

The faces-config.xml is further used to configure JF behaviours.

Implement first Java Facelet

Select `webapp` folder in the project tree and in the context menu new select `JSF/Facelets` menu item. There enter the name of the file (we choose for your first web form `hello`). Unfortunately IntelliJ version 2022.3 generates a template for JSF 3.0 valid for Jakarta EE 9 and JEE 8. Because Jakarta EE 10 introduce a major upgrade of JSF towards Java Faces 4.0 we have to reformat the generate to the following format (namely the header changed).

```
<!DOCTYPE html>
<html lang="en"
      xmlns:f="jakarta.faces.core"
      xmlns:h="jakarta.faces.html">
<h:head>
</h:head>
<h:body>
</h:body>
</html>
```

hello.xhtml

What we want is to have a simple form that ask for our same and returns a greeting to a variable below that form. For that we first implement the HelloBean class to provide this functionally,

```
@Named
@RequestScoped
public class HelloBean {
    private String input;
    private String output;

    public void submit() {
        output = String.format("Hello %s to the Jakarta World", input);
    }

    //region getter/setter
    public String getInput() {
        return input;
    }

    public void setInput(String input) {
        this.input = input;
    }
```

```
    public String getOutput() {
        return output;
    }
    //endregion
}
```

HelloBean.java

Now we can implement the Hello web form. We are using the *PanelGrid* to format the form.

```
<!DOCTYPE html>
<html lang="en"
      xmlns:f="jakarta.faces.core"
      xmlns:h="jakarta.faces.html">
<h:head>
</h:head>
<h:body>
    <h1>Hello World</h1>
    <h:form>
        <h:panelGrid columns="2">
            <f:facet name="header">
                <h:outputText value="Greeting"/>
            </f:facet>
            <h:outputLabel for="input" value="Input"/>
            <h:inputText id="input" value="#{helloBean.input}"
required="true"/>
            <f:facet name="footer">
                <h:panelGroup style="display:block; text-align: center">
                    <h:commandButton value="Submit"
action="#{helloBean.submit}">
                        <f:ajax execute="@form" render=":output"/>
                    </h:commandButton>
                </h:panelGroup>
            </f:facet>
        </h:panelGrid>
    </h:form>
    <h:outputText id="output" value="#{helloBean.output}"/>
</h:body>
</html>
```

hello.xhtml

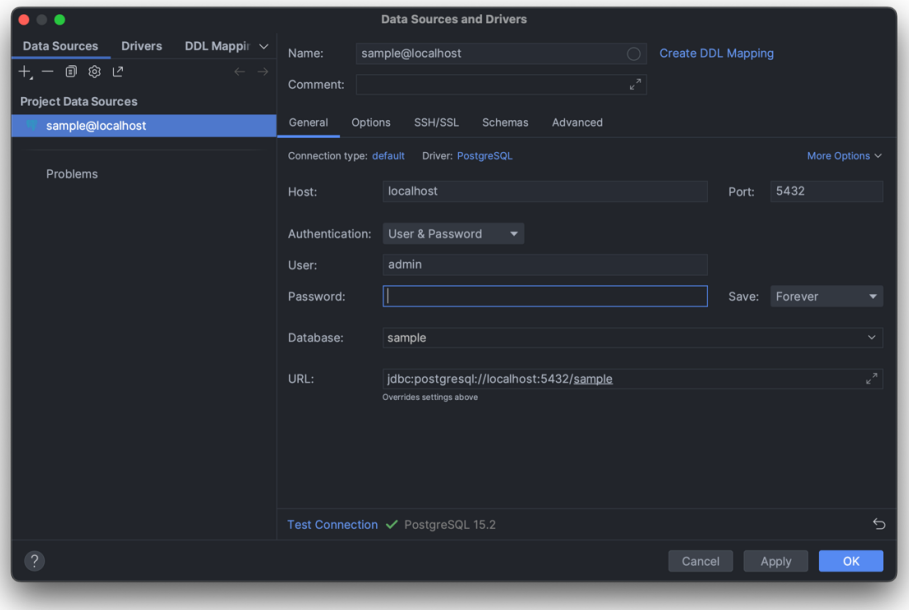
By default IntelliJ creates a index.jsp file that is displayed when the applicate starts. There we add the reference to our new Hello web form.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
<head>
    <title>JSP - Hello World</title>
</head>
<body>
<h1><%= "Hello World!" %>
</h1>
<br/>
<a href="hello-servlet">Hello Servlet</a><br>
<a href="hello.xhtml">Hello Facelet</a>
</body>
</html>
```

index.jsp

Add Database Access

To use the PostgreSQL database sample we created before, it may be useful to add the Data Source in IntelliJ.



Now we extend our simple hello world sample by storing and retrieving the message from a PostgreSQL database. Before we already installed the JDBC driver into the app server and added to your dependencies. In the add server we added the JDBC resources `jdbc/sample` . Now we add the persistence unit to our `persistence.xml` script.

```
<persistence-unit name="sample">
    <jta-data-source>jdbc/sample</jta-data-source>
    <properties>
        <property name="jakarta.persistence.jdbc.driver"
value="org.postgresql.Driver"/>
        <property name="jakarta.persistence.jdbc.user" value="admin"/>
        <property name="jakarta.persistence.jdbc.password" value="admin"/
>
        <property name="jakarta.persistence.schema-
generation.database.action" value="create"/>
    </properties>
</persistence-unit>
```

Then we add the Greeting entity that returns a greeting quote depending of the language. To make it easier to define setter and getter for properties attributes in Java you can use the lombok project. Add the following dependency to your gradle build script. You have to add also the *annotationProcessor* dependency.

```
dependencies {
    compileOnly 'org.projectlombok:lombok:1.18.26'
    annotationProcessor 'org.projectlombok:lombok:1.18.26'
    //...
}
```

Now you can use the annotation `@Getter` and `@Setter` to automatically generate the getter and setter for your class attributes.

```
@Entity
@Table(name="greetings")
@Getter @Setter
public class Greeting {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;
    @Column(length = 2, nullable = false)
```

```
    private String lang;
}
```

Greeting.java

After implement the entity class that represent a database table we add a DAO (often named as repository or service) that represent the access controller to that entity.

```
@Stateless
public class GreetingService {
    @PersistenceContext(name = "sample")
    private EntityManager entityManager;

    public List<Greeting> getAll() {
        return entityManager.createQuery("select g from Greeting g",
Greeting.class).getResultList();
    }
}
```

GreetingService.java

Test Data Access

In Jakarta there is currently no an easy way to test JPA entities and services link in Spring. To test classes using CDI you as use the approach described in link where we are using the CDI Test Core dependency. For using Mockito we have to add the Mockito Core and the Mockito Extension for Junit to our dependencies. For an sample using CDI-Test see link.

Test Test Implementation look likes · Snippet · One one side we cannot use the JAX-RS Response type,...

It is much easier and more efficient to use REST API to test the Service to access the database. So let us implement such a services we can then called from the shell using *Curl* or *HTTPie*. To use JAX-RS REST API we have first extend our *AppContext* (inherit from Application and annotate with the ApplicationPath).

```
@FacesConfig
@ApplicationPath("/api")
@ApplicationScoped
public class AppContext extends Application {
    @Produces
    public Logger getLogger() {
        return Logger.getLogger("hello");
    }
}
```

AppContext.java

Now we implement the *GreetingResource* as REST API to access the *GreetingRepository*.

```
@Path("/greeting")
public class GreetingResource {
    @Inject
    private GreetingService greetingService;

    @Inject
    private Logger logger;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Greeting> getAll() {
        List<Greeting> greetings = greetingService.getAll();
        logger.info(String.format("Got %d greetings", greetings.size()));
        return greetings;
    }

    @POST
    public Response addGreeting(Greeting greeting) {
        var savedGreeting = greetingService.addGreeting(greeting);
        logger.info(String.format(
                "Add greeting '%s', lang='%s' under %d",
                savedGreeting.getText(),
                savedGreeting.getLang(),
                savedGreeting.getId()));
         return Response
                 .status(Status.OK.getStatusCode(), "Greeting added")
                 .build();
    }
}
```

GreetingResource.java

Run the app by deploying the War to the application server and then we can access the REST API from the shell as

`http :8090/hello-jakarta/api/greeting`

Using HTTP Client

To repeated and more sophisticated REST call we can use the HTTP Client tool in IntelliJ. Create one and add following tests. After you created a HTTP Client file, which are stored by default in the global *scratches* folder you can move it to your project folder to make it private. For a detail description of the HTTP Client see link.

For that sample we complete the Greeting Services and Resource

Greeting Service (DB Access) · Snippet · GreetingService.java · Greeting Resource (REST API) · Snippe...

Now we can implement our HTTP Client that is a series of accesses to get all entries, to add one, to find that and finally to delete it.

```
### Get all
GET {{host}}/api/greeting
Accept: application/json

### Get by ID
GET {{host}}/api/greeting/1
Accept: application/json

### Add greeting
POST {{host}}/api/greeting
Content-Type: application/json

{
  "lang": "ru",
  "text": "добро пожаловать"
}

> {%
    client.global.set("added-greeting-id", response.body["id"]);
    client.log(client.global.get("added-greeting-id"))
%}

### Get added record by ID
GET {{host}}/api/greeting/{{added-greeting-id}}

### Delete an entry
```

```
DELETE {{host}}/api/greeting/{{added-greeting-id}}

### Search for an etry
POST {{host}}/api/greeting/search
Content-Type: application/json
Accept: application/json

{
  "lang": "de",
  "text": "willkommen"
}

> {%
    client.global.set("found-greeting-id", response.body["id"]);
    client.log(client.global.get("found-greeting-id"))
 %}

### Get found greeting
GET {{host}}/api/greeting/{{found-greeting-id}}
```

HTTP Client file

Extend Web Form

After we implement the JPA database access and test it, we extend our Hello sample form by adding the display of the content of the Greeting table we can select from. First we implement additional methods in our DAO *GreetingService*.

```
@Stateless
public class GreetingService {
    //...

    public Greeting findByLang(String lang) {
        return entityManager
                .createQuery("select g from Greeting g where g.lang
= :lang", Greeting.class)
                .setParameter("lang", lang)
                .getSingleResult();
    }

    public List<String> getLanguages() {
        return entityManager.createQuery("select distinct g.lang from
Greeting g", String.class).getResultList();
```

```
    }
}
```

And then we extend the *HelloBean* class to provide the necessary values to web form.

```
@Named
@RequestScoped
public class HelloBean {
    private String input;
    private String output;
    private String lang;

    @Inject
    private GreetingService greetingService;

    @Inject
    private Logger logger;

    public void submit() {
        try {
            var greeting = greetingService.findByLang(lang);
            output = String.format("%s %s", greeting.getText(), input);
            logger.info("Set output to " + output);
        } catch (NoResultException | NonUniqueResultException ignored) {
            FacesContext.getCurrentInstance().addMessage(null,
                    new FacesMessage(
                            FacesMessage.SEVERITY_ERROR,
String.format("No entry for language %s", lang),
                            null));
        }
    }

    public List<Greeting> getGreetings() {
        return greetingService.getAll();
    }

    public List<String> getLanguages() {
        return greetingService.getLanguages();
    }

    //...
}
```

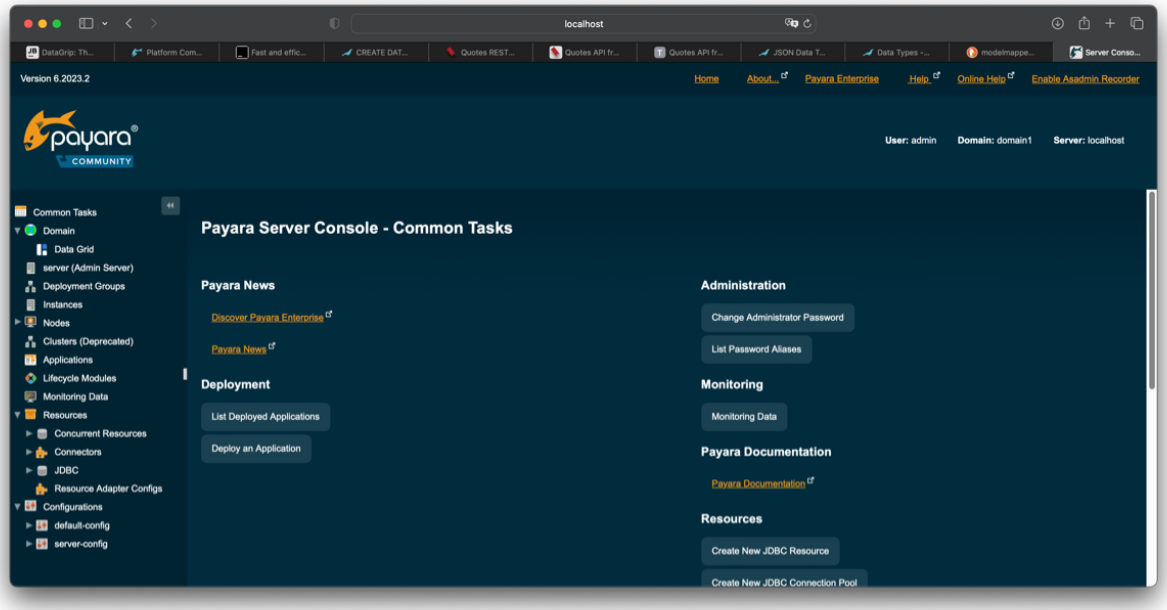
And ad leased we extend our web form *hello.xhtml*.

```
<h:body>
    <h1>Hello World</h1>
    <h:form>
        <h:panelGrid columns="2">
            <f:facet name="header">
                <h:outputText value="Greeting"/>
            </f:facet>
            <h:outputLabel for="input" value="Input"/>
            <h:inputText id="input" value="#{helloBean.input}"
required="true"/>
            <h:outputLabel for="languages" value="Language"/>
            <h:selectOneMenu id="languages" value="#{helloBean.lang}">
                <f:selectItems value="#{helloBean.languages}"/>
            </h:selectOneMenu>
            <f:facet name="footer">
                <h:panelGroup style="display:block; text-align: center">
                    <h:commandButton value="Submit"
action="#{helloBean.submit}">
                        <f:ajax execute="@form" render=":output"/>
                    </h:commandButton>
                </h:panelGroup>
            </f:facet>
        </h:panelGrid>
    </h:form>
    <h:outputText id="output" value="#{helloBean.output}"/>
    <h3>Greeting forums</h3>
    <h:dataTable id="table" value="${helloBean.greetings}"
var="greeting">
        <h:column>${greeting.lang}</h:column>
        <h:column>${greeting.text}</h:column>
    </h:dataTable>
</h:body>
```

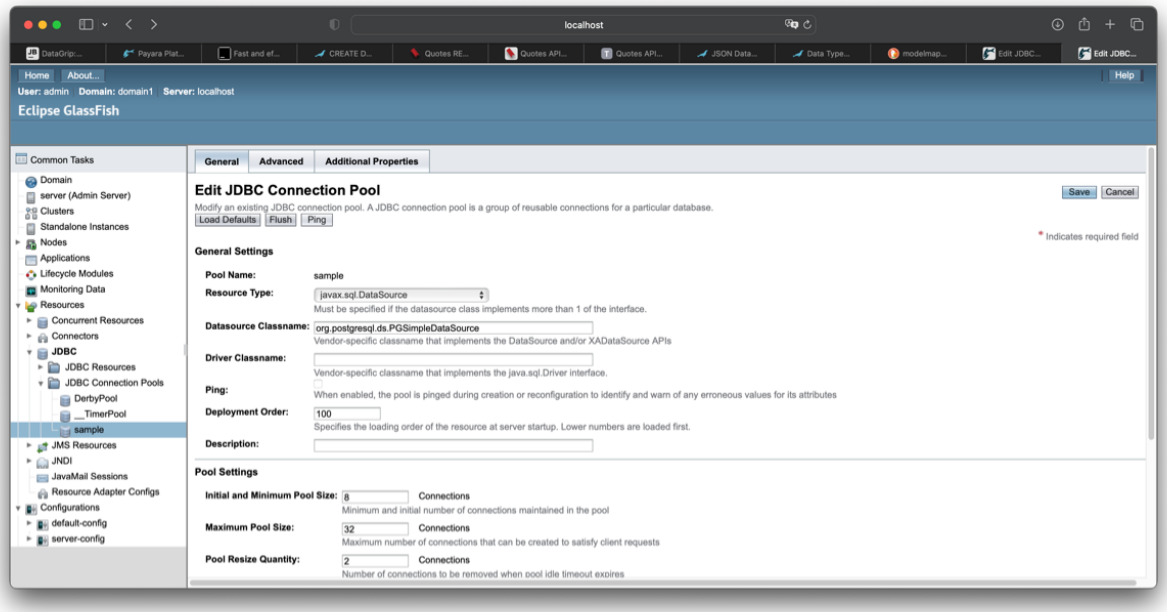
↑ Jakarta EE Tutorial

Create the JDBC connection pool and resources inside the application server using either the admin web console or the asadmin tool. Using either you have to start the domain server so that the admin server for that domain is running.

You may access the web console via http://localhost:4850/common/index.jsf



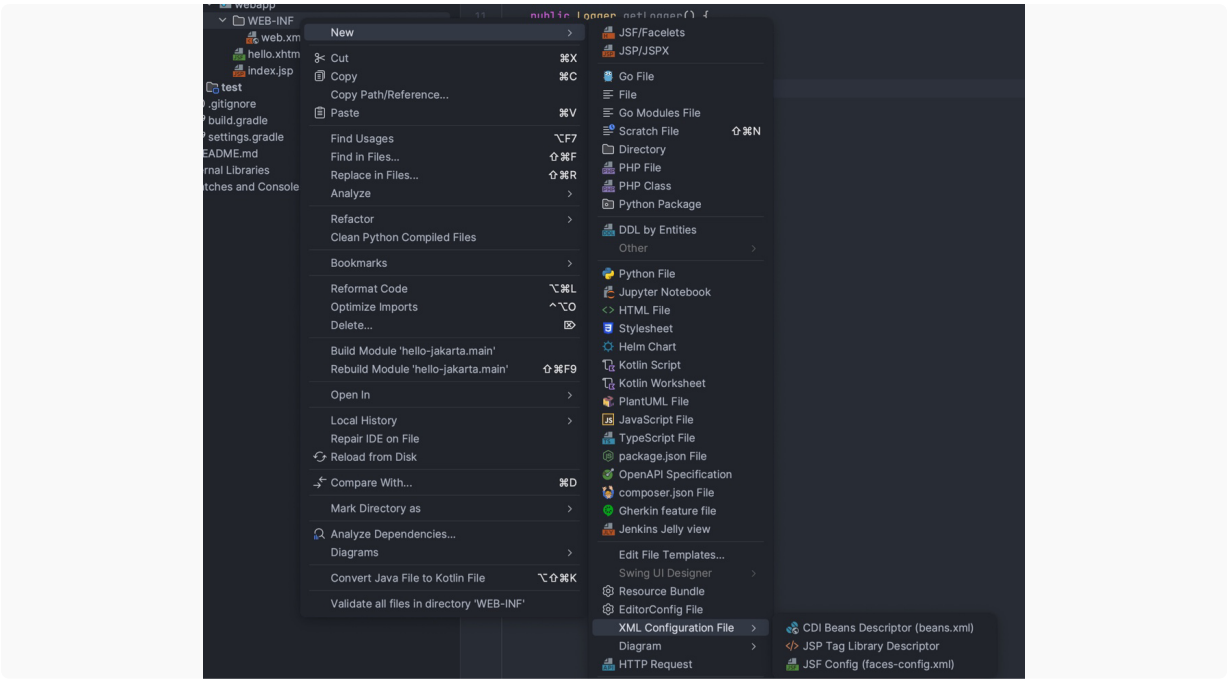
Or in Glassfish



↑ Jakarta EE Tutorial

Another way is to add the the faces-config.xml to webapp/WEB-INF folder.

You can do it by use the new context menu and there select to create a JSF Config file



↑ Jakarta EE Tutorial

In Jakarta there is currently no an easy way to test JPA entities and services link in Spring. To test classes using CDI you as use the approach described in link where we are using the CDI Test Core dependency. For using Mockito we have to add the Mockito Core and the Mockito Extension for Junit to our dependencies. For an sample using CDI-Test see link.

Test Test Implementation look likes

```
package com.example.hellojakarta;

import de.hilling.junit.cdi.CdiTestJunitExtension;
import jakarta.inject.Inject;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.mockito.Mockito.*;

@ExtendWith(CdiTestJunitExtension.class)
@ExtendWith(MockitoExtension.class)
class GreetingResourceTest {
    @Inject
    private GreetingResource greetingResource;

    @Mock
    private GreetingService greetingService;

    @Test
    void getAll() {
        var greeting = new Greeting("ru", "добро пожаловать");
        greetingResource.addGreeting(greeting);
        verify(greetingService).addGreeting(greeting);
        verifyNoMoreInteractions(greetingService);
    }
}
```

One one side we cannot use the JAX-RS Response type, because to use this one we have to include the Jakarta EE into the testImplementation dependency. But this gives a conflict we the CDI implementation by the cdi-test modules. After removing Response as return value we run into other dependencies problems.

↑ Jakarta EE Tutorial

For that sample we complete the Greeting Services and Resource

---

Greeting Service (DB Access)

```
@Stateless
public class GreetingService {
    @PersistenceContext(name = "sample")
    private EntityManager entityManager;

    public List<Greeting> getAll() {
        return entityManager.createQuery("select g from Greeting g",
Greeting.class).getResultList();
    }

    public Greeting addGreeting(Greeting greeting) {
        entityManager.persist(greeting);
        entityManager.flush();
        return greeting;
    }

    public boolean deleteGreeting(long id) {
        Greeting greeting = entityManager.find(Greeting.class, id);
        if (greeting != null) {
            entityManager.remove(greeting);
            entityManager.flush();
            return true;
        }
        return false;
    }

    public Greeting find(long id) {
        return entityManager.find(Greeting.class, id);
    }

    public Greeting findByContent(String lang, String text) {
        // Unfortunately getSingleResult throw an exception is no record
is found
        // therefore ew are using getFirst from a streaming list
        return entityManager
                .createQuery("select g from Greeting g where g.lang
= :lang and g.text = :text", Greeting.class)
                .setParameter("lang", lang)
                .setParameter("text", text)
                .getResultStream()
                .findFirst().orElse(null);

    }
}
```

GreetingService.java

Greeting Resource (REST API)

```
@Path("/greeting")
public class GreetingResource {
    public record SearchParam (String lang, String text) {}

    @Inject
    private GreetingService greetingService;

    @Inject
    private Logger logger;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Greeting> getAll() {
        List<Greeting> greetings = greetingService.getAll();
        logger.info(String.format("Got %d greetings", greetings.size()));
        return greetings;
    }

    @GET
    @Path("{id}")
    public Greeting getGreeting(@PathParam("id") long id) {
        return greetingService.find(id);
    }

    @POST
    public Response addGreeting(Greeting greeting) {
        if (greetingService.findByContent(greeting.getLang(),
greeting.getText()) == null) {
            var savedGreeting = greetingService.addGreeting(greeting);
            logger.info(String.format("Added greeting %s",
savedGreeting));
            return Response
                    .status(Status.OK.getStatusCode(), "Greeting added")
                    .entity(savedGreeting)
                    .build();
        } else {
            return Response.status(Status.FOUND.getStatusCode(), "Entry
already exists").build();
        }
    }
```

```
    @DELETE
    @Path("/{id}")
    public Response deleteGreeting(@PathParam("id") long id) {
        logger.info(String.format("Delete greeting with Id %d", id));
        var status = greetingService.deleteGreeting(id) ? Status.OK :
Status.NOT_FOUND;
        return Response.status(status).build();
    }

    @POST
    @Path("/search")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response searchGreeting(SearchParam q) {
        var greeting = greetingService.findByContent(q.lang, q.text);
        logger.info(String.format("Search found %s", greeting));
        return greeting == null ?
            Response.status(Status.NOT_FOUND).build() :
            Response.status(Status.OK).entity(greeting).build();
    }
}
```

GreetingResponse.java