Advanced Software Development: Continuous Integration / Development

CPSC 501: Advanced Programming Techniques
Fall 2022

Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Friday, September 2, 2022

UNIVERSITY OF CALGARY

Makefiles

To make or not to make

Makefiles

- **make** is an automation tool found in unix type operating system (and others) like Linux
- Since 1976. Designed around c and not Java.
- As software grew from single file programs to multiple files it became common that program could be compiled incorrectly (old compiled code vs newly fixed code)
- Makefiles are a standardization of compile scripts into standardized text file containing series of commands
- One key feature in a makefile is that it is designed to track dependencies of files so that one command can track by through source and compile and connect everything that is necessary in the right order

Makefile - example

```
all: helloworld

helloworld: helloworld.o
        # Commands start with TAB not spaces
        $(CC) $(LDFLAGS) -o $@ $^

helloworld.o: helloworld.c
        $(CC) $(CFLAGS) -c -o $@ $<

clean:
        rm -f helloworld helloworld.o
```

make all

- Will follow all -> helloworld -> helloworld.o -> helloworld.c
- Where it will compile it from .c to .o
- Then it will link .o to make binary
- Dependencies are dealt with before own command

make clean

- Will remove binary and .o, then follow

Make

- Basic idea of make commands in makefile is to be a one stop shop from someone downloading your codebase
- The user downloads your source, looks at readme, picks **target** make command with options relevant to their desired compiled result
- make **target**
- Many unix tools/programs are distributed or accessible as source (and not as a binary). When you download them they expect you to make the binary yourself.
  - **make** is not in windows (cygwin will get you access to it)

First came ANT

What is this a build automation tool for ants!

Apache ANT

- Another Neat Tool (ANT)
- Originally Apache Tomacat project
- Ant is not limited to Java but found its first big usage there as Makefiles didn’t really fit Java (c more naturally had .o dependencies that needed linking, Java compiles differently)
- A lot similar to makefiles, such that simple build.xml configs were easy
- From the era of structure files (xml), rather than relying on self managed structure like makefiles

Build.xml

```
<project>
    <target name="clean">
        <delete dir="classes" />
    </target>

    <target name="compile" depends="clean">
        <mkdir dir="classes" />
        <javac srcdir="src" destdir="classes" />
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="jar" />
        <jar destfile="jar/HelloWorld.jar" basedir="classes">
            <manifest>
                <attribute name="Main-Class"
                    value="antExample.HelloWorld" />
            </manifest>
        </jar>
    </target>

    <target name="run" depends="jar">
        <java jar="jar/HelloWorld.jar" fork="true" />
    </target>
</project>
```

Four different commands here

1. Setup **clean** (remove .class files)
2. Where to find src .java files and where to put .class files on **compile** command
3. Configuration of packing command to make a **jar**
4. Configuration for out to **run** the program

ANT

- ANT is flexible (like makefiles)
- But this means project coders can make really large and complex files which are hard to maintain
- No original built in dependency support (was added later)
- This original frustration drove Apache Maven

Then came Maven

What is this a build automation tool for mavens?

Apache Maven

- Like ANT it uses XML files for structure
- Added a lot of regularized structure seen in most project configurations so that certain commands could be automated
- **pom.xml**
- **mvn compile**

- Unlike ANT a standard directory structure developed
- Rather inflexible than Ant
- Great for quick projects
- Insufficient for complex

```
+---src
|   +---main
|   |   +---java
|   |   |   \---com
|   |   |       \---baeldung
|   |   |           \---maven
|   |   |                   HelloWorld.java
|   |   |
|   |   \---resources
|   \---test
|       +---java
|       \---resources
```

Apache Maven

```
<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>baeldung</groupId>
    <artifactId>mavenExample</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <description>Maven example</description>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

- A lot is direct encoded in xml
- Junit config is expected parts
- Name of jar and versioning
- Etc.

Now Gradle

What is this a build automation tool for gradles?

Gradle

- No XML (follows a recent trend away from strict structured files)
- Actually designed around specific language for
- Gradle needs user to add plugins to gain most features, i.e. regular java actions need a Java plugin
- Maven has most of market share but lots of growing support for Gradle (partially as it is less language specific)


```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    baseName = 'gradleExample'
    version = '0.0.1-SNAPSHOT'
}

dependencies {
    testImplementation 'junit:junit:4.12'
}
```

Continuous Integration (CI)

Continuous Integration

- JUnit testing enforces the idea that as a user makes changes to their changing part of codebase, that they run Unit Tests to ensure it stays correct
- This leads to a development process where a change is made and then immediately tested against existing unit tests
- The addition of version control leads to an environment where users are making changes on local repositories and/or branches (such as bugfixes) and desired to merge them back into a branch for the coming release
- Continuous integration is this process where users are continuously pushing in code and we design automated tests to build and verify the code each time
- Think of unit tests (and others) that run against codebase after a merge request is made

Continuous Integration

- Continuous integration can be done on your own systems, but is also available through web hosting services (share with your worldwide git project)
- Gitlab/Github: when project contributor push changes to remote, or commits to remote, a '**pipeline'** can be triggered
- Gitlab -> **.gitlab-ci.yml** file
  - Tells gitlab what commands to run (can also configure what triggers them)
  - Ex. On a commit, run static analysis, compile code, run unit tests, compile and package into a binary release file

CI Maven

```
#Pipeline stages
stages:
  - Build
  - Test
  - Package


#Compile program based on pom.xml
build:
  stage: Build
  script:
    - cd projectname
    - mvn compile
  tags:
    - cpsc501
```

```
#Test program based on pom.xml
test:
  stage: Test
  script:
    - cd projectname
    - mvn test
  tags:
    - cpsc501

#Package program based on pom.xml
package:
  stage: Package
  script:
    - cd projectname
    - mvn package
    - cd ..
    - cp projectname/target/projectname-1.0-SNAPSHOT.jar projectname-1.0-SNAPSHOT.jar
  artifacts:
    paths:
      - projectname-1.0-SNAPSHOT.jar
  tags:
    - cpsc501
  only:
    - master
```

CI Maven

```
#Pipeline stages
stages:
  - Build
  - Test
  - Package
```

Three pipeline stages
If one fails we don't move on

```
#Compile program based on pom.xml
build:
  stage: Build
  script:
    - cd projectname
    - mvn compile
  tags:
    - cpsc501
```

```
#Test program based on pom.xml
test:
  stage: Test
  script:
    - cd projectname
    - mvn test
  tags:
    - cpsc501
```

```
#Package program based on pom.xml
package:
  stage: Package
  script:
    - cd projectname
    - mvn package
    - cd ..
    - cp projectname/target/projectname-1.0-SNAPSHOT.jar projectname-1.0-SNAPSHOT.jar
  artifacts:
    paths:
      - projectname-1.0-SNAPSHOT.jar
  tags:
    - cpsc501
  only:
    - master
```

Leads to Continuous Deployment/Delivery (CD)

Continuous Deployment/Delivery

- Gitlab -> **.gitlab-ci.yml** file
  - Tells gitlab what commands to run (can also configure what triggers them)
  - Ex. On a commit, run static analysis, compile code, run unit tests, compile and **package into a binary release file**
- You can declare that certain finalized **artifacts** like a binary release file are created (or like a jar file)
- Continuous deployment is to make these available to manually be installed into production setups (if you've seen a github project that you can download a file from after each commit)
- Continuous delivery is to distribute these so that they are integrated into production systems

CI/CD

CI/CD by yourself

- You need to connect the repository (passive storage) to an active system which will run your commands
- In your regular dev environment you are used to running commands (like compile and unit testing yourself)
- Now you want the repository (often remote) to do it by itself
- Example for class: gitlab.ucalgary.ca
  - Gitlab uses program called **gitlab-runner**
  - Put it onto a machine, register it with repo, when repo has a commit it looks at .gitlab-ci.yml and pushes commands triggered to registered **gitlab-runner**
  - The gitlab-runner will received these **jobs** and execute
  - In general a job is a sequence of commands (ex. javac *.java to compile)

Docker

- Docker is a service/tool that hosts and provides virtual container configurations
- Example a linux system with latest version of python 3 installed
- You can make a CI file that will load a docker config, then run commands
- Allows you to make compile setups for multiple environments (load docker for windows and compile, load docker for linux red hat and compile, load osx docker, etc.)
- **Challenge (docker files aren't small files)**

- Docker often integrated into .gitlab-ci.yml files to indicate container environments used for pipeline steps.

Docker - CI

- Template config file to help gitlab-runner (**test-config.template.toml)**

[[runners]]

[runners.docker]

[[runners.docker.services]]

name = "postgres:latest"

[[runners.docker.services]]

name = "mysql:latest“

Declares two available docker services mysql and postgres databases and what images to use

Docker - CI

```
gitlab-runner register \
  --url "https://gitlab.example.com/" \
  --registration-token "PROJECT_REGISTRATION_TOKEN" \
  --description "docker-ruby:2.6" \
  --executor "docker" \
  --template-config /tmp/test-config.template.toml \
  --docker-image ruby:2.6
```

Setup gitlab-runner to use ruby image with the two sql services available during build

Images will be pulled from DockerHub

Docker - CI

Now you edit your gitlab-ci.yml file to add

image: ruby:2.6

To (or other name) to each pipeline stage that needs it

services:

- postgres:latest

Allows your to indicate which of the template images you want to use to provide the database, in this case postgres

Kubernetes

- Open source automated tool for managing containers (like docker)
- If you have an account you essentially request environments to be setup in their cloud environment to do your compile operations
- Like most cloud services it claims to scale to your needs as you add compile environments, complexity of actions, or customizations to the process

- Tools like gitlab will include ability that you can connect to your Kubernetes config so that you can install a gitlab-runner in their managed environment vs having to configure your own

Onward to … reflection.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY