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

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. Le principali librerie discusse sono state: React e Redux.

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. Come React essa offre un dataflow unidirezionale tuttavia lo stato non è più contenuto all'interno di una gerarchia di componenti grafici. Esso viene 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 mediante *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. 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 Model

3.2.1 Struttura

Per rappresentare la struttura dati dello stato della tabella pivot ho utilizzato le `data class` di Kotlin. Degli oggetti che hanno come unico scopo quello di contenere informazioni. Per prima cosa ho realizzato un nuovo package di nome `Entities`. Al suo interno ho definito le `data class` che costituiscono lo stato dell'applicazione.
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. Le entità che ho realizzato sono le seguenti:

- `TableState`;
- `DimensionsNode`;
- `NodeActionType`;
- `BodyCells`;
- `HeaderAction`.

3.2.1.1 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 3.1: TableState

```
data class TableState(
  val cols: ArrayList<ArrayList<DimensionsNode>>,
  val rows: ArrayList<ArrayList<DimensionsNode>>,
  val cells: ArrayList<ArrayList<BodyCells>>,
  val rowAction: ArrayList<ArrayList<HeaderAction>>,
  val colAction: ArrayList<ArrayList<HeaderAction>>
)
```

Descrizione campi dati

- `cols`: celle dell'intestazione della tabella delle colonne;
- `rows`: celle dell'intestazione della tabella delle righe;
- `cells`: celle contententi i dati della tabella;
- `rowAction`: pulsanti all'interno del componente TableActionUI;
- `colAction`: pulsanti all'interno del componente TableActionUI.

3.2.1.2 DimensionsNode

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

Listing 3.2: DimensionsNode

```
data class DimensionsNode(
  var id: String,
  var label: String,
  var level: Int? = null,
  var childDepth: Int? = null,
  var path: List<String>? = null,
  var actionType: NodeActionType = NodeActionType.NULL,
  var isChild: Boolean = false
)
```

Descrizione campi dati

- `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;
- `isChild`: indica se la cella è un figlio di un'altra cella.

3.2.1.3 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 3.3: HeaderAction

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

Descrizione campi dati

- `actionType`: indica il tipo dell'azione che bisogna eseguire;
- `dim`: indica la dimensione su cui applicare l'azione;
- `depth`: indica la profondità all'interno della dimensione su cui applicare l'azione.

3.2.1.4 NodeActionType

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

Listing 3.4: NodeActionType

```
1 enum class NodeActionType(val type: String) {
2   EXPAND("E"),
3   COLLAPSE("C"),
4   NULL("")
5 }
```

Descrizione campi dati

- `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.

3.2.1.5 BodyCells

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

Listing 3.5: BodyCells

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

Descrizione campi dati

- `value`: indica il dato;
- `cPath`: indica a che dimensioni delle colonne appartiene il dato;
- `rPath`: indica a che dimensioni delle righe appartiene il dato.

3.2.2 Utilizzare dati reali

Durante i primi due sprint di sviluppo ho utilizzato dati mockati in modo da realizzare una struttura adeguata alla renderizzazione. Ho suddiviso l'utilizzo di dati reali in tre passaggi:

1. effettuare richiesta API;
2. realizzare data class necessarie;
3. realizzare adapter;

3.2.2.1 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 3.6: 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 3.7: 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()
}
```

3.2.2.2 JSON Parser

Dopo aver ricevuto i dati, ho effettuato il parsing del JSON, per farlo ho per prima cosa studiato il JSON ritornante dall'API, in seguito ho sviluppato le `data class`. Infatti in Kotlin per effettuare il parsing di un JSON bisogna mapparne correttamente la struttura. La realizzazione del parser non è stata difficoltosa, tuttavia è stata tediosa a causa dell'alto numero di `data class` da realizzare.

Struttura JSON API
L'oggetto JSON che si ottiene dall'API ha la seguente struttura:

Listing 3.8: 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],
    [ ... ]
  ]
}
```

Entità @Serializable
Per utilizzare in modo appropriato il JSON risultante dalla richiesta all'API ho realizzato delle entità `@Serializable` che mi hanno permesso di tradurre tutto il JSON risultante in una gerarchia di `data class`.

3.2.2.3 Adapter

L'ultimo passo consiste nell'effettuare la traduzione da entità @Serializable nelle entità dello stato descritte precedentemente. La realizzazione dell'adapter può essere suddiviso in tre passaggi che corrispondono alle entità da tradurre.

Utilizzo della programmazione funzionale

Traduzione in BodyCells

Traduzione in HeaderAction

Traduzione in DimensionNode

3.3 Redux

Per rendere modulare lo stato è stata realizzata una "Slice". Uno "Slice" è definito come una parte di stato dell'applicazione che può essere unito con altri slice per realizzare lo stato completo dell'applicazione. Questa architettura è molto utile per garantire scalabilità allo stato; nell'applicazione lo stato è definito in un oggetto chiamato `TableStateSlice` definito in questo modo:

Listing 3.9: TableStateSlice

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

  // Thunk
  private val initTableState = InitState()
  fun initTable() : RThunk = initTableState
```

```
8
9     // Actions
10    class UpdateCells(val cells: ArrayList<ArrayList<BodyCells>>): RAction
11    ...
12
13    // Reducer
14    fun reducer(state: State = State(), action: RAction) : State { ... }
15  }
```

3.3.1 Stato

Lo stato gestito da Redux è equivalente alla struttura di `TableState` con l'aggiunta del campo dato `isLoading` così da avere un variabile per gestire la tabella durante il caricamento dall'API esterna di dati. Lo stato dell'applicazione è definito nel seguente modo:

Listing 3.10: State

```
1  data class State(
2    val cols: ArrayList<ArrayList<DimensionsNode>> = ArrayList(),
3    val rows: ArrayList<ArrayList<DimensionsNode>> = ArrayList(),
4    val cells: ArrayList<ArrayList<BodyCells>> = ArrayList(),
5    val rowActions: ArrayList<ArrayList<HeaderAction>> = ArrayList(),
6    val colActions: ArrayList<ArrayList<HeaderAction>> = ArrayList(),
7    val isLoading: Boolean = false
8  )
```

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 3.11: 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>()
```

Dall'interfaccia `RThunk` ho realizzato la classe `InitState` che, mediante la funzione `invoke` utilizza le coroutines di kotlin per riempire lo stato di `TableStateSlice` con i dati ricevuti dall'API.

Listing 3.12: InitState

```
1  class InitState : RThunk {
2    override fun invoke(
3      dispatch: (RAction) -> WrapperAction,
4      getState: () -> AppState
5    ): WrapperAction {
6      val mainScope = MainScope()
7        mainScope.launch {
8          val res : TableState = fetchCreavistaJson()
9          dispatch(TableStateSlice.UpdateRows(res.rows))
10         dispatch(TableStateSlice.UpdateCols(res.cols))
11         dispatch(TableStateSlice.UpdateCells(res.cells))
12         dispatch(TableStateSlice.UpdateRowActions(res.rowAction))
13         dispatch(TableStateSlice.UpdateColActions(res.colAction))
14       }
15     return nullAction
16   }
17 }
```

3.3.3 Actions

Le actions sono definiti come delle classi di tipo `RAction` che vengono usati per innescare l'update dello stato.

Listing 3.13: Actions

```
1 class UpdateCells(val cells: ArrayList<ArrayList<BodyCells>>): RAction
2 class UpdateRows(val rows: ArrayList<ArrayList<DimensionsNode>>): RAction
3 class UpdateCols(val cols: ArrayList<ArrayList<DimensionsNode>>): RAction
4 class SetIsLoading(val b: Boolean): RAction
5 class UpdateRowActions(val n: ArrayList<ArrayList<HeaderAction>>): RAction
6 class UpdateColActions(val n: ArrayList<ArrayList<HeaderAction>>): RAction
```

3.3.4 Reducer

Un *reducer* è una *funzione pura* che riceve come argomento un `RAction` e ritorna una copia dello stato modificato. L'implementazione utilizzata in `TableStateSlice` è la seguente:

Listing 3.14: Reducer

```
1 fun reducer(state: State = State(), action: RAction) : State {
2   return when (action) {
3     is UpdateCells -> state.copy(cells = action.cells)
4     is UpdateRows -> state.copy(rows = action.rows)
5     is UpdateCols -> state.copy(cols = action.cols)
6     is SetIsLoading -> state.copy(isLoading = action.b)
7     is UpdateRowActions -> state.copy(rowActions = action.n)
8     is UpdateColActions -> state.copy(colActions = action.n)
```

```
9       else -> state
10    }
11  }
```

3.4 Componenti grafici

Per quanto riguarda la struttura dei componenti grafici l'azienda mi ha dato completa libertà per quanto riguarda la loro struttura.Ho diviso la loro realizzazione in due parti: mockup della soluzione e codifica del componente React. Di seguito verranno elencati e spiegati, per ogni componente, i due passaggi dello sviluppo.

3.4.0.1 TableView

Questo componente è stato realizzato per definire la struttura generale del componente. L'idea alla base di questo componente è stata quella di fornire uno spazio suddivisibile in quattro sezioni. Ogni sezione è stata in seguito popolata dagli altri componenti.

Mockup
Come si può vedere dalla seguente immagine ho realizzato il componente utilizzando il layout CSS Grid. Questo mi ha permesso di gestire, in modo responsivo, la struttura del componente. Ho valutato la scelta del tipo di layout tra flex e grid. Ho deciso di optare per quest'ultimo perchè forniva un modo semplice e conciso di suddividere questo componente in quattro sezioni distinte tra loro.

Codifica

3.4.0.2 TableHeaderView

Questo componente rappresenta le dimensioni superiori della tabella pivot.

Mockup

Codifica

3.4.0.3 TableSidebarView

Questo componente rappresenta le dimensioni laterali della tabella pivot.

Mockup

Codifica

3.4.0.4 TableBodyView

Questo componente rappresenta i dati della tabella pivot.

Mockup

Codifica

3.4.0.5 TableLabel

Questo componente rappresenta una singola cella di intestazione della tabella.

Mockup

Codifica

3.4.0.6 TableActionUI

Questo componente rappresenta l'insieme delle interazioni che possono essere effettuate sulle dimensioni.

Mockup

Codifica

3.4.1 Container Redux

Per utilizzare lo stato di Redux nei componenti di React ho realizzato dei wrapper per ogni componente di React che necessitava di utilizzare lo stato di Redux. Questi wrapper mi hanno permesso di mappare lo stato e le funzioni per chiamare la funzione reducer nelle props di ogni componente grafico.

3.4.1.1 TableController

3.4.1.2 TableHeaderController

3.4.1.3 TableSidebarController

3.4.1.4 TableBodyController

3.4.1.5 TableActionUIController

3.4.1.6 TableLabelController

Capitolo 4

Codifica

4.1 Sprint di sviluppo

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**.

4.1.1 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.

4.1.1.1 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

4.1.1.2 Struttura del progetto

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

4.1.1.3 Sprint Review

Niente da segnalare.

4.1.1.4 Backlog refinement

Il Product Backlog non è stato modificato.

4.1.1.5 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`. Questi problemi non hanno però causato rallentamenti nell'implementazione degli altri requisiti dello Sprint Backlog.

4.1.2 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.

4.1.2.1 Sprint Backlog

R2.0	**Definizione e pianificazione 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

4.1.2.2 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
```

4.1.2.3 Sprint Review

Niente da segnalare.

4.1.2.4 Backlog refinement

Il Product Backlog non è stato modificato.

4.1.2.5 Problemi riscontrati

Niente da segnalare.

4.1.3 S3: Parser JSON e adapter

4.1.3.1 Sprint Backlog

4.1.3.2 Soluzioni implementate

4.1.3.3 Sprint Review

Niente da segnalare.

4.1.3.4 Backlog refinement

Il Product Backlog non è stato modificato.

4.1.3.5 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.

4.1.4 S4: Caricamento parziale

4.1.4.1 Sprint Review

Niente da segnalare.

4.1.4.2 Backlog refinement

Il Product Backlog non è stato modificato.

4.1.4.3 Sprint Review

4.1.4.4 Backlog refinement

4.1.4.5 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

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