Desenvolvimento de Aplicações Empresariais – 2025-26-1S

Engenharia Informática – 3.º ano – Ramo SI

Worksheet 1

Topics: Jakarta EE first Cup ☺ – Configuring a new Jakarta EE project using a Wildfly Application Server and a PostgreSQL database on top of Docker and Docker Compose.

In the laboratory classes of EAD (DAE) we will develop an enterprise application for academic management. In this initial worksheet, please execute the following steps:

1. Install Docker Desktop: https://www.docker.com/products/docker-desktop

   Optional (for Windows users **only**):
   You can install now or skip this step and install it on the step 3.

2. Download and install the **latest** version of IntelliJ IDEA **Ultimate** edition;

3. If you don't have the latest OpenJDK 17 (Temurin) installed on your machine, you can download it using any of the following options:
   - Directly from IntelliJ IDEA in step 5 (recommended for most users).
   - Using **choco** (for Windows users **only**) in step 4.
   - Following the steps described on the Adoptium webpage.

   In any case, make sure your OS command line is using the correct SDK version, by executing the following command: `$ java -version`

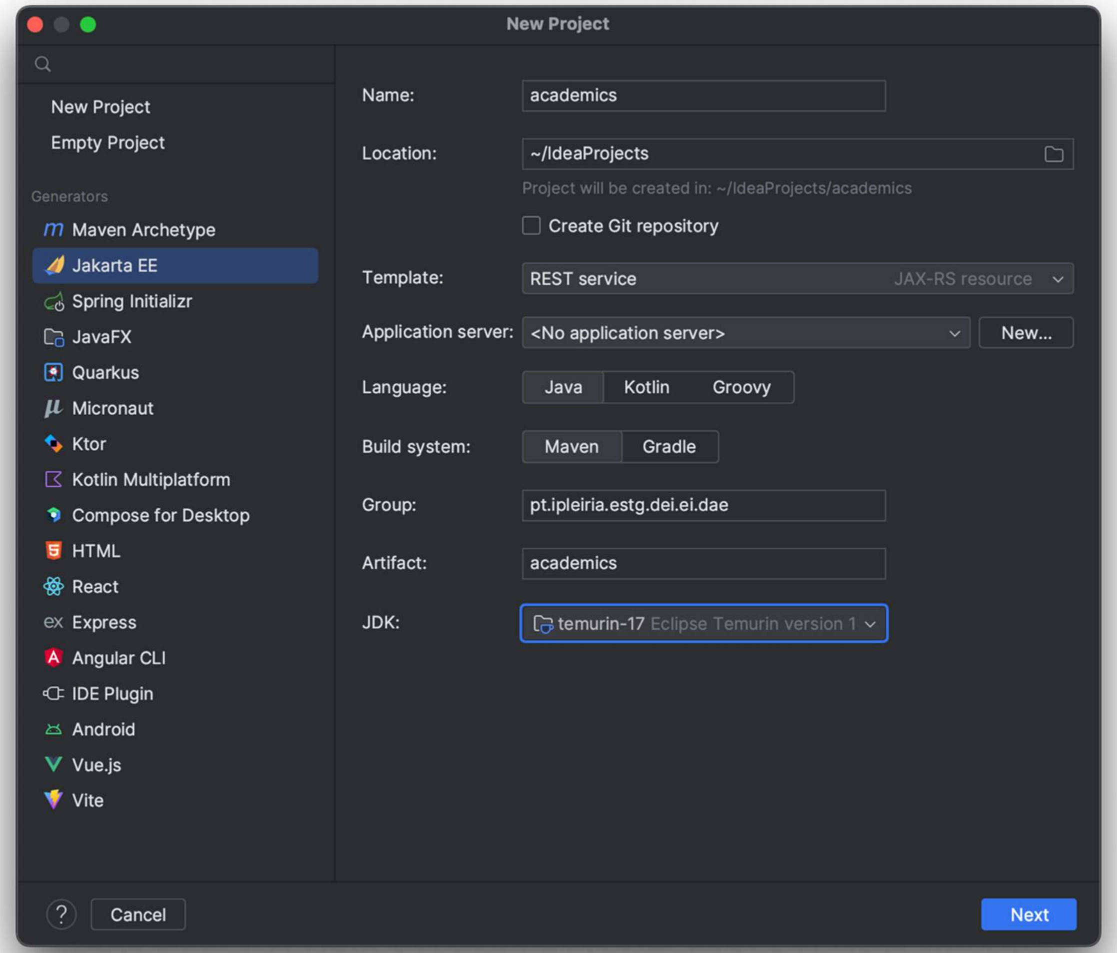
   It should result in something like:
   ```
   openjdk 17.0.12 2024-07-16
   ```

4. Download Apache Maven CLI. We will use this to build the war files.
   - If you are a **Mac** user, execute:
     ```
     $ brew install maven
     ```
   - If you are a **Linux** user (Ubuntu-based):
     ```
     $ sudo apt install maven
     ```
   - If you use **Windows**, with administration privileges:
     - Install Chocolatey: https://chocolatey.org/install
     - Install packages: `$ choco install maven make`

       NOTE: If you are a Windows user **and** skipped previous steps:
       ```
       $ choco install docker-desktop temurin17 maven make
       ```

5. Open the IntelliJ IDEA Ultimate and create a new project:

- Click on “New Project”

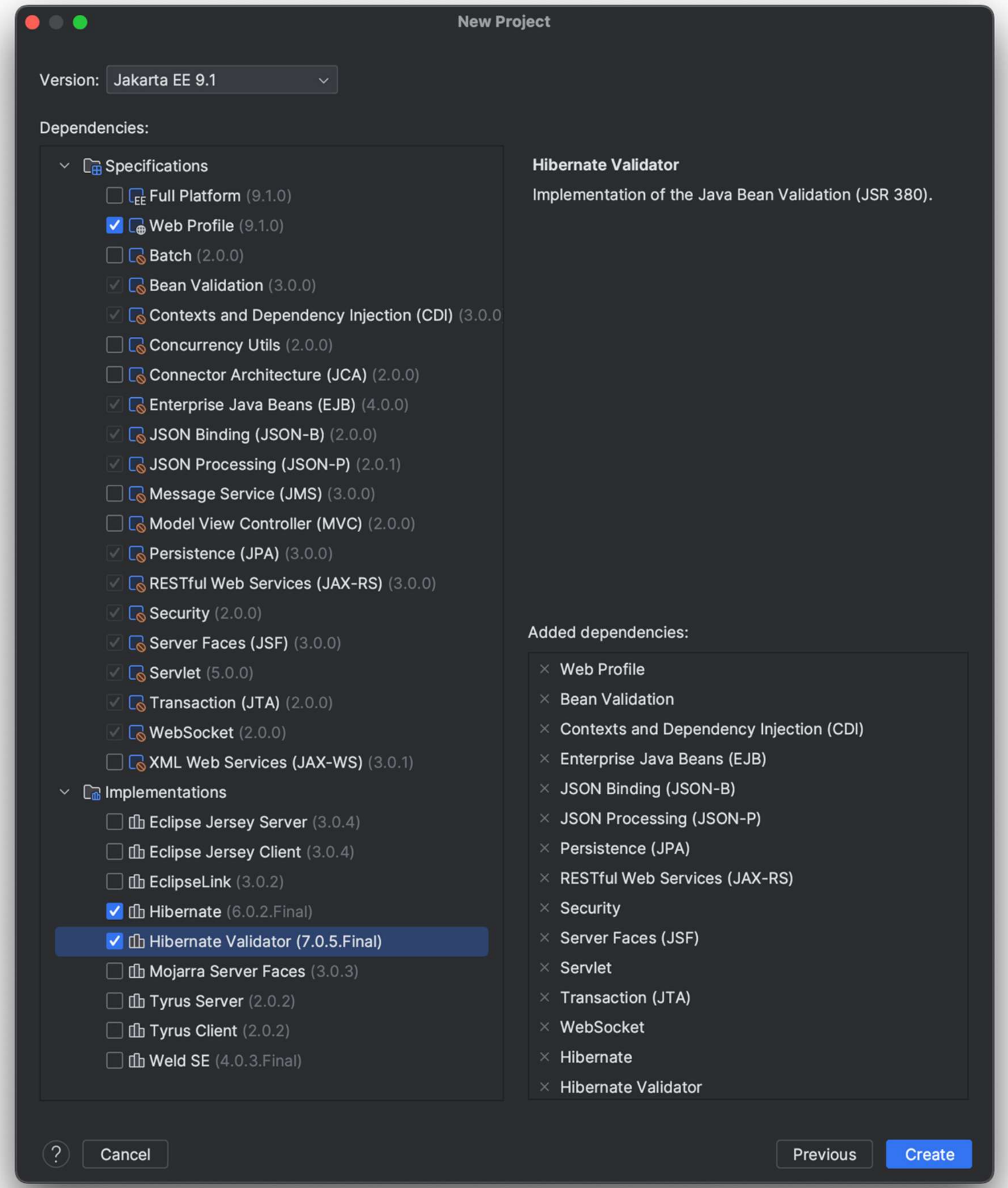


Name: academics
Location: ~/IdeaProjects
Project will be created in: ~/IdeaProjects/academics
Create Git repository
Template: REST service JAX-RS resource
Application server: <No application server> New...
Language: Java Kotlin Groovy
Build system: Maven Gradle
Group: pt.ipleiria.estg.dei.ei.dae
Artifact: academics
JDK: temurin-17 Eclipse Temurin version 1

- Configure the options:
  Name: **academics**
  Location: (Choose the location you prefer)
  Project template: **REST Service**
  Application Server: (Leave empty - we will use Wildfly in a container)
  Language: **Java**
  Build system: **Maven**
  Test framework: **Junit**
  Group: **pt.ipleiria.estg.dei.ei.dae**
  Artifact: **academics**
  Project SDK: **temurin-17**

  And click “Next”.

On the next screen, select "Jakarta EE 9.1" on the Version field, and "Web Profile" from "Dependencies→Specifications" (it should select a bundle of options for you). And then, on "Dependencies→Implementations" choose "Hibernate" and "Hibernate Validator".



Version: Jakarta EE 9.1

Dependencies:

- Specifications
  - [ ] Full Platform (9.1.0)
  - [x] Web Profile (9.1.0)
  - [ ] Batch (2.0.0)
  - [x] Bean Validation (3.0.0)
  - [x] Contexts and Dependency Injection (CDI) (3.0.0
  - [ ] Concurrency Utils (2.0.0)
  - [ ] Connector Architecture (JCA) (2.0.0)
  - [x] Enterprise Java Beans (EJB) (4.0.0)
  - [x] JSON Binding (JSON-B) (2.0.0)
  - [x] JSON Processing (JSON-P) (2.0.1)
  - [ ] Message Service (JMS) (3.0.0)
  - [ ] Model View Controller (MVC) (2.0.0)
  - [x] Persistence (JPA) (3.0.0)
  - [x] RESTful Web Services (JAX-RS) (3.0.0)
  - [x] Security (2.0.0)
  - [x] Server Faces (JSF) (3.0.0)
  - [x] Servlet (5.0.0)
  - [x] Transaction (JTA) (2.0.0)
  - [x] WebSocket (2.0.0)
  - [ ] XML Web Services (JAX-WS) (3.0.1)
- Implementations
  - [ ] Eclipse Jersey Server (3.0.4)
  - [ ] Eclipse Jersey Client (3.0.4)
  - [ ] EclipseLink (3.0.2)
  - [x] Hibernate (6.0.2.Final)
  - [x] Hibernate Validator (7.0.5.Final)
  - [ ] Mojarra Server Faces (3.0.3)
  - [ ] Tyrus Server (2.0.2)
  - [ ] Tyrus Client (2.0.2)
  - [ ] Weld SE (4.0.3.Final)

Hibernate Validator

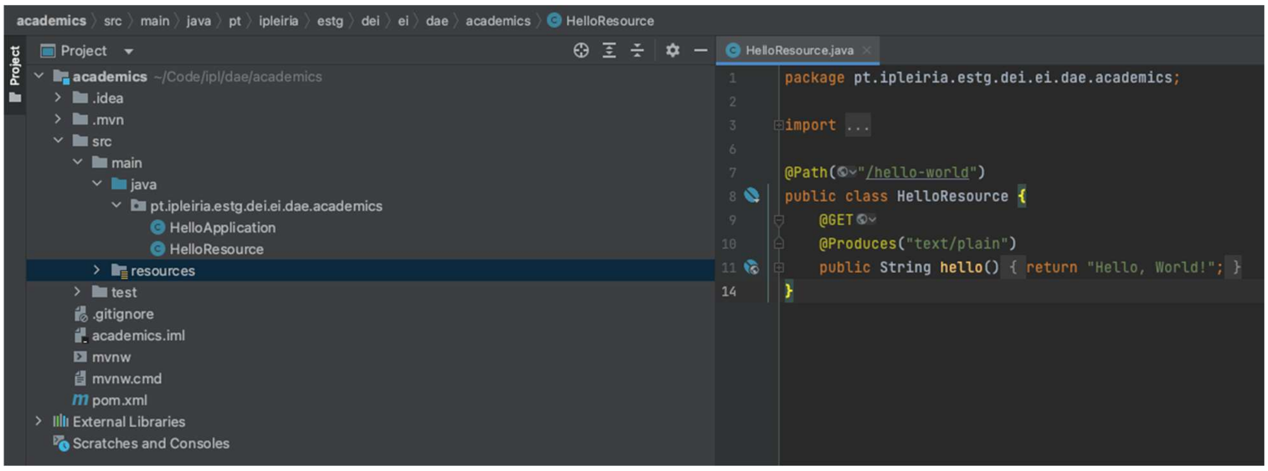
Implementation of the Java Bean Validation (JSR 380).

Added dependencies:

- Web Profile
- Bean Validation
- Contexts and Dependency Injection (CDI)
- Enterprise Java Beans (EJB)
- JSON Binding (JSON-B)
- JSON Processing (JSON-P)
- Persistence (JPA)
- RESTful Web Services (JAX-RS)
- Security
- Server Faces (JSF)
- Servlet
- Transaction (JTA)
- WebSocket
- Hibernate
- Hibernate Validator

- Finally, click "Create".

6. Now you should end up with something like this:


academics 〉 src 〉 main 〉 java 〉 pt 〉 ipleiria 〉 estg 〉 dei 〉 ei 〉 dae 〉 academics 〉 HelloResource
Project
HelloResource.java

```
academics ~/Code/ipl/dae/academics
  .idea
  .mvn
  src
    main
      java
        pt.ipleiria.estg.dei.ei.dae.academics
          HelloApplication
          HelloResource
      resources
    test
  .gitignore
  academics.iml
  mvnw
  mvnw.cmd
  pom.xml
External Libraries
Scratches and Consoles
```

```java
package pt.ipleiria.estg.dei.ei.dae.academics;

import ...

@Path("/hello-world")
public class HelloResource {
    @GET
    @Produces("text/plain")
    public String hello() { return "Hello, World!"; }
}
```

7. Let's first alter some configurations of our project.

Open the pom.xml file and change this section:

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.4.0</version>
</plugin>
```

To include this (`in green`):

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.4.0</version>
    <configuration>
        <outputDirectory>target</outputDirectory>
        <warName>academics</warName>
    </configuration>
</plugin>
```

This will indicate that the war file (Web ARchive file we'll be using to deploy our application) we want to build must be placed inside that directory, and that its name will be "academics.war".

8. Download the file **`docker-wildfly-postgres-smtp.zip`** from Moodle and extract the files `Dockerfile`, `docker-compose.yaml`, `Makefile` and `.env.example` into your project root directory.

9. Yet inside the project root directory, create another file called "`.env`" (**with no extension**) and copy/paste the values from "`.env.example`". The content of the .env file must look like this:

```
APPLICATION_NAME=academics

DATASOURCE_JNDI=java:/AcademicsDS
DATASOURCE_NAME=AcademicsDS

DB_USER=postgres
DB_PASS=dbsecret
DB_HOST=db
DB_PORT=5432
DB_NAME=academics

POSTGRES_DRIVER_VERSION=42.5.4

WILDFLY_ADMIN_PASSWORD=wildsecret
```

Our project uses a Wildfly Web Server container and a PostgreSQL database container, on top of Docker and Docker Compose. Everything is already configured and ready to be integrated with your Java EE project. Your ".env" contains all the configurations you need to set, in order to run the containers and according to your academics application.

These configurations will create a database called "academics", with a default user called "postgres" and a password "dbsecret". The Wildfly admin password is set to be "wildsecret". This will be helpful to access the Wildfly Admin Console later.

Finally, the datasource is configured for your Java project, with a datasource JNDI "java:/AcademicsDS". Your persistence.xml file must reference this datasource JNDI in order to access the database. Finally, you can also define the Hibernate and PostgreSQL driver versions.

10. Return to your project in IntelliJ IDEA Ultimate.

11. Go to "src/main/resources/META-INF" folder and open the "persistence.xml" file.

Remove only the tag <persistence-unit/> *(*not the entire content of persistence.xml)*:

```
<persistence-unit> ... </persistence-unit>
```

And, where the previously deleted tag was, add the following (copy/paste just the code in **bold**):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0">
    <persistence-unit name="AcademicsPersistenceUnit" transaction-type="JTA">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <jta-data-source>java:/AcademicsDS</jta-data-source>
        <properties>
            <property
                name="jakarta.persistence.schema-generation.database.action"
                value="drop-and-create" />
        </properties>
    </persistence-unit>
</persistence>
```

This configures our project to use the PostgreSQL database, already configured in the Wildfly application server with a datasource name "java:/AcademicsDS". The configuration (host, port, db, user, pass, etc.) is already defined in that datasource. We only need to reference the datasource, to use that configuration on our project.

12. Now we are ready to start. You can try to start the containers and check if they are running. Open a terminal at the project root directory (you can use the "terminal" tab on your IDE) and hit the command:

```
$ make up
```

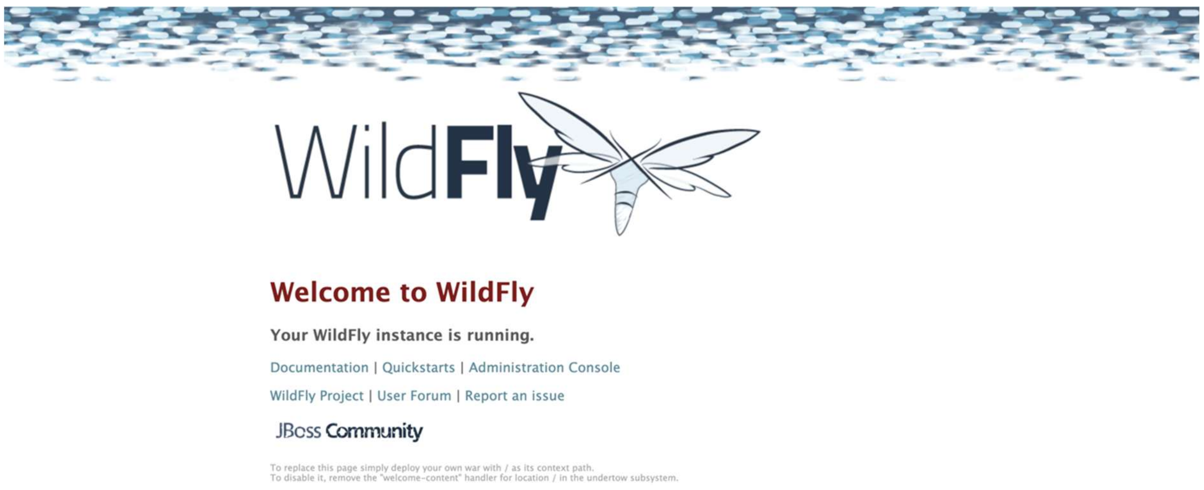
Containers should start running.

You can check their status any time with:

```
$ make ps
```

Access the URL: localhost:8080
(When you start the container, may need to wait more less 1 or 2 minutes in order to the webserver can serve the page)

You should see the following welcome page:



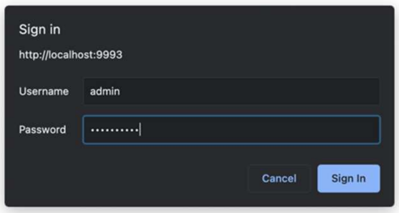
Welcome to WildFly

Your WildFly instance is running.

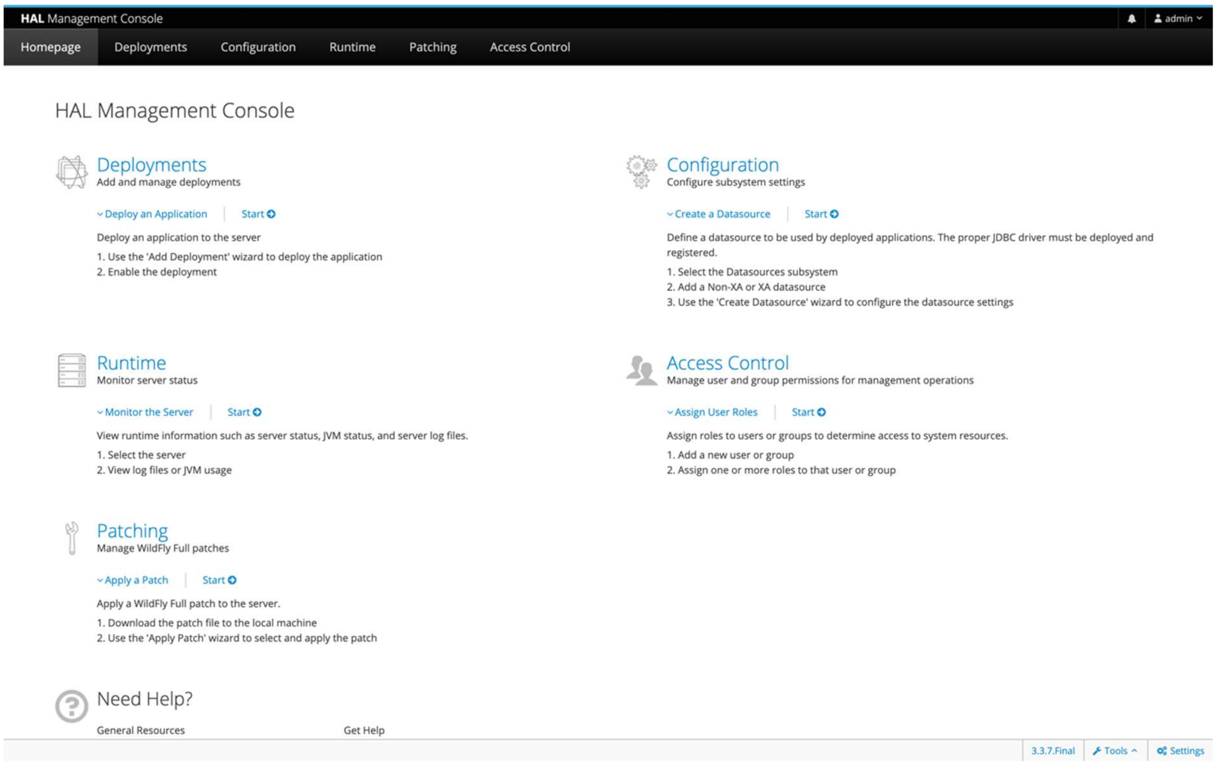
Documentation | Quickstarts | Administration Console
WildFly Project | User Forum | Report an issue

To replace this page simply deploy your own war with / as its context path.
To disable it, remove the "welcome-content" handler for location / in the undertow subsystem.

You can try access the Administration Console by clicking on the link you or just changing the URL to: localhost:9990. Test if the administration is also running and you can access with the credentials you've defined in your .env file.



Sign in
http://localhost:9993
Username: admin
Password: ••••••••••



HAL Management Console

Deployments
Add and manage deployments

Deploy an Application | Start

Deploy an application to the server
1. Use the 'Add Deployment' wizard to deploy the application
2. Enable the deployment

Configuration
Configure subsystem settings

Create a Datasource | Start

Define a datasource to be used by deployed applications. The proper JDBC driver must be deployed and registered.
1. Select the Datasources subsystem
2. Add a Non-XA or XA datasource
3. Use the 'Create Datasource' wizard to configure the datasource settings

Runtime
Monitor server status

Monitor the Server | Start

View runtime information such as server status, JVM status, and server log files.
1. Select the server
2. View log files or JVM usage

Access Control
Manage user and group permissions for management operations

Assign User Roles | Start

Assign roles to users or groups to determine access to system resources.
1. Add a new user or group
2. Assign one or more roles to that user or group

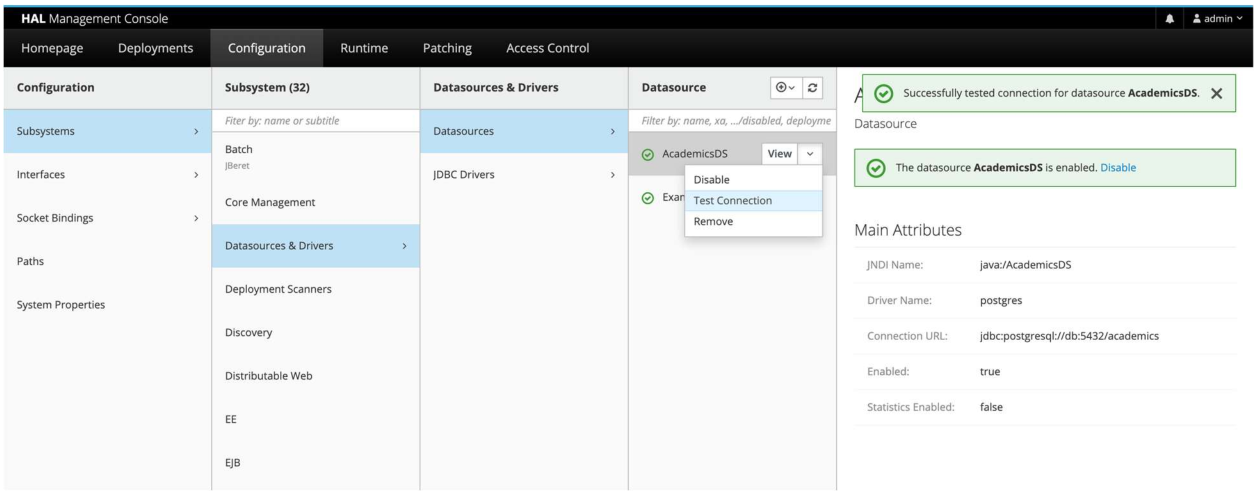
Patching
Manage WildFly Full patches

Apply a Patch | Start

Apply a WildFly Full patch to the server.
1. Download the patch file to the local machine
2. Use the 'Apply Patch' wizard to select and apply the patch

Need Help?
General Resources Get Help

Finally, check if the datasource was created and it's working. Click on the "Configuration" link. Select "Subsystems → Datasources & Drivers → Datasources → AcademicsDS". Unroll the combo box next to "AcademicsDS" and click "Test connection", to check it's working.



Successfully tested connection for datasource **AcademicsDS**.

Datasource

The datasource **AcademicsDS** is enabled. Disable

Main Attributes

JNDI Name:	java:/AcademicsDS
Driver Name:	postgres
Connection URL:	jdbc:postgresql://db:5432/academics
Enabled:	true
Statistics Enabled:	false

13. Great! Now that you have started your Docker containers, and the configurations are OK, you can start developing your enterprise application. We will start by creating a Student and save it on the database, to test that our project can write into the database.

14. First, let's modify the Jakarta EE version of our project, so that it can be compatible with our Wildfly application server. Head to the pom.xml file, and make sure that the Jakarta (Java EE) platform is set to version 9.1.0. If not, please update the version.

```
<dependency>
    <groupId>jakarta.platform</groupId>
    <artifactId>jakarta.jakartaee-web-api</artifactId>
    <version>9.1.0</version>
    <scope>provided</scope>
</dependency>
```

15. Create the package: `pt.ipleiria.estg.dei.ei.dae.academics.ejbs`.

16. Create a singleton Enterprise Java Bean (EJB - Java class) named `ConfigBean` in the `ejbs` package. This EJB should have the following Java EE annotations right above the class declaration of the EJB:

- `@Singleton` // this EJB will have only one instance in the application;
- `@Startup` //this EJB will be automatically instantiated once the application is deployed onto the Wildfly application server.

NOTE: if any of these annotations cannot be recognized by IntelliJ, resolve its Maven dependencies, and then reload your project's Maven (IntelliJ should suggest this to you);

17. Create a method `populateDB()`, which should be called by the server right after it instantiates the `ConfigBean` EJB (annotate the method with the `@PostConstruct` annotation). For now, this method should just print some welcome message in the console (we will be back here later). The full code for this EJB should be similar to this one:

```
package pt.ipleiria.estg.dei.ei.dae.academics.ejbs;

import jakarta.annotation.PostConstruct;
import jakarta.ejb.Singleton;
import jakarta.ejb.Startup;

@Startup
@Singleton
public class ConfigBean {
    @PostConstruct
    public void populateDB() {
        System.out.println("Hello Java EE!");
    }
}
```

18. To deploy your application into the webserver, access to the terminal (either a separated console application or using the integrated one in IntelliJ) and hit the command inside the project root directory:

```
$ make deploy
```

(if you get a "Duplicate mount point" error, just try to execute the `make down` (equivalent to `docker compose down`) and then `make deploy` afterwards)

19. Pay attention to the Wildfly's server log (you'll be looking into it most of the time in this course). Be sure that the application is deployed, and that you can see the "Hello Java EE!" output from the `populateDB()` method; You can check the logs via the command: `docker compose -f logs webserver` or using our shortcut `make` command:

```
$ make logs
```

20. Create the `Student` entity in the `pt.ipleiria.estg.dei.ei.dae.academics.entities` package (use the `@Entity` annotation, above the class declaration). It should have the following attributes: `username` (which is the entity's ID – use the `@Id` annotation above this attribute), `password`, `name` and `email`. The entity must have at least a default empty constructor and a getter and setter method for each attribute. Also write a constructor that receives and sets all attributes' values.

21. Create the stateless Enterprise Java Bean (EJB) (`@Stateless` annotation right above the class declaration) named `StudentBean` in the previously created `ejbs` package and write the method `create(…)` with all student attributes as parameters. This method should create and persist a student in the database. Do not forget to declare and use an `EntityManager` in the EJB (use the `@PersistenceContext` annotation right above the entity manager declaration). The resulting code should be like this:

```
package pt.ipleiria.estg.dei.ei.dae.academics.ejbs;

// . . .

@Stateless
public class StudentBean {

    @PersistenceContext
    private EntityManager entityManager;

    public void create( /* . . . */ ) {
        var student = new Student( /* . . . */ );
        entityManager.persist(student);
    }
}
```

22. Modify the populated method of the `ConfigBean` EJB so that it creates students and persists them in the database. You need to call the `create(…)` method of the `StudentBean` EJB. To do so, you need to declare a `StudentBean` variable in the `ConfigBean` EJB and annotate it with the `@EJB` annotation. Remember that the `populatedDB()` method is called by the server right after it instantiates the `ConfigBean` EJB (due to the `@PostConstruct` annotation). The resulting code should be like this:

```
// . . .
public class ConfigBean {

    @EJB
    private StudentBean studentBean;

    @PostConstruct
    public void populateDB() {
        // . . .
        studentBean.create( /* . . . */ );
    }
}
```

23. Run the application (make deploy) and verify the server logs for any issue (make logs);

24. In order to see the generated data table, do the following:

```
$ make sql
```

Enter the password you've defined in your .env file.
Verify that there is a student at the student table, by hitting:

```
<db_name>=# SELECT * FROM student;
```

--- Optional: Using a GUI to view the database academics---

Notice that you can also see the database contents by configuring your datasource in IntelliJ:

- On the right side of the main editor, click on the "Database" tab to expand it;
- Click on + -> Data Source -> PostgreSQL;
- On the dialog box that opens:
  - Name: `academics`; # or other if you changed your .env file
  - User: `postgres`; # or other if you changed your .env file
  - Password: `dbsecret`; # or other if you changed your .env file
  - Database: `academics`;
  - URL: `jdbc:postgresql://localhost:5432/academics`;
  - Leave the other fields as they are;
  - Click "Test Connection";
  - Click "Ok".
- Expand the `academics` datasource until you see the student table icon (if you cannot see it, click on the refresh button in the window toolbar).
- Double click on the student table to show its contents.
- Click the "Download missing drivers" link if it is available in this form.