POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Software e Sistemi Digitali

Tesi di Laurea Magistrale

Studio del linguaggio Kotlin per applicazioni distribuite: Server e Front-End Web

Relatore
Giovanni Malnati

Autore
Vito Scornavacche

A.A.2017/2018

Dediche

Alla mia famiglia che ha sempre saputo distinguere i momenti in cui rassicurarmi da quelli in cui era giusto spronarmi. Senza di loro questa tesi non sarebbe mai stata scritta e io non sarei la persona che sono.

Indice

Elenco delle figure

Listings

Capitolo 1

Introduzione

1.1 Panoramica del linguaggio Kotlin

La storia dell'informatica è caratterizzata dalla continua introduzione di nuovi linguaggi di programmazione, che in alcuni casi si sono rivelati fallimentari e in altri hanno avuto successo; alcuni di questi hanno introdotto novità interessanti, molte delle quali successivamente integrate anche nei linguaggi più datati.
Kotlin è un linguaggio di programmazione molto recente, il cui sviluppo da parte dell'azienda JetBrains è cominciato nel 2011 ed è ancora in corso; è open-source, fortemente tipizzato e orientato alla programmazione a oggetti. Uno dei requisiti iniziali del progetto era la piena compatibilità con il linguaggio Java, in modo da sfruttarne le librerie e i tool già pronti in grande quantità, vista la sua lunga e diffusa presenza nel mercato. Kotlin quindi è stato pensato per girare sulla Java Virtual Machine e per interoperare con la Java Runtime Environment. In quest'ottica è dichiarata da parte degli sviluppatori la possibilità di migrare a Kotlin qualunque applicativo Java; viene fornito persino un tool che permette questa trasformazione in automatico.
Contemporaneamente era necessario superare alcuni dei problemi comuni sul Java. Kotlin è molto conciso: si stima che vengano scritte in media il 40% [6] di righe di codice in meno rispetto al Java. Il sistema dei tipi evita il presentarsi di problemi fin troppo noti ai programmatori Java, come ad esempio i puntatori a *null*. Kotlin è caratterizzato, inoltre, da una grande flessibilità, permettendo il pieno utilizzo sia della programmazione a oggetti sia di quella funzionale.
Gli sviluppatori della Jetbrains, inoltre, puntano molto sullo sviluppo Android. La stessa Google nel maggio del 2017 ha cominciato a supportare ufficialmente Kotlin per il proprio sistema operativo mo-

bile [2]. La stessa Jetbrains per la KotlinConf del 2017 ha sviluppato un'applicazione ufficiale della conferenza in Android e iOS, con lo scopo di dimostrare le potenzialità del linguaggio.
Il compilatore Kotlin consente anche la generazione di codice Javascript, permettendo il suo utilizzo per lo sviluppo di applicazioni web.
Il linguaggio Kotlin è attualmente ancora in sviluppo e vengono spesso introdotte nuove funzionalità; al momento alcune di queste sono ancora in fase sperimentale, incluse alcune di quelle utilizzate per il lavoro svolto in questa tesi.

1.2 Applicazioni distribuite

[10] Le applicazioni distribuite sono software strutturati in due o più processi in comunicazione tra di loro tramite un canale condiviso. Il vantaggio principale di questa architettura software è sicuramente la possibilità di dividere le diverse parti dell'applicativo su più macchine, anche se non è una condizione necessaria.
Le applicazioni distribuite si dividono in due tipologie:

- **Client-Server**: è strutturata su due livelli, in cui il client invia delle richieste che il server si occupa di servire.
- **Multi-livello**: si ha un numero di livello superiore a due e viene utilizzata soprattutto per alleggerire il carico sui server, distribuendo le operazioni necessarie per servire le richieste su più livelli.
- **Completamente distribuite**: in questo caso si ha che ogni livello non risiede più su un singolo nodo, ma è distribuito geograficamente, con la distribuzione del carico applicativo che diviene la massima possibile.

La più usata è sicuramente la struttura client-server. Su ogni device può essere eseguito un sistema differente e addirittura le stesse parti dell'applicativo possono essere scritte in linguaggio diverso. Data la crescente complessità delle applicazioni, la programmazione distribuita è ormai diventata onnipresente sulla scena informatica; è necessaria quindi una conoscenza ampia e diversificata di diversi strumenti per poter sviluppare una singola applicazione.

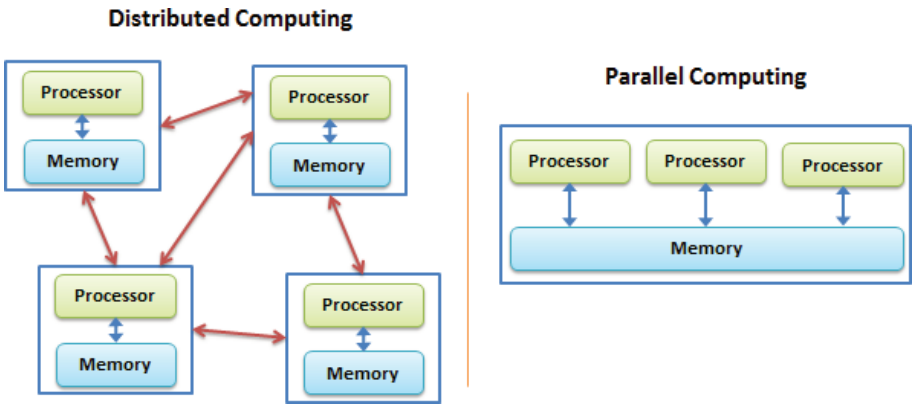

Figura 1.1: Differenze tra applicazioni distribuite e parallele [9]

1.3 Scopo della tesi

Una volta introdotti gli strumenti che il linguaggio Kotlin promette e la flessibilità necessaria per lo sviluppo di applicazioni distribuite, si comincia a delineare la possibilità di utilizzare questo nuovo linguaggio per limitare l'ampia gamma di conoscenze necessarie per la programmazione distribuita. Lo scopo di questa tesi è proprio lo studio delle funzionalità e degli strumenti offerti (ora o in futuro) dal linguaggio Kotlin per verificare che possa essere utilizzato nello sviluppo delle varie parti delle applicazioni distribuite. L'obiettivo è quello di verificare se a tendere potrebbe essere possibile per gli sviluppatori utilizzare solo il linguaggio Kotlin per lo sviluppo delle diverse parti dell'applicazione distribuita, con la ovvia conseguenza che non saranno più necessarie tante competenze diverse durante la fase di sviluppo.
Verrà quindi studiato a livello teorico il linguaggio e gli strumenti utilizzabili. Successivamente verranno realizzate un server e un'applicazione web come esempio di applicazione distribuita; è in programma anche un'applicazione Android che è stata sviluppata nell'ambito di un altro progetto di tesi. Lo sviluppo non ha scopo di prodotto, ma è finalizzato a verificare a seguito di requisiti verosimili l'effettiva efficacia del programma e la correttezza dello studio precedentemente effettuato.
A seguito di questo lavoro verrà data una valutazione della situazione attuale del linguaggio ed eventuali direzioni da prendere per raggiungere l'obiettivo di utilizzare esclusivamente Kotlin per lo sviluppo di tutte le parti di un'applicazione distribuita.

1.4 Organizzazione della tesi

La tesi comincerà con una panoramica del linguaggio Kotlin, in cui verrà esposto lo studio fatto sulla sintassi del linguaggio e sugli strumenti che mette a disposizione.
Nel capitolo successivo saranno presentate analisi, progettazione e implementazione del progetto realizzato, introducendo anche gli strumenti e i framework utilizzati.
Nella terza parte saranno esposte delle valutazioni dell'autore sullo studio fatto e sul lavoro svolto; il tutto nell'ottica di ricercare eventuali problematiche o limitazioni nell'utilizzo di Kotlin per lo sviluppo di applicazioni distribuite.
La quarta e ultima parte tratterà delle conclusioni a cui l'autore è giunto e degli eventuali sviluppi futuri che possano approfondire altri ambiti di ricerca per l'utilizzo del linguaggio Kotlin.

Capitolo 2

Kotlin

2.1 Storia del linguaggio

La JetBrains, precedentemente nota come IntelliJ, è un'azienda informatica nata nel 2000 e che si occupa principalmente della realizzazione di ambienti di sviluppo per linguaggi di programmazione diffusi. Il progetto principale dell'azienda è IntelliJ IDEA, IDE per il linguaggio Java che si è evoluto nel tempo fino a supportarne diversi altri, come ad esempio Scala o Clojure. La stessa Google si è basata sull'IDE di JetBrains per sviluppare il proprio ambiente proprietario per lo sviluppo di applicazioni Android.
Dato che IntelliJ IDEA è sviluppato in Java, questo era il principale linguaggio utilizzato da Jetbrains, ma i suoi limiti erano evidenti e si è quindi presentata la necessità di cambiare. Non fu un passaggio facile, dato che i requisiti che il nuovo linguaggio avrebbe dovuto avere erano piuttosto stringenti:

- Sarebbe dovuto essere totalmente compatibile con Java, dato che era già stata sviluppata una mole non indifferente di strumenti che giravano sulla JVM (Java Virtual Machine).
- Era molto importante anche la compilazione statica, per evitare tutti quei problemi relativi ai tipi che affliggevano il Java.

Vennero valutati molti linguaggi, ma nessuno di questi venne reputato sufficiente per poterlo utilizzare.
Quindi, nel 2011, Jetbrains decise di sviluppare un proprio linguaggio progettato sui requisiti stringenti che si erano imposti. Così nacque Kotlin, progetto open-source che vide il rilascio della prima versione stabile nel 2016, ottenendo fin da subito un discreto successo. In ambito Android soprattutto è molto utilizzato fino ad ottenere il supporto

ufficiale della stessa Google durante la Google I/O del 2017.
Essendo un progetto open-source, l'intero codice sorgente del linguaggio è disponibile su GitHub, anche se il suo sviluppo è guidato da Jetbrains e portato avanti quasi esclusivamente da loro. Nel corso del tempo, anche dopo il rilascio, sono state aggiunte sempre più funzioni, sperimentali e non, che contribuiscono ad arricchirne l'offerta e di conseguenza l'appetibilità.

2.2 Premessa alla descrizione del linguaggio

La spiegazione tecnica del linguaggio affrontata in questa tesi non ha lo scopo di descrivere interamente il linguaggio e tutte le sue caratteristiche, come se fosse diretta a un neofita della programmazione; in larga parte è simile alla maggioranza dei linguaggi moderni. L'obiettivo è quello di introdurre Kotlin a chi ha già dimestichezza nella programmazione a oggetti, evidenziandone le particolarità e le caratteristiche che lo contraddistinguono.

2.3 Sintassi di base

[5][14] L'analisi tecnica comincia per ovvi motivi con una panoramica sulla sintassi di base del linguaggio. Nonostante mantenga alcune caratteristiche e parole chiave della maggior parte dei linguaggi moderni (che si ispirano al C per le istruzioni di base), Kotlin introduce alcune novità che hanno soprattutto l'obiettivo di snellire la scrittura del codice. Esempio topico è l'eliminazione del punto e virgola alla fine dell'istruzione.

2.3.1 Variabili

Le variabili in Kotlin possono essere di due tipologie:

- **val**: In questo caso la variabile può essere assegnata solo una volta e non può più essere modificata; in pratica sono delle vere e proprie costanti.

- **var**: In questo caso, invece, la variabile può essere modificata ogni volta che il programmatore desidera.

Come già detto in precedenza, Kotlin è un linguaggio fortemente tipizzato; quindi ogni variabile ha un tipo ben definito che può essere specificato in fase di dichiarazione nel seguente modo:

Listing 2.1: Dichiarazione di una variabile

```
var nomeVariabile: tipoVariabile
```

Il compilatore può ricavare il tipo della variabile direttamente dall'istruzione di assegnazione, nella grande maggioranza dei casi; di conseguenza è possibile omettere la dichiarazione del tipo:

Listing 2.2: Assegnazione di una variabile

```
val nomeVariabile = Integer(1)
```

2.3.2 Espressioni condizionali

Le espressioni condizionali in Kotlin sono quasi totalmente identiche al C, e quindi anche alla maggior parte dei linguaggi moderni. Di seguito vengono presentati di conseguenza solo i casi particolari; **if**, **while** e le altre istruzioni di base non verranno esposte per ovvi motivi.

Il *for* viene usato esclusivamente per scorrere collezioni di oggetti tramite un iteratore. La sintassi è la seguente:

Listing 2.3: Istruzione for

```
for (item: String in collection ) {
    // codice ciclo
}
```

In questo modo ad ogni ciclo viene preso l'elemento seguente di **collection** e può essere manipolato all'interno del ciclo tramite l'oggetto **item**; da notare che **item** è di tipo String, quindi questo vuol dire che la collezione deve contenere solo elementi di tipo String, altrimenti il compilatore dà errore.

L'istruzione *when* prende il posto del classico **switch-case** e viene usato quando occorre svolgere azioni diverse basandosi sul valore di una variabile. La sintassi è la seguente:

Listing 2.4: Istruzione when

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

Se il valore di x è 0 o 1 allora verrà eseguita la relativa istruzione. *else* è usato come **default**: quando x non rientra in nessuno dei casi

definiti, viene eseguita questa istruzione. E' concesso anche l'utilizzo di funzioni che tornino un determinato valore.
E' possibile utilizzare anche delle parole chiave che definiscono delle condizioni:

- **in**: Definisce se la variabile considerata si trova o meno all'interno di una collection
- **is**: Definisce se la variabile considerata appartiene o meno a una determinata classe

Listing 2.5: Utilizzo di in

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

Listing 2.6: Utilizzo di is

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

2.3.3 Collezioni

In Kotlin le collezioni possono essere mutabili o non mutabili.

- **mutabili**: Possono essere modificate dopo la loro definizione.
- **non mutabili**: usate solo per operazioni di lettura dato che non possono essere modificate dopo aver fatto l'assegnazione.

Tutte le classi che rappresentano una collezione sono figlie della classe **Collection**; sono presenti tutte le collezioni tipiche dei linguaggi di programmazione: **List**, **Set**, ecc (e le relative versioni mutabili: **MutableList**, **MutableSet**, ecc).

Listing 2.7: Collezioni non mutabili

```
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
fruits
  .filter { it.startsWith("a") }
  .sortedBy { it }
  .map { it.toUpperCase() }
  .forEach { println(it) }

fruits.clear() //Does not compile
```

Alla variabile *fruits* viene assegnata una collection tramite la funzione *listOf*, su cui è possibile richiamarsi diversi metodi. In questo caso abbiamo una lista non mutabile, su cui quindi non potranno essere usate funzioni tipo l'aggiunta o la cancellazione di un elemento; infatti l'ultima riga, in cui viene usata la funzione *clear* non verrà compilata.

Listing 2.8: Collezioni mutabili

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
println (numbers)        // prints "[1, 2, 3]"
numbers.add(4)
println (readOnlyView)   // prints "[1, 2, 3, 4]"
```

In questo caso è possibile aggiungere o rimuovere un elemento dato che la lista è stata inizializzata con la funzione *mutableListOf*.

2.3.4 Eccezioni

In Kotlin le eccezioni sono gestite in modo molto simile al Java. Sono tutte figlie di una stessa classe, **Throwable**, e contengono uno stack trace per poter risalire al punto da cui sono state lanciate.
Per catturare una eccezione si usa l'espressione try/catch.

Listing 2.9: Eccezioni

```
try
{
    //try body
}
catch(e: Exception)
{
    //catch body
}
 finally
{
    //finally body
}
```

Viene eseguito un codice all'interno del blocco *try* e se questo scatena un'eccezione, questa viene catturata ed viene eseguito il blocco *catch* corrispondente. Possono essere messi più blocchi *cathc* in cascata, in modo tale da avere blocchi diversi per eccezioni diverse; in questo caso viene eseguito il primo blocco che rispetta i requisiti dell'eccezione che è stata lanciata. Il blocco *finally* viene eseguito comunque alla fine del codice precedente, sia nel caso in cui è stata lanciata un'eccezione, sia nel caso contrario.
Per sollevare un'eccezione si usa la parola chiave *throw*:

Listing 2.10: Uso di Throw

```
val n:Double = 10.0
if (n==0.0)
    throw Exception("Exception")
```

```
print(1/n)
```

2.3.5 Gestione dei null

null è utilizzata in quasi tutti i linguaggi per indicare un oggetto o una variabile che non ha un valore consistente; in pratica è un puntatore che non punta a una locazione di memoria. Gli ideatori del Kotlin avevano il preciso obiettivo di superare il problema tipico dei null, cioè quando si cerca di utilizzarli come se puntasserro a un valore.
Il sistema dei tipi di Kotlin di conseguenza impone che i tipi potenzialmente null vengano dichiarati a tempo di compilazione.

Listing 2.11: Variabili nullable

```
var x : String //variabile non nullable
var x : String? //variabile che pu essere nullable
```

Se una variabile viene dichiarata nullable, allora sarà compito del programmatore gestirla. Se invece viene dichiarata non nullable, allora il compilatore lancerà un eccezione nel caso vengano scritte istruzioni che rischiano di rendere la variabile *null*. Questo previene il presentarsi del temuto *NullPointerException*, perché viene segnalato già a tempo di compilazione.
Si presentano casi in cui una variabile nullable deve essere usata come se non lo fosse.

Listing 2.12: Imporre una variabile nullable

```
x!!.length
x?.length
```

Con *!!* si impone al compilatore di accettare l'istruzione. In questo caso il programmatore si assume la responsabilità di quello che sta facendo e deve essere totalmente certo che la variabile non è **null**; se così non fosse viene lanciata un'eccezione. Nel caso di *?* l'istruzione viene eseguita solo se la variabile non è **null**, altrimenti viene ignorata.

In ambito di variabili che possono essere **null**, si rivela di grande utilità l'operatore elvis: *?:*

Listing 2.13: Operatore elvis

```
val l = b?.length ?: -1
```

Che equivale all'istruzione:

Listing 2.14: Verifica null

```
val l: Int = if (b != null) b.length else -1
```

In questo caso nella variabile *l* viene salvata la lunghezza di b solo se b non è **null**, altrimenti viene salvato -1.

2.4 Funzioni

[5] La programmazione funzionale è un paradigma di programmazione in cui il flusso di esecuzione è rappresentato da una continua valutazione dei risultati di diverse funzioni dopo aver dato loro dei valori di ingresso. Tramite questo tipo di programmazione è possibile ottenere un codice conciso, dato che stesse operazioni ripetute molte volte sono concentrate in un'unica funzione in cui cambiano solo i valori di ingresso.
In Kotlin una funzione si dichiara nel seguente modo:

Listing 2.15: Dichiarazione di una funzione.

```
fun funName(valName: String) : String
{
 return valName
}
```

Le variabili all'interno delle parentesi tonde sono gli argomenti della funzione, cioè i valori di ingresso passati alla funzione; possono essere più di uno e per ognuno di essi ne è definito il tipo.
Il tipo di ritorno della funzione è specificato dopo i due punti, ma può essere omesso nel caso in cui per il compilatore è palese. Può servire anche una funzione che non abbia un valore di ritorno; in questo caso viene utilizzata la parola chiave *Unit* come tipo di ritorno.
Ci sono ulteriori modi per dichiarare una funzione che potrebbero essere utili per snellire un po' il codice scritto in alcuni casi particolari.

Listing 2.16: Dichiarazione di una funzione.

```
fun funName(valName: String)= valName
```

In questo caso il corpo della funzione viene descritto esclusivamente dopo l'uguale. Ovviamente può essere usato nei casi in cui il corpo è molto breve ed è possibile utilizzare una sola istruzione.

Listing 2.17: Dichiarazione di una funzione.

```
fun funName(valName: String = "value")= valueName
```

In questo secondo caso, viene assegnato un valore all'argomento in ingresso della funzione. Questo viene inteso come valore di default: se la funzione viene chiamata passando lo specifico argomento, si utilizza quello, altrimenti viene utilizzato il valore di default. In questi casi è

buona norma inserire gli argomenti con valore di default alla fine, in modo tale che è possibile chiamare la funzione con soli gli argomenti obbligatori, omettendo quelli che non lo sono.
Kotlin mette a disposizione anche gli argomenti nominati: cioè al momento in cui si richiama una funzione, si può nominare ogni parametro passato.

Listing 2.18: Argomenti nominati.

```
funName(valName=value)
```

In questo modo si assegna in modo esplicito il valore all'argomento. Ciò permette di passare gli argomenti della funzione in ordine differente rispetto alla dichiarazione.

Le funzioni in Kotlin possono essere di diversi tipi:

- **Funzioni locali**: dichiarate all'interno di altre funzioni.
- **Funzioni membro**: dichiarate all'interno di una classe.
- **Funzioni ad alto livello**: non sono dichiarate all'interno di altri elementi.

2.4.1 Funzioni Higher-Order

Kotlin fornisce una strumento molto potente in ottica di programmazione funzionale: le **Higher-Order functions**. Servono nei casi in cui si ha la necessità di passare una funzione come argomento a un'altra. Questo permette di poter passare e, di conseguenza, applicare funzioni diverse in base alle condizioni volute.
Di seguito viene visualizzato come passare una funzione come argomento:

Listing 2.19: Higher-Order come argomento

```
fun funName (higherOrderFun : (arg1, arg2) -> returnType)
{
    // body function
}
```

Col codice seguente, invece, abbiamo una funzione come tipo di ritorno:

Listing 2.20: Higher-Order come tipo di ritorno

```
fun funName (param1: String):(param) -> valoreRitorno
{
    // body function
}
```

2.4.2 Funzione Lambda

Una funzione lambda è una funzione non dichiarata, anonima e il cui corpo è esplicitato direttamente all'interno del codice in cui viene chiamata. Viene utilizzata nei casi in cui si ha bisogno di utilizzare una funzione molto semplice e che non verrà riutilizzata nuovamente.

Listing 2.21: Funzione Lambda

```
1 val sum = { x: Int, y: Int -> x + y }
```

La funzione lambda è quella tra parentesi graffe, riceve in ingresso due argomenti di tipo *int* (x e y) e ritorna la loro somma, che viene salvata nella variabile **sum**.
Nel caso in cui si ha un solo argomento in ingresso può essere usata la parola chiave **it** per utilizzare il parametro all'interno del corpo della funzione, omettendo gli argomenti.

Listing 2.22: it nelle funzioni lambda

```
1 ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

Da notare che se viene utilizzata la parola chiave **return** all'interno di una funzione lambda, non si esce dalla funzione lambda stessa, ma da quella che la contiene.

2.4.3 Funzioni Anonime

Il limite delle funzioni lambda è l'impossibilità di esplicitare il tipo di ritorno della funzione; solitamente viene dedotto dal compilatore stesso, ma ci sono casi in cui questo non è possibile.
Per questi casi vengono utilizzate le funzioni anonime:

Listing 2.23: Funzioni Anonime

```
1 ints.filter (fun(item) = item > 0)
```

Sono molto simili alla dichiarazione delle normali funzioni, tranne per il fatto che il nome viene omesso. Anche in questo caso, però, il tipo di ritorno può essere omesso, se si può facilmente dedurre dal corpo della funzione.
In questo caso, la parola chiave **return** all'interno della funzione anonima fa ritornare il flusso di esecuzione alla funzione che la contiene, a differenza delle lambda.

2.4.4 Funzioni Inline

Le Inline sono funzioni particolari che non vengono compilate normalmente, ma la loro chiamata viene sostituita con il codice stesso della funzione. Questo comporta un aumento delle righe di codice della funzione da cui vengono chiamate, ma evita tutte quelle operazioni che vengono effettuate automaticamente alla chiamata di una funzione. Di solito vengono utilizzate quando si ha un corpo ridotto per la quale non è conveniente effettuare tutte quelle operazioni che avvengono al momento della chiamata di una funzione.
Vengono definite aggiungendo la parola **inline** alla dichiarazione della funzione.

2.4.5 Modificatore tailrec

Nella programmazione funzionale la ricorsione, cioè quando una funzione richiama se stessa, è molto utilizzata; soprattutto nei casi in cui bisogna ripetere la stessa operazione definita dalla funzione più volte. Kotlin fornisce uno strumento utile in questi casi: se la chiamata a funzione viene fatta utilizzando il modificatore **tailrec**, il compilatore ottimizza il codice in modo tale da ottenere una struttura a cicli invece che una ricorsiva chiamata a funzioni, evitando i problemi di stack overflow.
Il caso in cui il **tailrec** funziona meglio è quando la chiamata ricorsiva viene fatta alla fine della funzione stessa.

Listing 2.24: modificatore tailrec

```
1 val eps = 1E-10 // "good enough", could be 10^-15
2
3 tailrec fun findFixPoint(x: Double = 1.0): Double
4         = if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

2.5 Programmazione ad Oggetti

[5] La programmazione a oggetti si basa sul concetto di oggetto, che viene definito come una struttura di variabili che ne definisce lo stato e funzioni che si possono applicare ad esso e che lo fanno cambiare. Ogni oggetto può interagire con altri utilizzando le funzioni per manipolarne lo stato.
La programmazione a oggetti si basa su alcuni concetti fondamentali:

classe, oggetti, ereditarietà e polimorfismo. In Kotlin gli attributi si chiamano *proprietà*, mentre le funzioni prendono il nome di *metodi*.

2.5.1 Package

In informatica il package è uno spazio di nomi a cui è possibile assegnare i diversi elementi che compongono il programma. In questo modo è possibile dividere il programma in diversi spazi di nomi in base alle varie correlazioni logiche tra classi diverse. In kotlin la definizione del package avviene nel seguente modo:

Listing 2.25: Definizione di un package

```
1 package nomePackage
```

Normalmente la visibilità degli elementi per una classe si limitano al proprio package; per poter usare altre classi si deve quindi specificarne l'intero percorso dello spazio dei nomi o in alternativa importare il relativo package:

Listing 2.26: Import del package

```
1 import nomePackage
```

2.5.2 Classe

Una classe è la struttura di base su cui si basano i relativi oggetti che vengono creati con il meccanismo dell'instanziazione.

Per la dichiarazione delle classi in Kotlin si utilizzano le seguenti istruzioni:

Listing 2.27: Dichiarazione di una classe.

```
1 class nomeClasse
2 {
3     private var nomeProprieta: int
4
5     public fun nomeMetodo(val nomeAttributo): valoreRitorno
6     {
7         //Codice metodo
8     }
9 }
```

Modificatori

Ogni proprietà o metodo può essere associata a dei modificatori di visibilità che ne definiscono la visibilità:

- **private**: visibile solo all'interno del file che contiene la dichiarazione

- **internal**: visibile solo all'interno dello stesso modulo
- **protected**: non è visibile nelle dichiarazioni di livello superiore
- **public**: visibile a tutti

Classi innestate

Una classe è detta **nested** se è dichiarata all'interno del body di un'altra classe.

Listing 2.28: Classe nested

```
class Class {
    class Nested {
    }
}
```

Se è dichiarata con la parola chiave *inner*, una classe nested può utilizzare le proprietà e i metodi della classe esterna.

Listing 2.29: Classe inner

```
class Class {
    inner class Inner {
    }
}
```

Costruttori

I costruttori sono metodi particolari utilizzati nella maggior parte dei linguaggi a oggetti. All'interno di questi metodi viene implementato il codice che viene lanciato nel momento in cui viene instanziato un oggetto della classe a cui il costruttore appartiene. In Kotlin possono essere definiti costruttori primari e secondari. Il costruttore primario è unico per ogni classe e viene chiamato ogni volta che viene creato un oggetto.

Listing 2.30: Costruttore primario.

```
class ClassName constructor(nameVar : String)
{
    //class code
}
```

In questo caso il costruttore primario non contiene codice, perché viene generato in automatico dal compilatore in relazione ai parametri passati. Nel caso si voglia inserire all'interno del costruttore primario del codice particolare, viene utilizzato **init**:

Listing 2.31: Init del costruttore primario

```
class ClassName(name: String) {
    val prop = name

    init {
        println(prop)
    }
}
```

In questo modo il codice del blocco **init** viene eseguito ogni volta che un oggetto viene instanziato. I costruttori secondari possono essere più di uno e vengono dichiarati utilizzando l'istruzione *constructor*:

Listing 2.32: Costruttore secondario

```
class ClassName (val name: String){
    val prop = name

    constructor (name: String) : this(name) {
        //constructor code
    }
}
```

In questo modo viene utilizzato il relativo costruttore in base ai parametri passati. Ogni costruttore secondario deve obbligatoriamente richiamare quello primario tramite la parola chiave *this*.

Oggetti

Un oggetto è l'instanziazione di una classe, cioè una struttura composta da uno stato ben definito all'interno del programma e una serie di metodi utilizzabili per manipolarlo, che vengono definiti a partire dalla classe di origine.

In Kotlin si instanzia un oggetto con la seguente istruzione:

Listing 2.33: Instanziazione di un oggetto

```
val oggetto = Classe()
```

Si noti che non è più necessaria la parola chiave *new* come in altri linguaggi.

Controllo dei tipi e cast

In certe situazioni potrebbe essere necessario verificare a runtime la classe a cui un oggetto appartiene. Questo è possibile farlo tramite la parola chiave *is*

Listing 2.34: Istruzione is

```
if(x is String)
{
    println(x.lenght) //cast automatico
}
```

In questo caso se la x è un oggetto appartenente alla classe String, la condizione è verificata e si entra nel corpo dell'*if*.
Inoltre il compilatore di Kotlin gode di una funzionalità che potremmo chiamare cast automatico; cioè all'interno dell'if è possibile usare la variabile x come String in automatico, dato che siamo sicuri appartenga a quella classe.
Se si prova a fare un cast che non è possibile, viene generata un'eccezione. Per questo viene usata la parola chiave *as* in caso di cast non certi.

Listing 2.35: Istruzione as

```
1 val x: String = y as String
```

Overloading degli operatori

Gli operatori possono essere trattate come dei metodi da applicare a degli oggetti. Per questo è possibile modificare le funzioni degli operatori che si usano su di esso; in questo modo è possibile specificare un comportamento particolare che quest'ultimo deve avere quando gli operatori vengono richiamati.

Per fare ciò si utilizza la parola chiave *operator*.

Listing 2.36: Overloading degli operatori

```
1 class Class(val number: Int)
2 {
3     operator fun plus(increment: Int) : Int
4     {
5         return (number + increment)
6     }
7 }
```

In questo modo quando viene chiamato l'operatore + su un oggetto di tipo *Class*, l'operazione ritorna la somma di *number* più il numero a cui viene sommato.

Data Class

Le *data class* in Kotlin sono delle classi che hanno come unico scopo quello di contenere una struttura dati e poterla utilizzare e manipolare.

Listing 2.37: Data class

```
1 data class User(val name: String, val age: Int)
```

Una data class deve rispettare i seguenti requisiti:

- Il costruttore primario deve avere almeno un parametro; tutti i parametri devono essere marcati come val o var.
- Si possono implementare solo interfacce; non possono essere astratte, aperte, innestate o sigillate.

Ad ogni dichiarazione di data class il compilatore crea in automatico i metodi *equals()*, *hashCode()*, *toString()*, le *componentN()* e *copy()*. *equals* è usata per confrontare due oggetti dello stesso tipo e definire se sono uguali o meno. *hashCode* definisce un code hash per verificare l'integrità dell'oggetto. *toString* ritorna una stringa a partire dai dati della classe. *copy* crea semplicemente una copia dell'oggetto. Per quanto riguarda le funzioni componentN(), servono a selezionare i componenti della classe (oggetto.component1() seleziona il primo componente).
Il compilatore crea in automatico la classe, i costruttori, le proprietà e relativi getter e setter.

Classi sealed

Le classi sealed rappresentano una gerarchia di classi soggetta a restrizioni. In pratica i tipi che può assumere un oggetto che appartiene alla classe *sealed* sono limitati a quelli definiti dal programmatore.

Listing 2.38: Sealed class

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1:Expr, val e2:Expr) : Expr()
object NotANumber: Expr()
```

Questo definisce che un oggetto di tipo *Expr* dato che è *sealed* non può essere instanza di altre classi oltre a *Const*, *Sum* e *NotANumber*.

Companion object

Dato che in Koltin non si possono avere classi con metodi statici, è stato introdotto il costrutto *companion object*.

Listing 2.39: Companion Object

```
class Class {
    companion object factory {
        fun create(): Class = Class()
    }
}
```

In questo modo la funzione *create* può essere chiamata staticamente, cioè senza dover necessariamente instanziare la classe.

2.5.3 Ereditarietà e Polimorfismo

Nella programmazione a oggetti l'ereditarietà è una relazione padre-figlio stabilita tra due classi diverse. La superclasse (o classe padre) definisce dei metodi e degli attributi che la sottoclasse (o classe figlia) deve implementare. Tutte le classi di Kotlin hanno come classe padre *Any*.

In Kotlin non possono essere ereditate tutte le classi, ma solo quelle definite con la parola chiave *open*.

Listing 2.40: Classe open

```
1 open class Base(p: Int)
2
3 class Derived(p: Int) : Base(p)
```

Nel caso in cui la classe padre ha un costruttore primario, i suoi tipi devono essere passati anche alla classe figlia affinchè questa li inizializzi richiamando in automatico il costruttore primario della superclasse. In caso di costruttori secondari si deve utilizzare la parola chiave *super*.

Listing 2.41: Richiamare il costruttore della superclasse

```
1 open class Base(p: Int)
2 {
3     constructor(str : String){}
4 }
5
6 class Derived(p: Int) : Base(p)
7 {
8     constructor(str : String) : super(str)
9 }
```

super si utilizza anche per richiamare i normali metodi della superclasse che ovviamente sono presenti anche nella sottoclasse.

Override

A volte è necessario che una sottoclasse che eredita un determinato metodo abbia la necessità di far avere allo stesso un comportamento diverso. In questo caso si utilizza il concetto di override; cioè si sovrascrive un metodo cambiandone il comportamento. Per rendere possibile l'override i metodi devono avere la parola chiave *open* altrimenti non possono essere sovrascritti.

Listing 2.42: Override di un metodo

```
1 open class Base {
2     open fun overMethod() {}
3     fun notOverMethod() {}
4 }
5 class Derived() : Base() {
6     override fun overMethod() {}
7 }
```

Quando si fa l'override di un metodo, esso stesso è considerato *open*. Per evitare che si possa sovrascrivere nuovamente bisogna utilizzare la parola chiave *final*.

Metodi e Classi astratte

Un metodo astratto è una funzione che non è stata implementata; in pratica non ha corpo e non può essere utilizzato in un'instanza. Si definisce un metodo astratto utilizzando la parola chiave *abtract*. In questo caso non c'è bisogno nemmeno di utilizzare *open*, che può essere omesso.
Una classe astratta è contiene almeno un metodo astratto e proprio per questo motivo non può essere instanziata. Per definirla viene utilizzato lo stesso modificatore *abstract* alla dichiarazione. Dato che non può essere instanziata, viene usata per essere ereditata da altre classi che possono implementare i metodi astratti.

Interfacce

Le interfacce sono delle classi che non possono salvare uno stato e che quindi devono avere tutte le proprietà segnate come *abstract*. Possono avere sia metodi astratti che non, ma devono essere tutti *open*.

L'interfaccia si dichiara nel seguente modo:

Listing 2.43: Dichiarazione di un'interfaccia

```
interface MyInterface {
    fun bar()
    fun foo() {
      // method body
    }
}
```

E di seguito ci sono le istruzione utilizzate per implementarla in una classe:

Listing 2.44: Implementazione di un'interfaccia

```
class Child : MyInterface {
    override fun bar() {
        // method body
    }
}
```

Le interfacce possono esse stesse implementarne altre.

Estensioni

In Kotlin è possibile nuovi metodi alle classi senza ereditarne di nuove, tramite le estensioni.

Listing 2.45: Estensione

```
fun Class().newMethod()
{
    //body
}
```

In questo modo la classe è possibile chiamare il nuovo metodo utilizzando **Class**. Da notare che questa non viene modificata; cioè non viene aggiunto un nuovo metodo alla classe, ma semplicemente si permette la chiamata al metodo utilizzando la notazione puntata con **Class.newMethod**. L'estensione viene elaborata a tempo di compilazione.
Nel caso venga usato un nome che è già utilizzato per un metodo della classe, quest'ultimo ha la precedenza nelle chiamate successive.

Polimorfismo

Il polimorfismo è la proprietà degli oggetti di essere instanze di più classi che hanno tra loro una relazione di ereditarietà. Cioè in pratica un oggetto può essere considerato instanza sia della classe padre che di quella figlia.

Listing 2.46: Polimorfismo

```
open class Base ()
{
    open fun print = print("base")
}

class Derived () : Base
{
    override fun print = print("derived")
}

fun main (args : Array<String>)
{
    val base : Base = Base()
    val derived : Base = Derived()

    println(base.print()) //print "base"
    println(derived.print()) //print "derived"
}
```

In questo esempio viene visto come anche se entrambi gli oggetti sono dello stesso tipo (**Base**), si comportano in maniera diversa, perché sono stati instanziati a partire da classi diverse; il compilatore è conscio di questo e chiama il body giusto del metodo in base a come è avvenuta l'instanza.

2.5.4 Generics

In Kotlin i generics sono i tipi utilizzati quando non si vuole legare una classe ad un determinato tipo di dato. Alla definizione di una classe, quindi, è possibile specificare che essa dipende da un parametro il cui tipo ancora non è stato definito e che verrà fatto solo al momento dell'utilizzo.

Listing 2.47: classe Generica

```
class Box<T>(t: T) {
    var value = t
}
```

In questo modo si dichiara che a questa classa possono essere passati dati di ogni tipo, che viene identificato con T e può appartenere a qualunque classe, a patto che sia coerente con il codice della classe:

Listing 2.48: Utilizzo dei generics

```
val box: Box<Int> = Box<Int>(1)
```

Variance

In Kotlin ad una classe che implementa generics possono essere passate anche eventuali superclassi o sottoclassi. Questo però potrebbe portare a problemi sui tipi a runtime: non è concesso ad esempio provare a richiamare un metodo su una superclasse di un oggetto passato alla funzione, perchè potrebbe non esistere.
Per poter passare in sicurezza classi che sono in relazioni con il generic si introduce il concetto di limite: si limita alle sole sottoclassi se so che all'interno della classe l'oggetto viene "consumato". Per fare ciò vengono utilizzati i modificatori **in** e **out**:

Listing 2.49: Esempio di una classe in e out

```
interface Source<out T> {
    fun nextT(): T
}

interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}
```

- in: viene usato per limitare gli oggetti passati alle sole sottoclassi del generic. Si usa nei casi in cui la funzione "consuma" l'oggetto, cioè utilizza dei metodi sull'oggetto passato; questo ovviamente potrebbe creare problemi nel caso di una classe padre dato che potrebbe non avere il metodo chiamato.

- out: viene usato per limitare gli oggetti passati alle sole superclassi del generic. Si usa nei casi in cui la funzione "produce" l'oggetto, cioè può essere passato solo al costruttore della classe e non ai metodi. In questo caso il valore che viene ritornato non può essere salvato in una sottoclasse.

2.6 Interoperabilità con Java

[5]

2.6.1 Utilizzo di Java su Kotlin

La piena interoperabilità con Java permette non solo alle applicazioni scritte in Kotlin di girare sulla JVM, ma anche di utilizzare tutte le librerie del Java.

Listing 2.50: Utilizzo di una libreria Java

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }
    // Operator conventions work as well:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

Metodi Getter e Setter

I metodi Getter e Setter che seguono la convenzione di java (un metodo senza argomenti che comincia con **get** e uno con un solo argomento che inizia con **set**) sono rappresentate in Kotlin come delle proprietà.

Listing 2.51: Esempio dell'uso di get e set in kotlin

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY  // call setFirstDayOfWeek()
    }
    if (!calendar.isLenient) {                     // call isLenient()
        calendar.isLenient = true                  // call setLenient()
    }
}
```

Quindi in questo modo nel momento in cui viene chiamata la proprietà *firstDayOfWeek*, in realtà sono i metodi getter e setter che vengono utilizzati.

Void

Il tipo di ritorno *void* in Java corrisponde al tipo *Unit* in kotlin.

Gestione dei null

Tutti i riferimenti in Java possono essere *null*, ne consegue che Kotlin deve trattare i tipi che vengono da classi Java in modo differente; questi tipi vengono definiti come *platform type*. Per questi tipi il controllo dei null non così stringente rispetto ai tipi normali, in modo da mantenere lo stesso approccio che ha Java.

Listing 2.52: Controllo dei null da classi Java

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)

item.substring(1) // allowed, may throw an exception if item == null
```

Si nota come l'ultima istruzione potrebbe lanciare un *NullPointerException*, ma in questo caso i controlli che normalmente il compilatore Kotlin fa sui tipi normali non viene effettuato; quindi l'istruzione viene compilata col rischio che lanci un'eccezione.

Arrays

In kotlin gli array sono invarianti, cioè tutti gli oggetti contenuti all'interno devono essere dello stesso tipo; quindi a differenza del Java, in cui è concesso aggiungere elementi di una sottoclasse ad un array che contiene superclassi, il compilatore Kotlin non lo permette.
Gli array sono tipi primitivi, in modo da evitare i costi di boxing/unboxing; questo implica che quando si deve interagire con il Java, bisogna trattare un array di Int in Kotlin come un int[] in Java.
Kotlin dispone di classi specializzate per ogni tipo di array: *IntArray*, *DoubleArray*, etc.

Metodi della classe principale Object

La classe "radice" in Java è *java.lang.Object*; il suo corrispettivo in Kotlin è la classe *Any*. Per Object di Java sono definiti alcuni metodi standard che possono quindi essere utilizzati per tutti gli oggetti:

- **toString()**: che permette di ritornare una stringa rappresentativa dell'oggetto.
- **hashCode()**: che ritorna un codice hash dell'oggetto
- **equals()**: che confronta l'oggetto con un altro e ritorna un boolean che definisce se sono uguali o meno.

Per implementarle, Kotlin utilizza le *extension function*. Mentre altri metodi come *wait()* e *notify()* non vengono implementati e per richiamarli è necessario fare il cast ad Object:

Listing 2.53: Utilizzo dei metodi wait e notify

```
1 (foo as java.lang.Object).wait()
```

Per quanto riguarda il metodo *getClass()*, viene utilizzata la seguente notazione:

Listing 2.54: Metodo getClass() di Java

```
1 val fooClass = foo::class.java
```

2.6.2 Utilizzo di Kotlin su Java

Kotlin permette di essere richiamato facilmente dal codice Java: il compilatore stesso si può occupare di generare codice Java a partire dal Kotlin.

Getter e Setter

Ogni proprietà di Kotlin viene compilata in Java generando i seguenti elementi:

- Un metodo getter, che comincia con il prefisso *get*.
- Un metodo setter, che comincia con il prefisso *set*.
- Un campo segnato con il modificatore *private* che ha lo stesso nome della proprietà in Kotlin.

Listing 2.55: Come viene generata una singola proprietà di Kotlin in Java

```java
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

Package-Level Functions

Tutte le funzioni di un file *example.kt* dichiarate all'interno di un determinato package vengono compilate in Java all'interno di una classe denominata *.ExampleKt* appartenente allo stesso package del Kotlin. In questo modo è possibile richiamare le funzioni di alto livello Kotlin come delle funzioni statiche della classe appena generata.

Instanze dei campi

Per compilare le proprietà Kotlin in campi Java è necessario aggiungere dei modificatori prima, in modo che il compilatore sappia come tradurli:

- **@JvmField**: In Java viene tradotto come un campo statico.
- **lateinit**: In questo caso viene tradotto come campo non statico.
- **const**: Viene creato un campo statico e costante in java.

2.7 Programmazione web

[5] Il linguaggio Kotlin ha come obiettivo quello di essere utilizzato anche nella programmazione web; è quindi possibile sfruttarlo per la creazione di front-end utilizzando l'interazione con linguaggi come JavaScript e HTML.

2.7.1 Tipo dynamic

Kotlin è un linguaggio fortemente tipizzato, il che può creare problemi in caso sia necessario interoperare con linguaggi non tipizzati come il JavaScript. A questo scopo esiste il tipo *dynamic* che può essere assegnato a ogni variabile e passato ovunque come parametro.

Le sue caratteristiche principali sono le seguenti:

- I makeNullable non hanno effetto su questo tipo
- Può mantenere un valore *null*
- *dynamic* e *dynamic?* sono la stessa cosa
- Non è possibile creare un array di *dynamic*
- non è possibile utilizzarlo come supertipo

N.B. Il tipo *dynamic* non può essere utilizzato se si esegue il codice nella JVM perché non è supportato.

2.7.2 Chiamare JavaScript da Kotlin

Internamente al codice Kotlin, è possibile chiamare del codice JavaScript utilizzando la seguente sintassi:

Listing 2.56: Chiamare JavaScript da Kotlin

```
1 js("Inserire qui il codice js")
```

In questo modo la stringa interna all'istruzione verrà considerata come una funzione JavaScript.

Definizioni esterne

In questo contesto è utile il modificatore *external* che può essere messo prima della dichiarazione di una classe o di un metodo; in questo caso si indica al compilatore che l'implementazione è definita esternamente. Questa istruzione è utile nel caso in cui abbiamo la dichiarazione di un metodo in linguaggio JavaScript che dobbiamo però utilizzare in Kotlin.

Listing 2.57: Modificatore external

```
1 external class Classe() {...}
2 external fun funzione() {...}
```

Nel caso in cui una funzione definita esternamente contiene dei parametri opzionali, è possibile utilizzare il modificatore *definedExternally*

Listing 2.58: Modificatore definedExternally

```
1 external fun funzione(n: Int, str: String = definedExternally)
```

In questo modo la funzione richiede un parametro intero obbligatorio e una stringa opzionale definita esternamente.

Estensioni

Si possono estendere senza problemi le classi JavaScript con classi Kotlin:

Listing 2.59: Estensione di classi JavaScript in Kotlin

```
external open class HTMLElement : Element() { ... }

class CustomElement : HTMLElement() { ... }
```

Ma ci sono alcune limitazioni:

- Quando una funzione deriva da una classe esterna, non si può fare l'override
- Non si può fare l'override di una funzione con argomenti di default

2.7.3 Chiamare Kotlin da JavaScript

Utilizzando il compilatore di Kotlin è possibile generare in automatico classi e funzioni JavaScript, che possono essere normalmente richiamate all'interno del codice front-end che si sta sviluppando. Compilando il codice viene generato un modulo JavaScript che contiene tutte le funzioni scritte precedentemente in Kotlin.

Tipi Kotlin in JavaScript

Quando il codice Kotlin genera le classi JavaScript, i tipi devono essere convertiti opportunamente. Questa attività non è banale, dato che la tipizzazione dei due linguaggi viene trattata in maniera molto diversa.

- I tipi numerici di Kotlin, tranne che per *Long*, vengono mappati in JavaScript con il tipo *Numeric*.
- Il tipo *Char* è mappato come *Numeric* e rappresenta il codice del carattere.
- In JavaScript i numeri a 64 bit non vengono gestiti, quindi il tipo *Long* di Kotlin viene emulato da una classe Kotlin
- Il tipo *Any* di Kotlin viene mappato in JavaScript come *Object*
- Il tipo *String* viene mappato nel tipo *String* di JavaScript
- Le collection di Kotlin non hanno un corrispettivo in JavaScript e non vengono quindi mappati. L'unica collezione ammessa è l'*Array*.

- La classe *Throwable* è mappata nella classe *Error* in JavaScript

Moduli JavaScript

Kotlin consente di compilare il progetto in moduli JavaScript specifici per sistemi di moduli noti:

- **Semplice**: Non viene compilato per nessun sistema di moduli, ma le funzioni generate da Kotlin vengono richiamate semplicemente per nome nello scope globale.
- **Asynchronous Module Definition (AMD)**: sistema usato principalmente per require.js e Dojo Toolkit.
- **CommonJS**: usato molto da node.js
- **Unified Module Definitions (UMD)**: che è compatibile sia con AMD che con CommonJS e viene generato come semplice se a runtime nessuno dei due precedenti sistemi sono utilizzati.

2.8 Kotlin/Native

[5] Kotlin/Native è una tecnologia per compilare il codice Kotlin direttamente in codice nativo che non ha bisogno di una virtual machine per essere eseguito. Questo strumento è pensato soprattutto per quegli ambienti dove non è consigliabile utilizzare una VM, oppure non è proprio possibile.
L'ambito per il quale è più utilizzato è sicuramente lo sviluppo di applicazioni che devono girare su iOS. La stessa applicazione per la KotlinConf del 2017 utilizza questa tecnologia.
Le piattaforme per cui può essere utilizzata attualmente sono:

- Windows (per il momento solo x86_64)
- Linux (x86_64, arm32, MIPS, MIPS little endian)
- MacOS (x86_64)
- iOS (arm32 e arm64)
- Android (arm32 e arm64)
- WebAssembly (wasm32)

2.9 Cooroutines

[5] Nella programmazione concorrente è molto comune il presentarsi di situazioni in cui un thread deve aspettare l'esecuzione di una attività che richiede molto tempo per essere terminata. In questi casi l'esecuzione del thread è bloccata e viene quindi fatto un context switch; cioè viene salvato lo stato del thread e ne viene eseguito un altro. Questa è un'operazione molto costosa da un punto di vista computazionale; quindi gli sviluppatori di Kotlin hanno pensato di introdurre un meccanismo che limitasse questi eventi. Questo strumento sono le coroutine.
Una coroutine è una computazione particolare che può essere sospesa senza interrompere il thread che la sta eseguendo. Lo sforzo principale è stato quello di rendere molto più semplice per il programmatore gestire un ambiente concorrente. Utilizzando le coroutine, infatti, il programmatore ha una vista del programma come se avesse un'esecuzione sequenziale, demandando tutta la complessità alle librerie del linguaggio.
La struttura del programma viene cambiata con tecniche particolari di compilazione in modo tale da distribuire le diverse attività su thread differenti. La presenza delle coroutine porta il compilatore a strutturare il programma come una sorta di macchina a stati, in cui il passaggio da uno stato all'altro corrisponde alla sospensione della coroutine. Per come sono pensate, le coroutine non possono essere interrotte in un punto casuale dell'esecuzione, come i thread; è il programmatore che decide il momento in cui sospendere una coroutine. In questo modo si evitano tutti i problemi di accesso condiviso alle risorse e consistenza dello stato tipiche della programmazione tramite thread.
Al momento le coroutine sono ancora in fase di sviluppo e le librerie ad esse dedicate possono ancora subire delle variazioni. Sono state introdotte a partire dalla versione 1.1 di kotlin e si trovano nella libreria *kotlin.coroutines.experimental*. Per questo motivo viene consigliato di aggiungere il suffisso *ecperimental* ai package che le utilizzano, il quale può essere tolto dopo che le coroutine verranno rilasciate in versione definitiva.

2.9.1 Suspending Functions

Lo strumento fondamentale su cui sono basate le coroutine sono le suspending function. Queste sono funzioni particolari che possono

essere sospese per proseguire con l'esecuzione di un altro ramo del programma. Nel seguente modo si dichiara una suspending function:

Listing 2.60: Funzioni Suspending

```
1 suspend fun suspendigFunction() {...}
```

Questo tipo di funzioni sono le uniche che possono sospendere una coroutine. Un vincolo fondamentale è che una suspending function può essere avviata solo all'interno di un'altra. Questo porta all'utilizzo di high-order function per avviare la prima suspending function:

Listing 2.61: Avviare la prim suspendig function

```
1 fun <T> async(block: suspend () -> T)
```

In questo modo può essere passata una suspending function ad *async* che si occuperà di avviarla.

2.9.2 In futuro

Attualmente sono in sviluppo sempre più librerie per la gestione del flusso dei programmi che utilizzano le coroutine. L'obiettivo è la creazione di funzioni di alto livello che permettano in modo semplice di interrompere una coroutine e avviarne un'altra. Queste funzioni si trovano nella libreria *kotlinx.coroutines.experimental* e subiscono frequenti modifiche mirati a migliorarne il comportamento. Il suffisso *experimental* verrà tolto quando si passerà a una versione definitiva di questo potente strumento.

Capitolo 3

Sviluppo del progetto

3.1 Scopo del progetto

Lo sviluppo del progetto ha lo scopo di sviluppare un front-end web e un back-end server in Kotlin, in modo da studiare il funzionamento del linguaggio in un contesto verosimile. Solo in questo modo sarà possibile verificare l'effettiva efficacia del linguaggio e i relativi punti deboli che sarebbero impossibili da rilevare in un contesto non applicativo.
Si è cercato, quindi, di utilizzare molti strumenti che possono rivelarsi utili nello sviluppo di una vera applicazione distribuita e ne si è verificata l'affidabilità in un contesto ancora sperimentale.

3.2 Requisiti

Il progetto consiste nella realizzazione di un server e di una console di amministrazione che fanno da supporto a un'applicazione sulla gestione e lo svolgimento di percorsi turistici attraverso dei punti di interesse. La parte dell'applicazione è sviluppata nel contesto di un'altra tesi.
Sarà quindi sviluppato un REST web service che deve avere l'obiettivo di rispondere alle richieste inviate dall'applicazione e una console web di amministrazione che invia le richieste allo stesso server.

Il progetto consiste di due parti:

- Un front-end di amministrazione, attraverso il quale è possibile gestire i luoghi di interesse.
- Un server di back-end che serve le richieste provenienti da applicazione e console di amministrazione.

3.2.1 Front-End

La parte di Front-End deve essere una console di amministrazione per poter gestire i luoghi di interesse. Gli utenti per cui è pensata possono essere eventuali stakeholder dell'applicazione, che aggiungono punti commerciali al fine di attrarre gli utenti durante i tour turistici. Altri utenti possono essere gli stessi sviluppatori che possono aggiungere dei luoghi di interesse più generali, come musei o monumenti.
Per ogni luogo da aggiungere, deve essere specificato un determinato poligono che deve definire la parte di mappa all'interno del quale ci si deve trovare per considerare quel luogo come vicino.
La parte centrale della console è data dall'interazione con le API di google dedicate alle mappe. Tramite quest'ultime deve essere possibile visualizzare la mappa e scegliere i luoghi da aggiungere.

3.2.2 Back-End Server

La parte server è quella che si occupa di servire tutte le richieste delle altre due parti. Si deve quindi ottenere un server che mantiene un database di informazioni relative a utenti e luoghi di interesse e che si occupi di fornire queste informazioni all'applicazione e alla console di amministrazione.
Si deve implementare un sistema di accounting e di autorizzazione per gli utenti che utilizzano le funzioni del back-end. Il server deve ricevere la posizione dall'applicazione e deve fornire solo i luoghi il cui poligono contiene le coordinate appena ricevute. Deve, inoltre, fornire una lista di luoghi che possono essere visitati dato un percorso scelto dall'utente.
Deve essere inoltre fornita la possibilità all'applicazione web di amministrazione di aggiungere su database nuovi luoghi.

3.3 Progettazione

Nella fase di progettazione verranno esposti gli strumenti che saranno utilizzati nello sviluppo del progetto, i mockup dell'applicazione web e la struttura che le due parti del progetto dovranno avere.

3.3.1 HTTP

Dato che il server sarà un REST web service, il protocollo di comunicazione da usare sarà l'HyperText Transfer Protocol (HTTP); è un protocollo a livello di applicazione per sistemi distribuiti, collaborativi e informazioni ipermediali ed è stato usato per il World-Wide Web fin dal 1990.
E' un protocollo basato sul paradigma richiesta/risposta e tipicamente usato nei sistemi di tipi client-server. Ogni richiesta viene fatta su una determinata URI e può avere diversi metodi, ognuno dei quali viene fatto per una richiesta specifica. I principali metodi utilizzati sono i seguenti:

- **GET**: Usata per richiedere dei dati dal server. Non contiene body
- **POST**: Usata per creare un nuovo dato su server
- **PUT**: Usata per aggiornare un dato sul server
- **DELETE**: Usata per eliminare un dato dal server

La risposta del server contiene un codice numerico che fornisce informazioni sull'esito della richiesta e può contenere un body. [11]

3.3.2 JSON

Uno dei requisiti è lo scambio di dati strutturati tra le diverse parti dell'applicativo; in questi casi devono essere inseriti nel body della richiesta/risposta secondo un formato che entrambe le parti possano utilizzare. Nell'ambito di questo progetto si è scelto di usare il JSON. Leggero, veloce e di facile comprensione anche per le persone, è supportato da moltissime librerie per il suo utilizzo. In questo caso è stato utile anche verificare la compatibilità del Kotlin con le librerie Java dedicate all'utilizzo del JSON. [12]

Struttura

La struttura di un messaggio JSON può essere costituita da una mappa di coppie chiave/valore che rappresentano i campi dell'oggetto; il valore può anche essere un array di altri elementi JSON.
Ogni elemento è contenuto all'interno di parentesi graffe; mentre gli array di elementi sono contenuti all'interno di parentesi quadre.

3.3.3 Gradle

Gradle è un software open-source che serve a gestire al meglio progetti di medie e grandi dimensioni. Serve ad automatizzare alcune parti dello sviluppo e alla gestione le dipendenze. E' utile, inoltre, per importare in automatico le librerie necessarie allo sviluppo del progetto, scaricandole da repository on-line.[1]

3.3.4 Front-End Web

Il normale codice Kotlin viene eseguito sulla JVM; ne consegue che non è possibile eseguirlo su browser. Prima di cominciare a considerare la realizzazione del sito, quindi, bisogna prima studiare gli strumenti che il linguaggio mette a disposizione per ottenere del codice eseguibile in ambiente web.

Compilazione

Utilizzando il Gradle è possibile configurare le opzioni di compilazione da impostare per il progetto. Nel caso del front-end è utile la funzione che gli sviluppatori hanno messo a disposizione per poter generare del codice JavaScript compilando un progetto Kotlin.
Per fare ciò bisogna usare il Gradle attraverso il quale è possibile definire le opzioni di compilazione per il progetto. In questo caso bisogna impostare il plug-in del compilatore *kotlin2js*. Con questa impostazione, ogni volta che viene compilato il progetto, viene generato un file JavaScript, contenente la traduzione del codice Kotlin.

Libreria JavaScript per Kotlin

Gli sviluppatori hanno fornito la libreria standard *kotlin.js* che consente di utilizzare le stesse funzioni del JavaScript per la gestione della pagina web, come ad esempio l'oggetto *document*.
Per utilizzare la libreria nel progetto, bisogna aggiungerne la dipendenza sul Gradle.

Listing 3.1: Aggiungere la dipendenza di kotlin.js

```
1 dependencies {
2     compile "org. jetbrains . kotlin : kotlin-stdlib-js: $kotlin_version "
3 }
```

Libreria HTML

Nel caso in cui l'obiettivo è quello di costruire una pagina web dinamica, che si adegui a dati variabili generati sul momento, potrebbe essere necessario richiamare del codice HTML dall'interno del codice JavaScript.
A tale scopo è in sviluppo da parte del team di Kotlin una libreria sperimentale che serve a ricreare l'alberatura della pagina HTML. La libreria è *kotlinx.html* e viene importata sempre tramite il Gradle. [7]

API di Google Maps

La parte principale del front-end è sicuramente l'interazione con Google Maps. L'utente deve interagire con una mappa e con dei luoghi di interesse e quindi il sito deve permettere la visualizzazione di una mappa con relativi luoghi presenti e l'interazione con essa.
Per fare ciò verranno utilizzate le API di Google Maps per JavaScript, che permettono di scaricare, durante il caricamento della pagina web, le funzioni per la gestione della mappa.
Per poter utilizzare le API è necessario possedere una API KEY fornita da google; al caricamento della pagina bisogna fare uno script che invii una richiesta HTTPS ai server di Google. In questo modo si possono scaricare le funzioni per l'uso delle mappe.

Listing 3.2: Script per la richiesta a Google Maps

```
1 <script src="https://maps.googleapis.com/maps/api/js?key=KEY&libraries=geometry"></script>
```

Il parametro *libraries=geometry* serve a scaricare le funzioni relative alla manipolazione dei poligoni nella mappa, che sarà sicuramente utile nello sviluppo del progetto.
Le funzioni scaricate, però, sono per il JavaScript che non prevede la gestione dei tipi; essendo Kotlin un linguaggio fortemente tipizzato, occorre avvolgere le funzioni di Maps in dei wrapper Kotlin che permettano la gestione dei tipi. Attualmente, però, non ci sono le librerie di Kotlin dedicate a questo compito e quindi i wrapper dovrebbero essere fatti customizzati dal programmatore. [3]

Libreria TypeScript e ts2kt

Il TypeScript è un linguaggio open source fortemente tipizzato. Nasce con l'obiettivo di estendere la sintassi del JavaScript e ogni programma viene prima ricompilato in quest'ultimo per poter essere interpretato

dai browser. Fornisce una vasta quantità di librerie che svolgono il ruolo di wrapper per il JavaScript.
Ovviamente esiste anche una libreria TypeScript per le API di Google Maps chiamata *@types/googlemaps* e che può essere scaricata grazie all'ecosistema Node.js.
Per poter utilizzare questa libreria per i nostri scopi dobbiamo prima tradurla in Kotlin. Consapevoli di questa necessità gli sviluppatori hanno creato un tool, anch'esso scaricabile tramite Node.js, che si chiama ts2kt. Tramite questo strumento è possibile convertire qualunque libreria TypeScript in una libreria Kotlin.
In questo modo possiamo ottenere tutte le funzioni wrapper che ci servono per gestire le funzioni Maps scaricate.

Vista

La pagina web si apre con la pagina di log-in dell'utente in cui inserire le credenziali di accesso. Non c'è la possibilità di registrarsi, perché essendo una console di amministrazione, ogni nuovo utente deve essere valutato dai possessori dell'applicazione.
Dopo il login viene ricevuta la lista di tutti i luoghi di interesse relativi al determinato utente e vengono visualizzati come marker nella mappa. Ogni luogo ha il suo relativo poligono, visibile anch'esso nella mappa.
Il pulsante **New Place** permette l'inserimento di un nuovo punto di interesse. Nel momento in cui viene premuto permette di selezionare direttamente sulla mappa il punto in cui si vuole inserire il nuovo luogo. Una volta che viene selezionato il luogo, verrà visualizzato:

- Un marker nel punto selezionato
- Un poligono editabile a piacimento dall'utente
- Le coordinate del punto selezionato
- Il costo del punto in base all'area del poligono in quel momento, che varia a seconda della dimensione di quest'ultimo
- I campi per inserire le informazioni sul luogo

I campi saranno il nome del luogo e la descrizione da inserire. Ci sarà, inoltre, la possibilità di inviare una foto al server relativa al luogo che si sta aggiungendo.
Dopo aver inserito tutte le informazioni, si preme il pulsante **Add**

place e il luogo viene inviato al server che si occupa di salvarlo. A questo punto viene visualizzata nuovamente la schermata con tutti i luoghi relativi all'utente loggato compreso quello appena inserito.

3.3.5 Back-End server

Il compito del server di back-end deve essere quello di salvare su un database le informazioni riguardanti gli utenti e i luoghi di interesse e servire richieste HTTP operando su coordinate geospaziali. Solo tenendo ben presente questi requisiti, si può prendere una decisione su quali strumenti utilizzare.

RESTfull web-service

L'architettura REST definisce uno stile architetturale per la progettazione di un sistema che risponde a richieste HTTP. Non è mai stato standardizzato da nessun ente ufficiale, ma rappresenta più che altro una serie di principi con cui progettare un'architettura distribuita.
Realizzare web service che rispettino i principi REST vuole dire rispettare i seguenti principi:

- Utilizzo di metodi HTTP
- Identificazione delle risorse
- Risorse autodescrittive
- Collegamento tra le varie risorse
- Comunicazione stateless

Non è scopo di questa tesi entrare nel dettaglio dell'architettura REST; occorre invece specificare lo strumento utilizzato per sfruttare i principi appena descritti nella progettazione del server.[4]

Spring Boot

Kotlin supporta l'utilizzo di Spring Boot per la realizzazione di web-service. Per utilizzare gli strumenti messi a disposizione bisogna prima importare tutto il necessario tramite Gradle.

Listing 3.3: Aggiungere la dipendenza di kotlin.js

```
1 dependencies {
2     buildscript  {
3   ext {
```

```
4       kotlinVersion = '1.2.41'
5       springBootVersion = '2.0.2.RELEASE'
6     }
7     repositories {
8       mavenCentral()
9     }
10    dependencies {
11      classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
12      classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:${kotlinVersion}")
13      classpath("org.jetbrains.kotlin:kotlin-allopen:${kotlinVersion}")
14    }
15  }
16
17  apply plugin: 'kotlin'
18  apply plugin: 'kotlin-spring'
19  apply plugin: 'org.springframework.boot'
20  apply plugin: 'io.spring.dependency-management'
21  }
```

Vengono usati tre plugin utili all'importazione degli strumenti di Spring. *org.springframework.boot* è la libreria che contiene le funzioni utili all'utilizzo di Spring Boot. *io.spring.dependency-management* è il gestore delle dipendenze del framework. *kotlin-spring* è utile a inserire il modificatore *open* nelle classi di kotlin segnate con le annotazioni di Spring, dato che in Kotlin di default sono dichiarate *final*.
Va aggiunta pure la dipendenza che permette di importare le funzioni di Spring Boot utili nello sviluppo.

Listing 3.4: Aggiungere la dipendenza di kotlin.js

```
1  compile('org.springframework.boot:spring-boot-starter-web')
```

Con il Gradle strutturato in questo modo, il progetto viene automaticamente avviato e si mette in ascolto per delle eventuali richieste in arrivo. [13]

MongoDB

In questo contesto dobbiamo avere un database su cui salvare dei luoghi di interesse con relative coordinate; una delle attività principali è quella di verificare un punto ricevuto da un client si trovi all'interno dei poligoni associati ai luoghi salvati su database.
A questo scopo è stato scelto come database per il progetto MongoDB. Database non relazionale, basato sull'utilizzo di documenti e con un efficace sistema di indicizzazione per la gestione di grandi quantità di dati. Un MongoDB può contenere diversi database e ognuno di questi può contenere diverse *collection* al cui interno è possibile aggiungere i *document*. Dato che si basa sui documenti, uno dei suoi principali punti di forza è la possibilità di inserire nella stessa tabella dati con strutture diverse, in modo da avere un'elevata flessibilità.

Il motivo principale per cui è stato scelto, però, è per la possibilità che mette a disposizione di utilizzare metodi e oggetti geospaziali per l'utilizzo e la manipolazione delle coordinate geografiche. In pratica è possibile salvare degli oggetti che contengono delle coordinate e agire su di esse tramite delle funzioni messe a disposizione direttamente dal MongoDB.
I documenti che vengono utilizzati per salvare elementi sul database sono di tipo BSON; in pratica sono degli oggetti che hanno la stessa struttura del JSON, ma che contengono informazioni in più soprattutto per quanto riguarda i tipi di dati.[8]

API da implementare

Il server si deve occupare di servire le richieste che arrivano dall'applicazione Android e dalla console di amministrazione. Quindi devono essere implementate delle API che si occupino di servire queste richieste.
Le richieste per la console di amministrazione sono le seguenti:

- **POST: Registrazione di un nuovo account**
  Parametri: *mail*, *name*, *pass*
  Si occupa di salvare sul database un nuovo utente di amministrazione per l'accesso alla console Web, se non è già presente un altro account con la stessa mail; la password è salvata come hash.

- **GET: Login per la console di amministrazione**
  Parametri: *mail*, *pass*
  Verifica sul database se esiste un utente con la mail indicata e che l'hash della password sia lo stesso di quello salvato.

- **POST: Aggiunta di un nuovo punto di interesse**
  Parametri: *mail*, *pass*
  Body: *Point*
  Verifica che l'utente e la password siano corretti e in caso positivo aggiungono l'oggetto di tipo Point inviato come body nel database.

Per quanto riguarda l'applicazione Android, invece, le API sono le seguenti:

- **POST: Registrazione di un nuovo account**
  Parametri: *mail*, *name*, *pass*

Salva sul database il nuovo account dell'applicazione android, se non è già presente un altro account con la stessa mail; la password è salvata come hash.

- **GET: Login di un'account dell'applicazione**
  Parametri: *mail*, *pass*
  Verifica sul database se esiste un utente con la mail indicata e che l'hash della password sia lo stesso di quello salvato.

- **GET: Punti nelle vicinanze**
  Parametri: *mail*, *pass*, *latitude*, *longitude*
  Verifica sul database i poligoni di quali punti di interesse contengono le coordinate ricevute e ritorna come risposta la lista di tali punti.

- **GET: On the Road**
  Parametri: *mail*, *pass*, *startingLat*, *startingLon*, *finalLat*, *finalLon*
  Verifica i poligoni di quali punti sono intersecati dal percorso tra coordinate iniziali e finali e torna una lista di tali punti.

3.4 Implementazione

In questa sezione verrà esposta l'implementazione delle due parti del progetto; per il lato Front-End saranno anche inserite le immagini relative al sito definitivo sviluppato.

3.4.1 Front-End Web

L'obiettivo di questa parte del progetto è quello di verificare la compatibilità di Kotlin con Javascript e della reale possibilità di generare del codice eseguibile da browser. Era molto importante, inoltre, verificare l'effettiva possibilità di integrare le librerie di Google Maps in Kotlin e testarne il funzionamento.
Si fa un largo uso della libreria *kotlinx.html* che permette di manipolare le pagine web e gli elementi in essa presenti. L'elemento principale dell'interfaccia è *root*, che può essere preso e manipolato tramite la seguente istruzione:

Listing 3.5: Prendere un elemento della pagina web

```
1 val root = document.getElementById("root")
```

In questo modo la variabile *root* è un oggetto di tipo *Element* che in Kotlin rappresenta un elemento JavaScript.
Per creare un nuovo elemento HTML, invece, viene usata la seguente funzione:

Listing 3.6: Creare un nuovo elemento della pagina web

```
val home = document.create.div{}
```

All'interno delle parentesi graffe possono essere usati gli elementi dell'HTML per la costruzione di pagine Web, come ad esempio *h1*, *input* o *br*. Un esempio sull'utilizzo di tali elementi:

Listing 3.7: Esempio di utilizzo di elementi HTML

```
        h1{
            +"HOME"
        }

        br
        input {
            id = "mail"
            placeholder = "mail"
            type = InputType.text
        }
```

Login

L'interfaccia di login è ovviamente molto semplice come si vede dalla figura 3.3.

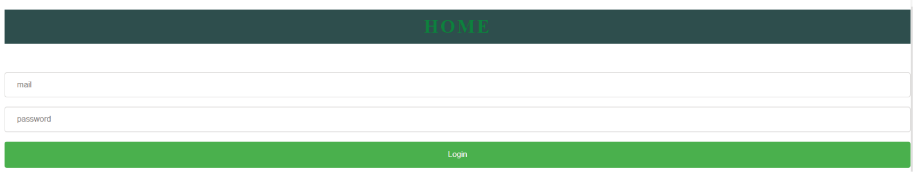

Figura 3.1: Login del Front-End Web

Inserendo username e password e cliccando su **Login** si accede alla propria pagina di amministrazione.
Questa interfaccia è sviluppata creando un nuovo elemento HTML e inserendolo come figlio all'elemento *root*.

Listing 3.8: Creazione del div che contiene la schermata di login

```
val home = document.create.div{
        id = "home"
        //home code
    }

    root !!. appendChild(home)
```

All'interno di questo elemento è possibile richiamarsi la mappa utilizzando le librerie di google:

Listing 3.9: Richiamarsi la mappa

```
val map = google.maps.Map(document.getElementById("map"))
map.setCenter(latlng = LatLng(0, 0))
map.setZoom(2)
```

Successivamente sono stati aggiunti tutti gli elementi che devono essere visualizzati nell'interfaccia del login:

Listing 3.10: Elementi del login: il titolo

```
h1{
    +"HOME"
}
```

Listing 3.11: Elementi del login: la textbox per la mail

```
input {
    id = "mail"
    placeholder = "mail"
    type = InputType.text
}
```

Listing 3.12: Elementi del login: la textbox per la password

```
input {
    id = "pass"
    placeholder = "password"
    type = InputType.password
}
```

Listing 3.13: Elementi del login: il pulsante di login

```
input{
    value = "Login"
    type = InputType.submit
    onClickFunction = { val mail = document.getElementById("mail") as HTMLInputElement
        val pass = document.getElementById("pass") as HTMLInputElement
        login(mail.value, pass.value, map)

        root !!. firstElementChild !!. remove()
    }
}
```

In quest'ultimo elemento viene inserito nella funzione di click del pulsante il codice per inviare la richiesta HTTP di login al server.
Combinando tutti questi elementi all'interno del div creato precedentemente, si ottiene l'interfaccia di login da aggiungere all'elemento *root*.

Visualizzazione dei luoghi

Se si clicca sul pulsante di login viene richiamata una funzione che si occupa di inviare i dati al server, attendere come risposta la lista dei luoghi di interesse relativi a username e password inseriti e infine visualizzare tutto nella mappa:

Listing 3.14: Funzione di login

```
fun login(mail: String, pass : String, map: Map)
{
    val req = XMLHttpRequest()
    val reqString = makeReqString(mail, pass, "login")

    req.open("GET", reqString, false)

    req.send()

    var admin = document.getElementById("admin")
    var console : HTMLElement

    while(req.readyState != XMLHttpRequest.DONE){}

    if ( req.status == 200.toShort())
    {
        val array = JSON.parse<Array<Point>>(req.responseText)

        if (array.size>0)
        {
            map.setCenter(LatLng(array[0].position.lat, array[0].position.lon))
            map.setZoom(10)
        }

        for (elem : Point in array)
        {
            addMarker(LatLng(elem.position.lat, elem.position.lon), elem.name, elem.description, map)

            addPolygon(elem.polygon, map, false)
        }


        console = document.create.div {
            input {
                value = "New place"
                type = InputType.submit
                onClickFunction = {
                    admin!!.firstElementChild!!.remove()
                    admin!!.appendChild(document.create.div { h2 { +"Click on map to choose the point"} })
                    event.addListener(map, "click", {event: MouseEvent -> addNewPlace(event, map, mail,
                        pass) })
                }
            }
        }
    }
    else
    {
        console = document.create.div {
            h2 {
                +"Problemi di autenticazione"
            }
        }
    }

    admin!!.appendChild(console)
}
```

Un volta che questa funzione viene eseguita si possono ricevere dal server due tipi di risposte:

- Se il login è andato a buon fine si riceve una risposta con status 200 (OK) e come body un JSON contenente tutti i luoghi appartenenti all'utente.
- Se c'è stato qualche problema si possono ricevere risposte con status 401 (Unauthorized) o 500 (Internal server error).

Nel caso la richiesta vada a buon fine, il risultato sarà simile a quello mostrato in figura 3.2.

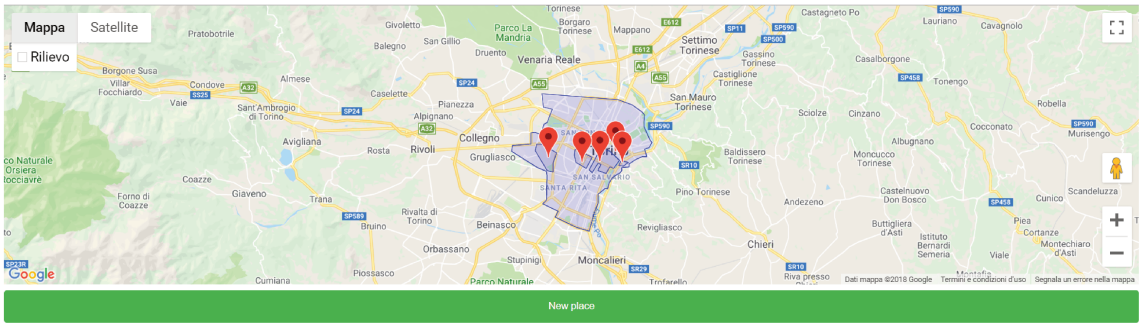

Figura 3.2: Visualizzazione dei luoghi di interesse

Per ogni punto di interesse tornato dal server, devono essere aggiunti alla mappa un *marker* e un *poligono*. Le funzioni che si occupano di fare questa operazione sono spiegate nelle sezione seguente.
In questa schermata è possibile spostare la mappa e cliccare su ogni punto di interesse per leggerne le informazioni e per la gestione.

Aggiunta di un marker alla mappa

La funzione che aggiunge i *marker* alla mappa è la seguente:

Listing 3.15: Funzione per aggiungere un marker alla mappa

```
fun addMarker(latLng: LatLng, name: String, description : String, map: Map) : Marker
{
    var contentString =
            "<div id=\"content\">"+
                    "<div id=\"siteNotice>"+
                    "</div>"+
                    "<h2 id=\"firstHeading\" class=\"firstHeading\">$name</h2>"+
                    "<div id=\"bodyContent\">$description</div>"+
                    "</div>"

    var infoWindow = google.maps.InfoWindow()
    infoWindow.setContent(contentString)

    var marker = google.maps.Marker()
    marker. setPosition (latLng)
    marker.setMap(map)

    marker.addListener("click", { infoWindow.open(map, marker) })

    return marker
}
```

Ogni punto di interesse deve avere un poligono attorno, perché il luogo deve essere tornato agli utenti dell'applicazione che si trovano al suo interno. Esiste quindi anche una funzione che aggiunge alla mappa i poligoni relativi ai punti ed è la seguente:

Listing 3.16: Funzione per aggiungere un poligono alla mappa

```
fun addPolygon(polygon: Array<Position>, map: Map, editable: Boolean) : Polygon
{
    var path = mutableListOf<LatLng>()

    for (elem : Position in polygon)
        path.add(LatLng(elem.lat, elem.lon))

    var polygon = google.maps.Polygon()
    polygon.setDraggable( false )
    polygon. setEditable ( editable )
    polygon.set("clickable", false)
    polygon.setPath(path.toTypedArray())
    polygon.set("strokeWeight", 1)
    polygon.set("strokeColor", "#0000FF")
    polygon.set("fillColor", "#0000FF")
    polygon.set("fillOpacity", 0.1)
    polygon.setMap(map)

    return polygon
}
```

Da notare come sia possibile rendere il poligono editabile con la funzione *setEditable*.

Aggiunta di un nuovo punto di interesse

Cliccando sul pulsante *New Place* viene aggiunto un *listener* per i *click* sulla mappa. Una volta scelto il punto in cui inserire il nuovo punto di interesse, ne vengono catturate le coordinate, viene aggiunto il *marker* sulla mappa e vengono visualizzate le caselle di testo per inserire le informazioni relative al punto:

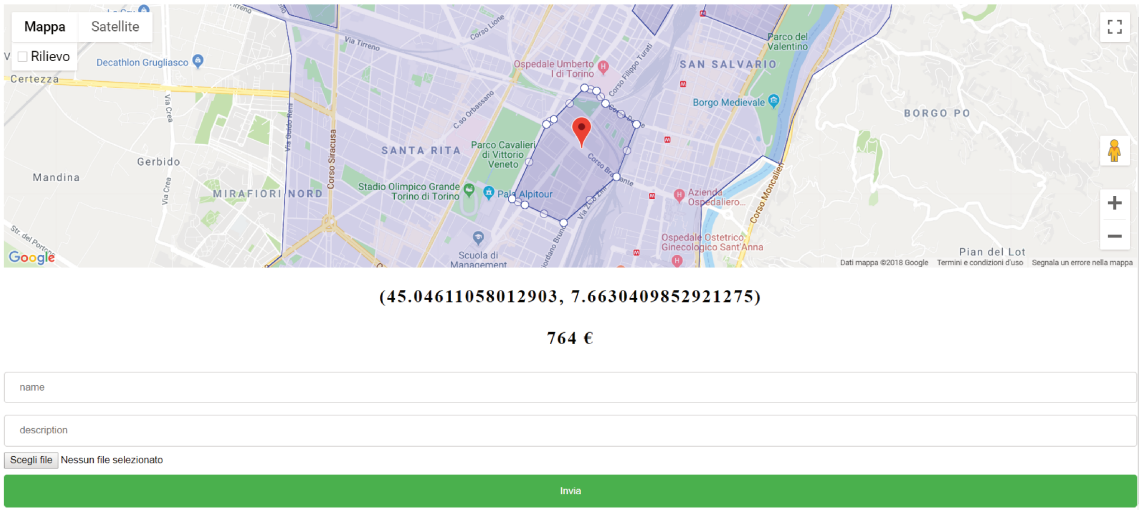

Figura 3.3: Aggiunta di un nuovo punto di interesse

Vengono visualizzate le coordinate del punto che si sta per aggiungere e una ipotetica cifra di pagamento per l'aggiunta del punto che è funzione dell'area del poligono. Si nota come i contorni del poligono sono modificabili direttamente sulla mappa; sarà l'applicazione stessa a prendere in automatico la lista di coordinate che lo descrivono.

Vi è pure la possibilità di aggiungere un'immagine al punto di interesse che si sta per aggiungere, scegliendo un file locale per l'upload.
Una volta cliccato sul pulsante *Add Place* viene fatta una richiesta HTTP di tipo POST che contiene un JSON relativo al nuovo punto di interesse da aggiungere.

Listing 3.17: POST per l'aggiunta di un nuovo punto

```
val reqString = makeReqString(mail, pass, "addPlace")

req.open("POST", reqString, false)
req.setRequestHeader("Content-Type", "application/json")
req.send(JSON.stringify(Point(Position(pos.lat().toDouble(), pos.lng().toDouble()),
        name.value,
        description.value,
        listPoint,
        req2.responseText)))
```

Se a questa richiesta il server risponde con status 201 (Created), il punto è stato aggiunto e l'applicazione torna alla schermata visualizzata nella figura 3.2

3.4.2 Back-End Server

Il back-end server è stato sviluppato come un RESTfull web service; sono state quindi definite tutte le API che dovevano essere implementate per rispondere alle esigenze sia della console web di amministrazione che dell'applicazione Android.
Prima di passare alla descrizione delle API, però, occorre mostrare la parte di modeling e di interazione con il database.

Modelling

In questa parte del progetto è stato necessario annotare in modo opportuno alcune classi per permettere la piena interoperabilità con il MongoDB; cioè è possibile utilizzare direttamente le classi definite nella parte di modeling per salvare dati sul database.

Listing 3.18: Classe User

```
class User {

    @SerializedName(MAIL)
    var mail: String? = null

    @SerializedName(NAME)
    var name: String? = null

    @SerializedName(PASS)
    var pass: String? = null

    constructor() {}

    constructor(mail: String, name: String, pass: String) : super() {
```

```
        this.mail = mail
        this.name = name
        this.pass = pass
    }

}
```

Questa classe rappresenta un utente e l'annotazione *@SerializedName* definisce il nome che il campo che viene generato dalla proprietà deve avere quando l'oggetto viene salvato su MongoDB.

Listing 3.19: Classe Place

```
class Place
{
    @SerializedName(COORDINATES)
    var coordinates: Point? = null

    @SerializedName(PLACE_NAME)
    var name: String? = null

    @SerializedName(DESCRIPTION)
    var description: String? = null

    @SerializedName(AREA)
    var area: Polygon? = null

    @SerializedName(MANAGER)
    var manager: String? = null

    @SerializedName(IMAGE)
    var image: String? = null

    constructor() {}

    constructor(coordinates: Point, name: String, description: String, area: Polygon, manager: String, image:
        String)
    {
        this.coordinates = coordinates
        this.name = name
        this.description = description
        this.area = area
        this.manager = manager
        this.image = image
    }
}
```

Questa classe rappresenta i singoli punti di interesse.
Questo però non basta per poter salvare i dati direttamente su database, perché ogni classe ha bisogno di un **codec** (codificatore/decodificatore) per poter serializzare gli oggetti in documenti BSON da salvare su MongoDB. Fortunatamente esistono dei metodi particolari messi a disposizione direttamente dalle librerie di Mongo per Java che permettono la creazione di codec per ogni classe.

Listing 3.20: Definizione dei codec

```
private  val mon = MongoClient(HOST,DEFAULT_PORT)
private val db = mon.getDatabase(DATABASE)
private var col = db.getCollection(USER, User::class.java)
private var manager = db.getCollection(MANAGERS, User::class.java)
private var places = db.getCollection(PLACES, Place::class.java)

private val codec = fromRegistries(MongoClient.getDefaultCodecRegistry(),
```

```
                fromProviders(PojoCodecProvider.builder().automatic(true).build()))

fun mongoCodec()
{
    col = col.withCodecRegistry(codec)
    places = places.withCodecRegistry(codec)
    manager = manager.withCodecRegistry(codec)
}
```

Con le funzioni statiche *fromRegistries* e *fromProviders* contenute nel package *org.bson.codecs.configuration.CodecRegistries* è possibile creare un codec automatico che permette la codifica e la decodifica delle classi a cui viene associato, come si vede nella funzione *mongoCodec*. Se queste istruzioni vengono fatte nel momento in cui parte il server, tutte le occasioni in cui serve salvare questi oggetti sul database è possibile farlo senza ulteriori decodifiche:

Listing 3.21: Inserire un user in una collection

```
col.insertOne(user)
```

All'interno del modeling è presente anche la dichiarazione di nuove eccezioni che vengono scatenate in caso di problemi nel servire una determinata richiesta:

Listing 3.22: Eccezioni nel RESTfull web service

```
@ResponseStatus(HttpStatus.CONFLICT)
class UserAlreadyRegisteredException(override var message:String) : Exception(message)

@ResponseStatus(HttpStatus.UNAUTHORIZED)
class UserNotFoundException(override var message:String) : Exception(message)

@ResponseStatus(HttpStatus.NOT_FOUND)
class NoImageFoundException(override var message:String) : Exception(message)
```

Come si nota l'annotazione *@ResponseStatus* definisce lo status code da tornare al richiedente in caso venga lanciata questa eccezione durante l'esecuzione di una richiesta.

Interazione con MongoDB

Di seguito verranno esposti i metodi che interagiscono con il database MongoDB.
Con il seguente metodo viene inserito un nuovo utente nel database:

Listing 3.23: Registrazione di un nuovo utente

```
fun newUser(user: User): Boolean
{
    try {
        if (col.find(eq(MAIL, user.mail)).count() > 0) {
            return false
        }
        else
        {
```

```
            val digest = MessageDigest.getInstance(HASH_ALGORITHM)
            user.pass = digest.digest(user.pass !!. toByteArray()).contentToString()
            col.insertOne(user)
            return true
        }
    } catch (iae: IllegalArgumentException){
        return false
    }
}
```

Priva viene verificato se la chiave (in questo caso la mail) è già presente sul database, in caso contrario viene fatto l'hash della password e viene salvato tutto l'elemento nella collection. La precedente funzione si riferisce all'inserimento di un normale utente dell'applicazione; per quanto riguarda l'inserimento di un nuovo utente di amministrazione, la funzione è la stessa, ma viene aggiunto nella collection *manager*.
La seguente, invece, è la funzione di login che verifica che user e password inviati siano presenti sul database:

Listing 3.24: Login

```
fun login(user: User) : Boolean
{
    try {
        val log = col.find(eq(MAIL, user.mail)). first ()
        val digest = MessageDigest.getInstance(HASH_ALGORITHM)
        user.pass = digest.digest(user.pass !!. toByteArray()).contentToString()
        if ( log != null )
        {
            return log.pass == user.pass
        }
        else
            return false
    }
    catch (iae: IllegalArgumentException)
    {
        return false
    }
}
```

La seguente funzione si occupa invece di ritornare, data una certa posizione, tutti i luoghi i cui poligoni contengono quelle specifiche coordinate:

Listing 3.25: Luoghi vicini a una data posizione

```
fun matchPosition(point: Point): List<Place>
{
    var list = mutableListOf<Place>()

    try {

        list .addAll(places.find( geoIntersects (AREA, point)))

        return list
    }
    catch (e: NullPointerException)
    {
        return listOf ()
    }
}
```

La funzione *geoIntersect* è una funzione appartenente alle classi Java messe a disposizione da MongoDB e permette di utilizzare una delle funzioni geospaziali del database; in pratica torna tutti gli elementi della collection in cui il parametro *AREA* contiene le coordinate di *point*.
Nel seguente caso invece abbiamo una funzione che ritorna tutti i punti di interesse in cui il campo *MANAGER* è lo stesso dell'user passato:

Listing 3.26: Luoghi appartenenti a un determinato account di amministrazione

```
fun matchManager(manager:String): List<Place>
{
    var list = mutableListOf<Place>()

    try {
        list.addAll(places.find(eq(MANAGER, manager)))

        return list
    }
    catch (e: NullPointerException)
    {
        return listOf()
    }
}
```

In questo caso la funzione *eq* è una funzione statica appartenente al package *com.mongodb.client.model.Filters* che sfrutta le funzioni di filtro del database. Avremo bisogno, infine, della funzione che si occupa di aggiungere dei punti di interesse al database:

Listing 3.27: Aggiunta di un punto di interesse al database

```
fun newPlace (place: Place) : Boolean
{
    try {
        if (places.find(and(eq(COORDINATES, place.coordinates), eq(PLACE_NAME, place.name))).count() > 0)
            {
            return false
        }
        else
        {
            places.insertOne(place)
            return true
        }
    } catch (iae: Exception){
        return false
    }
}
```

L'applicazione Android potrebbe inoltre inviare un punto iniziale e uno finale e aspettarsi come risposta una lista di punti di interesse tra questi due punti. Per questo è stata sviluppata anche la seguente funzione:

Listing 3.28: Funzione per l'on the road

```
fun matchLine(line: LineString) : List<Place>
{
    var list = mutableListOf<Place>()

```

```
    try {

        list.addAll(places.find(geoIntersects("area", line)))

        return list
    }
    catch (e: NullPointerException)
    {
        return listOf()
    }
}
```

Viene ricevuta una linea, che è rappresentata dai due punti alle estremità, e vengono restituiti tutti i punti i cui poligoni contengono anche solo parte della linea.

API per la console di amministrazione

Le API per servire le richieste della console di amministrazione vengono fatte al path *nomeServer/rest/management* e l'annotazione per **Spring** è la seguente:

Listing 3.29: Annotazione per l'API

```
@RestController
@RequestMapping("/rest/management")
class ManagementResource {...}
```

In questo modo si indica al compilatore che deve partire un controller di Spring che si mette in ascolto sul path indicato per eventuali richieste HTTP in arrivo.
Cominciamo con l'API per il login di un account di amministrazione:

Listing 3.30: API di login di un account di amministrazione

```
@GetMapping ("/login")
fun login(@RequestParam(value = "mail") mail: String,
          @RequestParam(value = "pass") pass: String) =
        logManager(User(mail, "", pass))

fun logManager(user: User) : List<Point>
{
    if (mongoLogManager(user))
    {
        val list = mutableListOf<Point>()
        for (place : Place in loginManager(user.mail!!))
        {
            list.add(makePoint(place))
        }
        return list
    }
    else
        throw UserNotFoundException("User not registered")
}

fun makePoint(place: Place) : Point
{
    val area = mutableListOf<Position>()

    for (pos : Position in place.area!!.coordinates.exterior.toTypedArray())
        area.add(Position(pos.values[1], pos.values[0]))

```

```
    return
        Point( Position (place . coordinates !!. position . values . get(1), place . coordinates !!. position . values . get(0)),
                        place . name!!,
                        place . description !!,
                        area . toTypedArray(),
                        place . image!!)
}
```

L'API quindi viene fatta al path */login*, riceve come parametri *mail* e *pass* e ritorna una lista di *Point*, cioè di punti di interesse appartenenti all'account che sta effettuando il login.
In questo caso al momento del login vengono ritornati tutti i punti di interesse aggiunti dall'utente che sta effettuando la connessione. La funzione *makePoint* si occupa di convertire un oggetto di tipo *Place* in uno di tipo *Point* in modo da tornare una classe che può essere decodificata dal client che riceve la risposta.
Nella seguente funzione invece si vede come viene aggiunto un nuovo elemento alla collezione degli utenti di amministrazione:

Listing 3.31: API di aggiunta di un account di amministrazione

```
@PostMapping ("/register")
@ResponseStatus(HttpStatus.CREATED)
fun  register (@RequestParam(value = "mail") mail: String,
               @RequestParam(value = "name") name: String,
               @RequestParam(value = "pass") pass: String) =
        newManager(User(mail, name, pass))

fun newManager(user: User)
{
    val mail = InternetAddress(user . mail)
    mail . validate ()

    if (newUserManager(user))
        return
    else
        throw UserAlreadyRegisteredException("User already registered")
}
```

La funzione *InternetAddress* si occupa di verificare che la mail fornita rispetti le regole riguardo la struttura della stringa che devono avere.
La API per l'aggiunta di un nuovo punto di interesse, invece è la seguente:

Listing 3.32: API di aggiunta di un nuovo punto di interesse

```
@PostMapping("/newPlace", consumes = [MediaType.APPLICATION_JSON_VALUE])
@ResponseStatus(HttpStatus.CREATED)
fun newPlace(@RequestParam(value = "mail") mail: String,
             @RequestParam(value = "pass") pass: String,
             @RequestBody(required = true) place: Point) : Point
        {
            logManager(User(mail, "", pass))
            return addPlace(mail, place)
        }

fun addPlace(mail:String, poi: Point): Point
{
    val file = Paths.get(poi.image)

```

```
    var pol = mutableListOf<Position>()

    for (elem: Position in poi.polygon)
        pol.add(Position(elem.lon, elem.lat))

    val place = Place(Point(Position(poi.position.lon, poi.position.lat)), poi.name, poi.description,
        Polygon(pol), mail, file.fileName.toString())

    val point = makePoint(place)

    if (newPlace(place))
        return point
    else
        throw InternalError()
}
```

In questo caso la funzione *logManager* è la stessa del login ed è già stata mostrata.

API per l'applicazione Android

Le API per servire le richieste dell'applicazione Android vengono fatte al path *nomeServer/rest/users* e l'annotazione per **Spring** è la seguente:

Listing 3.33: Annotazione per le API che servono le richieste dell'applicazione Android

```
@RestController
@RequestMapping("/rest/users")
class UserResource {...}
```

L'API di registrazione di un nuovo utente è la seguente:

Listing 3.34: API di registrazione di un nuovo utente

```
@PostMapping("/register")
@ResponseStatus(HttpStatus.CREATED)
fun register (@RequestParam(value = "mail") mail: String,
              @RequestParam(value = "name") name: String,
              @RequestParam(value = "pass") pass: String) =
        newUserApp(User(mail,name,pass))

fun newUserApp(user: User)
{
    val mail = InternetAddress(user.mail)
    mail.validate()

    if (newUser(user))
        return
    else
        throw UserAlreadyRegisteredException("User already registered")
}
```

Per il login degli utenti,invece, viene usata la seguente:

Listing 3.35: API di login di un utente dell'applicazione

```
@GetMapping("/login")
fun login(@RequestParam(value = "mail") mail:String,
          @RequestParam(value = "pass") pass:String) =
        logUser(User(mail, "", pass))

fun logUser(user: User)
```

```
{
    if (login(user))
        return
    else
        throw UserNotFoundException("User not registered")
}
```

Lo scopo di base dell'applicazione è quello di inviare la posizione del device e ricevere in risposta tutti i punti di interesse vicini. In questo senso l'API seguente è quella principale nella comunicazione con i dispositivi mobile:

Listing 3.36: API di ritorno dei punti vicini all'applicazione

```
@GetMapping("/points")
fun sendListPoints(@RequestParam(value = "latitude") lat: Double,
                   @RequestParam(value = "longitude") lon: Double,
                   @RequestParam(value = "mail") mail:String,
                   @RequestParam(value = "pass") pass:String) : List<PointOfInterest>
    {
        logUser(User(mail, "", pass))
        return matchPoints(lat, lon)
    }

fun matchPoints (lat : Double, lon: Double): List<PointOfInterest>
{
    var list = mutableListOf<PointOfInterest>()

    for (place: Place in matchPosition(Point(Position(lon, lat))))
    {
        try {
            list.add(PointOfInterest(place.coordinates!!.coordinates.values[1],
                                     place.coordinates!!.coordinates.values[0],
                                     place.name!!,
                                     place.description!!,
                                     place.image!!))
        }
        catch (npe: NullPointerException)
        {
            continue
        }
    }
    return list
}
```

In questo modo viene tornata all'applicazione una lista di *PointOfInterest* i cui poligoni contengono le coordinate inviate precedentemente. Per servire la richiesta di un percorso tra due punti che l'applicazione può fare viene utilizzata la seguente API:

Listing 3.37: API per l'on the road

```
@GetMapping("/road")
fun onTheRoad(@RequestParam(value = "startingLat") sLat: Double,
              @RequestParam(value = "startingLon") sLon: Double,
              @RequestParam(value = "finalLat") fLat: Double,
              @RequestParam(value = "finalLon") fLon: Double,
              @RequestParam(value = "mail") mail: String,
              @RequestParam(value = "pass") pass: String) : List<PointOfInterest>
{
    logUser(User(mail, "", pass))
    return matchLine(sLat,sLon,fLat,fLon)
}

fun matchLine(sLat: Double, sLon: Double, fLat: Double, fLon: Double): List<PointOfInterest>
```

```
{
    var list = mutableListOf<PointOfInterest>()
    var line = listOf<Position>(Position(sLon,sLat), Position(fLon,fLat))
    for (place: Place in matchLine(LineString(line)))
    {
        try {
            list.add(PointOfInterest(place.coordinates!!.coordinates.values[1],
                    place.coordinates!!.coordinates.values[0],
                    place.name!!,
                    place.description!!,
                    place.image!!))
        }
        catch (npe: NullPointerException)
        {
            continue
        }
    }
    return list
}
```

Viene creata una lista di punti con due elementi che viene usata per creare un oggetto di tipo *LineString*; questa viene usata per farne le intersezioni con tutti i punti salvati sul database.

Capitolo 4

Valutazioni finali

4.1 Introduzione alle valutazioni

Lo sviluppo del progetto si è concentrato sull'interoperabilità con il Java e il Javascript; sono state utilizzate le librerie Java e ci si è concentrati sull'utilizzo di strumenti già presenti negli altri linguaggi per verificarne la qualità di utilizzo su Kotlin. Nelle prossime sezioni di questo capitolo verranno esposte le valutazioni su quanto efficace sia questa presunta interoperabilità; prima di questo si tratterà delle caratteristiche intrinseche del linguaggio.

4.2 Valutazioni sul linguaggio

Il linguaggio è duttile e conciso; ha parecchie particolarità che permettono al programmatore di risparmiare la scrittura di molte righe di codice.

Stili di programmazione

Il server è stato sviluppato principalmente con uno stile di programmazione funzionale, ma con l'utilizzo di alcuni oggetti e classi che hanno permesso di ridurre il numero di linee di codice scritte. C'è la piena possibilità di utilizzo di entrambi gli stili di programmazione, in modo da poter utilizzare quello più opportuno nelle diverse situazioni.

Argomenti di default delle funzioni

Nella dichiarazione di una funzione, come è già stato detto nel capitolo dedicato a Kotlin, alcuni argomenti possono avere un valore di default. Spesso capita nella programmazione a oggetti di dichiarare metodi che

hanno gli stessi nomi, ma in cui varia il numero degli argomenti. In Kotlin basta dichiarare un metodo e poi impostare i valori di default; in questo modo se un valore non viene passato, viene impostato automaticamente al suo valore di default. Questo permette di risparmiare parecchio sia in numero di righe di codice scritte, sia per quantità di metodi dichiarati.

Tipi Nullable

L'esplicita dichiarazione nel codice di variabili che possono o meno essere null evita nella stragrande maggioranza dei casi il presentarsi di uno degli errori più comuni e temuti nella programmazione: il *NullPointerException*. Questo perché la maggior parte dei possibili errori si presentano direttamente a tempo di compilazione.

Data Class

Le *Data class* sono estremamente utili per la riduzione di complessità e dimensione del programma. Scrivendo una semplice riga di codice vengono implementate dal compilatore:

- *getter* e *setter* di ogni proprietà
- Tutte le funzioni *componentN()*
- La funzione *toString()*
- Le funzioni *equals()*, *hashCode()* e *copy()*

Interpolazione nelle stringhe

Antecedendo *$* a una espressione interna a una stringa, si sta implicitamente dicendo al compilatore che quella è un'espressione e che la stringa deve essere costruita calcolandone il valore prima di inserirlo. Sembra banale, ma in questo modo la gestione delle stringhe risulta essere molto più immediata.

4.3 Lato Server

Nello sviluppo del server non sono state incontrate particolari difficoltà nell'utilizzo di Kotlin; in realtà il suo utilizzo ha reso anche più facile la creazione del RESTfull web service. In questo aspetto, il linguaggio

è molto maturo e può essere tranquillamente sostituito al Java, anche sfruttando l'enorme bagaglio di librerie che offre quest'ultimo.

Spring

L'utilizzo di Spring per Kotlin è molto intuitivo. Come si è visto è sufficiente importare le librerie del framework tramite il Gradle e in automatico viene predisposto tutto l'ambiente per potersi mettere in ascolto di richieste HTTP. Le librerie messe a disposizione sono molto efficaci e con poche annotazioni è possibile sviluppare un'API con poche righe di codice scritte.
Spring offre anche un generatore on-line, in cui è possibile impostare alcuni parametri del progetto che si vuole iniziare e viene in automatico generato il progetto da cui cominciare a scrivere il codice. In pratica in qualche minuto si ha già pronto un ambiente su cui cominciare a sviluppare le API.

Interoperabilità con le librerie Java

Non è stato riscontrato nessun problema nell'utilizzo delle librerie Java all'interno del codice Kotlin. Qualunque libreria necessaria allo sviluppo del server è stata importata e utilizzata con una compatibilità del 100%.

Conversione automatica da Java a Kotlin

Come già detto in precedenza, è possibile convertire il codice Java in Kotlin automaticamente tramite un convertitore integrato nell'IDE IntelliJ. Questo strumento è utilissimo in quei casi in cui si vuole migrare un applicativo Java in Kotlin; purtroppo, però, non è infallibile. Capita molto spesso di essere costretti ad agire manualmente sul codice generato per risolvere alcuni problemi che il convertitore non è stato in grado di gestire in automatico. Nonostante questo, però, è molto utile anche solo per evitare di dover tradurre tutto il programma manualmente, consentendo una perdita di tempo minima per la correzione delle imperfezioni.

Coroutines

L'introduzione delle coroutine ha aiutato a migliorare la scalabilità del server. Queste, infatti, permettono di dividere l'esecuzione del

programma in parti distinte, trattando il flusso come se fosse una macchina a stati. In questo contesto dividere il programma su diversi dispositivi è un'operazione molto più semplice che in precedenza.
Le coroutine sono ancora in fase sperimentale, ma ci sono ottimi presupposti per un loro maggiore utilizzo futuro, quando verranno rilasciate ufficialmente.

4.4 Front-End Web

Nel caso del Front-End Web la situazione non è così rosea come nel lato server. L'interoperabilità con un linguaggio come il JavaScript non è per niente banale, soprattutto per la gestione dei tipi molto diversa per i due linguaggi.

Compilatore JavaScript

Il compilatore JavaScript di Kotlin si occupa della traduzione da un linguaggio all'altro. L'operazione viene fatta senza grossi problemi, ma bisogna conoscerne i limiti, soprattutto legati al sistema dei tipi. In JavaScript ad esempio non esistono oggetti di tipo lista; ne consegue che nel momento in cui si ha bisogno di una lista di oggetti, occorre anche in Kotlin utilizzare gli array per poter effettuare la traduzione.

Libreria kotlinx.html

L'utilizzo della libreria *kotlinx.html* consente tramite l'utilizzo di costruttori *type-safe* di generare codice html da Kotlin. L'interoperabilità in questo caso è piena; non è stato riscontrato nessun problema al momento di dover generare codice html per creare pagine web.

Tool ts2kt

Come spiegato in precedenza è stato usato un tool di conversione di codice TypeScript in codice Kotlin: *ts2kt*. La necessità era quella di utilizzare le librerie di Google Maps, che sono in TypeScript, per il loro utilizzo all'interno del codice del progetto.
La conversione, però, non è perfetta; anzi, è stato necessario agire sul codice generato per correggere errori di compilazione. Dato che la quantità di librerie TypeScript sono parecchie è necessario migliorare l'efficacia di questo strumento se si vuole pensare di utilizzare il

Kotlin anche per la creazione di pagine Web; oppure, potrebbe essere necessario implementare queste librerie anche per il Kotlin, risolvendo il problema in modo più diretto.

Capitolo 5

Conclusione

5.1 Resoconto

Dopo aver spiegato le caratteristiche del linguaggio, il lavoro svolto sul progetto e le valutazioni sul suo utilizzo, è possibile trarre delle conclusioni al fine di rispondere allo scopo di questa tesi.

Java e Server

Alla luce del lavoro fatto per lo sviluppo del server, si può affermare che il passaggio all'utilizzo di Kotlin in sostituzione al Java non è solo possibile, ma anche auspicabile. E' più conciso, più semplice da imparare e può essere utilizzato anche per la programmazione funzionale.
Il passaggio dal Kotlin al Java è facilitato dallo strumento messo a disposizione da JetBrains, se si sorvola su alcuni problemi minori di conversione. Molti framework stanno fornendo sempre più supporto a Kotlin, come nel caso di quello usato per il RESTful web service di questo progetto: Spring. L'interoperabilità con le innumerevoli librerie di Java è totale e permette di sfruttare l'enorme bagaglio di strumenti sviluppati in più di 20 anni. Inoltre, il linguaggio è ancora in sviluppo; ne consegue che tutti i piccoli problemi ancora presenti saranno con molta probabilità risolti con il tempo.
Da questa analisi, però, viene a galla un problema: l'interoperabilità con le librerie Java porta alla naturale conseguenza che in molti casi si utilizzano quest'ultime durante lo sviluppo in Kotlin. Quindi, se un programmatore vuole usare il Kotlin invece del Java è comunque costretto a essere consapevole di come integrarne le librerie e del loro funzionamento. In pratica è ancora troppo dipendente dal Java e non è banale imparare il Kotlin senza conoscere almeno l'uso delle librerie

del primo.

Javascript e Web

Per la programmazione web la situazione non è così positiva. Nonostante il compilatore JavaScript funzioni bene e garantisca la generazione senza errori del codice, l'utilizzo di Kotlin per la programmazione web non è così immediato.
Lo sviluppo di pagine HTML è facilitato dalla libreria *kotlinx.html*, che è molto semplice e immediata da usare, ma è ancora in fase sperimentale.
Per quanto riguarda il codice JavaScript, invece, le differenze sulla gestione dei tipi tra i due linguaggi non rendono semplice l'utilizzo di Kotlin per lo sviluppo, ma impongono delle limitazioni. Le librerie Kotlin per la programmazione web sono ancora molto poche e ci si ritrova costretti a tradurre quelle scritte per il TypeScript; il tool di traduzione, però, non funziona perfettamente e dover agire a mano sul codice per risolverne i problemi non è banale.
In pratica allo stato attuale utilizzare Kotlin per la creazione di pagine web porta più complicazioni che semplificazioni e quindi non è conveniente. Se in futuro verranno create molte più librerie per l'interazione con il JavaScript, tuttavia, la valutazione potrebbe essere diversa.

5.2 Sviluppi Futuri

Questa tesi si è concentrata sull'interoperabilità del linguaggio con Java e JavaScript, sull'utilizzo delle loro librerie e sulla sua interazione con framework e strumenti propri degli altri linguaggi.
Per eventuali sviluppi futuri è possibile analizzare meglio come l'utilizzo delle coroutine può migliorare prestazioni e scalabilità di un'applicazione Kotlin; soprattutto come possono aiutare nello sviluppo di server che devono gestire grosse moli di dati. Dato che sono in fase sperimentale potrebbe essere interessante anche osservarne l'evoluzione e verificarne l'efficacia col procedere dello sviluppo.
Un altro ambito di ricerca potrebbe essere rappresentato dallo studio di *kotlin/native*. Questo permette di compilare del codice nativo partendo da Kotlin; quindi si ottiene un eseguibile che non ha bisogno di una virtual machine per essere eseguito. Questo strumento potreb-

be rivelarsi molto utile nello sviluppo di applicativi per iOS e quindi studiarne l'efficacia realizzandone una sarebbe interessante.

Bibliografia

[1] *Build automation con Gradle*. URL: https://www.html.it/articoli/build-automation-con-gradle.

[2] *Google is adding Kotlin as an official programming language for Android development*. URL: https://www.theverge.com/2017/5/17/15654988/google-jet-brains-kotlin-programming-language-android-development-io-2017.

[3] *Google Maps JavaScript API*. URL: https://developers.google.com/maps/documentation/javascript/tutorial?hl=it.

[4] *I principi dell'architettura RESTful*. URL: https://www.html.it/pag/19596/i-principi-dellarchitettura-restful.

[5] JetBrains. *Kotlin Reference*. URL: https://kotlinlang.org/docs/reference.

[6] *Kotlin FAQ*. URL: https://kotlinlang.org/docs/reference/faq.html.

[7] *kotlinx.html wiki*. URL: https://github.com/kotlin/kotlinx.html/wiki.

[8] *MongoDB Java Driver Documentation*. URL: http://mongodb.github.io/mongo-java-driver/3.8.

[9] *Parallel versus distributed computing*. URL: https://www.packtpub.com/mapt/book/application_development/9781787126992/1/ch01lvl1sec10/parallel-versus-distributed-computing.

[10] *Programmazione Distribuita*. URL: http://linuxdidattica.org/docs/altre_scuole/planck/tecnologie-web/tecnologie-web16.html.

[11] *rfc2616*. URL: https://tools.ietf.org/html/rfc2616.

[12] *rfc7159*. URL: https://tools.ietf.org/html/rfc7159.

[13] *Spring Boot Reference Guide*. URL: https://docs.spring.io/spring-boot/docs/2.0.5.RELEASE/reference/htmlsingle.

[14] Stefan Bocutiu Stephen Samuel. *Programming Kotlin*.