Software Engineering

LESSON 04

Build Tools

Outline

1. Introduction
2. Apache Maven
3. Sample Maven project
4. Gradle
5. Sample Gradle project

Overview

In this chapter, you are going to learn about

- Know Build tools
- Know how to create project managed by a build tool
- Know how to use Build tools
- Know how to use Gradle
- Know how to implement Java Project and automate builds using build tools

Learning content

1. Introduction
   - What is build tools?
   - Directed Acyclic Graph (DAG)
   - Anatomy of build tools
2. Apache Ant
   - Apache Ant tasks
   - Ant build script
   - Pros and cons
3. Apache Maven
   - Build lifecycle
   - Dependency management
   - Pros and cons
4. Gradle
   - Why Gradle?
   - Compare Gradle with others
   - Gradle features
5. Sample project
   - Installing Gradle
   - Getting started with gradle
   - Gradle Command Line Interface (CLI)

Build tools

We need a tool that allows us to create a repeatable, reliable, and portable build without manual intervention

- What you need is a programming utility that lets you express your automation needs as executable, ordered tasks. Let’s say you want to compile your source code, copy the generated class files into a directory, and assemble a deliverable that contains the class files.
- A deliverable could be a ZIP file, for example, that can be distributed to a runtime environment.

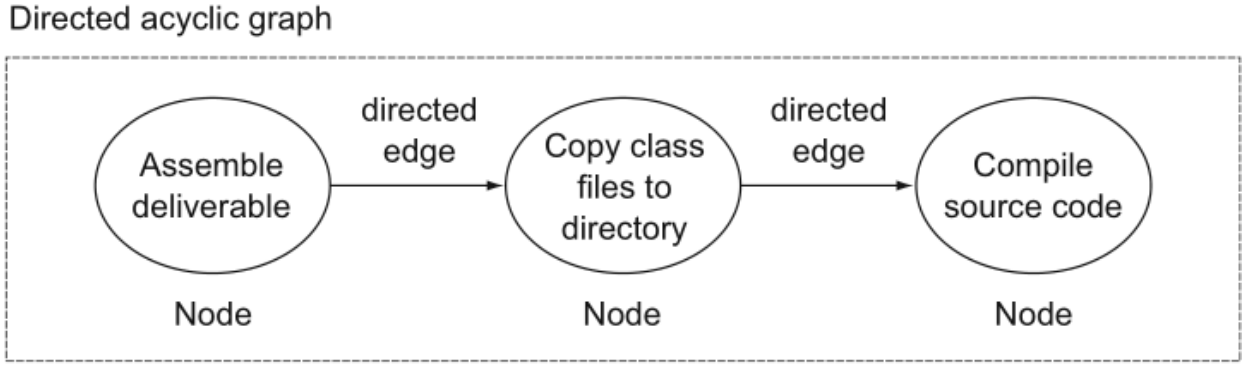
DIRECTED ACYCLIC GRAPH (DAG)

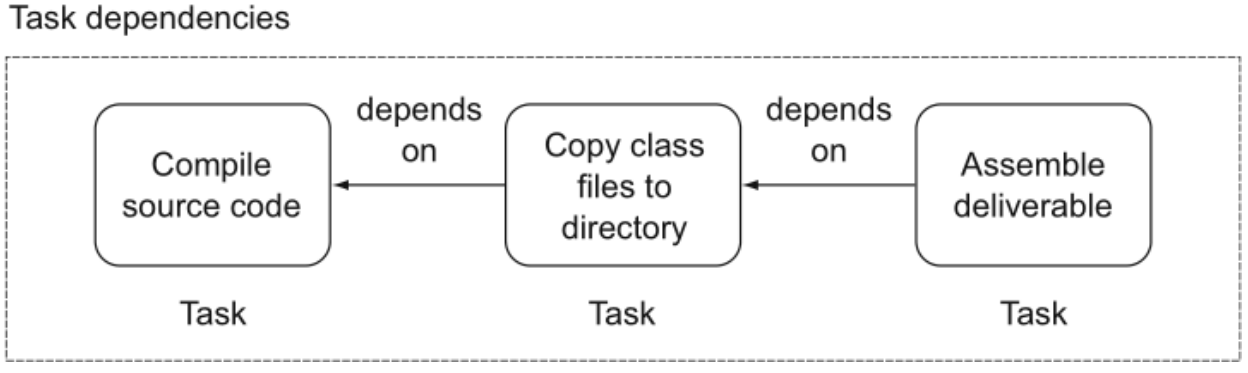
A DAG is a data structure from computer science and contains the following two elements:

- Node: A unit of work; in the case of a build tool, this is a task (for example, compiling source code).
- Directed edge: A directed edge, also called an arrow, representing the relationship between nodes. In our situation, the arrow means depends on. If a task defines dependent tasks, they'll need to execute before the task itself can be executed. Often this is the case because the task relies on the output produced by another task. Here's an example: to execute the task "assemble deliverable," you'll need to run its dependent tasks "copy class files to directory" and "compile source code."

Directed Acyclic Graph (DAG)

Each node knows about its own execution state. A node—and therefore the task—can only be executed once. For example, if two different tasks depend on the task "source code compilation," you only want to execute it once.

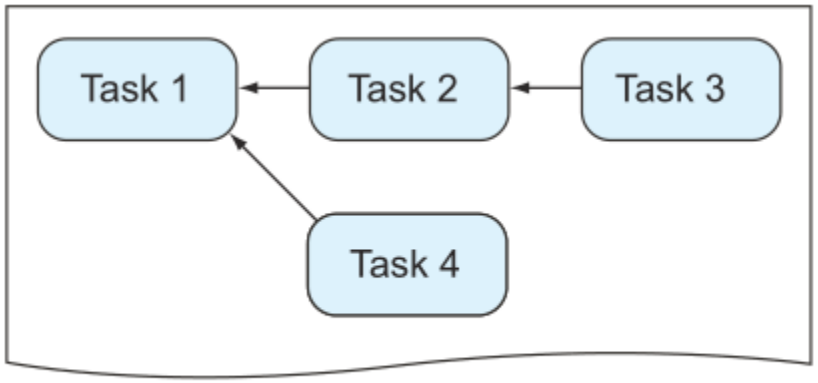




Anatomy of a build tool

- **BUILD FILE**

The build file contains the configuration needed for the build, defines external dependencies such as third-party libraries, and contains the instructions to achieve a specific goal in the form of tasks and their interdependencies.

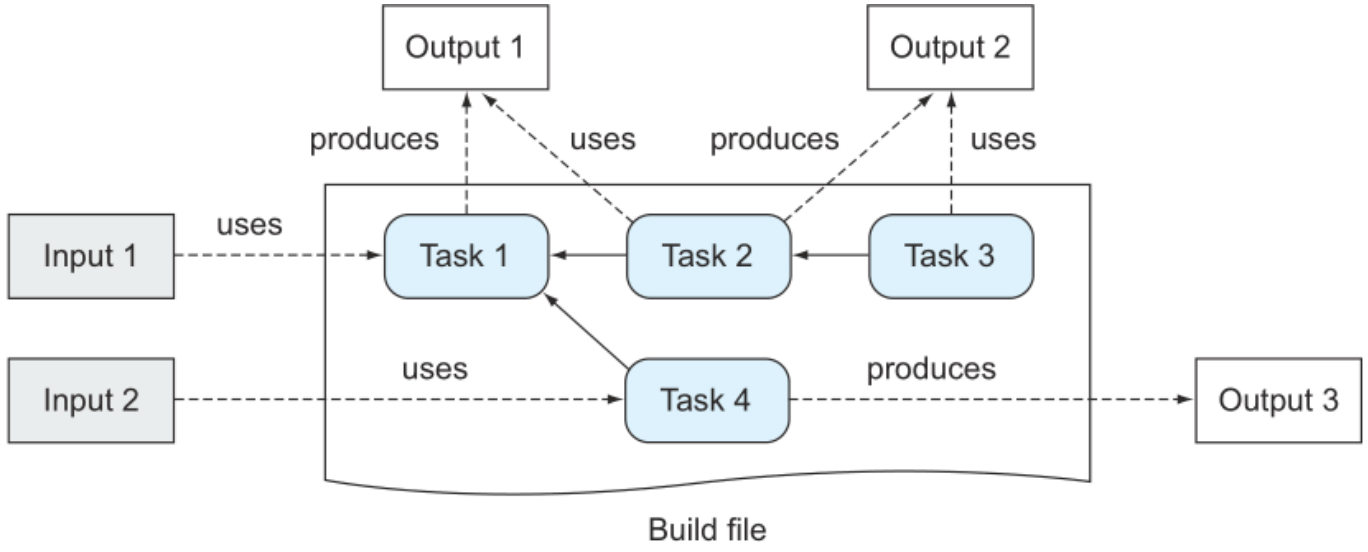


Build file

Anatomy of a build tool

- **BUILD INPUTS AND OUTPUTS**

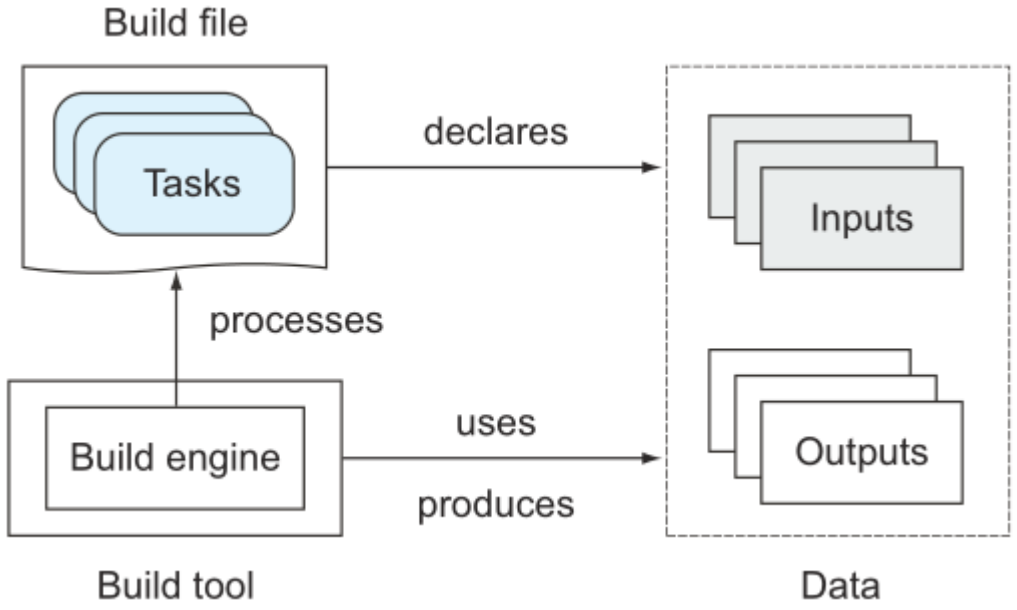
A task takes an input, works on it by executing a series of steps, and produces an output. Some tasks may not need any input to function correctly, nor is creating an output considered mandatory. Complex task dependency graphs may use the output of a dependent task as input.



Anatomy of a build tool

- **BUILD ENGINE**

The build file's step-by-step instructions or rule set must be translated into an internal model the build tool can understand. The build engine processes the build file at runtime, resolves dependencies between tasks, and sets up the entire configuration needed to command the execution



Anatomy of a build tool

- **DEPENDENCY MANAGER**

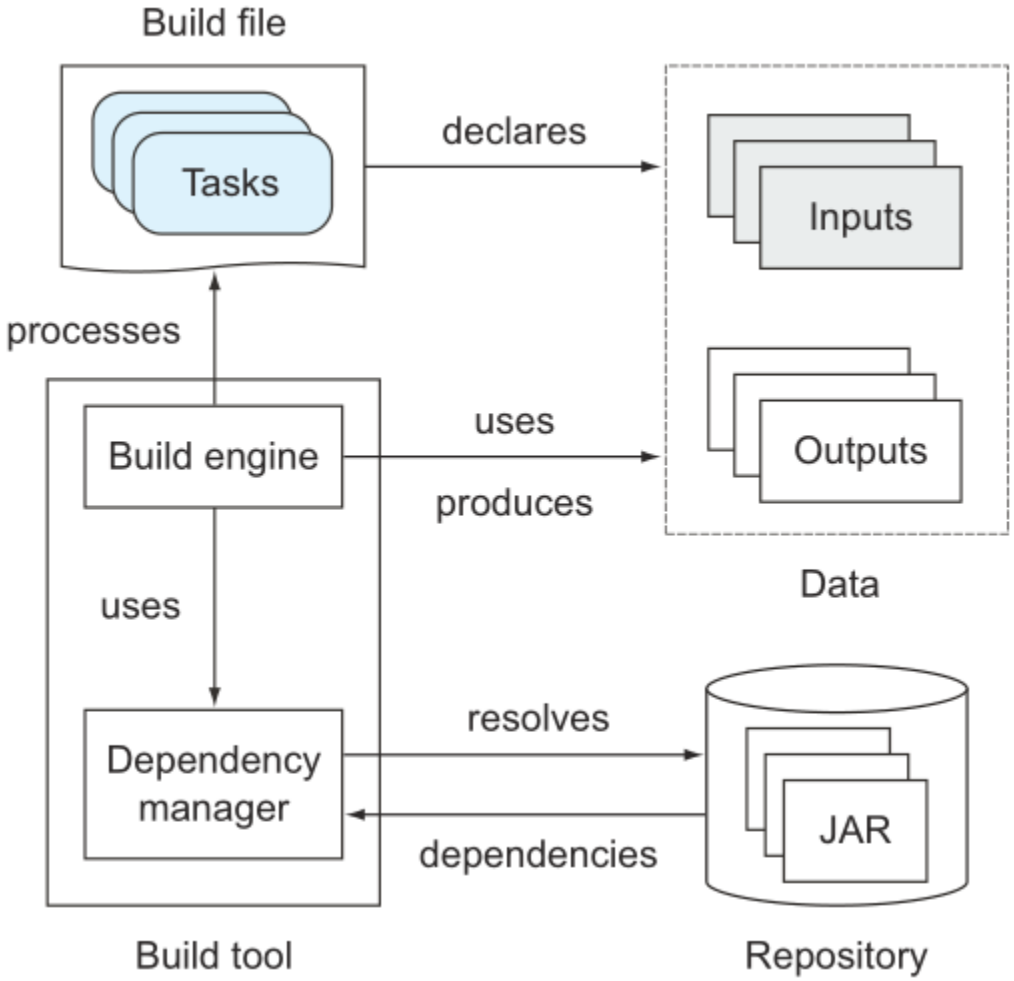
The dependency manager is used to process declarative dependency definitions for your build file, resolve them from an artifact repository (for example, the local file system, an FTP , or an HTTP server), and make them available to your project.

A *dependency* is generally an external, reusable library in the form of a JAR file (for example, Log 4J for logging support).

The *repository* acts as storage for dependencies, and organizes and describes them by identifiers, such as name and version. A typical repository can be an HTTP server or the local file system.

Anatomy of a build tool

Many libraries depend on other libraries, called *transitive dependencies*. The dependency manager can use metadata stored in the repository to automatically resolve transitive dependencies as well. A build tool is not required to provide a dependency management component.



Java build tools

- There are several build tools, but we will focus on 2 popular Java-based build tools: Ant and Maven

https://ant.apache.org/
Since 1999
Current version: 1.10.x (Aug 20, 2023)

Apache Ant (Another Neat Tool) is an open-source build tool written in Java.
(Since Ant requires advanced knowledge base, we will not focus it on our lesson)

https://maven.apache.org/
Since 2002
Current version: 3.9.x (Nov 18, 2023)

a Yiddish word meaning accumulator of knowledge, began as an attempt to simplify the build processes in the Jakarta Turbine project.

Java build tools – Apache Maven

- The Maven team realized the need for a standardized project layout and unified build lifecycle.
- Maven picks up on the idea of convention over configuration, meaning that it provides sensible default values for your project configuration and its behavior. The project automatically knows what directories to search for source code and what tasks to perform when running the build. You can set up a full project with a few lines of XML as long as your project adheres to the default values.
- As an extra, Maven also has the ability to generate HTML project documentation that includes the Javadocs for your application.

Java build tools – Apache Maven

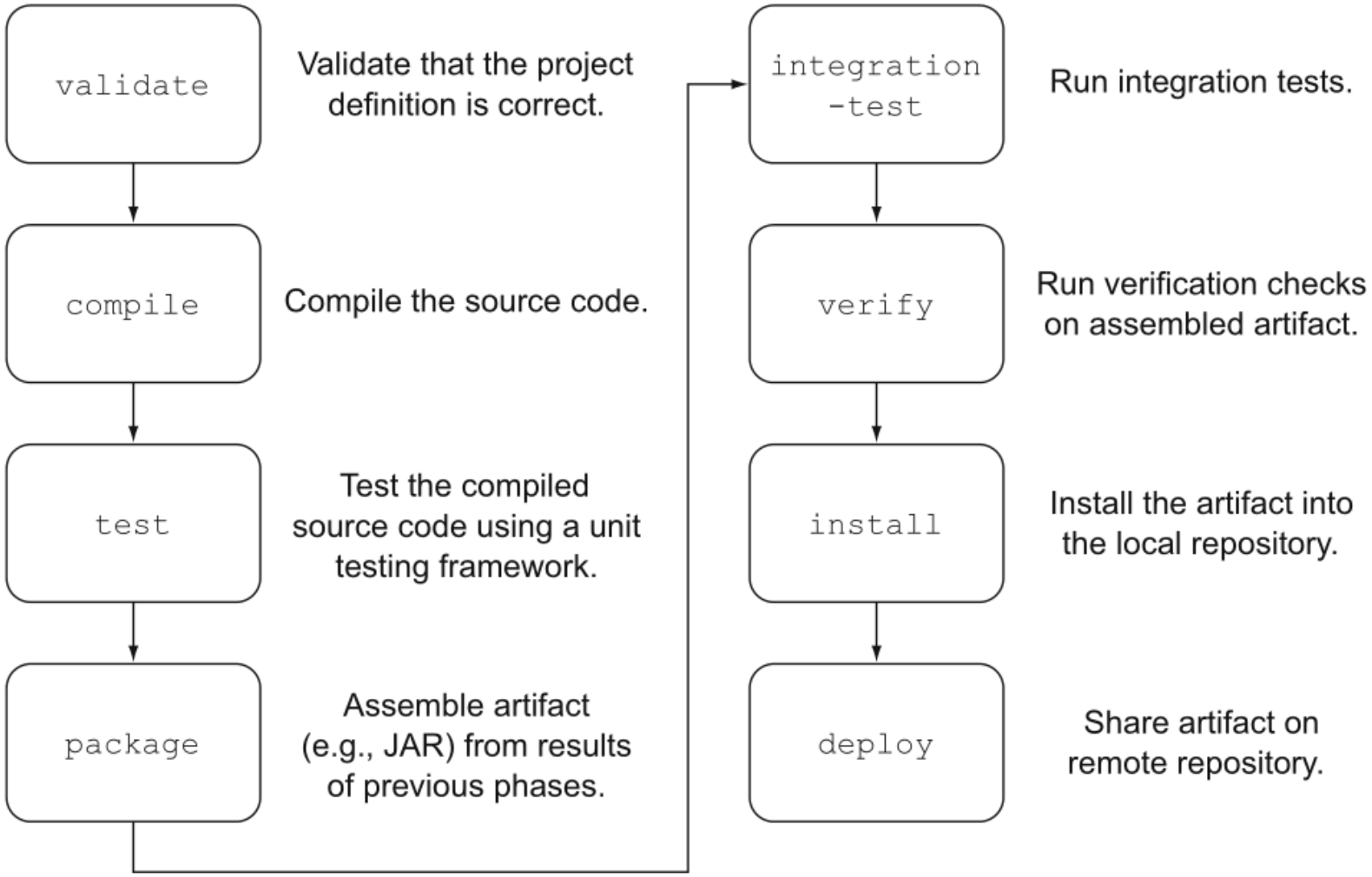
- Maven’s core functionality can be extended by custom logic developed as plugins.
- The community is very active, and you can find a plugin for almost every aspect of build support, from integration with other development tools to reporting. If a plugin doesn’t exist for your specific needs, you can write your own extension.

Maven Build Lifecycle

Maven is based on the concept of a build lifecycle. Every project knows exactly which steps to perform to build, package, and distribute an application, including the following functionality:

- Compiling source code
- Running unit and integration tests
- Assembling the artifact (for example, a JAR file)
- Deploying the artifact to a local repository
- Releasing the artifact to a remote repository

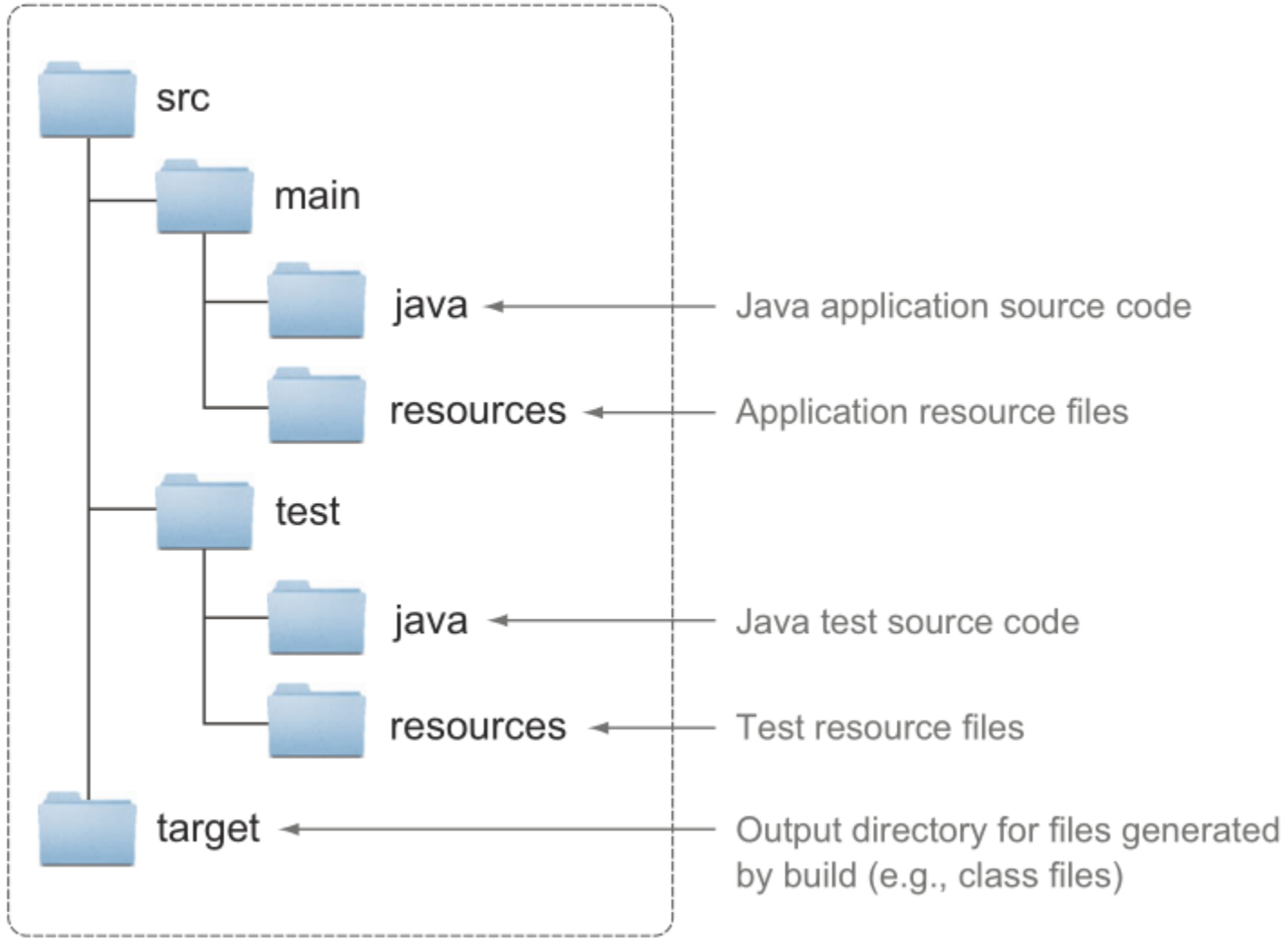
Maven Build Lifecycle



Maven Standard Directory Layout

- By introducing a default project layout, Maven ensures that every developer with the knowledge of one Maven project will immediately know where to expect specific file types. For example, Java application source code sits in the directory **src/main/java**.
- All default directories are configurable

Maven Default Project layout



Maven Dependency Management

- In Maven projects, dependencies to external libraries are declared within the build script. For example, if your project requires the popular Java library Hibernate, you simply define its unique artifact coordinates, such as organization, name, and version, in the dependency configuration block.

Every dependency is wrapped in a <dependency> tag

All dependencies of project must be declared within <dependencies> tag

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.2.Final</version>
  </dependency>
</dependencies>
```

Group identifier of dependency, usually an organization or company name

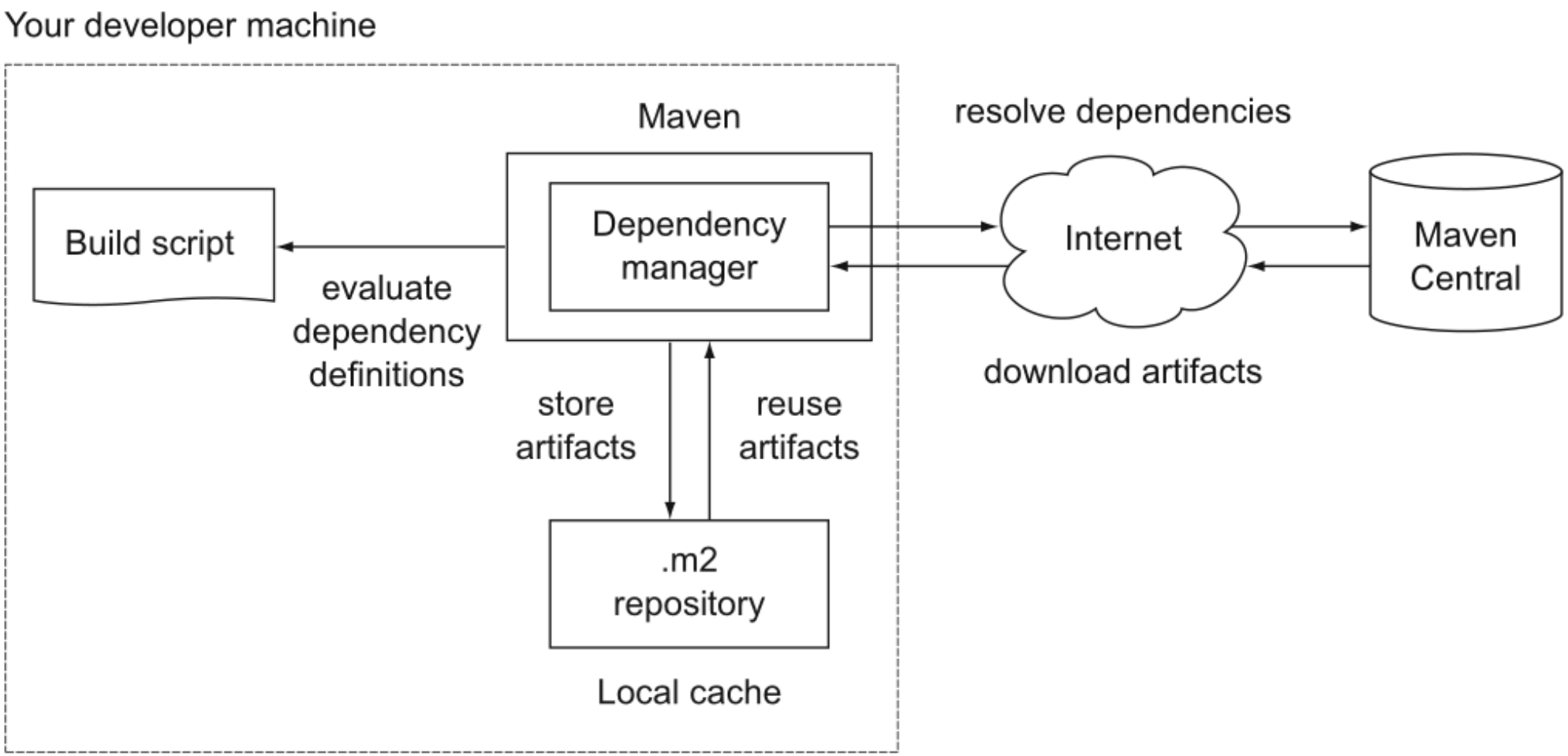
Name of a dependency

Version of a dependency, usually consisting of classifiers like minor and major version separated by a dot character

Maven Dependency Management

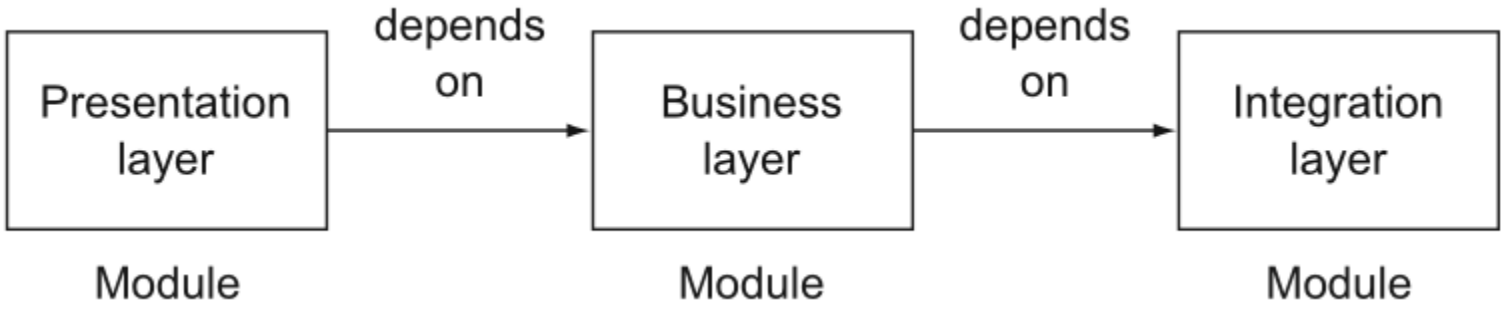
- At runtime, the declared libraries and their transitive dependencies are downloaded by Maven’s dependency manager, stored in the local cache for later reuse, and made available to your build (for example, for compiling source code).
- Maven preconfigures the use of the repository, Maven Central, to download dependencies. Subsequent builds will reuse an existing artifact from the local cache and therefore won’t contact Maven Central. Maven Central is the most popular binary artifact repository in the Java community.

Maven Dependency Management

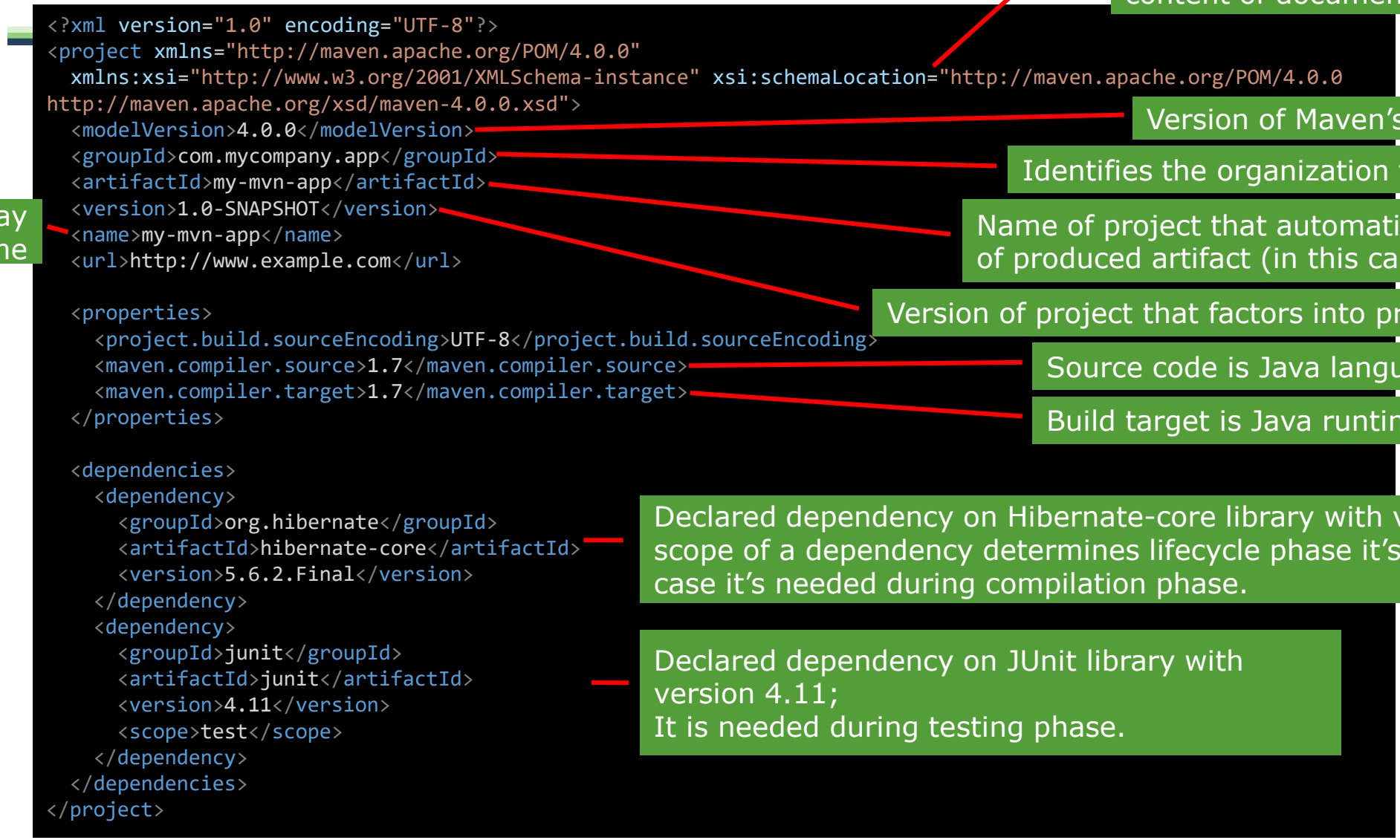


Maven Dependency Management

- Dependency management in Maven isn’t limited to external libraries. You can also declare a dependency on other Maven projects. This need arises if you decompose software into modules, which are smaller components based on associated functionality.



Sample pom.xml file



```
<?xml version="1.0" encoding="UTF-8"?>
<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>com.mycompany.app</groupId>
  <artifactId>my-mvn-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>my-mvn-app</name>
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.6.2.Final</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Apache Maven – Pros and Cons

- Maven proposes a default structure and lifecycle for a project that often is too restrictive and may not fit your project’s needs.
- Writing custom extensions for Maven is overly cumbersome. You’ll need to learn about Mojos (Maven’s internal extension API), how to provide a plugin descriptor (again in XML), and about specific annotations to provide the data needed in your extension implementation.

Needs for next generation build tools

Wouldn’t it be great if a build tool could cover a middle ground? Here are some features that an evolved build tool should provide:

- Expressive, declarative, and maintainable build language.
- Standardized project layout and lifecycle, but full flexibility and the option to fully configure the defaults.
- Easy-to-use and flexible ways to implement custom logic.
- Support for project structures that consist of more than one project to build deliverable.
- Support for dependency management.
- Good integration and migration of existing build infrastructure, including the ability to import existing Ant build scripts and tools to translate existing Ant/Maven logic into its own rule set.
- Emphasis on scalable and high-performance builds. This will matter if you have long-running builds (for example, two hours or longer), which is the case for some big enterprise projects.

Why Gradle?

- One big weak point of Apache Ant and Apache Maven is that their build logic must be described in XML. XML is great for describing hierarchical data, but falls short on expressing program flow and conditional logic. As a build script grows in complexity, maintaining the build code becomes a nightmare.
- We’re on the cusp of a new era of application development: polyglot programming. Many applications today incorporate multiple programming languages, each of which is best suited to implement a specific problem domain. It’s not uncommon to face projects that use client-side languages like JavaScript that communicate with a mixed, multilingual backend like Java, Groovy, and Scala, which in turn calls off to a C++ legacy application. It’s all about the right tool for the job. Despite the benefits of combining multiple programming languages, your build tool needs to fluently support this infrastructure as well. JavaScript needs to be merged, minified, and zipped, and your server-side and legacy code needs to be compiled, packaged, and deployed.

Why Gradle?

Flexibility
Full control
Chaining of targets

Dependency management

Convention over configuration
Multimodule projects
Extensibility via plugins

Groovy DSL on top of Ant

Comparing maven and Gradle scripts

Maven script

```
<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>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

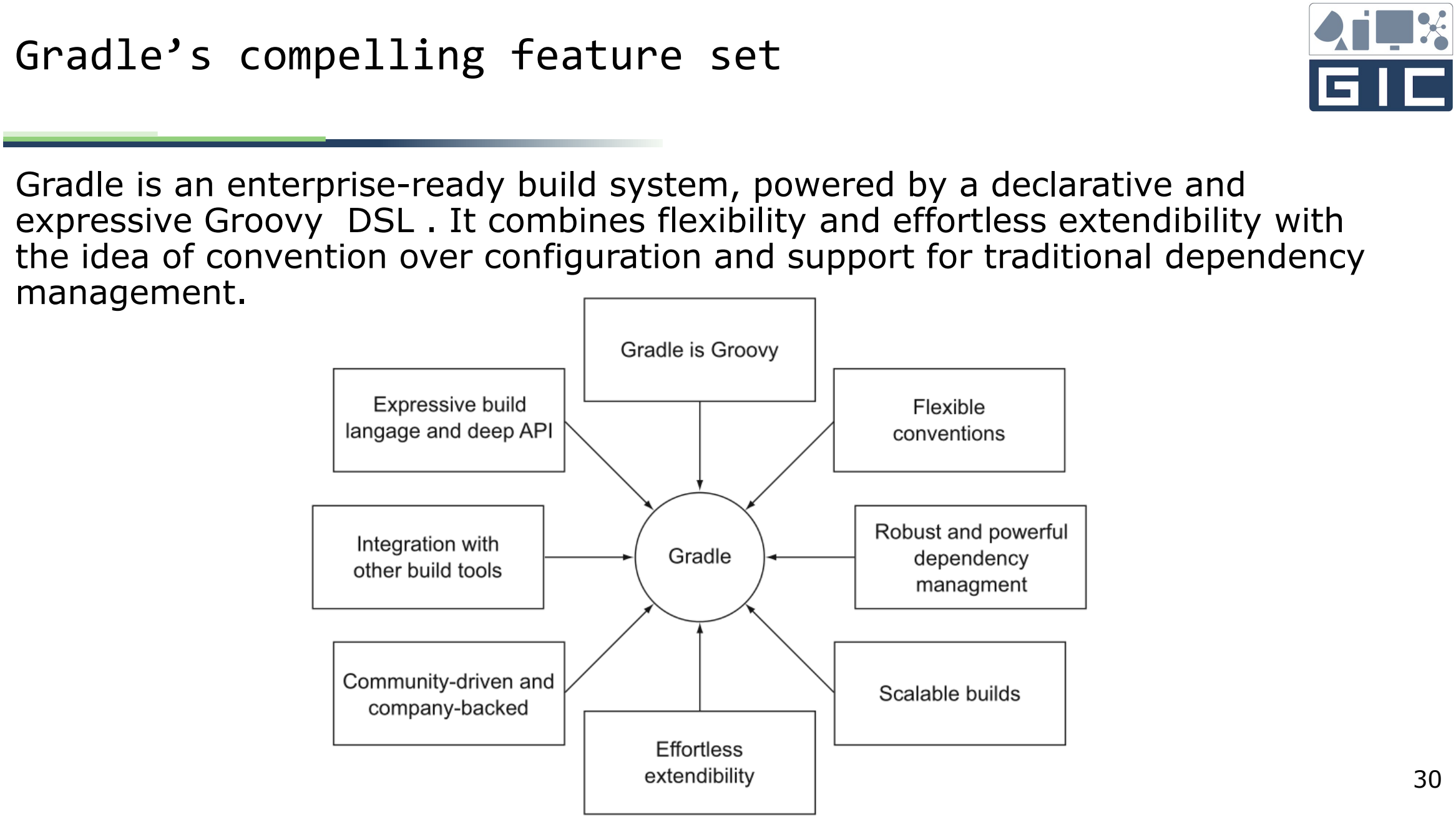
Gradle script (Groovy language)

```
apply plugin: 'java'
group = 'com.mycompany.app'
archivesBaseName = 'my-app'
version = '1.0-SNAPSHOT'

repositories {
  mavenCentral()
}

dependencies {
  testCompile 'junit:junit:4.11'
}
```

Gradle’s compelling feature set



Gradle is an enterprise-ready build system, powered by a declarative and expressive Groovy DSL . It combines flexibility and effortless extendibility with the idea of convention over configuration and support for traditional dependency management.

Installing Gradle

- Reference: https://gradle.org/install/
- As a prerequisite, make sure you've already installed the JDK with a version of 1.8 or higher
- To check your gradle version: **gralde –v**

Getting started with Gradle

- Every Gradle build starts with a script. The default naming convention for a Gradle build script is *build.gradle*. When executing the command gradle in a shell, Gradle looks for a file with that exact name. If it can't be located, the runtime will display a help message.
- Let's set the lofty goal of creating the typical "Hello world!" example in Gradle:
- Let's run it:

```
task helloWorld {
  doLast {
    println 'Hello world!'
  }
}
```

An action named doLast is almost self-expressive. It's the last action that's executed for a task.

```
>gradle -q helloWorld
Hello world!
```

Getting started with Gradle

- More advanced example:

```
task startSession {
  doLast {
    chant()
  }
}
def chant() {
  ant.echo(message: 'Repeat after me...')
}
3.times {
  task "yayGradle$it" {
    doLast {
      println 'Gradle rocks'
    }
  }
}

yayGradle0.dependsOn startSession
yayGradle2.dependsOn yayGradle1, yayGradle0
task groupTherapy(dependsOn: yayGradle2)
```

Run command:

```
gradle groupTherapy
```

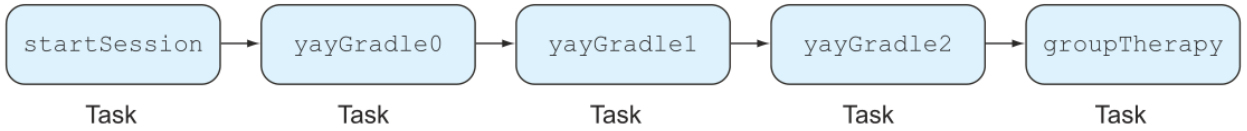
Output:

```
> Task :startSession
[ant:echo] Repeat after me...

> Task :yayGradle0
Gradle rocks

> Task :yayGradle1
Gradle rocks

> Task :yayGradle2
Gradle rocks
```



Command line Interface

- Listing available tasks of a project: `gradle -q tasks --all`
- Task execution: `gradle <task-name1> <task-name2> …`
- Task execution excluding tasks:

  `gradle <task-name1> -x <excluded-task-2>`

- Example:
  - A simple project call "To Do management"
  - To assemble an executable program, the source code needs to be compiled and the classes need to be packaged into a JAR file
  - First, we will use Maven as build tool
  - Then, we will use Gradle as build tool
  - Finally, we compare the 2 tools.

Generating the Project structure

- Wouldn’t it be great if you didn’t have to create the source directories manually?
  - Maven has a concept called project archetypes, a plugin to generate a project structure from an existing template. (sample: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html)
  - Gradle has CLI with a command called ***gradle init*** which is used to generate a project structure. (sample: https://docs.gradle.org/current/samples/sample_building_java_applications.html)

References

- Gradle tasks
  https://docs.gradle.org/current/userguide/more_about_tasks.html#more_about_tasks
- Installing Gradle
  https://gradle.org/install/
- Running Apache Maven
  https://maven.apache.org/run-maven/index.html
- Apache Ant build file
  https://ant.apache.org/manual-1.9.x/index.html
- Hibernate-core
  https://mvnrepository.com/artifact/org.hibernate/hibernate-core/5.6.2.Final