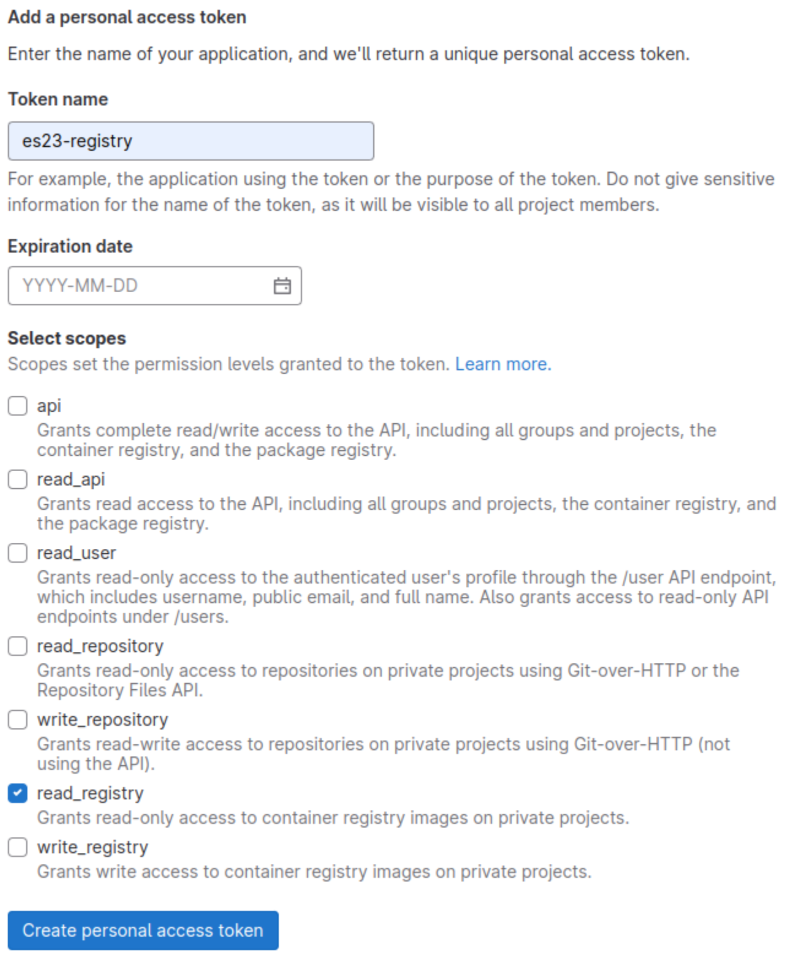
Using the Docker container for development

If you use the Docker container available, you will be able to run Quizzes Tutor quite easily.

1. Pre-requisites:

1. You will need Docker installed in your computer. Docker is free and available from: https://www.docker.com
2. You will need a Gitlab Access Token. You can create a Personal Access Token by going to Preferences > Access Tokens. Direct link: https://gitlab.rnl.tecnico.ulisboa.pt/-/profile/personal_access_tokens

   The access token only needs the scope `read_registry` (see image below). Click “Create personal token” and save the token.



Add a personal access token

Enter the name of your application, and we'll return a unique personal access token.

Token name

es23-registry

For example, the application using the token or the purpose of the token. Do not give sensitive information for the name of the token, as it will be visible to all project members.

Expiration date

YYYY-MM-DD

Select scopes

Scopes set the permission levels granted to the token. Learn more.

- [ ] api
  Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.
- [ ] read_api
  Grants read access to the API, including all groups and projects, the container registry, and the package registry.
- [ ] read_user
  Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.
- [ ] read_repository
  Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- [ ] write_repository
  Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).
- [✓] read_registry
  Grants read-only access to container registry images on private projects.
- [ ] write_registry
  Grants write access to container registry images on private projects.

2. Pulling the Docker image and creating a container

The following command should be executed in a terminal window (you can use the IDE terminal, for example).

1. Login to RNL's registry:

   ```
   docker login https://registry.rnl.tecnico.ulisboa.pt
   ```

   The Username is your IST ID (istxxxxxx) and the password is your Access Token. If successful, you should see "Login Succeeded"

2. Pull the docker image:

   ```
       docker pull
   registry.rnl.tecnico.ulisboa.pt/es/quizzes-tutor
   ```

3. Create/run a container. The following command creates/runs one called `qtutor`, and it exposes a port for the backend (`8080`), a port for the frontend (`8081`), and a port for remote debugging (`5005`). It also mounts the local folder `/home/user/quizzes-tutor` in the container as `/quizzes-tutor`

   ```
       docker run -it -p 8080:8080 -p 8081:8081 -p 5005:5005 -v
   /home/user/quizzes-tutor:/quizzes-tutor --name qtutor
   registry.rnl.tecnico.ulisboa.pt/es/quizzes-tutor:latest bash
   ```

   After running this command, you should see something similar to the following:

   ```
   root@e952cfacf33c:/#
   ```

4. You are now in the container and you can, for example, run the backend. Let's first start the database server:

   ```
   # /etc/init.d/postgresql restart
   ```

   Then, configure the required environment variables:

   ```
   # export POSTGRES_DB=tutordb
     export POSTGRES_USER=postgres
     export POSTGRES_PASSWORD=postgres
     export POSTGRES_HOST_AUTH_METHOD=trust
     export PSQL_INT_TEST_DB_USERNAME=postgres
     export PSQL_INT_TEST_DB_PASSWORD=postgres
     export cypress_psql_db_name=tutordb
   ```

```
export cypress_psql_db_username=postgres
export cypress_psql_db_password=postgres
export cypress_psql_db_host=localhost
export cypress_psql_db_port=5432
```

And now let's run the backend:

```
    # cd /quizzes-tutor/backend; mvn -Ptest-int
spring-boot:run
```

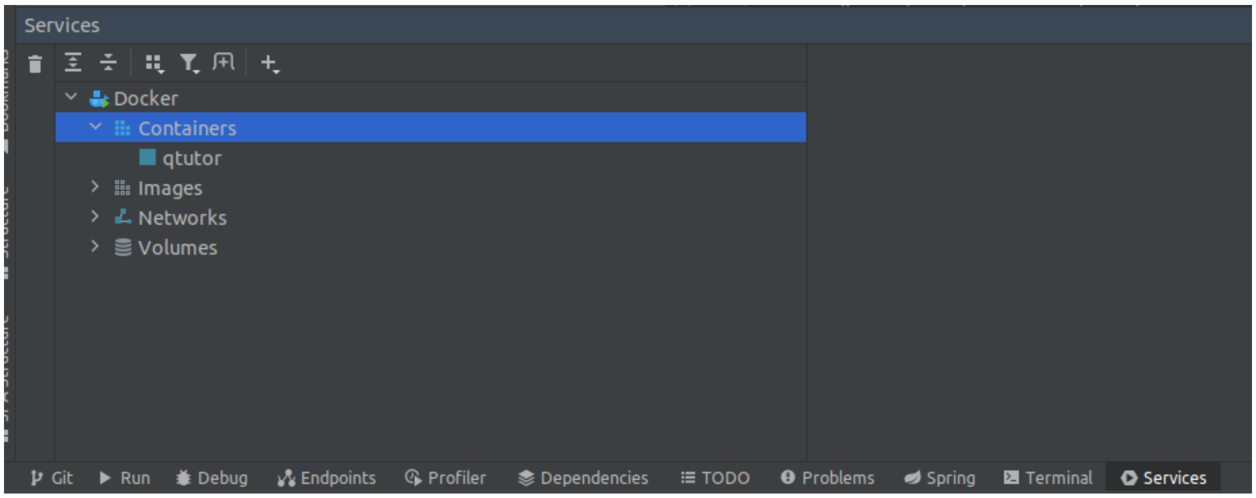
If you visit http://localhost:8080/swagger-ui/index.html , you'll see that the backend is up.

To exit the container (in the terminal), you can execute `exit` or you use CTR+D.

3. Using the Docker container in IntelliJ

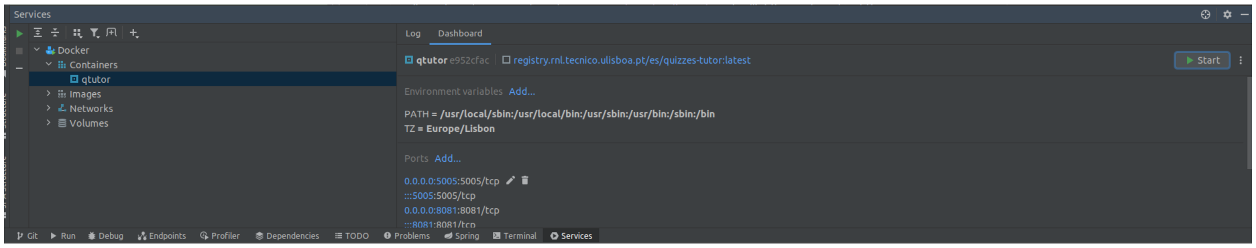
Once you have the container created, you can also manage the container from IntelliJ. This tutorial assumes that you are using the Ultimate Edition, which is free for students.

1. First, go to the tab Services and confirm that you have a container with the same name as before (in this case, `qtutor`).



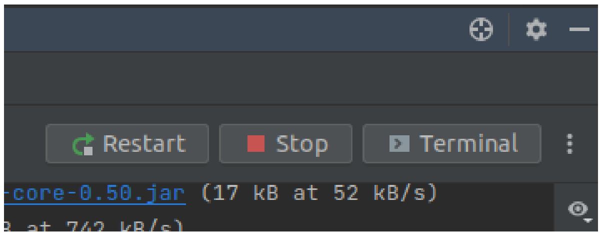
If you do not have the Docker service configured, you can create it by clicking "Add Service" → "Docker Connection" → "OK".

2. If you click the container and select the tab Dashboard, you should be able to start/stop it.



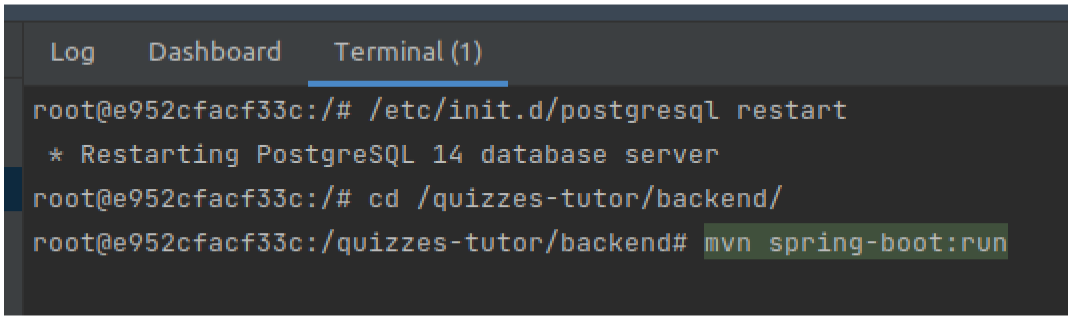
```
qtutor e952cfac    registry.rnl.tecnico.ulisboa.pt/es/quizzes-tutor:latest

Environment variables  Add...
PATH = /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TZ = Europe/Lisbon

Ports  Add...
0.0.0.0:5005:5005/tcp
:::5005:5005/tcp
0.0.0.0:8081:8081/tcp
:::8081:8081/tcp
```

3. Once it is started, you can open a terminal by clicking “Terminal”:



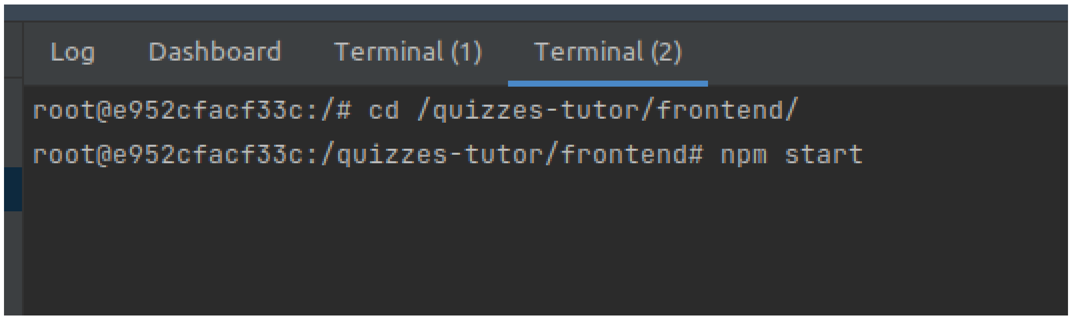
```
-core-0.50.jar (17 kB at 52 kB/s)
```

4. In the terminal, start the backend:



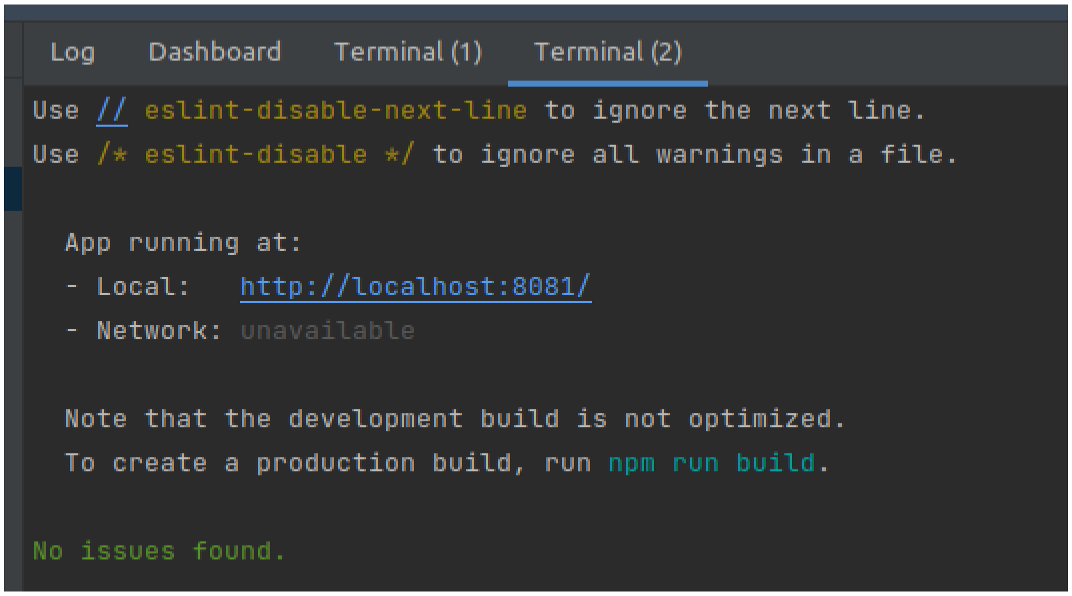
```
root@e952cfacf33c:/# /etc/init.d/postgresql restart
 * Restarting PostgreSQL 14 database server
root@e952cfacf33c:/# cd /quizzes-tutor/backend/
root@e952cfacf33c:/quizzes-tutor/backend# mvn spring-boot:run
```

5. Open a new terminal for the frontend by clicking “Terminal” again (in the Dashboard). In the new terminal, run the frontend:



```
root@e952cfacf33c:/# cd /quizzes-tutor/frontend/
root@e952cfacf33c:/quizzes-tutor/frontend# npm start
```

6. The frontend will be running when you see the following:

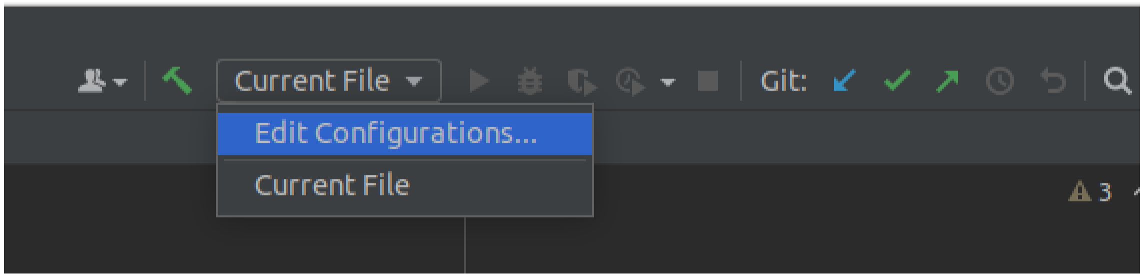


```
Use // eslint-disable-next-line to ignore the next line.
Use /* eslint-disable */ to ignore all warnings in a file.


  App running at:
  - Local:   http://localhost:8081/
  - Network: unavailable

  Note that the development build is not optimized.
  To create a production build, run npm run build.

No issues found.
```

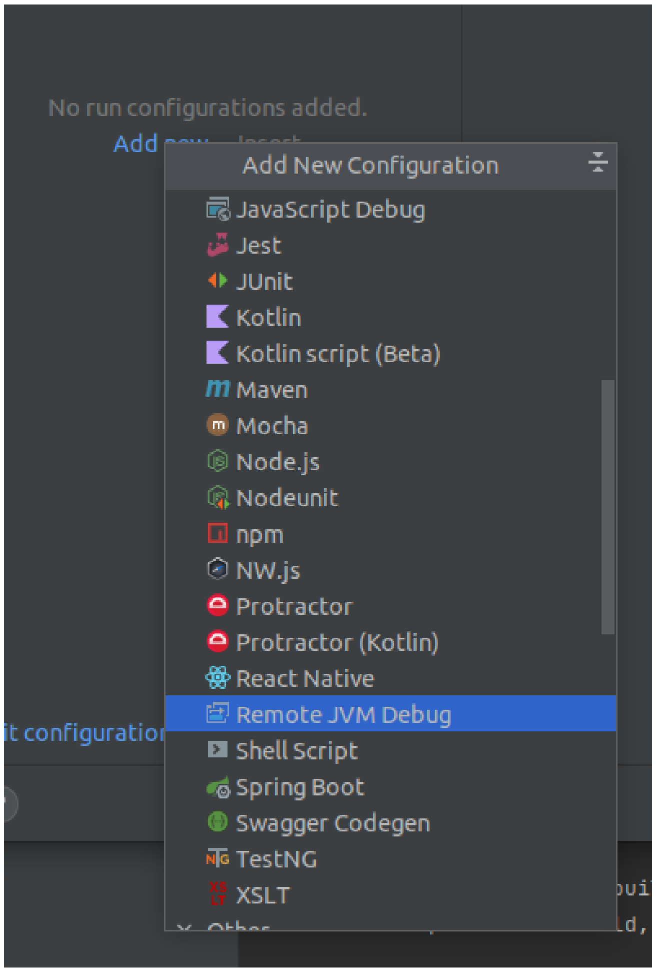
7. If you visit http://localhost:8081, you should see your local instance of Quizzes Tutor!

Debugging your code

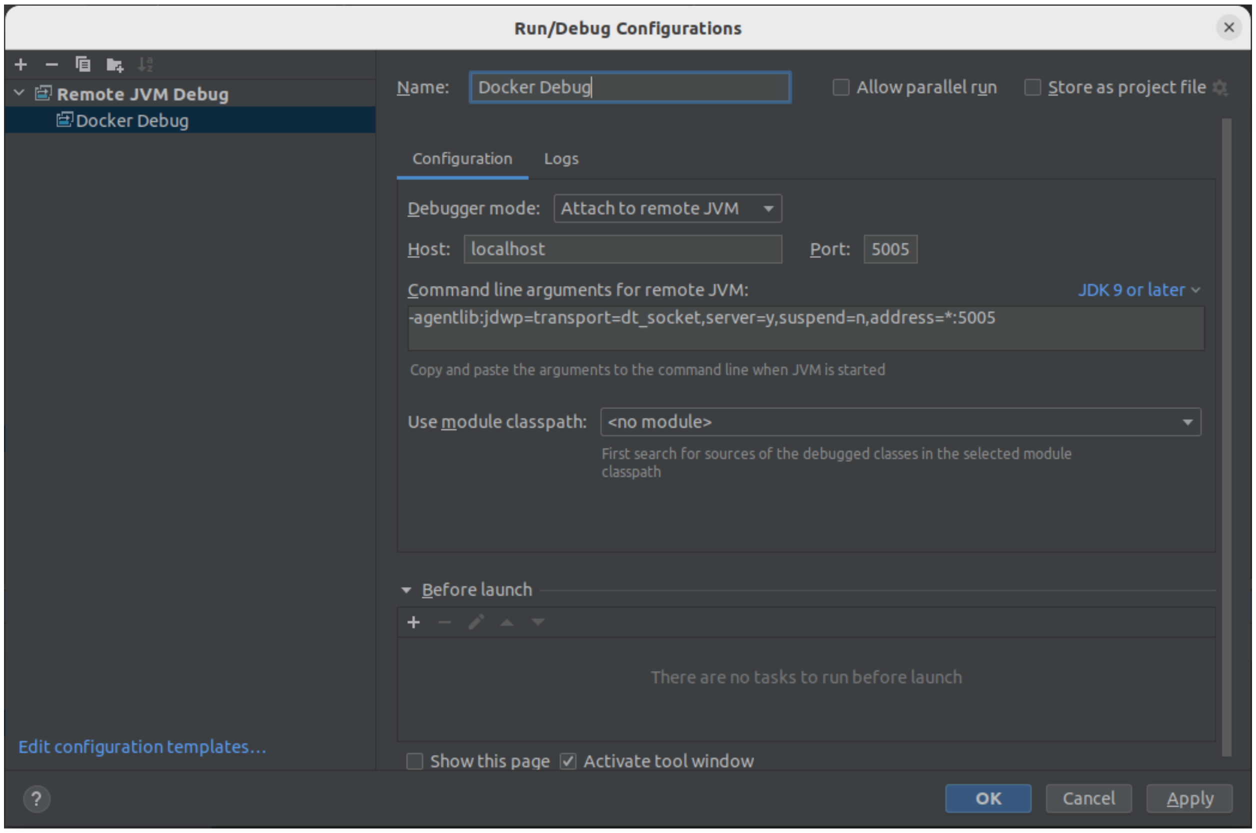
1. To debug your code using IntelliJ and the application running in the Docker container, we first need to create a new configuration. You can create one by clicking on "Edit Configurations…" (located in the top bar):



2. Create a new Remove JVM Debug configuration:



3. Name it something like “Docker Debug” and click OK:



Name: Docker Debug

Allow parallel run ☐ Store as project file ☐

Debugger mode: Attach to remote JVM

Host: localhost Port: 5005

Command line arguments for remote JVM: JDK 9 or later

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005
```

Copy and paste the arguments to the command line when JVM is started

Use module classpath: <no module>

First search for sources of the debugged classes in the selected module classpath

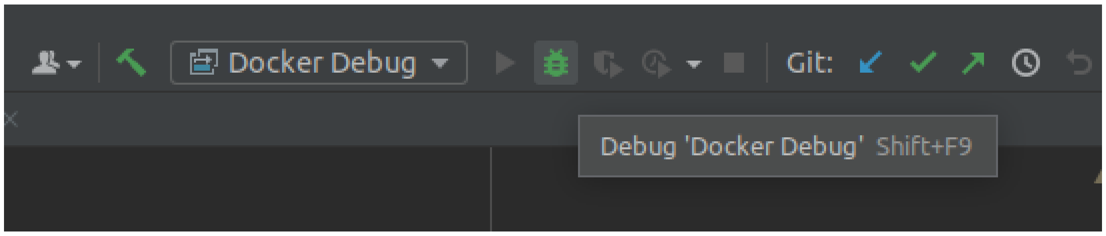
☐ Show this page ✓ Activate tool window

4. In the Docker terminal where you started your backend, stop it and restart it with the following command:

```
    mvn spring-boot:run
-Dspring-boot.run.jvmArguments="-agentlib:jdwp=transport=dt_socket,se
rver=y,suspend=n,address=*:5005"
```

This command is the same as before, but with additional options that will enable remote debugging.

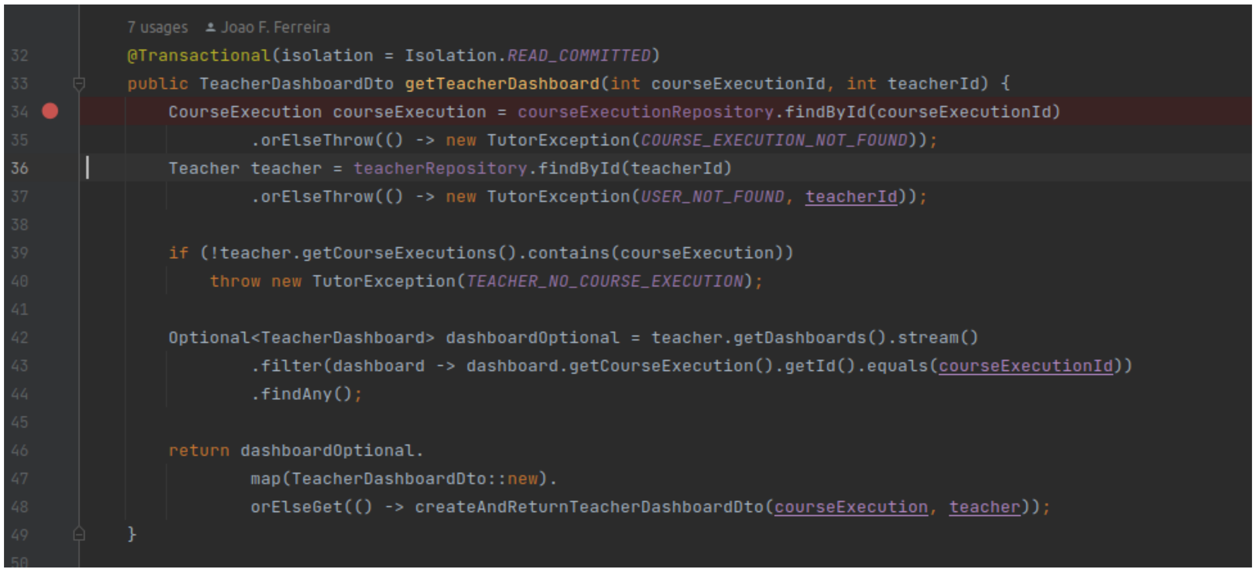
5. Once the backend is running, let's debug using the new "Docker Debug" configuration by clicking the "green bug" button:



If this was successful, you should see the following message:

```
Connected to the target VM, address: 'localhost:5005', transport:
'socket'
```

6. We can test remote debugging by creating a new debug breakpoint. Let's create one in the TeacherDashboard service (TeacherDashboardService.java). Here's an example of a debug breakpoint in line 34:



```
   7 usages  Joao F. Ferreira
32 @Transactional(isolation = Isolation.READ_COMMITTED)
33 public TeacherDashboardDto getTeacherDashboard(int courseExecutionId, int teacherId) {
34     CourseExecution courseExecution = courseExecutionRepository.findById(courseExecutionId)
35             .orElseThrow(() -> new TutorException(COURSE_EXECUTION_NOT_FOUND));
36     Teacher teacher = teacherRepository.findById(teacherId)
37             .orElseThrow(() -> new TutorException(USER_NOT_FOUND, teacherId));
38
39     if (!teacher.getCourseExecutions().contains(courseExecution))
40         throw new TutorException(TEACHER_NO_COURSE_EXECUTION);
41
42     Optional<TeacherDashboard> dashboardOptional = teacher.getDashboards().stream()
43             .filter(dashboard -> dashboard.getCourseExecution().getId().equals(courseExecutionId))
44             .findAny();
45
46     return dashboardOptional.
47             map(TeacherDashboardDto::new).
48             orElseGet(() -> createAndReturnTeacherDashboardDto(courseExecution, teacher));
49 }
50
```

Now, let's use the frontend (http://localhost:8081) and visit the Demo Teacher Dashboard (Demo As Teacher > Dashboard). The application should pause in the browser and you should be redirected to the IDE, where you will see that the breakpoint was reached: