ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria e Scienze Informatiche

Build automation systems per Java: analisi dello stato dell'arte

Tesi di laurea in
(Programmazione ad oggetti)

Relatore
Prof. Danilo Pianini

Candidato
Simone Bonasegale

Correlatore
Prof. Mirko Viroli

Quarta sessione di Laurea
Anno Accademico 2019-2020

Sommario

Il processo di sviluppo di un software richiede tanti compiti da svolgere, ed è interesse dello sviluppatore che questi vengano automatizzati per potersi concentrare sugli aspetti specifici dell'ambito del software che viene costruito. I build systems sono gli strumenti preposti a tale scopo, in questo documento ne viene analizzato lo stato dell'arte dei principali che lavorano per la Java Virtual Machine (JVM), anche attraverso lo stile di programmazione con cui essi vengono implementati.

Dedicato alla mia famiglia che mi ha sempre sostenuto, in particolare a mio fratello Nicolò che mi ha aiutato a sbloccare quando il percorso per la laurea poteva interrompersi senza il suo conseguimento.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

1.1 Ecosistema Java/JVM

Il linguaggio *Java* è facile da comprendere e contiene poche astrazioni rispetto agli altri linguaggi di programmazione. La JVM (*Java Virtual Machine*) fornisce una base solida, portabile e ad alte prestazioni per l'esecuzione di Java o di altri linguaggi. Prendendole insieme, queste 2 tecnologie connesse forniscono una base su cui le aziende possono sentirsi sicure nella scelta di come impostare il proprio lavoro di sviluppo [2].

Sin dall'inizio di Java, è estremamente cresciuto un grande ecosistema di librerie e componenti di terze parti. Questo significa che un team di sviluppo può trarre enormi vantaggi dall'esistenza di driver e connettori per praticamente ogni tecnologia immaginabile, sia proprietaria che open source. È questo fatto che è stato uno dei principali motori dell'adozione delle tecnologie Java da parte di aziende e grandi imprese. I team di sviluppo sono stati in grado di sbloccare il loro potenziale facendo uso di componenti e librerie preesistenti [2].

1.1.1 Piattaforma Java

Java non è però il solo linguaggio di programmazione che viene utilizzato per scrivere un comune programma; seguendo il paradigma **"Write once, run anywhere"** lo si è reso indipendente dalla piattaforma hardware specifica (*platform-independent*), attraverso la quale si andrà ad eseguire il software sviluppato.

Per poter conseguire questo obiettivo è stata progettata la piattaforma Java (*Java Platform*) composta da 2 macro-componenti come mostrato nella Figura 1.1:

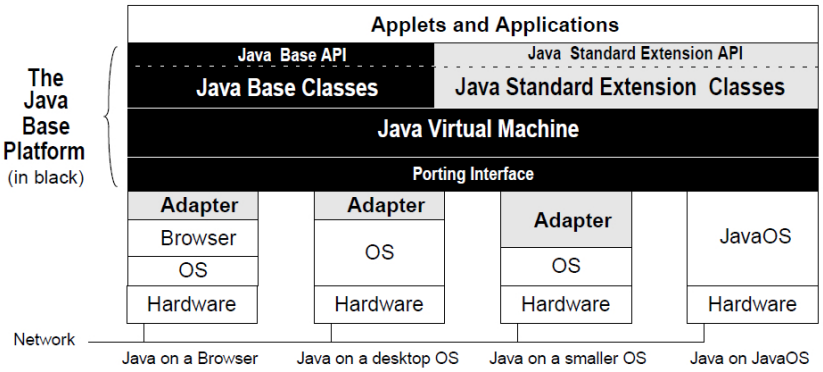

Figura 1.1: Struttura della Java platform e interoperabilità

- **Java Virtual Machine:** macchina astratta progettata per essere implementata sulla base dei processori e dei sistemi operativi attualmente esistenti [3].

- **Java Application Programming Interface (API):** sono una serie di interfacce standard per lo sviluppo di *applets* e *applicazioni* a prescindere dal tipo di sistema operativo sottostante [3].

Un ambiente Java è formato da 2 fasi di sviluppo, quello a tempo di compilazione (*compile-time environment*) dove viene prodotto un particolare file che viene definito *bytecode* di estensione `.class` e quello di runtime (*Java Runtime Environment*) (JRE). La piattaforma Java è rappresentata dal JRE [3] che attraverso il bytecode permette alla JVM di eseguire qualunque programma o applicazione che lo possa generare. Possono essere quindi definiti dei veri e propri linguaggi JVM adatti a tale scopo promuovendo **l'interoperabilità** di tutto l'ecosistema Java.

Essendo un ambiente *platform-independent*, la piattaforma Java può eseguire più lentamente del codice basato sulla macchina sottostante. Tuttavia i progressi nelle tecnologie dei compilatori e delle macchine virtuali stanno portando le prestazioni vicine a quelle del codice macchina senza minacciare la portabilità[1].

1.1.2 Java Virtual Machine

La Java Virtual Machine (JVM) è la pietra angolare della piattaforma Java. È il componente della tecnologia Java responsabile della sua indipendenza dall'hardware e dal sistema operativo sottostante, delle dimensioni ridotte del codice compilato e dalla sua capacità di proteggere gli utenti da programmi dannosi [10].

[1] https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

A differenza della piattaforma Java, la JVM è *platform-dependent*; ogni sistema operativo ne fa una propria implementazione per farla girare al meglio delle proprie performance e caratteristiche.

Il suo compito è quello di fungere da collo di bottiglia andando a nascondere l'architettura della macchina sottostante alle varie applicazioni sviluppate, ma occupandosi di un bytecode `file.class` introduce il concetto di **portabilità** all'ambiente Java, che consente ad ogni sistema operativo che utilizza la propria versione, di poter eseguire qualsiasi programma previsto per essa come mostra la Figura 1.1.

La JVM ha un motore di esecuzione (*execution engine*) che per far eseguire un file bytecode messo a disposizione, si avvale di 2 componenti che sono stati creati a tale scopo, l'*interpreter* e il *JIT compiler*.

Interpreter Legge il bytecode e lo interpreta istruzione per istruzione in linguaggio macchina. All'occorrenza lo interpreta più volte alla stessa maniera, riducendo le performance del sistema sottostante.

Just in time compiler (JIT) Controbilancia gli svantaggi dell'interpreter avvalendosi di un'analisi statistica delle parti di codice più utilizzate e traducendole all'atto dell'esecuzione, migliorando le performance.

È anche preparato per eseguire dinamicamente i file bytecode che potrebbero non essere disponibili all'inizio dell'esecuzione di un programma [1].

1.2 Build systems

1.2.1 Build systems: cosa sono e a che cosa servono

Un build system è un sistema di automazione per lo sviluppo di un software. Può gestire qualsiasi tipo di attività che comporti la traslazione di una forma di dati (l'input) in un'altra forma di dati (l'output) [8].

In qualsiasi ambiente di sviluppo software, un build system incontra i seguenti scenari di cui si deve occupare:

- Gestione delle dipendenze
- Compilazione software
- Testing
- Generazione della documentazione
- Generazione di report

- Assemblaggio artefatti
- Firma artefatti

1.2.2 Regole

Un build system deve rispettare certe regole da seguire:

1. Scalabilità
2. Correttezza
3. Usabilità

Scalabilità Dato un build system già aggiornato, per applicare un cambiamento al sistema e rieffettuare un processo di build dovrebbe richiedere un tempo proporzionale alla modifica svolta per consentirne il nuovo aggiornamento [9]. Questa regola è necessaria perchè l'aggiornamento è la più comune operazione che viene fatta durante lo sviluppo.

I progetti crescono nel tempo, ma le loro dimensioni in un singolo ciclo di sviluppo rimangono pressocchè le stesse come se si trattasse di un piccolo progetto. La *scalabilità* di un build system è rapportata soltanto a ciò che riguarda la modifica fatta e non alla dimensione dell'intero progetto, perchè il trascorrere l'attesa che l'intero processo venga aggiornato è tempo perso [9].

Correttezza Dato un build system in un determinato stato X, apportandogli una modifica e un successivo aggiornamento lo portano ad uno stato Y che se ne verrà poi annullata quella modifica, alla prossima esecuzione dovrà ritornare allo stato X.

La regola deve essere rispettata per assicurare che il programma usato per il *testing* di un progetto, sia effettivamente rappresentativo dei suoi file sorgenti. Non dovrebbe esserci alcuna preoccupazione per *side-effect* di funzioni, file o directory.

Con l'annullamento delle modifiche e l'esecuzione dell'aggiornamento di un processo di build, si deve poter recuperare l'albero dei sorgenti dello stato precedente, cosicchè possa essere provato un altro approccio senza che il sistema possa risentirne e finire in un cattivo stato. Questa deve essere una normale operazione che deve essere supportata dal build system e non un caso specifico da essere gestito manualmente dallo sviluppatore [9].

Usabilità Non ci devono essere modi differenti che lo sviluppatore debba ricordarsi per effettuare la build di un progetto.

Indipendentemente da come e dove la modifica viene fatta, un build system deve funzionare attraverso l'esecuzione di un solo comando da poter usare in qualsiasi momento [9].

Capitolo 2

Build systems per prodotti JVM

2.1 Imperativi

2.1.1 Apache Ant

Apache Ant ("Another Neat Tool") è un build system che fu creato nel 1999 per far fronte alle inconsistenze riscontrate nell'utilizzo di *make*, il build system che all'epoca era il più utilizzato fra tutti. Sebbene make seguisse molti concetti riguardanti un build system, c'erano difetti nella sua progettazione, come la dipendenza intrinseca dei comandi dalla piattaforma all'interno delle sue regole e l'architettura ricorsiva trovata in altri build system da esso derivanti [6].

Per risolvere questi difetti, Ant venne progettato come tool di sviluppo completamente indipendente dal sistema operativo previsto per la sua esecuzione, andando a superare il concetto di shell-based attraverso la sua collocazione nell'ambiente Java.

Funzionamento

Il codice 2.1 mostra un esempio di build script di Ant, viene prodotto attraverso un file XML chiamato `build.xml` i cui elementi principali sono i *project*, i *target*, i *task* e le *properties*, sottoforma di opportuni tag[1].

Il processo di build viene eseguito da linea di comando usando semplicemente la sola parola chiave `ant` che prenderà in esame il generico file `build.xml`, oppure facendo seguire all'argomento `-buildfile` il nome di un file XML che lo sviluppatore può creare come proprio build file alternativo[2]. Il solo utilizzo di un singolo file XML e la struttura delle dipendenze dei *target* che vedremo successivamente, danno ad Ant una caratterizzazione **imperativa** del build system, poichè la

[1] http://ant.apache.org/manual/using.html
[2] http://ant.apache.org/manual/running.html

costruzione di un progetto avviene in modo piuttosto simile all'esecuzione di un programma scritto attraverso un linguaggio imperativo come C o Java.

Project Ogni build file contiene un tag <`project`> che stabilisce la radice del progetto che si sta sviluppando. Ha 3 attributi, il nome del progetto `name`, il nome del target di `default` e la directory di base `basedir` da dove vengono calcolati tutti i percorsi dello sviluppo. Opzionalmente può essere inserito un ulteriore tag <`description`> per fornire una descrizione[3].

Target Un <`target`> definisce una funzionalità che il processo di una build vuole realizzare. Un target è un contenitore di *tasks* che cooperano per raggiungere uno stato desiderato all'interno del processo di build, come ad esempio l'inizializzazione, la compilazione e la distribuzione del progetto.

I target possono dipendere gli uni dagli altri sviluppando una catena di dipendenze che determina l'ordine della loro esecuzione. Questa specifica viene data dall'attributo `depends` che può indicare non solamente un singolo ma anche una lista di targets che Ant valuterà da sinistra verso destra. La cosa importante è che nell'estensione della catena, un target verrà eseguito una volta sola anche se verranno trovate multiple occorrenze di una dipendenza dallo stesso target.

Per un target bisogna per forza specificarne il `name` e si può aggiungere una `description` come attributo[4].

Task Un *task* è una parte di codice da eseguire, che rappresenta un'attività che si vuole svolgere. Ha una struttura comune che prevede l'utilizzo di un tag XML fatto nella maniera <`taskname attribute1="value1" attribute2="value2" ... /`>, con un nome e vari attributi i cui valori possono contenere riferimenti a delle proprietà che vengono risolte prima della sua esecuzione.

Properties Il tag <`property`> specifica le proprietà aggiuntive in un build file. La loro struttura è formata da coppie chiave-valore rappresentate da un `name` (chiave) e un *value* che una volta settate possono essere usate più volte ma non sono più modificabili, il nome è case-sensitive.

Una proprietà può essere utilizzata come stringa, per fissare un valore nell'*attributo* di un task attraverso la referenziazione nella forma `${name}`. Il modo più semplice per definire una proprietà è integrarla all'interno del build file, ma può essere fatto anche da riga di comando o in un file XML creato a parte[5].

[3] http://ant.apache.org/manual/using.html
[4] http://ant.apache.org/manual/targets.html
[5] http://ant.apache.org/manual/properties.html

Listato 2.1: Un build file di Ant

```
<project name="MyProject" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by
       compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source">
    <!-- Compile the Java code from ${src} into ${build}
        -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution">
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject
       -${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
       basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up">
    <!-- Delete the ${build} and ${dist} directory trees
        -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

2.2 Dichiarativi

2.2.1 Apache Ivy

Apache Ivy è un popolare sistema di gestione delle dipendenze (*dependency management*) che possono far parte di un progetto software, basato sulla semplicità e la flessibilità. Le sue principali caratteristiche sono[6]:

- l'integrazione con Ant seguendo i suoi principi di progettazione.
- non è intrusivo perchè comunemente usato per risolvere le dipendenze e copiarle nella directory `lib` di un progetto, dopdichè un build file non ha più il bisogno di dipendere da esso.
- forte gestione delle dipendenze transitive e dei conflitti, risolti per ciascuna di esse che viene incontrata, scrivendo semplicemente un unico file.
- compatibilità con repository esterni come quelli di Maven, dando la possibilità di costruirne uno proprio.
- produzione di reports HTML che danno una buona comprensione delle dipendenze presenti, e di grafi che mostrano una panoramica di quelle transitive e di eventuali conflitti.

Essendo stato concepito come sottoprogetto integrato con Apache Ant, utilizza elementi di significato completamente diverso rispetto a quelli di un build system, perchè non ha il compito di eseguire la build di un progetto, ma di trattare le sue dipendenze. I suoi elementi principali sono il descrittore di un modulo (*module descriptor*) e il modulo stesso (*module*).

Module descriptor

Il descrittore di un modulo (*module descriptor*) è quello che Ivy utilizza come elemento primario per definire i propri file rappresentativi. I più comuni module descriptors sono gli **Ivy Files** che sono file XML con una specifica sintassi, di solito denominati anche `ivy.xml`. Ma rientrano in questa categoria anche altri tipi di file come i POM di *Maven* grazie alla compatibilità con il loro formato metadata[7].

Nel codice 2.2 si ha un esempio di ivy file che definisce i suoi blocchi caratteristici a cominciare dall'`ivy-module` che è l'elemento radice. Essendo blocchi di tag XML danno a questo sistema una caratterizzazione di tipo **dichiarativo**, poichè

[6] http://ant.apache.org/ivy/features.html
[7] https://ant.apache.org/ivy/history/2.5.0/terminology.html

Listato 2.2: Un Ivy file

```
<ivy-module version="1.0">
    <info
        organisation="org.apache.ivy.example"
        module="find"
        status="integration"/>
    <configurations>
        <conf name="core"/>
        <conf name="standalone" extends="core"/>
    </configurations>
    <publications>
        <artifact name="find" type="jar" conf="core"/>
    </publications>
    <dependencies>
        <dependency name="version" rev="latest.
           integration" conf="core->default"/>
        <dependency name="list" rev="latest.integration"
            conf="core"/>
        <dependency org="commons-collections" name="
           commons-collections" rev="3.1" conf="core->
           default"/>
        <dependency org="commons-cli" name="commons-cli"
            rev="1.0" conf="standalone->default"/>
    </dependencies>
</ivy-module>
```

ogni blocco ha al proprio interno una determinata parte del lavoro che si deve occupare. Un module descriptor è necessario sia prima che dopo la pubblicazione di ogni revisione del modulo.

Durante il tempo di sviluppo (*development time*) che occorre tra le pubblicazioni, il descriptor aiuta nel gestire tutti i possibili cambiamenti delle dipendenze di un modulo. A tale scopo un ivy file può dichiarare dipendenze dinamiche per consentire una maggiore flessibilità di utilizzo. Pertanto a development time i file ivy sono dinamici (*dynamic*), perchè possono riprodurre risultati diversi nel tempo e sono considerati come *source files* conservati sotto un sistema di controllo della versione (*source control management*) come *Subversion* o *Git*[8].

Ad ogni pubblicazione si ha bisogno di un altro tipo di module descriptor per documentare le dipendenze di quella determinata revisione del modulo. Esse diventano fisse e risolte, pertanto il descriptor viene chiamato "risolto" (*resolved*) perchè produce sempre lo stesso risultato. I file ivy risolti sono comparabili agli artefatti archiviati in un repository e vengono conservati all'interno con essi[8].

Module

Un modulo è un'unità software autonoma e riutilizzabile che segue uno schema di controllo delle revisioni. In Ivy un modulo viene visto come una catena di revisioni ciascuna comprendente il descriptor che lo dichiara e uno o più artefatti. Ogni revisione di un modulo viene gestita in un repository[9].

Come si può vedere nel codice 2.2 il tag `ivy-module` definisce un modulo, dichiarandolo come radice di un file ivy, specificando la versione (`version`) del sistema compatibile. Un modulo è formato da altri tag che svolgono una parte specifica del lavoro:

- **info** fornisce l'identificazione e le informazioni di base sul modulo definito in un file ivy[10].
- **configurations:** definisce le configurazioni intese come modo di usare o costruire un modulo[11].
- **publications:** descrive gli artefatti pubblicati di un modulo e le eventuali configurazioni di appartenenza[12].
- **dependencies:** descrive le dipendenze di un modulo[13].

[8] https://ant.apache.org/ivy/history/2.5.0/ivyfile.html
[9] https://ant.apache.org/ivy/history/2.5.0/terminology.html
[10] https://ant.apache.org/ivy/history/2.5.0/ivyfile/info.html
[11] https://ant.apache.org/ivy/history/2.3.0/concept.html
[12] https://ant.apache.org/ivy/history/2.5.0/ivyfile/publications.html
[13] https://ant.apache.org/ivy/history/2.5.0/ivyfile/dependencies.html

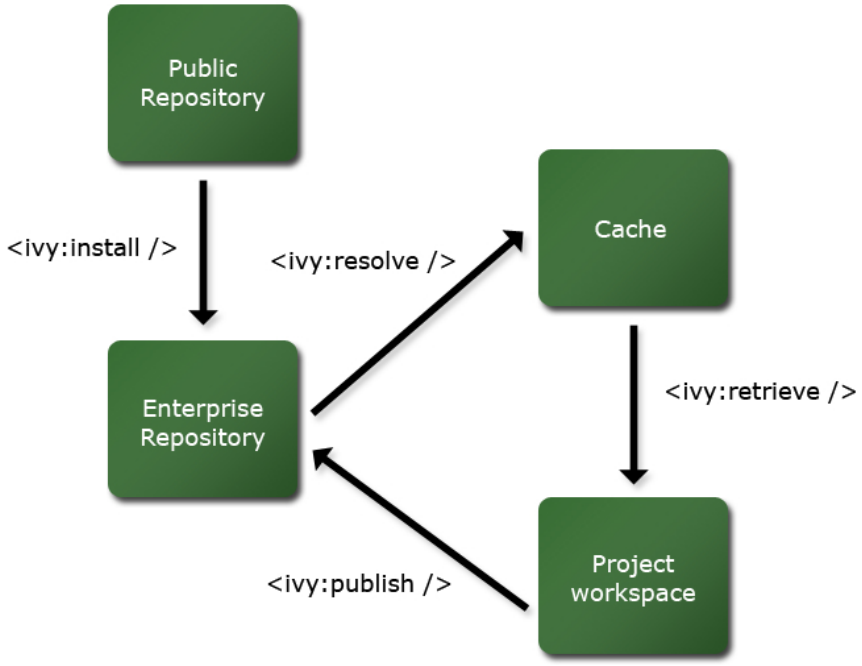

Figura 2.1: Chiamata e funzionamento degli ivy tasks

Funzionamento

Il modo principale e più frequente con il quale viene utilizzato Ivy è attraverso la chiamata da un *task* in un build file di *Ant* come viene mostrato nella Figura 2.1 che mostra il ciclo di funzionamento.[14].

Configurazione Ivy deve essere configurato per risolvere le dipendenze. Questa *configurazione* di solito viene fatta con un file d'impostazioni (*settings file*) chiamato `ivysettings.xml` che definisce un insieme di risolutori di dipendenze (*dependency resolvers*). Un resolver è in grado di cercare le dipendenze in un file ivy e di scaricare gli artefatti utilizzando informazioni specifiche come l'organizzazione, il nome e il numero di revisione di un modulo; il nome, tipo e l'estensione di un artefatto. La configurazione è anche responsabile di indicare quale resolver deve essere utilizzato per risolvere un determinato modulo[14].

Risoluzione Il tempo di risoluzione (*resolve*) è il momento in cui si risolve effettivamente le dipendenze di un modulo. Si accede al file ivy del modulo, poi per ogni dipendenza dichiarata nel file secondo le impostazioni definite nel settings file,

si chiede al resolver appropriato di trovare il modulo utilizzando anche una *cache* basata sul file system per evitare di scaricare dipendenze già in essa presenti.

Quando il modulo è stato trovato, una volta scaricato il suo file ivy, si controlla che la dipendenza abbia altre dipendenze, nel qual caso attraversa ricorsivamente il *grafo delle dipendenze*. Durante l'attraversamento viene effettuata la gestione dei conflitti (*confict management*) per impedire che l'accesso ad un modulo avvenga non appena sia possibile farlo. Una volta attraversato il grafo, i resolvers scaricano nella cache tutti gli artefatti corrispondenti alle dipendenze che già non erano presenti e che non sono state rimosse dai gestori dei conflitti (*conflict managers*).

Infine nella cache viene generato un *report XML* che permette ad Ivy di sapere quali sono tutte le dipendenze di un modulo, senza dover riattraversare il grafo[14].

Recupero Il recupero (*retrieve*) è l'atto di copiare gli artefatti dalla cache in un'altra directory. Questo viene fatto utilizzando un *pattern* che indica dove i file devono essere copiati e il report generato a tempo di resolve che consente al modulo di recuperarli conoscendo quali artefatti devono essere copiati.

In alternativa attraverso il report, si può costruire un percorso che permette di utilizzare gli artefatti richiesti, direttamente nella cache[14].

Pubblicazione In ultima istanza viene pubblicata (*publish*) la revisione di un modulo in un repository, in modo che diventi disponibile per future risoluzioni[14].

2.2.2 Apache Maven

Apache Maven è un build system principalmente pensato per raggiungere i seguenti obiettivi relativi a progetti Java, ma anche di altri linguaggi JVM come *Scala* o *Kotlin* [4]:

- rendere semplice il processo di sviluppo.
- fornire un build system uniforme.
- fornire un'informazione di qualità di un progetto.
- incoraggiare migliori pratiche di sviluppo.

Si basa su 2 aspetti principali:

- **Project Object Model (POM)**: un file script XML che contiene la maggior parte delle informazioni per creare un progetto nel modo desiderato.

[14] https://ant.apache.org/ivy/history/2.5.0/principle.html

- **Build Lifecycle**: il ciclo di vita della build di un progetto.

In questo modo lo sviluppatore, dovrà conoscere un limitato numero di comandi per eseguire un qualsiasi progetto e il POM garantirà che si ottengano i risultati voluti.

Tutto ciò fa si che Maven sia un build system di tipo **dichiarativo** perchè ogni "blocco" dichiarato nel file POM si occupa di svolgere una determinata funzionalità ed in più segue il paradigma **convention over configuration** definendo un layout standard per la struttura delle directory di un progetto.

POM

Maven costruisce la propria struttura di lavoro attraverso un'unità di base chiamata **Project Object Model** o semplicemente **POM**. E' un file XML che contiene le informazioni e i dettagli di configurazione che servono per costruire un progetto[15].

Da come si può vedere nell'esempio 2.3, un file `pom.xml` ha una serie di elementi chiave sottoforma di tag[16]:

- **project:** è l'elemento radice che rappresenta il progetto.
- **modelVersion:** è la versione del modello utilizzato dal POM.
- **groupId:** è l'identificatore univoco dell'organizzazione o del gruppo che ha creato il progetto. Si basa sul nome del dominio completo dell'organizzazione.
- **artifactId:** indica il nome di base univoco dell'artefatto principale generato da questo progetto. L'artefatto principale per un progetto è in genere un file JAR. Un tipico artefatto prodotto da Maven ha la forma <artifactId>-<version>.<extension> (ad esempio, `myapp-1.0.jar`).
- **version:** indica la versione dell'artefatto generato dal progetto. Può essere accompagnato dal descrittore `SNAPSHOT` il che indica che un progetto è nello stato di sviluppo.
- **name:** visualizza il nome utilizzato dal progetto. È spesso usato nella documentazione generata da Maven.
- **url:** indica dove è possibile trovare il sito del progetto. È spesso usato nella documentazione generata da Maven.
- **properties:** definisce le proprietà del progetto. Sono elementi referenziabili come variabili in qualsiasi parte del POM nella forma `${project.groupId}`.

[15] https://maven.apache.org/guides/introduction/introduction-to-the-pom.html
[16] https://maven.apache.org/guides/getting-started/index.html

Listato 2.3: Un esempio di POM in 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>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>my-app</name>
  <!-- FIXME change it to the projects website -->
  <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>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <pluginManagement><!-- lock down plugins versions to
        avoid using Maven defaults (may be moved to
       parent pom) -->
        ... lots of helpful plugins
    </pluginManagement>
  </build>
</project>
```

- **dependencies:** definisce al proprio le dipendenze del progetto. È una delle parti principali di Maven.

- **build:** definisce la struttura delle directory del progetto e la gestione dei plugins.

Coordinate di Maven La tripla costituita dagli elementi *groupId:artifactId:version* marca uno specifico posto in un *repository*, agendo come un sistema di coordinate di progetto. Questa forma può anche essere utilizzata per riferirsi agli artefatti come *dipendenze* ed è anch'essa facente capo al paradigma *convention over configuration* seguito da questo build system.

I 3 elementi indicano una versione specifica di un progetto facendo sapere a Maven di che cosa sta trattando e quando nel ciclo di vita di un software, verrà utilizzato[17].

Packaging Una volta determinate le coordinate di un artefatto, in Maven c'è un ultimo elemento da poter aggiungere in un POM che è formato dal tag <`packaging`> che ci dice come viene impacchettato l'artefatto. Di default il tipo utilizzato è il `.jar`, che viene predisposto anche in caso di mancata definizione; ce ne sono poi altri come ad esempio: `.pom`, `.war`, `.rar`, `.maven-plugin`.

Dipendenze Maven gestisce la lista delle dipendenze scaricandole e collegandole alla build di un progetto come propria caratteristica fondamentale. Inoltre introduce le dipendenze transitive, consentendo di concentrarsi esclusivamente sulle dipendenze richieste dal progetto.

Una dipendenza viene specificata da una serie di elementi [4]:

- **groupId**, **artifactId**, **version**: forniscono le coordinate Maven della dipendenza.

- **classifier:** serve per distinguere gli artefatti prodotti dallo stesso progetto.

- **type:** fornisce il tipo della dipendenza che determina l'estensione del file e il packaging dell'artefatto.

- **optional:** specifica se una dipendenza è opzionale.

- **scope:** determina le sole dipendenze appropriate per lo stato corrente di una build, in quali classpaths sono disponibili e limitandone anche la transitività. Ha diversi valori:

[17] https://maven.apache.org/pom.html

  - **compile:** la dipendenza nella compilazione è disponibile in tutti i classpath.
  - **provided:** è come `compile` ma indica al JDK o a un contenitore di fornirla in fase di runtime.
  - **runtime:** la dipendenza è richiesta solo per l'esecuzione inclusi i test.
  - **test:** la dipendenza è richiesta solo per la compilazione e l'esecuzione dei test, e non è transitiva.
  - **system:** simile a `provided` tranne per il fatto che bisogna fornire il JAR che la contiene esplicitamente. L'artefatto è sempre disponibile e non viene cercato in un repository.

- **systemPath:** viene usato solo in caso di scope `system`. Fornisce il percorso assoluto della dipendenza.

- **exclusions:** elenca gli artefatti che non sono ereditati quando si calcolano le dipendenze transitive.

Con le dipendenze transitive, il grafo delle librerie incluse può crescere rapidamente e ingrandirsi. Per questo motivo, esistono funzionalità aggiuntive che limitano le dipendenze incluse[18]:

- *Dependency mediation:* determina quale versione di un artefatto verrà scelta quando vengono rilevate più versioni come dipendenze. Maven sceglie la "nearest definition". Cioè, utilizza la versione della dipendenza più vicina al proprio progetto nell'albero delle dipendenze. Inoltre, se due versioni sono alla stessa profondità nell'albero delle dipendenze, viene utilizzata la prima che è stata dichiarata.

- *Dependency management:* consente allo sviluppatore del progetto di specificare direttamente le versioni degli artefatti da utilizzare quando vengono rilevati in dipendenze transitive o in cui non è stata specificata alcuna versione.

- *Excluded dependencies:* se il progetto X dipende dal progetto Y e il progetto Y dipende dal progetto Z, si può fare in modo che il progetto X possa escludere esplicitamente il progetto Z come dipendenza, utilizzando l'elemento `<exclusion>`.

[18] https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

- *Optional dependencies:* Se il progetto Y dipende dal progetto Z, si può marcare il progetto Z come dipendenza opzionale, utilizzando l'elemento <optional>. Se un progetto X dipende dal progetto Y, X dipenderà solo da Y e non dalla dipendenza opzionale Z di Y.

Repository Un repository in Maven contiene artefatti e dipendenze di vari tipi. Esistono esattamente 2 tipi di archivi[19]:

- **locale:** è una directory locale sul computer in cui viene eseguito Maven. Utilizzata sottoforma di *cache*, permette di scaricare e contenere artefatti temporanei che non sono stati ancora rilasciati.

- **remoto:** questi repository potrebbero essere impostati da terze parti per fornire i propri artefatti per il download. Maven mette a disposizione il proprio **Maven Central Repository** per scaricare le librerie utilizzate come dipendenze nei propri progetti. Oppure possono essere repository interni impostati su un file o un server HTTP all'interno di un'azienda, per condividere artefatti privati tra i team di sviluppo e per le release.

Plugins Maven è in realtà un framework per la raccolta dei plugins che svolgono le vere azioni all'interno del build system.

Un plugin è un artefatto che fornisce uno o più **goals** che gli danno la capacità di svolgere una determinata funzionalità (come ad esempio la creazione di un file JAR, o la compilazione di un test unit) il cui compito è quello di contribuire alla build di un progetto[20].

I principali plugins utilizzati da Maven, sono quelli che vengono denominati **Build plugins** che possono essere suddivisi in 2 specifiche di utilizzo:

- i cosiddetti "*core plugins*" che sono quelli di default corrispondenti alle fasi dei *build lifecycles*.

- quelli che devono essere configurati attraverso l'elemento <plugin> nell'opportuna sezione <build> del POM.

Maven permette di eseguire un determinato plugin anche insieme ad una o più eventuali fasi di lifecycle. La sintassi per l'esecuzione da linea di comando è: `plugin:goal`, dove il *goal* è l'azione concreta che svolge la funzionalità richiesta [4].

Inoltre, se un goal è collegato a una o più fasi di build, verrà chiamato in tutte quelle fasi. D'altro canto se una determinata fase non è associata ad alcun goal non verrà eseguita, però se ha più goal ad essa collegati, li eseguirà tutti. Per i plugins da configurare, viene mostrato l'esempio 2.4.

[19] https://maven.apache.org/guides/introduction/introduction-to-repositories.html
[20] https://maven.apache.org/guides/introduction/introduction-to-plugins.html

Listato 2.4: Un esempio di plugin in Maven

```
<plugin>
   <groupId>com.mycompany.example</groupId>
   <artifactId>display-maven-plugin</artifactId>
   <version>1.0</version>
   <executions>
     <execution>
       <phase>process-test-resources</phase>
       <goals>
         <goal>time</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
```

Ci sono poi ulteriori plugins chiamati **Reporting plugins**, che verranno eseguiti durante la generazione del sito e dovrebbero essere configurati nella sezione <reporting/> del POM.

Build Lifecycle

Maven mette a disposizione 3 Build Lifecycle standard:

1. `default`: gestisce la distribuzione del progetto.
2. `clean`: gestisce la pulizia del progetto.
3. `site`: crea la documentazione del sito del progetto.

Ognuno di questi lifecycles è definito da un elenco di *fasi* che al suo interno rappresentano uno stato del ciclo.

Ad esempio il lifecycle `default` comprende le seguenti fasi[21]:

- `validate`: validare la correttezza del progetto.
- `compile`: compilare il codice sorgente.
- `test`: testare il codice sorgente compilato utilizzando un adeguato framework di unit test.

[21] https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

- `package`: prende il codice compilato e lo impacchetta in un suo formato distribuibile, ad esempio un file JAR.
- `verify`: esegue eventuali controlli sui risultati dei test di integrazione per garantire il rispetto dei criteri di qualità.
- `install`: installa il package nel repository locale, per utilizzarlo localmente come dipendenza in altri progetti.
- `deploy`: distribuisce il package finale, copiandolo nel repository remoto per condividerlo con altri sviluppatori e progetti.

Queste fasi vengono eseguite in sequenza per completare il build lifecycle di `default`.

Lo sviluppatore esegue da riga di comando la fase che gli interessa svolgere per il proprio progetto; ad esempio se si vuole eseguire il comando `mvn verify`, vengono eseguite in ordine sequenziale, tutte le fasi precedenti (`validate`, `compile`, `package`, ect.) a `verify` ed in più altre fasi intermedie che qui non sono citate[22].

Si possono inoltre eseguire insieme lifecycles differenti come ad esempio per la chiamata `mvn clean deploy` che prima pulisce il progetto e poi lo ricrea con la distribuzione finale nel repository condiviso. Il comando funziona anche nello scenario di un progetto formato da più sottoprogetti, Maven attraverserà ogni sottoprogetto eseguendo prima `clean` e poi `deploy` comprese tutte le fasi antecedenti.

2.2.3 Bazel

Bazel è un build system open-source parte di *Blaze*, il sistema di build utilizzato da **Google**. Esso è stato concepito per operare su progetti Java, Android, iOS, C++, Go. Questo build system ricostruisce solo ciò che è necessario sfruttando le proprie caratteristiche di poter lavorare con l'esecuzione parallela, il caching locale e distribuito facendo anche uso di un'ottimizzata analisi delle dipendenze[23]. Nello specifico, alcune delle caratteristiche di Bazel comprendono[24]:

- orientamento all'alto livello. A differenza di altri build systems, evita chiamate a compilatori o linker esterni.
- mantenimento della storia passata. Mantiene lo storico dei comandi utilizzati per le build precedenti.
- multiplatform-based. Può costruire file binari e packages distribuibili nei più comuni sistemi operativi come Windows, macOS, Linux.

[22] https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
[23] https://www.bazel.build/
[24] https://docs.bazel.build/versions/4.0.0/bazel-overview.html

Bazel definisce il proprio file di configurazione principale chiamato `BUILD` che comunica quali elementi compilare, quali sono le loro dipendenze e come costruirli[25].

Il risultato è quello di generare il grafo delle dipendenze (*action graph*) e determinare le azioni che vengono svolte per creare l'output finale. È descritto tramite un *domain specific language* (DSL) chiamato **Starlark**, un dialetto di *Python* destinato all'uso come linguaggio di configurazione. Viene utilizzato sia come notazione per i file `BUILD` che per estendere Bazel come logica personalizzata per supportare nuovi linguaggi e compilatori. Nel codice 1 si ha un esempio di file script di Bazel in Starlark[26].

Bazel utilizza come elementi principali la *build rule* e i *build target*.

Rules

Una *rule* è l'implementazione di una funzione che registra le azioni da eseguire sugli artefatti di input per produrre una serie di artefatti di output. Un'*azione* è responsabile della reale operazione di build, per tale motivazione utilizza la maggior parte delle risorse necessarie per portarla a termine.

Bazel offre allo sviluppatore alcune rules di default che forniscono il supporto per certi linguaggi. Come si nota nel codice 1 la rule (`cc_library`) per *C++* definisce una libreria che prevede una lista variabile di argomenti. Nell'esempio considerato sono presenti il nome in `name`, un file sorgente in `srcs` e un header in `hdrs`. Una rule può anche essere implementata dallo sviluppatore con la parola chiave `rule`[27].

Una rule definisce al suo interno un file *target* che rappresenta un'unità di costruzione, rappresentata in maniera univoca dal valore assegnato a `name`. Ogni rule è in grado, a partire dall'elaborazione dei propri file di input, di generare file di output chiamati *generated files* o anche file derivati. Tali file possono essere a loro volta utilizzati come input per ulteriori rules definendo vere e proprie catene di rules.

Per le rules dei file binari (`cc_binary`) e dei test (`cc_test`) si prevedono analogamente nome e sorgente, con l'aggiunta dell'argomento `deps` che rappresenta la lista delle dipendenze.

Targets

Un *target* è l'unità di costruzione che viene generata da Bazel utilizzando file sorgenti ed eventualmente file derivati in accordo a quanto specificato nella relativa

[25] https://docs.bazel.build/versions/4.0.0/build-ref.html
[26] https://docs.bazel.build/versions/4.0.0/skylark/language.html
[27] https://docs.bazel.build/versions/4.0.0/skylark/rules.html

```
package(default_visibility = ["//visibility:public"])

cc_library(
    name = "hello-lib",
    srcs = ["hello-lib.cc"],
    hdrs = ["hello-lib.h"],
)

cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = [":hello-lib"],
)

cc_test(
    name = "hello-success_test",
    srcs = ["hello-world.cc"],
    deps = [":hello-lib"],
)

cc_test(
    name = "hello-fail_test",
    srcs = ["hello-fail.cc"],
    deps = [":hello-lib"],
)
```

Listing 1: Un file script di Bazel

rule.

Questo consente di determinare la catena delle dipendenze relative ad un progetto attraverso il grafo dei targets (*target graph*)[28].

Fasi di sviluppo

Quando viene eseguita una build o un test, Bazel si avvale di un modello di valutazione che è formato da 3 fasi da seguire[29]:

1. Caricamento

[28] https://docs.bazel.build/versions/4.0.0/build-ref.html
[29] https://docs.bazel.build/versions/4.0.0/skylark/concepts.html

2. Analisi

3. Esecuzione

Caricamento Nella fase di caricamento (*loading phase*) vengono caricati e valutati tutti i `BUILD` file di script che fanno parte della build di un progetto.

La concretezza della fase avviene attraverso l'istanziamento delle *rules* alla cui chiamata ne scaturirà l'aggiunta al grafo delle dipendenze (*target graph*) che qui viene prodotto[30].

Analisi La fase di analisi (*analysis phase*) esegue il codice delle rules e istanzia le *actions* da loro registrate.

Processa il target graph costruito nella fase di caricamento e genera un grafo delle azioni (*action graph*) che determina l'ordine della loro esecuzione. Il grafo delle azioni può includere sia gli artefatti che esistono come codice sorgente, sia quelli che in questa fase vengono generati ma che non sono menzionati nel build file[31]

Esecuzione La fase di esecuzione (*execution phase*) è quella dove vengono eseguite le *actions* nell'ordine stabilito dal grafo prodotto nella fase di analisi.

Le azioni invocano elementi eseguibili (compilatori, scripts) per leggere e scrivere gli artefatti di output. Se un file richiesto viene perduto oppure un comando non riesce a generare un output, la build fallisce.

In questa fase vengono anche eseguiti i *test*[31]

2.3 Funzionali

2.3.1 Gradle

Gradle nato nel 2008 e sviluppato da Gradleware, è un build system concepito con l'idea di prendere i migliori aspetti dei build systems imperativi come Ant e quelli dichiarativi come Maven e Ivy. Combina flessibilità ed estensibilità seguendo la filosofia "convention over configuration", supportando la gestione delle dipendenze [7].

Le principali caratteristiche di Gradle sono[32]:

- il supporto alle alte performance andando ad eseguire solamente i task di cui gli input e output sono stati cambiati.

[30] https://docs.bazel.build/versions/4.0.0/skylark/concepts.html
[31] https://docs.bazel.build/versions/4.0.0/glossary.html
[32] https://docs.gradle.org/current/userguide/what_is_gradle.html

- l'utilizzo di una *build cache* per abilitare il riutilizzo degli output di un task da esecuzioni precedenti o da macchine differenti.
- è facilmente estendibile per consentire lo sviluppo di task e modelli di build personalizzati.
- attraverso una *build scan* si forniscono informazioni sull'esecuzione di build per identificare i problemi, anche da poter condividere con altri sviluppatori.
- viene eseguito sulla *Java Virtual Machine* (JVM) ed è un vantaggio per gli utenti della piattaforma Java potendo usare le API nella logica di build.
- non si limita ai soli progetti JVM ma fornisce anche il supporto per quelli nativi (C/C++) ed è il build system ufficialmente utilizzato per **Android**.

Pur facendo propri i concetti presi dai suoi predecessori, Gradle definisce un suo approccio basato sulla scrittura attraverso un *Domain Specific Language* (DSL) che permette ad un build script di poter essere espresso come codice testabile e superare l'impiego di un file XML che ha un punto debole: è ottimo per descrivere i dati gerarchici, ma non riesce a esprimere il flusso del programma e la logica condizionale. Quando un build script cresce in complessità, mantenere il codice di build diventa un incubo [7].

Gradle mette a disposizione 2 linguaggi DSL per poter sviluppare i propri build scripts:

- **Groovy:**[33] linguaggio dinamico (con anche aspetti statici) per la Java platform, usato maggiormente per la scrittura degli scripts. Un build file è identificato dall'estensione `.gradle`.
- **Kotlin:**[34] linguaggio multiplatform object-oriented, staticamente tipizzato, che aggiunge nuove funzionalità al predecessore Java, da cui prende spunto. Un build file è identificato dall'estensione `.gradle.kts`.

In particolare, l'impiego di Kotlin con l'introduzione delle *lambda-expressions*, dei *nullable-types* e delle *extension functions* permette allo sviluppatore che è già a conoscenza del linguaggio Java, di utilizzare elementi di programmazione **funzionale** che rendono Gradle un build system di tale tipologia.

[33] https://groovy-lang.org/
[34] https://kotlinlang.org/spec/introduction.html

Un build script di Gradle è formato principalmente da 3 elementi[35]:

1. **Project:** rappresenta ciò che si vuole fare con Gradle. Non necessariamente deve essere una cosa che si vuol costruire.

2. **Task:** è l'unità atomica che rappresenta una parte del lavoro che viene svolto. Ogni progetto (`project`) è formato da uno o più `task`.

3. **Plugin:** estende la capacità di un progetto consentendo un più alto grado di modularizzazione, incapsulando la logica imperativa permettendo di essere il più dichiarativo possibile[36].

Project

Ogni build di Gradle è fatto da uno o più *project*. L'elemento `Project` è il tipo di un particolare oggetto che viene assemblato per ogni progetto che compone la build e delegato (*delegate object*) all'atto di esecuzione dello script che lo rappresenta. Ha una relazione uno a uno con il file di script che lo descrive a seconda del DSL scelto per la sua implementazione (Groovy o Kotlin)[37]. Un esempio di build file in Kotlin viene mostrato nel codice 2.5.

Dipendenze Il blocco `dependencies` dichiara le dipendenze che in Gradle vengono applicate per ambiti (scope) specifici. Un ambito viene anche chiamato *configuration* che viene identificato con un nome univoco ed è a sua volta definito da un plugin che ne dà un proprio set di dipendenze[38].

il plugin *application* applica implicitamente anche il *Java plugin* di base che definisce le seguenti configurazioni di dipendenze[39]:

- `compileOnly` dove vengono dichiarate le dipendenze che sono richieste a tempo di compilazione e non a runtime.

- `runtimeOnly` dove vengono dichiarate solamente le dipendenze richeste a runtime.

- `implementation` dove vengono dichiarate le dipendenze richieste in entrambi i tempi di sviluppo del progetto.

[35] https://docs.gradle.org/current/userguide/tutorial_using_tasks.html
[36] https://docs.gradle.org/current/userguide/plugins.html
[37] https://docs.gradle.org/current/dsl/org.gradle.api.Project.html
[38] https://docs.gradle.org/current/userguide/declaring_dependencies.html
[39] https://docs.gradle.org/current/userguide/java_plugin.html

Listato 2.5: Un esempio di build file di Gradle

```
plugins {
    application
    kotlin("jvm") version "1.3.50"
}
repositories {
    mavenCentral()
}
dependencies {
    implementation(kotlin("stdlib"))
    implementation("com.omertron:thetvdbapi:[1.6, 1.7]")
    runtimeOnly("javax.xml.ws:jaxws-api:+")
}
application {
    mainClassName = "it.unibo.ci.PrintSeriesKt"
}
```

`testCompileOnly`, `testRuntimeOnly`, `testImplementation` sono le configurazioni che hanno lo stesso comportamento precedentemente descritto, ma per dichiarare le dipendenze dei test.

In Gradle ci sono diversi tipi di dipendenze che si possono dichiarare, quella più comunemente utilizzata è il *modulo*. Un modulo è una parte di software che evolve nel tempo, di solito memorizzato in un repository[40]. L'utilizzo di una dipendenza può essere una dipendenza dichiarata nel build file o una *dipendenza transitiva* presente in un *grafo delle dipendenze* con la corrispondente configurazione. Gradle offre entrambe le capacità di visualizzazione delle dipendenze tramite scansioni delle build (**Build scans**) o come istruzioni da riga di comando[41].

La *risoluzione delle dipendenze* è un processo che consiste in 2 fasi che vengono ripetute fino al completamento del grafo delle dipendenze[42]:

1. quando una nuova dipendenza viene aggiunta al grafo, viene eseguita la **risoluzione dei conflitti** per determinare quale versione deve essere aggiunta al grafo.

2. quando una specifica dipendenza viene identificata come parte del grafo, se ne recuperano i metadati in modo che le sue dipendenze possano essere aggiunte a loro volta.

[40] https://docs.gradle.org/current/userguide/dependency_management_terminology.html
[41] https://docs.gradle.org/current/userguide/viewing_debugging_dependencies.html
[42] https://docs.gradle.org/current/userguide/dependency_resolution.html

Listato 2.6: I vincoli delle dipendenze di Gradle

```
dependencies {
    implementation("org.apache.httpcomponents:httpclient
       ")
    constraints {
        implementation("org.apache.httpcomponents:
           httpclient:4.5.3") {
            because("previous versions have a bug
               impacting this application")
        }
        implementation("commons-codec:commons-codec:1.11
           ") {
            because("version 1.9 pulled from httpclient
               has bugs affecting this application")
        }
    }
}
```

Quando si esegue la risoluzione delle dipendenze, Gradle gestisce due tipi di conflitti[43]:

- conflitti di versione: quando due o più dipendenze richiedono una determinata dipendenza ma con versioni diverse.
- conflitti di implementazione: quando il grafo delle dipendenze contiene un modulo che fornisce la stessa implementazione o capacità.

È abbastanza comune che i problemi con la gestione delle dipendenze riguardino le dipendenze transitive, quelle di cui ne ha bisogno un componente soltanto perchè un'altra sua dipendenza ne ha il bisogno. Gradle risolve questo problema con l'introduzione dei vincoli sulle dipendenze (*dependency constraints*) come mostrato nel codice 2.6[44].

Un vincolo permette di definire la versione o un range di versioni sia per le dipendenze dichiarate che per quelle transitive, quando Gradle prova a risolvere una dipendenza nella versione di un modulo, tiene in considerazione tutti questi aspetti che trova, selezionando la versione di numero maggiore (*highest version*) che soddisfa tutte le condizioni, contrariamente a Maven che invece ne prende

[43] https://docs.gradle.org/current/userguide/dependency_resolution.html
[44] https://docs.gradle.org/current/userguide/dependency_constraints.html

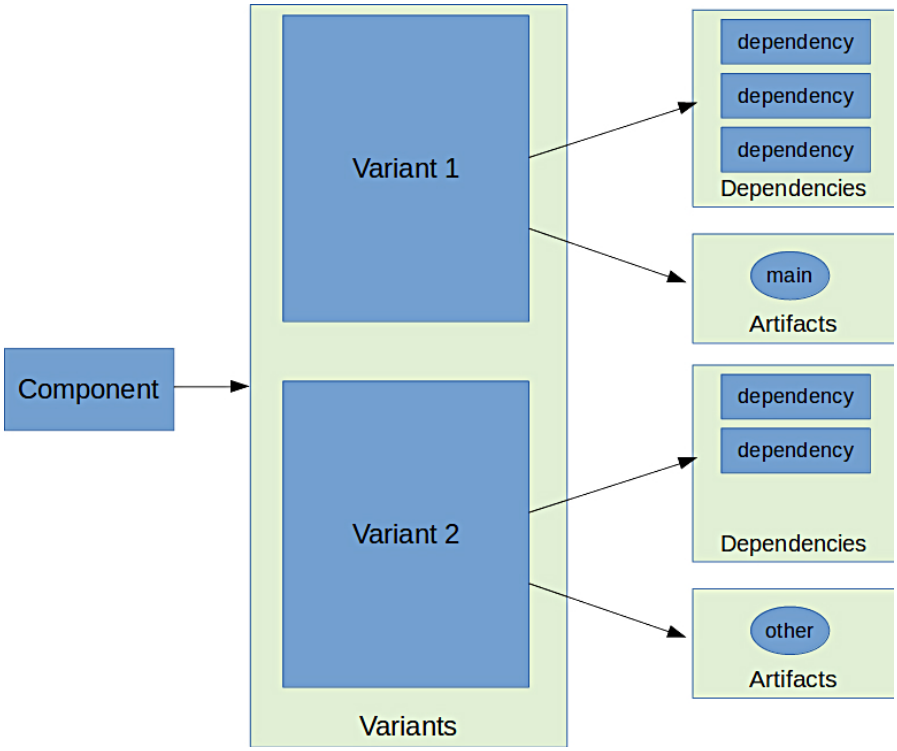

Figura 2.2: Modello di gestione delle dipendenze di Gradle

quella più vicina (*nearest first*) che è dipendente dall'ordine di ritrovamento. Se non viene trovata alcuna versione, Gradle fallisce la build con un errore che mostra i conflitti fra le dipendenze. Come per le dipendenze dichiarate, i vincoli sono legati ad un ambito di configurazione e quindi possono essere definite come parti di una build. I vincoli sulle dipendenze sono supportati solamente con l'utilizzo del *Gradle Module Metadata* per specificare una dipendenza[45].

Per i *conflitti di implementazione* tra le dipendenze, Gradle concepisce come proprio motore di gestione la consapevolezza delle varianti (**variant aware**) come mostrato nella Figura 2.2[46].

In aggiunta al modulo di una dipendenza con le sue coordinate, viene introdotto il concetto di variante (*variant*) che rappresenta una diversa "vista" con la quale un componente (versione di un modulo) viene pubblicato alle stesse coordinate group-artifact-version (GAV). Questo livello intermedio, che associa artefatti e dipendenze alle varianti invece che direttamente al componente, consente a Gradle di modellare correttamente per cosa ogni artefatto viene utilizzato.

[45] https://docs.gradle.org/current/userguide/dependency_constraints.html
[46] https://docs.gradle.org/current/userguide/variant_model.html

Per sapere come fa Gradle a scegliere quale variante utilizzare quando ce ne sono più di una, la selezione avviene per mezzo di *attributi* che forniscono la semantica alle varianti e aiutano a raggiungere un coerente risultato di risoluzione. Con la selezione consapevole delle varianti (*variant aware selection*), Gradle distingue tra due tipi di componenti[47]:

- componenti locali, costruiti dai sorgenti, per i quali le varianti sono mappate alle configurazioni uscenti.
- componenti esterni, pubblicati sui repository, il cui modulo è stato pubblicato con il modello metadata di Gradle e le varianti sono già supportate, oppure con i metadati Ivy / Maven da cui le varianti vengono derivate.

Repositories Nel blocco `repositories` possono essere dichiarati uno o più repository tra cui quelli più popolari come **Maven Central** e **JCenter** che essendo delle collezioni di moduli consentono a Gradle di sapere dove andare a recuperare un modulo per utilizzarlo nella build, attraverso la notazione `group:name:version` specificata nella configurazione di una dipendenza[48].

I repository possono essere remoti o presenti localmente in una directory utilizzando il percorso del file system per potervi accedere. Gradle supporta diversi formati di repository tra cui quelli di Maven e Ivy secondo l'aspetto che quando viene dichiarato il modulo di una dipendenza, viene cercato il rispettivo *module metadata file* che lo descrive (`.pom` per Maven, `ivy.xml` per Ivy) prima di effettuare il download dell'artefatto del modulo. Viene anche dato un ordine di ricerca degli artefatti se viene previsto un utilizzo multiplo di differenti repositories, dando la precedenza al *Gradle module metadata* (file `.module`) che è stato progettato per supportare il proprio modello di gestione delle dipendenze[49].

Task

Il lavoro che viene svolto da Gradle in un progetto (`project`) viene definito in uno o più *task*. Il task è l'elemento che rappresenta l'unità atomica di lavoro eseguita da una build. Può essere ad esempio la compilazione di classi, la creazione di un JAR, la generazione di Javadoc, la pubblicazione di un archivio in un repository (ecc..)[50].

Un task consiste in[51]:

[47] https://docs.gradle.org/current/userguide/variant_model.html
[48] https://docs.gradle.org/current/userguide/declaring_repositories.html
[49] https://docs.gradle.org/current/userguide/declaring_repositories.html
[50] https://docs.gradle.org/current/userguide/tutorial_using_tasks.html
[51] https://docs.gradle.org/current/userguide/what_is_gradle.html

Listato 2.7: Definizione dei task in Gradle

```
tasks.register("taskX") {
    dependsOn("taskY")
    doLast {
        println("taskX")
    }
}
tasks.register("taskY") {
    doLast {
        println("taskY")
    }
}
```

- **Azioni:** le parti in cui viene effettivamente svolto il lavoro.
- **Inputs:** valori, file e directory su cui le azioni operano.
- **Outputs:** file e directory che le azioni generano o modificano.

L'esempio del codice 2.7 mostra come possono essere definite le azioni (metodo `doLast`) che un determinato task svolge e anche come dichiarare eventuali dipendenze (metodo `dependsOn`) da altri task presenti. É possibile in un task aggiungere una dipendenza anche prima dell'esistenza del task da cui dipenderà, in tal caso si parla di *lazy dependsOn*.

Grafo delle dipendenze Gradle garantisce che i task sono eseguiti nell'ordine delle dipendenze tra essi e ognuno viene eseguito una sola volta. Questo fa si che venga creato un *grafo diretto aciclico* (DAG) tra i task ed è una delle ragioni che permette la sua flessibilità. Ci sono build systems che costruiscono il grafo delle dipendenze mentre eseguono i propri task, ma Gradle come propria caratteristica lo fa *prima* di arrivare alla loro esecuzione[52].

Gradle Wrapper Il modo più raccomandato di eseguire una build in Gradle è quello di utilizzare quello che viene definito **Gradle Wrapper**. É uno script che invoca una dichiarata versione di Gradle, scaricandola in anticipo se necessario.

Per ogni build file è consentita l'esecuzione di un task `wrapper` che genera tutti i *Wrapper files* necessari nella directory del progetto[53]:

[52] https://docs.gradle.org/current/userguide/build_lifecycle.html
[53] https://docs.gradle.org/current/userguide/gradle_wrapper.html

- `gradle-wrapper.jar` contiene il codice per scaricare la distribuzione di Gradle.
- `gradle-wrapper.properties` un file di proprietà responsabile del comportamento del Wrapper a runtime.
- `gradlew`, `gradlew.bat` gli script di Linux e Windows per eseguire una build con il Wrapper.

I maggiori benefici dell'utilizzo del Wrapper sono quelli di standardizzare un progetto su una data e corretta versione di Gradle garantendone un'esecuzione affidabile e controllata, ma soprattutto di eseguirlo senza dover effettuare alcuna procedura d'installazione a parte.

Plugin

Per usufruire di tutte le funzionalità utili che Gradle mette a disposizione dello sviluppatore, si può applicare un elemento apposito chiamato `Plugin` che in un build file può essere dichiarato nel blocco predisposto `plugins`.

Applicare un plugin vuol dire estendere la capacità di una build aggiungendo nuovi task e configurazioni che incapsulano la logica imperativa di un build script rendendolo maggiormente dichiarativo. Consente anche un più alto grado di modularizzazione, migliorandone la comprensibilità e l'organizzazione.

Per utilizzare la logica incapsulata in un plugin, Gradle deve effettuare 2 step. *Risolverlo* cercando la versione corretta del jar che lo contiene, aggiungendolo al classpath dello script di appartenenza e poi *applicarlo* ad un *target object* che solitamente è il `Project`[54]. I plugin principali facenti parte della distribuzione di Gradle vengono automaticamente risolti, pertanto il loro impiego avviene in un solo step.

Java Application Plugin Il codice 2.5 mostra l'utilizzo del plugin `application` che è quello principale per lo sviluppo di build su applicazioni Java.

L'applicazione di questo plugin, comporta l'implicita applicazione anche del più basilare `Java Plugin` che introduce l'elemento di `Source Set` rappresentante un insieme di file sorgenti e risorse logicamente raggruppati per le loro dipendenze e classpaths. I *source sets* vengono collocati in directory separate le cui 2 principali definite da Gradle, sono `main` per il codice dell'applicazione Java e `test` per i relativi test appositi[55].

Questo plugin introduce una serie di propri task tra cui:

[54] https://docs.gradle.org/current/userguide/plugins.html
[55] https://docs.gradle.org/current/userguide/java_plugin.html

- `run` per eseguire l'applicazione Java.

altri aggiunti dal `Java Plugin`:

- `compileJava` per compilare i sorgenti dell'applicazione.
- `processResources` per copiare le risorse nell'apposita directory.
- `jar` per assemblare il file JAR dell'applicazione.
- `javadoc` per generare la documentazione API dell'applicazione.
- `test` per eseguire i test con opportuni tools tipo *JUnit*.

compresi anche quelli relativi a *test* e *source sets*. Ed infine quelli inseriti dal `Base Plugin`, anch'esso implicitamente applicato[56]:

- `clean` per eliminare il contenuto della build directory.
- `assemble` per assemblare tutti gli archivi nel progetto.
- `check` per svolgere i test e i controlli della qualità del codice.
- `build` per eseguire la completa build del progetto.

Build lifecycle

I build scripts vengono valutati ed eseguiti in 3 fasi diverse che formano il cosiddetto **Gradle Build lifecycle**[57]:

1. **Inizializzazione**
2. **Configurazione**
3. **Esecuzione**

Seppur il concetto di *build lifecycle* possa ricondurre alle fasi di Maven, quelle di Gradle sono differenti. Maven utilizza le sue fasi per dividere l'esecuzione della build in più steps, in Gradle un ruolo similare è quello svolto dal grafo dei task, sebbene in modo meno flessibile[58]. I *lifecycle tasks* definiti nel `Base Plugin` (`assemble`, `check`, `build`) di Gradle hanno una concezione vagamente similare al build lifecycle di Maven[59].

[56] https://docs.gradle.org/current/userguide/base_plugin.html
[57] https://docs.gradle.org/current/userguide/build_lifecycle.html
[58] https://docs.gradle.org/current/userguide/what_is_gradle.html
[59] https://docs.gradle.org/current/userguide/base_plugin.html

La fase di *inizializzazione* determina quali progetti entrano a far parte di una build visto che Gradle permette lo sviluppo sia di build singolari che multi-progetto. Questo viene specificato da un file d'impostazioni (*settings file*) convenzionalmente chiamato `settings.gradle(.kts)` che Gradle definisce accanto ai propri build scripts ed è richiesto in un multi-progetto nella radice (*root project*) della gerarchia che lo compone. Il settings file viene eseguito durante la fase di inizializzazione ed è invece previsto opzionalmente per i progetti singolari. Inoltre aldilà che una build sia singolare o multipla viene creata in questa fase un'istanza dell'oggetto `Project` per ognuno dei progetti che ne prende parte.

La fase di *configurazione* costruisce il grafo dei task (*task graph*) di una build, determina quali task devono essere eseguiti e in quale ordine. In questa fase vengono configurati i *project objects* e vengono eseguiti i build scripts di tutti i progetti che sono parte di una build. Un altro aspetto importante è che il codice viene valutato ogni volta che una build viene eseguita e non si vedranno i cambiamenti che accadono nella fase di esecuzione.

La fase di *esecuzione* esegue i task creati e configurati durante la fase di configurazione. Vengono eseguite soltanto le *azioni* dei task il cui nome è passato come argomento da linea di comando e il sottoinsieme di tutte le sue dipendenze.

2.3.2 Sbt

Sbt ("simple build tool") è un build system costruito per progetti Scala e Java. La sua funzionalità è quella di poter costruire un progetto con più versioni di Scala differenti. È basato sulla velocità e sull'estensibilità[60]. Sbt possiede le seguenti caratteristiche[61]:

- supporta progetti in più linguaggi come *Scala* e *Java*.
- da la possibilità di effettuare test diversi con ScalaCheck, ScalaTest e JUnit.
- è basato sul *read-eval-print-loop* (REPL) di Scala con classi e dipendenze nel loro percorso specifico.
- si ha la possibilità di includere dipendenze esterne sottoforma di progetti situati nei vari repositories come Git
- utilizza il parallelismo per eseguire i propri tasks e test.

Sbt ha come elementi principali per la propria definizione la *build definition*, il *task graph* e gli *scopes*.

[60] https://www.scala-sbt.org/index.html
[61] https://www.scala-sbt.org/1.x/docs/

Build definition

Sbt definisce `build.sbt` come proprio file di script. E' formato da un insieme di *subprojects* che sono definiti da una o più coppie chiave-valore[62]

```
lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalaVersion := "2.12.7"
  )
```

Listing 2: Un subproject di sbt

Il codice 2 mostra un esempio di subproject. `name` è una chiave e "hello" un valore. Le coppie chiave-valore sono elencate nella direttiva `.settings` e sono chiamate anche *setting expressions*. Fra di loro ci sono anche le cosiddette *task expressions* che sono composte dai seguenti 3 elementi:

1. sul lato sinistro è presente la *chiave*.
2. al centro un *operatore*, ad esempio: (`:=`).
3. sul lato destro il *corpo*.

Keys sbt ha 3 tipi di chiavi da poter suddividere uno script:

- `SettingKey[T]`: una chiave per un valore calcolato una volta sola ad ogni subproject e mantenuto tale.
- `TaskKey[T]`: una chiave per un valore, chiamato *task*, che può essere ricalcolato ogni volta, potenzialmente con presenza di *side-effects*.
- `InputKey[T]`: una chiave di un task che ha per input, argomenti da linea di comando.

Una `TaskKey[T]` definisce un *task* che è un'operazione come `compile` o `package`, di cui `T` ne rappresenta il tipo. Può ritornare `Unit` (che è come `void` per Scala), oppure un valore relativo al task. Ad esempio `package` è un `TaskKey[File]` e il suo valore è il file jar che crea.

Ogni volta che comincerà l'esecuzione di un task, ad esempio digitando `compile` sul prompt di Sbt, esso rieseguirà tutti i task relativi al comando esattamente una sola volta.

[62] https://www.scala-sbt.org/1.x/docs/Basic-Def.html

Le coppie chiave-valore di Sbt che descrivono un subproject, possono riferirsi a un valore fissato come stringa per una propria impostazione (*setting*) tipo il nome; però possono riferirsi anche al codice eseguibile di un task come `compile`, anche se il codice eventualmente restituisce una stringa che dovrà essere rieseguita ogni volta. Una data chiave si riferisce sempre a un task o a un semplice setting.

Task graph

Piuttosto che vedere i `settings` come coppie chiave-valore, una migliore analogia è quella di pensarli come fossero un *grafo aciclico diretto* (DAG) dove gli archi denotano ciò che prima accade. Questo viene semplicemente chiamato *task graph*.

Terminologia Si esprimono i seguenti concetti chiave[63]:

- Setting/Task expression: entry all'interno della direttiva `.settings`.
- Key: il lato sinistro di una setting expression. Può essere una `SettingKey[A]`, una `TaskKey[A]`, o una `InputKey[A]`.
- Setting: definito da una setting expression con `SettingKey[A]`. Il valore è calcolato una volta durante il caricamento.
- Task: definito da una setting expression con `TaskKey[A]`. Il valore è calcolato ogni volta che viene invocato.

Un `build.sbt` utilizza la direttiva `.value` per dichiarare una dipendenza interna da un task o da un setting. L'esempio 3 mostra come poter definire l'esecuzione dei task all'interno del rispettivo file di script.

Una cosa importante di Sbt è che non impone un ordine di esecuzione dei task senza che vengano esplicitamente inserite come dipendenze da parte dello sviluppatore. Questa modalità di esecuzione ricorda similarmente quella che viene adottata da *Ant*. Nell'esempio i task `update` e `clean` verranno eseguiti in un ordine determinato da Sbt o anche in parallelo[64].

Scopes

Precedentemente abbiamo specificato che una chiave come ad esempio `name` corrisponde a una sola entry nella mappa di coppie chiave-valore utilizzata da Sbt. Con l'introduzione di quello che viene chiamato *scope* ("ambito") si può fare in modo che una chiave abbia un diverso valore associato in più di un contesto rappresentante.

[63] https://www.scala-sbt.org/1.x/docs/Task-Graph.html
[64] https://www.scala-sbt.org/1.x/docs/Task-Graph.html

```
val scalacOptions = taskKey[Seq[String]]("Options for the Scala compiler.")
val update = taskKey[UpdateReport]("Resolves and optionally
retrieves dependencies, producing a report.")
val clean = taskKey[Unit]("Deletes files produced by the build
, such as generated sources, compiled classes, and task caches.")

scalacOptions := {
  val ur = update.value  // update task happens-before scalacOptions
  val x = clean.value    // clean task happens-before scalacOptions
  // ---- scalacOptions begins here ----
  ur.allConfigurations.take(3)
}
```

Listing 3: Definizione dei task in sbt

Esempi concreti si possono avere quando si ha a che fare con progetti multipli in una build definition, dove una chiave può avere un valore differente per ogni subproject. Oppure nello stesso modo se vogliamo compilare file sorgenti e test.

Dato uno scope è possibile avere solamente un valore attribuito alla stessa chiave.

Sbt processa la lista di `settings` contenuta nel file di script generando una mappa di coppie chiave-valore, dove una volta determinato lo scope di una chiave, queste prenderanno il nome di *scoped keys*. Spesso uno scope è implicito o di default, tuttavia se l'impostazione è errata va inserito lo scope desiderato esplicitamente nel file di script.

Uno scope è definito come nel codice 2.8. È formato da un *project/subproject* (`projA`), una *configuration* (`Compile`) e un *task value* (`console`) dove il task è (`scalacOptions`)[65].

Una configuration descrive un grafo di tutte le dipendenze di un progetto come sorgenti e packages caratterizzate dal proprio classpath.

Dipendenze

Le librerie di dipendenze sono aggiunte in 2 modi[66]:

- dipendenze non gestite (*unmanaged dependencies*): sono file jar collocati nella cartella `lib`, posizionati nel classpath di un progetto.

[65] https://www.scala-sbt.org/1.x/docs/Scopes.html
[66] https://www.scala-sbt.org/1.x/docs/Library-Dependencies.html

Listato 2.8: Un esempio di scope in Sbt

```
scalacOptions in (
  Select(projA: Reference),
  Select(Compile: ConfigKey),
  Select(console.key)
)
```

- dipendenze gestite (*managed dependencies*): vengono configurate nella build definition e scaricate da appositi repositories.

In particolare le managed dependencies vengono trattate da risolutori esterni come **Coursier** che è scritto interamente in Scala per supportare la programmazione funzionale, ma anche **Ivy**.

Per dichiarare una dipendenza esterna viene utilizzata la keyword `libraryDependencies`; può essere fatto in diversi modi come ad esempio:

```
libraryDependencies += groupID % artifactID % revision
```

dove `groupID`, `artifactID`, e `revision` sono stringhe. Oppure:

```
libraryDependencies += groupID % artifactID % revision %
    configuration
```

dove si può indicare una `configuration` espressa come stringa o valore definito dall'utente tipo `Test` o `Compile`. Il simbolo % crea un `ModuleID` dalle stringhe. O anche:

```
val libraryDependencies = settingKey[Seq[ModuleID]]("
   Declares managed dependencies.")
```

dove rendiamo una dipendenza come chiave `settingKey` propria di Sbt.

I modi descritti precedentemente indentificano una dipendenza ben definita dallo sviluppatore. Per evitare questo tipo di approccio, è possibile apoggiarsi ad un build system esterno come *Ivy*, che è in grado di recuperare automaticamente l'ultima versione dell'artefatto in accordo con costrutti esplicitamente specificati[67].

2.3.3 Leiningen

Leiningen è un build system che automatizza lo sviluppo di progetti scritti con il linguaggio **funzionale** *Clojure* JVM-based. Clojure è un dialetto del linguaggio

[67] https://www.scala-sbt.org/1.x/docs/Library-Dependencies.html

Lisp creato per integrare la componente di scripting all'interno di un linguaggio dinamico e funzionale dandone un interpretazione basata sulla JVM.

Leiningen è caratterizzato dalle seguenti funzionalità[68]:

- creare, compilare e distribuire progetti.
- risolvere e gestire le dipendenze necessarie allo sviluppo.
- esegue in piena configurazione REPL.
- permette di integrare l'interoperabilità con altri build system come Maven.
- caricare progetti sottoforma di librerie in opportuni repositories come *Clojars*.

Leiningen ha come elementi principali per la propria definizione i *projects* e utilizza il concetto di *template* per la costruzione dinamica dei propri progetti [5].

Projects

Leiningen definisce `project.clj` come proprio file di script. È definito dal nome e dalla descrizione del progetto che si costruisce, dalle librerie di dipendenze ad esso necessarie e dalla versione di Clojure utilizzata, tutti specificati dai relativi percorsi. Nel codice 4 viene mostrato un esempio di build file di Leiningen.

```
(defproject my-stuff "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "https://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "https://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.8.0"]]
  :main ^:skip-aot my-stuff.core
  :target-path "target/%s"
  :profiles {:uberjar {:aot :all}})
```

Listing 4: Un file script di Leiningen

[68] https://github.com/technomancy/leiningen/blob/stable/doc/TUTORIAL.md

Dipendenze Anche Leiningen prendendo ispirazione da Maven, è basato sul paradigma **convention over configuration**[69]:

- fa in modo di costruire un progetto sfruttando un albero di directory preimpostate.
- le librerie utilizzate hanno identificatori (artifact group, artifact id) e versioni basate sulle convenzioni di Maven.

Leiningen recupera gli artefatti visti come dipendenze, attraverso repositories esterni tipo **Clojars** che è il riferimento principale della comunità di Clojure oppure Maven Central.

Per poter fare ciò, nella direttiva `:dependencies` vengono definite le dipendenze come si può vedere nel codice 5. Con l'istruzione `clj-http` si ottiene una duplice funzione di collegarsi a Clojars come repository di riferimento per recuperare gli artefatti e specificare allo stesso tempo il groupID e l'artifactID come avviene su Maven; per completare la tripla che identifica l'artefatto si indica successivamente la versione sottoforma di stringa. Altrimenti si possono esplicitare il nome del gruppo e dell'artefatto separati da / prima della versione. Sono anche previste le versioni `-SNAPSHOT` per i progetti in fase di sviluppo che non sono ancora stati rilasciati.

Profili In un build file di Leiningen vengono definiti i `:profiles` ("profili") che sono usati per specificare configurazioni personalizzate nel progetto. I profili di Leiningen influenzano il comportamento di un progetto, in questo modo è possibile separare diversi setup in modo tale da non includere ad esempio le dipendenze relative a framework di test durante le fasi di sviluppo.

Templates

Leiningen utilizza il concetto di template per la costruzione dinamica dei propri progetti. Esistono quattro diversi templates [5]:

- **template:** un meta-template utilizzato per definire nuovi templates.
- **default:** un template di progetto generico per le librerie.
- **app:** un template di progetto per le applicazioni.
- **plugin:** un template di progetto per un plugin di Leiningen.

Nel caso in cui lo sviluppatore intendesse distribuire le proprie librerie, è possibile utilizzare il comando `lein deploy clojars` che consente di caricarle nel repository online Clojars.

[69] https://github.com/technomancy/leiningen/blob/stable/doc/TUTORIAL.md

```
:dependencies [[org.clojure/clojure "1.8.0"]
               [clj-http "2.0.0"]]
```

Listing 5: Le dipendenze in Leiningen

Capitolo 3

Conclusioni

L'obiettivo di questa tesi è stato quello di analizzare il maggior numero di build systems in linguaggi JVM-based. Tutti i build system che sono stati descritti in questo documento hanno un metodo di funzionamento molto simile basato sulla scrittura di un build file che definisce tutti i task che eseguono i vari compiti preposti. Con la modernizzazione di questi sistemi si è cercato sempre più di facilitare la scrittura del build file allo sviluppatore tramite l'utilizzo di linguaggi DSL che consentono di programmare direttamente le attività da svolgere all'interno di un sistema, senza l'ausilio di script file stringenti che vincolano la configurazione a cui lo sviluppatore si debba per forza attenere.

Bibliografia

[1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 280–290. ACM, 1998.

[2] David Flanagan. *Java in a nutshell - a desktop quick reference: covers Java 5.0 (5. ed.)*. O'Reilly, 2005.

[3] Douglas Kramer. The java platform. *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[4] Apache Maven. Apache maven. *URL http://maven. apache. org/. Accessed*, pages 11–07, 2014.

[5] Mark McDonnell. *Quick Clojure*. Apress, 2017.

[6] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The evolution of ANT build systems. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, May 2010.

[7] Benjamin Muschko. *Gradle in action*, volume 27. Manning, 2014.

[8] William Del Ra. Software build systems: principles and experience by peter smith. *ACM SIGSOFT Softw. Eng. Notes*, 36(4):33–34, 2011.

[9] Mike Shal. Build system rules and algorithms. *Published online (2009). Retrieved July*, 18:2013, 2009.

[10] Frank Yellin and Tim Lindholm. The java virtual machine specification, 1996.