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 del Dott. 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 già 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 riguardo la sua struttura e le sue funzionalità. Nelle prossime sottosezioni descriverò gli elementi che la compongono e le possibili interazioni con l'utente. Durante la discussione dei requisiti del componente, l'azienda mi ha fornito un accesso ad una istanza della loro applicazione correntemente in uso. Mi sono stati forniti dei dati da usare nello sviluppo del componente ed un accesso all'interfaccia di una tabella esistente per capire al meglio

il suo funzionamento.

		Tutti le tipologie	DIRETTA	INDIRETTA	N.D.
		Tutte le età	Tutte le età	Tutte le età	Tutte le età
Tutte le associazioni	Tutti i generi	286 165	10 743	275 422	0
CASARTIGIANI	Tutti i generi	1 788	0	1 788	0
PD	Tutti i generi	19	0	19	0
RO	Tutti i generi	21	0	21	0
TV	Tutti i generi	1 253	0	1 253	0
VE	Tutti i generi	9	0	9	0
VI	Tutti i generi	422	0	422	0
VR	Tutti i generi	64	0	64	0
CGIL	Tutti i generi	46 565	0	46 565	0
CISL	Tutti i generi	43 101	0	43 101	0
CNA	Tutti i generi	2 786	0	2 786	0
CONFARTIGIANATO	Tutti i generi	22 914	0	22 914	0
UIL	Tutti i generi	9 490	0	9 490	0
N.D.	Tutti i generi	159 521	10 743	148 778	0

1.3.1 Struttura

La tabella pivot si può suddividere in tre parti ben distinte: le celle contenente i dati, le celle d'intestazione delle righe e quelle delle colonne. Le celle d'intestazione sono definite come le *dimensioni di analisi* della tabella. Esse rappresentano tutte le variabili riguardanti i dati contenuti nella tabella. Le *dimensioni di analisi* della tabella pivot possono essere molteplici sia per le righe che per le colonne. Ad esempio, nello screenshot precedente, le *dimensioni di analisi* delle righe sono l'insieme delle due variabili: Associazioni e Genere. Quindi ogni cella dei dati corrisponde ad una intersezione tra le *dimensioni di analisi* delle righe e delle colonne.

1.3.2 Funzionalità

La tabella deve essere completamente esplorabile da un utente, per farlo bisogna identificare le azioni che possono essere eseguite:

- aprire e chiudere tutti i figli di una dimensione;
- aprire e chiudere un singolo figlio di una dimensione.

Mediante queste azioni i big data possono essere esplorati ed analizzati dall'utente.

1.4 Organizzazione del testo

Riguardo la stesura di questo 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 fornita una descrizione dei metodi e dei processi messi in atto durante il tirocinio e in particolare riguardo: la metodologia agile, la programmazione funzionale, 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. Infatti applicarlo nella sua interezza 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 sezione 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.

I documenti 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'ultimo rilascio del software.

La metodologia agile applicata è caratterizzata inoltre dai seguenti eventi:

- **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* (l'insieme delle modifiche corrisponde ad un *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 da una composizione di funzioni dichiarative rispetto ad una gerarchia di classi con una sequenza di operazioni imperative. In questo progetto si è cercato di applicare questo paradigma sotto due aspetti. Il primo corrisponde all'architettura dell'applicazione, infatti ho evitato di utilizzare gerarchie di classi e ho prediletto un insieme di funzioni dichiarative.
Il secondo invece riguarda le Funzioni di ordine superiore / Higher order functions fornite da Kotlin; sono funzioni molto utili perchè permettono di scrivere un codice più leggibile, conciso e hanno la caratteristica di evitare Side effects. 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 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 distribuito che permette di tenere traccia delle modifiche in un prodotto software e di organizzarne la codifica.

Feature-branch

In questo tirocinio è stata usata la tecnica del *feature-branch*: ogni aggiunta di una funzionalità corrisponde all'apertura di un nuovo branch che deve essere poi approvato, previa verifica, prima di essere unito al branch `master` mediante una *pull request*.

2.3.2 Gitlab

Gitlab è uno strumento web che permette di implementare un DevOps lifecycle. In particolare fornisce un'interfaccia per gestire una repository git, un ITS 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 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 un'interfaccia grafica per eseguire task di Gradle.

2.4.2 Gradle

Gradle è uno strumento di Build automation 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 modo per utilizzare funzioni e oggetti tipici di 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 JavaScript 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; questo risulta molto utile quando si codifica un'interfaccia web dato che i cambiamenti vengono visualizzati automaticamente.

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.
Per quanto riguarda le riunioni telematiche è stato usato *Google meet*.

Capitolo 3

Progettazione

In questo capitolo verrà descritta la progettazione dell'intero prodotto, in particolare verranno chiariti i seguenti argomenti:

- progettazione delle classi che definiscono lo stato dell'applicazione;
- gestione dello stato dell'applicazione;
- progettazione del parser Json e dell'*adapter* per trasformare le informazioni ottenute dall'api nelle classi che definiscono lo stato dell'applicazione;
- gestione del caricamento parziale dei dati della tabella;
- descrizione e progettazione dei componenti grafici.

3.1 Entità dello stato

Per quanto riguarda lo stato dell'applicazione ho analizzato la struttura e le funzionalità della tabella fornita dall'azienda. Durante la progettazione della struttura dello stato ho dato priorità alla semplicità 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 una azione sulle dimensioni, 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 che può essere:

- `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.

3.2 Gestione dello stato

In questa sezione verrà descritto il ragionamento e il modo in cui è avvenuta la progettazione degli elementi che si occuperanno di gestire lo stato dell'applicazione.

3.2.1 Dataflow dell'applicazione

Un aspetto che è stato molto discusso dal team di sviluppo riguarda il dataflow 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 alle possibili soluzioni per garantirne la scalabilità. Nelle prossime sezioni verranno descritti i *dataflow* forniti da React e Redux, infine verrà identificato quello adottato nell'ambito del progetto.

React

React è una libreria per realizzare interfacce utente che utilizza un *dataflow* unidirezionale, questo significa che il flusso dei dati deve passare mediante dei componenti grafici. I dati possono essere contenuti nello stato locale di un componente che è accessibile solo dal componente stesso. Le informazioni possono invece essere passate tra un componente padre ai suoi componenti figli mediante le Props.
Questo *dataflow* è molto semplice da utilizzare però presenta alcune limitazioni per quanto riguarda la scalabilità. Per avere uno stato unico di tutta l'applicazione bisognerebbe dare ad un componente grafico la responsabilità di mantenerlo nel suo stato locale. L'architettura che ne deriva, nel caso di applicazioni complesse, è difficile da manutenere e poco scalabile.
Tuttavia questo semplice *dataflow* ha il vantaggio che per un numero ristretto di componenti i cambiamenti di stato locale e i passaggi di informazioni mediante le *props* sono molto veloci e semplici.

Redux

Per garantire scalabilità nella gestione dello stato dell'applicazione è stato deciso di utilizzare la libreria Redux. Essa offre un *dataflow* unidirezionale dove lo stato è gestito all'intero di una una struttura di Redux chiamata *store*. Questa struttura è esterna ai componenti grafici, quindi la gestione dello stato dell'applicazione diventa 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 che creano un nuovo stato per evitare *side effects* .

Gli elementi dell'architettura di Redux sono i seguenti:

- **Actions**: oggetti che rappresentano un'azione che innesca un cambiamento dello stato;
- **Reducers**: *funzioni pure* che accettano come parametro un *actions* e si occupano di modificare lo stato generandone una copia modificata;
- **Store**: lo *store* è un'interfaccia che contiene lo stato dell'applicazione e fornisce funzioni per leggere, modificare e registrare $listeners_G$ allo stato.

3.2.2 Soluzione

Per lo sviluppo di questo componente oltre all'architettura Redux descritta nella sezione precedente è stato utilizzato anche il *dataflow* di React per gestire gli aggiornamenti visivi. Il *dataflow* finale del prodotto è quindi il seguente:

1. se l'utente esegue un'azione che implica un cambiamento dello stato verrà mandata allo *store* un *actions*;
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.

Per applicare al meglio il *dataflow* precedentemente descritto ho progettato i componenti di Redux nel seguente modo. Per prima cosa ho suddiviso lo stato dell'applicazione in *slice* cioè in sezioni di stato. In questo modo la sua struttura è modulare e scalabile dato che, se questa applicazione verrà ampliata basterà aggiungere uno *slice* allo stato. Ogni slice deve essere definito 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 { ... }
}
```

Stato

In ogni *slice* deve essere definito un `data class` che contiene tutti i campi dati che definiscono lo stato. Per quanto riguarda la realizzazione della tabella pivot ho individuato la necessità di uno *slice* con i seguenti campi dati:

- `isLoading`: valore 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;
- `rowTree`: struttura ad albero che definisce la relazione tra le dimensioni delle righe;
- `colTree`: struttura ad albero che definisce la relazione tra le dimensioni delle colonne;
- `rowActions`: insieme di possibili azioni eseguibili sulle dimensioni delle righe;
- `colActions`: insieme di possibili azioni eseguibili sulle dimensioni delle colonne.

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 SetIsLoading(): RAction`;

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

Thunk

Nell'ambito di questo progetto i cambiamenti dello stato devono essere gestiti in modo asincrono, in quanto viene fornita una *API* di supporto. Dato che i cambiamenti allo stato innescati dalle *actions* possono essere solo sincroni ho utilizzato un *middleware* di Redux 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 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 ne implementa le sue funzionalità più significative. In particolare l'esecuzione asincrona degli update dello stato.

Listing 3.2: Interfaccia Thunk

```
1  interface RThunk : RAction {
2          operator fun invoke(
3          dispatch: (RAction) -> WrapperAction,
4          getState: () -> AppState
5          ) : WrapperAction
6  }
7
8  fun rThunk() =
9  applyMiddleware<AppState, RAction, WrapperAction, RAction, WrapperAction>(
10 { store ->
11         { next ->
12                 { action ->
13                         if (action is RThunk)
14                           action(store::dispatch, store::getState)
15                         else
16                           next(action)
17                 }
18         }
19 }
20 )
21
22 val nullAction = js {}.unsafeCast<WrapperAction>()
```

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 utilizzare 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 SetIsLoading -> state.copy(...)
    is UpdateRows -> state.copy(...)
    is UpdateCols -> state.copy(...)
    is UpdateRowTree -> state.copy(...)
    is UpdateColTree -> state.copy(...)
    is UpdateCells -> state.copy(...)
    is UpdateRowActions -> state.copy(...)
    is UpdateColActions -> state.copy(...)
    else -> state
  }
}
```

3.3 Parser e adapter

Per quanto riguarda la progettazione del parser Json per l'api ho per prima cosa definito i passaggi necessari per ottenere i dati dall'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 ad un'azione effettuata da un utente. Per realizzare il parser 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"],
```

```
33              "Type": "list",
34              "Name": "associazioni_provincia",
35              "Label": "Associazioni per provincia"
36            }
37          ]
38  }
```

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 analizzato le data class ottenibili dal Json e le `data class` definite nello stato. Da questa analisi 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`.

3.4 Gestione del caricamento parziale

L'azienda ha identificato due requisiti riguardanti il caricamento parziale dei dati della tabella in modo da non sovraccaricare l'*API* con file Json troppo grandi. In particolare mi ha richiesto di implementarlo sotto due aspetti. Il primo riguarda il caricamento parziale di informazioni quando si "apre" una cella d'intestazione delle dimensioni. Il secondo invece riguarda il caricamento parziale durante lo scroll di un utente.

Caricamento parziale di un nodo

Le dimensioni di analisi della tabella pivot vengono fornite dall'*API* sotto forma di struttura ad albero, da cui verrà generata una matrice di oggetti in modo da semplificare l'iterazione di essa all'interno della funzione `render()` dei componenti React. La struttura ad albero fornisce però dei vantaggi per quanto riguarda il caricamento parziale di un nodo infatti basterà modificare un nodo dell'albero e poi generare la matrice di oggetti. Per farlo ho pensato di mantenere nello stato una copia della struttura ad albero, in questo modo quando viene aperto o chiuso una cella di una dimensione della tabella basterà semplicemente sostituire il nodo nella copia della struttura ad albero dello stato e generare nuovamente la matrice di oggetti.

Caricamento parziale scroll

L'azienda come obiettivo voleva ottenere una tabella veloce da esplorare anche se contenente molte righe. Per farlo è stato pensato di eseguire un caricamento parziale delle righe e colonne di una tabella. In questo modo quando l'utente raggiunge la fine della tabella, un componente React si occuperà di notificare l'*API* in modo che lo stato riceva una nuova sezione della tabella.

3.5 Componenti grafici

Per quanto riguarda la progettazione dei componenti grafici l'azienda mi ha dato completa libertà sulla definizione della 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`

In seguito ho suddiviso 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

Analisi dei requisiti

4.1 Definizione delle User Stories

Per definire in modo semplice i requisiti del progetto sono state realizzate le *user stories* che sono state definite mediante la seguente struttura.

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

Tabella 4.1: Esempio tabella User Story

Per ogni descrizione di una *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.

4.2 Definizione del Product Backlog

Uno dei primi obiettivi del progetto 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 4.2: Tabella priorità User Story

Questo mi ha permesso di categorizzare le funzionalità principali da quelle opzionali. Ho quindi popolato il *Product Backlog* per ordine di priorità. In questo modo la suddivisione dei requisiti per *sprint*, mediante le riunioni di *Sprint Planning*, è stata immediata.
Le funzionalità principali sono state pianificate per i primi due sprint così da avere, già a partire dal terzo sprint, un prodotto funzionante.

4.3 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

4.4 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	-
R3.1.1	Sviluppo Gruppo4JSON	O	SI	-
R3.1.2	Sviluppo Gruppo4Data	O	SI	-

R3.1.3	Sviluppo Gruppo4Filter	O	SI	-
R3.1.4	Sviluppo Gruppo4Actions	O	SI	-
R3.1.5	Sviluppo Gruppo4Node	O	SI	-
R3.2	**Sviluppo adapter da JSON a TableState**	O	SI	-
R3.2.1	Sviluppo funzione convertListOfGruppo4Node()	O	SI	-
R3.2.2	Sviluppo funzione convertTree()	O	SI	-
R3.2.3	Sviluppo funzione convertListOfGruppo4Actions()	O	SI	-
R3.2.4	Sviluppo funzione convertCells()	O	SI	-
R4.0	**Sviluppo funzioni per effettuare richieste HTTP**	O	SI	-
R4.1	Sviluppo funzione fetch()	O	SI	-
R4.2	Sviluppo funzione sendAction()	O	SI	-
R5.0	**Sviluppo caricamento parziale**	O	SI	-
R5.1	Sviluppo componente infinteScroller	O	SI	-
R5.2	Sviluppo funzioni per aggiornare struttura ad albero	O	SI	-

Capitolo 5

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 i seguenti:

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

5.1 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; in particolare la progettazione e codifica della struttura dello stato e l'architettura di Redux.

5.1.1 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

5.1.2 Soluzioni implementate

Ho realizzato i seguenti `package`:

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

5.1.2.1 redux.slice

Listing 5.1: 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  }
```

5.1.2.2 redux.thunks

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

Listing 5.2: 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  }
```

5.1.2.3 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 5.3: 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 }
```

5.1.2.4 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 5.4: 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 5.5: 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 5.6: 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 (apertura, chiusura o nessuna azione).

Listing 5.7: 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 5.8: BodyCells

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

5.1.3 Sprint Review

Niente da segnalare.

5.1.4 Backlog refinement

Il Product Backlog non è stato modificato.

5.1.5 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`. In particolare definire correttamente lo "Slice" dello stato ha richiesto molte ore di studio e ricerca dei wrapper di Kotlin per *Redux* Questi problemi non hanno però causato rallentamenti nell'implementazione degli altri requisiti dello Sprint Backlog.

5.2 S2: Componenti grafici e container Redux

Durata: *5 giorni*
Nel secondo sprint di sviluppo ho lavorato sull'interfaccia grafica e quindi i componenti React e i relativi container Redux.

5.2.1 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
R2.1.5	Sviluppo di TableActionUI	O
R2.1.6	Sviluppo di TableLabelView	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
R2.2.6	Sviluppo di TableLabelController	O
R2.2.7	Sviluppo di TableActionUIController	O

5.2.2 Soluzioni implementate

Le componenti grafiche realizzate sono le seguenti:

		Tutti le tipologie	DIRETTA	INDIRETTA	N.D.
		Tutte le età	Tutte le età	Tutte le età	Tutte le età
Tutte le associazioni	Tutti i generi	286 165	10 743	275 422	0
CASARTIGIANI	Tutti i generi	1 788	0	1 788	0
PD	Tutti i generi	19	0	19	0
RO	Tutti i generi	21	0	21	0
TV	Tutti i generi	1 253	0	1 253	0
VE	Tutti i generi	9	0	9	0
VI	Tutti i generi	422	0	422	0
VR	Tutti i generi	64	0	64	0
CGIL	Tutti i generi	46 565	0	46 565	0

Per rendere l'interfaccia utente chiara e utilizzabile da un dispositivo mobile ho implementato la funzionalità di nascondere le dimensioni d'analisi della tabella per ampliare lo spazio relativo ai dati in caso ce ne fosse bisogno. Inoltre ogni cella di intestazione può essere cliccata per attivare l'azione, questo perchè cliccare un pulsante molto piccolo da un dispositivo mobile è risultato difficile nei test manuali effettuati.

5.2.2.1 redux.container

Per ogni componente che richiedeva un collegamento con lo stato di Redux ho realizzato un wrapper che permette di mappare lo stato di Redux alle *props* di un componente React. I wrapper sono stati definiti nel seguente modo:

Listing 5.9: BodyCells

```
1  // campi dato dello stato da mappare alle props
2  private interface StateProps : RProps {
3          var rows: ArrayList<DimensionsNode>
4          ...
5  }
6
7  // funzioni per modificare lo stato da mappare alle props
8  private interface DispatchProps: RProps {
9          var updateRows: (rows: ArrayList<DimensionsNode>) -> Unit
10         ...
11 }
12
13 val reactComponent: RClass<RComponentProps> =
14 rConnect<AppState, RAction, WrapperAction, RProps, StateProps, DispatchProps,
       RComponentProps>(
15 mapStateToProps = { state, _ ->
16         rows = state.tableState.rows
17 },
18 mapDispatchToProps = { dispatch, _ ->
19         updateRows = { rows -> dispatch(TableStateSlice.UpdateRows(rows)) }
20 }
21 )(TableActionUI::class.js.unsafeCast<RClass<RComponentProps>>())
```

5.2.3 Sprint Review

Niente da segnalare.

5.2.4 Backlog refinement

Il Product Backlog non è stato modificato.

5.2.5 Problemi riscontrati

All'inizio dello sprint ho realizzato i componenti grafici con una scrittura funzionale, tuttavia dopo aver implementato i container Redux ho notato la presenza di alcuni limitazioni nell'uso di Redux e componenti funzionale di React. In particolare le funzioni dei wrapper *Redux* di Kotlin non permettono di eseguire la connessione tra i componenti funzionali React e lo stato di Redux. Per risolvere questo problema ho quindi riscritto i componenti grafici con una scrittura a classi. Questo problema non ha causato ritardi gravi.

5.3 S3: Parser JSON e adapter

Durata: *8 giorni*
Nel terzo sprint ho realizzato le funzioni per eseguire richieste HTTP e le ho implementate in modo da funzionare correttamente con le funzioni eseguite nel primo sprint. In seguito ho realizzato le `data class @Serializable` per eseguire il parsing del json. Infine ho scritto le funzioni che mi hanno permesso di tradurre i dati ricevuti dall'api nelle strutture dati definite nel primo sprint.

5.3.1 Sprint Backlog

R3.0	**Sviluppo parser per JSON dell'API**	O
R3.1	Sviluppo data class @Serializable	O
R3.1.1	Sviluppo Gruppo4JSON	O
R3.1.2	Sviluppo Gruppo4Data	O
R3.1.3	Sviluppo Gruppo4Filter	O
R3.1.4	Sviluppo Gruppo4Actions	O
R3.1.5	Sviluppo Gruppo4Node	O
R3.2	**Sviluppo adapter da JSON a TableState**	O
R3.2.1	Sviluppo funzione convertListOfGruppo4Node()	O
R3.2.2	Sviluppo funzione convertTree()	O
R3.2.3	Sviluppo funzione convertListOfGruppo4Actions()	O
R3.2.4	Sviluppo funzione convertCells()	O
R4.0	**Sviluppo funzioni per effettuare richieste HTTP**	O
R4.1	Sviluppo funzione fetch()	O
R4.2	Sviluppo funzione sendAction()	O

5.3.2 Soluzioni implementate

5.3.2.1 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 5.10: Funzione fetch()

```
1 suspend fun fetch() {
2   val res = window.fetch(url, RequestInit(
3     method = "GET",
4     credentials = RequestCredentials.Companion.INCLUDE,
5     headers = {
6       json("Accept" to "application/json")
7       json("Content-Type" to "application/json")
```

```
8       }))
9       .await()
10      .json()
11      .await()
12
13      // Codice identificativo dell'istanza della tabella pivot necessario
14      // per le chiamate successive
15      INSTANCE_KEY = (res as kotlin.js.Json)["InstanceKey"] as String?
16  }
```

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. Quando un utente esegue un'azione all'interno della tabella questa deve essere comunicata all'API. Per ottenere questa funzionalità 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 5.11: Funzione sendAction()

```
1  suspend fun sendAction(body: String, type: String) {
2      val res: Any? = window.fetch("$url$INSTANCE_KEY/$type", RequestInit(
3        method = "POST",
4        body = body,
5        credentials = RequestCredentials.Companion.INCLUDE,
6        headers = {
7          json("Accept" to "application/json")
8          json("Content-Type" to "application/json")
9        }))
10      .await()
11      .json()
12      .await()
13 }
```

5.3.2.2 Entità @Serializable

Le entità `@Serializable` sono state scritte per ogni oggetto JSON all'interno del JSON in arrivo dall'API. Per non dilungare questa sezione con codice molto simile mostrerò solo la struttura di una entità:

Listing 5.12: data class Gruppo4Action

```
1  @Serializable
2  data class Gruppo4Action(
3    @SerialName("Action")
4    val action: String? = null,
5    @SerialName("Dim")
6    val dim: Int? = null,
7    @SerialName("Depth")
8    val depth: Int? = null,
9    @SerialName("Path")
10   val path: Array<String>? = null
11 )
```

5.3.2.3 Funzioni adapter

Di seguito indico le funzioni principali per eseguire la conversione tra le entità `@Serializable` e le entità dello stato. Dalle seguenti funzioni si può vedere l'utilizzo delle *higher order functions* per semplificare il più possibile il codice.

Listing 5.13: Funzione convertListOfGruppo4Node()

```
fun convertListOfGruppo4Node(list: ArrayList<Gruppo4Node>?): ArrayList<
    DimensionsNode>? {
        return list?.map { it.toDimensionsNode() }?.toCollection(ArrayList())
}
```

Listing 5.14: Funzione convertListOfGruppo4Actions()

```
fun convertListOfGruppo4Actions(list: ArrayList<Gruppo4Action>, limit: Int):
    ArrayList<ArrayList<HeaderAction>> {
        val res: ArrayList<ArrayList<HeaderAction>> = ArrayList()
        var index = 0
        while (index < limit) {
                res.add(index, ArrayList())
                index++
        }

        list.groupBy { it.dim }.entries.map {
                val tmpList = it.value.map { c -> c.toHeaderAction() }.
                    toCollection(ArrayList())
                val limit = tmpList.last().depth
                res.set(it.key!!, normaliseListAction(tmpList, limit))
        }

        return res
}
```

Listing 5.15: Funzione convertCells()

```
fun convertCells(list: Array<Array<Double>>, rowsPaths: Array<Array<String>>?
     , colsPaths: Array<Array<String>>?): ArrayList<ArrayList<BodyCells>>? {
        val tmp: ArrayList<ArrayList<BodyCells>> = ArrayList()
        var cellTmp: ArrayList<BodyCells> = ArrayList()
        if (rowsPaths == null || colsPaths == null) {
                return null
        }
        for ((indx, items) in list.withIndex()) {
                val rPath = rowsPaths[indx]
                for ((index, item) in items.withIndex()) {
                        cellTmp.add(
                        BodyCells(
                        value = item.toInt(),
                        cPath = colsPaths[index],
                        rPath = rPath
                        )
                        )
                }
                tmp.add(cellTmp)
                cellTmp = ArrayList()
        }
        return tmp
}
```

Listing 5.16: Funzione convertTree()

```
fun convertTree(node: DimensionsNode?, level: Int, currentPath: List<String>, depth: Int): List<DimensionsNode?> =
if (node == null) {
        listOf()
} else {
        val subDimNode = node.subDim
        val action = createActionFor(node.children)
        val newNode = node.setLevel(level).setPath(currentPath).setActionType(action).setDepth(depth)
        (
        listOf(newNode) +
        (
        subDimNode?.flatMap {
                getRow(
                it.setLevel(level + 1),
                level + 1,
                currentPath + it.id,
                if (!node.isChild) depth else 0
                )
        } ?: listOf()
        ) +
        (
        node.children?.flatMap {
                getRow(
                it.setLevel(level).setIsChild(true),
                level,
                currentPath.dropLast(1) + it.id,
                depth + 1
                )
        } ?: listOf()
        )
        ).sortedWith(compareBy { it?.level })
}
```

Listing 5.17: Funzione parseJSON()

```
fun parseJSON(res: String): TableState {
  val obj = Json.decodeFromString<Gruppo4Json>(res)

  val tableData: ArrayList<ArrayList<BodyCells>> = convertCells(obj.cells, obj.rows?.paths, obj.cols?.paths)!!
  val rowTree = convertListOfGruppo4Node(obj.rows?.tree)
  val colTree = convertListOfGruppo4Node(obj.cols?.tree)

  val rowCells = getCellsFromTree(rowTree !!)
  val colCells = getCellsFromTree(colTree !!)

  val rowNormalisedCells = normaliseList(rowCells)
  val colNormalisedCells = normaliseList(colCells)

  val rowAction = gruppo4actionsToActionType(obj.rows?.actions !!, rowCells.groupBy { it?.level }.size)
  val colAction = gruppo4actionsToActionType(obj.cols?.actions !!, colCells.groupBy { it?.level }.size)

  val rows = rowCells.toCollection(ArrayList())
  val cols = colCells.toCollection(ArrayList())

  return TableState(
    cells = tableData,
    cols = colNormalisedCells,
    rows = rowNormalisedCells,
```

```
24      colTree = cols,
25      rowTree = rows,
26      rowAction = rowAction,
27      colAction = colAction
28    )
29  }
```

5.3.3 Sprint Review

Niente da segnalare.

5.3.4 Backlog refinement

Il Product Backlog non è stato modificato.

5.3.5 Problemi riscontrati

La realizzazione dell'adapter è stata rallentata da cambiamenti della struttura dell'API da parte dell'azienda. In particolare il cambiamento di alcuni oggetti JSON che hanno implicato un'ampliamento delle entità `@Serializable`. Inoltre la struttura ad albero che contiene le dimensioni d'analisi della tabella è stata modifica estensivamente. La nuova struttura identifica per ogni nodo due tipologie di figli: i nodi graficamente sotto di esso e i nodi graficamente laterali ad esso. Questi cambiamenti hanno implicato una riscrittura delle funzioni dell'adapter per ovviare a questo cambiamento.
La realizzazione dell'adapter ha quindi occupato parte della settimana riservata al quarto sprint.

5.4 S4: Caricamento parziale

Durata: *4 giorni*
Nell'ultimo sprint finale sono state realizzate le funzioni che riguardano il caricamento parziale dei dati della tabella. Tuttavia per alcuni problemi riguardanti l'API fornita da Gruppo4 è stato impossibile realizzare in modo completo la funzionalità del caricamento parziale.

5.4.1 Sprint Backlog

R5.0	**Sviluppo caricamento parziale**	O
R5.1	Sviluppo componente infinteScroller	
R5.2	Sviluppo funzioni per aggiornare struttura ad albero	O

5.4.2 Soluzioni implementate

Il seguente componente utilizza funzioni della programmazione funzionale fornite da React come: `useState` e `useEffect` per controllare quando l'utente ha raggiunto la fine della tabella in modo da effettuare una successiva chiamata all'API.

Listing 5.18: Funzione parseJSON()

```
1  val infiniteScroller = functionalComponent<InfiniteScrollerProps> { props ->
2          val (isFetchingVertical, setIsFetchingVertical) = useState(false)
3          val (isFetchingHorizontal, setIsFetchingHorizontal) = useState(false)
4
5          val isScrolling = { e: Event ->
6                  val el = document.getElementById("pivot-table-body");
7                  if (el != null && el.scrollTop.toInt() + el.clientHeight ==
                       el.scrollHeight) {
8                          setIsFetchingVertical(true)
9                  }
10
11                 if (el != null && el.scrollLeft.toInt() + el.clientWidth ==
                       el.scrollWidth) {
12                         setIsFetchingHorizontal(true)
13                 }
14         }
15
16         useEffect(listOf(isFetchingVertical)) {
17                 if (isFetchingVertical) {
18                         // eseguire chiamata api
19                 }
20         }
21
22         useEffect(listOf(isFetchingHorizontal)) {
23                 if (isFetchingHorizontal) {
24                         // eseguire chiamata api
25                 }
26         }
27         div {
28                 attrs.id = "pivot-table-body"
29                 attrs.onScrollFunction = isScrolling
30                 props.children()
31         }
32 }
```

Per quanto riguarda il caricamento parziale all'apertura o chiusura di un nodo di una dimensione ho realizzato le seguenti funzioni:

Listing 5.19: Funzione parseJSON()

```
fun updateNode(tree: ArrayList<DimensionsNode>, node: DimensionsNode):
    ArrayList<DimensionsNode> {
        if (tree.size > 1) {
                tree.map {
                        findNode(it, node)
                }
        } else {
                findNode(tree[0], node)
        }
        return tree
}

fun findNodeChildren(current: DimensionsNode, node: DimensionsNode) {
        current.children?.map { child ->
                findNode(child, node)
        }
}

fun findNode(current: DimensionsNode, node: DimensionsNode) {
        if (current.subDim.isNullOrEmpty())
        return

        if (current.path == node.path) {
                current.replace(node)
                return
        }

        current.subDim!!.map {
                findNode(it, node)
                it.children?.map { child ->
                        if (child.subDim.isNullOrEmpty())
                          findNodeChildren(child, node)
                        else
                          findNode(child, node)
                }
        }
}
```

5.4.3 Sprint Review

Niente da segnalare.

5.4.4 Backlog refinement

Il Product Backlog non è stato modificato.

5.4.5 Problemi riscontrati

Nell'ultimo sprint sono stati riscontrati dei problemi per quanto riguarda la realizzazione del caricamento parziale durante lo scroll di un utente. Il componente che si occuperà di realizzare la chiamata all'API è stato implementato, tuttavia l'API di Gruppo4 non presenta ancora le informazioni necessarie per implementare questa funzionalità; l'azienda mi ha informato che queste modifiche potranno essere implementate solo

dopo la fine del tirocinio curriculare. Le funzione e il componente richiesto dall'azienda è stato comunque implementato.

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

React è una libreria JavaScript dichiarativa, efficiente e flessibile che viene usata per costruire interfacce utente. Permette di realizzare interfacce utente complesse da piccoli e isolati "pezzi di codice" chiamati componenti. Le principali caratteristiche che lo rendono una delle librerie usate più usate sono la presenza di:

- componenti (che possono essere scritti in modo funzionale o a classe), questo implica un alto riutilizzo del codice;
- virtual DOM, React crea una cache della struttura del DOM, in seguito computa le differenze e infine esegue l'update solo dei componenti che sono cambiati (riconciliazione), questo implica migliori prestazioni.

6.3 Redux

Redux è un contenitore prevedibile dello stato per applicazioni JavaScript. Lo stato in Redux viene definito come dati che cambiano nel tempo. Lo stato determina ciò che viene renderizzato nell'interfaccia utente. Oltre alla semplice funzione di contenitore dello stato, Redux esegue anche la gestione di esso, in particolare si occupa di:

1. recupero e memorizzazione dei dati;
2. assegnare i dati agli elementi dell'interfaccia utente;
3. modifica dei dati.

6.4 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. Questi wrapper mi hanno permesso di scrivere componenti React e utilizzare funzioni e classi delle due librerie.

6.4.1 kotlin-react

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

6.4.2 kotlin-redux

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

6.4.3 kotlin-react-redux

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

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 parziale di sezioni della tabella dato i problemi riscontrati durante la codifica dello sprint numero 3.

7.2 Conoscenze acquisite e valutazioni personali

In questa sezione verranno identificate le conoscenze che ho acquisito per ogni metodologia e tecnologia utilizzate nell'ambito del progetto e la loro efficacia da un punto di vista personale.

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 per lavorare in un team di sviluppo che pratica una metodologia agile.

Efficacia

considero 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 una comunicazione costante tra i membri del progetto 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 a suddividerlo in piccoli incrementi. Il vantaggio che ho notato maggiormente durante la codifica in sprint è stato il fatto che la metodologia agile poneva come obiettivo la realizzazione delle funzionalità più importanti nei primi sprint, questo mi ha permesso di avere una visione completa dell'applicazione già dai primi sprint.

7.2.2 Kotlin

Durante lo studio iniziale e la codifica del componente d'interfaccia grafica ho maturato la mia conoscenza del linguaggio di programmazione Kotlin. Considero questa

conoscenza molto importante dato che Kotlin è utilizzato da molte aziende odierne. Inoltre in quanto è un linguaggio che permette di realizzare molti prodotti (api, web app, applicazioni android native, etc..) penso che questo progetto mi abbia fornito le basi per espandere le mie abilità su altri campi oltre che naturalmente lo sviluppo di applicazioni web.

Efficacia

Lo sviluppo in Kotlin di interfacce utente per il web è una soluzione valida ma dal mio punto di vista deve maturare ancora per quanto riguarda lo sviluppo di applicazioni web. Infatti la documentazione sull'argomento è scarsa e gli esempi disponibili online sono molto limitati. Inoltre i wrapper forniti da Jetbrains sono ancora incompleti per quanto riguarda l'insieme delle funzionalità di React e Redux.

7.2.3 React e Redux

La libreria React è utilizzata molto per la realizzazione di applicazioni web da molte aziende. In passato avevo già lavorato con React in progetti personali tuttavia, in questo progetto, ho ampliato e soprattutto affinato le mie conoscenze di questa libreria. Lo stesso vale per 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.

Efficacia

Considero questo due librerie 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. La scrittura di componenti React permette di riutilizzare codice, inoltre la divisione per componenti aiuta a organizzare meglio la struttura di una interfaccia utente.

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.

Efficacia

L'utilizzo delle *higher order function* ha 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 chiara e concisa delle funzioni.

7.2.5 Lavorare da remoto

Dato le circostanze sociali ho lavorato principalmente da remoto, questo ha portato ad alcune difficoltà per quanto riguarda la comunicazione, tuttavia l'applicazione della metodologia agile ha aiutato.

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia

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