**Build tools**

Build tools are commonly known as programs that automate the process of building an executable application from source code. This building process includes activities like compiling, linking and packaging the code into an executable form

**The activities include:**

* Downloading the dependencies.
* Compiling source code to the form of binary code.
* Packaging that binary code.
* Running the tests.
* Deploying them to the production systems.

**Types:**

* Scala oriented Build Tool (SBT):
* CMake:
* Terraform:
* Apache Maven:

**Software Development Kit:**

In order to use an SDK, a developer needs an iOS SDK to build iOS apps, an Android toolkit for building Android apps,

SDKs include documentation, application programming interfaces (APIs), code samples, libraries and processes, as well as guides that developers can use and integrate into their apps. Developers can use SDKs to build and maintain applications without having to write everything from scratch.

**Difference between SDKs and APIs**

An API is code that enables two software programs to communicate with each other. An API defines the correct way for a developer to request services from an operating system or other application and expose data within different contexts and across multiple channels.

When a developer uses an SDK to develop applications and create systems, those apps have to communicate with other applications. An SDK includes an API to enable that communication.

**Compilation:**

The compilation is **a process of converting the source code into object code(machine readable format)**. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

**What are Dependencies**

Dependency is a broad software engineering term used to refer when a piece of software relies on another one. Simply put, if Program A requires Program B to be able to run, Program A is dependent on Program B. This makes Program B a dependency of Program A.

It doesn’t really matter what it is, if your program needs to run correctly, it’s a dependency. Common examples of dependencies are programming libraries, Online services, programming scripts etc.

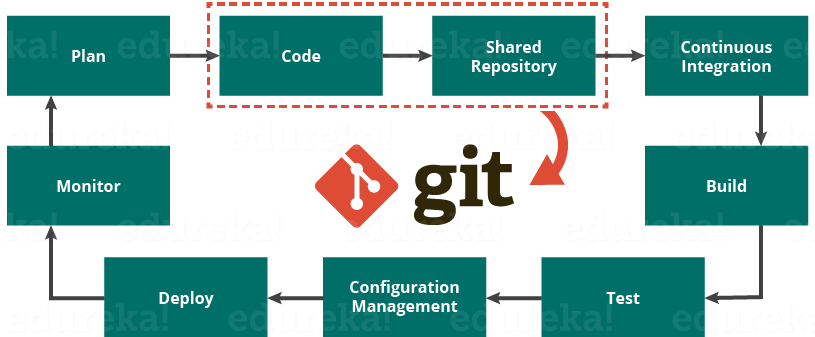
**Artifact**

**Artifact** is highly associated and related to specific methods or processes of development. Methods or processes can be project plans, business cases, or risk assessments. Distinct gathering and collections of detailed information are generally organized and incorporated into artifact sets. A set generally represents complete aspect of system. This is simply done to make development and establishment of complete software system in manageable manner. An artifact is one of many kinds of tangible by-products produced during the development of software.

**DevOps pipeline**

A DevOps pipeline is a set of automated processes and tools that allows both developers and operations professionals to work cohesively to build and deploy code to a production environment. While a DevOps pipeline can differ by organization, it typically includes build automation/continuous integration, automation testing, validation, and reporting. It may also include one or more manual gates that require human intervention before code is allowed to proceed

**Software development process:**



* DevOps trying to automate it with least human intervention
* Multiple developers on multiple platforms
* From repository and integration, DevOps come into picture
* Build process starts through build tools

**Build tools:**

1. C – make
2. Java – Maven

**MAKE:**

* We define makefile.
* It has set of instructions to read the code and to compile for C or C++
* Every piece of code contains a main code and dependencies. How these two interact has to be defined in makefile.

**Dependencies:**

* A piece of code interacts with another piece of code or application, to get desired result.
* Dependency file:- **<main.o>**
* Just in the form of functions, of what code has to interact with which dependency.
* Why? The easier the code, the easier is the integration, build process, development. We can call for a function but not a code.
* At the end we get a **<.exe>** file as output file. Final result.

Code from different languages cannot be merged. But the functionalities of the codes can be merged

**Artifactory** –

* all the necessary files needed for compilation and mapped in **<main.o>** file so execution **<.exe>** file contains binary which has makefile instructions and **<main.o>** file.
* Artifactory can be used to store dependency files in cloud available for everyone, API calls (URL of file location)

**Ex:**

While <.c> file is given, it should be as a function. That is, void main () should be changed to void fact()

Three codes: big3.c fact.c fib.c

**main.c**

#include <stdio.h>

main() {

fact();

fib();

big3();

}

the order in which it is compiled in is according to <main.o> file.

**makefile:**

gcc – compiler for c

-o –compiles object file

-c – compiles c files

**makefile**

**makefile**

Final.exe:main.o big3.o fact.o fib.o

gcc -o Final.exe:main.o big3.o fact.o fib.o

main.o:main.c

gcc -c main.c

fact.o:fact.c

gcc -c fact.c

big3.o:big3.c

gcc -c big3.c

fib.o:fib.c

gcc -c fib.c

clean:

rm -rf \*.o

here in the first step as final.exe doesn’t exist,

so we have to give final.exe in the second line

to the compiler

(gcc -o Final.exe:main.o big3.o fact.o)

* **sudo yum install gcc –y**

Commands for make file:

* **make**
* **make clean**

Should be saved as makefile.

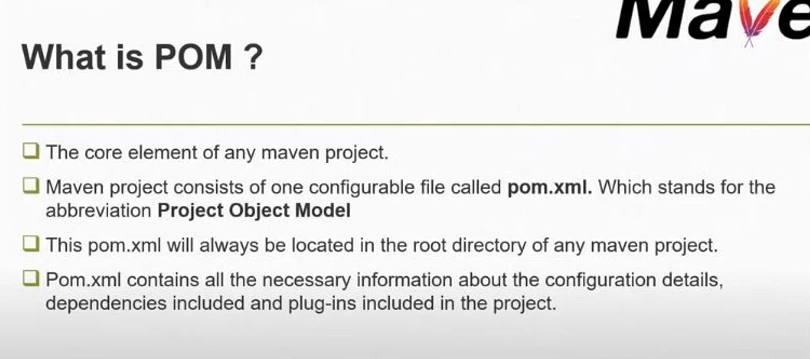
If it is not saved as makefile, then

* **make –f <name>**
* **make clean –f <name>** 🡪 for deleting .o files
* The order in which it is compiled in is according to <main.c> file.
* If there are several functions, we can create one (library) resource () {a1 a2 a3} in <main.c> and give all the functions necessary in the resource file.
* Indentation is very important in makefile, else it throws up an error.

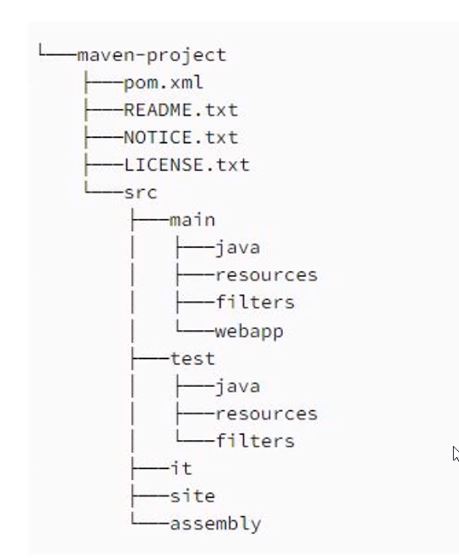
**Assignment:**

* Object file vs Makefile vs C file and other files
* Different strategies in writing makefile.

**MAVEN**



* Maven as a build tool to manage or compile java projects
* Earlier it was hosted on **maven-apache**
* Now C-shark, ruby, scala languages code can also be compiled in maven.
* In **make** 🡪 dependency file is **object file <main.o>** in which all the libraries, dependencies are mentioned or defined
* In **maven** 🡪 <**pom.xml>** is the dependency



**Maven folder**

What does <**pom.xml**> has?

1. Dependencies/library
2. Plugins
3. Project versions
4. Mailing list
5. Repository details
6. Build profiles

Generally, java scripting is used where web applications are to be done. We need a server to deploy the end compiled files. In **make** it gives executable file in the location itself, Linux server itself. But when it is **maven**, it is web application for internet. We have to deploy them to web server like **apache tomcat**. For example, in tomcat installation 🡪 **webapps folder** webapps - all our compiled files will come and sit here.

* In **8080 port 🡪** 🡪 webapps 🡪 here we deploy the artifact/executable files
* for maven, final output is **.jar, .war, .ear** also known as **artifacts** used for other environments.
* Artifacts are not executable files. Unpack to achieve functionality. This cannot be used on interface or cli
* <.jar> is like a zip file which has code and dependencies together. When we unpack,
* Then it can be used for creating website interface or its components (departments, components, etc.)
* Java is very dynamic language 🡪 it has many libraries/frameworks/network-protocols available in the internet
* Java can integrate with many frameworks for getting the desired outputs
* Whenever the code is executed, the compiler goes to the library or framework and gets it to work area or environment to give the final result(.jar, .war, .ear).
* Maven is not just a compiling tool, but also sometimes it deploys the compiled files
* If there is a build failure in maven project because of a discrepancy in pom.xml, we should know what the issue is by looking at it.
* The dependency/library files will not be present in pom.xml 🡪 but the link to those dependency/libraries are present in pom.xml 🡪 hence will be downloaded

**Maven build cycle:**

8 stages:

1. **Validate** - check if the project is correct. If the dependencies, necessary details, libraries are mentioned in pom.xml otherwise throws up an error
2. **Compile** - compiles source code line by line and start generating binary
3. **Test** – this is testing of the compiled source code in the suitable framework as defined in pom.xml, done by maven itself
4. **Package** - made into distributable packages like .jar, .war, .ear (most commonly used as they can be unpacked in any env)
5. **Integration-testing** - prototype step - ex: if it has micro services: it gives different jar. All these jars are brought together integrated and tested. in above testing, it is tested individually
6. **Verify** – verifies the compiled package and checks if it meets the quality criteria that are defined in pom.xml
7. **Install** - if local repository is defined, goes to pom.xml, use all the dependencies and installs the package.
8. **Deploy** – wherever the final package is created, it can go and get deployed in the desired environment. All developers, testers can access all these.

* **Microservice**? 🡪 In an application (Menu is one micro service, checkout is one micro service.)
* When there is a build failure at any step, all the previous steps will be cached and for debugging we can selectively start from any step. Yet, all the steps are run in the background.
* Before maven, there used to be ANT. ANT's evolved version is maven
  + Ant was slow,
  + long scripting,
  + no pom.xml architecture,
  + tough to debug

**Maven Commands:**

* **sudo yum install maven -y**
* **sudo yum install java -y**
* **mvn clean install 🡪** Cleans the project and removes all files generated by the previous build and deploys the packaged JAR/ WAR file to the local repository.
* **mvn compile 🡪** Compiles source code of the project.
* **mvn test-compile 🡪** Compiles the test source code.
* **mvn test 🡪** Runs tests for the project.
* **mvn package 🡪** Creates JAR or WAR file for the project to convert it into a distributable format.
* **mvn install 🡪** Deploys the packaged JAR/ WAR file to the local repository.
* **mvn deploy 🡪** Copies the packaged JAR/ WAR file to the remote repository after compiling, running tests and building the project.

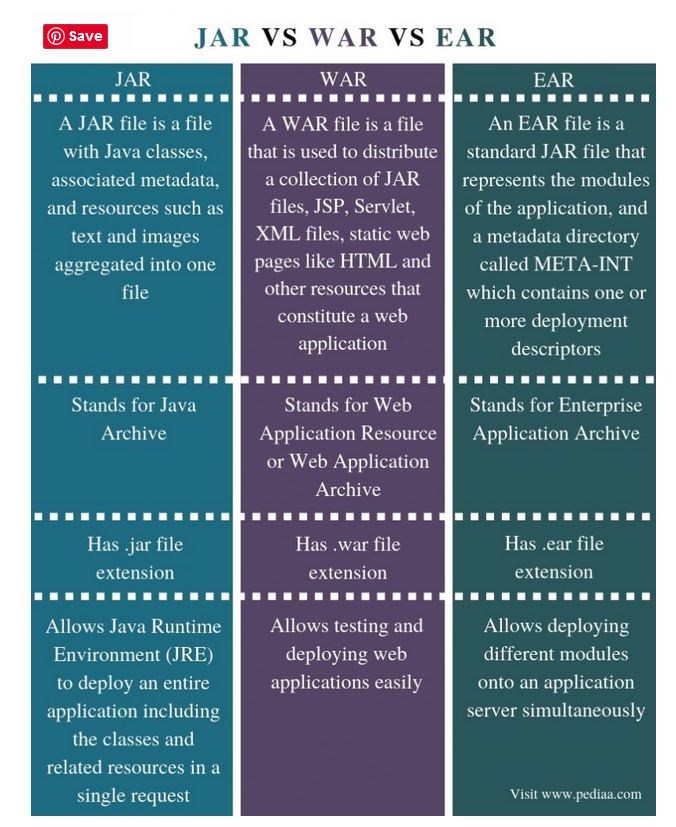
**Maven Repository**

A maven repository is a directory of packaged JAR file with pom.xml file. Maven searches for dependencies in the repositories.

1. Local Repository
2. Central Repository
3. Remote Repository

* Maven **local repository** is located in your local system. It is created by the maven when you run any maven command. We can change the location of maven local repository by changing the **settings.xml** file.
* Maven **central repository** is located on the web. It has been created by the apache maven community itself.
* Maven **remote repository** is located on the web. Most of libraries can be missing from the central repository such as JBoss library etc, so we need to define remote repository in pom.xml file.

<https://www.tutorialspoint.com/maven/maven_repositories.htm>



**Branching strategy:**

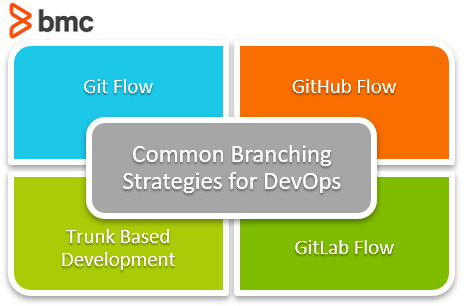
[**https://www.bmc.com/blogs/devops-branching-strategies/**](https://www.bmc.com/blogs/devops-branching-strategies/)

Branching strategy is something a software development team **uses when interacting with a version control system for writing and managing code**. branching strategy focuses on **how branches are used in the development process.**

It helps define how the delivery team functions and **how each feature, improvement, or bug fix is handled**. It also reduces the complexity of the delivery pipeline by allowing developers to focus on developments and deployments only on the relevant branches—without affecting the entire product.

**Selecting branching strategy**

Factors like the development method, scale, user preferences highly impact this selection. Additionally, other factors like CI/CD (Continuous Integration and Continuous Delivery) tools decide what branching strategies can be used in your DevOps pipeline.



SCM- GIT

we have many branches ,which are used to push the Source code.

**Branching Strategy –**

[**https://medium.com/@jbradyaudio/streamline-a-super-efficient-branching-and-ci-strategy-ffa864aa99d4**](https://medium.com/@jbradyaudio/streamline-a-super-efficient-branching-and-ci-strategy-ffa864aa99d4)

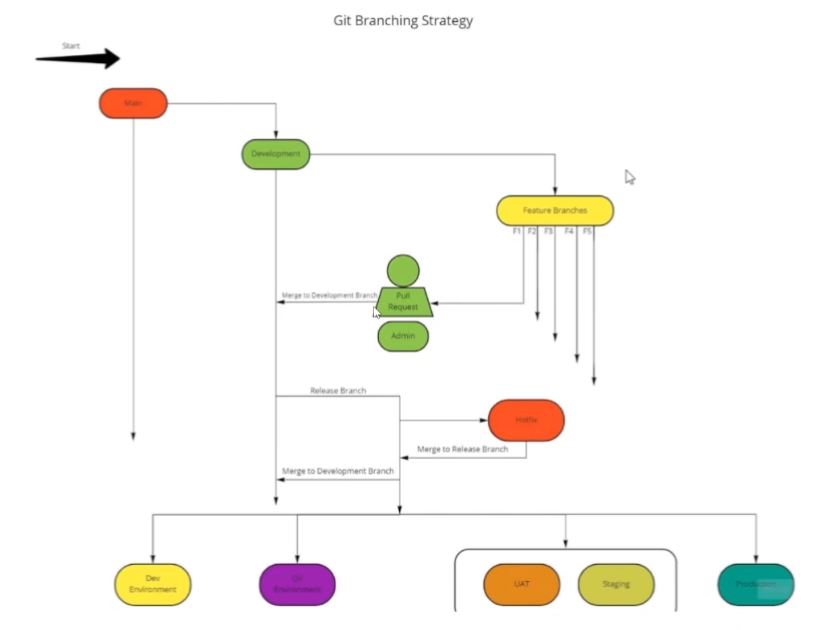
* Practice used in software development process that can be integrated with pipeline tool
* The branching strategy focuses on how branches are used in the development process to enable a collaborative development environment without affecting the code.
* Aiding in end to end automation.

Main Branches:

* **Master branch** - The primary branch where all the production code is stored. Once the code in the “develop” branch is ready to be released, the changes are merged to the master branch and used in the deployment.
* **Develop branch** - This is where all the actual development happens. All the pre-production code is stored here, and the completed code of all the supporting branches is merged directly to the develop branch.

Support Branches:

* **Feature branch** - feature branches are used to develop new features and branches off exclusively from the develop branch.
* **Release branch** - This branch is used to aggregate fixes and improvements and prepare for the production release. It will be branched from the develop branch and merged to both develop and master.
* **Hotfix branch** - This is to deal with production issues where quick fixes are required. They can branch off from the master itself, but need to be merged to both master and develop branches.



**Branching Strategy steps:**

* designed and interlinked in such a way that the progress is easier. depending on nature of code we design this
* we will have a branching strategy
* any repository in a git has a master branch. whenever a commit is made, a checkpoint is reached.
* as time passes, commits added, a time line is created
* we have a master branch. from this, we create dev branch. from now, the code that was there in master branch will be there in dev branch.
* it is not that we cannot work on dev branch.
* from dev branch feature branches are created, the code in dev branch are still available on feature branches
* once the work is done in feature branch, they initiate a pull request to be merged in dev branch. according to admin (DevOps), if it is okay, merges it with dev branch
* if feature is successful, it becomes a checkpoint.
* then in a sprint, it is ready to deploy. the code is in dev branch, we create another branch called release branch, i.e., after building we make a binary out of it and push to different testing environments
* after merging if there is failure in build, we create hotfix branch, push the code to that branch and here we try to resolve the issue, merge it to the main code using pull request.
* then again it goes to testing environment. when test cases are successful, we push it to the dev branch.
* hotfix branch can be in the middle if the issue is identified and resolved there itself, no need to go through entire process.

**Testing environment:**

A testing environment is a setup of software and hardware for the testing teams to execute test cases

**-Dev**

* **-QA -quality analysis.**
* **SIT-System integration test.**
  + Type of software testing carried out in an integrated hardware and software environment to verify the behaviour of the complete system.
  + It is conducted on a complete, integrated system to evaluate the system’s compliance with its specified requirement.

**UAT-User acceptance Testing.**

* It is a type of testing in a controlled environment performed by the end user or the client to verify/accept the software system before moving the software application to the production environment.
* **Staging:**
* It is a replica of a production environment for software testing.
* Staging environments are made to test codes, builds, and updates to ensure quality under a production-like environment before application deployment

**-Prod:**

* **-PTE(pre-prod)-** This is done before the applic