MET CS 622:
Build and Deployment Tools

Reza Rawassizadeh

Build and Deployment Tools

- Source Control
- Build Tools
  - Apache ANT
  - Apache Maven
  - Gradle
- Continuous Integration

Build and Deployment Tools

- **Source Control**
- Build Tools
  - Apache ANT
  - Apache Maven
  - Gradle
- Continuous Integration

Why do we need source code control?

Source Control

- **Git:** It is the most used source control application, especially github.com is the most source code repository used around the world. It is compatible with HTTP, FTP and SSH protocol and very robust for distributed source control.

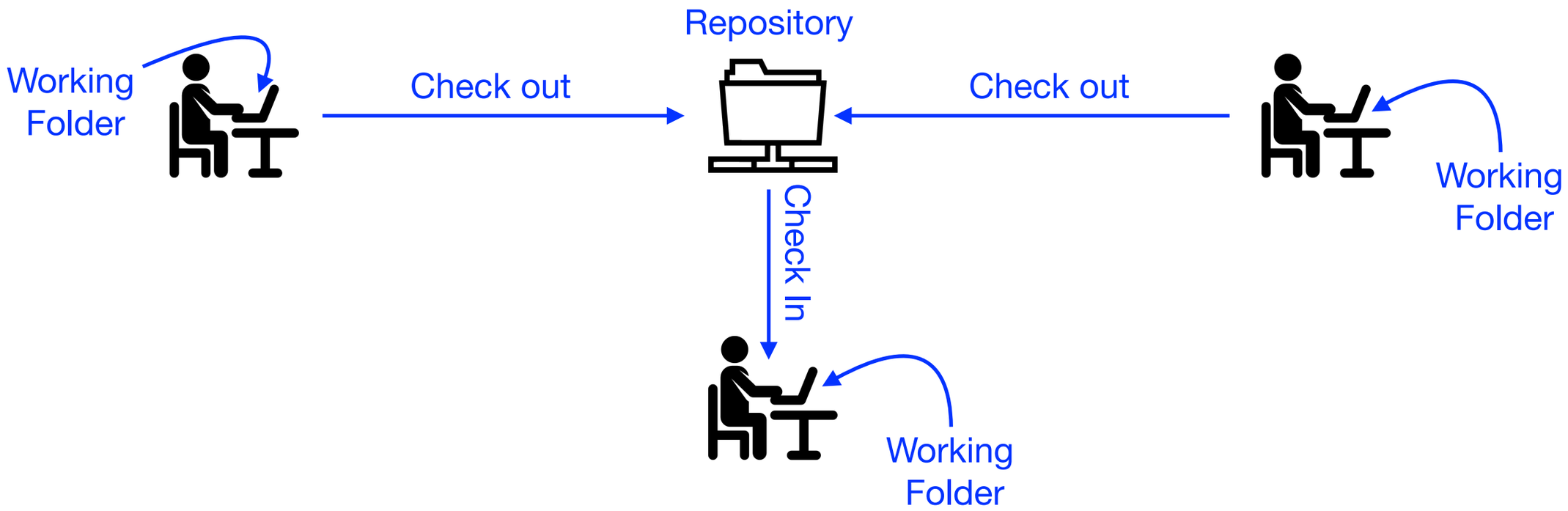
- **CVS:** It is simpler than Git with easy command line, but platform dependent. Therefore, it is not good for distributed source control.

- **SVN:** A very easy to learn source code versioning system. Easy to setup and easy to administrate, unlike Git. However, it is much more limited than Git.

- **Mercurial:** It is a newest source control. Git is complicated to learn and Mercurial tries to reduce the complexity of Git.

Source Control Concepts

- **Repository:** It is the database that stores our code files and content. It also maintain a history of changes.
- **Working folder (sandbox):** It is the local folder that we are doing our development inside it and it is isolated from the repository.
- **Check out:** Sending our local code from the working folder into repository is called *check out*.
- **Check In:** Downloading others' code from repository to the local working folder is called *check in*. It also includes the **Commit** process, which means accepting the new changes that should go to the repository.

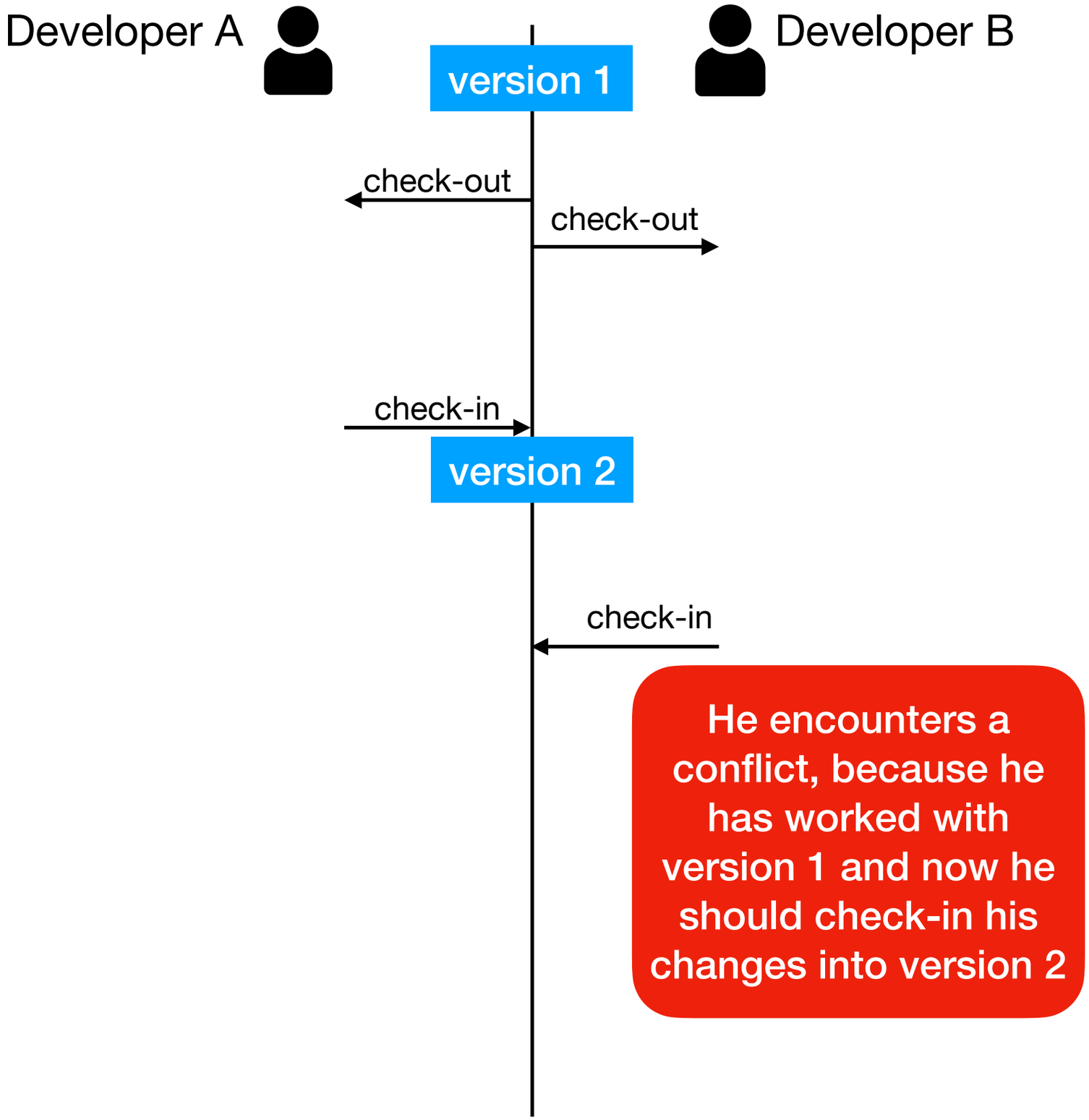


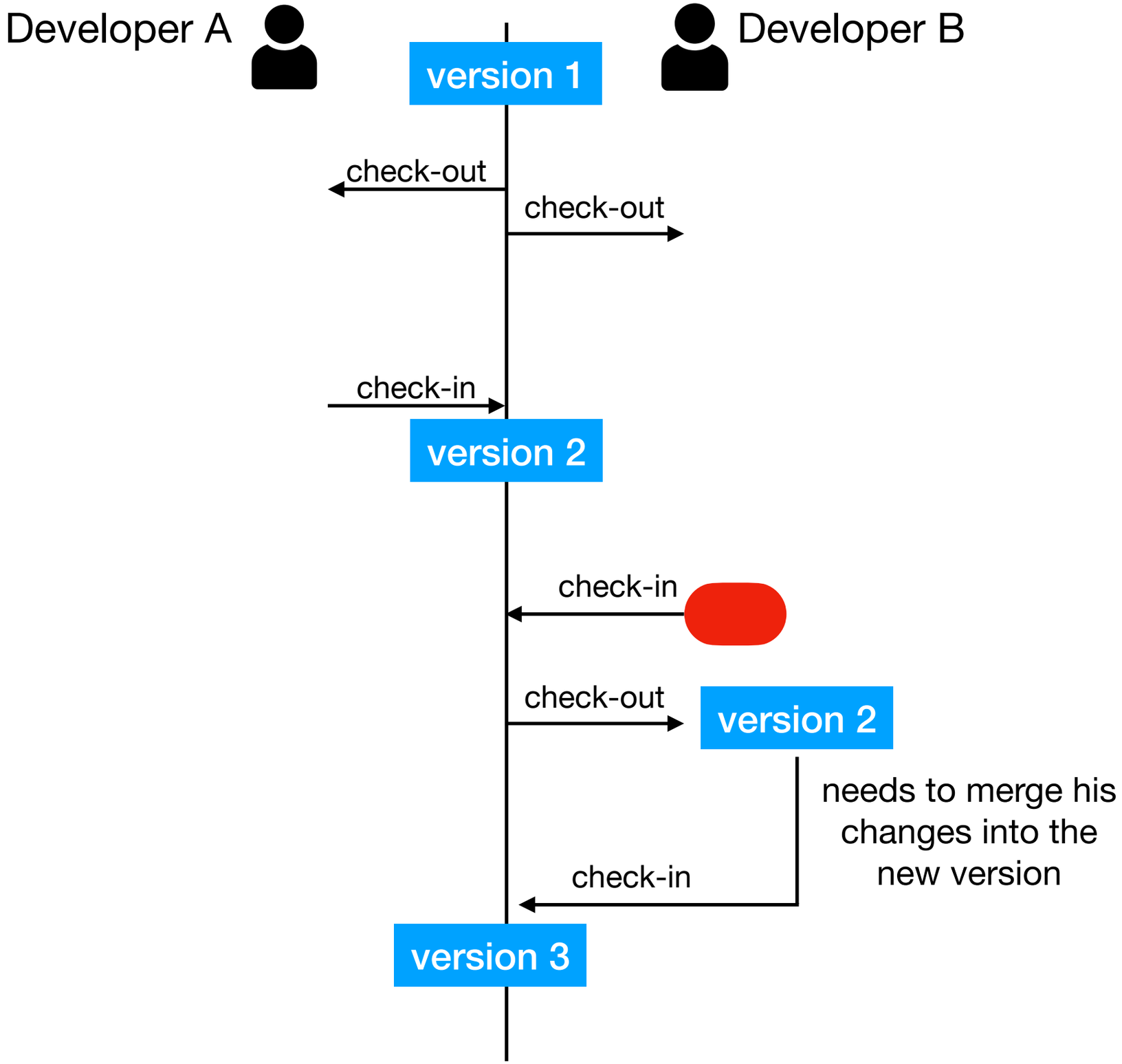
Source Control Concepts

- **Revision:** If a developer makes a successful commit/check-in to the repository. A new version of code is created in the repository and it gets a new revision number.

- **Branching:** Some times we need to test something (e.g. a new library or component), but we are not sure if we should bring it to the product or not. In those cases we create a new branch, which is different than the main code (**trunk**) and work on that branch, until we are sure that our changes can get into trunk.

- **Merging and Conflicts:** Developer A performs a check-in on a code and starts to work on it. Developer B did the same, developer A check-out performs a his changes. Now developer B wants to submit his changes, but he realize the code in the repository has been changed. There is a conflict and developer B should resolve it by **merging** his code into the new changes. This is usually a manual process.


Developer A
Developer B
version 1
check-out
check-out
check-in
version 2
check-in
He encounters a conflict, because he has worked with version 1 and now he should check-in his changes into version 2


Developer A
Developer B
version 1
check-out
check-out
check-in
version 2
check-in
check-out
version 2
needs to merge his changes into the new version
check-in
version 3

Git Work Flow

Workspace

Staging Area

Local Repository (Head)

Remote Repository

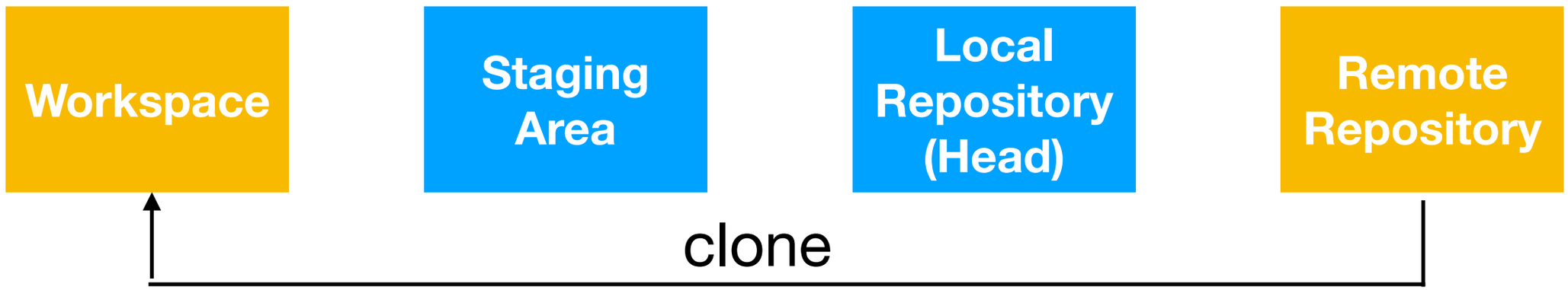
Committed:
Data are Stored in Local Repository

Modified:
Files are changed and not stored in Local Repository

Staged:
Files are indexed and in the staging area. They are tagged for the next commit

Sending File from Local to Remote Repo

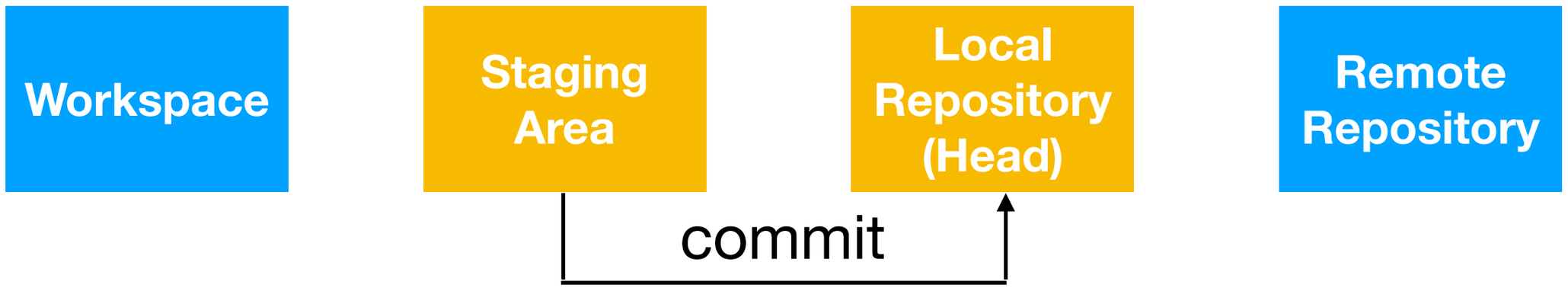
- clone: getting the project from remote repository and downloading them into the local repository.



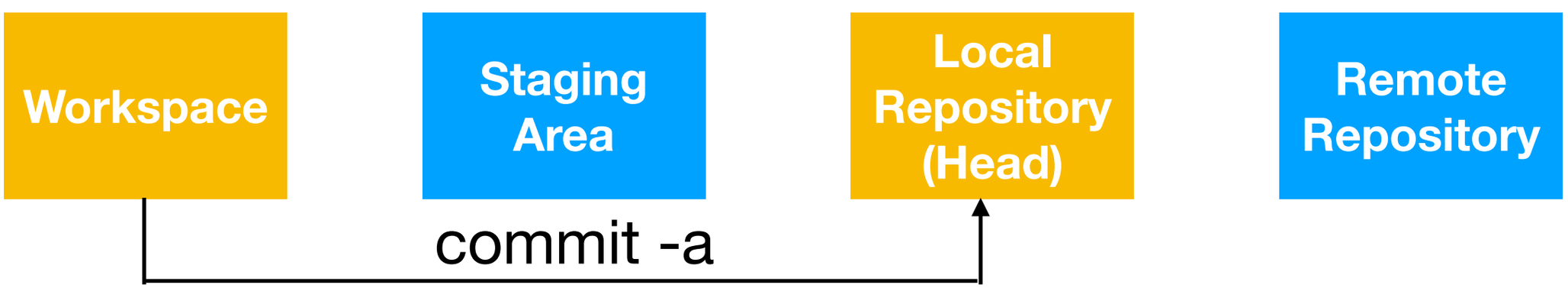
- add: add file(s) from workspace to staging area. The files are not committed yet.

Sending File from Local to Remote Repo

- commit: files from staging area are moved to the local repository.



- "`commit -a`" stage the file and commit, so no need to add them.



Sending File from Local to Remote Repo

- Push: push changes from local repository to the remote repository

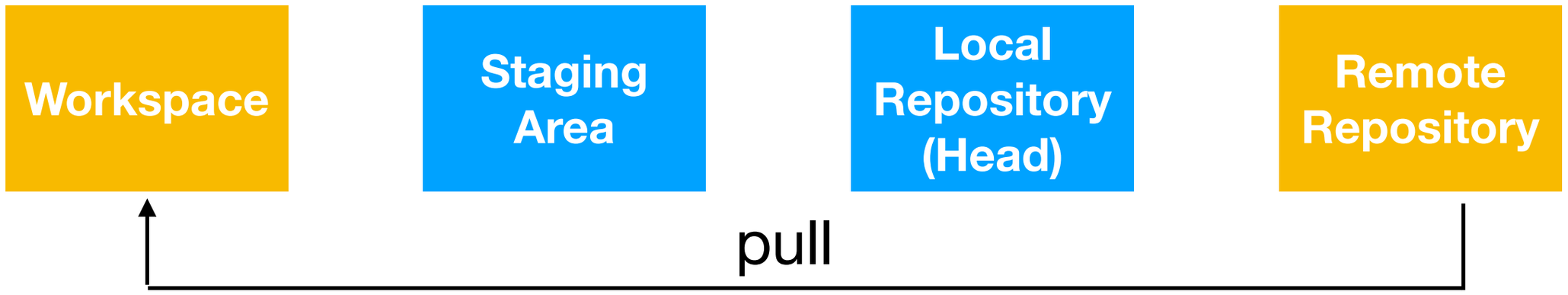
Getting File from Remote Repo to Local Repo

- fetch: get files from remote repo. to local repo.

- merge: get files from local repo. to workspace.

Getting File from Remote Repo to Local Repo

- pull: it is a short cut that directly get from remote repo to local repo (fetch+merge).



Getting File Comparison

- diff HEAD: we usually need to compare the content of remote repo (local repo) with our workspace.

- diff: Before commit we need to check what are things prepared to commit and do we need to add/remove something or not.

Experiment Github Together

```
echo "# test622" >> README.md

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/Rezar/test622.git

git push -u origin main
```

Build and Deployment Tools

- Source Control
- **Build Tools**
  - Apache ANT
  - Apache Maven
  - Graddle
- Continuous Integration

introduced in 2000

Apache ANT

- ANT (Another Neat Tool) is a build tool written in XML for Java.
- ANT is automating following tasks for the developer:
  - Compile
  - Packaging binary files
  - Deploying binary files on the server
  - Testing changes
  - Copying codes and files from one location to another location
- ANT is platform neutral (linux, windows,…)
- ANT is very useful to automate complicated repetitive tasks
- source for ANT tutorial: https://www.tutorialspoint.com/ant

Installing ANT

- Download ANT from here: https://ant.apache.org
- Check and ensure JAVA_HOME is set to the folder where the JDK is installed.
- Create a new environment variable ANT_HOME that points to the installation folder. e.g. `c:\projects\apache-ant-1.8.2-bin`
- Append an ANT_HOME to the PATH environment variable. `c:\projects\apache-ant-1.8.2-bin\bin`
- To verify the installation is performed successfully use this command:

```
C:\>ant -version
```

ANT

- ANT is use a file called **build.xml**. We write all XML build commands inside that file and use a **build.properties** to write constants inside it.

- build.xml

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
  <property file = "build.properties"/>
   <target name = “hello”>
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
   <target name = “info" depends=“hello”>
      <echo>Apache Ant version is ${ant.version} -
            You are at ${classname}
      </echo>
   </target>
</project>
```

- build.properties

```
classname = metcs622
buildversion = 3.3.2
```

ANT Example

```
<?xml version=“1.0”?>
<project name = "Hello World Project" default = "info">
<property file = "build.properties"/>

   <target name = “hello”>
      <echo>1- Hello World - Welcome to Apache Ant!</echo>
   </target>

   <target name = “info" depends=“hello”>
      <echo>2- Apache Ant version is ${ant.version} -
            You are at ${classname}
      </echo>
   </target>

</project>
```

build.xml

- **`name:`** the name of the project
- **`default:`** the default target for build script. A project can include any number of targets but it will starts from the default target.
- **`basedir:`** the root folder of the project
- **`target:`** a collection of tasks we need to run as one unit.

build.xml

```
<target name = "deploy" depends = "package">
   ....
</target>

<target name = "package" depends = “clean, compile">
   ....
</target>

<target name = "clean" >
   ....
</target>

<target name = "compile" >
   ....
</target>
```

1- clean
2- compile
3- package
4- deploy

build.properties

- **`ant.file`**: the location of the build.xml build file.
- **`ant.version`**: the version of the installed apache Ant.
- **`basedir`**: the root folder of the project
- **`ant.project.name`**: the name of the project.
- **`ant.core.lib`**: the location of Ant jar file, to execute our build.xml.
- **`ant.home`**: the home directory of the installed Ant.
- **`ant.java.version`**: the version of the JDK that the Ant should use.

build.properties

- **`ant.file`**: the location of the build.xml build file.
- **`ant.versio`** … e Ant.
- **`basedir`**: the …
- **`ant.projec`** …
- **`ant.core.l`** … cute our build.xml.
- **`ant.home`**: th … ..
- **`ant.java.version`**: the version of the JDK that the Ant should use.

You can define properties in the build.xml file as well.

However, this attitude is not recommended. Please keep always your constant data separate from the code.

ANT Data Types

- **`fileset`**: a collection of files, with filters, used for a specific goal. For example, following specify java source files.

```
<fileset dir = "${src}" casesensitive = "yes">
   <include name = "**/*.java"/>
   <exclude name = "**/*Stub*"/>
</fileset>
```

- **`patternset`**: a pattern which is used to filter files and folders.

```
<patternset id = "java.files.without.stubs">
   <include name = "src/**/*.java"/>
   <exclude name = "src/**/*Stub*"/>
</patternset>
```

- **`filelist`**: is similar to fileset, but require explicit name of files.

```
<filelist id = "config.files" dir = "${webapp.src.folder}">
  <file name = "applicationConfig.xml"/>
  <file name = "web.xml"/>
</filelist>
```

- **`path`**: it is used to specify the class path.

```
<path id = "build.classpath.jar">
   <pathelement path = "${env.J2EE_HOME}/${j2ee.jar}"/>
   <fileset dir = "lib">
      <include name = "**/*.jar"/>
   </fileset>
</path>
```

Create a JAR file with ANT

```
<jar destfile = "${web.dir}/lib/util.jar"
   basedir = "${build.dir}/classes"
   includes = "faxapp/util/**"
   excludes = “**/*.java”>
   <manifest>
      <attribute name = "Main-Class" value =
"com.tutorialspoint.util.FaxUtil"/>
   </manifest>
</jar>
```

Java de-facto Standards for Files and Folders

- Keep all source files (*.java) inside **src** folder.
- Keep class files (*.class) inside **bin** folder or **classes**.
- Database scripts inside **db** folder.
- All web related files, including css, images, and js files inside **war** folder. To create a war file we need **META-INF** file.
- All third party libraries should be stored inside **lib** folder.
- All other files in **res** folder.

```xml
<?xml version = "1.0"?>
<project name = "fax" basedir = "." default = "build">
   <property name = "src.dir" value = "src"/>
   <property name = "web.dir" value = "war"/>
   <property name = "build.dir" value = "${web.dir}/WEB-INF/classes"/>
   <property name = "name" value = "fax"/>

   <path id = "master-classpath">
      <fileset dir = "${web.dir}/WEB-INF/lib">
         <include name = "*.jar"/>
      </fileset>

      <pathelement path = "${build.dir}"/>
   </path>

   <target name = "build" description = "Compile source tree java files">
      <mkdir dir = "${build.dir}"/>

      <javac destdir = "${build.dir}" source = "1.5" target = "1.5">
         <src path = "${src.dir}"/>
         <classpath refid = "master-classpath"/>
      </javac>
   </target>

   <target name = "clean" description = "Clean output directories">
      <delete>
         <fileset dir = "${build.dir}">
            <include name = "**/*.class"/>
         </fileset>
      </delete>
   </target>
</project>
```

…# ant <— build

…# ant clean <— clean

https://www.tutorialspoint.com/ant/ant_build_project.htm

introduced in 2004

Installing Maven

1. Check Java version to be JDK 1.7 or newer version.

   ```
   java -version
   ```

2. download and run: `unzip apache-maven-3.6.1-bin.zip`

3. OPTIONAL: `export PATH=/opt/apache-maven-3.6.1/bin:$PATH`

Maven

- It is a build tool based on Project Object Model (POM) concept.
- The build process is defined into **pom.xml** file.
- Maven has a default behavior for each project. Therefore, developers do not need to configure it by themselves.
- Assuming `${basedir}` points to the project location, Maven has following default locations:

```
${basedir}/src/main/java  –> Source code

${basedir}/src/main/resources –> Resources

${basedir}/target –> Compiled code

${basedir}/target/classes –> the result JAR file
```

POM.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
```

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
      <scope>compile</scope>
   </dependency>
 </dependencies>

</project>
```

POM.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

<dependencies>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
      <scope>compile</scope>
   </dependency>

</dependencies>

</project>
```

groupId: the unique organization name that owns the project.

artifactId: is the name the project is known by.

version: the current version of the project.

Why Maven

- Superior dependency management over ANT
  - it does not need to store 3rd party library local.
  - adds or updates new dependency easily and quickly to all projects.
  - can identify unused dependencies.
- More powerful build features
  - it can add java doc
  - integrates and run unit testing (e.g. JUnit) easily
  - analyzes dependencies of the project
  - creates reports about the build process

Why Maven

- Superior dependency management over ANT
  - **It doesn't need to store 3rd party library local.**

A jar file can be shared across many different projects. This is very useful, in team working environment that all developers should use the same version of a library.

- possible to add java doc
- integrate and run unit testing (e.g. JUnit) easily
- Analyze dependencies of the project
- create reports about the build process.

Why Maven

- Superior dependency management over ANT
  - It doesn't need to store 3rd party library local.
  - **Add or update new dependency easily and quickly.**

In a team project once the library will get updated and then developers can use it, without even knowing that a library is updated.

  - integrate and run unit testing (e.g. JUnit) easily
  - Analyze dependencies of the project
  - create reports about the build process.

Maven Build Lifecycle

- The build process in maven follows the three life cycle, and there could the three life cycles in Maven.

  - **default:** it is the main life cycle as it's responsible for project deployment.

  - **clean:** to cleans the project and remove all files generated by the previous build.

  - **site:** creates the project's site documentation.

- Each lifecycle contain a sequence of phases.
  The default lifecycle, has 23 phases, clean 3 has phases and site has 4 phases.

Phase, Goal and Plugin

- Maven has a sequence of **phases**, which specify the order of goal execution. In other words, each phase includes a sequence of **goals**.

- A **goal** represent a specific task which contributes to the building of the project. A single goal can bound to zero or more other goals (similar to ‘depends’ in ANT).

- A maven **plugin** is a group of goals, which are not necessary bounded together.

Maven Build Lifecycle

A typical Maven Build Lifecycle consists of the following sequence of phases:

prepare-resources: Resource copying can be customized here.
validate: Validates whether our project include all necessary information.
compile: Source code compilation will be done in this phase.
test: Testing the compiled source code.
package: It creates the JAR/WAR/EAR packages.
install: This phase installs the package in local or remote maven repository.
deploy: Copies the final package to the remote repository.

Phases

- ***validate:*** checks if all necessary information for the build are available.
- ***compile:*** compile our source code.
- ***test:*** run unit tests (e.g. Junit)
- ***package:*** packages compiled source code into the deployable (jar, war, …)
- ***install:*** installs the package to a local repository.
- ***deploy:*** copies the package to the remote repository.

```
mvn <PHASE>
```

Goals

- ***compiler:compile*** – the *compile* goal from the *compiler* plugin is bound to the *compile* phase.

- ***compiler:testCompile*** – it is bound to the *test-compile* phase.

- ***surefire:test*** – it is bound to *test* phase.

- ***install:install*** – it is bound to *install* phase.

- ***jar:jar***, ***war:war*** - they are bound to *package* phase.

```
mvn help:describe -Dcmd=compile
```

Plugins

Plugins are where much of the real action is performed, plugins are used to: create jar files, create war files, compile code, unit test code, create project documentation, and on and on. Almost any action that you can think of performing on a project is implemented as a Maven plugin. (https://maven.apache.org/guides)

- A plugin can be executed as follows: `mvn [plugin]:[goal]`

  e.g.
  ```
  mvn clean:clean
  mvn compiler:compile
  mvn compiler:testCompile
  ```

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-xx-plugin</artifactId>
            <version>${maven.xx.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>integration-test</goal>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Plugins

- A plugin
e.g

```
<build>
    <plu
    </pl
</build>
```

- Every task in Maven is executed by a plugin.
- A plugin can be executed as follows:

```
mvn [plugin]:[goal]
```

e.g. `mvn clean:clean`

`mvn compiler:compile`

`mvn compiler:testCompile`

- Every task in Maven is executed by a plugin.
- Each plugin can have more or more than one goals.
- First, maven will try to download the plugin from repository.
- A plugin can be executed as follows:

```
mvn [plugin]:[goal]
```

```
<project …”>
   <modelVersion>4.0.0</modelVersion>
   <groupId>edu.bu.met622</groupId>
   <artifactId>testproject</artifactId>
   <version>1.0</version>
   <build>
      <plugins>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.1</version>
            <executions>
               <execution>
                  <id>id.phaseone</id>
                  <phase>phaseone</phase>
                  <goals>
                     <goal>run</goal>
                  </goals>
                  <configuration>
                     <tasks>
                        <echo>Phase 1 executed</echo>
                     </tasks>
                  </configuration>
               </execution>
            </executions>
         </plugin>
      </plugins>
   </build>
</project>
```

https://www.tutorialspoint.com/maven/maven_plugins.htm

POM.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

Name: The name of our current project

Version: is the current version of our project

Packaging: How we deliver the project, e.g. JAR, WAR, EAR

Dependencies: What are the other libraries our projects use.

Plugins: Plugins are frameworks that designed to do a specific task, e.g. clean, install, deploy, jar, antrun, pdf, …

- The first thing we need to specify for maven, is the project group, i.e. **`groupid`**, and project name, i.e. **`artifactId`**.

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd”>

   <modelVersion>4.0.0</modelVersion>
   <groupId>edu.bu.pom-sample</groupId>
   <artifactId>sampleProject</artifactId>
   <version>1.0</version>
   <url>http://maven.apache.org</url>
   <dependencies>
     <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.11</version>
          <scope>test</scope>
     </dependency>
   </dependencies>
</project>
```

- **project** This is the top-level element in all Maven pom.xml files.

- **modelVersion** This element indicates what version of the object model this POM is using. The version of the model itself changes very infrequently but it is mandatory in order to ensure stability of use if and when the Maven developers deem it necessary to change the model.

- **groupId** This element indicates the **unique identifier of the organization or group that created the project**. The groupId is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example `org.apache.maven.plugins` is the designated groupId for all Maven plugins.

- **artifactId** This **primary artifact for a** of their final name. A typic example, `mya`

- **packaging** Th not only means if the a uild process. (The lifecycle i ging of a project can pla you do not have to specif

- **version** This e lp you with version management and you will often see the `SNAPSHOT` designator in a version, which indicates that a project is in a state of development. We will discuss the use of snapshots and how they work further on in this guide.

- **name** This element indicates the **display name used for the project**. This is often used in Maven's generated documentation.

- **url** This element indicates **where the project's site can be found**. This is often used in Maven's generated documentation.

- **description** This element provides a **basic description of your project**. This is often used in Maven's generated documentation.

Usually your IDE support in creating a pom.xml for your project, but it requires lots of manual configuration as well.

https://maven.apache.org/guides/getting-started/index.html

- **project** This is the top-level element in all Maven pom.xml files.

- **modelVersion** This element indicates what version of the object model this POM is using. The version of the model itself changes very infrequently but it is mandatory in order to ensure stability of use if and when the Maven developers deem it necessary to change the model.

- **groupId** This element indicates the **unique identifier of the organization or group that created the project**. The groupId is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example `org.apache.maven.plugins` is the designated groupId for all Maven plugins.

- **artifactId** This element indicates the unique base name of the primary artifact being generated by this project. **The primary artifact for a project is typically a JAR file.** Secondary artifacts like source bundles also use the artifactId as part of their final name. A typical artifact produced by Maven would have the form <artifactId>-<version>.<extension> (for example, `myapp-1.0.jar`).

- **packaging** This element indicates **the package type to be used by this artifact (e.g. JAR, WAR, EAR, etc.).** This not only means if the artifact produced is JAR, WAR, or EAR but can also indicate a specific lifecycle to use as part of the build process. (The lifecycle is a topic we will deal with further on in the guide. For now, just keep in mind that the indicated packaging of a project can play a part in customizing the build lifecycle.) The default value for the `packaging` element is JAR so you do not have to specify this for most projects.

- **version** This element indicates the **version of the artifact generated by the project**. Maven goes a long way to help you with version management and you will often see the `SNAPSHOT` designator in a version, which indicates that a project is in a state of development. We will discuss the use of snapshots and how they work further on in this guide.

- **name** This element indicates the **display name used for the project**. This is often used in Maven's generated documentation.

- **url** This element indicates **where the project's site can be found**. This is often used in Maven's generated documentation.

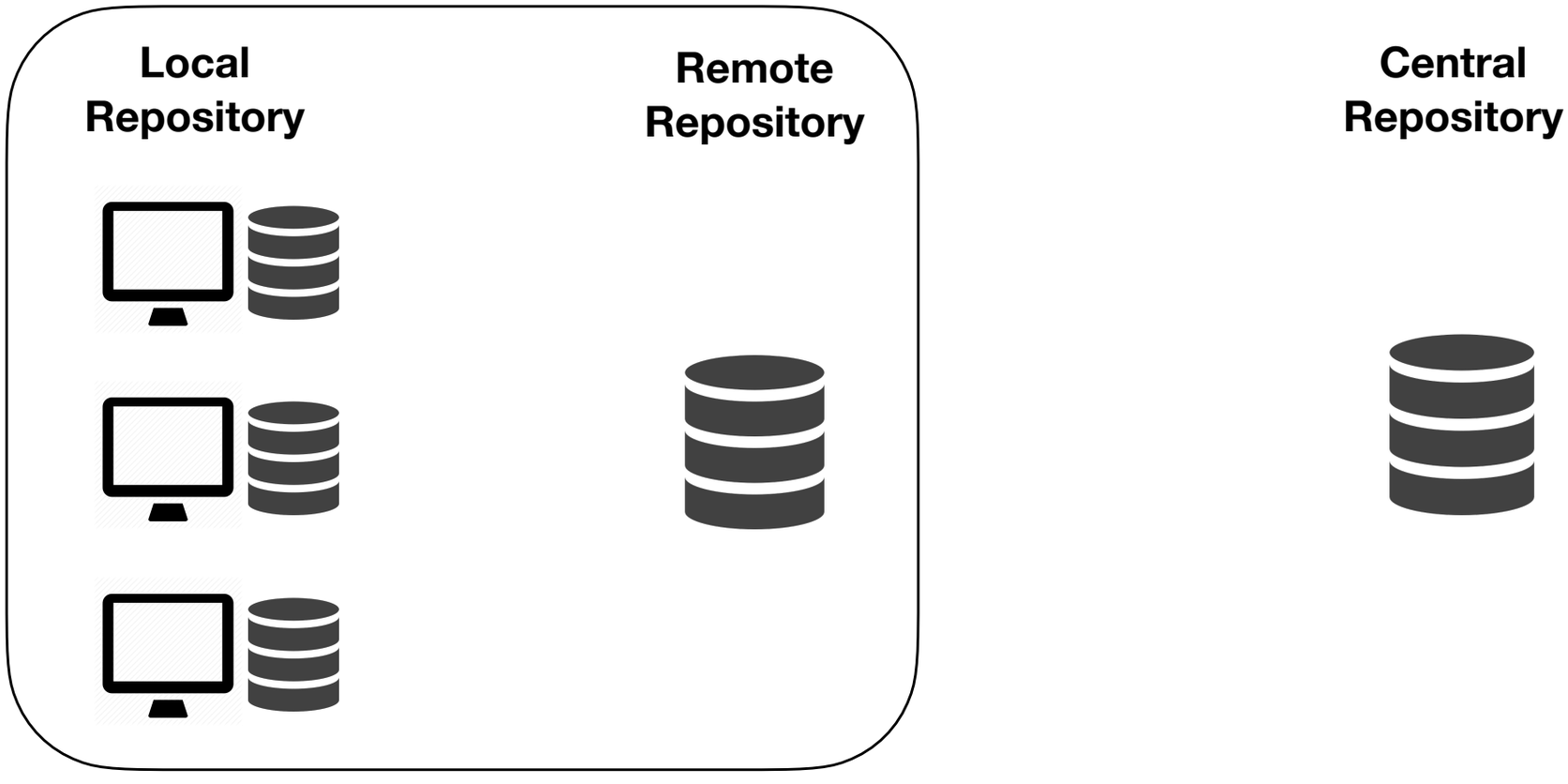
- **description** This element provides a **basic description of your project**. This is often used in Maven's generated documentation.

https://maven.apache.org/guides/getting-started/index.html

Local and Remote Repository

- A repository in Maven is a directory that host .jar files (libraries used in a project) and other project specific artifacts.

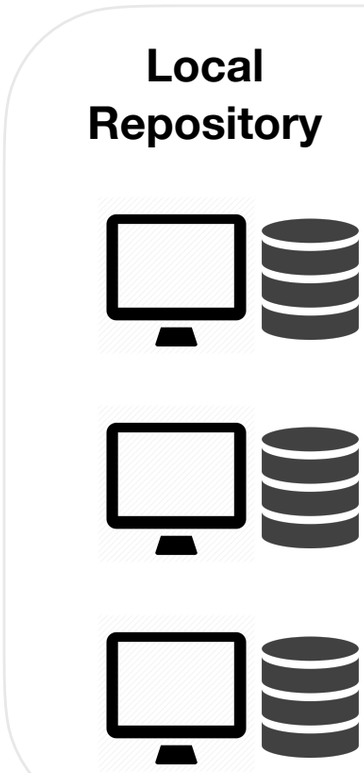
- There are three types of repositories supported by Maven, central, remote and local.



Local and Remote Repository

- A repository in Maven is a directory that host .jar files (libraries used in a project) and other project specific artifacts.

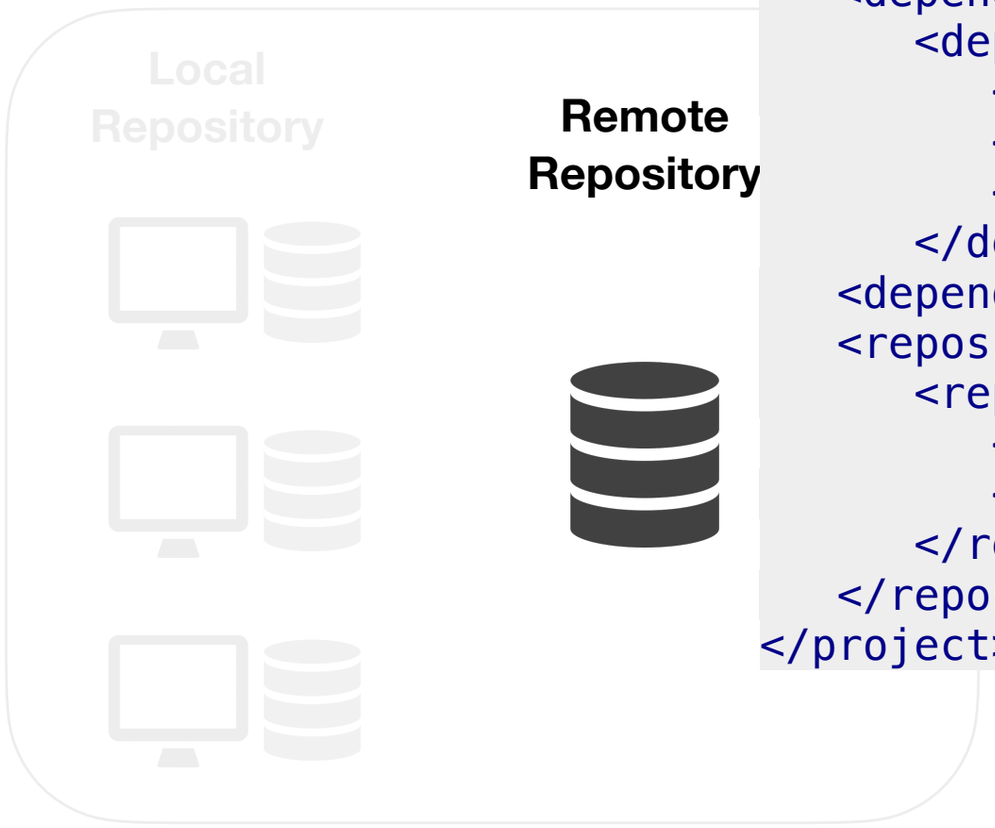
- There are three types of repositories supported by Maven, central, remote and local.




```
<settings xmlns = "http://maven.apache.org/SETTINGS/1.0.0"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://maven.apache.org/SETTINGS/1.0.0
   http://maven.apache.org/xsd/settings-1.0.0.xsd">
   <localRepository>C:/MVNRepository</localRepository>
</settings>
```

Local and Remote Repository

- A repository in Maven is a directory that host .jar files (libraries used in a p… …ate
- There are three type… …d by Maven, central, remote and …




```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>edu.bu.met622</groupId>
   <artifactId>project</artifactId>
   <version>1.0</version>
   <dependencies>
      <dependency>
         <groupId>edu.bu.met622.buildx</groupId>
         <artifactId>buildx</artifactId>
         <version>1.0.0</version>
      </dependency>
   <dependencies>
   <repositories>
      <repository>
         <id>edu.bu.met622.testx</id>
         <url>http://download.bu.edu/maven2/testx</url>
      </repository>
   </repositories>
</project>
```

Local and Remote Repository

- A repository in Maven is a directory that host .jar files (libraries used in a project) and other project specific artifacts.

- There are three types of repositories supported by Maven, central, remote and local.

https://repo1.maven.org/maven2

Central Repository

It is not required to be described inside pom.xml.

Maven Dependency Search

1. It searches the **local repository** for the required .jar file.

2. It searches the **central Maven repository** to find the jar file that did not existed in the local repository.

3. If there is no remote repository specified inside pom.xml maven stops processing and throws error.

4. If there is a **remote repository** defined and existed, maven searches that repo. for the missing file. If it does not existed there as well. Then, maven stops and throws error.

How to add a resource into a maven repository?

```
<project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>

</project>
```

How to deploy our .jar into a remote repository?

```
<project …>
  …
  <dependencies>
  …
  </dependencies>

  …

  <distributionManagement>
    <repository>
      <id>remote-repository</id>
      <name>Remote Repository</name>
      <url>scp://repository.foo.bar/repository/maven2</url>
    </repository>
  </distributionManagement>

</project>
```

Running Maven

`mvn [options] [<goal(s)>] [<phase(s)>]`

- `mvn package`
- `mvn clean deploy site-deploy`
- `mvn verify`
- `mvn clean install` <— *the most used command*

introduced in 2012

Gradle

It builds on top of ANT and Maven. Groovy or Kotlin languages used Domain Specific Language over the XML.

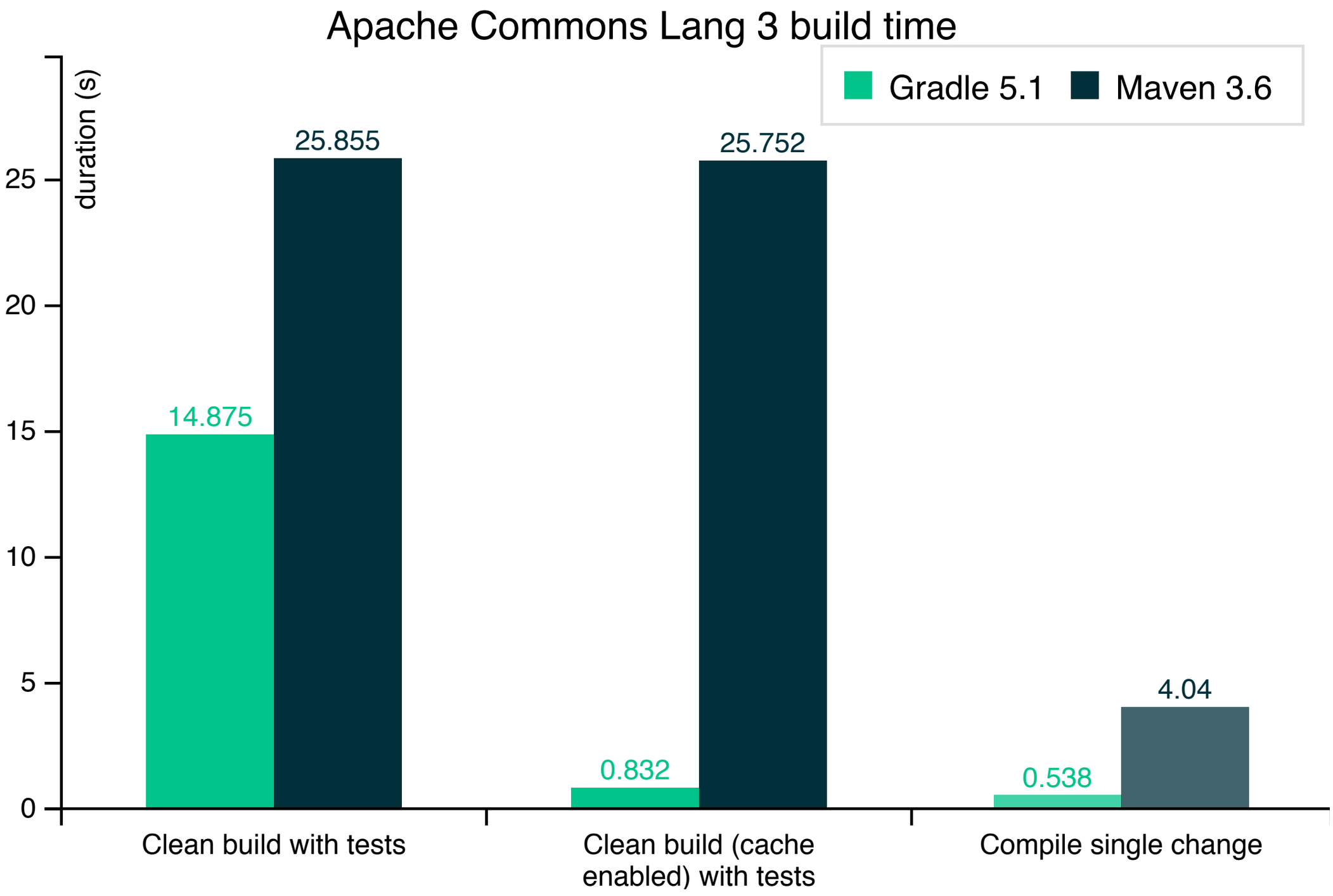
- The main drawback of XML build scripts is that they are **not appropriate for procedural programming**. By adding some programming modules into the build script. The XML file will get enormously big.

- Gradle is more flexible and enables us to **structure our build script**. Also it enables us using the native C++ inside the build script. Google has chosen Gradle for Android (previously it uses Ant).

- Gradle Wrapper allows us to **execute Gradle builds on machines where Gradle is not installed**. This is useful for continuous integration of servers.

- In large scale project, **build time** is important and Graddle is faster than other Java build tools.

Gradle Performance



Gradle Installation

1. Check Java version to be JDK 1.6 or newer version.

   ```
   java -version
   ```

2. download and run: `unzip -d /opt/gradle gradle-5.5-bin.zip`

3. OPTIONAL: `export GRADLE_PATH = /opt/gradle`

4. OPTIONAL: `export PATH=/opt/gradle/bin:$PATH`

Task and Project

- Gradle is working with two concepts, **task** and **project**.
- Every Gradle build presents one or more than one projects.
- Every project include at least one or more than one tasks.
- A task is a unit of work that the build process will perform.

Project Object and Script Object

- Gradle **task** is very similar to Ant target. Task is a piece of work that Gradle does.

- A Gradle script is working with two data objects: **Project object** and **Script object**.

  - **Project Object:** Each script describes about one or multiple projects. While in the execution this scripts configures the Project Object.

  - **Script Object:** It is the object that Gradle uses to execute script build codes. All the properties and methods declared by the script interface are available in your script.

Gradle Task

- A task is a unit of code which the build script will execute. For example, compiling a class, creating a jar, …

```
task simpleTask {
   doLast {
      println ‘a very simple task'
   }
}
```

- A task is written in **build.gradle**

- It is executed as follows:

```
C:\> gradle –q simpleTask
```

Hello world Example

- Hello world example

```
task hello {
   doLast {
      println ‘first gradle code'
   }
}
```

- Execute above task:

```
C:\> gradle –q hello
```

- save the content of the file in **build.gradle.**

- We can invoke the saved task in build.gradle as follows:

```
C:\> gradle –q hello
```

Adding Dependency into a Gradle Task

- Execution of TaskY depends on TaskX

```
task taskX << {
   println 'taskX'
}
task taskY(dependsOn: 'taskX') << {
   println "taskY"
}
```

- Execute above task:

```
C:\> gradle –q taskY
```

- The output will be as follows:

```
taskX
taskY
```

Building Java Application

make a demo folder.

```
$ mkdir demo
$ cd demo
```

Run Gradle init

```
$ /Users/rawassizadeh/WORK/gradle-5.5/bin/gradle init
```

Answer questions accordingly

```
$ gradle init

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: grade plugin
Enter selection (default: basic) [1..4] 2

Select build script DSL:
  1: C++
  2: Groovy
  3: Java
  4: kotlin
Enter selection (default: groovy) [1..4] 3

Select build script DSL:
  1: Groovy
  2: Kotlin
Enter selection (default: Groovy) [1..2]

Select test framework:
1: JUnit 4
2: TestNG
3: Spock
4: JUnit Jupiter (default: Junit 4): [1..4] 1

Source package (default: demo): src

Project name (default: demo): testmet622
Source package (default: testmet622): src
> Task :init Get more help with your project: https://docs.gradle.org/5.5/userguide/
tutorial_java_projects.html
BUILD SUCCESSFUL in 4m 33s 2 actionable tasks: 2 executed
```

Dependency Management

- Obviously each project has many dependencies to other libraries and jar files.
- Gradle enable us to copy dependencies locally, or upload them into a remote repository such as Maven repo. This process of publishing a task is **publication**.
- Example:

```
apply plugin: 'java'

repositories {
   mavenCentral()
}

dependencies {
   compile group: 'org.hibernate', name: 'hibernate-core',
version: '3.6.7.Final'
   testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

Dependency Configuration

There are four standards dependency command in Gradle. configuration.

- **Compile** : The dependencies required to compile the source codes.
- **Runtime** – The dependencies required by the bin files (classes) at runtime. By default, also includes the compile time dependencies.
- **Test Compile** – The dependencies required to compile the test source of the project. By default, it includes compiled production classes and the compile time dependencies.
- **Test Runtime** – The dependencies required to run the tests. By default, it includes runtime and test compile dependencies.

Dependency Management

- Obviously each project has many dependencies to other libraries and jar files.
- Gradle enables us to copy dependencies locally, or upload them into a remote repository such as Maven repo. This process of publishing a task is **publication**.
- Example:

```
apply plugin: 'java'

repositories {
   mavenCentral()
}

dependencies {
   compile group: 'org.hibernate', name: 'hibernate-core',
version: '3.6.7.Final'
   testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

External repositories

```
repositories {
   maven {
      url "http://repo.mycompany.com/maven2"
   }
}
```

Publishing with Gradle

- Published files in Gradle are called **artifacts**.

- Gradle enable us to copy dependencies locally, or upload them into a remote repository such as Maven repo. This process of publishing a task is **publication**.

- Example:

```
apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

Android build.gradle

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 23
    buildToolsVersion '23.0.1'

    defaultConfig {
        applicationId "com.insight.insight"
        minSdkVersion 19
        targetSdkVersion 23
        versionCode 25
        versionName "1.0.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                                                 'proguard-rules.pro'
        }
    }
}

dependencies {
    wearApp project(':wear')
    compile fileTree(include: ['*.jar'], dir: 'libs')
    compile 'com.google.android.gms:play-services:6.5.87'
    compile 'org.jsoup:jsoup:1.7.2'
    compile 'com.android.support:appcompat-v7:23.1.0'
}
```

Build Tools

- Apache ANT
- Apache Maven
- Gradle
- **Continuous Integration**

Continuous Integration

- A development team is composed of several developers, which are most of the times located remotely and not in the same location.

- Continuous Integration is a set of practices that when ever there is a small changes in the code. The entire system will be built from the code repository, tested and deployed.

- This process should be done frequently (e.g. nightly builds) and automatically.

- CI tools are used to automate, building, testing and deploying the changes into the operational application server.

Continuous Integration, Delivery and Deployment

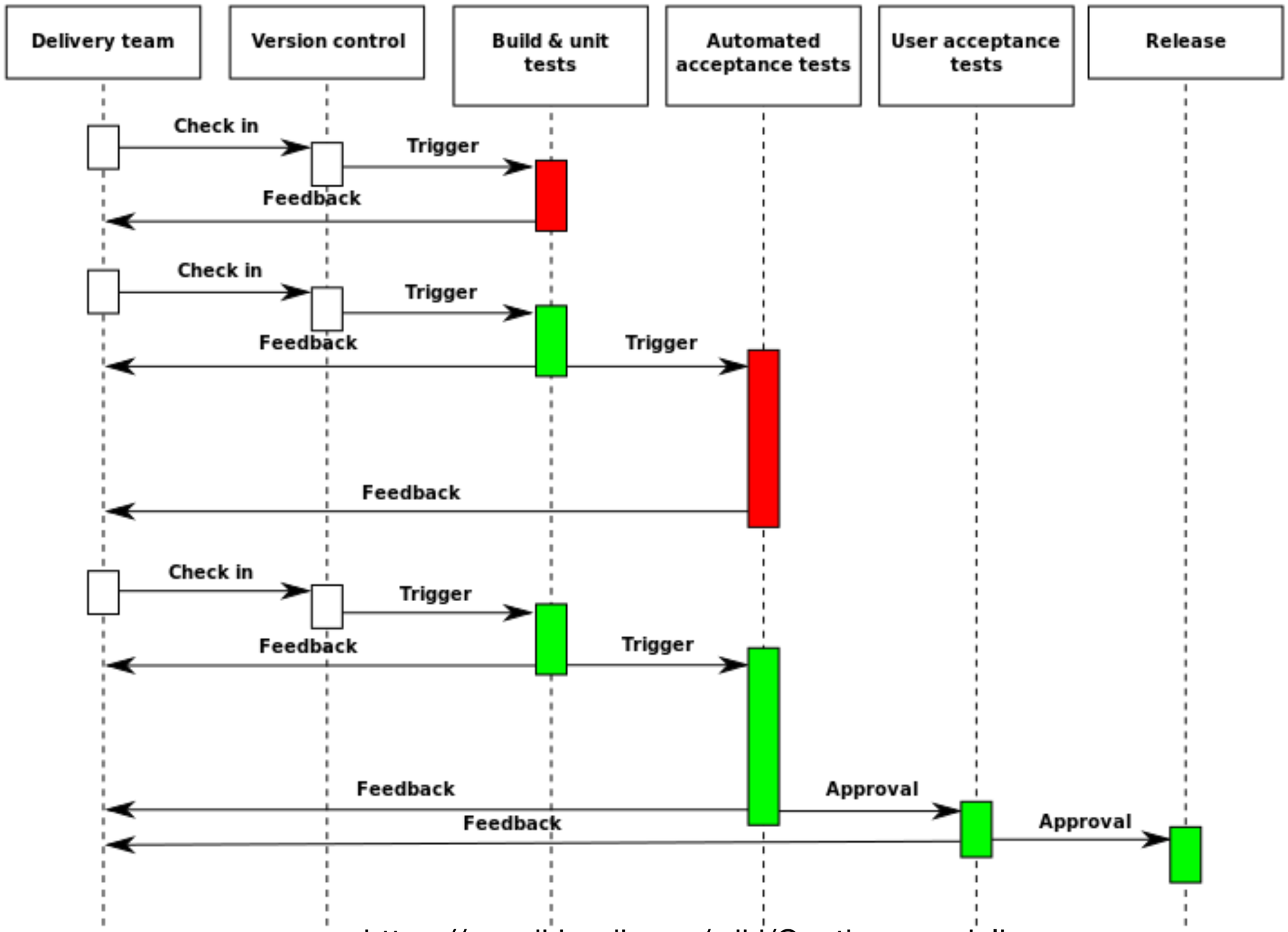
- Continuous Integration are tools that enable developers to **frequently integrate their code changes into their product**.

- In particular CI tools are responsible to **check** and **validate** new changes developers perform on their code.

- CI tools are automating the process of building, packaging, and testing the application

- Continuous delivery and deployment is similar to the CI, but they are used for continuous delivery and deployment of the application.

Continuous Delivery



source: https://en.wikipedia.org/wiki/Continuous_delivery

Continuous Integration Tools

Jenkins CI is the most used continuous integration server. Built with Java, it provides over 300 plugins to support building and testing virtually any project.

Jenkins

https://jenkins.io

It is a newer version of CI, easy to use and setup. It enables all configurations in code (not in GUI like Jenkins).

Concurse

https://concourse-ci.org/

References

- tutorialspoint.com
- Ant homepage: https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html
- Maven homepage: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html
- Gradle homepage: https://guides.gradle.org/building-java-applications/