



Universidad of Oviedo



SOFTWARE
ARCHITECTURE

Software architecture

Lab. 05

Building automation

Maven, Gradle, npm, grunt,...

Dependency management

2020-21

Jose Emilio Labra Gayo
Pablo González
Irene Cid
Paulino Álvarez

Software builders

- **Tasks**
 - **Compilation**
 - From source code to binary code
 - **Packaging**
 - Dependency management and integration
 - Also called linking
 - **Test execution**
 - **Deployment**
 - **Documentation creation / *release notes***

Building automation

- Automate building tasks
- Objectives:
 - Avoid errors (minimize “*bad buildings*”)
 - Eliminate redundant and repetitive tasks
 - Manage complexity
 - Improve the product quality
 - Store a building and release history
 - Continuous integration
 - Save time and money

Automation tools

- Makefile (C)
- Ant (Java)
- Maven (Java)
- Npm (Node.js)
- SBT (Scala, JVM languages)
- Gradle (Groovy, JVM languages)
- rake (Ruby)
- etc.

Maven

Building automation tool

- Describe how to build the software
- Describe software dependencies

Principle: Convention over configuration

- Maven provides a default behaviour for the project

Maven

Building phases:

clean, compile, build, test, package, install, deploy

Module identification

3 coordinates: Group, Artifact, Version

Dependencies between modules

Configuration: XML file (Project Object Model)

`pom.xml`

Maven

Artifacts storages

Store different types of artifact

JAR, EAR, WAR, ZIP files, plugins, etc.

All the interactions are done through the repository

Without relative paths

Share models between development teams

Maven

POM file (pom.xml)

XML syntax

Describe a project

Name and version

Artifact type (jar, pom, ...)

Source code localization

Dependencies

Plugins

Profiles

Alternative building configurations

Maven

Project identification

GAV (Group, Artifact, version)

Group: Group identifier

Artifact: Project name

Version: Format {Bigger}.{Smaller}.{Development}

"-SNAPSHOT" can be added (during development)

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>es.uniovi.asw</groupId>
  <artifactId>Entrecine8</artifactId>
  <version>1.0</version>
</project>
```

Maven

Directory structure

Maven uses a conventional structure

src/main

src/main/java

src/main/webapp

src/main/resources

src/test/

src/test/java

src/test/resources

...

Maven

Development cycle

generate-sources/generate-resources

compile

test

package

integration-test

install

deploy

clean

Invocation:

```
mvn clean
```

```
mvn compile
```

```
mvn clean compile
```

```
mvn compile install
```

```
...
```

Maven

Automatically managing of dependencies

Identification through GAV

Environment

compile

test

provided

Type

jar, pom, war,...

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
      <scope>provided</scope>
    </dependency>
    . . .
  </dependencies>
</project>
```

Maven

Automatically managing of dependencies

Dependencies are downloaded

Stored in a local repository

Intermediate repositories can be created (proxies)

Example: common artifacts for a company

Transitivity

B depends of C

A depends of B -> C is also downloaded

Maven

Multiple modules

Big projects can be divided

Each Project creates an artifact

They have their own pom.xml file

The parent project groups all of them

```
<project>
  ...
  <packaging>pom</packaging>
  <modules>
    <module>extract</module>
    <module>game</module>
  </modules>
</project>
```

Maven

Other phases and plugins

`archetype:generate` – Generates the archetype of the project

`eclipse:eclipse` – Generate eclipse project

`site` – Generate website of the project

`site:run` - Generate website and runs server

`javadoc:javadoc` – Generate documentation

`cobertura:cobertura` – Informs of the code coverage

`checkstyle:checkstyle` – Check the codification style

Gradle

- Designed specifically for projects based on Java.
- Based on Groovy instead of XML
- To build multi-projects.

Gradle

- Two basic concepts
 - Project: Something that we build (for example jar files) or what we do (deploy our application)
 - Task: Atomic pieces of work during a build (for example compile our project or launch tests)

Gradle

- **Tasks:**
 - Scripts are saved in build.gradle.
 - Next example defines a task named “hello” that is used to print “ASW”

```
task hello {  
    doLast {  
        println 'ASW'  
    }  
}
```

- **Execution:**

```
C:\> gradle -q hello
```

Gradle

- Add dependencies to the tasks: A task can be only executed when the tasks that it depends on finish

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

```
task taskY << {  
    println 'taskY' }  
task taskX << {  
    println 'taskX' }  
taskY.dependsOn taskX
```

- Execution result:

```
taskX  
taskY
```

Gradle

- dependencies: Similar to Maven the libraries are downloaded from a repository (it can even be a Maven repository)

```
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.+'
}
```

Gradle

- **Dependency configuration**
 - **Compile:** The dependencies required to compile the source code of the project.
 - **Runtime:** Dependencies required by the production classes during runtime.
 - **Test Compile:** Dependencies used to compile the test classes.
 - **Test Runtime:** Dependencies required to execute the tests.

Gradle

- **External dependencies:** Dependencies which some of their files are built outside the current build. They are stored in an external repository like Maven central:

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

Gradle

- **Repositories:** When external dependencies are added Gradle searches them in a repository

```
repositories {  
    mavenCentral()  
}
```

Gradle - plugins

- Plugin: Set of tasks
 - Extends the basic model of Gradle
 - Configs the Project
 - Applies specific configurations
- 2 types
 - Scripts: Can be applied locally or remotely

```
apply from: 'other.gradle'
```

- Binaries: Identified by a plugin id

```
apply plugin: JavaPlugin
```

```
plugins {  
    id 'java'  
}
```

```
plugins {  
    id "com.jfrog.bintray"  
    version "0.4.1"  
}
```


npm

Node.js Package Manager

Initially created by Isaac Schlueter

Later became Npm inc.

3 things:

1. Website (<https://www.npmjs.com/>)

User and organization management

2. Software registry

Public/private packages

3. CLI application

Dependency and task management

Configuration file: package.json

npm configuration: package.json

- Configuration file: package.json
 - npm init creates a simple skeleton

- Fields:

```
{
  "name":           "...mandatory...",
  "version":        "...mandatory...",
  "description":    "...optional...",
  "keywords":       "...",
  "repository":     {... },
  "author":         "...",
  "license":        "...",
  "bugs":           {...},
  "homepage":       "http://. . .",
  "main":           "index.js",
  "devDependencies": {... },
  "dependencies":   {... }
  "scripts":        { "test": " ... " },
  "bin":            {...},
}
```

Note: Yeoman provides fully featured scaffolding

npm packages

Registry: <http://npmjs.org>

Installing packages:

2 options:

Local

```
npm install <packageName> --save (--save-dev)
```

Downloads <packageName> contents to node_modules folder

Global

```
npm install -g <packageName>
```

Store the dependency in the package.json

Only for development

npm dependencies

Dependency management

Local packages are cached at `node_modules` folder

Access to modules through: `require('...')`

Global packages (installed with `--global` option)

Scoped packages marked by `@`

npm commands and scripts

Npm contains lots of commands

`start -> node server.js`

`test -> node server.js`

`ls` lists installed packages

...

Custom scripts:

`run <name>`

More complex tasks in NodeJs

Gulp, Grunt

<https://docs.npmjs.com/cli-documentation/>

NPM packages

- Dependencies: Stored in package.json
- Package: Identified by name and version
- Rule for names:
 - Less than or equal to 214 characters.
 - Can't start with a dot or an underscore.
 - New packages must not have uppercase letters in the name.
 - The name ends up being part of a URL, an argument on the command line, and a folder name. Therefore, the name can't contain any non-URL-safe characters.

NPM semantic versioning

- Version of the package: Semantic versioning
 - Must be parseable by [node-semver](#)
- Ranges: Comparators which specify versions that satisfy the range
 - For example, the comparator `>=1.2.7` would match the versions 1.2.7, 1.2.8, 2.5.3, and 1.3.9, but not the versions 1.2.6 or 1.1.0.
 - More at <https://docs.npmjs.com/misc/semver>

NPM package.json fields

Reference: <https://docs.npmjs.com/files/package.json>

Fields:

- description
- keywords
- homepage: URL to Project homepage
- bugs: URL of project's issue tracker and/or the email address to which issues should be reported
- people fields: author, contributors.
 - The “author” is one person. “contributors” is an array of people. A “person” is an object with a “name” field and optionally “url” and “email”

NPM package.json fields

- files: An array of file patterns that describes the entries to be included when your package is installed as a dependency
- file patterns follow a similar syntax to .gitignore, but reversed:
 - Including a file, directory, or glob pattern (*, **/*, and such) will make it so that file is included in the tarball when it's packed.
 - Omitting the field will make it default to ["*"], which means it will include all files.

NPM files included

- Certain files are always included, regardless of settings:
 - `package.json`
 - `README`
 - `CHANGES / CHANGELOG / HISTORY`
 - `LICENSE / LICENCE`
 - `NOTICE`
 - The file in the “main” field

NPM package.json fields

- **main:** module ID that is the primary entry point to your program
 - This should be a module ID relative to the root of your package folder.
 - For most modules, it makes the most sense to have a main script and often not much else.
- **browser:** If the module is meant to be used client-side the browser field should be used instead of the main field.
 - This is helpful to hint users that it might rely on primitives that aren't available in Node.js modules (eg a window).

NPM package.json fields

- repository: the place where the code lives.

```
"repository": {  
  "type" : "git",  
  "url" : "https://github.com/npm/cli.git"  
}
```

```
"repository": {  
  "type" : "svn",  
  "url" : "https://v8.googlecode.com/svn/trunk/"  
}
```

NPM package.json fields

- **config**: Used to set configuration parameters used in package scripts that persist across upgrades.

```
{  
  "name" : "foo" ,  
  "config" : { "port" : "8080" }  
}
```

NPM package.json fields

- dependencies: Dependencies are specified in a simple object that maps a package name to a version range.
 - The version range is a string which has one or more space-separated descriptors.
 - Version ranges based on semantic versioning:
 - See <https://docs.npmjs.com/misc/semver>

NPM package.json fields

- **devDependencies:** Dependencies required to develop the application such as unit tests.
- **URL dependencies:**
 - You may specify a tarball URL in place of a version range.
 - This tarball will be downloaded and installed locally to your package at install time.

```
<protocol>://[<user>[:<password>]@]<hostname>[:<port>][:][/]<path>[  
#<commit-ish> | #semver:<semver>]
```

NPM

- GIT URLs: Following form:

```
<protocol>://[<user>[:<password>]@]<hostname>[:<port>][:]/<path>[#<commit-ish>|#semver:<semver>]
```

- Example

```
git+ssh://git@github.com:npm/cli.git#v1.0.27  
git+ssh://git@github.com:npm/cli#semver:^5.0  
git+https://isaacs@github.com/npm/cli.git  
git://github.com/npm/cli.git#v1.0.27
```


Task Execution : Grup and Gulp

Execute JavaScript tasks:

- Compress images
- Package modules (webpack)
- Minimize js and css files
- Run tests
- Transcompile – babel.js

These tasks can be directly run with npm scripts or with Gulp and/or Grunt

Task Execution : Grup y Gulp

- Grup:

- Module fs
- Installation:

```
npm install -g grunt
npm install -g grunt-cli
```

- package.json configuration

```
{ "name": "ASW",
  "version": "0.1.0",
  "devDependencies": {
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
  }
}
```

- Gulp:

- Module stream
- Installation:

```
npm install --save-dev gulp
npm install -g gulp-cli
```

- gulpfile.js configuration

```
function defaultTask(cb) {
  // tareas
  cb();
}
exports.default = defaultTask
```

Examples

Wrapper

```
module.exports = function(grunt) {  
  // CONFIGURE GRUNT  
  grunt.initConfig({  
    (pkg.name)  
    pkg: grunt.file.readJSON('package.json'),  
  });  
  grunt.loadNpmTasks('grunt-contrib-uglify');  
  grunt.registerTask('default', ['uglify']);  
};
```

Wrapper

```
gulp.task('jpgs', function()  
{ return gulp.src('src/images/*.jpg')  
  .pipe(imagemin({ progressive: true }))  
  .pipe(gulp.dest('optimized_images')); });
```

End