Build tools

NTI/TUL, LS 2025, LenkaKT

What the “build” means?

“Historically, build has often referred either to the process of converting source code files into standalone software artifact(s) that can be run on a computer, or the result of doing so.”

(Wikipedia, https://en.wikipedia.org/wiki/Software_build)

Compilation is just a part of the building process.

Build is the path from the source to the working application.

What must be done?

- **Source compilation**
  - File order (if we have more than 1 file)
  - Compile only what has changed since last compilation
  - Check, if patches are needed in linked libraries
  - Define the right target (development? debugging? production deployment?)
  - Define the target architecture

- **Pack all artifacts and binaries and get it ready to deploy (.jar, .zip… etc.)**
- **Prepare installers**
- **Update related data (**pictures, media, databases, documentation).

What is useful as well?

- Version control
- Continuous integration
- Continuous deployment
- Update and build of documentation
- Team communication
- Synchronization with other tools used to man development process

Build automation

- Proces sestavení může být automatizován, jestliže jsou jednotlivé kroky:
  - přesně definovány,
  - mohou být jasně popsány,
  - jsou opakovatelné.
- Výhody automatizace:
  - Minimalizace chyb
  - Přesně opakovatelný proces
  - Podrobná dokumentace procesu

Jak to udělat?

- Užitím nástrojů: make, ant, maven, gradle, cmake...
- Jaký typ nástroje:

  Záleží na jazyce, prostředí, operačním systému, náladě v týmu…

Co je CI/CD?

- **CI** = **Continous Integration** = nová verze software je automaticky překládána, testována atd.
- **CD = Continous deployment** = nová verze software je automaticky připravena k nasazení
- CI pipeline: version - compile - build (merge)
- CD pipeline: release to repository - release to production
- Vše probíhá automaticky
- Dnes podporováno na mnoha úrovních skládáním mnoha nástrojů
- Komplexní nástroje (Jenkins, Spinnaker, GoCD, GitLab apod.)

Aktuální trend: CI/CD pipelines například přímo v gitlabu apod.: od zdrojáku k výsledku se správou verzí a řízením týmu.

Otec všech - Make

- 1977!!!
- Úlohy se jmenují **targets**
- Syntaxe skoro shell, logika též
- Targety na sobě mohou záviset
- Linuxové prostředí
- Nejsilnější reference: the Linux kernel

GNU make – What is wrong?

- Kvůli navázání na shell pomalé
- Příšerná syntaxe „Sometimes whitespace matters; sometimes it does not.“
- Závislosti se řeší manuálně
- Někdy složitá vazba na automatické CI/CD

- What is wrong with GNU make?
- https://web.archive.org/web/20160813235106/http://www.conifersystems.com/whitepapers/gnu-make/
- Well, it is not SO wrong: https://bost.ocks.org/mike/make/

A co takhle něco podobného?

- Problém závislostí
- Navrch lepší, uvnitř make

  Příklady: Cmake (***C**ross**Make***): Makefiles, NinjaBuild files, KDEvelop, Xcode, VisualStudio…

```
project(directory_test)

#Bring the headers, such as Student.h into the project
include_directories(include)

#Can manually add the sources using the set command as follows:
set(SOURCES src/mainapp.cpp src/Student.cpp)

#However, the file(GLOB...) allows for wildcard additions:
file(GLOB SOURCES "src/*.cpp")
add_executable(testStudent ${SOURCES})
```

Bitbake

- Překlad celých distribucí OS LInux
- Nad miliony řádek kódu řízených přes Makefiles, cmake, whatever
- Recepty
- Kombinace: Python, Shell, make
- **Tasks:**

  do_patch

  do_fetch

  do_compile...

```
do_install() {
 install -m 0755 -d ${D}${bindir} ${D}${docdir}/myhelloworld
 install -m 0644 ${S}/myhelloworld ${D}${bindir}
 install -m 0644 ${WORKDIR}/README.txt ${D}${docdir}/myhelloworld
}
```

Ant: XML & Java

- “make” ve světě Javy
- Základ slabý, extensions
- Výkon neslavný
- Nelze integrovat další nástroje (git)
- XML – not so human readable
- Neřeší závislosti

```
<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>
```

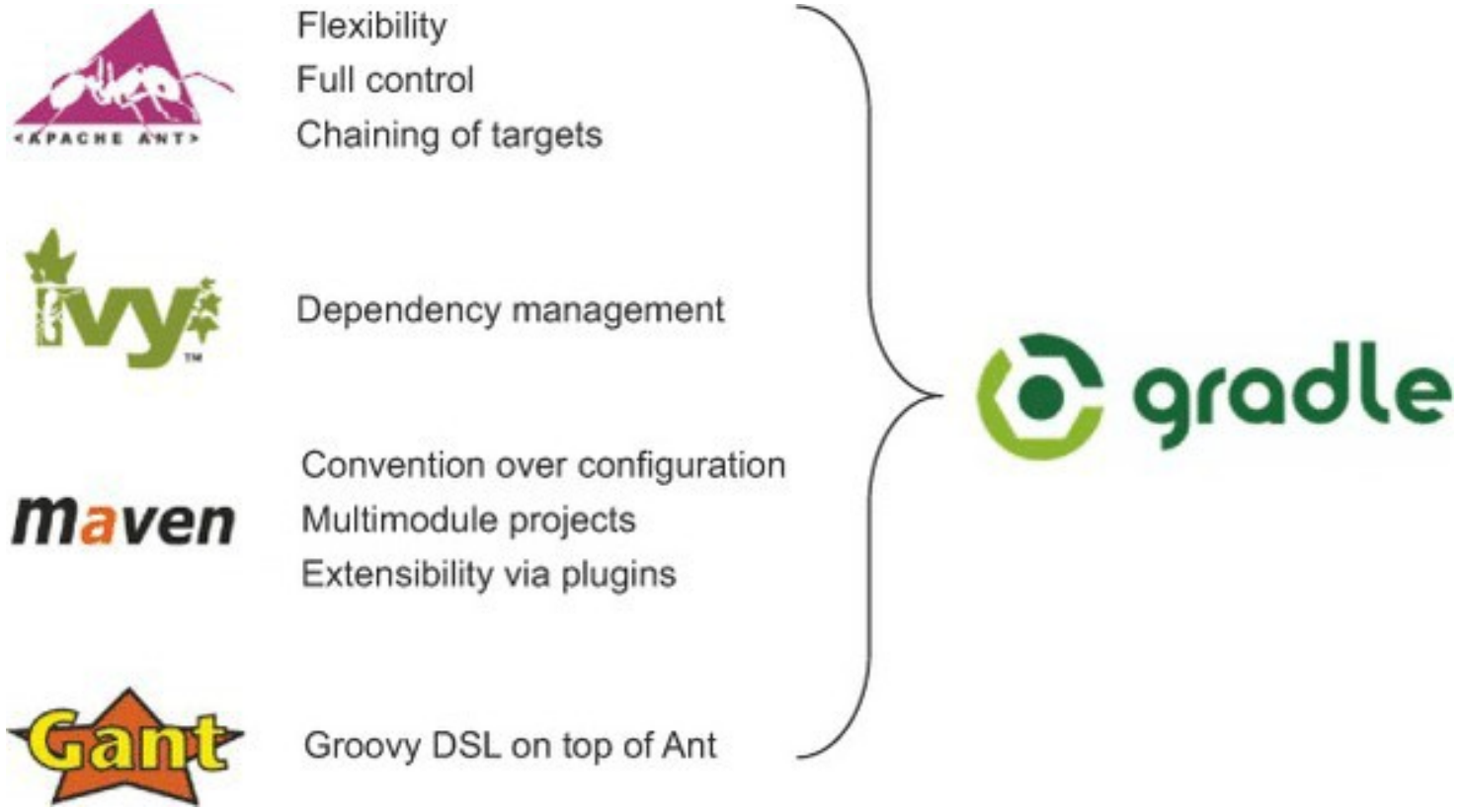
Maven: O krok dál

- XML, ale přesnější
- Integrace Git & versioning tools
- Project Object model (pom.xml - závisosti, pluginy, verze, repozitáře, mail listy)

- mvn compile
- mvn package
- mvn install
- Konečná množina úkolů

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="...“ …">
    <modelVersion>4.0.0</modelVersion> <groupId>org.springframework</groupId>
<artifactId>gs-maven</artifactId>
    <packaging>jar</packaging> <version>0.1.0</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTrans
former">
                                    <mainClass>hello.HelloWorld</mainClass>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Když zkombinujeme dobré s dobrým



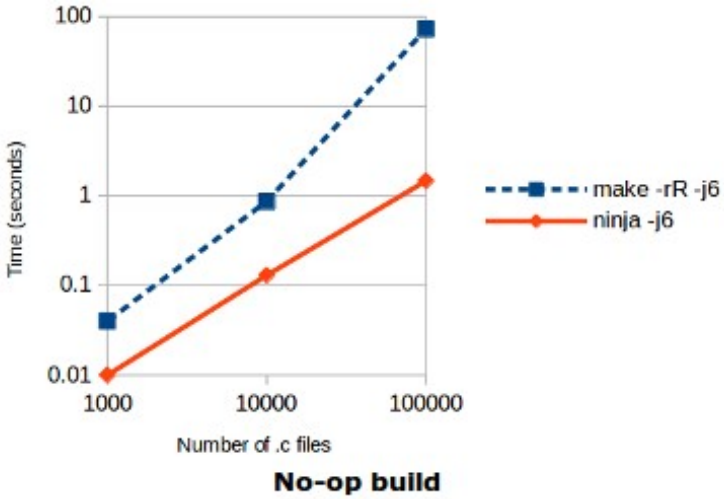
Gradle: The best of Ant and Maven

- Inspirace u Ant, ale použitelné pro projekty
- Vlastní konfigurační jazyk (DSL, inspired by Groovy): nemusíte si psát plugin, dá se integrovat
- Dobré řešení pro závislosti
- https://gradle.org/maven-vs-gradle

```
pply plugin: 'java'
apply plugin: 'checkstyle'
apply plugin: 'findbugs'
apply plugin: 'pmd'

version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit',
name: 'junit', version: '4.11'
    testCompile group:
'org.hamcrest', name: 'hamcrest-
all', version: '1.3'
}
```

Neviditelný a silný? Ninja!

- Navrženo pro extrémně rychlý překlad
- Used to compile: Chrome, Android, LLVM...




```
cc     = clang
cflags = -Weverything
rule compile
  command = $cc $cflags -c $in -o $o
rule link
  command = $cc $in -o $out
build hello.o: compile hello.c
build hello: link hello.o
default hello
```

A nyní vědecky: SCons

- Vytváří a prochází strom závislostí
- Taskmaster generuje seznam akcí podle stromu závislostí
- Vychází z Pythonu
- Rychlejší než make, ale náročnšjší na paměť
- Blízké shellu

```
import os
 env = Environment(CC = 'gcc', CCFLAGS = '-O2')
 env.Program('foo.c')

dict = env.Dictionary()
keys = dict.keys()
keys.sort()
for key in keys:
      print "construction variable = '%s', value = '%s'" % (key, dict[key])
```

Windows: MS Build

- VisualStudio a .NET
- XML
- Tasky určeny pevně (build, delete, exec…)
- Ale lze přidávat (C#)
- Nástroj pro závislosti NuGet

```
<?xml version="." encoding="." ?>
<Project xmlns="...">
  <Target Name="Build">
    <Message Text="Building msbuildintro" />
    <MSBuild Projects="msbuildintro.csproj" Targets="Build" />
  </Target>
</Project>
…
…
<Target Name="BeforeBuild">
</Target>
<Target Name="AfterBuild">
</Target>
```

Ruby and rake

- Překlad Ruby řízený v Ruby
- Hodně blízké programu make
- Mainstream ve světě Ruby, jinde skoro ne

```
task :default do
  puts "Hello World!"
end
task :manipulate_files do
  mkdir 'new_dir'
  mv 'new_dir', 'lenkakt'
  chmod 0777, 'lenkakt'
  touch 'lenkakt/lenkakt.txt'
  rm_rf 'lenkakt'
end
```

A dál?

https://en.wikipedia.org/wiki/List_of_build_automation_software