Project Assignment 2

This project is about setting up a CI/CD pipeline using Docker and Jenkins.

We already have a backend (at least partial), a database, and a frontend from the first project. But now it's time to Dockerize everything, so that it's all platform-independent and more easily distributable and scalable, and to set up a system that automatically picks up code changes, tests the new code, rebuilds the application, and deploys the new build on a web server, ready to be used.

Docker containers

Everything in this project will be happening inside Docker containers.

We'll have a group of four 4 containers: `backend`, `database`, `frontend`, and `jenkins`.

You need to create a `docker-compose.yml` file and define those four containers. The database and the frontend are going to be exactly the same as before, so feel free to use the definitions from the first Project Assignment. You can also improve the `frontend` container and save development time by changing the `node_modules` mount from `/project-root/node_modules` to `nodemodules:/project-root/node_modules`.

All containers can have a restart policy of `unless-stopped`.

backend container

image

The starting image for the `backend` container will be `tomcat`, tag `11.0.0-M1-jdk17` (because we want the exact `11.0.0-M1` version of Tomcat and exactly `jdk17` included in it, otherwise you might have compatibility issues).

port mapping

This container will have a Tomcat server on port 8080 (inside the container), but you need to expose that server to the outside, i.e. to your host computer, by mapping it to another port. The port we want to use for accessing the backend from `localhost` is 8099, just like before. So you need to map host port 8099 to container port 8080. (*Note*: if you're wondering why the `frontend` container can't access the `backend` container without any port mappings, the answer is *it can*, but that requires that we change some ports inside those containers. For the sake of simplicity, we'll expose the backend server to the host.)

volumes

You will mount two volumes on this container.

The first one is the host directory `./webapps` to the container directory `/usr/local/tomcat/webapps`. That is where Tomcat keeps the apps it serves. You're going to package your backend app in a `war` file and put that file in this location, so that it's served by Tomcat. That's explained below in Note about deploying apps in a

standalone Tomcat server.

The second mount is a named volume: `m2repo:/root/.m2/repository`. This will create a volume inside Docker called `m2repo` and it will mount it on the container path `/root/.m2/repository`. That's where all Maven dependencies will be stored.

jenkins container

image

The starting image for the `jenkins` container will be `jenkins/jenkins`, tag `jdk17` (we want the latest version of Jenkins and exactly `jdk17` included in it, otherwise you might have compatibility issues).

port mapping

Jenkins uses two ports (inside the container!) for functioning properly: 8080 and 50000. However, it is advisable to map 8080 to another port on the host, because 8080 is likely to be taken. For example, you could map host port 8050 to container port 8080, or you could use any other free port that you want, as long as you map it to container port 8080. As for the other port, it can be mapped to 50000.

volumes

You will mount two volumes on this container.

The first is one a named volume: `jenkins_home:/var/jenkins_home`. This will create a volume inside Docker called `jenkins_home` and it will mount it on the container path `/var/jenkins_home`. This is the Jenkins home folder, where most config files are stored, as well as all project files and workspaces.

The second one should mount the host directory `./webapps` to the container directory `/artifacts`. This is going to be the same `webapps` directory that's mounted on the `backend` container. This volume is going to be shared by `backend` and `jenkins`, so anything you put here will also be present in the other container. You're going to use this volume for moving the `war` artifacts produced by a Jenkins job to the Tomcat server. *This is the crux of this project*.

Named volumes

At the end of your `docker-compose`, you should add the following code at the root level, as a root-level key (not as part of `servies`!). That will make sure that three named volumes will be created and managed by Docker. They are used in your containers, referenced by name.

```
volumes:
  m2repo:
    name: m2repo
  nodemodules:
    name: nodemodules
  jenkins_home:
    name: jenkins_home
```

Java Spring Boot project

You are going to use the same backend project that you've created in the first Project Assignment. Don't worry if it's not done or if not everything is implemented. The point of this exercise is not to have a perfect backend. Just make sure that you have a functioning Spring Boot project with the right dependencies and exposing at least one of the endpoints used by the frontend.

Place your backend project (the source files, the pom, etc.) in the `backend` directory of the repository's root folder.

However, instead of using your own computer to build and serve the backend, you will use the `jenkins` container to build it, and the `backend` container to serve it. So far we've been using an embedded Tomcat server to start the application. But for this project, we will serve it in a standalone Tomcat server, running in a Docker container.

You will package the application as a `war` archive (a file with the extension `.war`) and put that artifact in a certain location inside the `backend` container. That will be enough for the Tomcat server installed on that container to unpackage and run your application. This is called "*deploying*" the app.

You'll have to make some changes in order to make your Spring Boot app servable by a standalone Tomcat server. The following is what you need to change to make that happen.

pom.xml

- make sure java version is 17 in the `pom`
- add packaging `war` to pom (an `xml` element `<packaging>` with the value `war` inside it, placed within the root-level `<project>` element)
- add dependency `org.springframework.boot.spring-boot-starter-tomcat` with scope `provided`

- inside `build.plugins` (in the `plugins` element, within the `build` element), add plugin `org.apache.maven.plugins.maven-war-plugin` (version 3.3.1 will work fine)
- you will learn below that the name of `war` artifact needs to be `ROOT.war` before it's deployed; if you want the name of the `war` artifact to be automatically generated as `ROOT.war` so that you don't have to rename it each time you deploy it, you can add `<finalName>ROOT</finalName>` inside the `build` element

Main class

Your main class (it doesn't matter what it's called; it's the class annotated with `@SpringBootApplication`, the class where you have your `main` method, which calls `SpringApplication.run()`) should extend `SpringBootServletInitializer`. That's the only change you need to make to your main class.

Spring config

(e.g. `application.yml`)

- update the `datasource url` to fit the new URL of the database (it's going to be inside a container with the name `database`; the backend and database containers are going to be in a network, which means they can reach each other by using their names as the host name)
- you can completely remove all `server` configurations here (like `port` and `servlet.contextPath`) because they're not going to matter; they are only used in the embedded Tomcat server and they are ignored by the standalone Tomcat server in the `backend` container
    - see below about `contextPath`; the context path is set to the name of the `war` artifact, and anything in `application.yml` is ignored
    - the server port is also ignored here; the port through which the app is accessed is the backend Docker container's port

Note about deploying apps in a standalone Tomcat server

The root of the server is the value of the environment variable `CATALINA_HOME`. By default it's `/usr/local/tomcat`. You can see that by inspecting the `backend` container and looking at the environment variables.

The location where `war` artifacts are kept in a Tomcat server is `CATALINA_HOME/webapps/`. Whenever a new `war` file is placed there, the server refreshes itself and deploys the new app. It reads the `war` and creates a new directory in the same directory, with the same name as the `war` file (except for the `.war` extension, of course).

Context path

Context path is the common path of all your endpoints that directly follows the host name and port. For example, if your endpoint is `http://localhost:8099/api/v1/products`, the *context path* is `/api/v1`, and the endpoint path is `/products`.

The context path of the new app will correspond to the name of the new directory (i.e. the name of the `war` file). That means that, for example, if the server is running on port 8080, and your artifact is named `my-app.war`, that will create a new directory `CATALINA_HOME/webapps/my-app/` and your app will be available at `http://localhost:8080/my-app`.

One exception is if the app is in a folder named `ROOT`. In that case the context path will be `/`, i.e `http://localhost:8080/` (no path). This is what we want for our app!

If you name the `war` artifact `ROOT.war`, the context path will be `/`, and your endpoints will be available at `http://localhost:8099/ENDPOINT_PATH`. Then it's your responsibility to make sure that all your endpoints start with `/api/v1`.

Any context path defined in your Spring boot project from before will now be ignored. That means that if you were using `server.servlet.contextPath` with the value of `/api/v1` before, you're going to have to add the `/api/v1` path to your endpoints another way. The easiest and most straightforward way is to annotate your Controller classes with `@RequestMapping("/api/v1")`.

- If you already had an annotation like `@RequestMapping("/products")`, you can prepend the `/products` path with `/api/v1` so that it becomes `/api/v1/products` (the same goes for `carts`, of course!)
- If you already had an annotation like `@RequestMapping("/api/v1/products")`, then you're already set (the same goes for `carts`, of course!)

Note about packaging the backend

Maven can package your app into a `war` artifact (an archive file with the extension `.war`). The Maven command to package the app is `mvn package`. If your `pom` declares that `packaging` should be `war`, then Maven will create a `war` archive. Otherwise it will create a `jar` by default, which is not what you want.

Another Maven command to package the app as a `war` archive is to use a specific Maven goal instead of a Maven phase. That goal is `war:war`, which means the whole command is `mvn war:war`.

You can run these commands on your computer (if you have Maven installed), but ultimately, the goal is to make Jenkins run these commands inside a container.

Important note

The state of your backend is not important for this assignment. Even if you haven't implemented all endpoints and they're not functioning correctly, that's fine as long as you manage to deploy the app and it's working with at least one endpoint.

In other words, you don't need to have a perfect Project Assignment 1 in order to complete Project Assignment 2!

Jenkins

Your `jenkins` container will be running an instance of Jenkins. First, go ahead and create a user and install the plugins that you want/need.

Then you need to create a pipeline project (name it whatever you want), with the following settings.

Build Triggers

You can set them up any way you want, but one is required: Poll SCM. Set the schedule to `H/10 * * * *` in order to poll your SCM (i.e. GitHub) for changes every 10 minutes. If you want, you can change that to any other interval you want, if you feel like learning `cron` syntax. For example, `H/30 * * * *` will poll every 30 minutes. This is not important.

This way, whenever one of your team members pushes new commits to your repository, Jenkins will trigger a new build within 10 minutes.

Pipeline definition

Set it up to pull the pipeline definition from SCM. You need to enter your GitHub repository URL and add a credential set so that it can pull the pipeline (the `Jenkinsfile`) from your repository.

Note: Since you'll be working in a team, one of your team members can create a GitHub repository and invite the rest as collaborators. That way everyone in the team will have access to the repository.

Put everything in this folder in that repository. This will be the repository's root folder.

Make sure to set up the correct branch in the "Branches to build" section. If your branch name is `master`, then that field needs to be `*/master`. If the branch name is `main`, it should be `*/main`, etc.

And "Script Path" can stay `Jenkinsfile`, because that is the name of the file containing the pipeline script. That means that you'll need to create a file with that name in the root of the repository, and write your declarative pipeline inside it.

The pipeline script

As stated above, the script will be in a file called `Jenkinsfile`, located at the root of the repository.

The pipeline itself may contain as many stages as you want, with as many steps as you want, as long as it gets the application built and deployed.

The first or one of the first steps has to be cloning the repository in the Jenkins project directory. You can do that with the Jenkins command `git`. Here's an example of its usage, which specifies the branch to checkout, a credential set, and the repo's URL:

```
git branch: 'my-branch', credentialsId: 'my-credentials-id', url: 'https://github.com/my-git-hub-user-name/my-repository-name.git'
```

The most important parts are the following.

Maven tools

The `pipeline` object should contain a `tools` object, which declares a `maven` installation to be used. The code block for that looks something like this:

```
    tools {
        // Install the Maven version configured as "MY_MAVEN" and add it to the path.
        maven "MY_MAVEN"
    }
```

This block will try to find a Maven installation setup defined in `Manage Jenkins > Global Tool Configuration > Maven`. The name (in the example above "`MY_MAVEN`") needs to correspond to the name given in the "Maven installations" settings.

Build stage

(At least) One of the stages should be a build stage where you package the application (i.e. you build a `war` artifact). It's your choice whether you lump all Maven phases into one (e.g. by running `mvn package`) or you separate them by goals (e.g. `mvn compile` or `mvn test-compile`, then `mvn surefire:test`, then `mvn war:war`, or first `mvn test` before `mvn war:war`, or any other combination). For extra points, do separate them into several stages, and begin by `mvn clean`.

The simplest step in this stage can be just `mvn package` and nothing else, which goes through all previous phases of the Maven lifecycle first (compile, test-compile, test, and package). It produces a file titled `ROOT.war` in `backend/target/`.

Important note about working directories

When you run any shell script/command in the pipeline, the current working directory is the Jenkins project's root (which is also the repository's root). For example, if you run `sh 'pwd'`, it will print the current working directory, which will print something like `/var/jenkins_home/workspace/YOUR_PROJECT_NAME/` (the project's root).

However, you will want to run the `mvn` commands in the same directory where your `pom.xml` is, which is `./backend` (relative to the project's root). There are two main options to achieve that.

- One is to run `mvn` with the `-f` option (`f` for *file*, as in "location of the `pom` *file*") and the location of the pom file. For example `sh 'mvn clean -f backend'`, because the `pom` is inside the `backend` directory
- Another is to wrap the `sh` command in a `dir` block, which moves the current working directory somewhere else. In the parentheses you pass the directory you want to move to. The commands within that code block will be executed from the passed directory (but note that anything outside it will still be executed from the Jenkins project's root). For example, this will run `mvn clean` inside the `./backend` directory:

```
dir('backend') {
    sh 'pwd' // prints /var/jenkins_home/workspace/YOUR_PROJECT_NAME/backend
    sh 'mvn clean'
}
```

Deploy stage

For deploying the app, you'll simply move the produced `war` artifact to the `backend` container, where Tomcat will pick it up and serve the app.

If you've set up the Docker containers properly and mounted all volumes, the `./webapps` directory (in the repository's root) will be mounted on both the `jenkins` and the `backend` container. In `jenkins`, it will be mounted on the path `/artifacts`. In `backend`, it will be mounted on the path `/usr/local/tomcat/webapps`. That means that whatever is in `jenkins`'s `/artifacts` will also be in `backend`'s `/usr/local/tomcat/webapps`!

As explained above in Note about deploying apps in a standalone Tomcat server, when you place a `war` archive in the `CATALINA_HOME/webapps` directory of a Tomcat server, Tomcat picks it up, unpacks it into a folder with the same name as the archive, and serves it as a web app. You will use that knowledge to easily deploy your newly built app by moving the `war` file that Jenkins produced to Tomcat's `webapps` directory.

This stage will be only about moving that file to the appropriate place. The location of the produced `war` archive (relative to the repository root) is `./backend/target/`, and the place it needs to be moved to is `/artifacts` (absolute path in the `jenkins` container).

You can simply add a shell script step that runs the following command:

`cp backend/target/ROOT.war /artifacts`

(or you can wrap it in `dir('backend')` and run `cp target/root/ROOT.war /artifacts`).

The artifact will show up in `/usr/local/tomcat/webapps` in the `backend` container, due to the way the volume is mounted, which will trigger Tomcat to serve the app, thus successfully *deploying* the app.