**Università degli Studi di Padova**

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica


# Metodologie agili applicate allo sviluppo di una componente d'interfaccia grafica web in Kotlin per l'analisi di Big Data

*Tesi di laurea*

*Relatore*
Prof. Claudio Enrico Palazzi

*Laureando*
Marco Rampazzo

Anno Accademico 2019-2020

# Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di trecentoventi ore, dal laureando Marco Rampazzo presso l'azienda GRUPPO4 S.r.l. L'obiettivo principale da raggiungere era quello di realizzare una componente d'interfaccia grafica il cui scopo è quello di permettere ad un utente di esplorare dati mediante una tabella pivot. Questo componente verrà utilizzato dall'azienda ospitante per sostituire un loro software correntemente in uso da oltre dieci anni.

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

# Introduzione

Questa tesi descrive l'esperienza e il percorso lavorativo svolto presso l'azienda GRUPPO4 sotto la supervisione di Tobia Conforto.

## 1.1 L'azienda

Gruppo4 è una web agency Padovana, da oltre vent'anni ha accumulato competenze e l'esperienza necessaria per fornire soluzioni efficaci e innovative nel settore web. Mediante un modello organizzativo consolidato e certificato sviluppano applicazioni web che si distinguono per la chiarezza dell'interfaccia utente (UX/UI) e per la loro usabilità.

## 1.2 Gli obiettivi del progetto

L'obiettivo principale di questo progetto consiste nella realizzazione di un componente di interfaccia grafica per il web in Kotlin. Il componente deve essere una tabella pivot interattiva che permette ad un utente la possibilità di esplorare liberamente i *big data* contenuti al suo interno. Oltre alla realizzazione del componente, questo progetto ha anche lo scopo di valutare l'efficacia di Kotlin nel realizzare interfacce utente per il web in quanto l'azienda utilizza Kotlin nello sviluppo di Application Program Interface (API).

## 1.3 Tabella pivot

L'azienda mi ha fornito una descrizione accurata della tabella pivot, in particolare della sua struttura e delle sue funzionalità. Nelle prossime sottosezioni descriverò gli elementi che la compongono e le possibili interazioni con l'utente.

### 1.3.1 Struttura

La tabella pivot è stata suddivisa principalmente in due parti ben distinte: le dimensioni e i dati ad esse. Le dimensioni rappresentano le celle di intestazione della tabella. Esse sono presenti su tutti e due gli assi (righe e colonne) e su un singolo asse possono essere presenti molteplici dimensioni. I dati riferiti alle celle di intestazione corrispondono alla

seconda parte della tabella pivot e corrispondo a delle semplici celle che contengono valori interi.

### 1.3.2 Funzionalità

La tabella deve essere completamente esplorabile da un utente per questo motivo sono presenti due tipi di azioni: sulle dimensioni e sulle singole celle di una dimensione.
Ogni cella di una dimensione deve poter essere aperta o chiusa con un semplice pulsante. Mentre ogni cella di una dimensione può essere aperta per renderizzare i figli di quel nodo se essi esistono.

## 1.4 Organizzazione del testo

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: $parola_G$;
- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

# Capitolo 2

# Processi e metodologie

In questo capitolo verrà fornito una descrizione dei metodi e dei processi messi in atto durante il tirocinio, in particolare riguardo: la metodologia agile, la programmazione funzionale e la gestione del progetto in locale e l'organizzazione del lavoro.

## 2.1 Metodologia Agile

Per lo sviluppo del prodotto è stato deciso di applicare una metodologia agile in modo da reagire velocemente a possibili problemi e cambiamenti dei requisiti così da migliorare e velocizzare la realizzazione dell'applicativo. L'azienda ha deciso di utilizzare una metodologia agile simile a $SCRUM_G$. Infatti applicare nella sua interezza il metodo $SCRUM_G$ sarebbe stato impossibile dato il ristretto numero di sviluppatori nel team di sviluppo e il limitato periodo riservato alla codifica.
Le caratteristiche principali della metodologia agile applicata per la realizzazione di questo progetto sono le seguenti:

- **Modello incrementale**: vengono realizzati rilasci multipli e successivi che aiutano a definire più chiaramente i requisiti più importanti dato che essi verranno implementati per primi. Ogni rilascio corrisponde ad una parte funzionante di applicazione;
- **Modello iterativo**: un modello iterativo ha la caratteristica di avere una maggior capacità di adattamento in seguito a problemi di implementazione e cambiamenti nei requisiti;
- **Organizzazione in sprint di sviluppo**: il periodo di codifica viene suddiviso in sprint di sviluppo, data la breve durata del tirocinio curriculare essi avranno una durata di circa 4-5 giorni;

Gli elementi che caratterizzano la metodologia agile utilizzata sono:

- **Product backlog**: documento molto importante che contiene i requisiti e le funzionalità del prodotto definiti mediante le User Stories ordinate per priorità (*Business value*);
- **Sprint backlog**: rappresenta l'insieme delle User Stories da realizzare nello sprint;

- **Increment**: insieme di tutte le user stories che sono state completate dall'ultima release del software.

La metodologia agile applicata presenta inoltre i seguenti eventi che sono stati ripetuti ogni settimana in corrispondenza di ogni sprint:

- **Sprint Planning**: riunione tra i partecipanti del team di sviluppo dove vengono determinati quali user stories del product backlog verranno completate nello sprint;
- **Daily Stand-up**: breve riunione giornaliera dove ogni membro descrive velocemente i progressi dall'ultimo *Daily Stand-up*, i problemi riscontrati e i suoi prossimi obiettivi, questo evento è stato, in molte occasioni, sostituito da una riunione telematica;
- **Sprint Review**: riunione dove viene mostrato il prodotto con tutte le modifiche completate durante lo sprint (*increment*);
- **Backlog refinement**: riunione per aggiungere, modificare o eliminare user stories dal *product backlog*;
- **Retrospective**: riunione finale dove vengono determinati i fattori positivi e negativi in modo da identificare le strategie migliori per ottenere un miglioramento continuo dei processi.

## 2.2 Programmazione funzionale

La programmazione funzionale è un paradigma di programmazione dichiarativa dove un programma è costituito dall'applicazione e dalla composizione di funzioni. In questo progetto si è utilizzata, dove possibile, la programmazione funzionale mediante le *Funzioni di ordine superiore*$_G$ fornite da Kotlin. Queste funzioni sono molto utili perchè permettono di scrivere codice più leggibile, conciso e soprattutto hanno la caratteristica di evitare *side-effects*$_G$. Come definito dalla documentazione di Kotlin, le funzioni scritte nel linguaggio Kotlin sono considerate come *first-class* quindi esse possono essere contenute in variabili e strutture dati, passate come argomento di altre funzioni e ritornate da altre *Funzioni di ordine superiore*. Le higher-order functions più usate nel progetto sono state:

- `map`: ritorna una nuova lista contenente i risultati ottenuti dalla funzione di trasformazione per ogni elemento della collezione originale;
- `sortedWith`: ritorna una lista contenente tutti gli elementi della lista originale ordinati secondo un `comparator` una funzione che impone una condizione tra gli elementi della lista;
- `flatMap`: stesse funzionalità di `map` ma può essere invocato su più di una lista e ritorna una collezione unica;
- `groupBy`: raggruppa gli elementi di una lista ottenuta con i risultati della funzione `keySelector` applicata ad ogni elemento della lista originale, ritorna una collezione di tipo: `Map<K, V>`.

## 2.3 Versionamento della soluzione

### 2.3.1 Git

Git è un *VCS*$_G$ (Version Control System) distribuito che permette di tenere traccia delle modifiche in un prodotto software e di organizzarne la codifica.

#### 2.3.1.1 Feature-branch

In questo tirocinio è stata usata la tecnica del feature-branch: ogni aggiunta di feature corrisponde all'apertura di un nuovo branch$_G$ che deve essere poi approvato previa verifica prima di essere unito al branch `master`.

### 2.3.2 Gitlab

Gitlab è uno strumento web che permette di implementare un DevOps lifecycle che fornisce una gestione di repository git, un *ITS*$_G$ (Issue Tracking System) e altri strumenti quali la *Continuous integration* e *Continuous deployement*. Per lo sviluppo di questo progetto mi è stato fornito l'accesso al server privato aziendale Gitlab.

## 2.4 Ambiente di sviluppo locale

### 2.4.1 IntelliJ Idea

IntelliJ IDEA Community Edition è una *IDE*/glosp realizzata da JetBrains che fornisce funzionalità di supporto per lo sviluppo di molti linguaggi, specialmente Kotlin. Questa IDE è stata vivamente consigliata dal mio tutor aziendale per lo sviluppo in Kotlin rispetto ad altri editor per molti vantaggi come l'autocompletamento, la possibilità di eseguire un refactoring automatico di funzioni e classi e una interfacia grafica per eseguire task di Gradle.

### 2.4.2 Gradle

Gradle è uno strumento di *build automation*$_G$ per molti linguaggi tra cui Kotlin e Java. Gradle è stato usato per la gestione e l'installazione delle dipendenze del componente, il file di configurazione di Gradle è `build.gradle.kts`, al suo interno sono definite le seguenti dipendenze:

- `stdlib-js`: insieme di classi e funzioni in kotlin che forniscono un entry-point per funzioni e oggetti Javascript;
- `kotlin-react`: implementazione di kotlin della liberia React;
- `react`: pacchetto npm della libreria React necessario per il funzionamento di `kotlin-react`;
- `kotlin-redux`: implementazione di kotlin della libreria Redux;
- `kotlin-react-redux`: implementazione di kotlin della libreria React, Redux;
- `kotlin-extensions`: libreria che fornisce dei *wrapper* per oggetti JS e alcune funzione *helper* per `kotlin-react`;

- `kotlinx-serialization-core`: libreria utilizzata per ottenere funzioni di parse per JSON;
- `test-js`: libreria di test per Kotlin/js;
- `kotlinx-coroutines-core`: libreria utilizzata per eseguire funzioni asincrone.

Oltre alla gestione delle dipendenze Gradle offre delle task, utili per compilare ed eseguire l'applicativo. Nell'ambito del progetto ho utilizzato la task: `runDevelopment` e come parametro della task: `--continuous` in modo da eseguire automaticamente la compilazione del codice quando viene cambiato uno dei file sorgente.

### 2.4.3 Organizzazione del lavoro

Per l'organizzazione del lavoro, in particolare per la gestione dei macro obiettivi di ogni sprint, ho utilizzato l'*ITS* fornito da Gitlab che fornisce un'interfaccia *Kanban* che permette di categorizzare in modo efficace le singole attività da realizzare.

# Capitolo 3

# Progettazione

In questo capitolo verrà descritta la progettazione dell'intero prodotto. In particolare verranno spiegati i seguenti argomenti:

- Dataflow dell'applicazione;
- Gestione dello stato;
- Stato;
- Parser e adapter;
- Componenti grafici.

## 3.1 Dataflow dell'applicazione

Un aspetto che è stato molto discusso dal team di sviluppo riguarda il *dataflow*$_G$ dell'applicazione. Durante la prima settimana del tirocinio, oltre allo studio delle tecnologie, si è pensato a come verrà gestito lo stato dell'applicazione e in particolare delle possibili soluzioni per garantirne la scalabilità. In questa sezione verrà descritto il dataflow fornito da React e Redux, infine verrà identificato il dataflow adottato nell'ambito del progetto.

### 3.1.1 React

React è una libreria per realizzare interfacce utente che utilizza un *dataflow* unidirezionale. Questo perchè ogni componente può avere uno stato locale accessibile solo da se stesso e può passare informazioni ai suoi componenti figli mediante le *props*$_G$. Il risultato è uno stato che dipende dalla gerarchia di componenti grafici.
Questo dataflow è molto semplice però presenta alcune limitazioni per quanto riguarda la scalabilità. Per avere uno stato unico di tutta l'applicazione bisognerebbe dare la responsabilità ad un componente grafico di gestirlo e passarlo mediante le sue *props*. L'architettura che ne deriva, nel caso di applicazioni complesse con molti componenti, è limitata, difficile da manutenere e poco scalabile.
Tuttavia questo semplice dataflow ha il vantaggio che per famiglie di componenti piccole i cambiamenti di stato locale e i passaggi di informazioni mediante le *props* sono molto veloci e semplici.

### 3.1.2 Redux

Per garantire scalabilità e facilità nella gestione dello stato dell'applicazione abbiamo discusso riguardo l'utilizzo di Redux. Essa offre un dataflow unidirezionale dove lo stato è gestito in una struttura di Redux chiamata *store*. Questa struttura esterna ai componenti grafici rende la gestione dello stato dell'applicazione più prevedibile e manutenibile. Redux si basa su tre principi:

- lo stato è l'unica fonte di verità;
- lo stato non è modificabile direttamente;
- le modifiche avvengono medianti *funzioni pure*$_G$ che creano un nuovo stato per evitare *side effects*$_G$.

Gli elementi dell'architettura di Redux sono i seguenti:

- **Actions**: oggetti che rappresentano un azione che innesca un update dello stato;
- **Reducers**: *funzioni pure* che hanno come parametro un Actions e si occupano di modificare lo stato;
- **Store**: lo Store è un'interfaccia che contiene lo stato dell'applicazione e fornisce funzioni per leggere, modificare e registrare *listeners*$_G$ allo stato.

### 3.1.3 Soluzione

La soluzione adottata è stata quella di usare sia React che Redux. React è stato usato per gestire solo gli aggiornamenti visivi dell'applicazione, mentre Redux per gestire lo stato completo dell'applicazione. Il dataflow del prodotto è quindi il seguente:

1. se l'utente esegue una azione che implica un cambiamento dello stato verrà mandata allo Store una Action;
2. lo Store si occuperà di chiamare un Reducer in modo da ricevere lo stato successivo;
3. lo stato verrà aggiornato e i cambiamenti saranno visibili a tutti i componenti React che sono registrati allo Store.

Utilizzando questo dataflow lo stato dell'applicazione non dipende da un componente grafico perchè è esterno alla gerarchia dei componenti grafici. Quindi ogni componente React che ha bisogno di una frazione dei dati dello stato può semplicemente registrarsi allo Store per ricevere i dati.

## 3.2 Stato

Per quanto riguarda lo stato dell'applicazione ho analizzato la struttura della tabella fornita dall'azienda e le funzionalità richieste. L'idea alla base della struttura dello stato era quella di avere una struttura dati semplice da iterare in modo da avere funzioni di renderizzazione concise e facili da manutenere. Dalla struttura della tabella pivot e dalla descrizione delle sue funzionalità ho ricavato le seguenti strutture dati:

**data class DimensionsNode** - rappresenta una cella d'intestazione, deve contenere:

- `id`: codice della cella;
- `label`: testo da renderizzare;
- `level`: indica a che dimensione dell'intestazione appartiene la cella;
- `childDepth`: indica la profondità della cella in una gerarchia ad albero;
- `path`: codice identificativo della cella costituito da una lista di `id` che rappresenta la gerarchia di una cella;
- `actionType`: indica il tipo dell'azione che bisogna eseguire, esso verrà indicato con una classe di tipo: `enum class NodeActionType`;
- `isChild`: indica se la cella è un figlio di un'altra cella.

**data class BodyCells** - rappresenta una cella dei dati, deve contenere:

- `value`: indica il dato;
- `cPath`: indica l'insieme dei codici delle dimensioni delle colonne a cui appartiene il dato;
- `rPath`: indica l'insieme dei codici delle dimensioni delle righe a cui appartiene il dato.

**data class HeaderAction** - rappresenta un tipo di azione, deve contenere:

- `actionType`: indica il tipo dell'azione che bisogna eseguire, esso verrà indicato con una classe di tipo: `enum class NodeActionType`;
- `dim`: indica la dimensione su cui applicare l'azione;
- `depth`: indica la profondità all'interno della dimensione su cui applicare l'azione.

**enum class NodeActionType** - rappresenta un tipo di azione:

- `EXPAND`: indica che la cella può espandere per mostrare i suoi figli;
- `COLLAPSE`: indica che la cella può nascondere i suoi figli;
- `NULL`: indica che la cella non ha un'azione.

Quindi per rappresentare l'intero stato della tabella ho definito un ultimo `data class`:

**data class TableState**

- `rows`: matrice di DimensionsNode, rappresenta le dimensioni delle righe;
- `cols`: matrice di DimensionsNode, rappresenta le dimensioni delle colonne;
- `cells`: matrice di BodyCells, rappresenta le celle contente i dati della tabella;
- `rowActions`: matrice di HeaderAction, rappresenta le azioni disponibili per le dimensioni delle righe;
- `colActions`: matrice di HeaderAction, rappresenta le azioni disponibili per le dimensioni delle colonne.

## 3.3 Gestione dello stato

Come descritto nella sezione precedente nell'ambito di questo progetto è stata adottata la libreria Redux, per applicare al meglio la sua architettura ho progettato la sua codifica nel seguente modo. Per prima cosa ho suddiviso lo stato dell'applicazione in "Slice" cioè in pezzo di stato. In questo modo la struttura dello stato è modulare e scalabile dato che, se questa applicazione verrà ampliata in un futuro basterà aggiungere uno slice allo stato. Ogni slice deve essere strutturato nel seguente modo:

**Listing 3.1:** Esempio Slice

```
object Slice {
        // Stato
        data class State( ... )

        // Thunk
        private val thunk = Thunk()
        fun funcThunk() : RThunk = thunk

        // Actions
        class Action(): RAction
        ...

        // Reducer
        fun reducer(state: State = State(), action: RAction) : State { ... }
}
```

### 3.3.1 Stato

Lo stato gestito da Redux è definito da un data class che contiene tutte le informazioni dello Slice. Più precisamente conterrà i seguenti campi dati:

- `isLoading`: booleano che indica se si stanno effettuando chiamate http o funzionalità asincrone;
- `rows`: insieme di celle che rappresentano le dimensioni delle righe;
- `cols`: insieme di celle che rappresentano le dimensioni delle colonne;
- `cells`: insieme di celle che contengono i dati relativi alle dimensioni;
- `rowActions`: insieme di possibili azioni eseguibili sulle dimensioni delle righe;
- `colActions`: insieme di possibili azioni eseguibili sulle dimensioni delle colonne.

### 3.3.2 Thunk

Dato che i cambiamenti allo stato innescati dalle Actions possono essere solo sincroni ho utilizzato un'interfaccia che fornisce le stesse funzionalità di un Actions ma ne espande l'utilità permettendo operazioni asincrone, questa interfaccia è definita come `thunk`. I *thunk* sono definiti come *middleware* di Redux e sono usati per effettuare operazioni asincrone complesse, l'interfaccia dei Thunk purtroppo non era presente nell'implementazione di Redux di kotlin quindi ho realizzato una semplice interfaccia che implementa le funzionalità dei thunk.

**Listing 3.2:** Interfaccia Thunk

```
interface RThunk : RAction {
        operator fun invoke(
        dispatch: (RAction) -> WrapperAction,
        getState: () -> AppState
        ) : WrapperAction
}

fun rThunk() =
applyMiddleware<AppState, RAction, WrapperAction, RAction, WrapperAction>(
{ store ->
        { next ->
                { action ->
                        if (action is RThunk)
                          action(store::dispatch, store::getState)
                        else
                          next(action)
                }
        }
}
)

val nullAction = js {}.unsafeCast<WrapperAction>()
```

### 3.3.3 Actions

Le actions sono definiti come delle classi di tipo `RAction` che vengono usati per innescare l'update dello stato. Solitamente un'action si occuperà di modificare solo un campo dato dello stato. Quindi dalla mia precedente progettazione dello stato ho individuato cinque Actions:

- `class UpdateRows(): RAction;`
- `class SetIsLoading(): RAction;`
- `class UpdateCols(): RAction;`
- `class UpdateCells(): RAction;`
- `class UpdateRowActions(): RAction;`
- `class UpdateColActions(): RAction.`

### 3.3.4 Reducer

Un *reducer* è una *funzione pura* che riceve come argomento un `RAction` e ritorna una copia dello stato modificato. In kotlin il modo per realizzare una funzione reducer è realizzare una struttura switch (in kotlin corrisponde ad una struttura when) dove vengono definiti tanti casi quanto sono le actions disponibili che vogliono essere utilizzate. Per le actions definite precedentemente:

**Listing 3.3:** Interfaccia Thunk

```
fun reducer(state: State = State(), action: RAction) : State {
  return when (action) {
    is UpdateRows -> state.copy(...)
    is SetIsLoading -> state.copy(...)
    is UpdateCols -> state.copy(...)
```

```
6       is UpdateCells -> state.copy(...)
7       is UpdateRowActions -> state.copy(...)
8       is UpdateColActions -> state.copy(...)
9       else -> state
10    }
11  }
```

## 3.4 Parser e adapter

Per la progettazione ho per prima cosa definito i passaggi necessari per ricevere i dati dalla API di Gruppo4. Ho individuato la necessità di due funzioni `fetch` e `sendAction`, la prima per richiedere i dati iniziali e la seconda per ottenere nuovi dati in seguito all'azione passata per argomento a `sendAction`.
Per quanto riguarda il parser del json ho studiato la struttura del json ritornato dall'API in modo da identificare tutte le data class `@Serializable` che mi permetteranno di mappare il json in data class. La struttura del json è la seguente:

**Listing 3.4:** Struttura JSON API

```
{
        "Rows": {
                "Paths": [
                ["_all", "_all", ...], [ ... ]
                ],
                "Actions": [
                { "Action": "C", "Dim": 1, "Depth": 0 }, { ... }
                ],
                "Tree": [
                {
                        "Code": "_all",
                        "Label": "All",
                        "SubDim": [
                        {
                                "Code": "_all",
                                "Label": "All",
                                "SubDim": null,
                                "Children": null
                        }
                        ]
                }
                ]
        }
        "Cols": // stessa struttura di "Rows"
        "Cells": [
          [51515, 2315, 747, 22],
          [ ... ]
        ],
        "Filters": [
          {
            "Filtered": false,
            "ActiveFilters": ["f1", "f2", "f3"],
            "Type": "list",
            "Name": "associazioni_provincia",
            "Label": "Associazioni per provincia"
          }
        ]
}
```

Da questa struttura ho identificato le seguenti `data class @Serializable`:

- `Gruppo4Json`: rappresenta la struttura generale del json;
- `Gruppo4Data`: rappresenta la struttura dei campi dati "Rows" e "Cols";
- `Gruppo4Filter`: rappresenta la struttura contenuta nel campo dati "Filters";
- `Gruppo4Actions`: rappresenta la struttura contenuta nel campo dati "Actions" all'interno di "Rows" e "Cols";
- `Gruppo4Node`: rappresenta la struttura contenuta nel campo dati "Tree" all'interno di "Rows" e "Cols" e corrisponde ad un nodo della struttura ad albero.

Per quanto riguarda la progettazione dell'adapter ho individuato le data class ottenibili dal JSON e le data class definite nello stato. Da esse ho individuato le seguenti funzioni da realizzare:

- `fun convertListOfGruppo4Node()`:
  - si occuperà di convertire una lista di `Gruppo4Node` in `DimensionsNode`;
- `fun convertTree()`:
  - si occuperà di convertire la struttura ad albero in una struttura più semplice da iterare;
- `fun convertListOfGruppo4Actions()`:
  - si occuperà di convertire una lista di `Gruppo4Actions` in `HeaderAction`;
- `fun convertCells()`:
  - si occuperà di unire i dati definiti nel campo dati "Cells" con i "Path" in modo da ottenere una matrice di `BodyCells`.

## 3.5 Componenti grafici

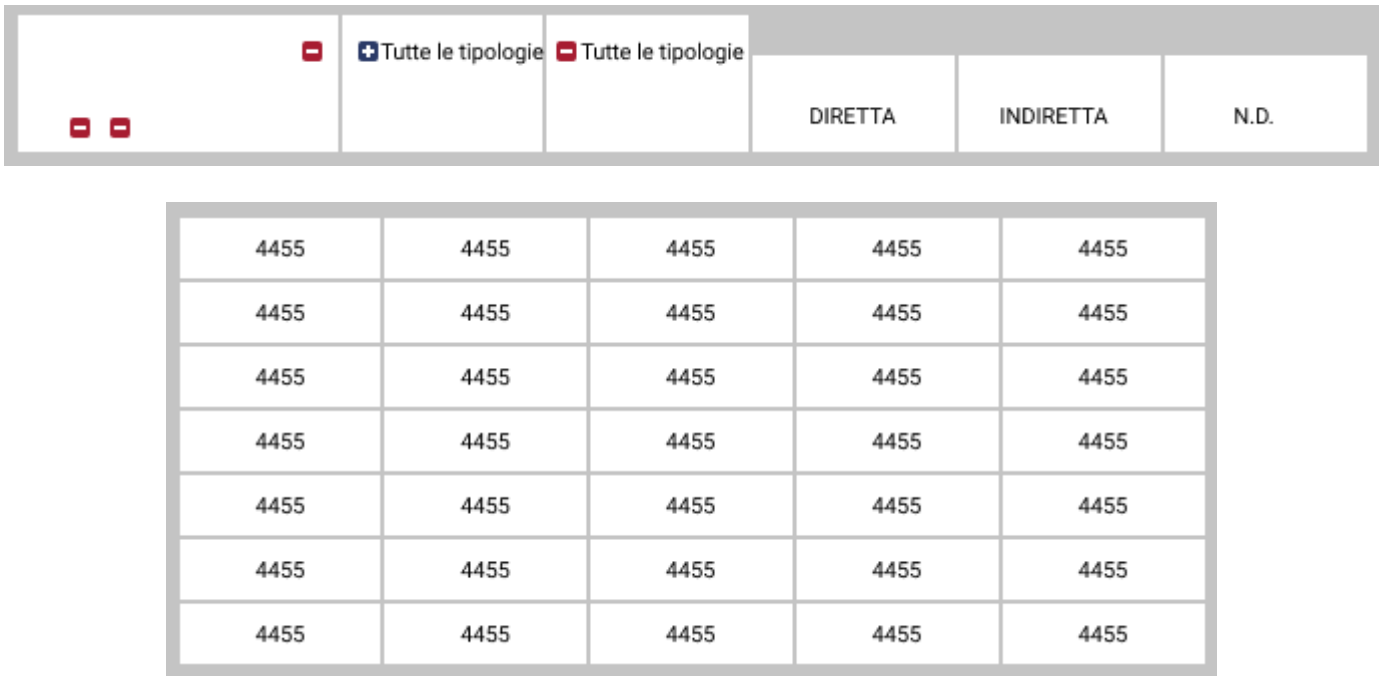
Per quanto riguarda la struttura dei componenti grafici l'azienda mi ha dato completa libertà per quanto riguarda la loro struttura. La progettazione di essi consiste nel loro mockup e la definizione degli stili grafici necessari. Questa prima immagine rappresenta il mockup iniziale da cui sono partito:

| | Tutte le tipologie | Tutte le tipologie | | | |
|---|---|---|---|---|---|
| | | | DIRETTA | INDIRETTA | N.D. |
| Tutti le associazioni | 4455 | 4455 | 4455 | 4455 | 4455 |
| CASARTIGIANI | 4455 | 4455 | 4455 | 4455 | 4455 |
| CGIL | 4455 | 4455 | 4455 | 4455 | 4455 |
| BL | 4455 | 4455 | 4455 | 4455 | 4455 |
| PD | 4455 | 4455 | 4455 | 4455 | 4455 |
| ABANO | 4455 | 4455 | 4455 | 4455 | 4455 |
| ALBIGNASEGO | 4455 | 4455 | 4455 | 4455 | 4455 |

Da questo mockup generale ho individuato i seguenti componenti grafici. In particolare ho definito la loro gerarchia di composizione:

- `Table`
  - `TableActionUI`
  - `TableHeader`
    - `TableLabel`
  - `TableSidebar`
    - `TableLabel`
  - `TableBody`

Di seguito sono dei mockup di tutti i componenti che ho individuato nella tabella pivot:




Da questi mockup ho deciso in che modo organizzare questi componenti. Ho deciso di suddividere il componente `Table` in quattro quadranti utilizzando `css grid`. Ogni quadrante conterrà i singoli componenti.
Per quanto riguarda `TableActionUI` ho pensato di suddividerlo in quattro quadranti in modo similare a `Table` così da posizionare i pulsanti delle azioni in modo semplice. In `TableBody` ho definito la sua struttura come una tabella html. Nel componente `TableHeader` ho deciso di realizzare una tabella per ogni dimensione disponibile con una singola riga contenente un'insieme di `TableLabel`. Il componente `TableSidebar` è simile a `TableHeader`, quindi per ogni dimensione disponibile viene creata una tabella con una singola colonna contenente un'insieme di `TableLabel`. Infine per quanto riguarda il componente `TableLabel` ho pensato di realizzarlo con un elemento `<th>` con al suo interno un possibile pulsante per l'azione eseguibile e una etichetta, posizionati mediante l'uso di `css flex`.

# Capitolo 4

# Codifica

La codifica del prodotto, come descritto nel capitolo 2 è stata suddivisa in sprint di sviluppo che corrispondono a circa 4-5 giorni lavorativi dove vengono implementati parte delle user stories indicate nel product backlog. Per ogni sprint verrano descritti lo Sprint Backlog e le Soluzioni che sono state implementate e i seguenti eventi: **Sprint Review** e **Backlog refinement**. Infine verranno elencati i possibili ritardi, i problemi riscontrati e le soluzioni trovate. Ogni sprint è stato identificato da un codice univoco e da un titolo. In questo progetto gli sprint individuati sono stati:

- **S1: Struttura dello stato e dell'architettura Redux**;
- **S2: Componenti grafici e container Redux**;
- **S3: Parser JSON e adapter**;
- **S4: Caricamento parziale**.

## 4.1 S1: Struttura dello stato e dell'architettura Redux

**Durata:** *5 giorni*
Nel primo sprint di sviluppo sono stati implementati i requisiti considerati più importanti per porre delle solide basi dell'applicazione web; in particolare la progettazione e codifica della struttura dello stato e l'architettura di Redux.

### 4.1.1 Sprint Backlog

In particolare sono stati implementati questi requisiti.

| **Id** | **Descrizione** | **Tipo** |
|---|---|---|
| R1.0 | Definizione dello stato dell'applicazione | O |
| R1.0.1 | Sviluppo TableState | O |
| R1.0.2 | Sviluppo NodeDimensions | O |
| R1.0.3 | Sviluppo BodyCells | O |

| | | |
|---|---|---|
| R1.0.4 | Sviluppo HeaderAction | O |
| R1.0.5 | Sviluppo NodeActionType | O |
| R1.1 | Definizione architettura Redux | O |
| R1.1.1 | Sviluppo TableStateSlice | O |
| R1.1.2 | Sviluppo di Thunk | O |
| R1.1.3 | Sviluppo delle Actions | O |
| R1.1.4 | Sviluppo dei Reducers | O |

### 4.1.2 Soluzioni implementate

Ho realizzato i seguenti `package`:

- `redux`
- `redux.slices`
- `redux.thunks`
- `redux.state`
- `entities`

#### 4.1.2.1 redux.slice

**Listing 4.1:** TableState

```
1  object TableStateSlice {
2          data class State(
3            val cols: ArrayList<ArrayList<DimensionsNode>> = ArrayList(),
4            val rows: ArrayList<ArrayList<DimensionsNode>> = ArrayList(),
5            val cells: ArrayList<ArrayList<BodyCells>> = ArrayList(),
6            val rowActions: ArrayList<ArrayList<HeaderAction>> = ArrayList(),
7            val colActions: ArrayList<ArrayList<HeaderAction>> = ArrayList(),
8            val isLoading: Boolean = false,
9          )
10
11         private val initTableState = InitState()
12         fun initTable() : RThunk = initTableState
13
14         class UpdateCells(val cells: ArrayList<ArrayList<BodyCells>>):
               RAction
15         class UpdateRows(val rows: ArrayList<ArrayList<DimensionsNode>>):
               RAction
16         class UpdateCols(val cols: ArrayList<ArrayList<DimensionsNode>>):
               RAction
17         class SetIsLoading(val b: Boolean): RAction
18         class UpdateRowActions(val n: ArrayList<ArrayList<HeaderAction>>):
               RAction
19         class UpdateColActions(val n: ArrayList<ArrayList<HeaderAction>>):
               RAction
20
21         fun reducer(state: State = State(), action: RAction) : State {
22                 return when (action) {
```

```
23                          is UpdateCells -> state.copy(cells = action.cells)
24                          is UpdateRows -> state.copy(rows = action.rows)
25                          is UpdateCols -> state.copy(cols = action.cols)
26                          is SetIsLoading -> state.copy(isLoading = action.b)
27                          is UpdateRowActions -> state.copy(rowActions = action
                                .n)
28                          is UpdateColActions -> state.copy(colActions = action
                                .n)
29                          else -> state
30                  }
31          }
32  }
```

#### 4.1.2.2 redux.thunks

La classe `InitState` si occuperà di riempire lo stato della tabella con il risultato della richiesta HTTP non ancora implementata.

**Listing 4.2:** TableState

```
1   class InitState : RThunk {
2           override fun invoke(dispatch: (RAction) -> WrapperAction, getState:
                () -> AppState): WrapperAction {
3                   val mainScope = MainScope()
4                   mainScope.launch {
5                           // val res : TableState = fetchCreavistaJson()
6                           dispatch(TableStateSlice.UpdateRows(res.rows))
7                           dispatch(TableStateSlice.UpdateCols(res.cols))
8                           dispatch(TableStateSlice.UpdateCells(res.cells))
9                           dispatch(TableStateSlice.UpdateRowActions(res.
                                rowAction))
10                          dispatch(TableStateSlice.UpdateColActions(res.
                                colAction))
11                  }
12                  return nullAction
13          }
14  }
```

#### 4.1.2.3 redux.state

Lo stato dell'applicazione è definito in modo che sia completamente modulare. Infatti la `data class AppState` contiene un'istanza dello stato di `TableStateSlice`. In questo modo in un futuro, se si vorrà espandere questo componente si potrà semplicemente aggiungere un nuovo "slice" e modificare questo file in modo da collegarlo a `AppState`.

**Listing 4.3:** TableState

```
1   data class AppState (
2     val tableState: TableStateSlice.State = TableStateSlice.State()
3   )
4
5   // funzioni per collegare AppState a TableStateSlice
6   fun rootReducer(): Reducer<AppState, RAction> = getRootReducers(
7     mapOf(
8       AppState::tableState to TableStateSlice::reducer
9     )
10  )
11
```

```
12 fun <S, A, R> getRootReducers(reducers: Map<KProperty1<S, R>, Reducer<*, A>>)
       : Reducer<S, A> {
13         return combineReducers(reducers.mapKeys { it.key.name })
14 }
```

#### 4.1.2.4 entities

**TableState**
L'entità `TableState` è la `data class` che contiene l'intero stato dell'applicazione. Al suo interno sono presenti tutte le informazioni necessarie per la corretta renderizzazione della tabella pivot.

**Listing 4.4:** TableState

```
1 data class TableState(
2   val cols: ArrayList<ArrayList<DimensionsNode>>,
3   val rows: ArrayList<ArrayList<DimensionsNode>>,
4   val cells: ArrayList<ArrayList<BodyCells>>,
5   val rowAction: ArrayList<ArrayList<HeaderAction>>,
6   val colAction: ArrayList<ArrayList<HeaderAction>>
7 )
```

**DimensionsNode**
L'entità `DimensionsNode` è la `data class` che contiene le informazioni riguardanti una cella d'intestazione della tabella pivot.

**Listing 4.5:** DimensionsNode

```
1 data class DimensionsNode(
2   var id: String,
3   var label: String,
4   var level: Int? = null,
5   var childDepth: Int? = null,
6   var path: List<String>? = null,
7   var actionType: NodeActionType = NodeActionType.NULL,
8   var isChild: Boolean = false
9 )
```

**HeaderAction**
L'entità `HeaderAction` è la `data class` che contiene le informazioni riguardanti i pulsanti che si occupano di aprire intere colonne o righe di dimensioni nella TableActionUI.

**Listing 4.6:** HeaderAction

```
1 data class HeaderAction(
2   val actionType: NodeActionType,
3   val dim: Int,
4   val depth: Int,
5 )
```

**NodeActionType**
L'entità `HeaderAction` è una `enum class` che definisce il tipo di azione che può essere eseguita da una cella.

**Listing 4.7:** NodeActionType

```
enum class NodeActionType(val type: String) {
  EXPAND("E"),
  COLLAPSE("C"),
  NULL("")
}
```

**BodyCells**
L'entità `BodyCells` è la `data class` che contiene le informazioni riguardanti le celle che contengono i dati della tabella.

**Listing 4.8:** BodyCells

```
data class BodyCells(
  val value: Int,
  val cPath: List<String>,
  val rPath: List<String>
)
```

### 4.1.3 Struttura del progetto

```
kotlin
├── redux
│   ├── slices
│   │   └── TableStateSlice.kt
│   ├── thunks
│   │   ├── RThunk.kt
│   │   └── InitState.kt
│   └── state
│       └── Index.kt
└── entities
    └── StateEntities.kt
```

### 4.1.4 Sprint Review

Niente da segnalare.

### 4.1.5 Backlog refinement

Il Product Backlog non è stato modificato.

### 4.1.6 Problemi riscontrati

Implementare correttamente l'architettura Redux è risultato difficoltoso a causa della mia inesperienza con la libreria e la mancanza di documentazione relativa a `kotlin-redux`. Questi problemi non hanno però causato rallentamenti nell'implementazione degli altri requisiti dello Sprint Backlog.

## 4.2 S2: Componenti grafici e container Redux

**Durata:** *4 giorni*
Nel secondo sprint di sviluppo ho lavorato sull'interfaccia grafica e quindi i componenti React e i relativi container Redux.

### 4.2.1 Sprint Backlog

| | | |
|---|---|---|
| R2.0 | **Definizione dei componenti** | O |
| R2.1 | Sviluppo dei componenti React | O |
| R2.1.1 | Sviluppo di TableView | O |
| R2.1.2 | Sviluppo di TableHeaderView | O |
| R2.1.3 | Sviluppo di TableSidebarView | O |
| R2.1.4 | Sviluppo di TableBodyView | O |
| R2.2 | Sviluppo dei container Redux | O |
| R2.2.2 | Sviluppo di TableController | O |
| R2.2.3 | Sviluppo di TableHeaderController | O |
| R2.2.4 | Sviluppo di TableSidebarController | O |
| R2.2.5 | Sviluppo di TableBodyController | O |

### 4.2.2 Soluzioni implementate

#### 4.2.2.1 view

#### 4.2.2.2 redux.container

### 4.2.3 Struttura del progetto

```
kotlin
├── redux
│   ├── slices
│   │   └── TableStateSlice.kt
│   ├── thunks
│   │   ├── RThunk.kt
│   │   └── InitState.kt
│   ├── state
│   │   └── Index.kt
│   └── container
│       ├── TableViewController.kt
│       ├── TableHeaderViewController.kt
│       ├── TableSidebarViewController.kt
│       ├── TableBodyViewController.kt
│       ├── TableActionUIController.kt
│       └── TableLabelViewController.kt
└── entities
```

```
    ├── StateEntities.kt
└── view
    ├── TableView.kt
    ├── TableHeaderView.kt
    ├── TableSidebarView.kt
    ├── TableBodyView.kt
    ├── TableActionUI.kt
    └── TableLabelView.kt
```

### 4.2.4 Sprint Review

Niente da segnalare.

### 4.2.5 Backlog refinement

Il Product Backlog non è stato modificato.

### 4.2.6 Problemi riscontrati

Niente da segnalare.

## 4.3 S3: Parser JSON e adapter

### 4.3.1 Sprint Backlog

### 4.3.2 Soluzioni implementate

#### 4.3.2.1 Gestione delle richieste all'API

Il primo passo per utilizzare dei dati reali all'interno della tabella pivot è stato quello di realizzare una funzione per effettuare richieste HTTP alla API di Gruppo4. Per farlo ho utilizzato l'implementazione in Kotlin di `window.fetch`. La funzione risultante è la seguente:

**Listing 4.9:** Funzione fetch()

```
suspend fun fetch() {
val res = window.fetch(url, RequestInit(
method = "GET",
credentials = RequestCredentials.Companion.INCLUDE,
headers = {
json("Accept" to "application/json")
json("Content-Type" to "application/json")
}))
.await()
.json()
.await()

// Codice identificativo dell'istanza della tabella pivot necessario
// per le chiamate successive
INSTANCE_KEY = (res as kotlin.js.Json)["InstanceKey"] as String?
}
```

La seguente funzione effettua una richiesta HTTP GET alla API di Gruppo4 la quale ritorna il JSON risultante e una chiave necessaria per effettuare le chiamate successive per la corretta tabella pivot. Per le richieste in seguito ad un'azione da parte di un utente ho utilizzato la seguente funzione `sendAction()` che in modo similare alla precedente esegue una richiesta HTTP POST all'API della tabella di un JSON che contiene le seguenti informazioni:

- `axis`: può essere "R" o "C" (righe o colonne);
- `path`: array identificativo della cella su cui è stata effettuata l'azione.

L'implementazione di `sendAction()` è la seguente:

**Listing 4.10:** Funzione sendAction()

```
suspend fun sendAction(body: String, type: String) {
val res: Any? = window.fetch("$url$INSTANCE_KEY/$type", RequestInit(
method = "POST",
body = body,
credentials = RequestCredentials.Companion.INCLUDE,
headers = {
json("Accept" to "application/json")
json("Content-Type" to "application/json")
}))
.await()
.json()
.await()
}
```

### 4.3.3 Sprint Review

Niente da segnalare.

### 4.3.4 Backlog refinement

Il Product Backlog non è stato modificato.

### 4.3.5 Problemi riscontrati

La realizzazione dell'adapter è stata rallentata da alcuni cambiamenti della struttura dell'API da parte dell'azienda. I rallentamenti hanno causato ad una lunghezza dello sprint più elevata. La realizzazione dell'adapter ha infatti anche occupato parte della settimana riservata al quarto sprint.

## 4.4 S4: Caricamento parziale

### 4.4.1 Sprint Review

Niente da segnalare.

### 4.4.2 Backlog refinement

Il Product Backlog non è stato modificato.

### 4.4.3 Sprint Review

### 4.4.4 Backlog refinement

### 4.4.5 Problemi riscontrati

# Capitolo 5

# Analisi dei requisiti

## 5.1 Definizione delle User Stories

Per prima cosa il team di sviluppo si è occupato di realizzare le User Stories definite mediante la seguente struttura.

| **Id** | **Descrizione** | **Priorità** | **Implementato** |
|---|---|---|---|
| US1.1 | Descrizione dell'user story | A | SI |

**Tabella 5.1:** Esempio tabella User Story

Per ogni descrizione di un user story si possono identificare le seguenti informazioni:

- **Ruolo**: definisce il tipo di utente;
- **Obiettivo**: definisce di che cosa ha bisogno l'utente;
- **Beneficio**: definisce i vantaggi che porta all'utente.

La sinteticità e la facilità nel definire le user story porta a vantaggi nella comunicazione tra il team di sviluppo e il cliente, rende più semplice l'aggiornamento dei requisiti e i costi di scrittura e manutenzione delle user stories sono molto bassi.

## 5.2 Definizione del Product Backlog

Uno dei primi obiettivi del progetto posti dal team di sviluppo è stato quello di definire il Product Backlog, cioè i requisiti del prodotto. Per ognuna delle user story precedentemente scritta è stata assegnata una priorità seguendo la seguente legenda:

| | | |
|---|---|---|
| **A** | *Priorità alta* | funzionalità necessarie per il corretto funzionamento dell'applicazione |
| **M** | *Priorità media* | funzionalità che migliorano il prodotto |
| **B** | *Priorità bassa* | funzionalità non necessarie per il corretto funzionamento dell'applicazione |

**Tabella 5.2:** Tabella priorità User Story

Questo ci ha permesso di categorizzare le funzionalità principali del componente d'interfaccia grafico da quelle opzionali. Abbiamo quindi popolato il Product Backlog per ordine di *priorità*, in questo modo la suddivisione delle user story per sprint, mediante le riunioni di *Sprint Planning*, è stata chiara e veloce.
Infatti abbiamo concentrato le funzionalità principali da implementare nei primi due sprint così da avere, già a partire dal terzo sprint, un prodotto con le funzionalità principali già implementate. Dato che ogni sprint di sviluppo prevede una riunione di *backlog refinement* verrano elencate tutte le iterazioni del Product Backlog.

## 5.3 Product Backlog

| **Id** | **Descrizione** | **Priorità** | **Implementato** |
|---|---|---|---|
| US1 | Come **utente** voglio poter visualizzare i miei dati e le dimensioni relative ai dati | A | SI |
| US2 | Come **utente** voglio avere una interfaccia non ostruttiva | M | SI |
| US3 | Come **utente** voglio poter utilizzare questa applicazione web dal mio PC | A | SI |
| US4 | Come **utente** voglio poter utilizzare questa applicazione web dal mio tablet | A | SI |
| US5 | Come **utente** voglio poter utilizzare questa applicazione web dal mio telefono | A | SI |
| US6 | Come **utente** voglio avere un caricamento veloce | M | SI |
| US7 | Come **utente** voglio poter esplorare liberamente i dati | A | SI |

## 5.4 Requisiti individuati dal Product Backlog

| Id | Descrizione | Tipo | Impl. | User Story |
|---|---|---|---|---|
| R1.0 | **Definizione dello stato dell'applicazione** | O | SI | US1.1 |
| R1.0.1 | Sviluppo TableState | O | SI | US1.1 |
| R1.0.2 | Sviluppo NodeDimensions | O | SI | US1.1 |
| R1.0.3 | Sviluppo BodyCells | O | SI | US1.1 |
| R1.0.4 | Sviluppo HeaderAction | O | SI | US1.1 |
| R1.0.5 | Sviluppo NodeActionType | O | SI | US1.1 |
| R1.1 | Sviluppo architettura Redux | O | SI | - |
| R1.1.1 | Sviluppo TableStateSlice | O | SI | - |
| R1.1.2 | Sviluppo di Thunk | O | SI | - |
| R1.1.3 | Sviluppo delle Actions | O | SI | - |
| R1.1.4 | Sviluppo dei Reducers | O | SI | - |
| R2.0 | **Definizione e pianificazione dei componenti** | O | SI | US1.1 |
| R2.1 | Sviluppo dei componenti React | O | SI | US1, US2, US3, US4, US5 |
| R2.1.1 | Sviluppo di TableView | O | SI | - |
| R2.1.2 | Sviluppo di TableHeaderView | O | SI | - |
| R2.1.3 | Sviluppo di TableSidebarView | O | SI | - |
| R2.1.4 | Sviluppo di TableBodyView | O | SI | - |
| R2.2 | Sviluppo dei container Redux | O | SI | - |
| R2.2.2 | Sviluppo di TableController | O | SI | - |
| R2.2.3 | Sviluppo di TableHeaderController | O | SI | - |
| R2.2.4 | Sviluppo di TableSidebarController | O | SI | - |
| R2.2.5 | Sviluppo di TableBodyController | O | SI | - |
| R3.0 | **Sviluppo parser per JSON dell'API** | O | SI | - |
| R3.1 | Sviluppo data class @Serializable | O | SI | - |
| R3.2 | Sviluppo adapter da JSON a TableState | O | SI | - |

| | | | | |
|---|---|---|---|---|
| R3.2.1 | Sviluppo funzioni XXX | O | SI | - |
| R3.2.2 | Sviluppo funzioni XXX | O | SI | - |
| R3.2.3 | Sviluppo funzioni XXX | O | SI | - |
| R3.2.4 | Sviluppo funzioni XXX | O | SI | - |
| R3.2.5 | Sviluppo funzioni XXX | O | SI | - |
| R3.4 | Sviluppo funzioni per effettuare richieste HTTP | O | SI | - |

# Capitolo 6

# Tecnologie

## 6.1 Kotlin

Kotlin è un linguaggio tipizzato, realizzato da JetBrains, utilizzato da molti sviluppatori per il fatto che il codice è conciso, sicuro e permette di lavorare utilizzando librerie per la JVM, Android e il browser.

## 6.2 React

Libreria utilizzata per la realizzazione dei componenti grafici dell'applicazione.

## 6.3 Kotlin Wrappers

Durante la codifica del progetto sono state utilizzati alcuni wrapper forniti da JetBrains per lo sviluppo web di React, Redux e React-Redux.

### 6.3.1 kotlin-react

Wrapper utilizzato per l'utilizzo di funzioni che permettono la codifica di componenti React.

### 6.3.2 kotlin-redux

Wrapper utilizzato per la realizzazione dell'architettura Redux utilizzata nell'applicazione per la gestione dello stato.

### 6.3.3 kotlin-react-redux

Wrapper utilizzato per la realizzazione del collegamento tra i componenti React e lo stato di Redux.

## 6.4 Redux

Libreria utilizzata per la realizzazione dello stato dell'applicazione e della sua gestione

# Capitolo 7

# Conclusioni

## 7.1 Raggiungimento degli obiettivi

Gli obiettivi del progetto sono stati raggiunti quasi completamente. Tutti i requisiti richiesti dall'azienda sono stati soddisfatti tranne il caricamento continuo allo scroll di un utente dato i problemi riscontrati durante la codifica dello sprint numero 3. Il componente finale è il seguente. Tutte le funzionalità previste identificate nella progettazione iniziale del progetto sono state state implementate.

## 7.2 Conoscenze acquisite

In questa sezione verranno identificate le conoscenze per ogni metodologia e tecnologia utilizzate nell'ambito del progetto.

### 7.2.1 Metodologia agile

La metodologia agile simile a SCRUM utilizzata dal team di sviluppo mi ha permesso di identificare dal punto di vista pratico le componenti necessarie a lavorare in un team di sviluppo che pratica una metodologia agile.

### 7.2.2 Kotlin

Durante lo studio iniziale e la codifica del componente d'interfaccia grafica mi ha permesso di arricchire la mia conoscenza del linguaggio Kotlin. Considero questa conoscenza acquisita molto importante dato che Kotlin è utilizzato da molte aziende famose tra cui: ... . Inoltre in quanto è un linguaggio che permette di realizzare molti prodotti (api, web app, applicazioni android native, etc..) penso che questo stage mi abbia fornito le basi per espandere le mie abilità su altri campi oltre che naturalmente lo sviluppo di applicazioni web.

### 7.2.3 React e Redux

La libreria React è utilizzata molto per la realizzazione di applicazioni web da moltissime aziende. In passato avevo già lavorato con React in progetti personali tuttavia, mediante questo progetto, penso di aver ampliato e soprattutto affinato le mie conoscenze di questa libreria. Lo stesso vale per la libreria Redux, infatti grazie allo studio e alla

codifica di componenti React-Redux considero di aver migliorato e ampliato le mie conoscenze nello sviluppo web di applicazioni scalabili e manutenibili.

### 7.2.4 Programmazione funzionale

Lo studio delle first class functions e del loro utilizzo mi hanno permesso di entrare nel mondo della programmazione funzionale. Insieme a Kotlin e alla metodologia agile considero che queste nozioni di programmazione funzionale mi permetterranno in futuro di velocizzare la codifica e la progettazione di nuove applicazioni.

### 7.2.5 Lavorare in un team di sviluppo

Lavorare in un team di sviluppo mi ha fatto capire quanto importante sia la comunicazione all'interno di un progetto. Ringrazio Tobia Conforto per aver imposto un livello di comunicazione molto alto.

### 7.2.6 Lavorare da remoto

Dato le circostanze sociali mi sembra una skill molto importante saper lavorare da remoto.

## 7.3 Valutazione personale

### 7.3.1 Effettività delle metodologie agili

Consider l'utilizzo di una metodologia agile in un progetto software molto utile dato che il suo utilizzo permette di risolvere problemi in modo efficiente. Inoltre grazie alle sue riunioni favorisce un ambiente ricco di comunicazione che considero molto importante. La suddivisione in sprint di sviluppo, se pianificato correttamente, aiuta a mantenere il lavoro concentrato su un numero ristretto di funzionalità di un progetto e questo può aiutare per suddividere il lavoro in piccoli incrementi. Il vantaggio che ho notato maggiormente durante la codifica in sprint è stato il fatto che, dato che la struttura della metodologia agile poneva come obiettivo la realizzazione delle funzionalità più importanti nei primi sprint, questo permette di avere una visione completa di un'applicazione già dai primi sprint.

### 7.3.2 Effettività di Kotlin per la realizzazione di UI per il web

Lo sviluppo in Kotlin di interfacce utente per il web è una soluzione valida ma secondo me è una tecnologia che deve maturare ancora per quanto riguarda lo sviluppo di applicazioni web.

### 7.3.3 Effettività di React e Redux

Molto utili, specialmente Redux che aiuta a gestire lo stato dell'appplicazione in modo molto metodico e separato dall'interfaccia grafica. Questo permette di realizzare applicazioni scalabili, un fattore molto importante specialmente nello sviluppo di applicazioni web.

### 7.3.4 Effettività della programmazione funzionale

L'utilizzo delle higher order function ha secondo moltissimi vantaggi per quanto riguarda la codifica di una applicazione. I vantaggi che ho identificato sono stati: aumento della velocità della codifica, aumento della leggibilità del codice durante la verifica (a causa della natura dichiarativa) e una scrittura più efficiente e concisa delle funzioni.

# Appendice A

# Appendice A

Citazione

Autore della citazione

# Glossario

**API** in informatica con il termine *Application Programming Interface API* (ing. interfaccia di programmazione di un'applicazione) si indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello semplificando così il lavoro di programmazione. 39

# Acronimi

**API** Application Program Interface. 1

# Bibliografia

## Riferimenti bibliografici

James P. Womack, Daniel T. Jones. *Lean Thinking, Second Editon*. Simon & Schuster, Inc., 2010.

## Siti web consultati

*Funzioni di ordine superiore*. URL: https://kotlinlang.org/docs/reference/lambdas.html.

*Manifesto Agile*. URL: http://agilemanifesto.org/iso/it/.

# Ringraziamenti

*Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Claudio Enrico Palazzi, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.*

*Desidero ringraziare con la mia famiglia per essermi stata vicina in ogni momento durante gli anni di studio.*

*Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.*

*Padova, Dicembre 2020* Marco Rampazzo