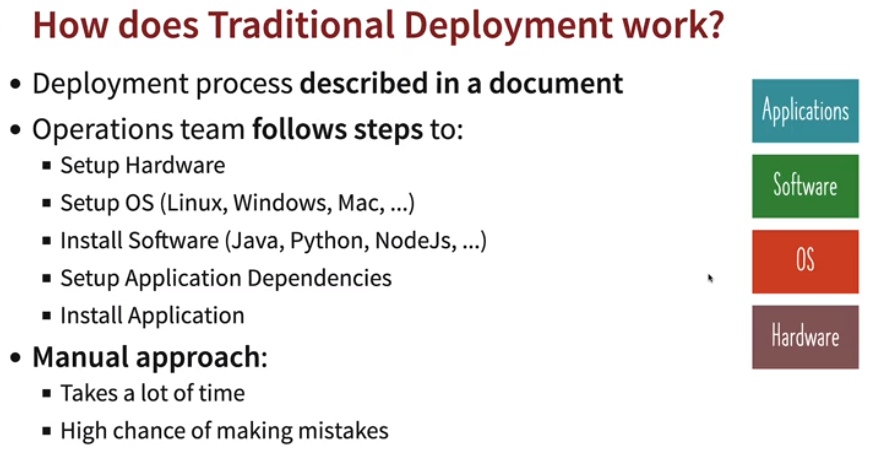
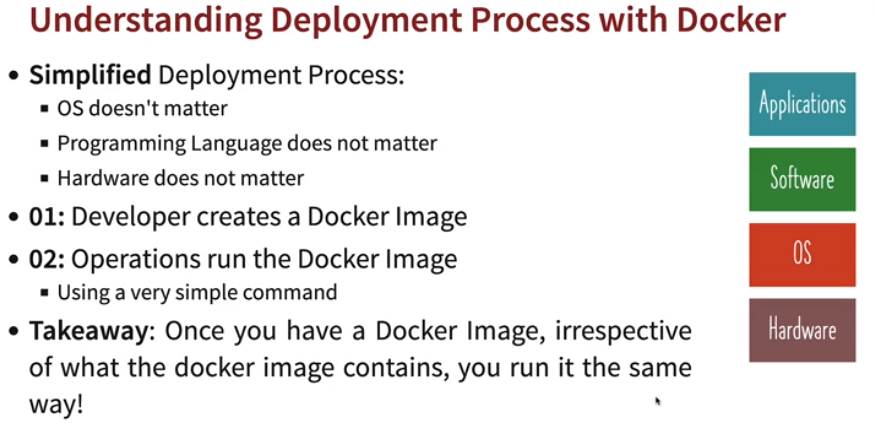
**DOCKER**

In traditional deployment generally operations or developers needed to do setup hardware, setup OS, install software, application dependencies and then install application. This process takes a lot of time and high chances of some mistakes. *Docker* reduces this complexity by doing all of this.

******

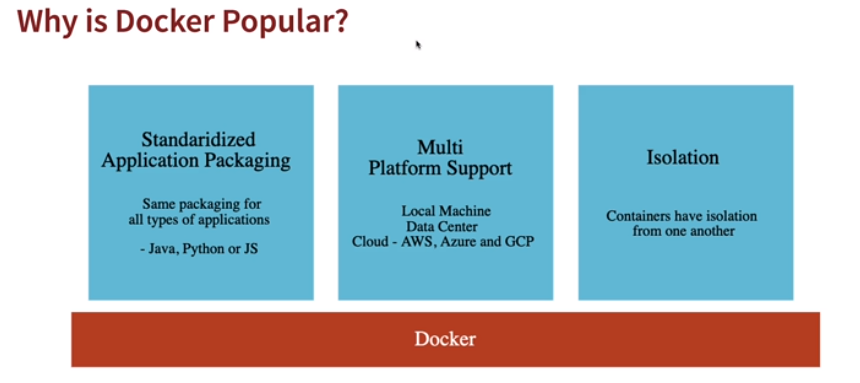
*Docker* has simplified deployment process as it is platform independent, irrespective of language and hardware used.

******

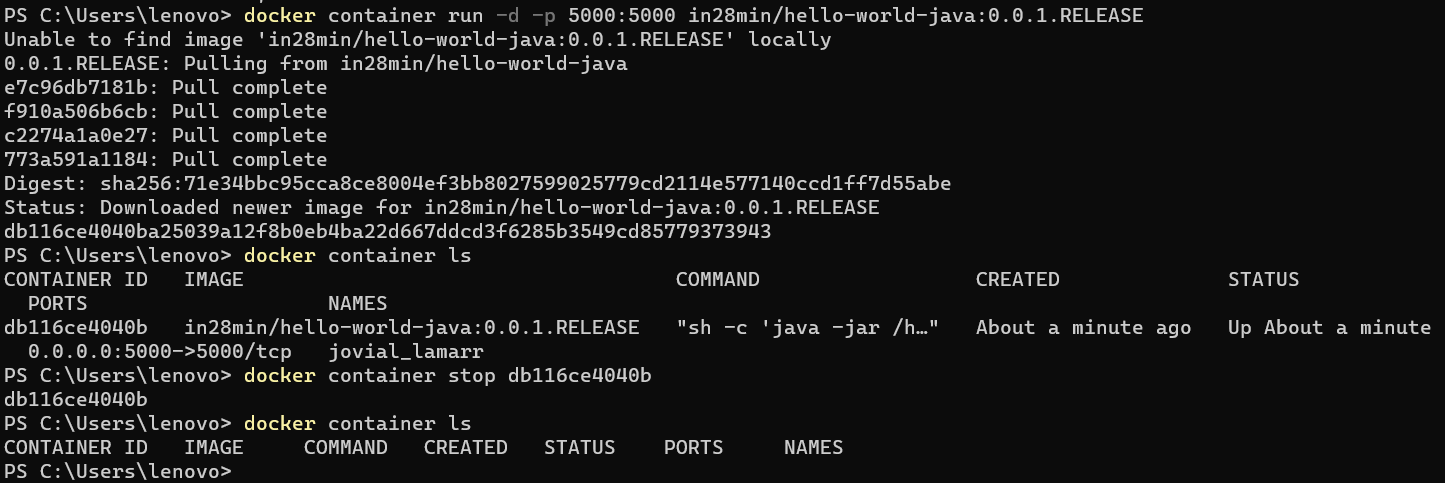
*Docker* image has everything configured like OS, application runtime (JDK, Nodejs etc) and application dependencies as well. Can run *Docker image* at anywhere which has *Docker RE*.

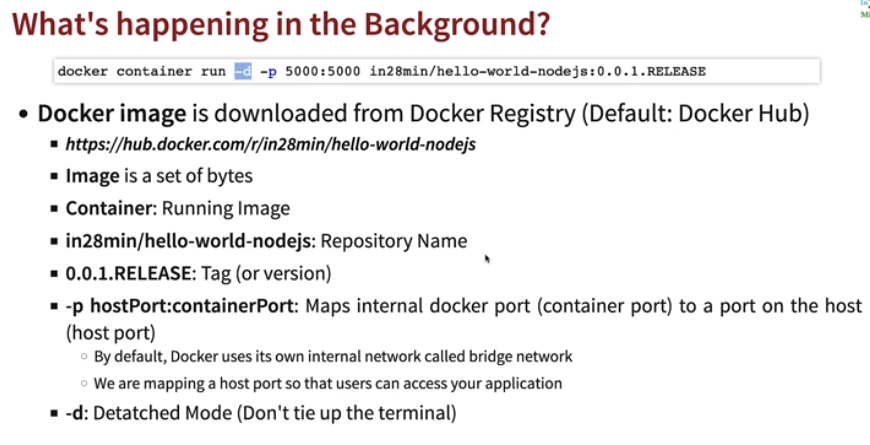
******

* **Understanding how Docker works:** 
  + *Docker* provides same packaging for all kinds of application (Java, python, JS etc).
  + *Docker* provides multi-platform support whether it’s local machine, Data center or Cloud (AWS or GCP).
  + *Docker containers* are isolated from one another. Means same machine can have multiple containers running independent from each other.

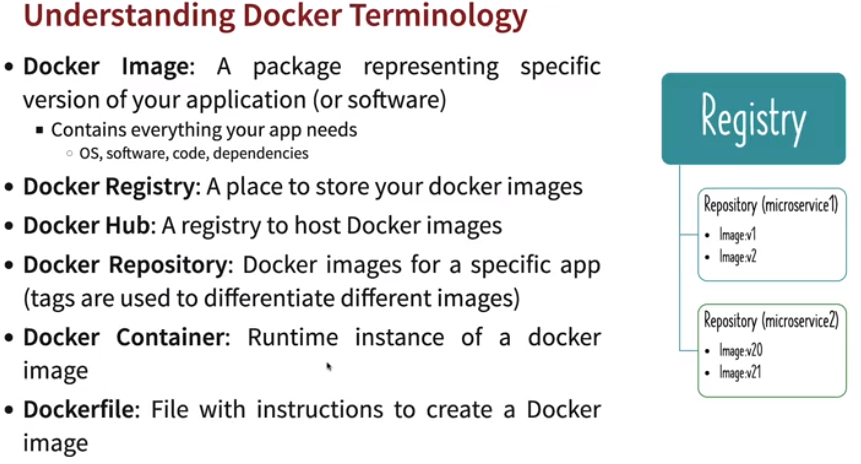
******

* + *Docker images* are downloaded from *Docker Registry (Docker Hub)* to local*.*
  + *Docker Image* is set of bytes.
  + *Docker container* is just running *Image.*
  + *Docker Image is a set of bytes. Running Image is called a Container. You can create multiple containers for the same image.*
  + *docker container run -d -p hostport:containerport <Image\_Name>:0.0.1.RELEASE* 
    - The above command can be used to run *Docker Container.*
    - *-p hostport:containerport:* Maps internal *Docker port (container port)* to a port host.Host port number can be different but needs to be open whereas container port can’t be changed only developers can do.
    - *-d (detached mode):* If we don’t use *–d* in command then the command will run till the lifetime of *Docker container* or until it’s stopped.
  + *docker container stop <ContainerId>:* To stop a running *Docker container.*
  + *docker container ls:* Lists all running *Docker container.*

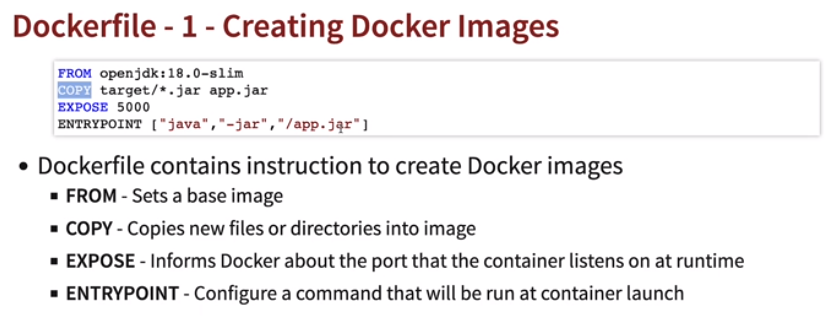
******

******

* **Docker Terminology:**

******

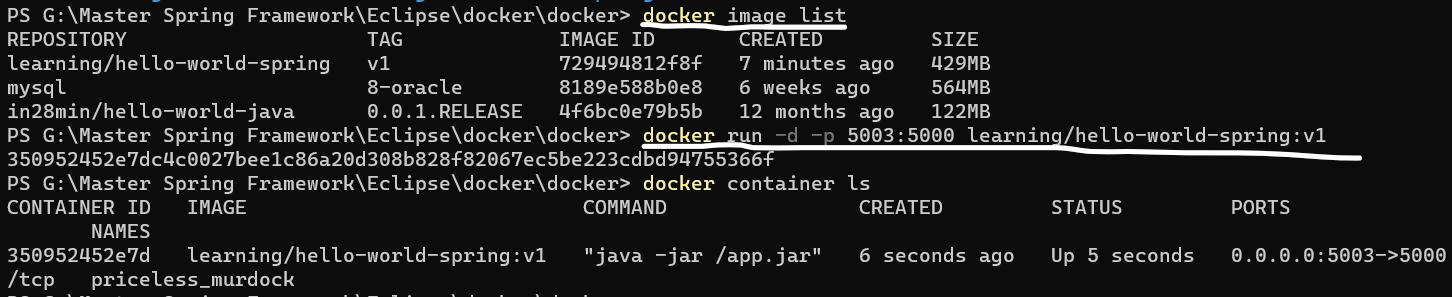
* + *Docker Image* is a package representing specific version of application.
  + *Docker Registry* is a place to keep or store *Docker Images.*
  + *Docker Hub* is to host *Docker images.*
  + *Docker Repository* it is repository for specific application where all versions are kept with tags and version.
  + *Docker Container* is a running *Docker image* or running instance of *Docker Image.*
  + *DockerFile* is a file that has instructions to create a *Docker Image.*
* **Creating a docker image:**

**

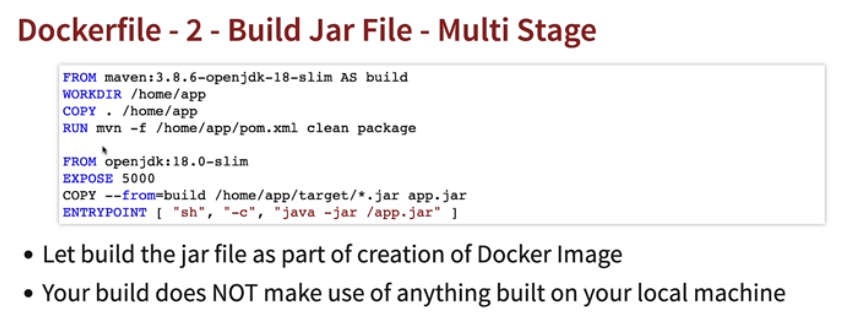
* + Create a *Dockerfile* in your project.
  + Paste following commands in *Dockerfile.*
    - *FROM <jdk> 🡪* To run spring application we need jdk and we can find one from *docker hub*.
    - *COPY target/\*.jar app.jar 🡪* Once do maven install to generate a jar file. Once a jar of your application created in target folder, we can use the command to copy generated jar into *Docker Image* as name *app.jar.*
    - *EXPOSE <PORT> 🡪* Port on which application is running inside container.
    - *ENTRYPOINT [“java”, “-jar”, “/app.jar”] 🡪* Command to launch the java application.
  + Open cmd in folder where *Dockerfile* located.
  + Run below command to create a tag name and image.
  + *docker build -t <imageName>:<version> .*

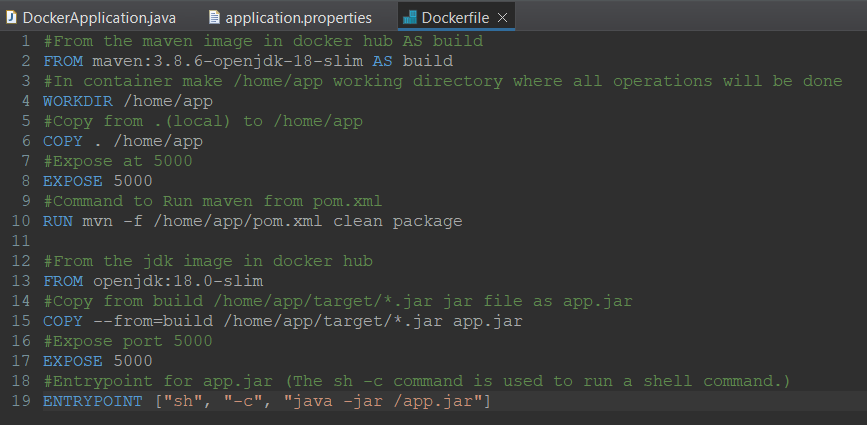
**

* + Running new image.
    - *docker container list*
    - *docker run –d –p <hostport>:<containerport> <imageName>:<version>*

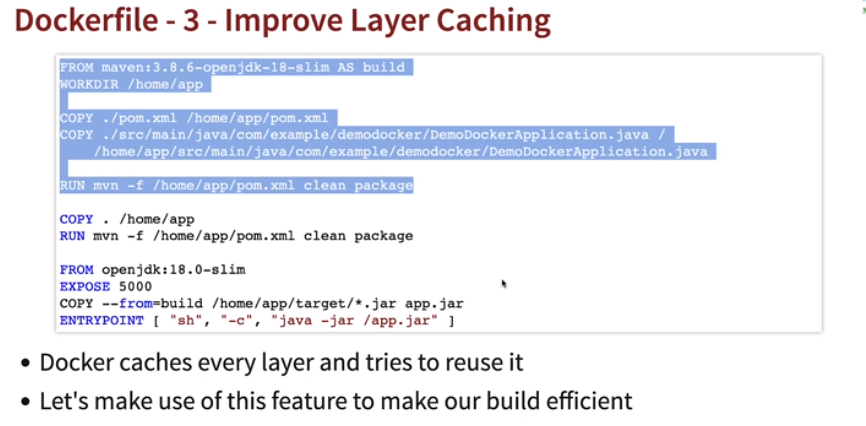
**

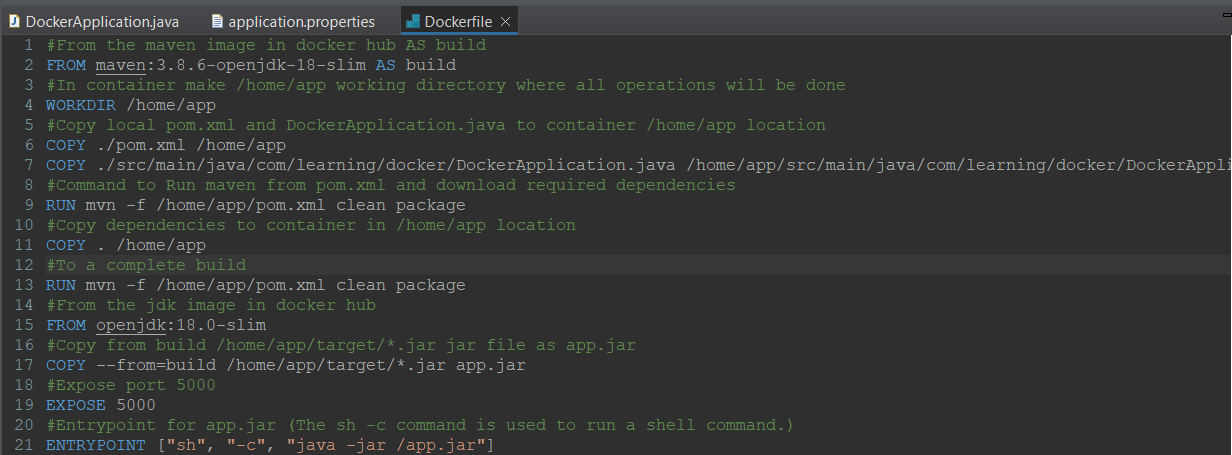
* **Multi stage build:** In the above we’re building *Docker image* from local created application jar file, which may or may not be compatible with other systems and we want to build the application build to happen inside container itself or as part of *Docker image* creation. In that case we can use multi stage.

**

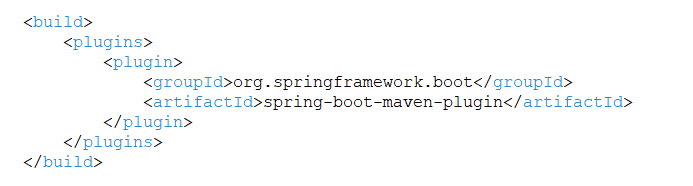
******

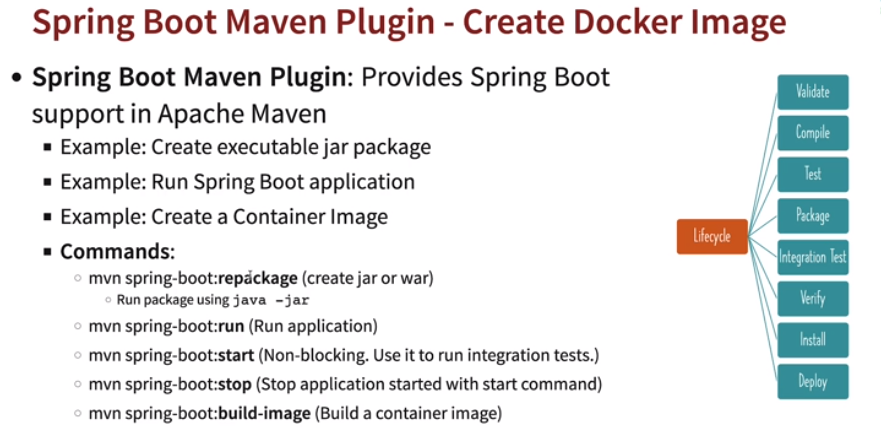
* **Optimizing Docker file further:** If some files are not changing then docker reuses them to build quickly. Indirectly not changed files are cached to reduce time to copy again during next build. In our example *pom.xml* and *main.java* is not going to change. So we can copy these files directly and do maven clean to download dependencies and once dependencies are copied we can copy rest files like controller, service etc and again do a *mvn clean* inside container.

******

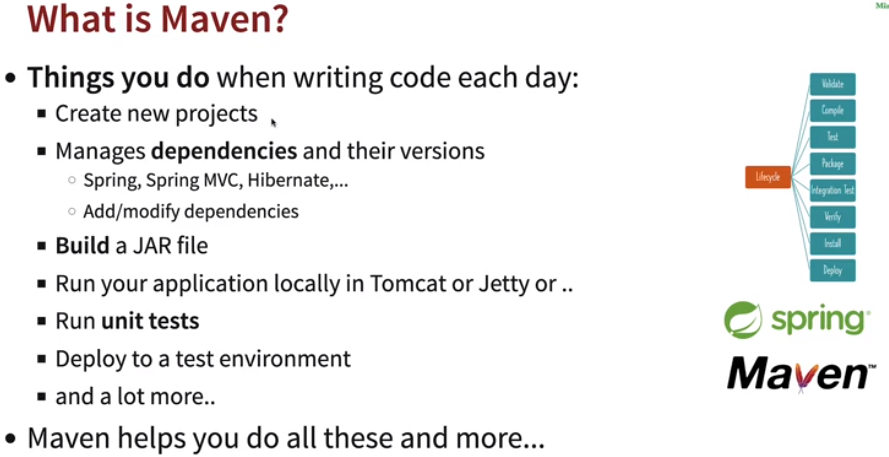
******

* **Spring boot Maven plugin:** In case we don’t want to use create images using *Dockerfile,* we can *spring boot maven plugin* provided in *pom.xml.* It reduces the size of image created when compared to *Dockerfile* and is also much faster.
  + Command to create an image using maven plugin: *Run as 🡪 Maven build 🡪 spring-boot:build-image*.
  + This will create an image which can be run by any Image Running environment and *Docker* is one of them to run images as containers.

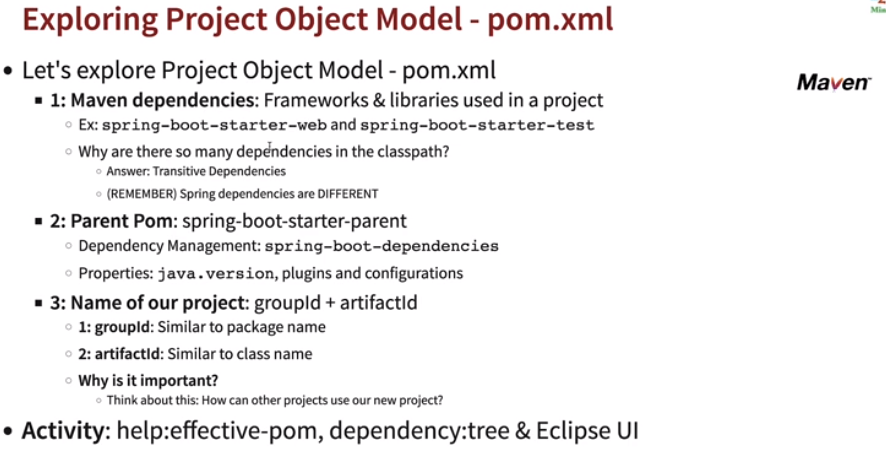
****

******

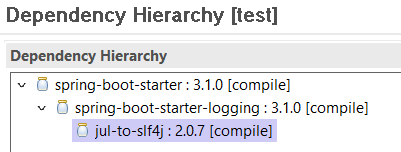
* **Maven:** *Maven* is a build tool that helps in building *Jar* files, manage dependencies and their versions.

****

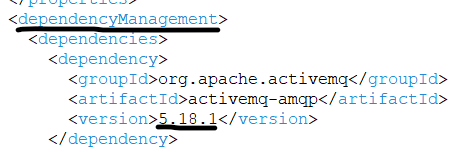
* **Group Id:** *GroupId* is similar to package name.
* **Artifact Id:** *ArtifactId* is the name of the project or application.
* **POM.xml (Project Object Model):** *POM.xml* is an XML file that contains information about the project and configuration details used by Maven to build the project like dependencies, version etc.

****

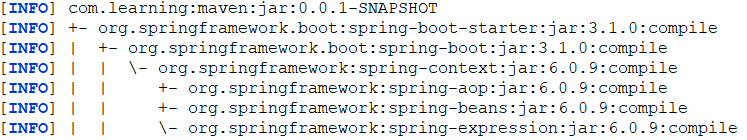
* + Dependencies can depend on other dependencies which are known as *transitive dependencies.* Ex: sl4j is transitive dependency for spring-boot-starter-logging.

****

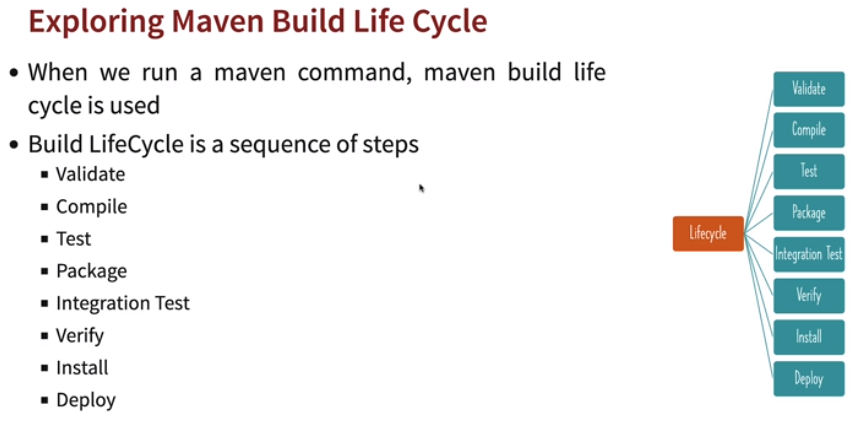
* In *pom.xml* under *<dependencies/>* we specify the dependencies to be included in our application.
* *Dependency management* is used to specify the versions of dependencies, so in case if we do not specify the version is our *pom.xml,* it can get dependencies from *parent pom* or *effective pom <dependencyManagement/>.*

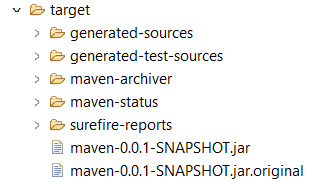
****

* *Maven dependencies* are set of libraries and frameworks for project as a whole, whereas *Spring dependencies* are dependencies required for a particular class or entities.
* *dependency:tree:* Command will give information about *dependencies* and their *transitive dependencies* in tree structure format in console.

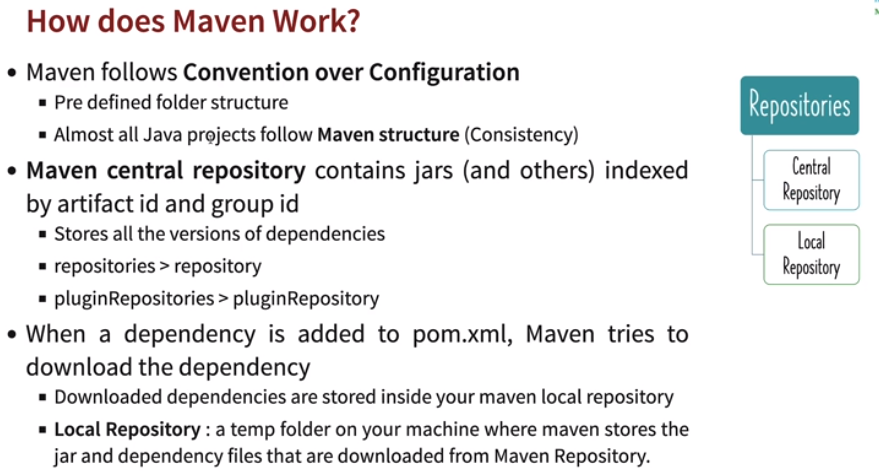
**

* *Effective POM* combines all the default settings from the super POM file and the configuration defined in our application POM.
* *help:effective-pom:* Command will the generate the *effective POM* in console.
* **Maven Lifecycle:** During Maven lifecycle, *install* maven copies resources required, compiles java code and java test code, build a jar file and place them logs and jar in *target* folder.

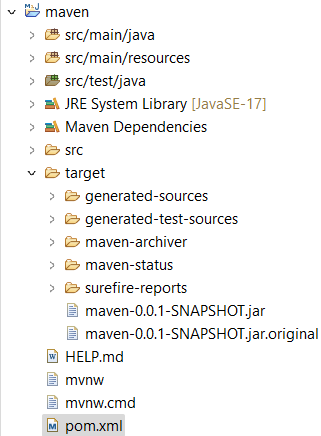
**

**

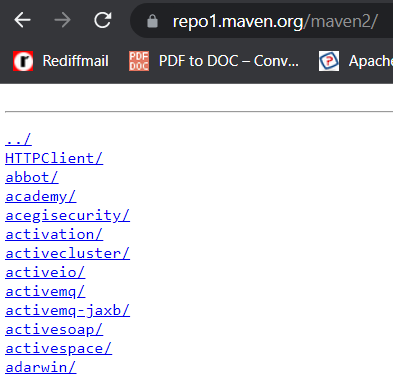
* **Maven Workflow:**

**

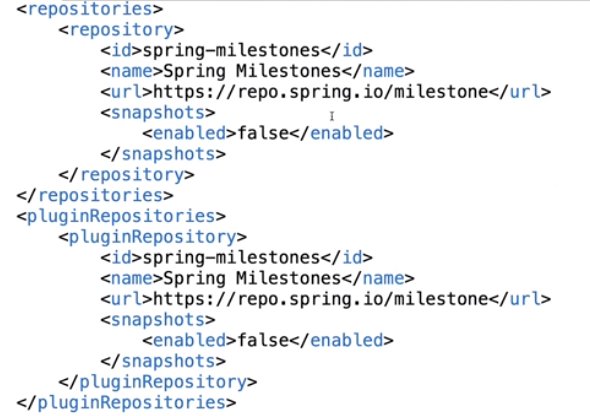
* + Maven follows a convention for folder structure like *src/main/java* for program java classesor *src/test/java* for test java classes and almost all projects follow same project structure.

**

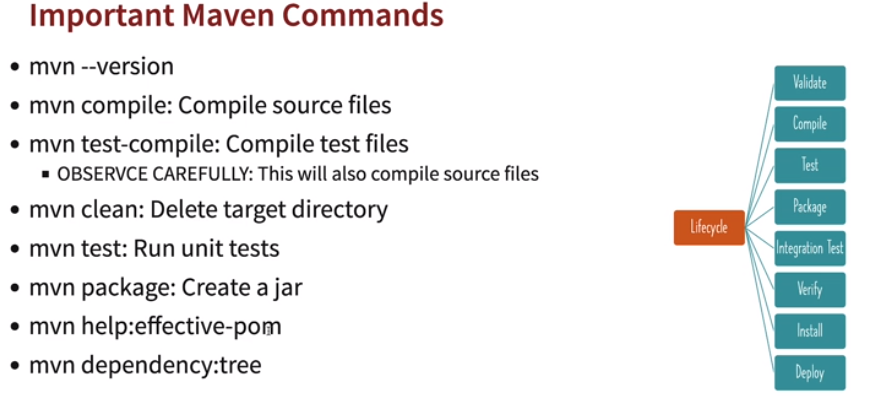
* + *Maven central repository (https://repo1.maven.org/maven2/)* contains all jars indexed by artifact id and group id.

**

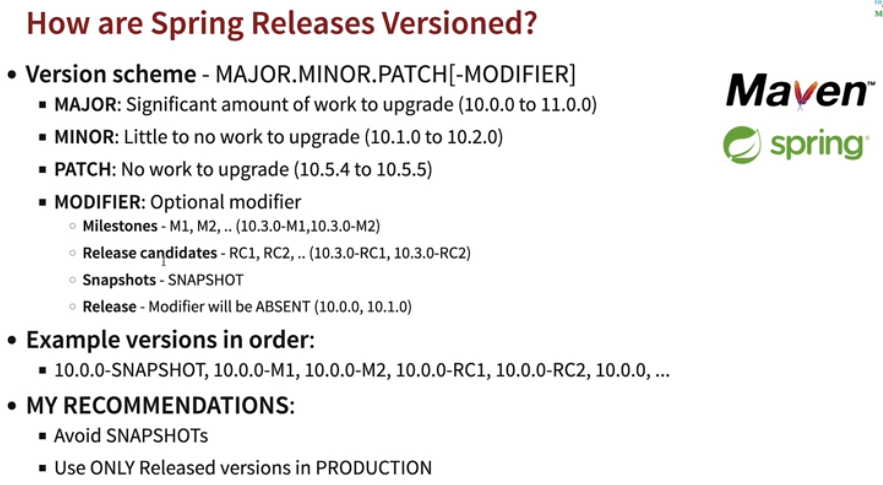
* + In *pom.xml,* maven will look for dependencies first from central repository. If in case the dependencies are not in *maven central repository* we can specify other repository links or plugin repository links.

**

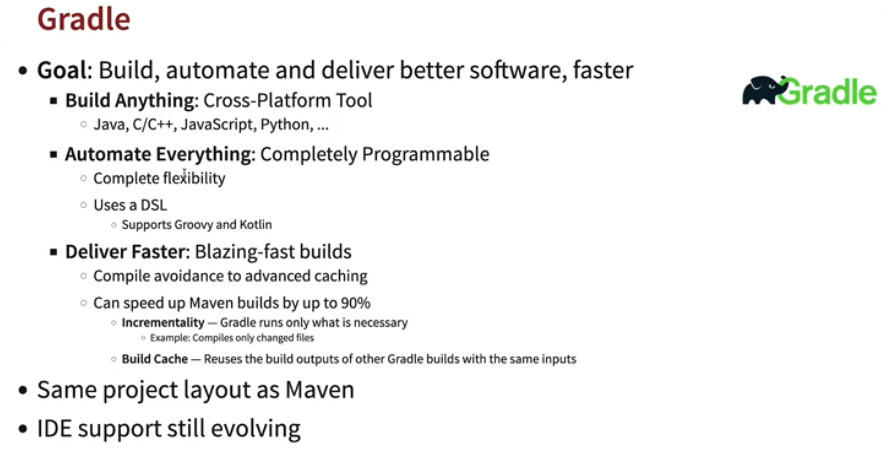
* + Once dependencies specified in *pom.xml,* maven tries to download dependencies from *maven central repository,* dependencies are downloaded to local *.m2/…* folder and in project the dependencies are used from the local *.m2/…* folder.
  + *Maven Commands:*
    - *mvn --version:* Version of maven.
    - *mvn compile:* Compiles only source files.
    - *mvn test-compile:* Compiles only test file including source files.
    - *mvn clean:* Clears target folder.
    - *mvn package:* Creates the jar file.
    - *mvn test:* Run unit test files.
    - *mvn spring-boot:build-image:* Can be used to build a container image using Spring Boot Maven Plugin.

**

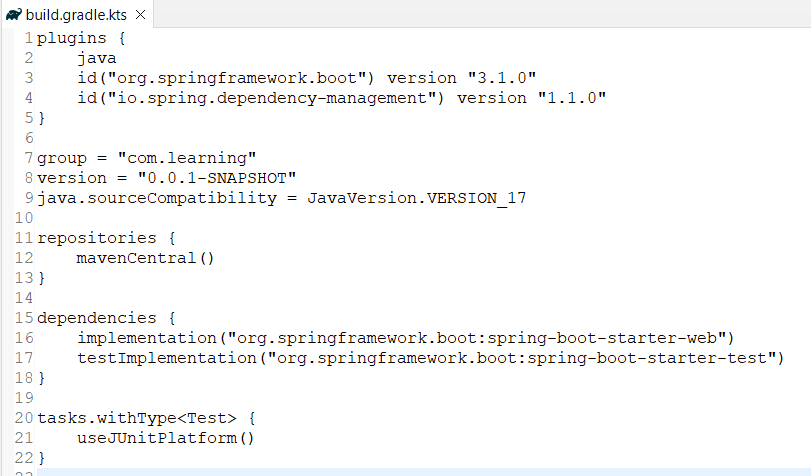
* **Spring version:** Avoid snapshot releases instead use *RELEASED* version for production ready application.
  + **Version Scheme:**
    - *Major changes:* Ex: 11.0.0 to 12.0.0
    - *Minor changes:* Ex: 10.1.0 to 10.2.0
    - *Patch changes:* Ex: 10.1.1 to 10.1.2
    - *Modifier:* Ex: M (Milestones), RC (Release Candidates), SNAPSHOTS, RELEASE.

**

* **Gradle:** *Gradle* is used build, automate and develop software faster.
  + It is cross-platform tool and can be used for Java, python, JS etc.
  + Completely programmable, flexible and uses DSL instead of XML like maven.
  + Faster than maven builds by 90%.
  + Same project structure as Maven however IDE support is still improving.

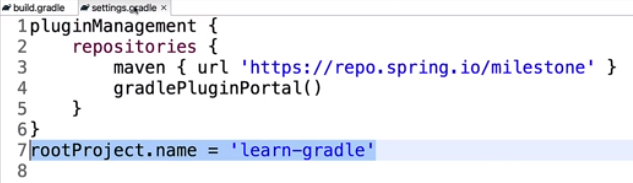
**

* **Build.gradle.kts:** Unlike *pom.xml* in maven we specify dependency in *build.gradle* when using *gradle*.
  + *Repositories:* Contains the maven central repo. *Gradle* also uses maven central repository.
  + *Dependencies:* Dependencies are specified in *dependencies{}.*

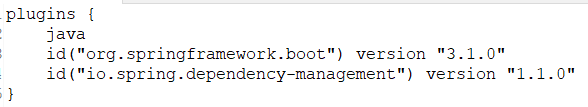
**

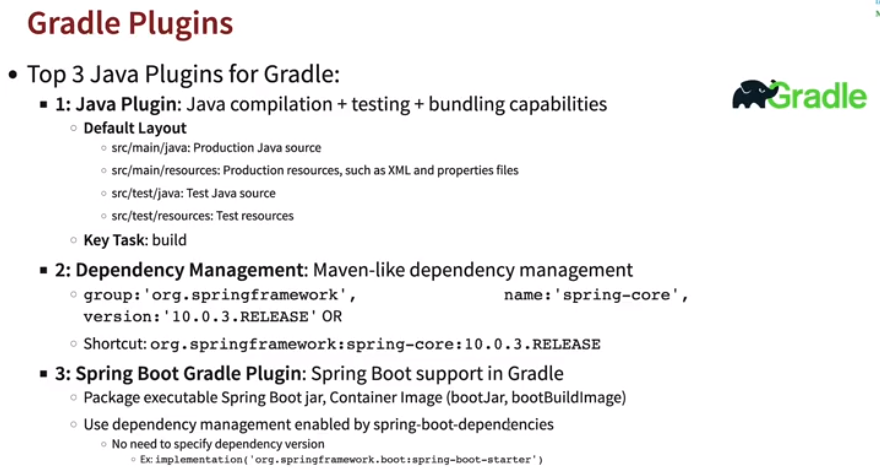
**

* **setting.build.gradle.kts:** Contains root project name, plugin management and other repository links.

**

* **Gradle plugins:** 
  + *Java plugin:* It is used for java compilation, testing and packaging.
  + *Dependency Management:* Maven like dependency management. Basically in *Gradle* we need not to defined dependency version it is will manage from plugins defined.

**

**

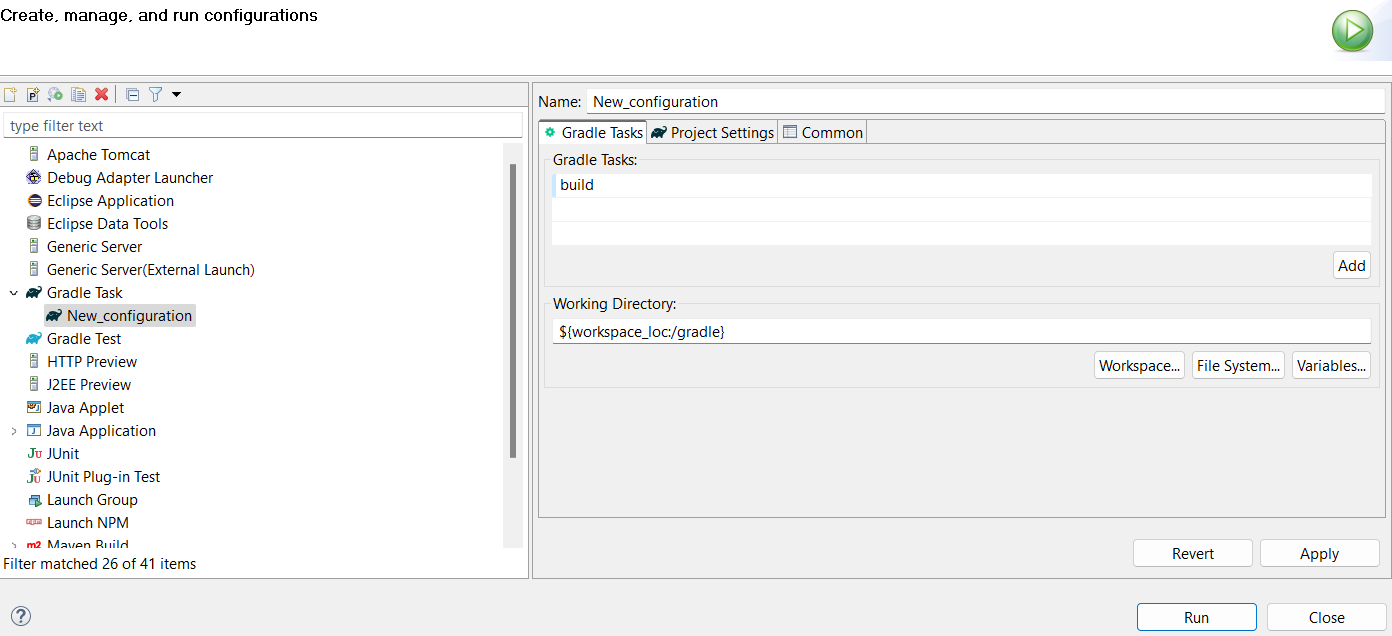
* **Maven vs Gradle:**
  + *Maven* is simple, familiar with lot programmers and restrictive.
  + *Gradle* is having faster build times and less verbose.
  + If project using much more build time then we can go with *Gradle*, otherwise *Maven.*

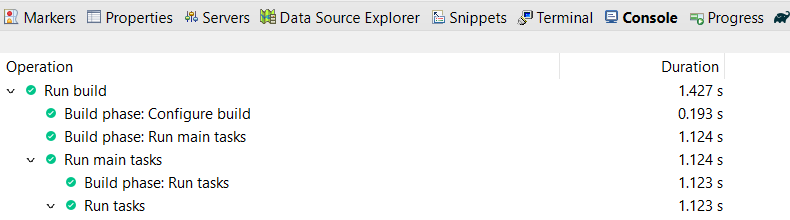
**

* + *Gradle* can be used to run tasks as well, hence anyone can easily modify code.

**

* **Building gradle project:** *Run As 🡪 Gradle Task 🡪 New Configuration 🡪 Task Name (build) 🡪 Select project 🡪 Apply 🡪 Run.*

**

**