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 Gruppo4 ha accumulato competenze e l'esperienza necessaria per fornire soluzioni efficaci e innovative nel settore web. Mediante un modello organizzativo consolidato e certificato sviluppano WebApp 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. La soluzione deve essere una tabella pivot interattiva dove l'utente ha 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 è particolarmente interessata alle sue applicazioni nello sviluppo di interfacce utente perchè Kotlin è già utilizzato nello sviluppo di API di Gruppo4.

1.3 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 il concetto di *dataflow*[g]dell'applicazione.

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 applicare nella sua interezza il metodo SCRUM 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/artifacts che sono stati utilizzati sono i seguenti:

- **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 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 date le circostanze sociali;

- **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 in particolare mediante le *Funzioni di ordine superiore* 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*. Come definito dalla documentazione di Kotlin (bib: https://kotlinlang.org/docs/reference/lambdas.html), le funzioni scritte nel linguaggio Kotlin sono considerate come *first-class* quindi esse possono essere contenute in variabili e strutture dati, possono essere passate come argomento di altre funzioni e possono essere ritornate da altre *Funzioni di ordine superiore*. Le higher-order functions più usate nel progetto sono state:

- map;

- sortedWith;

- flatMap;

- groupBy;

- more?

2.3 Versionamento della soluzione

2.3.1 Git

Git è un VCS (Version Control System) distribuito che permette di tenere traccia delle modifiche in un prodotto software e di organizzarne la codifica. In questo tirocinio è stata usata la tecnica del feature-branch: ogni aggiunta di feature corrisponde all'apertura di un nuovo branch 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 (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 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* 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`:;
- `kotlin-react-redux`:;
- `kotlin-extensions`:;
- `kotlinx-serialization-core`:;
- `kotlinx-coroutines-core`:.

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* 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 garantire la scalabilità del componente. Per gestire lo stato dell'applicazione sono state individuate due possibili soluzioni: 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*.
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 deriva da questo dataflow, 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 medianti funzioni pure (glossario) 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 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 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.
Dato che i cambiamenti effettuati dalle Actions possono essere solo asincroni ho utilizzato un interfaccia che fornisce le stesse funzionalità di un Actions ma ne espande l'utilità permettendo operazioni asincrone.

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 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 richiest HTTP alla API di Gruppo4. Per farlo ho utilizzato l'implementazione di Kotlin di `window.fetch`. La funzione risultante è la seguente:

Listing 3.6: BodyCells

```
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?
}
```

Per quanto riguarda le richieste di ricezione di nuovi dati ho utilizzato la stessa implementazione di Kotlin per realizzare la seguente funzione:

Listing 3.7: BodyCells

```
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")
```

```
8         json("Content-Type" to "application/json")
9       }))
10      .await()
11      .json()
12      .await()
13  }
```

3.2.3 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` necessarie per effettuare la lettura del JSON. La realizzazione di questa parte del progetto è stata rallentata molto da un problema all'API dovuto al cambiamento della struttura durante lo sviluppo da parte dell'azienda.

3.2.3.1 Struttura JSON API

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

Listing 3.8: BodyCells

```
{
  "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],
    [ ... ]
  ]
}
```

3.2.3.2 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 `data class`.

3.2.3.3 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.

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: BodyCells

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

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

  // Actions
  class UpdateCells(val cells: ArrayList<ArrayList<BodyCells>>): RAction
  ...

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

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: BodyCells

```
data class State(
  val cols: ArrayList<ArrayList<DimensionsNode>> = ArrayList(),
  val rows: ArrayList<ArrayList<DimensionsNode>> = ArrayList(),
  val cells: ArrayList<ArrayList<BodyCells>> = ArrayList(),
  val rowActions: ArrayList<ArrayList<HeaderAction>> = ArrayList(),
  val colActions: ArrayList<ArrayList<HeaderAction>> = ArrayList(),
  val isLoading: Boolean = false
)
```

3.3.2 Thunk

I thunk sono definiti come middleware di Redux e sono usate 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: BodyCells

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

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

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

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: BodyCells

```
class InitState : RThunk {
  override fun invoke(
    dispatch: (RAction) -> WrapperAction,
    getState: () -> AppState
  ): WrapperAction {
    val mainScope = MainScope()
      mainScope.launch {
        val res : TableState = fetchCreavistaJson()
        dispatch(TableStateSlice.UpdateRows(res.rows))
        dispatch(TableStateSlice.UpdateCols(res.cols))
        dispatch(TableStateSlice.UpdateCells(res.cells))
```

```
12          dispatch(TableStateSlice.UpdateRowTree(res.rowTree))
13          dispatch(TableStateSlice.UpdateColTree(res.colTree))
14          dispatch(TableStateSlice.UpdateRowActions(res.rowAction))
15          dispatch(TableStateSlice.UpdateColActions(res.colAction))
16          console.log(res)
17        }
18      return nullAction
19    }
20  }
```

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: BodyCells

```
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

funzione pura che riceve come argomento un `RAction` e ritorna una copia dello stato modificato.

Listing 3.14: BodyCells

```
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

3.4.1 View

3.4.1.1 TableView

Questo componente è stato realizzato per definire la struttura generale del componente e per suddividere in modo responsivo lo spazio in quattro quadranti. 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.

3.4.1.2 TableHeaderView

Questo componente rappresenta le dimensioni superiori della tabella pivot. Dire che la struttura della tabella è stata realizzata con flex, table, etc.

3.4.1.3 TableSidebarView

Questo componente rappresenta le dimensioni laterali della tabella pivot. Dire che la struttura della tabella è stata realizzata con flex, table, etc.

3.4.1.4 TableBodyView

Questo componente rappresenta i dati della tabella pivot Dire che la struttura della tabella è stata realizzata con flex, table, etc.

3.4.1.5 TableLabel

Questo componente rappresenta i dati della tabella pivot Dire che la struttura della tabella è stata realizzata con flex, table, etc.

3.4.1.6 TableActionUI

Questo componente rappresenta i dati della tabella pivot Dire che la struttura della tabella è stata realizzata con flex, table, etc.

3.4.2 Container Redux

3.4.2.1 TableController

3.4.2.2 TableHeaderController

3.4.2.3 TableSidebarController

3.4.2.4 TableBodyController

3.4.2.5 TableActionUIController

3.4.2.6 TableLabelController

3.5 Parser JSON

3.5.1 Struttura dati @Serializable

3.6 Adapter

3.6.1 Utilizzo del paradigma della programmazione funzionale

3.6.1.1 Funzioni utilizzate

Capitolo 4

Codifica

4.1 Studio delle tecnologie

4.2 Sprint di sviluppo

4.2.1 Primo sprint di sviluppo: Realizzazione dei componenti grafici

Descrizione

Sprint Backlog

Soluzioni implementate

4.2.2 Secondo sprint di sviluppo: Realizzazione della struttura dei dati e del parser JSON

Descrizione

Sprint Backlog

Soluzioni implementate

4.2.3 Terzo sprint di sviluppo:

Descrizione

Sprint Backlog

Soluzioni implementate

4.2.4 Quarto sprint di sviluppo:

Descrizione

Sprint Backlog

Soluzioni implementate

4.3 Codifica dei test

4.3.1 Unit test

Capitolo 5

Analisi dei requisiti

5.0.1 Definizione delle User Stories

Per prima cosa il team di sviluppo si è occupato di realizzare le User Stories. Per definire un singolo elemento è stata utilizzata 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.

5.0.2 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 una prodotto con le funzionalità principali già implementate.

5.1 Product Backlog

Id	**Descrizione**	**Priorità**	**Implementato**
US1.1	Come Cliente voglio poter visualizzare i miei dati e le dimensioni relative ai dati	A	SI
US1.2	Come Cliente voglio poter utilizzare questa applicazione web dal mio PC, dal mio telefono e dal mio Tablet	A	SI
US1.3	Come Cliente voglio poter visualizzare solo i dati senza le dimensioni relative ad essi	M	SI
US2.1	Come Cliente voglio poter utilizzare l'API precedente di Creavista	A	SI
US2.2	Come Cliente voglio poter utilizzare la nuova API di Creavista	A	SI
US2.3	Come Cliente voglio avere un caricamento veloce	M	SI
US3.1	Come cliente voglio poter salvare la mia configurazione	M	SI
US3.2	Come cliente voglio poter esportare i miei dati	B	SI
US3.3	Come cliente voglio poter decidere quali filtri voglio utilizzare per ogni dimensione	A	SI
US3.4	Come cliente voglio poter modificare i filtri che sto utilizzando	A	SI

5.2 Requisiti individuati dal Product Backlog

Id	Descrizione	Tipo	Impl.	User Story
R1.0	Definizione e pianificazione dei componenti	O	SI	US1.1
R1.1	Sviluppo dei componenti React visivi	O	SI	US1.1
R1.1.1	Sviluppo di TableView	O	SI	-
R1.1.2	Sviluppo di TableHeaderView	O	SI	-
R1.1.3	Sviluppo di TableSidebarView	O	SI	-
R1.1.4	Sviluppo di TableBodyView	O	SI	-
R1.2	Sviluppo dei container Redux	O	SI	US1.1
R1.1.1	Sviluppo di TableController	O	SI	-
R1.1.2	Sviluppo di TableHeaderController	O	SI	-
R1.1.3	Sviluppo di TableSidebarController	O	SI	-
R1.1.4	Sviluppo di TableBodyController	O	SI	-
R2.0	Sviluppo dei JSON parser	O	NO	US2.1
R2.1	Sviluppo parser per Creavista	O	SI	-
R2.1.1	Sviluppo data structure in Kotlin	O	SI	-
R2.1.2	Sviluppo adapter	O	SI	-
R2.2	Sviluppo parser per API nuova	O	NO	US2.2
R2.2.1	Sviluppo data structure in Kotlin	O	NO	-
R2.2.2	Sviluppo adapter	O	NO	-
R3.0	Sviluppo di un sistema di caricamento automatico	O	NO	US2.3
R3.1	Sviluppo di InfiniteScroller per gestire il caricamento automatico	O	SI	-

Capitolo 6

Tecnologie e strumenti

6.1 Tecnologie

INSERIRE L'IDEA DI COME LE TECNOLOGIE SONO STATE UTILIZZATE NEL PROGETTO

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 libreria per la JVM, Android e il browser.

React

React è una libreria che presenta principalmente due caratteristiche:

- l'uso del Virtual DOM;
- realizzazione di componenti migliorare la reutilizzazione del codice.

Kotlin Wrappers

kotlin-react

kotlin-redux

kotlin-react-redux

6.2 Strumenti

Di seguito viene data una descrizione delle tecnologie che sono stati utilizzati durante il tirocinio.

Capitolo 7

Conclusioni

7.1 Problemi riscontrati

7.2 Raggiungimento degli obiettivi

7.3 Conoscenze acquisite

7.3.1 Metodologia agile

7.3.2 Kotlin

7.3.3 React e Redux

7.3.4 Programmazione funzionale

7.3.5 Lavorare in un team di sviluppo

7.4 Valutazione personale

7.4.1 Effettività delle metodologie agili

7.4.2 Effettività di Kotlin per la realizzazione di UI per il web

7.4.3 Effettività di React e Redux

7.4.4 Effettività della programmazione funzionale

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia

"Life is really simple, but we insist on making it complicated"

— Confucius

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