ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

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

Gradle, Kotlin e lo Sviluppo di un'Applicazione Multipiattaforma

Tesi di laurea in
Programmazione ad Oggetti

Relatore
Chiar.mo Prof. Andrea Omicini

Candidato
Mariano Caldara

Correlatore
Dott. Giovanni Ciatto

Seconda Sessione di Laurea
Anno Accademico 2018-2019

Abstract

Gestire un applicazione multipiattaforma è una operazione onerosa vista la diversità delle tecnologie e delle piattaforme con le quali fa fronte, tuttavia è diventato l'obbiettivo e il fine di molti progetti, che in termini di guadagno possono godere di un minor codice implementativo e di una maggiore utenza finale, raggiungibile in tempi di produzione minore. Gralde che è uno strumento navigato nella gestione del codice, ha collaborato con JetBrains nella costruzione di un progetto più grande nominato Kotlin Multipiattaforma, che coinvolge non soltanto lo strumento in sè ma anche la piattaforma di sviluppo intelliJ IDEA, creando un connubio perfetto per la creazione dei progetti multipiattaforma. L'obbiettivo di questo documento non è soltanto riconoscere questi strumenti ed esplicarli ma avere anche un ruolo guida nella costruzione di un progetto di questo tipo, infatti lo studio di questi argomenti è stato unito alle sperimentazioni su diversi progetti, infatti nei capitoli finali sarà evidente il contributo personale.

Un giorno mio padre venne a prendermi a scuola e mi portò al mare. Era troppo freddo per fare il bagno, così ci sedemmo sull'asciugamano a mangiare la pizza. Quando tornai a casa avevo le scarpe piene di sabbia e sporcai il pavimento della mia stanza. Avevo sei anni neanche me ne accorsi. Mia madre mi sgridò per tutto quel casino, invece lui non era arrabbiato. Disse che miliardi di anni fa, la rotazione della terra e il movimento degli oceani avevano portato quella sabbia in quel punto della spiaggia e che io l'avevo portata via. Disse: "Tutti i giorni cambiamo il mondo", ma non riesco a pensare in quanti giorni di quante vite riuscirei a portare una scarpa piena di sabbia a casa fino a svuotare la spiaggia, finché questo non faccia la differenza. Tutti i giorni cambiamo il mondo, ma perché il cambiamento sia significativo ci vuole più tempo di quanto ne abbiamo. Non accade mai niente in una sola volta. E' lento, è metodico, è estenuante, non tutti abbiamo lo stomaco per farlo.

tratto da **Mr. Robot**

Ringraziamenti

Ringrazio il mio relatore Andrea Omicini e il mio correlatore Giovanni Ciatto per avermi ispirato e assegnato il progetto. Infine, ringrazio i miei amici e la mia famiglia per avermi supportato.

Indice

Elenco delle figure

Listati

Capitolo 1

Introduzione

Con la continua crescita di dispositivi e sistemi sempre più diversi tra loro, è nata la necessità di creare applicazioni eseguibili su più piattaforme. Questo bisogno, dettato dalle aziende nel compito di raggiungere più utenti possibili, si è tramutato nell'obbiettivo, per i programmatori, di creare un codice più universale possibile e che potesse essere distribuito su più sistemi.

Questa tendenza porta diversi benefici per gli sviluppatori come riusabilità, costi e tempi di produzione migliori, mentre per gli utenti finali avranno una maggiore possibilità di godere di un'applicazione simile o addirittura uguale, indipendentemente dal sistema.

Kotlin è un linguaggio che al più si appresta a dare vita ad un'applicazione multipiattaforma, ultimamente questo linguaggio ha raggiunto una notevole popolarità, il motivo è riconducibile a due eventi precisi, il supporto ufficiale di Google per Android (Cleron, 2017), e l'integrazione in Spring Boot 2 (Deleuze, 2017). Tuttavia, il suo successo nel campo dei progetti multipiattaforma è dovuto soprattutto a Gradle e JetBrains, che offrono un'integrazione completa con questo linguaggio.

L'obbiettivo di questa tesi non è soltanto mostrare il lato implementativo, che comporrà comunque la parte fondamentale, ma fornire anche una panoramica di questi strumenti, cercando quindi di avere un ruolo guida nel creare un'applicazione multipiattaforma con Gradle e Kotlin.

Il fine di questo progetto inoltre è analizzare e comprendere questo tipo di tecnologia nel suo processo di sviluppo, soffermandosi soprattutto sulla fase di testing e deploy.

Capitolo 2

Stato dell'arte

Durante la programmazione si riconduce, spesso, nel codice la chiave del successo di un ottimo progetto software. Nonostante la qualità di quest'ultimo sia un elemento imprescindibile nello sviluppo del codice, una visione globale e ingegneristica suggerirebbe l'adozione di ulteriori strumenti per la produttività del programmatore. Uno su tutti è Gradle, oggetto di studio approfondito di questa tesi, è uno strumento che si pone come sistema per automatizzare diverse fasi dello sviluppo del codice, ricalcando la logica di sistemi già navigati come Apache Ant e Apache Maven, apportando però diversi miglioramenti su più punti. Per comprendere al meglio questi miglioramenti, in questo capitolo si analizzerà lo strumento in questione prima in un'ottica generale e successivamente nell'ambito degli applicativi multipiattaforma.
Per conferire una visione più chiara dello strumento verrà prima introdotto Kotlin, nello specifico Kotlin DSL, come linguaggio perfetto per la sua configurazione.
La divisione tra i due argomenti non sarà netta ma combinata per dare una visione più chiara e omogenea.

2.1 Kotlin, e i motivi del successo

Kotlin, è un linguaggio general-purpose, multi-paradigma e fortemente tipizzato; si ispira apertamente a Java, garantendone una completa compatibilità, infatti

può essere compilato su JVM, e a livello di sintassi si presenta anche molto simile. Questa somiglianza e compatibilità ha permesso una veloce diffusione e adozione da parte di aziende del calibro di Google, Amazon, Netflix; tuttavia il ruolo di questo linguaggio non è soltanto da copione di Java, ma introduce rispetto a quest'ultimo numerose funzionalità che sopperiscono ai limiti del linguaggio sviluppato da Microsystems.
Per esempio, in primis, si presenta come un linguaggio più sintetico e veloce rispetto al linguaggio di Oracle.

Listato 2.1: HelloWorld in Java

```
class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello World!");
        }
}
```

Listato 2.2: HelloWorld in Kotlin

```
package helloworld
fun main() {
        println("Hello World!")
}
```

Un esempio comune e lampante si può ritrovare nella creazione delle classi, difatti una classe in Kotlin, automaticamente assegna ad ogni attributo detto "property" i suoi metodi getters e setters in automatico.

Listato 2.3: Creazione class Person in Java

```
class Person {
    private String name;
    private String surname;
    public Person() {

    }
```

```
    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSurname() {
        return surname;
    }
    public void setSurname(String surname) {
        this.surname = surname;
    }
}
```

Listato 2.4: Creazione class Person in Kotlin

```
class Person(var name: String, var surname: String)
```

Inoltre, il linguaggio di JetBrains mira a superare i limiti di Java, infatti gode di un'architettura di base più solida e sicura, che lo permette di ridurre le eccezioni a runtime. Nel concreto, le variabili non possono essere, per impostazione predefinita, istanziate a null. Questa caratteristica detta *null-safety*, permette al compilatore di verificare ogni *null-reference* prima di eseguire il codice.
Le potenzialità presentate sono solo una minima parte di ciò che questo linguaggio può offrire, tuttavia quelle che sono fondamentali per lo studio di un'applicazione multipiattaforma sono Kotlin DSL e Kotlin Multipiattaforma, che verranno analizzate nei paragrafi a seguire.

2.2 Linguaggi DSL e Kotlin Script DSL

Un linguaggio general purpose come Kotlin visto fin'ora, è un tipo di linguaggio adatto a programmare ogni tipo di applicazione, indipendentemente dal suo dominio, tuttavia diverse funzionalità permettono a Kotlin di creare un linguaggio domain specific, appunto specifico per un determinato dominio applicativo. Kotlin DSL, come il resto dei linguaggi domain-specific, è espressivo e facilmente leggibile, principalmente molto simile ad una libreria *API*, differendosi da quest'ultima per l'uso di una determinata grammatica o struttura ben definita. Le funzionalità che permettono di rendere Kotlin un linguaggio più semplice da comprendere anche a chi non programma sono:

- Espressioni lambda
- Funzione High-Order
- Espressioni lambda con ricevitore

Nei paragrafi seguenti si analizzerà Kotlin DSL nello specifico poiché ampiamente adoperato nella configurazione dello strumento Gradle.
Seguiranno esempi pratici per comprendere al meglio le funzionalità elencate.
Le espressioni *lambda* sono essenzialmente funzioni anonime che possono essere trattate come valori, accettano un numero predefinito di parametri e ritornano un unico valore automaticamente.

Listato 2.5: Lambda label

```
/*
Logica di base:
( Lista di parametri ) -> Tipo di valore restituito
*/

() -> Unit //Nessun parametro, restituisce Unit,
   equivalente di void
```

```
(Int) -> Int //accetta intero e restituisce intero

(String) -> Unit //Stringa per Unit

val lambda : (String) -> Unit = { println(it) }

lambda("Hello")
```

Essendo appunto valori, le lambda possono essere sia parametri che valori di ritorno di una funzione, e in questi casi si introduce il concetto di High-Order Function. Per ridurre ulteriormente la sintassi, se la lambda in questione è l'ultimo parametro, questa può essere spostata fuori dalle parentesi tonde e se è anche l'unico parametro, le parantesi possono essere omesse.

Listato 2.6: High Order Function

```
class DependencyHandler {
    fun compile(coordinate: String){
        //aggiunge coordinate di una collezione
    }
    fun testCompile(coordinate: String){
        //aggiunge coordinate di una collezione
    }
}

fun dependencies(action:
    (DependencyHandler) -> Unit)
    : DependencyHandler {
        val dependencies = DependencyHandler()
        action(dependencies)
        return dependencies
}
```

```
//dependencies({...})
dependencies {
    it.compile("") //it istanza di DependencyHandler
    it.testCompile("")
}
```

Nel caso del codice 2.6 la funzione *dependencies* accetta come parametro una lambda composta da come parametro di input un *DependencyHandler* e ritorna *Unit* (il correspettivo di *void* in Java), la funzione infine ritorna un'istanza di DependencyHandler. I linguaggi DSL, mirano ad eliminare le ripetizioni, infatti il codice presentato può essere ancora migliorato, eliminando *it* mediante l'uso di una lambda con ricevitore. Questa funzionalità permette di scrivere il corpo di un'espressione lambda omettendo il riferimento *it* o *this* dell'oggetto in input, modificando quindi le proprietà di un oggetto. La stesura finale, può essere un esempio di configurazione delle dipendenze di un progetto tramite Kotlin DSL e Gradle.

Listato 2.7: Stesura finale Kotlin DSL

```
fun dependencies(action: DependencyHandler.() -> Unit)
: DependencyHandler {
    val dependencies = DependencyHandler()
    dependencies.action()
    return dependencies
}

dependencies {
   compile("")     //this.compile
   testCompile("")
}
```

In conclusione Kotlin DSL sta diventando sempre più popolare nella configurazione di Gradle (Carbonnelle, 2019), e mira a sostituire Groovy[1], apportando diver-

[1]linguaggio DSL JVM-based dinamico adoperato largamente per la compilazione di Gradle

se migliorie. Infatti, questo linguaggio che è stato ideato grazie al connubio tra JetBrains e Gradle, permette una completa integrazione con gli IDE; supportando funzionalità base che Groovy non offre, come l'auto-completamento e il refactoring (Beams, 2016).

2.3 Automazione del progetto

Le tecniche di automazione nella produzione di artefatti durante lo sviluppo software di un progetto sono diventate sempre più raffinate e determinanti per un progetto di successo. Queste operazioni, come produzione di codice binario, packaging dello stesso, testing e produzione di documentazione venivano in un primo momento eseguite dal programmatore medianti stesura di script[2], ma negli ultimi anni vengono affidate a strumenti di *Build Automation* più completi come Gradle. L'obbiettivo centrale di questi strumenti è la possibilità di distribuire il codice in modo consistente e ripetibile, minimizzando l'intervento umano e gli errori durante la fase di compilazione.
Tra gli strumenti di *build automation* Gradle occupa un market share del 19% secondo il sondaggio di Paraschiv, 2017 ed è ampiamente impiegato tra i programmatori come si denota dal grafico 2.1. In questo capitolo veranno illustrati gli elementi cardine di questa tecnologia e di questo strumento.

2.3.1 Gradle, API principali

Project è l'elemento su cui verte il sistema Gradle, infatti è la rappresentazione logica di ciò che deve diventare artefatto, ovvero l'obbiettivo finale del lavoro di un programmatore in termini di sviluppo software. A livello tecnico, questo elemento si rappresenta come l'API principale, su cui vertono tutti i metodi di Gradle e quindi le sue funzionalità, come creazione di *Tasks* o gestione delle dipendenze, questi e gli altri metodi a disposizione per il programmatore sono visibili nell'interfaccia

[2]make, *GNU Make Manual*

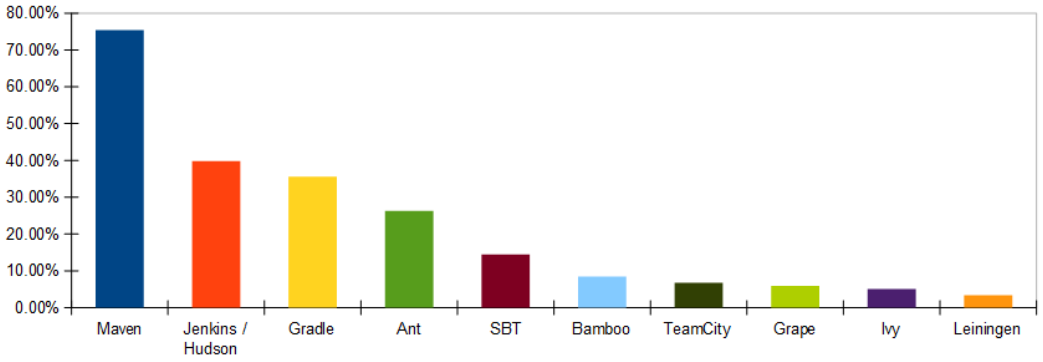

Figura 2.1: Statistiche di popolarità tra i *build tools*

2.2 e sono accessibili attraverso il file di build[3]. Questo tipo di file, non è unico, e durante la fase di build per ognuno di questi, Gradle, seguendo i comandi descritti nello script, istanzia uno o più *Projects* e i *Tasks* relativi. I primi, infatti, possono essere visti, come un'insieme dei secondi, e non sono altro che singole operazioni specifiche, eseguibili anche autonomamente, che susseguite tra loro portano alla creazione del artefatto. Questo susseguirsi di operazioni definisce il flusso di svolgimento dell'intero processo di *build*, e si modella su un grafo aciclico diretto (DAG), che evidenzia le dipendenze tra i *Task*, infatti spesso questi lavorano su artefatti di altri, in particolare nella struttura DAG, i nodi corrispondono alle unità di lavoro *Tasks* (per esempio *produzione di documentazione*), mentre gli archi diretti rappresentano le loro dipendenze (*Task A depends on Task B*). Un nodo conosce soltanto il suo stato di esecuzione, e durante il flusso, può essere eseguito solo una volta, indipendentemente se è oggetto di più dipendenze. Come si nota dalla sua interfaccia, figura 2.5, dove sono rappresentanti soltanto i metodi principali, '*DoFirst*' e '*DoLast*' permettono di eseguire le azioni in due momenti diversi. Per comprendere al meglio questi metodi, e il loro funzionamento e necessario introdurre e comprendere il ciclo di vita della *build* di Gradle. Ogni *build* di Gradle si articola in tre fasi, nella prima detta appunto fase di inizializzazione, lo strumento

[3] *gradle.build* file di configurazione di Gradle scritto in DSL con Groovy o in Kotlin DSL descritto in 2.2

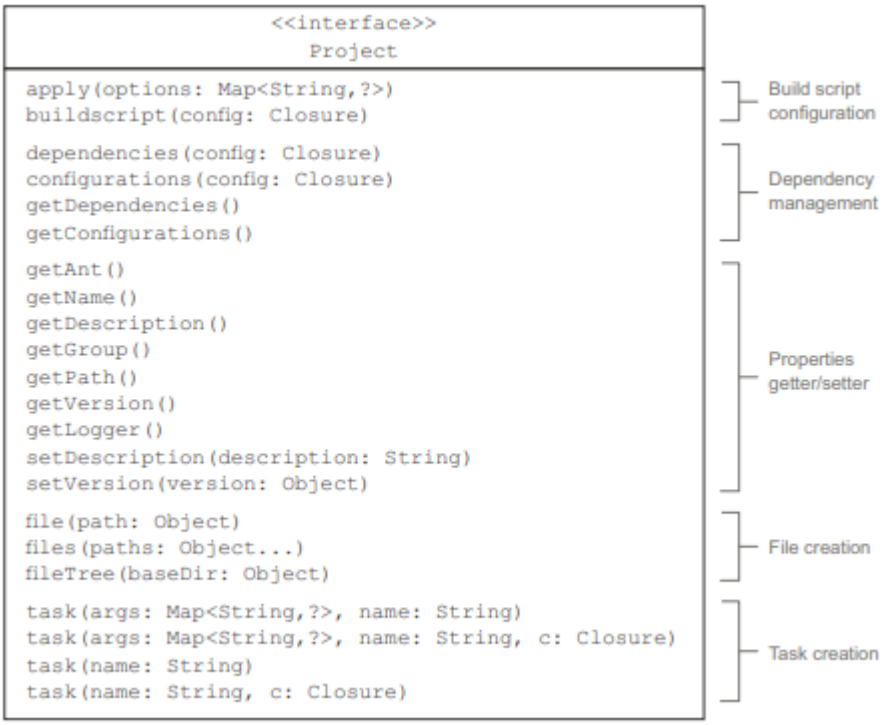

Figura 2.2: Interfaccia di *Project*, con i metodi principali

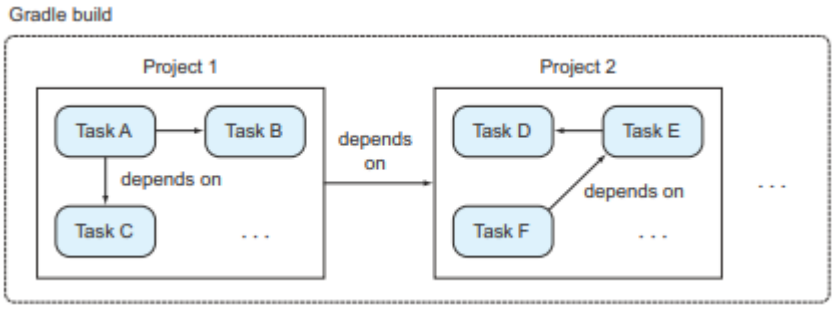

Figura 2.3: *Tasks* e *Project*

in questione una volta identificato il progetto o i progetti su cui lavorare, analizzando il file *settings.gradle*[4] istanzia per ognuno di questi un *org.gradle.api.Project*. Segue, quindi, la fase di configurazione; Gradle in questo frangente riconosce i pro-

[4]File di confiugrazione di Gradle, utile alla costruzione della gerarchia dei progetti

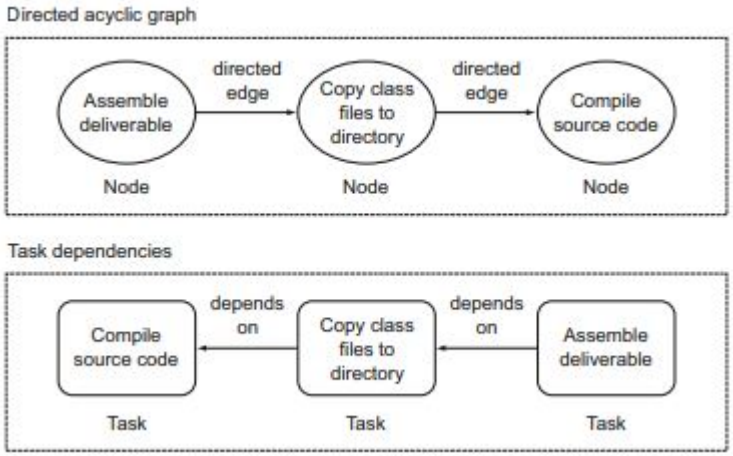

Figura 2.4: Rappresentazione DAG dei *tasks*

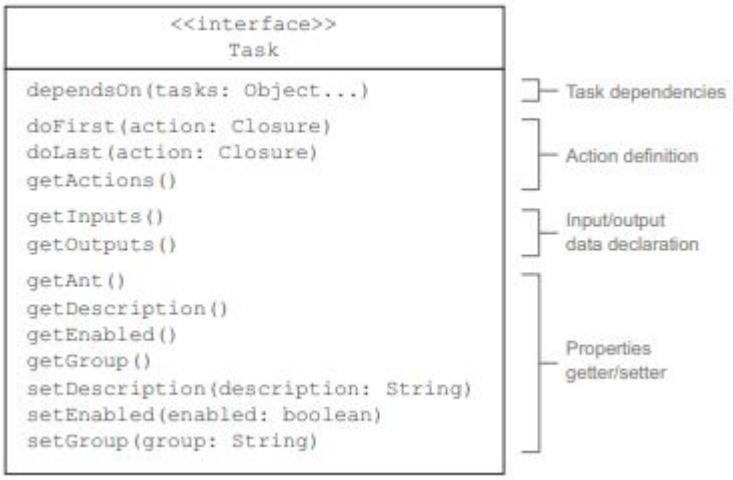

Figura 2.5: Interfaccia con metodi principali di *Task*

getti utili alla build, e costruisce, mediante i relativi *build-script*, il già citato DAG. Il tempo di configurazione rispetto ad altri strumenti di *build-automation* è generalmente ridotto, questo perchè questo strumento introduce una nuova tecnologia denominata *"Configuration on-demand"*(*Authoring Multi-Project Builds*), che gli permette di configurare solo i progetti e i task utili alla *build*. Nell'ultima fase, vengono eseguiti i *Task* tenendo conto delle loro dipendenze, basandosi ovviamente sul DAG creato nella fase precedente.

In conclusione, segue una possibile creazione di *Task* in Kotlin DSL(2.2), concretizzando anche le definizioni dei metodi *"DoLast()"* e *"DoFirst()"*; in particolare, queste funzioni permettono di influenzare le azioni del *task* nel tempo di *build*, eseguendo l'azione rispettivamente all'inizio o alla fine della fase di esecuzione.

Listato 2.8: Creazione di *task*

```
tasks.register("firstTask") {
    group = "Try"
    description = "Trying doLast, doFirst"
    doFirst {
        println("This is executed first during
        the execution phase")
    }
    doLast {
        println("This is executed last during
        the execution phase")
    }
    println("This is executed during
    the configuration phase")
}
```

2.3.2 Gestione delle dipendenze

La maggior parte dei progetti dipende da librerie esterne, inizialmente, quando i programmi di *build-automation* non erano ancora popolari, era compito del programmatore includere e gestire le dipendenze esterne, aumentando i tempi di produzione del software. Una visione più ingegneristica, invece, vede la gestione delle dipendenze incluse nella definizione di *build*. In questo modo, Gradle è a conoscenza di ogni singola dipendenza, e riesce a gestire al meglio anche la transitività di queste, operazione che manualmente risulterebbe eccessivamente laboriosa, soprattutto in programmi complessi. Un esempio concreto può essere la libreria Hibernate, come si denota dalla fig. 2.6, il programmatore non soltanto deve co-

noscere le dipendenze transitive, ma anche le loro versioni per evitare conflitti. Un

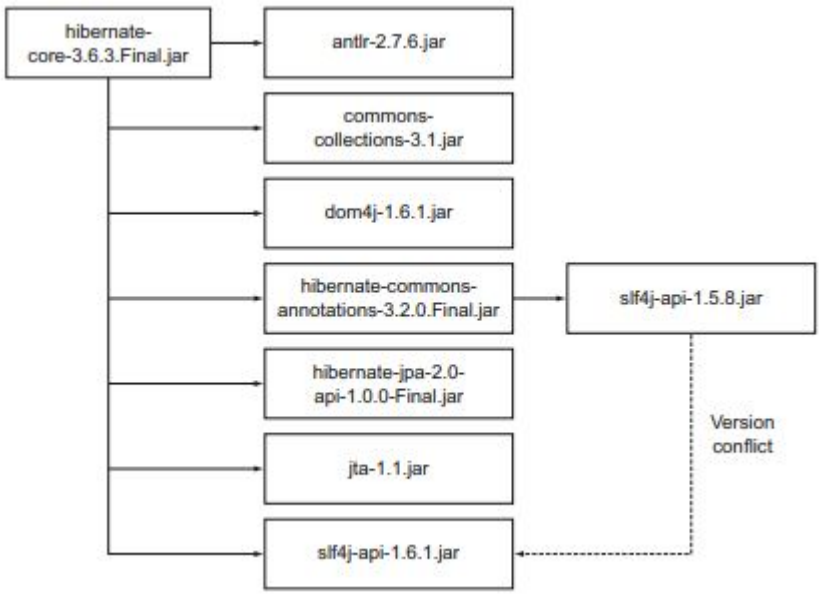

Figura 2.6: Grafico delle dipendenze della libreria Hibernate

esempio di configurazione, in Kotlin DSL, delle dipendenze è il listato 2.9, dove vengono introdotti tre concetti fondamentali, configurazione (*api*, *implementation*, *testImplementation* sempre riferendosi al listato 2.9), dipendenza (*junit:junit:4.12*) e infine repositery (*mavenCentral()*). Le configurazioni sono un elemento chiave in Gradle, rappresentano un gruppo di dipendenze, che si differenzia l'uno dall'altro in base a diversi fattori, come la fase della *build* nella quale si applicano al progetto, o la possibilità di accedervi all'utente finale o meno.

Listato 2.9: Esempio di configurazione delle dipendenze

```
dependencies {
    api("com.android.tools.build:gradle:3.5.1")
    implementation("com.android.tools.build
    :gradle:3.5.1")
    testImplementation("com.android.tools.build
    :gradle:3.5.1")
}
```

```
repositories {
    mavenCentral()
    maven {
        url = uri("https://repository-cdn.liferay.com/
        nexus/content/groups/public")
    }
}
```

Possono essere costruite ad-hoc dal programmatore, ma generalmente vengono introdotte attraverso *plugin*[5], come quelle più basilari introdotte con Java plugin:

- *implementation*
- *api*
- *compileOnly*
- *runtimeOnly*
- *testImplementation*
- *testCompileOnly*
- *testRuntimeOnly*

Il primo può essere considerato quello di default, le dipendenze in questione sono disponibili in tutte le fasi della build, ma non sono esposte all'utente finale, quindi non verranno aggiunte al classpath compilato. Nell'aggiornamento a Gradle 3.0 come annunciato da Google al I/O '17 (*Speeding Up Your Android Gradle Builds (Google I/O '17)*), *implementation* ha sostituito *compile*. *Api*, è del tutto uguale a *implementation*, tranne per il motivo che aggiunge transitività alla dipendenza, quindi la rende accessibile anche al cliente finale del progetto. Nel caso in cui il programmatore volesse limitare l'uso di un modulo a tempo di compilazione o a

[5]Elemento che permette di estendere le funzionalità di Gradle, sarà oggetto di approfondimento nella prossima sezione

runtime, esistono le configurazioni rispettive, *compileOnly* e *runtimeOnly*. Ricalcando la logica di *implementation*, *testImplementation* rende i moduli accessibili soltanto durante la compilazione e durante il tempo d'esecuzione dei test. *testCompileOnly* e *testRuntimeOnly* invece espongono i moduli rispettivamente durante la fase di compilazione o di *runtime* dei test.
Nella terminologia di Gradle, le librerie esterne, generalmente distribuite sottoforma di JAR, vengono denominate moduli, e sono identificabili attraverso un vettore di tre parametri: *group*, *name* e *version*. Riferendosi al listato 2.9, e analizzando quindi la libreria *gradle*, *group* identifica la compagnia, il progetto o l'organizzazione che detiene il modulo ("com.android.tools.build"), mentre *name* identifica la libreria rispetto alle altre ("gradle"), mentre *version* identifica quale release è oggetto di dipendenza (3.5.1). A volte, le librerie utili al progetto in questione possono anche non essere distribuite in appositi *repository* online, indipendentemente dal motivo questo strumento mette a disposizione al programmatore una modo di gestire queste dipendenze. Questo modo di operare, è generalemente sconsigliato, perchè al crescere di questo tipo di dipendenze aumenta anche il tempo impiegato nella gestione e nella manutenzione delle stesse, tuttavia una buona pratica è riunire le librerie interessate in una cartella *"lib"*, e di richiamarle complessivamente (*fileTree*) o singolarmente (*files*) come nel listato 2.10.

Listato 2.10: Dipendenze da file locali

```
dependencies {
    implementation(
        fileTree(
            mapOf("dir" to "libs", "include" to listOf("
               *.jar")) //1
        )
    )
    implementation(
        files("$projectDir/lib/3rdparty/gson-2.8.5.jar")
            //2
    )
```

```
}
```

Gradle non gestisce solamente dipendenze esterne, ma anche interne, infatti in una *build* multi-progetto, è possibile che un progetto sia dipendente da un altro. Come rappresentato nel listato 2.11, il *project* app è dipendente dal *subproject* http, dal *task* "install" del *subproject* react e dagli artefatti prodotti durante la configurazione "pubblished" del *subproject* react.

Listato 2.11: Dipendenze tra progetti

```
project(":app") {
    dependencies {
        "implementation"(project(":http"))
        "implementation"(project(":react:install"))
        //dependency of subproject's task
        "implementation"(project
            (path = ":api", configuration = "pubblished")
        )
        //dependency
        //of specific configuration' subproject
    }
}
```

In conclusione, nella maggiore parte dei casi le dipendenze sono moduli esterni vincolati su appositi *repository* online, ogni progetto detiene una lista di questi, e durante la *build*, Gradle risolve le dipendenze caricando in cache i moduli opportuni. Esistono tre tipi di *repository*:

- Maven
- Ivy
- cartelle statiche

Nei prossimi paragrafi, i tre tipi saranno analizzati e supportati di configurazione in Kotlin DSL.

Maven è il tipo di repository più diffuso (*Gradle in Action*), il modulo è formato dalla libreria sotto-forma di JAR, e di un file di configurazione XML, detto POM. Questo file contiene informazioni rilevanti sulla libreria e in particolare sulla sua transitività. Entrambi gli artefatti vengono localizzati sul *repository* grazie agli attributi dichiarati nella *build script*, in particolare la notazione *dot* di *group* indica il percorso specifico sul *repository*. Tra i più considerevoli di questo tipo troviamo Maven Central (Berglund, 2013), non solo per la varietà di librerie che ospita, ma anche per la sua accessibilità facilitata rispetto ai *repository* Maven ospitati su reti private. Infatti come si denota nel listato 2.12, Gradle offre supporto diretto per Maven Central, permettendo di omettere il suo url (`http://repo1.maven.org/maven2`), seguono anche le configurazioni per aggiungere un *repository* con url specifico e il Maven *repository* locale (*/.m2*)

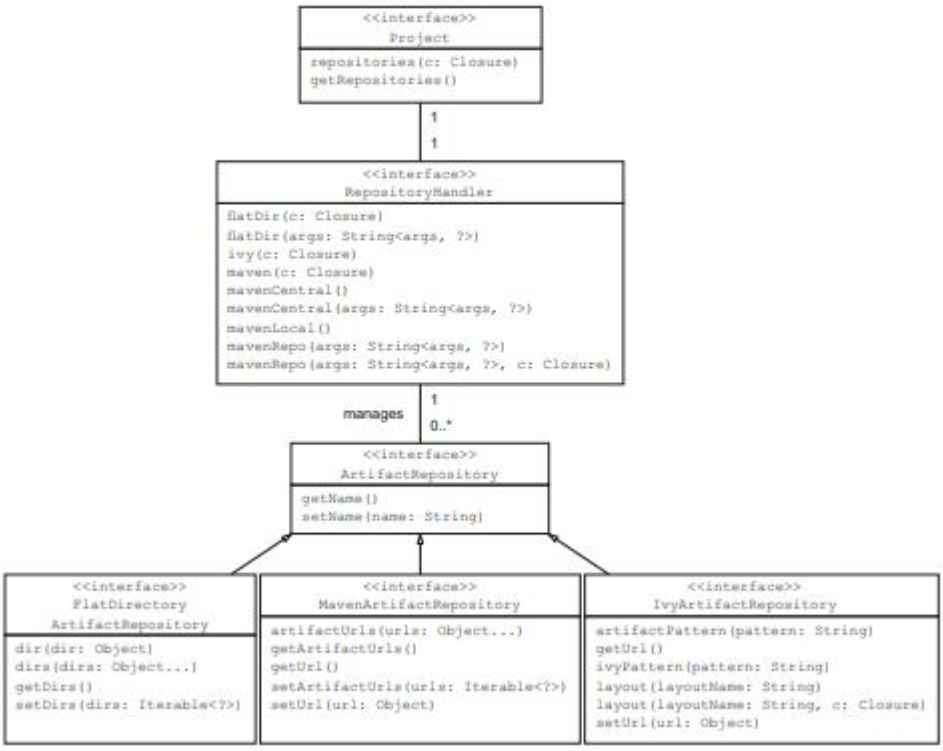

Figura 2.7: Interfacce rilevanti nelle API di Gradle per la gestione dei repositories

Listato 2.12: Dichiarazione dei repository Maven

```
repositories {
    mavenCentral() //http://repo1.maven.org/maven2
    mavenLocal()
    maven(url = "<MAVEN REPO URL>")
}
```

Se la dichiarazione degli artefatti in Maven seguono un layout preciso, previa l'impossibilità di risolvere le dipendenze, in Ivy il layout può essere stabilito a piacimento, infatti la sua struttura può essere profondamente diversa. Generalmente Gradle assume che la struttura sia uguale a quella di Maven, nel caso si volesse modificare il pattern di dichiarazione degli artefatti si può usare la funzione "patternLayout", come mostrato nel listato 2.13.

Listato 2.13: Dichiarazione dei repositery Ivy

```
repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
        patternLayout {
            artifact("3rd-party-artifacts/
                [organisation]/[module]/
                [revision]/[artifact]-[revision].[ext]"
            )
            artifact("company-artifacts/
                [organisation]/[module]/
                [revision]/[artifact]-[revision].[ext]"
            )
            ivy("ivy-files/
                [organisation]/[module]/
                [revision]/ivy.xml"
            )
        }
```

```
    }
}
```

Gralde non solo supporta una dipendenza locale, ma permette di dichiarare anche *repository* locali, questa funzionalità è utile quando un programmatore vuole usare un proprio archivio di librerie, tuttavia è una pratica sconsigliata perchè Gralde non si prende carico nella gestione delle dipendenze transitive. In seguito, le dipendenze locali possono essere dichiarate soltanto per nome e versione come nel listato 2.14.

Listato 2.14: Dichiarazione *repository* locale

```
repositories {
    flatDir {
        dirs("lib")
    }
    flatDir {
        dirs("lib1", "lib2")
    }
}
```

Per completare il concetto di *repository*, Gradle può dichiarare anche archivi protetti da password (es. *repository* aziendali o di organizzazioni) con il metodo "credentials", che permette di aggiungere un nome utente e un password di login. Rimane buona pratica, tuttavia, non dichiarare le credenziali nel *build script*, ma dichiararle nel file di configurazione "gradle.properties" e di adoperarle come mostrato nel listato 2.15.

Listato 2.15: Autenticazione tramite credenziali

```
//settings.gradle
username = "myusername"
password = "mypassword"

//gradle.build.kts
repositories {
```

```
    maven {
        url = uri("http://repo.mycompany.com/maven2")
        credentials {
            username = username
            password = password
        }
    }
}
```

In conclusione, Gradle si dimostra uno strumento avanzato nella gestione delle dipendenze rispetto alla concorrenza (*Gradle in Action*), infatti offre una performante gestione dei moduli attraverso la cache, memorizzando separatamente in locale i moduli e i loro metadata (*pom.xml* e *ivy.xml*). Mentre i secondi sono identificati dalla tupla *group ID*, *artifact ID*, versione e *repository* di provenienza, i secondi sono classificati in base alla funzione crittografica SHA-1. Distaccando l'informazione del *repository* di provenienza, l'artefatto diventa localmente universale e indipendente, permettendo a Gradle di scaricare una sola copia della libreria, e non più versioni della stessa per ogni *repository* dichiarato. In questo modo, se il modulo non è già presente in cache, questo strumento tenta di scaricare quello con lo SHA di grandezza minore, ottimizzando i tempi di download e di build. Esistono una coppia di comandi per modificare il comportamento di Gradle, `--offline` forza l'uso della cache nel risolvere le dipendenze, permettendo di lavorare senza nessuna connessione, mentre `--refresh-dependencies` in contrasto, aggiorna i meta-data utilizzando la connessione internet. Per completare i concetti precedenti, è fondamentale per il programmatore comprendere la strategia applicata da questo strumento nella gestione dei conflitti tra le dipendenze. Spesso, data la transitività dei moduli, può essere necessario la stessa libreria ma di versioni differenti, Gradle appunto di default scarica la versione più recente, almeno che il programmatore non abbia dichiarato nel *build script* la versione specifica da preferire.

Listato 2.16: Gestione dei conflitti tra dipendenze

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.group == "org.gradle") {
            useVersion("1.4")
            because("API breakage in higher versions")
        }
    }
}
```

2.3.3 Estensione delle funzionalità di Gradle

Gradle offre diversi approcci, con i suoi pro e i suoi contro, per riusare il codice, uno tra questi è l'uso di plugin. Questo elemento che è già stato introdotto durante lo studio delle dipendenze, in particolari nelle configurazioni per Java, permette di estendere le funzionalità di Gradle in tre modi differenti. Il primo, modifica l'oggetto *Project*, arricchendo la *build* corrente con *tasks*, *SourceSets*, *dependencies* e *repositories*. Il secondo modo, può introdurre uno o più moduli per delegare complessi lavori a librerie esterne. Infine, il terzo modo, decisamente fondamentale per facilitare l'uso del plugin, è la possibilità di introdurre nuove *keyword* e *domain object* nel linguaggio di compilazione di Gradle. Questo è necessario, perché i DSL come analizzato nella sezione 2.2, non possono per definizione essere adatti ad ogni scenario e dominio nel quale il progetto può andare in contro. In Gradle, esistono due tipi di plugin, *script plugins* e *binary plugins*. I primi sono essenzialmente composti da *build script* che si aggiungono alla *build* corrente e implementano un approccio dichiarativo nel manipolare il progetto, si differenziano dai secondi, che invece estendono l'interfaccia Gradle *Plugin* e adottano un approccio pragmatico nella manipolazione della *build*. Per usare la logica di un *Plugin*, Gradle come nelle dipendenze, deve risolverlo, ovvero identificare la giusta versione del jar che lo contiene e aggiungerlo al *classpath*, in seguito, generalmente viene già applicato al target (generalmente un *Project*) attraverso la funzione `Plugin.apply(T)`,

o come consiglia Gradle stesso (*Using Gradle Plugins*) attraverso il *plugins* DSL, come mostrato nel listato 2.17.

Listato 2.17: Diversi modi di applicare un plugin

```
apply(from = "other.gradle.kts") //apply script plugin

plugins {
    java //apply core plugin
    //apply community plugin
    id("com.jfrog.bintray") version "0.4.1"
}
```

In conclusione, Gradle è uno strumento che permette non soltanto di gestire il codice in più aspetti, ma anche di cambiare profondamente la natura di un progetto, soprattutto tramite i plugin. Questi elementi, potenziano questo strumento ulteriormente, infatti nei prossimi capitoli, sulla base del plugin "kotlin-multiplatform", verrà analizzata la costruzione di un applicazione multipiattaforma, e in particolare come si organizza il codice e come si scrive il build-script.

Capitolo 3

Applicazioni multipiattaforma

Con il rilascio di Kotlin 1.1 nel 2013, JetBrains ha rilasciato Koltin JS, permettendo di compilare, appunto, Kotlin in Javascript (Belov, 2017). Se prima i programmatori attraverso questo linguaggio, avevano la possibilità di modellare la logica di un applicativo e poi tradurlo in Java, con questo aggiornamento ora è possibile modellare anche il front-end e compilarlo quindi con Javascript. Più avanti, alla fine del 2017, Jetbrains è andato oltre, rilasciando Kotlin Multiplatform (Jemerov, 2017), questa funzione , ancora sperimentale, permette di condividere il codice attraverso qualsiasi client JS o JVM based, come un frontend in React, un client Android, un client Desktop, e molto altro.

L'idea principale di un progetto multipiattaforma è condividere il codice tra diversi team di sviluppo e piattaforme. Se da una parte questa prerogativa, nata in concomitanza con la stesura del codice, è diventata l'obiettivo di moltissimi progetti odierni (Tobias Heine, 2018), organizzare il codice in modo ordinato e ottimizzato tra più piattaforme rimane una sfida ancora aperta. Le difficoltà nate con il crescere e il diversificarsi delle tecnologie, hanno portato alla nascita di diversi tool come PhoneGap, Titanium, Xamarin, Ionic, RubyMotion e molti altri. Questi strumenti si sono rilevati nel corso del tempo, poco flessibili e poco adatti a coprire i diversi SDK, e le esigenze delle varie piattaforme, soprattutto per quanto concerne UX e UI dell'applicativo, portando inevitabilmente gli sviluppatori a diversificare il team in progetti differenti (touchlab, 2019; Tobias Heine, 2018).

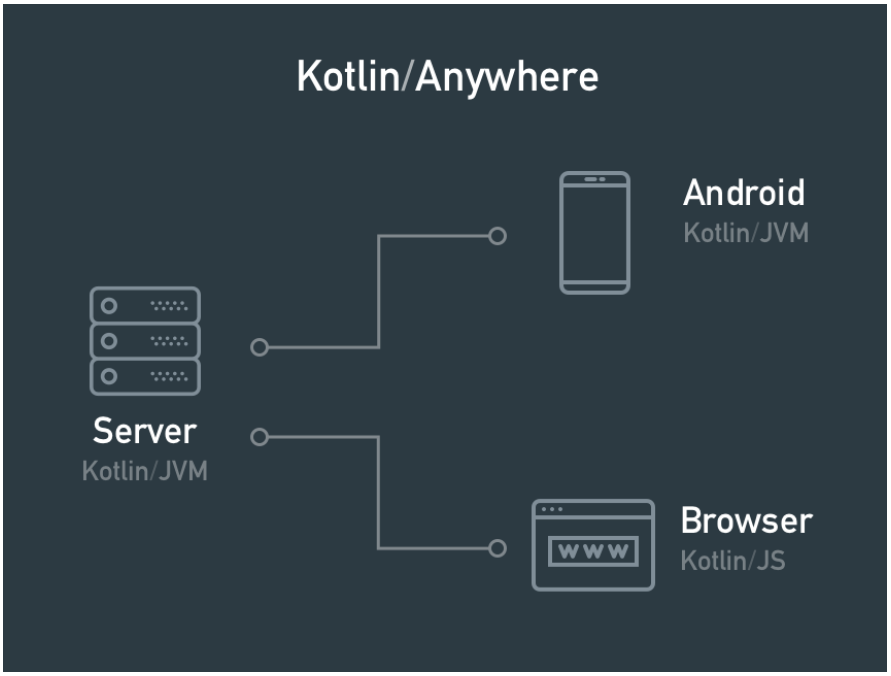

Figura 3.1: Distribuzione di Kotlin su più piattaforme JS/JVM based

Kotlin si differenzia dai suoi concorrenti perché non traduce tutto il codice in quello specifico della piattaforma, ma lascia allo sviluppatore pieni poteri nel decidere quali parti del codice rendere condivise e quali native. Generalmente, soprattutto per le piattaforme front-end, i progetti multipiattaforma mirano a condividere la logica e a lasciare implementare le parti critiche agli SDK, in modo da sviluppare un'unica logica robusta di business per ogni piattaforma, abbattendo quindi anche i costi e i tempi di produzione.

3.1 Logica di base

Il progetto è organizzato in moduli che condividono più o meno diverse parti di codice, la logica alla base della divisione in moduli segue tre blocchi fondamentali:

- Target, è la parte di build responsabile alla costruzione, testing, e packaging di una parte di software per una delle piattaforme. Generalmente, seguono più target in un progetto multipiattaforma.

- Compilazioni, la finalizzazione in target del codice segue una o più fasi di compilazione del sorgente. Può esistere un tipo di compilazione per la produzione del sorgente principale, e uno per i test.
- Source sets, sono l'insieme di cartelle che contengono i sorgenti del codice in Kotlin o nel linguaggio specifico della piattaforma. Raggruppano con s'è anche le dipendenze specifiche per la piattaforma e possono avere relazioni di dipendenza tra loro.

Per ogni compilazione esiste, di default, un suo *source set*, che in un applicazione multipiattaforma generale viene utilizzato anche per indirizzare altri set di sorgenti alla compilazione, mediante la relazione "dipende da" come mostrato nella fig. 3.2. Il layout in questione viene ottenuto di default attraverso la creazione di due

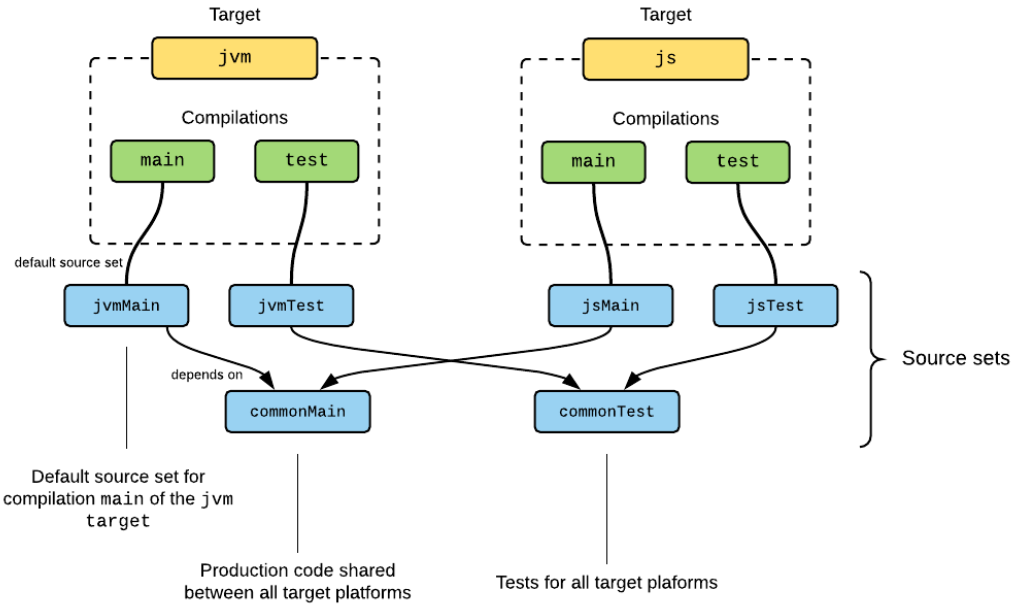

Figura 3.2: Per ogni target, due compilazioni di default, main e test, e i relativi source set anch'essi di base

target jvm e js senza nessuna configurazione aggiuntiva. In particolare, "jvmMain" condivide il sorgente con "commonMain", che attraverso la compilazione "main"

riescono a soddisfare la richiesta del target specifico jvm.
Nel concreto, questa relazione di dipendenza si basa sul meccanismo "actual/expect", una tecnica avanzata che permette di gestire le specificità delle piattaforme e che è oggetto di approfondimento nei prossimi paragrafi.

3.1.1 Meccanismo Expect/Actual

Kotlin è uno strumento importante nell'applicazioni multipiattaforma, perché come è già stato accennato, permette di organizzare il codice tra più piattaforme in modo coeso e organizzato, permettendo allo sviluppatore di accedere anche alle funzioni specifiche di una piattaforma. Una delle funzionalità chiave del codice multipiattaforma di questo linguaggio è il modo in cui il sorgente comune dipende dalle dichiarazioni specifiche della piattaforma, che tecnicamente si realizza nella costruzione di una serie di interfacce (*expect declaration*, appunto le dichiarazione che il codice comune si aspetta) che verranno poi implementate in modo adeguato per ogni piattaforma (*actual declaration*).
Un esempio concreto nell'uso di questo meccanismo può essere la costruzione di un framework di logging minimalista, il listato 3.1 mostra il codice in comune tra le piattaforme e l'uso della keyword *expect*

Listato 3.1: Esempio di uso *expect* nel codice comune

```
//Common Code

enum class LogLevel { //compiled for all platform
    DEBUG, WARN, ERROR
}

//expected API

internal expect fun writeLogMessage(message: String,
   logLevel: LogLevel)
```

```
//expected API could be used in common code

fun logDebug(message: String) = writeLogMessage(message,
    LogLevel.DEBUG)
fun logWarn(message: String) = writeLogMessage(message,
   LogLevel.WARN)
fun logError(message: String) = writeLogMessage(message,
    LogLevel.ERROR)
```

che è limitato in questo caso alle funzioni, ma può essere adoperata su qualsiasi dichiarazione in Kotlin, come classi astratte, interfacce e più in generale qualsiasi funzione top-level (anche le annotazioni). In questo modo, `writeLogMessage()` può essere usato nel sorgente comune indipendentemente dalla sua implementazione, che verrà concretizzata correttamente nei moduli specifici, come mostrato in questo caso specifico nel listato 3.2.

Listato 3.2: Implementazione di `writeLogMessage` su ogni piattaforma specifica

```
//JVM

internal actual fun writeLogMessage(message: String,
   logLevel: LogLevel) {
    println("[$logLevel]: $message")
}

//JS

internal actual fun writeLogMessage(message: String,
   logLevel: LogLevel) {
    when (logLevel) {
        LogLevel.DEBUG -> console.log(message)
        LogLevel.WARN -> console.warn(message)
        LogLevel.ERROR -> console.error(message)
```

```
    }
}
```

In conclusione, durante la compilazione, il compilatore si assicura che tutte le dichiarazioni *actual* e *expect* coincidano, per questo le dichiarazione correlate devono coincidere nel nome completo.

3.2 Setup di un progetto multipiattaforma con Gradle

JetBrains, azienda sviluppatrice di Kotlin, nel suo IDE IntelliJ IDEA, attraverso il plugin Kotlin (aggiornato alla versione 1.2), ha messo a disposizione degli sviluppatori un wizard guidato per la creazione di progetti multipiattaforma. Per iniziare la configurazione è necessario Gradle aggiornato almeno alla versione 4.7, che a procedura terminata applica il plugin "kotlin-multiplatform" permettendo al programmatore di accedere a tutte le funzionalità di Kotlin Multipiattaforma (configurazione dei target, source sets, dipendenze e altro).

Listato 3.3: kotlin-multiplatform plugin

```
plugins {
    kotlin("multiplatform") version "1.3.50"
}
```

A configurazione completata, il progetto dispone tre target principali, JVM, JS e uno per la piattaforma nativa in uso, questi vengono configurati come nel listato 3.4 mediante l'uso delle funzioni predefinite: `jvm()`, `js()` e `mingwX64()` nel caso l'host sia Windows64.

Listato 3.4: Configurazione dei target nel file gradle.build.kts

```
plugins {
    kotlin("multiplatform") version "1.3.50"
}
```

```
repositories {
    mavenCentral()
}

kotlin {
    jvm() // Creates a JVM target with the default name
       'jvm'
    js()  // JS target named 'js'
    mingwX64("mingw") // Windows (MinGW X64) target
       named 'mingw'

    sourceSets { /* ... */ }
}
```

Tuttavia queste non sono le uniche piattaforme supportate, ma l'elenco delle funzioni prestabilite si espande alle seguenti piattaforme:

- `android` per librerie e applicazioni Android;
- `androidNativeArm32` e `androidNativeArm64` per Android NDK;
- `iosArm32`, `iosArm64` e `iosX64` per iOS;
- `linuxArm32Hfp`, `linuxMips32`, `linuxMipsel32`, `linuxX64` per Linux;
- `macosX64` per MacOs;
- `wasm32` per WebAssembly;
- `watchosArm32`, `watchosArm64`, `watchosX86` per watchOs;
- `tvosArm64`, `tvosX64` per tvOs.

Di default, ogni progetto contiene due source sets, "commonMain" e "commonTest" collegati alle rispettive configurazioni "main" e "test", questi due moduli contengono tutto il sorgente condiviso dalle piattaforme. Oltre a questi due moduli, per ogni target dichiarato, seguono due compilazioni "main" e "test" e la serie

di source sets, che prendono il nome in base alla compilazione e al target in questione; questi moduli che seguono quindi la dicitura *¡nomeTarget¿¡nomeCompilazione¿* di default contengono il codice specifico per quella piattaforma e partecipano alla compilazione del target.
In conclusione in un progetto multipiattaforma su JS, JVM, i moduli di base saranno "commonMain" e "commonTest" per il sorgente comune, "jvmMain" e "jvmTest" per JVM, infine "jsMain" e "jsTest" per JS. Queste diciture nascondono, senza nessun tipo di configurazione in Gradle, delle relazioni di dipendenza, infatti è evidente che nella costruzione del target vengano compilati sia i sorgenti comuni che quelli del source set specifico.

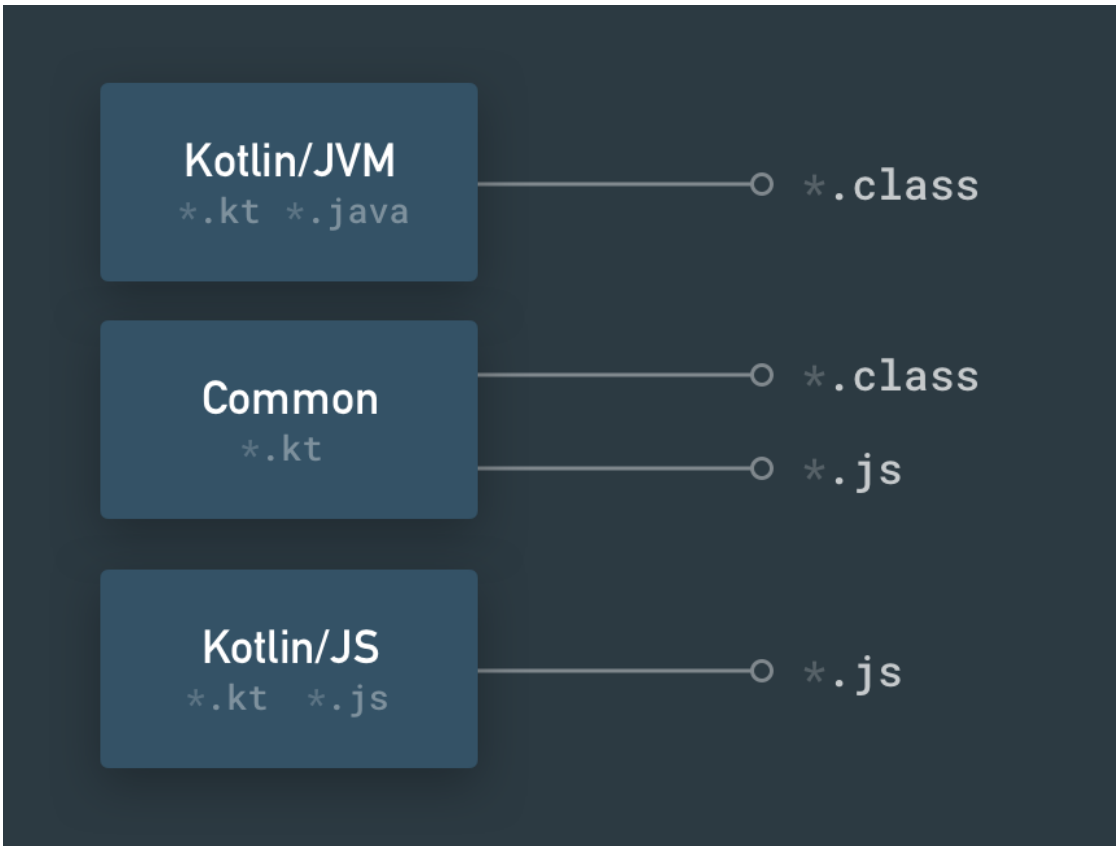

Figura 3.3: Moduli e i loro artefatti

3.2.1 Configurazione dei Target

Per Target si intende quella parte di build responsabile alla compilazione, test e packaging di una porzione di software adibita ad una piattaforma specifica. Queste porzioni possono contenere sia del codice comune che quello specifico della piattaforma, tecnicamente i target vengono configurati attraverso le funzioni predefinite già accennate nei paragrafi precedenti, queste opzionalmente, accettano il nome del target e una codice per la configurazione dello stesso, come mostrato nel listato 3.5

Listato 3.5: Configurazioni dei target con le funzioni prestabilite di Kotlin

```
kotlin {
    jvm() { // Create a JVM target with the default name
        'jvm'
        withJava()
    }
    jvm("jvmLibrary")
    js("nodeJs") // Create a JS target with a custom
       name 'nodeJs'

    linuxX64("linux") {
        /* Specify additional settings for the 'linux'
           target here */
    }
}
```

Ogni target creato viene aggiunto alla collezione `kotlin.targets` che permette di accedere globalmente ad ognuno di loro, o singolarmente attraverso il nome; se invece si vuole creare più target contemporaneamente o configurarli si può adoperare la notazione `targetFromPreset()` che accetta come argomento un preset (contenuti in `kotlin.presets`) e un blocco di codice configurativo per i target interessati. Da Kotlin 1.3.40 il target predefinito jvm supporta anche i sorgenti Java attraverso il relativo plugin, applicabile con la funzione `withJava()`.

Listato 3.6: Diversi modi di accedere ai target creati

```
import org.jetbrains.kotlin.gradle.plugin.mpp.
   KotlinNativeTargetPreset



kotlin {
    targets.all {
        compilations["main"].defaultSourceSet { /* ...
           */ }
    }
    presets.withType<KotlinNativeTargetPreset>().forEach
        {
        targetFromPreset(it) {
            /* Configure each of the created targets */
        }
    }
}
```

3.2.2 Configurazione dei Source Sets

Un source set è una collezione di sorgenti Kotlin, risorse e dipendenze che partecipano alla compilazione di uno o più target, il suo contenuto dipende dalla sua applicazione, infatti se partecipa a più compilazioni, il suo codice dipende solamente dalla libreria comune di Kotlin (kotlin-stdlib-common), mentre se partecipa alla costruzione di un singolo target, può usufruire delle sue specificità sia per le dipendenze che per il codice. I source set predefiniti di Kotlin, come "commonMain" e "commonTest" sono già configurati, per modificarli o aggiungerne di nuovi si può usare la notazione `sourceSets ...` come mostrato nel listato 3.7

Listato 3.7: Configurazione del build.file per creare e modificare un source set esistente

```
kotlin {
```

```
    sourceSets {
        val foo by creating { /* ... */ } // crea un
           nuovo source set con il nome "foo"
        val commonMain by getting { // configura un
           source set esistente e modifica il nome delle
            cartelle contenenti il sorgente e le risorse
            kotlin.srcDir("src")
            resources.srcDir("res")
        }
    }
}
```

Generalmente i source sets sono creati mediante le funzioni predefinite, ma ovviamente Kotlin non vieta la creazione di set personalizzati anche se questi non portano con sé nessun tipo di configurazione, infatti per partecipare alla costruzione di uno specifico target devono essere dipendenti dalla compilazione di un source set predefinito.
Supponendo che un ipotetico set ”foo” dipenda da un secondo ipotetico set ”bar”, le relazioni di dipendenza si articolano su 5 punti:

- Nel momento in cui ”foo” partecipa nella compilazione di un determinato target, ”bar” prende parte a quella compilazione, producendo gli stessi binari;
- ”foo” può accedere alle dichiarazioni di ”bar”, nonostante siano dichiarate con ”internal”, e alle sue dipendenze, anche se specificate come ”implementation”;
- ”foo” può implementare le dichiarazioni expect contenute in ”bar”;
- le risorse di ”bar” vengono processate e copiate sempre insieme a quelle di ”foo”;
- i settaggi sul linguaggio adoperato in ”foo” e in ”bar” devono essere consistenti.

Le relazioni di dipendenza si esprimono con la notazione `dependsOn()`, e come riportato nel documento ufficiale *Building Multiplatform Projects with Gradle*, nella maggior parte dei casi, i set personalizzati seguono la configurazione mostrata nel listato 3.8.

Listato 3.8: Target personalizzati e relazioni di dipendenza

```
kotlin {
    mingwX64()
    linuxX64()

    sourceSets {
        // custom source set with tests for the two
           targets
        val desktopTest by creating {
            dependsOn(getByName("commonTest"))
            /* ... */
        }
        // Make the 'windows' default test source set
           for depend on 'desktopTest'
        mingwX64().compilations["test"].defaultSourceSet
            {
            dependsOn(desktopTest)
            /* ... */
        }
        // And do the same for the other target:
        linuxX64().compilations["test"].defaultSourceSet
            {
            dependsOn(desktopTest)
            /* ... */
        }
    }
}
```

Kotlin non esclude l'utilizzo di moduli esterni al progetto e gestisce tramite Gradle le dipendenze esterne con la funzione `dependencies ...` appartenente al DSL di source sets.
Sono quattro i tipi di dipendenze supportati:

- "api", il modulo in questione può essere utilizzato sia durante in fase di compilazione che di runtime, inoltre è accessibile all'utente finale;
- "implementation", la libreria esterna è accessibile nelle stesse fasi build di "api" ma si differenzia da quest'ultimo non esponendo il modulo al consumatore finale;
- "compileOnly", il modulo in questione partecipa solamente alla sua fase di compilazione, e non rimane visibile in nessuna fase di build degli altri moduli;
- "runtimeOnly", segue la stessa logica della dipendenza precedente rimanendo invisibile durante la compilazione di altri moduli, tuttavia si differisce usando la libreria esterna in questione durante la fase di runtime;

Listato 3.9: Esempio di configurazione delle dipendenze dei source set "commonMain" e "jsMain"

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation(kotlin("stdlib-common"))
                api(project(":foo-lib")) //project
                    dependency
            }
        }
        val jsMain by getting {
            dependencies {
                api("com.example:foo-js6:1.0")
```

```
            }
        }
    }
}
```

Come mostrato nel listato 3.9, Kotlin Multipiattaforma supporta anche le dipendenze di progetti esterni, infatti nel listato in questione, la compilazione che adopererà il source set "commonMain" riceverà una versione specifica dell'artefatto del progetto, risolvendola per quel target specifico.

Listato 3.10: Dichiarazione alternativa delle dipendenze attraverso la funzione top-level di Gradle DSL

```
dependencies {
    "commonMainApi"("com.example:foo-common:1.0")
    "jvm6MainApi"("com.example:foo-jvm6:1.0")
}
```

Il linguaggio contenuto nei source set può anch'esso subire diverse configurazioni attraverso la funzione `languageSettings`, che permette di modificare globalmente o singolarmente l'analisi dei sorgenti da parte dell'IDE. L'unico limite posto riguarda le dipendenze tra i set, infatti se un ipotetico "foo" dipende da "bar":

- "foo" deve possedere una versione di linguaggio superiore a quella di "bar";
- "foo" deve abilitare tutti le funzioni di linguaggio instabili che "bar" ha abilitato;
- "foo" deve usare tutte le annotazioni sperimentali usate da "bar".

Listato 3.11: Impostazioni relativi ai sorgenti

```
kotlin {
    sourceSets {
        val commonMain by getting {
```

```
            languageSettings.apply {
                languageVersion = "1.3" // possible
                    values: '1.0', '1.1', '1.2', '1.3'
                apiVersion = "1.3" // possible values:
                    '1.0', '1.1', '1.2', '1.3'
                enableLanguageFeature("InlineClasses")
                    // language feature name
                useExperimentalAnnotation("kotlin.
                    ExperimentalUnsignedTypes") //
                    annotation FQ-name
                progressiveMode = true // false by
                    default
            }
        }
    }
}

//Modificare tutti i sorgenti di tutti i soruce sets

kotlin.sourceSets.all {
    languageSettings.progressiveMode = true
}
```

3.2.3 Criticità e Interoperabilità in JVM e JS

Kotlin per natura si differenzia da Java e da JavaScript sotto diversi aspetti, in primis il concetto di "static" in Java è gestito attraverso gli oggetti, mentre rispetto a JavaScript, sempre Kotlin, si presenta per natura come un linguaggio fortemente tipizzato e non dinamico.
La staticità infatti rispetto al linguaggio di Oracle è modellata come un oggetto di una classe, più precisamente come singleton, e se seguita dalla notazio-

ne "companion", come mostrato nel listato 3.12, non deve essere istanziata per accedervi.

Listato 3.12: Companion Object essendo un oggetto può essere esteso o implementare delle interfacce

```
class C {
    companion object {
        @JvmStatic fun callStatic() {}
        fun callNonStatic() {}
    }
}

//in Java

C.callStatic(); // works fine
C.callNonStatic(); // error: not a static method
C.Companion.callStatic(); // instance method remains
C.Companion.callNonStatic(); // the only way it works
```

Durante la traduzione del codice in Java è utile apportare l'annotazione `@JvmStatic`, in questo modo il compilatore genererà sia un metodo statico che un metodo istanziato nella classe corrispondente.
A differenza di Kotlin e anche Java in questo caso, Javascript non supporta l'overload dei metodi e soprattutto non ha variabili tipizzate, ma solo dinamiche. Questa criticità è stata risolta dagli sviluppatori di JetBrains attraverso l'introduzione della notazione "dynamic"; principalmente una variabile dichiarata in questo modo segue la logica di una variabile Javascript rimanendo libera da qualsiasi tipo. A livello tecnico, il compilatore traduce queste espressioni così come sono, non apportando nessun controllo (null-check disabilitati).
Per quanto concerne l'overload dei metodi, in Kotlin le funzioni interessate vanno diversificate mediante l'annotazione `@JsName` che accetta come stringa il nome con il qualle il metodo verrà tradotto in JS.

Listato 3.13: Uso di @JsName()

```
// Module 'kjs'
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
// Javascript Ecosystem

var person = new kjs.Person("Umberto");   // refers to
   module 'kjs'
person.hello();                           // prints "
   Hello Umberto!"
person.helloWithGreeting("Servus");       // prints "
   Servus Umberto!"
```

In conclusione, costruire un applicazione multipiattaforma attraverso Gradle e Kotlin è sicuramente più veloce grazie alle configurazioni predefinite del plugin "koltin-multiplatform", questa nuova funzione di Kotlin è ancora in fase sperimentale, tuttavia JetBrains sta investendo molto nel suo sviluppo visto i repentini aggiornamenti apportati. In questo capitolo è stata introdotta la logica di base, e sono stati approfonditi gli elementi che stanno alla base di questa tecnologia, come i target e i source set, mostrando valide configurazioni DSL. Tuttavia, non tutti gli argomenti sono stati sviscerati, nei prossimi capitoli l'attenzione si porrà sulla fase di testing e deploy.

Capitolo 4

Test e Deploy di una libreria multipiattaforma

Una visione ingegneristica non esclude un'applicazione multipiattaforma dalla fase di collaudo prima del deploy. Queste operazioni, che sono più complesse vista le diversità dei sorgenti, sono pienamente supportate da Gradle. Nel corso di questo capitolo analizzeremo il supporto specifico offerto da questo strumento, supportando la teoria con numerosi esempi di codice, in modo da dare continuità al ruolo guida di questo documento.

4.1 Testing

Il testing dei sorgenti è una procedura atta ad individuare le carenze di correttezza, completezza e affidabilità di un prodotto software in corso di sviluppo, il suo obbiettivo è garantire un prodotto di qualità e di ridurre al minimo i malfunzionamenti dopo il deploy. Gradle a riguardo, offre pieno supporto, con una consistente configurazione di base, per gli ecosistemi JVM, Android, Linux, Windows e macOS mentre per il target JS, necessità di configurazioni aggiuntive che verranno analizzate nei paragrafi a seguire.
L'API kotlin.test è la libreria sviluppata da Kotlin che permette di sviluppare i test per applicazioni mutlipiattaforma, questa racchiude in sé una serie di annotazioni

e di funzioni utili a perfomare delle asserzioni indipendentemente dal framework di test usato. I moduli raccolti supportano i test su più piattaforme:

- kotlin-test-common – per asserzioni sul codice comune;
- kotlin-test-annotations-common – annotazioni utili al compilatore durante i test del sorgente comune;
- kotlin-test – un implementazione in JVM di kotlin-test-common;
- kotlin-test-junit e kotlin-test-junit5 forniscono un implementazione della classe Asserter in Junit e Junit 5 e delle rispettive annotazioni di kotlin-test-annotations-common;
- kotlin-test-testng - implementazioni di Asserter in TestNG, e mapping delle annotazioni di kotlin-test-annotations-common in annotazioni TestNG;
- kotlin-test-js - Implementazioni delle asserzioni e annotazioni comuni di Kotlin in Javascript, con supporto a framework esterno(Jest, Mocha, Karma, Jasmine).

La configurazione base delle dipendenze segue il listato 4.1.

Listato 4.1: Configurazione base nella gestione delle dipendenze per kotlin.test

```
kotlin {
    val commonMain by getting { ... }
    val commonTest by getting {
        dependencies {
            implementation(kotlin("test-common"))
            implementation(kotlin("test-annotations-
                common"))
        }
    }

    //JS
```

```
    js().compilations["main"].defaultSourceSet  { ... }
    js().compilations["test"].defaultSourceSet  {
        dependencies {
            implementation(kotlin("test-js"))
        }
    }

    //JVM

    jvm().compilations["main"].defaultSourceSet { ... }
    jvm().compilations["test"].defaultSourceSet {
        dependencies {
            implementation(kotlin("test-junit"))
        }
    }
}
```

Queste dipendenze permettono di usare metodi come `assertTrue` o `assertFalse`, che insieme alle annotazioni `@Test`, `@AfterTest`, `@BeforeTest` e `@Ignore` consentono di scrivere test in Kotlin.

Listato 4.2: Esempio di test in Kotlin

```
@Test
fun twoSideConversionTest() {
    val dateFormatted = "2018-10-12T12:00:01"
    assertEquals(dateFormatted, dateFormatted.parseDate
       ().toDateFormatString())
}
```

Per eseguire test specifici su un target, Kotlin Multipiattaforma mette a disposizione dello sviluppatore dei task predefiniti, che seguono la dicitura ¡nomeTarget¿Test, inoltre, per eseguirli tutti Kotlin crea il task *check*.

Se la configurazione base permette di eseguire test sulla maggior parte delle piattaforme, il target JS a differenza degli altri dipende da un framework esterno. Con l'aggiornamento alla versione 1.3.40 di Kotlin, la configurazione per questo ambiente è stata semplificata, aggiungendo il supporto agli ambienti esecutivi NodeJs e Browser (*Kotlin 1.3.40 released* 2019). I test possono essere eseguiti su più browser a piacimento o su NodeJS se preferibile, Gradle con il DSL mostrato nel listato 4.3, configura Karma[1] e le dipendenze NPM automaticamente.

Listato 4.3: DSL configurativo per eseguire test in JS, Karma può essere sosituito con Mocha o NodeJs

```
kotlin {
    js { //target per le applicazioni non
       multipiattaforma
        browser {
            testTask {
                useKarma {
                    useIe()
                    useSafari()
                    useFirefox()
                    useChrome()
                    useChromeCanary()
                    useChromeHeadless()
                    usePhantomJS()
                    useOpera()
                }
            }
        }
    }
}
```

[1]Test Runner per Javascript, non è l'unico adoperabile

I risultati generati dopo l'esecuzione dei task predefiniti, sono consultabili nella cartella *"build/reports/tests"* sottoforma di pagina HTML, suddivisi per target in diverse cartelle (4.1).

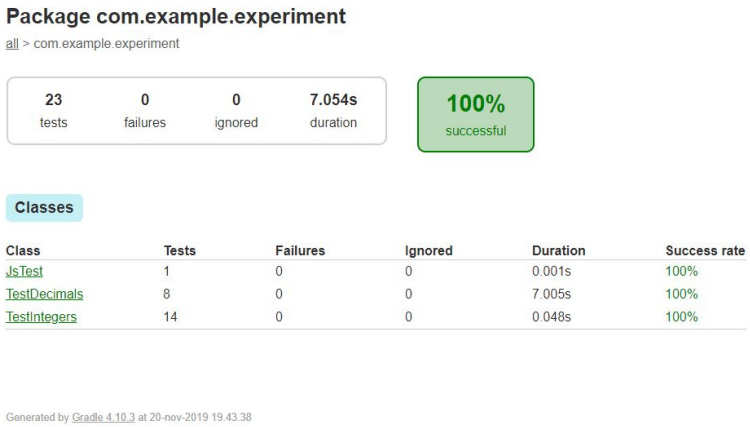

Figura 4.1: Report in HTML dei risultati dei test, generati attraverso il task "allTests"

4.2 Deploy

La stragrande maggior parte dei progetti software crea qualcosa che mira a essere consumato in qualche modo. Potrebbe essere una libreria utilizzata da altri progetti software o un'applicazione per gli utenti finali. La pubblicazione è il processo attraverso il quale l'oggetto che viene costruito viene messo a disposizione dei consumatori. Prima di comprendere il funzionamento di questa fase è importante per lo sviluppatore o per il lead programmer comprendere al meglio cosa pubblicare di un progetto multipiattaforma e sopratutto dove. A livello tecnico questi step dipendono soprattutto dalla scelta del repository adatto alla pubblicazione della libreria, i più comuni sono Maven e Ivy per librerie Java, e npm registry per librerie Javascript.
Prima però di condividere il proprio codice è fondamentale fornire la documenta-

zione adeguata, e per eseguire questo nei migliori dei modi esiste il plugin "Dokka". Un esempio configurativo basilare è il listato 4.4

Listato 4.4: Configurazione di Dokka di un progetto multipiattaforma tramite Gradle

```
kotlin {  // Kotlin Multiplatform plugin configuration
    jvm()
    js("customName")
}

val dokka by getting(DokkaTask::class) {
        outputDirectory = "$buildDir/dokka"
        outputFormat = "html"

        multiplatform {
            val customName by creating { // The same
               name as in Kotlin Multiplatform plugin,
               so the sources are fetched automatically
                includes = listOf("packages.md", "extra.
                   md")
                samples = listOf("samples/basic.kt", "
                   samples/advanced.kt")
            }

            register("differentName") { // Different
               name, so source roots must be passed
               explicitly
                targets = listOf("JVM")
                platform = "jvm"
                sourceRoot {
                    path = kotlin.sourceSets.getByName("
                       jvmMain").kotlin.srcDirs.first().
```

```
                    toString()
                }
                sourceRoot {
                    path = kotlin.sourceSets.getByName("
                        commonMain").kotlin.srcDirs.first
                        ().toString()
                }
            }
        }
    }
```

Ovviamente, essendo presente la piattaforma JS, l'output della documentazione non può essere javadoc. Il plugin in questione apporta anche un task di nome "dokka" per produrre effettivamente la documentazione, è buona pratica estendere diversi task di tipo dokkaTask con diversi output se necessario. Mentre il plugin "org.jetbrains.dokka" permette di produrre la documentazione, i plugin "com.moowork.node" o "com.liferay.node", e "maven-pubblish" o "ivy-pubblish", permettono di pubblicare i sorgenti JS e Java.

4.2.1 Maven Repository e Ivy Repository

La pubblicazione su Maven, a livello tecnico è molto simile a quella su Ivy per questo i listati mostrati faranno riferimento al primo. In primis, va applicato l'apposito plugin, il listato 4.5, oltre a dimostrare la sua applicazione con Kotlin DSL, indica anche il modo di impostare gli identificativi *group* e la versione del progetto. Il nome della libreria, inoltre, è impostato automaticamente da Gradle con il nome della cartella del progetto, tuttavia è buona pratica esplicare il nome anche nel file di configurazione *settings.gradle.kts*.

Listato 4.5: Maven Pubblish Plugin

```
plugins {
    /* ... */
    id("maven-publish") //ivy-publish for ivy repo
```

```
}

group = "com.example.my.library"
version = "0.0.1"
```

Impostate le coordinate del progetto, per pubblicare il modulo su repository è necessario configurare le pubblicazioni attraverso il DSL `publishing ...`. Per i progetti Java, Kotlin Multipiattaforma contiene già una configurazione base che crea diverse pubblicazioni per ogni target supportato, sarà compito del programmatore configurare solamente la locazione del repository dove si vuole pubblicare e includere la documentazione.

Listato 4.6: Configurazione personalizzata per pubblicare su repository Maven e includere la documentazione Dokka

```
publishing {
    publications.withType<MavenPublication>().apply {
        val jvm by getting { /* Setup the publication
           for target jvm */ }
        val metadata by getting { /* Setup the
           publication for Kotlin metadata */ }

        if("dokka" in tasks.names) {
            artifact(tasks.getByName("dokka")) {
                classifier = "html"
            }
        }
    }
}

    repositories {
        maven {
            name = "myRepo"
```

```
            url = uri("file://${buildDir}/repo") //
                localRepo, you can put online repo, too
        }
    }
}
```

Come si può notare dal listato 4.6, una pubblicazione di nome ”metadata” è aggiunta di default per i progetti multipiattaforma, questa racchiude tutte le dichiarazione usate in Kotlin serializzate, ed è utile durante l'importazione nell'IDE del progetto. L'identificativo di questa pubblicazione è formato da ”¡nomeProgetto¿-metadata”, anche le altre pubblicazioni ricevono un identificativo di base, che segue la forma ”¡nomeProgetto¿-¡nomeTargetinMinuscolo¿”.

Listato 4.7: Gli identificativi di base possono essere cambiati

```
kotlin {
    jvm() {
        mavenPublication { // Setup the publication for
            the target 'jvm'
            // The default artifactId was 'foo-jvm6',
                change it:
            artifactId = "foo-jvm"

            // Add a docs JAR artifact (it should be a
                custom task):
            artifact(jvmDocsJar)
        }
    }
}
```

La funzione `pubblishing..` mette a disposizione anche il metod `pom...` per modificare il metadata ”pom.xml” in ogni suo attributo.
In conclusione se per molti target nativi, la pubblicazione è una procedura semplice vista la configurazione di base apportata per i plugin, pubblicare una libreria JS

richiede configurazioni particolari con Task ad-hoc, sarà obbiettivo della prossima sezione comprendere al meglio queste configurazioni.

4.2.2 NPM

NPM, abbreviazione di Node Package Manager, è appunto un gestore di pacchetti per Javascript ed è predefinito per l'ecosistema Node.js. I pacchetti disponibili sono ospitati su un repository interno, che è anche disponibile, tramite comandi shell, ad upload di progetti esterni. Includere queste operazioni nella *build* di un progetto, garantisce una migliore gestione e ripetibilità delle stesse, ciò è possibile attraverso il plugin "com.moowork.node", che permette di scaricare Node.js a livello progettuale ed eseguire script js al suo interno.

Listato 4.8: Plugin "com.moowork.node"

```
plugins {
  id("com.moowork.node") version "1.3.1"
}
```

Il comando per pubblicare su npm è `npm publish` che accetta la cartella contenente il file package.json che rappresenta il metadata che contiene le dipendenze e la descrizione generale dello stesso in formato JSON. I comandi shell sono supportati attraverso i task di tipo "npmTask", il listato 4.9, ne mostra un esempio di creazione.

Listato 4.9: Task per pubblicare un package su npm

```
tasks.register<NpmTask>("pubblish"){
    setWorkingDir(buildDir.toPath().toString() + "/js/
       packages/project")
    setArgs(setOf("publish"))
}
```

Per eseguire il task e pubblicare i sorgenti online con successo, è necessario prima però possedere un account sulla piattaforma npm ed effettuare il login da shell con il comando `npm adduser`.

Le operazioni rappresentate possono essere racchiuse in unico tipo di task, grazie ad un plugin diverso, "com.liferay.gradle.plugins.node", che mette a disposizione diversi task relativi ad npm, tra cui PublishNodeModuleTask, che permette sia di accedere ad npm che pubblicare sul registro un pacchetto. Si presenta più completo rispetto al task del listato 4.9, in quanto consente la generazione del metadata package.json, e la sua configurazione.
In conclusione, queste due fasi sono fondamentali e devono essere susseguite e non indipendenti, JS rimane ancora una parte critica e non totalmente supportata da Kotlin, tuttavia con gli ultimi aggiornamenti (Kotlin 1.3.40) le configurazioni manuali sono ridotte. Nel prossimo capitolo, le notazioni e gli argomenti espressi verranno applicati a due progetti multipiattaforma.

Capitolo 5

Caso di studio: Kt-math

La documentazione e le notazioni espresse nei capitoli precedenti sono finalizzate alla costruzione di diversi aspetti del processo di sviluppo di una libreria multipiattaforma, denominata kt-math. Il progetto in questione consiste in un porting delle classi BigInteger e BigDecimal appartenenti a java.math in Kotlin, in modo da rendere il progetto multipiattaforma ed estendere le funzionalità di queste classi ad altri ambienti sviluppativi. Queste classi superano i limiti tecnici di alcune variabili primitive, permettendo di gestire a livello di programmazione numeri abbastanza grandi anche per long o int, o di usare numeri con una precisione decimale maggiore di float e double. Le applicazioni di questa libreria vanno dal campo finanziario, nella gestione delle valute e dei loro decimali, a quello della crittografia, nella gestione di numeri estremamente grandi.
Il contributo personale si è focalizzato sull'integrazione delle fase di testing e di deploy in ambiente Javascript, che come si è denotato dal capitolo precedente necessita di configurazioni aggiuntive per integrarsi con Gradle e Kotin.
Questo porting eseguito dal dott. Giovanni Ciatto mancava delle configurazioni aggiuntive per i test e i deploy nell'ecosistema JS, il mio contributo è iniziato prima dell'update 1.3.40 di Koltin che ha apportato diversi miglioramenti in questo campo, quindi verranno mostrate due configurazioni diverse, pre e post update.
Dal capitolo precedente, è emersa l'incompletezza della libreria kotlin "js-test" che da sola non riesce a garantire l'esecuzione dei test in Javascript ma necessita di

un framework esterno per completare le funzioni di asserzione, per questo è fondamentale aggiungere alle dipendenze del progetto quello opportuno. La maggior parte dei framework viene distribuito attraverso NPM, per questo il primo passo è stato aggiungere un plugin che includesse la sua logica e permettesse di usufruire l'ambiente node.js, la scelta iniziale è ricaduta su "com.moowork.node". La configurazione, dopo aver aggiunto questo plugin, passa per la creazione di diversi task, il primo nominato "popoulateNodeModules" permette per semplicità nelle operazioni successive di copiare tutte i moduli e le dipendenze del progetto per Javascript nella cartella di build, in questo modo si potrà accedere al framework dopo averlo scaricato.

Listato 5.1: Filtra e copia tutti i file js contenuti nelle dipendenze e usati durante la compilazione del target JS nella cartella node_modules contenuta nella folder build

```
val populateNodeModules = tasks.create<Copy>("
   populateNodeModules") {
    afterEvaluate {
        from(compileOutput)
        from(testOutput)

        configurations["testCompile"].forEach {
            if (it.exists() && !it.isDirectory) {
                from(zipTree(it.absolutePath).matching {
                    include("*.js") })
            }
        }
        for (sourceSet in kotlin.sourceSets) {
            from(sourceSet.resources)
        }
        into("$buildDir/js/node_modules")
    }
    dependsOn(setOf("compileKotlinJs", "
```

```
        compileTestKotlinJs"))
}
```

Una volta predisposto questo metodo, è fondamentale scaricare il framework, in questo caso è stato adoperato Jest, tuttavia è facile cambiarlo con un altro in quanto a livello funzionale il risultato rimane pressoché identico.

Listato 5.2: Jest può essere cambiato con Mocha, Karma, Jasmine o Tape

```
tasks.create<NpmTask> ("installJest") {
    setArgs(setOf("install", "jest"))
}
```

Una volta installato, il plugin "com.moowork.node" mette a disposizione un "NpmTask" utile per eseguire lo script js di Jest, questo è contenuto nella cartella "node_modules" popolata con il task del lisitato 5.1.

Listato 5.3: Lo script accetta come argomento i sorgenti js del test

```
tasks.create<NodeTask> ("runJest") {
    setDependsOn(setOf("installJest", "
       populateNodeModules", "compileTestKotlinJs"))
    setScript(file("node_modules/jest/bin/jest.js"))
    setArgs(compileTestKotlinJs.outputs.files.
       toMutableList().map {projectDir.toURI().
       relativize(it.toURI())})
}

tasks.getByName("test").dependsOn("runJest")
```

Infine impostata la dipendenza tra il task test e runJest, in modo da non escludere i test in Javascript.
L'aggiornamento 1.3.40 di Kotin ha portato con sé una maggiore integrazione dell'ambiente JS in Kotlin, in particolare dei test e del ambiente esecutivo nodejs, questo si traduce in minor codice di configurazione.

Listato 5.4: L'ambiente esecutivo è il browser ma poteva essere adoperato anche nodeJS

```
js {
    browser {
        testTask {
            useKarma {
                useFirefox()
                useChrome()
            }
        }
    }

    ...
}
```

Questa nuova configurazione, sicuramente più sintetica, predilige l'uso di Karma in quanto questo framework è già presente insieme a Mocha nel dominio applicativo del DSL, a differenza degli altri strumenti, inoltre rende obsoleto il plugin "com.moowork.node". Questo è stato sostituito con il plugin "com.liferay.node", che non è stato adoperato nella fase di testing ma bensi durante la pubblicazione su NPM della libreria in JS.

Listato 5.5: I dati di accesso ad NPM non sono stati dichiarati nella build

```
tasks.register<PublishNodeModuleTask>("publishNpmModule"
    ){
    val npmUsername: String? by project
    val npmPassword: String? by project
    val npmEmail: String? by project

    setProperty("workingDir", buildDir.toPath().toString
        () + "/js/packages/experiment")
    setProperty("moduleAuthor", "Ciatto Giovanni")
```

```
    setProperty("moduleDescription", "Pubblish Compiled
       Module to npmjs.com, pass username and password "
        +
            "by arguments or set them in gradle.
               properties in GRADLE_USER_HOME")
    setProperty("moduleName", "kt-math")
    setProperty("moduleKeywords", listOf("kt-math", "
       math", "biginteger", "bigdecimal", "decimal", "
       integer"))
    setProperty("npmEmailAddress", npmEmail)
    setProperty("npmUserName", npmUsername)
    setProperty("npmPassword", npmPassword)
    setProperty("overriddenPackageJsonKeys", setOf("name
       "))
}
```

Le variabili d'accesso a NPM, sono state dichiarate nel file gradle.properties contenuto nella cartella d'installazione ".gradle", ciò è stato necessario per non esporre questi dati privati a Git e quindi a terzi. In conclusione, Kotlin e Gradle si sono rilevati degli strumenti in continuo aggiornamento, fornendo soprattutto Kotlin, attraverso i suoi plugin, delle configurazioni base sempre più performante e complete, minimizzando non soltanto le configurazioni del programmatore ma anche i possibili errori e i tempi di produzione.

Capitolo 6

Conclusioni

L'obbiettivo centrale di questa tesi non era soltanto indagare lo stato d'arte della tecnologia nel campo delle applicazioni multipiattaforma ma anche garantire un ruolo guida nella costruzione di un progetto del genere. Nel secondo capitolo (2), è stata impostata la impalcatura concettuale di un sistema multipiattaforma, mostrando gli strumenti Gradle e Kotlin. Il primo si è rilevato un sistema già navigato nella gestione dello sviluppo del codice, ed è stato fondamentale nella costruzione di un progetto multipiattaforma visto la sua natura estendibile. Infatti, le funzionalità espresse da questo strumento sono in continua crescita attraverso il continuo sviluppo di plugin, che permettono non soltanto di gestire il codice in modo diverso ma di cambiare profondamente l'organizzazione e la struttura interna di un progetto, come in questo caso con il plugin "kotlin-multipiattaforma". Kotlin, che si è mostrato estremamente flessibile, è un linguaggio in grande crescita visti i continui aggiornamenti, e sicuramente è una valida alternativa se non il futuro di molti linguaggi di programmazione nell'ambito delle applicazioni multipiattaforma e non solo. La potenzialità espresse nel setup di un applicazione multipiattaforma come mostrato nel terzo capitolo, sono molteplici e non riguardano soltanto il livello tecnico ma anche operativo di un ipotetico team di sviluppatori, infatti, scrivere in Kotlin vuol dire disinteressarsi della piattaforma di rilascio, e dare maggiore importanza alla logica di base. Questo approccio si traduce in una maggiore coesione del team, che non si deve diversificare necessariamente per ogni

piattaforma. Infine, scrivere un applicativo in Kotlin, oggi si dimostra anche un investimento per il futuro, infatti non è importante capire quale sarà la piattaforma dominante del mercato futuro, perché questo linguaggio potrà supportarla con pochi accorgimenti, indipendentemente che sia il Web, Android o iOS.

Bibliografia

Authoring Multi-Project Builds. URL: https://docs.gradle.org/current/userguide/multi_project_builds.html#sec:configuration_on_demand.

Beams, Chris (2016). *Kotlin Meets Gradle*. Gradle. URL: https://blog.gradle.org/kotlin-meets-gradle.

Belov, Roman (2017). *Kotlin 1.1 Released with JavaScript Support, Coroutines and more*. URL: https://blog.jetbrains.com/kotlin/2017/03/kotlin-1-1/.

Berglund, Tim (2013). *Gradle Beyond the Basics*. 1ª ed. The name of the publisher. ISBN: 1449304672.

Building Multiplatform Projects with Gradle. URL: https://kotlinlang.org/docs/reference/building-mpp-with-gradle.html.

Carbonnelle, Pierre (2019). *PYPL PopularitY of Programming Language*. http://pypl.github.io/PYPL.html. Kudos.

Cleron, Mike (2017). *Android Announces Support for Kotlin*. https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html. Google.

Deleuze, Sébastien (2017). *Introducing Kotlin support in Spring Framework 5.0*. https://spring.io/blog/2017/01/04/introducing-kotlin-support-in-spring-framework-5-0. spring.

GNU Make Manual. URL: https://www.gnu.org/software/make/manual/make.pdf (visitato il 22/05/2016).

Jemerov, Dmitry (2017). *Kotlin 1.2 Released: Sharing Code between Platforms*. URL: https://blog.jetbrains.com/kotlin/2017/11/kotlin-1-2-released/.

Kotlin 1.3.40 released (2019). URL: https://blog.jetbrains.com/kotlin/2019/06/kotlin-1-3-40-released/.

Muschko, Benjamin. *Gradle in Action*. Manning Publications Co. ISBN: 9781617291302.

Paraschiv, Eugen (2017). *Java in 2017 Survey Results*. URL: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm (visitato il 21/02/2018).

Speeding Up Your Android Gradle Builds (Google I/O '17). URL: https://www.youtube.com/watch?v=7ll-rkLCtyk&feature=youtu.be&t=22m20s (visitato il 17/05/2017).

Tobias Heine, Said Tahsin Dane (2018). *(Code) Sharing is caring - An Introduction to Kotlin Multiplatform Projects*. URL: https://blog.novoda.com/introduction-to-kotlin-multiplatform/.

touchlab, cur. (2019). *Cross-Platform? We Don't Say That Around Here Anymore*. URL: https://hackernoon.com/cross-platform-we-dont-say-that-around-here-anymore-cd269fcd4a5b.

Using Gradle Plugins. URL: https://docs.gradle.org/current/userguide/plugins.html#sec:plugins_block.