Lab 1 Team practices for enterprise Java development

Last update: 2024-09-19. [Use the online version for the latest updates…]

Context and learning objectives

Professional software engineering relies on team-oriented tools and practices to enhance development effectiveness. These tools should promote productivity, collective ownership, code/effort reusing, and deter from the mentality of "it works on my computer…".
This lab addresses the basic practices to set up a development environment that facilitates cooperative development for enterprise Java projects, specifically:

- use a **project build tool** to configure the development project and automatically manage dependencies.
- collaborate in code projects using git for **source code management (SCM)**.
- apply a **container technology** (Docker) to speed up and reuse deployments.

References

- "Apache Maven: A Practical Introduction" [Video course]
- "Pro Git" [Free book]
- Docker introduction [short video]

Lab submission

This is a two-week lab. You may submit as you go but be sure to complete and submitted by the deadline outlined in the class plan (see slides 10 and 11 from the first class)

What to submit:

- most exercises will result in a code project. Create the project under the appropriate subfolder for the submission, e.g.: exercise 1.2 should have a corresponding project in 🗁lab1/lab1_2
- you should create a "study notebook" for each lab, placed in the root folder of that lab, e.g. in 🗁lab1/README.md (use the Markdown format).

  What to include in the notebook? **This should be a notebook that you (or other learner) could use for study;** during the lab activities: take note of key concepts, save some important/useful links, maybe paste some key visuals on the topics being addressed, etc. Think of the notebook as a "notes to self" resource, explaining and summarizing the key concepts and practices of the lab.

How to submit:

- create a personal Git repository for IES based on the template (https://classroom.github.com/a/umxVeBzl). You will use it to maintain your individual classes portfolio[1]. Please name it according to the pattern "**IES_2425_*123456***", replacing the greyed text with your student number.

[1] You are suggested to create a private repository; the faculty will then ask you to share your repository.

— you should prepare a folder for each lab (e.g.: 🗁lab01, 🗁lab02,…) and a subfolder for each relevant section of the lab (e.g.: lab01/**lab1_1**, lab01/**lab1_2**,…).

— you will submit your work by committing into the main line in your repository. You are required to have at least one commit per week. The final commit for each lab should be described with the message "**Lab x completed.**" (replacing *x* with the lab number.)

1.1 Basic setup for Java development

Setting up Maven

a) Be sure that you have Java installed in your environment.

The recommended version is OpenJDK 21 LTS. You may use a more recent version, but the exercises are tested for the stable LTS version. If needed, install Java development environment. In the terminal, test if you have the JDK installed:

```
$ javac -version
```

b) Install Maven in your environment.

Note: the environment variable "**JAVA_HOME**" should be defined in your system, pointing to the root of the JDK installation (not the JRE).

Test the Maven installation:

```
$ mvn --version
Apache Maven 3.9.9
Maven home: /usr/share/maven
Java version: 21.0.4, vendor: Private Build, runtime: /usr/lib/jvm/java-21-
openjdk-amd64
Default locale: en, platform e
```

Expected: v3.9 or later

Expected: OpenJDK 21 (or 17, 11...)

Setting up Git

Make sure that you have git installation available in the command line in your development machine.

```
$ git --version
$ git config --list
```

If you do not get an output from git config including your **name** and **email address**, then

- configure the git installation.

You should also configure your environment to use a ssh key to access git (instead of a password). Exercises will be based on the command line (CLI). The use of graphical Git clients is optional (e.g.: GitHub Desktop, SourceTree, GitKraken). In addition, IDEs have built-in support for git *commands.*

1.2 Build management with the M*aven* tool

The regular "build" of a (large) project takes several steps, such as obtaining dependencies (required external components/libraries), compiling source code, packaging artifacts, updating documentation, installing on the server, etc.
In medium to large projects, these tasks are coordinated by a ***build tool***; the most common build tools for Java projects are Maven and Gradle. Maven is widely used among professional projects.

Getting started with Maven

Java Maven projects can be opened in the main IDEs, configured to use Maven. However, we will start by using Maven in the command line (CLI). The entire ***build cycle*** can be managed from the

command line, which makes Maven a convenient tool across multiple operating systems and in continuous integration scenarios (remote servers, without interactive terminal/GUI).

c) Get started with "Maven in 5 minutes" hands-on.

Note that **some parts are to be adapted for your own case**. In particular, "*groupId*" and "*artifactId*" should be specific for each project, taking into consideration the naming conventions for these properties:

```
$ mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Note: "-D" switch is used to define/pass a property to Maven in CLI.

Pick a popular IDE of your choice and open this Maven project. You should have some support to "import" the folder or pom.xml. Suggested IDEs for IES classes:

➔ IntelliJ IDEA Ultimate, available from student's license. Built-in support for Maven and Spring Boot (later).

➔ VS Code (free). Good support for Maven (with Java Extensions Pack) and great plug-ins ecosystem. Great for using GitHub *Copilot*!

Remember to take notes as you go. Use your "notebook" (i.e., the README file). You may wish, for example, to explain what is a "Maven archetype" or prepare a quick cheat sheet on the naming conventions for *groupId* and *artifactId* (you will need this often…).

Using Maven – weather forecast project

Let's now create a small **Java application to invoke the weather forecast API** available from IPMA.

d) Start by analyzing the structure of the API requests and the replies (e.g.: check the 5-days aggregate forecast for a location

→ "Previsão Meteorológica Diária até 5 dias agregada por Local"). You may use any HTTP client, such as the *browser* or the *curl* utility. For example, to get a 5-day forecast for Aveiro (which has the internal id=1010500):

```
$ curl http://api.ipma.pt/open-data/forecast/meteorology/cities/daily/1010500.json | json_pp
```

Note: json_pp (*json pretty print*) may not be available in your system. In that case, just omit it.

e) Create a **new Maven project** (MyWeatherRadar) for a standard Java application.

Note that you can create the project from the command line, using archetype:generate, or using your favorite IDE. Usually, it will be easier to use the IDE support.

In all your projects, you should define the "groupId" and "artifactId"; the initial "version" is automatically "1.0-SNAPSHOT" by default.

f) Check the content of the POM.xml and the folder structure that was created.

g) Change/add some additional properties in your project, such as the development team, character encoding or the Java version to use by the compiler. E.g.:

```
<properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
</properties>
```

Note for VS Code users:
Some tools will generate a huge POM with very old dependencies!… Note that you just need something like this sample POM!

Declaring project dependencies

Build tools allow to state the project dependencies on external artifacts. Maven will be able to retrieve well-known dependencies from the Maven central repository automatically.
In this project, we will need to open a HTTP connection, create a well-formatted GET request, get the JSON response, process the response content. Instead of programming all these (demanding) steps by hand, we could use a good component/library, or, in Maven terms, declare *dependecies* for *artifacts*.

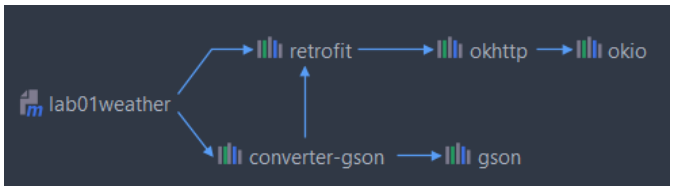
h) We will use two helpful artifacts in this project: Retrofit and Gson. Find out more about these projects…

   Declare both dependencies in your POM (as in step #2, here). In general, you should explicit the version of the artifact that you want to use; stick with a recent version.

   Note the artifact coordinates below; try to locate this artifact by searching the Maven central.

```
<groupId>com.squareup.retrofit2</groupId>
<artifactId>retrofit</artifactId>
```

i) In POM, we declare direct dependencies; these artifacts will usually require other dependencies, forming a graph of project dependencies. For example, Retrofit will require OkHTTP.



j) To jump start the implementation, you can use these base implementations available:

   - suggested starter class (*main*);

   - base implementation for supporting classes (IpmaService, IpmaCityForecast, CityForecast).

Compile and run the project, either from the IDE or the CLI:

```
$ mvn package #get dependencies, compiles the project and creates the jar
$ mvn exec:java -Dexec.mainClass="weather.WeatherStarter" #adapt to match your own
package structure and class name
```

k) Change the implementation to receive the city code as a parameter in the command line and print the forecast information in a more complete and user-friendly way.

   Note that *mvn exec:java* can receive command line arguments (*exec.args*).

Remember to take study notes as you go. How about creating your own "cheat sheet" on most useful/frequent Maven commands?...

1.3 Source code management using Git

Introduction

a) For a ***git* introduction**, take the exercise "Learn Git with Bitbucket Cloud".

   If you are **already familiar with *git*,** you can proceed to the next steps.

   Note: Although the tutorial uses Bitbucket, you may adapt to other platforms, such as GitHub (suggested).

Add a project to Git versioning

b) Let's add the project from the previous exercise (MyWeatherRadar) to your personal git.

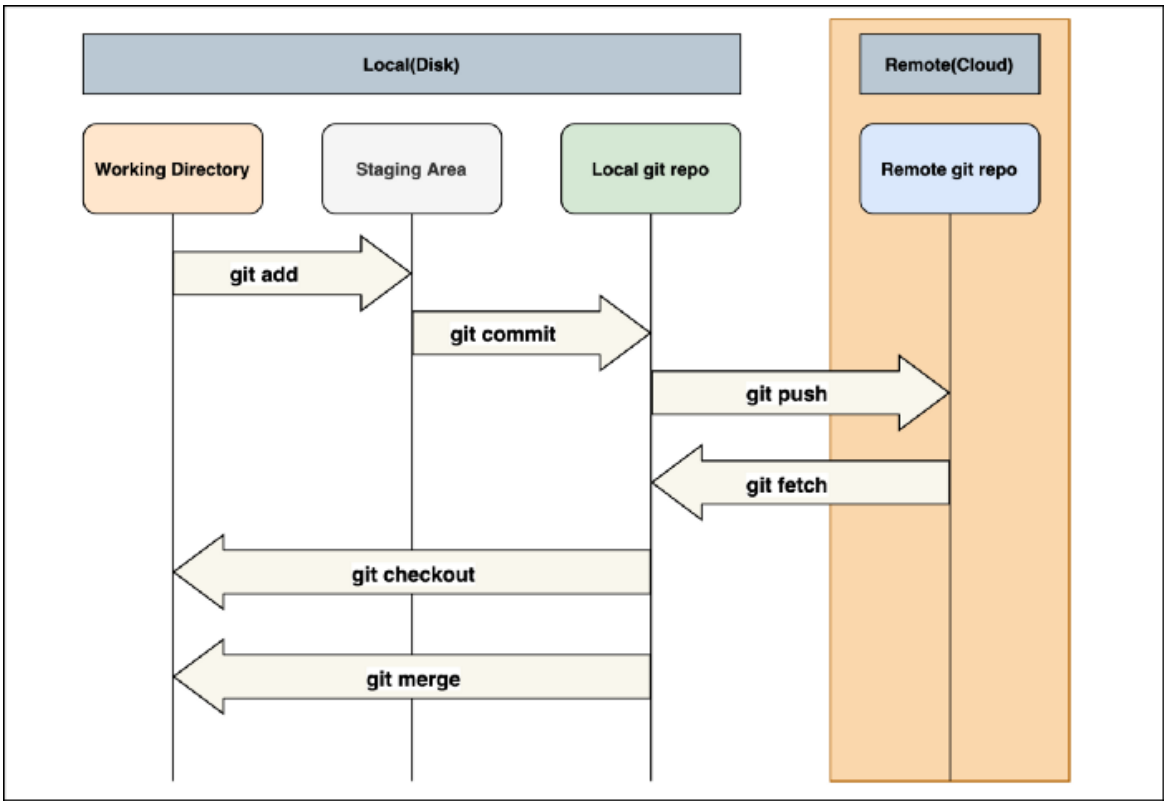
Start by adding information about the exclusions, that is, indicate the files/folders that are only of local interest and should not be passed to the common repository (e.g.: the ./target folder, which has the compiled versions, should not go to the repository).

To do this, be sure to add a ".gitignore" **in the root** of your projects[2]. You can find useful .gitignore templates here or use this suggested .gitignore file.

c) Import your project into the Git version control and synchronize with to the cloud [for GitHub, see section adding-a-project-to-github-without-github-cli]

Illustrative example: you should adapt for your own case.

```
$ cd project_folder  # move to the root of the working folder to be imported
$ git init           # initialize a local git repo in this folder
$ git remote add origin <REMOTE_URL>     #must adapt the url for your repo
$ git add .          # mark all existing changes in this root to be commited
$ git commit -m "Initial project setup for exercise 1_3"          #create the
commit snapshot locally
$ git push -u origin main         #uploads the local commit to the shared repo
```



Your files can "be" in the Working Directory, Staging Area, Local repo or Remote repo; what is the difference/purpose of each one?

[2] Note that .gitignore provides patterns for files or folders to exclude, from the directory in which it is include. You may create .ignore in multiple folders, but is usual to do it in the root of a project. If you have a repository with several project subfolders, you may choose to propagate certain rules from root into the sub-folders, in the exclusion list.

Simple collaboration

d) Let's simulate the existence of **multiple contributors** to the project. For simplicity, you will be the only developer, but working in different places, so you will have "changes" that need to be synchronized into a coherent repo. Let's call the main IES root as "location1" and another auxiliar folder as "location2" (should be a temporary space, **outside the main root** of IES).

   Clone your (remote) repository into location2.

e) Using "location2" as your current working directory, add a new feature.

   The new feature is adding *logging* support, i.e., the application should **write a log** tracking the operations being executed, as well as any problems that occurred. The log can be directed to the console or to a file (or to both). Use a proper **logging library** for Java such as Log4j 2.

   Note: the sample from the link above uses a configurarion file named log4j2.xml that should be placed in the Maven **resources folder**, following the standard project structure.

f) Once you are satisfied with the implementation, be sure to commit the changes into to the main line (remote repository) and then synchronize your project at the initial "location1".

   Note: make sure your commit messages are relevant and reflect the exercise you have completed. A simple test to check whether your commit messages are useful is to consider that later, you or a different developer, will look into project history using solely the commit messages; it should be possible to get useful, clean flow of the project progress. Have a look at your repository history:

   ```
   $ git log --reverse --oneline
   ```

Use your "notebook" to prepare your study materials; it should be a useful go-to resource for quick reference… Illustrations are welcome.

1.4 Introduction to Docker

Working in large projects, you will often need to deploy assets into "environments", i.e., a set of resources and configurations that must be prepared for each (re)deployment. To optimize deployments and make then portable, it is easier to use "containers" and deploy into containers, instead of configuring each required server from scratch. We will often use containers in IES labs.

Docker setup

a) Start by installing Docker Desktop or follow a non-Desktop approach (for Linux). If you are in Linux, add your user to the docker group.

b) Take the Getting started exercises (up to Part 5).

c) The management of Docker servers can be done either from the Docker Desktop app or the CLI. Sometimes, in a server environment, it would be useful to rely on a web interface to manage Docker.

   The Portainer app is a web application that facilitates the management of Docker containers. Interesting enough, Portainer is deployed as a container itself.

   Install the Portainer CE server app using the "Docker" deployment option.

   Note: default installation assumes that ports 8000 and 9000 are available in your machine. Otherwise, just change the port mapping when issuing the Docker *run* command.

   Afterwards, access http://localhost:9000 (in the first access, set an administration password and choose the Docker *containers* option, not Kubernetes).

Multiple services (Docker compose)

d) More often than not, deployment environments required several interdependent services, mapped into different containers. In those cases, it is convenient to define a "graph" of services and corresponding containers, using the "Docker composer" tool.

Complete the tutorial exercise available from Docker documentation. You will use a "composition" of two containers (Flask service, depending in Redis).

Remember to groom your notebook...

1.5 Wrapping-up & integrating concepts

Let us refactor the weather forecast project to accommodate the scenario in Figure 1.

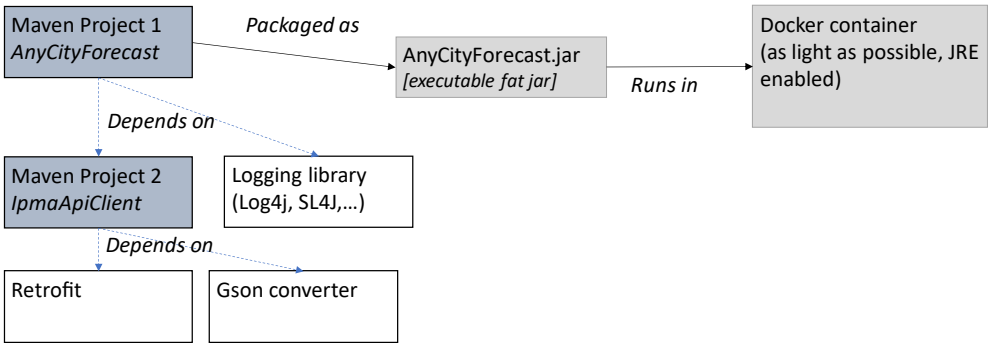


Figure 1

- Project 2: IpmaApiClient → Does not have user interaction. Has the programmatic interface to make requests to the IPMA API for weather forecasts for a city. [refactor from previous exercises]
- Project 1: AnyCityForecast → simple application that creates periodic forecast requests for random cities (e.g.: a random city at every 20 secs). No user interaction required. Outputs the forecast results to a console logger. Project 1 does not handle the network requests! It just calls classes on Project 2 for that.
- *Dockerize* the AnyCityForecast application, i.e., it should run inside a container; starting the container, starts de application; stop the container to stop the application.

  Note: the link above shows a way to build the runnable application *jar* that works when the project does not need to package additional (external) dependencies. In this case, however, there are more dependencies that should be included in a "fat *jar*". Consider an approach to build **an executable *jar* that includes everything** instead, e.g., by using the Apache Maven Shade plugin (section #2.3 of the referenced page).
- Inspect the container logs to observe the (periodic) output of the application.

Don't change the response for section 1.2; create two new projects (in the section 1.5 folder) and refactor the code you may need. Do not forget to handle basic exceptions graciously.
(Remember to stop the container to stop the application.)

Work submission:

- You submit your work by committing into your personal Git repository. Since your notebook is included in the repository space, it will be committed to.
- There should be a special commit at the end of each lab to clarify you have completed all activities, e.g.:

```
$ git commit -m "Lab 1 completed."
```

(And don't forget to push the changes upstream!)