Build systems

Build automation is the process of automating the creation of a software build and the associated processes including:

- **compiling** computer source code into binary code,
- **packaging** binary code,
- running automated **tests**.

https://www.slideshare.net/yandex/ss-42642539?qid=1c67a684-2636-43b6-bfe7-1371a1d49875&v=&b=&from_search=5

Without build automation

```
$ javac MyClass.java
```

Disadvantages:

- difficult to work with large amount of files,
- platform dependency,
- absence of logical connections,
- …

Shell-script

```
if test ! -e .nugget; then
     mkdir .nugget
     cp $cachedir /nugget.exe
fi
```

Advantages:

- possibility to run a couple of commands,
- provide primitive logical connections,
- we can divide by stages: **clean.sh**, **compile.sh**, **test.sh**

Disadvantages:

- platform dependency,
- no single approach for every project.

make (1977)

```
$ cat Makefile

.PHONY: all clean install uninstall
all: hello
clean:
                        rm -rf hello *.o
main.o: main.c
                        gcc -c -o main.o main.c
hello.o: hello.c
                        gcc -c -o hello.o hello.c
hello: main.o hello.o
                        gcc -o hello main.o hello.o
install:
                        install ./hello /usr/local/bin
uninstall:
                        rm -rf /usr/local/bin/hello

$ make clean

$make
```

make (1977)

Advantages:

- Single build process description

Disadvantages:

- platform dependency,
- no support of Java tasks, parameters, plugins.

Apache Ant (2000)

```
$ cat build.xml

<project>
    <target name="clean">
        <delete dir="build"/>
    </target>

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

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

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

$ ant compile jar run
```

Apache Ant (2000)

<APACHE ANT>

Advantages:

- Support of Java-specific tasks
- Platform independency
- Extensible (plugins)
- Able to build with parameters

Disadvantages:

- No strict code versioning convention
- No strict convention on code layout
- No automatic dependency management (/lib folder)
- Doesn't support JUnit4

Apache Ivy (2004)

```
$ cat ivy.xml

<ivy-module version=“1.0”>
    <info organization=“ru.yandex.qatools.allure” module=“allure-testing-ant”/>
        <dependencies>
                 <dependency org=“ru.yandex.qatools.allure“ name=“allure-testing-
adaptor” rev=“1.4.0”/>
                 <dependency org=“org.aspectj“ name=“aspectjweaver” rev=“1.7.4”/>
         </dependencies>
</ivy-module>

$ ant
```

Advantages:

- + Automatic dependency management

Apache Maven (2004)

```
$ cat pom.xml

<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.0</version>
        <configuration>
                <source>${compiler.version}</source>
                <target>${compiler.version}</target>
        </configuration>
<plugin>

$ mvn clean compile
```

Apache Maven (2004)

Advantages:

- Strict convention on code layout
- Strict lifecycle: goals are predefined
- Code sharing through remote repositories
- Dependencies info stores in local repo
- Code versioning rules
- Multimodul projects support
- Build description through Declarative approach

Gradle (2009)

```
$ cat build.gradle

repositories {
        mavenCentral()
}

configurations {
        agent
}

dependencies {
        agent “org.aspectj:aspectjweaver:${aspectjVersion}”
        testCompile “ru.yandex.qatools.allure:allure-testing-adaptor:${allureVersion}”

}

$ gradle clean compile
```

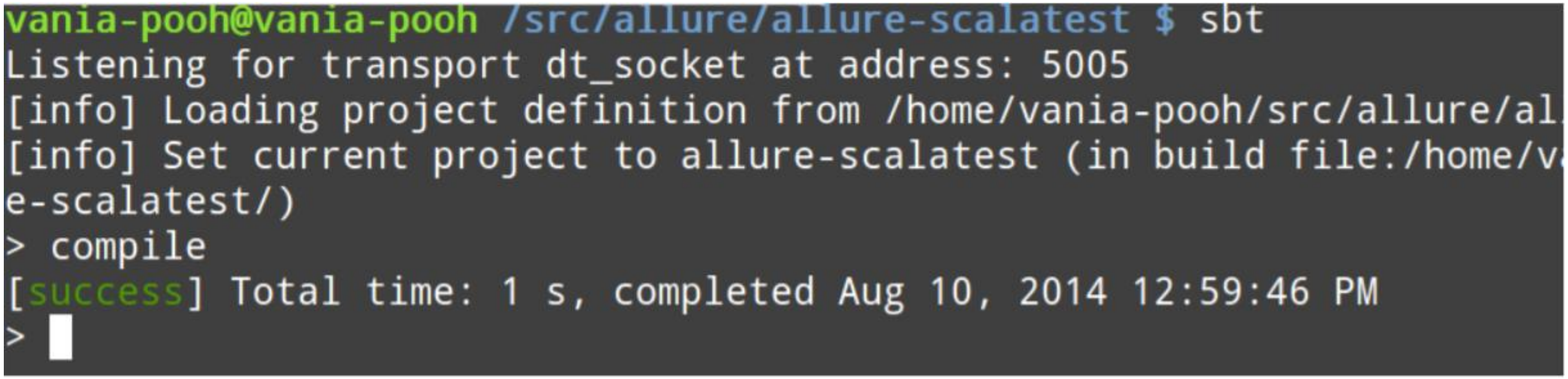
Gradle (2009)

Advantages:

- Supports main Maven fetures
- build.gradle file with domain-specific language (DSL) on Groovy
- Incremental compilation support
- Uses same remote repo as Maven
- Able to emulate Maven behaviour
- Supports plugins, which Maven does not

Scala build Tool SBT (2011)

```
$ cat build.sbt build.scala

version := “0.0.1”

scalaVersion := “2.10.3”

resolvers +=
   “Sonatype OSS Snapshots” at https://oss.sonatype.org/content/repositories/snapshots

libratyDependencies ++= Seq(
   “org.scalatest” % “scalatest_2.10” % “2.1.4” % “test”,
   “ru.yandex.qatools.allure” % “allure-scalatest_2.10” % “1.4.0-SNAPSHOT” % “test”
)

testOptions in Test ++= Seq(
   Tests.Argument(TestFrameworks.ScalaTest, “-oD”),
   Tests.Argument(TestFrameworks.ScalaTest, “-C”, “ru.yandex.qatools.allure.scalatest.All
)
```



```
vania-pooh@vania-pooh /src/allure/allure-scalatest $ sbt
Listening for transport dt_socket at address: 5005
[info] Loading project definition from /home/vania-pooh/src/allure/al
[info] Set current project to allure-scalatest (in build file:/home/v
e-scalatest/)
> compile
[success] Total time: 1 s, completed Aug 10, 2014 12:59:46 PM
> 
```

Scala build Tool SBT (2011)

Advantages (~Features):

- Interactive console
- Incremental code compilation
- Plugins, which Maven does not support

Leiningen(2009)

Advantages (~Features):

- project.cli written on Clojure
- Support dependencies from Maven repos
- Has alternative repo Clojars (http://clojars.org)

Build tools
for non-Java languages

Rake (Ruby)

Advantages (~Features):

- description in Rakefile
- uses dependencies management system - RubyGems

Grunt (Javascript)

Advantages (~Features):

- description in Gruntfile.js
- uses dependencies management system - Bower

Cabal (Haskell)

Advantages (~Features):

- own repo Hackage
- archives are in *.tar.gz format, not *.jar

Modern build automation system

- The main goal is to automate actions with code on the local machine of the developer
- Automatic dependency management
- Artifact Repositories
- Clear life cycle
- Versioning conventions
- Source code location conventions

The advantages of build automation

- A necessary precondition for continuous integration and continuous testing
- Improve product quality
- Accelerate the compile and link processing
- Eliminate redundant tasks
- Minimize "bad builds"
- Eliminate dependencies on key personnel
- Have history of builds and releases in order to investigate issues
- Save time and money - because of the reasons listed above.