Università Politecnica delle Marche

Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Utilizzo di REACT per la realizzazione del front-end in applicazioni Web basate sul framework Laravel

Use of REACT front-end in Web applications based on the Laravel framework

Relatore:
Prof. Alessandro Cucchiarelli

Candidato:
Mattia Scuriatti

Anno accademico 2020/2021

INDICE

INTRODUZIONE

Nell'era moderna, le applicazioni web sono ormai all'ordine del giorno, per qualsiasi ambito e contesto. Queste ci permettono di usufruire comodamente di innumerevoli servizi, che possono andare dai classici siti web ai vari software adibiti a molteplici usi. Proprio per questa loro diffusione di massa, è fondamentale creare e gestire tali applicazioni nel miglior modo possibile, con la finalità di offrire un servizio di qualità sempre maggiore. Tra tutte le parti che compongono le applicazioni web, sicuramente l'interfaccia utente (UI) gioca un ruolo fondamentale. Un'interfaccia ben progettata e realizzata permette una percezione dell'applicativo più piacevole, ma allo stesso tempo può rendere più funzionale la fruizione da parte dell'utente. Le tecnologie che caratterizzano questo mondo sono innumerevoli. In generale però, le più diffuse sono:

- JavaScript: un linguaggio di programmazione orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client (esteso poi anche al lato server). Esso permette la creazione, in siti e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da eventi innescati a loro volta in vari modi dall'utente sulla pagina web in uso.
- AJAX: una tecnica di sviluppo software per la realizzazione di applicazioni web interattive, basata su uno scambio di dati in background fra web browser e server, consentendo così l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente.
- React js o Angular js o Vue.js: librerie JavaScript per la creazione delle componenti dell'interfaccia utente.

- CSS: un linguaggio usato per definire la stilizzazione di applicazioni web.
- HTML: un linguaggio usato per strutturazione di applicazioni e pagine web.
- JQuery: una libreria JavaScript che nasce con l'obiettivo di semplificare la selezione, la manipolazione, la gestione degli eventi e l'animazione di elementi DOM in pagine HTML, nonché semplificare l'uso di funzionalità AJAX, la gestione degli eventi e la manipolazione dei CSS.

Questi linguaggi/librerie permettono, tra le varie cose che possono fare, la realizzazione del cosiddetto front-end di un'applicazione web. Quest'ultimo rappresenta l'interfaccia dell'applicativo che interagisce con l'utente. Nel progetto trattato, il protagonista principale sarà proprio una libreria che può essere utilizzata per lo sviluppo del front-end: React. Questa è una delle librerie JavaScript più utilizzate per la creazione di interfacce lato utente. React è stata creata da Facebook nel 2013 e pone le sue radici sul concetto di componente. I componenti sono elementi grafici che vengono definiti tramite delle funzioni o delle classi. Accettano in input dati arbitrari e hanno lo scopo di ritornare elementi React che descrivono cosa dovrebbe apparire sullo schermo. A riprova di ciò, i componenti React possono essere sottoposti a rendering su un particolare elemento nel DOM tramite l'uso di alcuni metodi che verranno illustrati nel corso della tesi. Per molte delle librerie esistenti, e React non fa eccezione, il linguaggio HTML nello specifico viene utilizzato quasi esclusivamente per creare "componenti Web" riutilizzabili, a volte estendendo il linguaggio HTML stesso. La forza di React rispetto ad altre librerie è quella di consentire l'uso di un approccio dichiarativo simile all'HTML, quindi molto familiare, per definire i componenti che rappresentano parti significative e logiche dell'interfaccia utente. Lo scopo che questa tesi si è prefissata è quello di comprendere a fondo la tecnologia fornita da React e di imparare ad utilizzarla per lo sviluppo di un progetto pratico. Come verrà illustrato

l’applicazione da sviluppare ha lo scopo di fornire un’interfaccia per controllare un sistema di cui abbiamo ignorato la tecnologia sottostante. L’obiettivo, dunque, sarà quello di creare un applicativo moderno ed intuitivo cercando di rispettare tutti i requisiti per esso definiti.

CAPITOLO 1
CONTESTO APPLICATIVO

1.1 Introduzione

Con il termine "Applicazione Web", s'intende un'applicazione risiedente in un Server Web. Quest'ultimo è un processo sempre attivo su una macchina che ascolta richieste HTTP, reperisce il documento richiesto (oppure compie operazioni più complesse come eseguire un programma o degli script) e restituisce il risultato al client web. Il Server Web (o server HTTP), come tutti i servizi basati su TCP/IP (protocolli di rete per la trasmissione dei pacchetti e per l'instradamento), si attiva su una porta, che è il numero a cui si fa corrispondere il servizio. In questo modo una macchina può far girare svariati servizi differenziando le porte. Ci sono porte standard per i vari servizi (ftp, telnet, posta elettronica, web). La porta di default del web server è la 80, ma può essere configurato per funzionare su un'altra porta libera. In questo caso la porta va specificata nella configurazione del web server e nell'URL (indirizzo che individua univocamente una risorsa web) a cui fa riferimento il web server locale:

http://localhost:8080/

La parola chiave *localhost* sta ad indicare il server web attivo sulla macchina locale alla porta 8080 (che può funzionare senza una connessione internet). Quindi la macchina locale svolge contemporaneamente il ruolo di server e di client. Si accede al Server Web tramite un browser (software per la visualizzazione di risorse sul web) o un altro programma con funzioni di navigazione, operante secondo gli standard del World Wide Web [1].

L’interazione che avviene si basa sul modello Client/Server, nel quale vi sono due entità: il client, colui che richiede il servizio e il server, colui che lo offre. Il ruolo dei processi è dinamico in base alle coppie che fanno parte del collegamento, ma durante un’interazione esso non può cambiare. Il client e il server possono risiedere sullo stesso calcolatore, o come si vede in Figura 1, su elaboratori differenti interfacciati tramite una rete, che si scambiano informazioni tramite messaggi (strutture in cui verranno iniettati i dati).

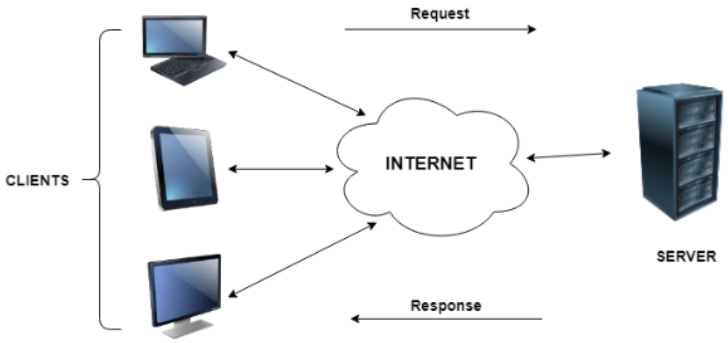

Figura 1. Modello Client/Server con tre client

1.2 Modello Client Server

Il modello C/S, in precedenza, è stato introdotto in modo molto discorsivo ma ovviamente vi sono aspetti più tecnici da prendere in considerazione. Prendendo come riferimento un’interazione tra il browser (client) e il server, dalla Figura 2 si nota che il tutto inizia dall’interfaccia utente che è a disposizione del client. Da qui si manda una richiesta, tramite URL, all’applicazione risiedente nel server, la quale recupera il file corrispondente nel proprio file system e in uscita produce l’interfaccia utente.

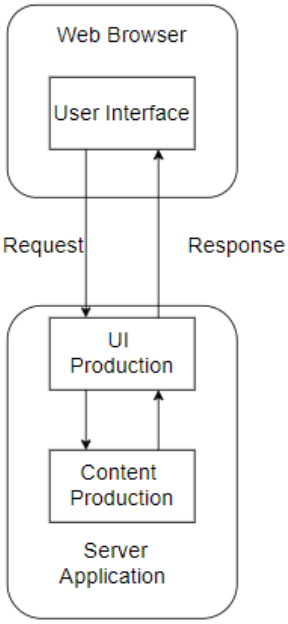

Figura 2. Modello C/S classico

L'interazione nel caso in Figura 2 è puramente statica ma ci possono essere diverse varianti; ad esempio, si possono fornire contenuti in modo dinamico, oppure con la tecnologia AJAX si possono iniettare oggetti parziali in modo asincrono rispetto al caricamento della pagina web. Come ultima variante, troviamo il modello che verrà usato nell'applicazione sviluppata per questo progetto, la Single Page Application. Questa tipologia di applicazione è diversa dal modello classico. Qua il server non fornisce intere pagine, ma ne carica solo una inizialmente. Sarà l'applicazione lato Server a generare dei contenuti, i quali verranno mandati direttamente al client (ad esempio in formato JSON). A quest'ultimo verrà demandato il compito di produrre la UI iniettando i contenuti appena ricevuti dal server.

1.3 World Wide Web e Internet

Come citato, le applicazioni web sono rese disponibili agli utenti grazie all'ausilio di reti di vario genere, tra le quali sappiamo quella per eccellenza essere Internet. Essa è definita come una federazione di reti che comunicano

attraverso un insieme di protocolli, che fanno parte del cosiddetto World Wide Web. Quest'ultimo è stato progettato e implementato da Tim Berners Lee, ed è nato per codificare e condividere facilmente documenti, articoli scientifici e materiale multimediale, su una rete TCP/IP. Il W3 è basato su tre concetti chiave:

- URL (Uniform Resource Locator): protocollo per identificare il riferimento alla posizione del documento e ha la seguente struttura:

  protocollo://dominio/path/nome del file

- HTTP (HyperText Transfer Protocol): protocollo che implementa la negoziazione del formato tra client e server.
- HTML (HyperText Markup Language): linguaggio usato per la strutturazione dei documenti.

Il W3 è costituito da particolari nodi, presenti nella rete, che rendono disponibili le informazioni in essi contenute sotto forma di pagine. Un ipertesto è un documento con struttura non sequenziale, che contiene riferimenti ad altri documenti, atto a favorire la consultazione a seconda delle esigenze (fruizione personalizzata), senza la rigidità della struttura fisica sequenziale. Un altro principio cardine del W3 è la negoziazione del formato, citata nel protocollo HTTP; ovviamente le informazioni che viaggiano nella rete non sono tutte codificate nello stesso modo, dunque non essendo possibile definire uno standard unico si è dovuta trovare una soluzione. Il client all'atto della richiesta dovrà comunicare al server i formati che esso è in grado di gestire, quindi il server, invierà il documento in accordo con le informazioni ricevute. Naturalmente si intuisce che il W3 dipende fortemente dal modello C/S e che ha bisogno di efficaci meccanismi di ricerca, tramite i quali si potranno trovare documenti ipertestuali grazie a richieste sotto forma di testo.

1.3.1 Protocollo Http

Il protocollo http si basa su un modello di scambio di messaggi, i quali svolgono il ruolo di richieste o risposte. Una richiesta HTTP comprende diversi campi, quali:

- *Metodo*: specifica il tipo di operazione che il client richiede al server (es: GET,POST…).
- *URL*: per identificare la risorsa richiesta.
- *Informazioni addizionali* (es: data e ora, tipi di dato accettabili per la visualizzazione sul browser, versione del protocollo HTTP, tipi di software utilizzati dal client…).

Il metodo più frequentemente usato (e sarà così anche nella richiesta che faremo alla nostra applicazione) è il GET. Grazie ad esso si richiede una risorsa, la quale può essere statica o dinamica. Altro metodo molto utilizzato è il POST, che invece permette di trasferire informazioni dal client al server , ad esempio per aggiornare delle risorse esistenti o per fornire dei dati in ingresso. Per quanto riguarda le risposte HTTP, queste sono a grandi linee simili nel formato alle richieste. La differenza più evidente è l'informazione riguardante lo stato della risposta e il suo eventuale contenuto.

1.4 Applicazione Web

Lasciando alle spalle il contorno delle applicazioni web cerchiamo di approfondire meglio quest'ultime. Ci sono due parti fondamentali che compongono la struttura di un'applicazione web e che ne permettono l'utilizzo:

- Back-end
- Front-end

1.4.1 Back-end

Il back-end è la parte nascosta all'utente che accede alla piattaforma. Questa, infatti, è destinata alla gestione del sito/applicazione da parte del personale incaricato e in essa sono incapsulate tutte le funzionalità messe a disposizione dall'applicativo. Il back-end developer, dunque, deve avere competenze in vari settori quali: programmazione, database, servizi web, sicurezza informatica e progettazione di software.

1.4.2 Front-end

Il front-end, invece, è la sezione destinata alla navigazione dell'utente. Anche l'elemento visivo, infatti, è una parte strategica delle applicazioni web. É importante però sottolineare che non si tratta solo di grafica (che è di competenza del web designer) ma di *user experience*, cioè di rendere piacevole e funzionale l'esperienza di chi deve navigare sul sito (o utilizzare l'applicazione). Per strutturare un front-end vi sono tre linguaggi di programmazione da cui non si può prescindere: HTML che abbiamo brevemente citato, il CSS che si occupa di definire il formato di visualizzazione dei contenuti e JavaScript, il quale ci permette di aggiungere alle pagine elementi interattivi e dinamici [2]. Sarà proprio JavaScript, grazie anche all'ausilio di alcune librerie che verranno introdotte nel seguito, il linguaggio principe per la formazione del front-end dell'applicazione richiesta dal progetto.

1.4.3 Costruzione di un'applicazione web

La strutturazione di un'applicazione web può anche essere eseguita in modo molto semplice e modulare, grazie all'uso di librerie e framework. Entrando nel dettaglio, un framework permette ad un utente di avere uno scheletro della propria applicazione senza dover iniziare tutto da zero: sarà poi il developer a specializzare l'applicazione per un determinato scopo. D'altro canto, le librerie

servono per eseguire operazioni ben precise. Non sono altro che un insieme di classi e metodi che possono essere richiamate, senza farci perdere tempo a creare funzioni che sono state già implementate da altri per noi.

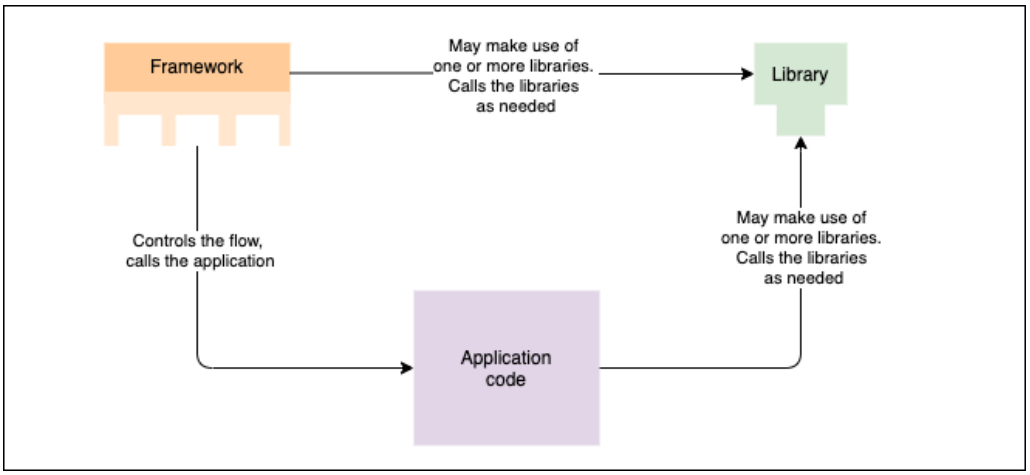

Figura 3. Funzionamento di un'applicazione che fa uso di un framework e di librerie

Osservando la Figura 3, si ha un'idea un po' più concreta del funzionamento di una applicazione web. In sostanza, il framework definisce un'architettura per la nostra applicazione e la controlla nel suo ciclo di vita. D'altro canto, ci sono le librerie, alcune appartenenti al framework, altre esterne, che invece vengono chiamate dall'applicazione o dal framework quando necessario. Al giorno d'oggi l'uso di queste tecnologie è fondamentale per evitare perdite di tempo. Ormai, quasi nessun sito viene codificato interamente da zero, perché richiederebbe una tempistica troppo elevata, per un risultato che si può ottenere molto più facilmente con le soluzioni appena descritte.

CAPITOLO 2
OBIETTIVI DEL PROGETTO

2.1 Scopo generale

Il progetto in questione, come detto, ha lo scopo di creare il front-end di una *single page application*, usata come interfaccia per interagire con un determinato sistema. Alla tecnologia dietro il sistema, ovviamente non è stata data troppa attenzione, visto che l'obiettivo principale è quello di creare l'interfaccia per l'applicazione, predisposta per l'interazione dell'utente con i componenti di tale sistema. Parliamo dunque di una pagina web, strutturata in HTML, graficata tramite i fogli di stile CSS, e che fa uso del framework Laravel e della libreria React (JavaScipt), di cui parleremo nel dettaglio all'interno del Capitolo 3. Logicamente il progetto non ha come unico fine quello di creare l'interfaccia dell'applicazione. L'ulteriore finalità è quella di apprendere tutti gli aspetti essenziali che caratterizzano la tecnologia React, prendendoci dimestichezza. Dalla creazione di nuovi componenti grafici, alla gestione dell' interazione tra loro stessi ma anche con l'utente (sempre riguardante l'aspetto front-end), React offre moltissime funzionalità al programmatore. Mettendo come si dice "le mani in pasta" infatti, si è riusciti a cogliere meglio la teoria che c'è dietro questa libreria JavaScript, che oggi è uno degli strumenti maggiormente utilizzati nel suo settore.

2.2 Applicazione nel dettaglio

Nella prima fase di progetto, la cosa più importante è comprendere bene le funzionalità demandate all'applicazione e naturalmente anche l'aspetto grafico che deve assumere, fondamentale per un progetto incentrato sullo sviluppo del

front-end. Scendendo nel dettaglio, l'interfaccia del sistema deve fornire un pannello suddiviso in sei sezioni. Ognuna di queste sezioni è destinata ad ospitare un componente, avente lo scopo di far interagire l'utente con una determinata parte del sistema. Essenziale quindi, è il fatto di predisporre una pagina con elementi graficamente gradevoli, intuitivi e moderni. Il risultato finale dovrà prevedere una visuale dell'applicazione il più possibile simile a quella mostrata in Figura 4. In aggiunta, si dovranno implementare delle funzionalità per ogni componente, le quali verranno descritte nelle sezioni a seguire. Da non sottovalutare affatto è la disposizione dei componenti; quest'ultima svolge un ruolo fondamentale nel risultato finale del nostro applicativo.

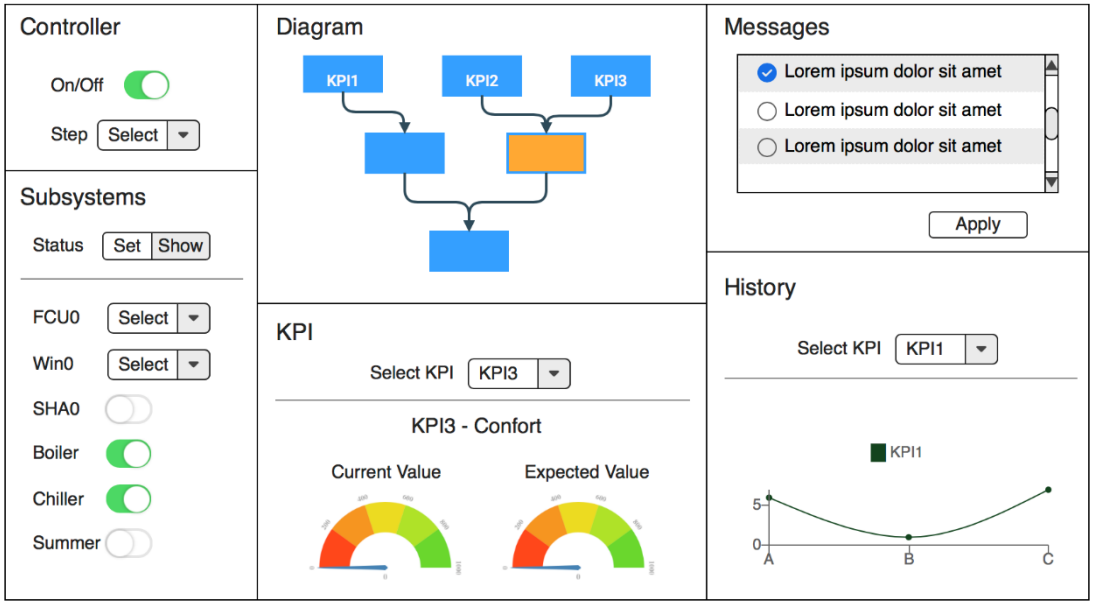

Figura 4. Visuale richiesta dell'interfaccia per l'applicazione web del progetto

2.2.1 Controller

Il primo componente, che viene mostrato in Figura 5, prende il nome di Controller. Controller molto semplicemente, come tutti gli altri componenti del sistema, deve essere contenuto in un riquadro che in questo caso è collocato

nella parte in alto a sinistra della finestra. Lo scopo di Controller è quello di visualizzare a schermo due elementi:

- un bottone On/Off: in gergo prende il nome di toggle switch button**.** Un bottone, dunque, che nel momento in cui è acceso deve avere uno sfondo colorato, mentre se è spento deve essere privo di colore. In aggiunta, nel momento in cui avviene la transizione da accesso a spento e viceversa, si deve visualizzare nel bottone un'animazione a testimonianza di questo cambiamento.
- una select: dunque una *drop-down list* che permette di selezionare una tra le diverse opzioni.

Sia il bottone che la select inoltre, hanno un'etichetta relativa, per far intuire più facilmente all'utilizzatore lo scopo di quest'ultimi.

Figura 5. Componente Controller

2.2.2 Subsystems

Il secondo componente, in Figura 6, invece si occupa di gestire i cosiddetti sottosistemi (*Subsystems*). Esso stesso è collocato sotto al Controller ed è caratterizzato da una funzionalità interattiva. Questa funzionalità, che ora verrà descritta, comporta delle conseguenze grafiche tra i diversi elementi che costituiscono il componente. In sostanza, nella parte superiore del riquadro vi sarà lo *status*, ossia una coppia di bottoni che avranno il compito di mettere il componente *Subsystems* in due stati differenti:

- *"Set"*: modalità che abilita l'uso degli elementi sottostanti a *status*.
- *"Show"*: modalità che disabilita l'uso di tutti i componenti sottostanti a *status* ma ne permette la visualizzazione.

Pertanto, cliccando su uno dei due bottoni il componente cambia il proprio stato e agisce di conseguenza. Ovviamente, se ci si trova in un determinato stato, e si clicca il bottone associato al medesimo stato non deve avvenire nulla. Sotto a *status,* inoltre, ci sarà una riga nera per rimarcare la suddivisione tra esso e il resto di *Subsystems*. Gli elementi con cui interagirà *status* sono: due select e quattro toggle switch button. Le prime sono identiche a quelle del controller e permettono semplicemente di impostare una sola opzione. Infine, i quattro toggle switch button, sono equivalenti anche loro nella funzionalità a quelli del Controller, e inizialmente due di questi saranno in stato di On e gli altri in Off. Anche in questo caso, per ogni elemento del componente abbiamo un'etichetta relativa.

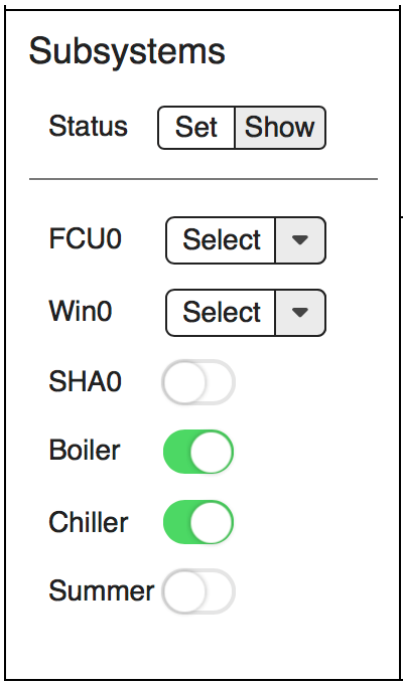

Figura 6. Componente Subsystems

2.2.3 Diagram

In alto e al centro troviamo il componente adibito a gestire il Diagram. Come si evince dalla Figura 7, esso visualizza informazioni organizzate secondo una struttura dati di tipo albero. In questo grafico, ogni nodo visualizza al centro del suo riquadro un nome. Al momento del click su ciascuno di questi nodi, l'applicazione deve far apparire una finestra, in cui verranno visualizzate delle informazioni inerenti al nodo in questione. Oltre a questo, sempre come conseguenza del click del nodo, il riquadro dovrà colorarsi in modo da far intendere all'utilizzatore quale nodo è stato cliccato. Naturalmente, se ci si trova nella condizione in cui il nodo A è stato cliccato in precedenza, e successivamente viene cliccato il nodo B, nodo A deve ritornare con il colore di sfondo della configurazione iniziale.

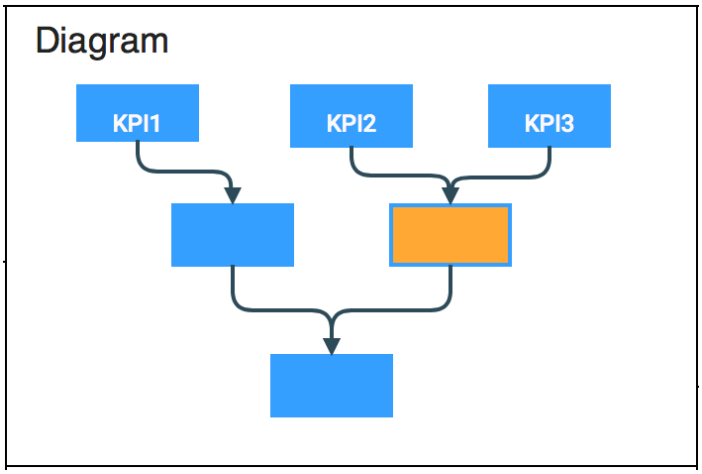

Figura 7. Componente Diagram

2.2.4 KPI

Sotto al Diagram abbiamo la parte di interfaccia che opera sul cosiddetto KPI. Questa è divisa in due sezioni come Subsystems. Nella sezione superiore troviamo una semplice select. Dando uno sguardo alla Figura 8, si intuisce che selezionando una delle opzioni, il sistema visualizzerà tramite due tachimetri, due valori associati alla scelta opzionata. Come in Subsystems anche in KPI vi

è una riga nera per dividere la select dai tachimetri. I tachimetri sono divisi in cinque fasce, con una scala cromatica che va dal rosso al verde. Ognuna di queste fasce avrà un relativo range, che verrà visualizzato all'esterno del tachimetro. Ultima caratteristica sarà mostrare entrambi i valori attuali dei tachimetri, che inizialmente partiranno da 0.

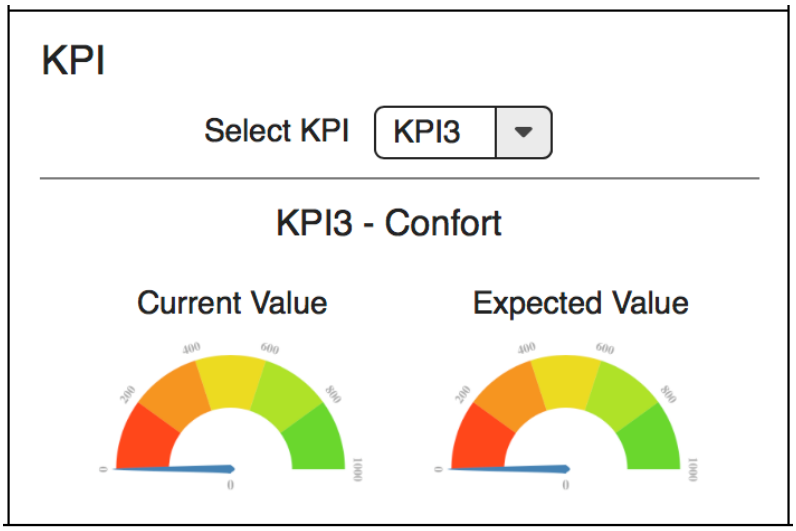

Figura 8. Componente KPI

2.2.5 Messages

In alto a destra troviamo Messages, il quale contiene un box con una lista di messaggi, come in Figura 9. Ad ognuno di questi messaggi è associata un'azione da eseguire sul sistema a cui l'interfaccia si riferisce. Nel contenitore ci sarà anche una barra di scorrimento, questo perché verranno visualizzati nella schermata iniziale solo alcuni messaggi rispetto al totale. Funzionalmente il componente agisce in modi differenti in base a due casistiche. Nella prima, l'utente può cliccare sul *radio button* affianco ad ogni messaggio per selezionare l'azione, ma quest'ultima non verrà eseguita. Solo dopo aver fatto click sul bottone "Apply", l'azione verrà inoltrata al sistema per apportare la relativa modifica (seconda casistica) . Nello specifico, lo scopo applicativo di Messages sarà quello di effettuare delle modifiche nei tachimetri del componente KPI al

momento del click o dell'invio del messaggio. Rispettivamente verranno modificati i valori visualizzati nei tachimetri di "Expected Value" o "Current Value".

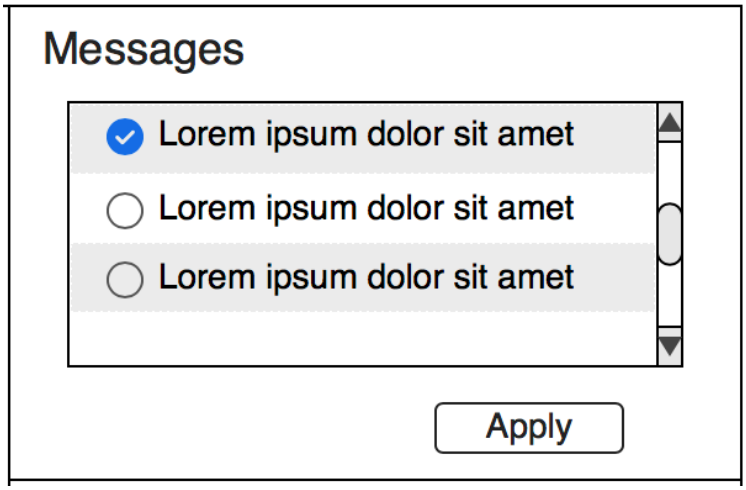

Figura 9. Componente Messages.

2.2.6 History

L'ultimo componente invece va sotto il nome di History. Osservando la Figura 10, troviamo al suo interno, una select nella parte superiore, la quale, selezionando un'opzione (tra i vari KPI) ci permette di visualizzare un grafico relativo alla nostra scelta.

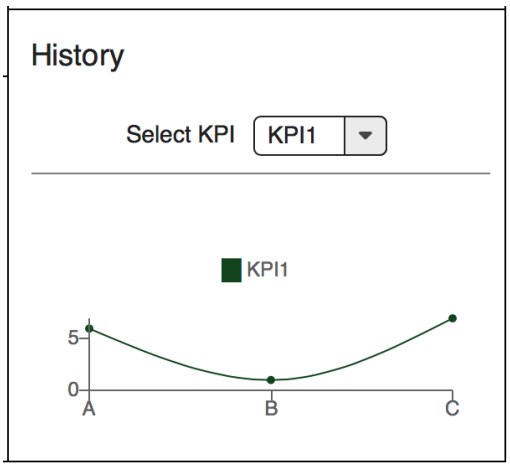

Figura 10. Componente History

Un dettaglio riguarda il colore delle funzioni rappresentate nel piano cartesiano. Per ognuna di esse viene associato un diverso colore per una maggiore distinzione. Anche in questo caso, a dividere il grafico con la select ci sarà una riga nera. Per quanto riguarda il grafico, esso è composto da un solo quadrante di un piano cartesiano bidimensionale. Nell'asse delle ascisse ci sono delle etichette per alcuni punti della funzione, mentre nell'asse delle ordinate si leggono i valori relativi. Infine, sopra al piano cartesiano è disposta una sorta di "legenda". Questa mostrerà il nome del KPI che è rappresentato dal grafico corrente ed un rettangolino del medesimo colore della funzione.

CAPITOLO 3
STRUMENTI UTILIZZATI

3.1 Software richiesti da un'applicazione web

Un'applicazione web, per essere "reperibile" e funzionale, richiede l'utilizzo di molteplici tecnologie:

- un *browser* che ci permette di richiedere l'applicazione.
- un *server* in grado di ricevere richieste HTTP e rispondere con delle pagine web o dei contenuti da iniettarvi.
- un *database* per immagazzinare e gestire i dati.

Nel nostro caso non necessitiamo di un database, essendo il progetto spoglio per quanto riguarda la parte di back-end. Gli elementi imprescindibili sono il server HTPP e il browser, ovviamente. Un altro dettaglio importante è il fatto che l'applicazione non è reperibile ad un indirizzo pubblico visto che come abbiamo detto è stata sviluppata nell'ambiente locale.

3.1.1 XAMPP

Per creare il nostro web server è stato utilizzato XAMPP, una distribuzione Apache gratuita adibita appositamente a questo tipo di operazioni. XAMPP è composta da diversi moduli: Apache HTTP Server, MySQL (o MariaDB), PHP, Tomcat, Mercury e Perl. La X iniziale fa riferimento al suo essere multipiattaforma, infatti è compatibile con i sistemi operativi Windows, Linux, Mac e Solaris. XAMPP viene utilizzato per creare un web hosting locale sul proprio computer grazie ad una distribuzione facile da installare; è particolarmente usato per scopi di testing nella programmazione web [3]. Come citato nel nostro caso, avendo bisogno unicamente della parte di server web,

all'avvio del software è necessario solamente avviare il modulo di Apache come illustrato in Figura 11.

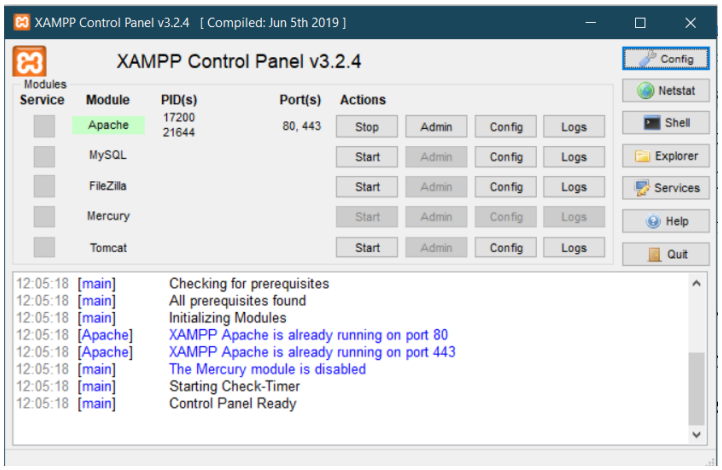

Figura 11. Configurazione di XAMPP usata per il progetto

XAMPP ci permette, grazie all'aiuto di un browser per la visualizzazione, di effettuare delle richieste al server Apache. Quest'ultimo, tramite un interprete, elabora la richiesta e risponde con delle risorse (file php o dati letti dal database) per poi ritornare al browser un documento HTML con iniettati i contenuti espressi nella richiesta. (Figura 12)

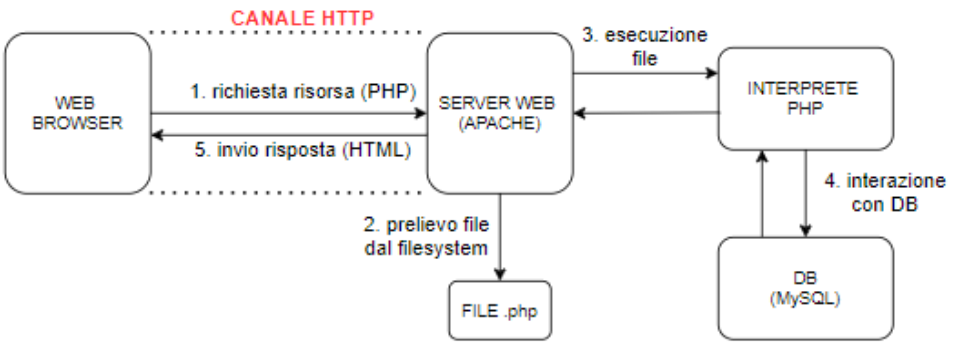

Figura 12. Ciclo di interazione tra client (browser) e il server (Apache)

3.1.2 Browser

Il Browser è un'applicazione predisposta per l'acquisizione, la presentazione e la navigazione di risorse sul web. Tali risorse (pagine web, immagini o video)

sono messe a disposizione sul World Wide Web, su una rete locale o sullo stesso computer dove il browser è in esecuzione. Il programma implementa da un lato le funzionalità di client per il protocollo HTTP, dall'altro quelle di visualizzazione dei contenuti ipertestuali (solitamente all'interno di documenti HTML) e di riproduzione di contenuti multimediali (rendering). Tra i browser più utilizzati vi sono Google Chrome, Internet Explorer, Mozilla Firefox, Microsoft Edge Safari e Opera.

3.2 Software per lo sviluppo

3.2.1 Visual Studio Code

Per lo sviluppo del codice sorgente è stato adoperato Visual Studio Code, un editor di testo molto innovativo, che può essere usato con vari linguaggi di programmazione, tra cui la famiglia di linguaggi C (C, C++, C#), HTML , PHP, Java e molti altri. Molto utile nell'editor è anche la presenza di una modalità di debug e una finestra per il terminale che permette di interagire con il filesystem e di eseguire svariati comandi. Questo come vedremo, ritornerà molto utile per eseguire comandi artisan (interfaccia a linea di comando di Laravel) ma soprattutto quelli relativi al gestore dei pacchetti npm.

3.2.2 Laravel

Essendo la creazione di applicazioni web complessa e onerosa vi sono dei cosiddetti framework a venirci in aiuto. Essi creano un'architettura logica di supporto sulla quale un programma può essere progettato e realizzato. Nel nostro caso abbiamo usato Laravel, tra i migliori per lo sviluppo di servizi o applicazioni web full-stack. È un framework open-source che adotta l'architettura MVC e che è basato su PHP, infatti è composto da una libreria di componenti implementate come classi PHP.

3.2.2.1 Architettura MVC

Uno dei principali punti di forza di Laravel è la sua architettura modulare. Tale aspetto permette di separare architetturalmente l'interfaccia, il modello dei dati e la gestione delle interazioni con gli utenti. Essa prende il nome di architettura MVC (Model View Controller) [4]:

- *Model*: modella i dati e implementa i metodi relativi (orientato agli oggetti).
- *View*: implementa l'interfaccia utente dell'applicazione e cattura gli input dell'utente.
- *Controller*: si occupa della gestione degli eventi trasformando le interazioni dell'utente in azioni sui dati.

Questa modularità comporta però una complessità maggiore rispetto ad un'architettura tradizionale, perché i moduli, come rappresentato in Figura 13, interagiscono e anche in modo frequente:

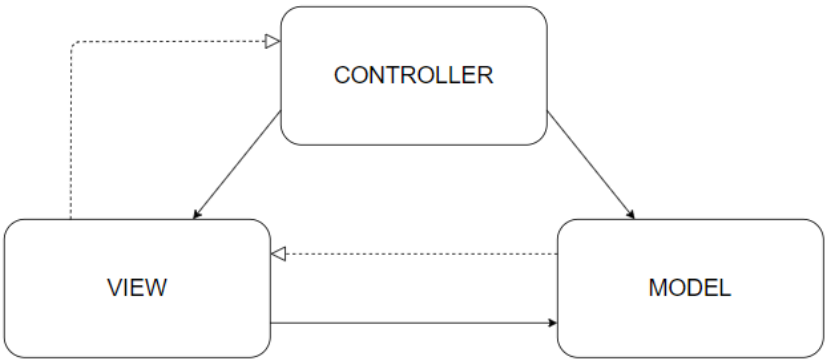

Figura 13. Schema delle associazioni dell'architettura MVC

- da *View* a *Model*: le viste accedono direttamente ai modelli per iniettare nell'output eventuali modifiche.
- da *Model* a *View*: i Model hanno un'associazione indiretta con le viste (es: se c'è un cambiamento nel model, la vista si aggiorna automaticamente con i nuovi dati).

- da *View* a *Controller*: le viste interagiscono indirettamente con i controller tramite degli eventi.
- da *Controller* a *View*: il controller accede direttamente alle componenti della vista (es: nei metodi dei controller possiamo ritornare in output delle viste).
- da *Controller* a *Model*: il Controller interagisce direttamente con il Model, visto che nei suoi metodi potrebbe richiamare dei dati che si riferiscono a modelli.

3.2.2.2 Ciclo di vita di una richiesta HTTP

Come si nota in Figura l4, il punto di ingresso per tutte le richieste ad un'applicazione Laravel è il file *public/index.php* che funge da punto di partenza per avviare il resto del framework. La prima azione che compie il file *index.php* è quella di caricare l'autoloader, la parte che consente di caricare i file di classe quando sono necessari, senza caricarli o includerli esplicitamente. Fatto ciò, viene creata un'istanza dell'applicazione tramite il file *bootstrap/app.php.* Successivamente, la richiesta viene inviata al kernel (*app/Http/Kernel.php)* che permette la configurazione della gestione degli errori, della registrazione, ma anche la rivelazione dell'ambiente in cui opera l'applicazione che può tornare utile in alcuni contesti (es: utilizzare localmente un driver di cache diverso da quello che si usa sul server di produzione). L'azione fondamentale eseguita dal kernel però è l'istanziazione e l'avvio dei provider di servizi della nostra applicazione (configurati nell'array del file *config/app.php*). I provider di servizi sono responsabili dell'avviamento di tutti i vari componenti del framework, come il database, la validazione e i componenti di instradamento (*routing*). Da questo momento Laravel ha terminato la sua fase di set up; da qui in poi la richiesta HTTP verrà inoltrata al *router (*anche lui caricato da un

provider). Quest'ultimo poi attiverà un *controller* e infine genererà la vista finale che verrà inoltrata al web server.

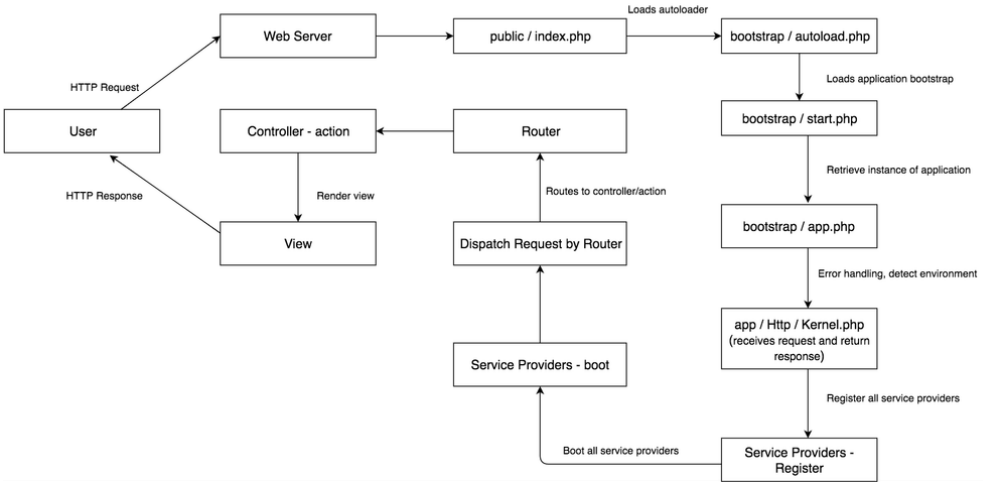

Figura 14. Ciclo di vita di una richiesta HTTP in Laravel

3.2.2.3 Concetti base

Durante l'illustrazione del ciclo di vita di una richiesta sono stati citati dei concetti come: *routing, controller* e *view.*

- Il *routing,* ossia l'instradamento delle richieste, si basa sull'URL (Uniform Resource Locator), una sequenza di caratteri che individua in modo univoco una risorsa all'interno del web. Laravel attua l'instradamento attraverso delle rotte, che sono specificate nell'URL della richiesta di seguito al dominio. L'elenco di queste rotte si trovano dentro dei file nella directory routes della cartella che contiene i files dell'applicazione, e ad ognuna è associato un verbo (get, post, put, patch, delete, redirect, view e option) che definirà come la richiesta dovrà essere elaborata. Inoltre, alle rotte può essere associato un nome (utile quando dovranno essere richiamate in altre parti dell'applicazione) e in esse potranno essere iniettati dei parametri con i loro relativi valori.
- I *controller* permettono di definire tutta la logica di gestione delle richieste al di fuori dei file di routing, usando apposite classi che

estendono la classe controller di base. Esse sono costituite da metodi che prendono il nome di *action*. Ad esempio, una classe potrebbe gestire tutte le richieste in entrata relative agli utenti, inclusa la visualizzazione, la creazione, l'aggiornamento e l'eliminazione degli utenti.

- Le *view* nascono vista l'inefficienza di restituire intere stringhe di documenti HTML direttamente dalle rotte o dai controller. Le viste si occupano dunque della logica di presentazione della richiesta generando un Response Object a partire da un Template nel quale verranno iniettati i dati prodotti dalle action.

3.2.2.4 Artisan

Laravel fa uso di questa shell, dalla quale possono essere attivati comandi (per creare componenti e database, cancellare la cache ecc..). La sintassi usata è la seguente:

php artisan *comando*

3.2.2.5 Composer

Altro elemento fondamentale è Composer, Laravel lo utilizza come gestore delle dipendenze. Permette di dichiarare e gestire le librerie (es: aggiornamento delle versioni) da cui dipende il progetto, installandole in una directory all'interno del progetto.

3.2.2.6 Directory applicazione Laravel

Nel momento in cui si crea un'applicazione Laravel (illustrata nel capitolo 4), si crea una directory predefinita che ha lo scopo di fornire un punto di partenza per il developer. Nello specifico abbiamo le seguenti cartelle [5]:

- *App:* in essa vi sono una serie di sottodirectory come Console, Http, Exceptions e Providers. La Console directory contiene tutti i comandi di Artisan, mentre la Http directory contiene i controller, i middleware e le richieste. La Exceptions directory contiene il gestore delle eccezioni mentre la cartella Providers ospita tutti i fornitori di servizi per la tua applicazione. Nonostante non sia presente nella configurazione iniziale, in App si può creare anche una directory Models, per modellare i nostri dati e farli interagire con le relative tabelle nei database.
- *Bootstrap:* contiene il file app.php che avvia il framework. Inoltre, ospita anche una cache directory, contenente i file generati dal framework per l'ottimizzazione delle prestazioni (es: file della cache del percorso e dei servizi).
- *Config:* contiene tutti i file di configurazione dell'applicazione.
- *Database:* contiene i file per la configurazione del database associato all'applicazione e i seeds per popolare le tabelle del database.
- *Public:* in essa è presente la document root dell'applicazione, ossia il file index.php, citato nei capitoli precedenti.
- *Resources:* immagazzina le viste, le risorse grezze e non compilate come CSS o JavaScript. Questa directory ospita anche tutti i file per la localizzazione del sito in lingue diverse .
- *Routes:* vi sono diversi file in cui vengono definite le rotte dell'applicazione. Nella nostra applicazione viene utilizzato solo il file web.php.
- *Storage:* contiene le sottodirectory App, Framework e Logs. La prima è utilizzata per archiviare qualsiasi file generato dall'applicazione. La seconda per memorizzare i file e le cache generati dal framework. Infine, la Logs directory contiene i file di log dell'applicazione.

- *Test:* è costituita da test automatici utilizzabili tramite il seguente comando artisan:

  php artisan test

- *Vendor:* contiene tutte le dipendenze della nostra applicazione.

3.2.2.7 Laravel Mix

Laravel Mix è un pacchetto che funge da interfaccia tra Laravel e Webpack (raccoglitore di moduli per applicazioni JavaScript). Esso fornisce una API, per definire i passaggi di configurazione di Webpack per l'applicazione Laravel, utilizzando diversi preprocessori CSS e JavaScript comuni. In sostanza, Laravel Mix compila, minimizza e archivia le risorse nella cartella pubblica dell' applicazione per un facile riferimento. Mix è un livello di configurazione sopra Webpack e per implementare le sue attività si deve eseguire uno degli script NPM inclusi nel file *package.json* [8]:

npm run dev

Quando si eseguirà tale comando, Webpack eseguirà le istruzioni nel file webpack.mix.js. Nel nostro caso questa operazione ci permetterà di compilare automaticamente *js/app.js* in cui noi includeremo i vari file Javascript con le nostre componenti React. Molto utile è anche il comando *npm run watch,* il quale continua a funzionare nel terminale e verifica continuamente tutti i file CSS e JavaScript rilevandone le modifiche. Webpack ricompilerà automaticamente le risorse quando rileverà una modifica a uno di questi file.

3.2.2.8 Blade

Blade è un template engine per la creazione di viste incluso in Laravel. Le viste che fanno uso di Blade vanno definite in un file con estensione *.blade.php,* e possono far uso di una serie di direttive molto comode codificate con la seguente

sintassi: *@nome_direttiva*. Ci sono diverse categorie di direttive per i seguenti scopi:

- controllo condizionale (@if, @else, @switch, @case , @elseif, @endelseif)
- controllo iterativo (@for, @endfor, @while, @endwhile, @foreach, @endforeach)
- controllo per l'autenticazione (@auth, @endauth, @guest, @endguest)
- gestione delle form (@csrf, @error , @enderror)
- definizione di layout per template

Nello specifico, nella nostra applicazione abbiamo fatto uso dell'ultima categoria, di cui nel capitolo 4 verranno spiegate le funzionalità in dettaglio. Il contenuto di questi file verrà tradotto in PHP e memorizzato nella cache dell'applicazione per evitare di ripetere più volte la fase di traduzione

3.2.2.9 Facade

Facade è un design pattern che definisce un oggetto, il quale fornisce una semplice interfaccia statica, alle classi disponibili nel *service container* dell'applicazione, che sono molto complesse. Tutte le facades di Laravel sono definite in *Illuminate\Support\Facades.*

3.2.3 React

React è una libreria Javascript per creare interfacce utente complesse, mantenendo modulabilità nello sviluppo. Permette di comporre UI complesse a partire da piccoli ed isolati stralci di codice chiamati "componenti". React può essere integrata in Laravel, sostituendola allo scaffolding di base Vue con dei comandi che verranno illustrati nel capitolo 4.

3.2.3.1 Virtual DOM

Nelle applicazioni web per la visualizzazione delle informazioni entra in gioco il DOM (Document Object Model). Il DOM è un'API di programmazione per i documenti, esso ne definisce la struttura logica e il modo in cui un documento è accessibile e manipolabile. Si basa su una struttura di oggetti ad albero che somiglia molto alla struttura dei documenti che modella [6] (Figura 15).

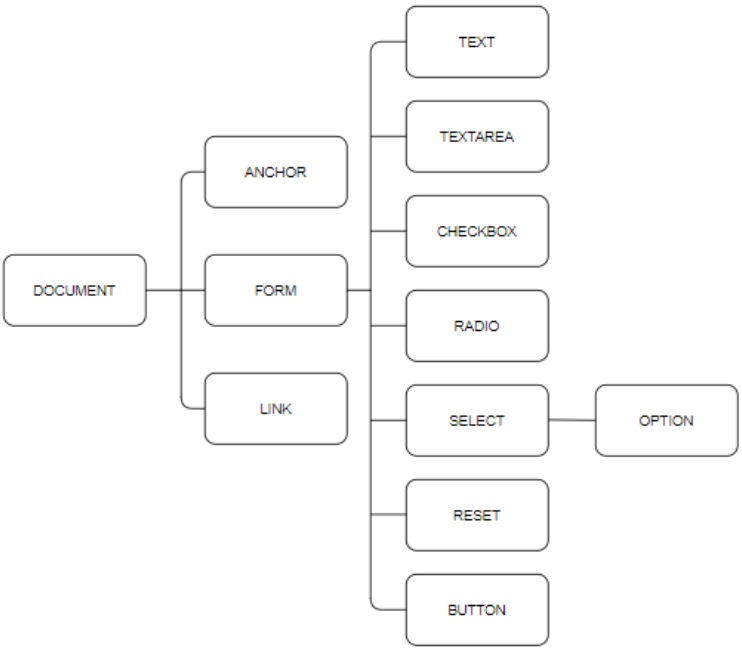

Figura 15. Esempio di DOM di un documento HTML

Ogni volta che si verifica una modifica nel documento, il DOM viene aggiornato per rappresentare tale modifica. Le frequenti manipolazioni all'interfaccia grafica dovute alle modifiche apportate, possono portare a un rallentamento delle prestazioni. Il re-rendering anche delle parti non interessate dalla modifica, infatti, è un procedimento che comporta un deterioramento delle prestazioni. Come soluzione al problema, React ha introdotto il concetto di Virtual DOM. Il Virtual DOM è una rappresentazione virtuale copiata in memoria del DOM reale. Ogni qual volta che cambia lo stato dell'applicazione viene aggiornato solamente il DOM virtuale, creando un nuovo albero ma mantenendo in memoria il precedente. Dopo la creazione di un nuovo albero, esso viene confrontato con l'albero del DOM virtuale prima dei cambiamenti di stato e

viene fatta un'analisi delle differenze. Le incongruenze che verranno trovate saranno le effettive modifiche da apportare al nuovo rendering dell'applicazione.

3.2.3.2 JSX

React permette di utilizzare una sintassi che estende quella del JavaScript chiamata JSX. La sintassi in questione permette di iniettare dentro delle variabili i tradizionali tag di HTML. JSX nel dettaglio rappresenta degli "elementi React", i quali non sono altro che descrizioni di ciò che si vuole visualizzare sullo schermo. React legge questi oggetti e li utilizza per costruire il DOM e tenerlo aggiornato. In merito a ciò React, durante la fase di compilazione, traduce l'espressione JSX attraverso il metodo *React.createElement* nell'oggetto a destra della Figura 16.

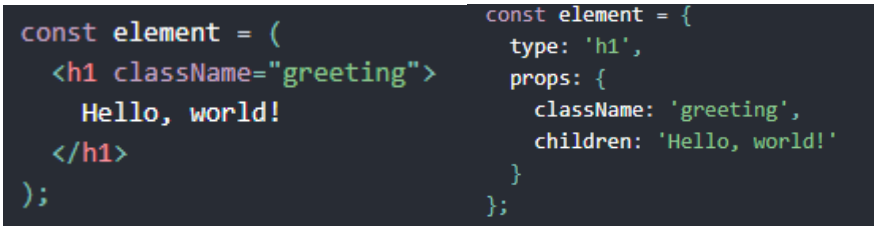

Figura 16. A sinistra un esempio di JSX e a destra l'oggetto creato dopo la compilazione.

3.2.3.3 Componenti con Stato e Props

Come annunciato, gli elementi React prendono il nome di componenti. Questi possiedono due caratteristiche specifiche: *stato* e *props*. Entrambi gli elementi controllano la vita del componente. Le props sono proprietà costanti che possono essere passate al componente, mentre lo stato è gestito dal componente stesso e si può modificare al presentarsi di un evento. Le props essendo immutabili sono variabili di sola lettura, e perciò una volta ricevute in input non potranno mai essere modificate da parte di quel componente. Ogni componente

di React necessita di alcune props per la sua inizializzazione, le quali si possono muovere in una sola direzione, cioè dal componente padre al componente figlio e non viceversa. Le props hanno perciò due fondamentali funzionamenti: il primo consiste nel creare una relazione ulteriore tra componente padre e componente figlio, mentre il secondo è la caratterizzazione del componente (stessi componenti con props differenti hanno un comportamento diverso). Le variabili rappresentanti lo stato invece, hanno un comportamento diverso, esse sono state create appositamente per essere mutevoli nel tempo. Vengono utilizzate dai componenti per tenere traccia delle loro informazioni e gestire le interazioni con l'utente. Esse assumeranno un valore differente a seconda della logica implementata per una determinata interazione. La modifica dello stato però non può avvenire con una banale operazione di assegnamento tra variabili; React, infatti mette a disposizione una funzione chiamata *setState()* per questo procedimento. La funzione *setState()*, quando attivata, determina un re-rendering del componente e di tutti i suoi figli in modo da poter visualizzare le modifiche apportate. Quest'ultima azione è opzionale, dato che tramite l'uso della funzione *shouldComponentUpdate()* è possibile evitare un re-rendering del componente. Un aspetto molto importante riguardante lo stato è la condivisione. Se due o più componenti vogliono condividere lo stato, devono "spostarlo" nel componente antenato più vicino che hanno in comune. Tale stato potrà poi essere acquisito dai figli iniettandolo nelle loro props, dunque non potranno modificarlo[7].

3.2.3.4 Metodi di Lifecycle

Precedentemente, abbiamo citato la funzione *shouldComponentUpdate(),* la quale fa parte di un gruppo di cosiddetti "metodi di lifecycle". Questi metodi hanno lo scopo di gestire il ciclo di vita di un componente React. Vengono suddivisi in tre categorie, in base al momento del ciclo di vita in cui vengono eseguiti:

- metodi di *montaggio*: vengono eseguiti dopo che l'output del componente è stato renderizzato nel DOM.
- metodi di *smontaggio*: vengono eseguiti quando un componente viene rimosso dal DOM.
- metodi di *aggiornamento*: vengono invocati immediatamente dopo che avviene un aggiornamento del componente.

3.2.3.5 React Developer Tools

Durante la fase di sviluppo di un'applicazione web con React è molto utile l'uso di React Developer Tools. È un'estensione di Chrome DevTools open source ma è disponibile anche per gli altri browser. Consente di ispezionare le gerarchie dei componenti React e di visualizzare i valori, sia dello stato che delle props, dei vari componenti come in Figura 17.

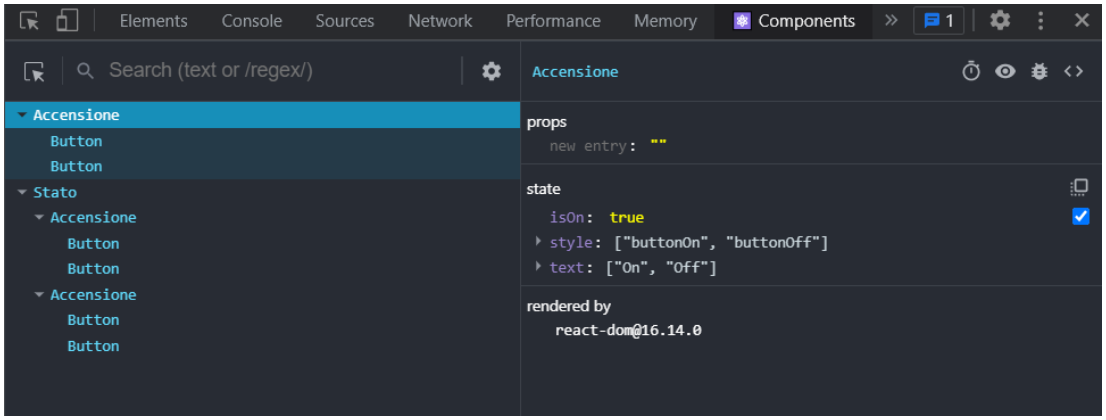

Figura 17. Visuale di React Developer Tools, a sinistra la gerarchia delle componenti e a destra le props e lo stato

3.2.4 NPM

Al giorno d'oggi, è svantaggioso creare componenti da zero quando ci sono dei pacchetti software che offrono componenti già definite che possiamo adattare alle nostre esigenze. Per fare ciò è necessario un gestore dei pacchetti, ed NPM

è il più diffuso nell'ambiente Javascript. NPM implementa le seguenti funzionalità [9]:

- Adatta i pacchetti di codice per le app o incorpora i pacchetti così come sono.
- Mette a disposizione un repository pubblico da cui scaricare pacchetti di codice pronti all'uso.
- Esegue i pacchetti senza scaricarli usando npx.
- Gestisce più versioni del codice e le dipendenze del codice.
- Aggiorna facilmente le applicazioni quando viene aggiornato il codice sottostante.
- Offre una CLI (Command Line Interface) per eseguire comandi NPM.

CAPITOLO 4
APPLICAZIONE SVILUPPATA

4.1 Configurazione dell'ambiente di sviluppo

La configurazione dell'ambiente di sviluppo svolge un ruolo fondamentale in un'applicazione web. Il primo passo nella fase iniziale ovviamente è stato quello di creare una nuova applicazione Laravel. Per una creazione totalmente da zero i passi da seguire sono i seguenti:

1. Installazione e configurazione di Composer
2. Installazione di Laravel Installer tramite il comando:

   composer global require laravel/installer

3. Creazione della cartella dell'applicazione con il comando:

   laravel new nome_progetto

Ovviamente tali comandi vanno eseguiti nella shell del proprio elaboratore, oppure nel proprio ambiente di sviluppo, nel caso sia fornito di un'interfaccia a riga di comando. In alternativa, come è stato fatto per l'applicazione trattata, si può utilizzare un progetto già precedentemente configurato, farne una copia e usare quella come punto di partenza. La cartella contenente l'applicazione è stata collocata nella directory seguente:

C:\xampp\htdocs

Questo perché, se vogliamo visualizzare la nostra applicazione tramite un server web, dobbiamo inserirla in una cartella che comprende già i file utili alla configurazione del web server. In questo caso *htdocs* è quella predefinita da

XAMPP per fornire questo servizio. Alla fine della configurazione, l'URL a cui farà riferimento la nostra applicazione sarà:

http://localhost/React/public

in cui React sarà il nome della cartella contenente l'applicazione (come anticipato nel capitolo 3 public è la document root dei nostri progetti Laravel). Il passo successivo è quello di cambiare lo scaffolding JavaScript di default, che è Vue, con React tramite i comandi:

composer require laravel/ui

php artisan react

php artisan ui react –auth

Con il primo installeremo il pacchetto "user interface" di Laravel tramite Composer. In esso, è collocato di default lo scaffolding di Vue basilare , insieme al suo scaffolding adibito all'autenticazione e alla registrazione. Tramite le ultime due istruzioni verranno sostituiti entrambi con quelli di React. Una volta fatto questo procedimento si noterà sulla cartella *js/components* (directory in cui inietteremo le componenti React) un nuovo file chiamato Example.js. Ultimo comando da eseguire sarà:

npm install

che ci installerà il gestore dei pacchetti insieme a Laravel Mix.

4.2 Sviluppo dell'applicazione

Una volta terminata la fase di configurazione si è iniziato subito con la realizzazione del codice. Lo sviluppo è stato eseguito, sulla base della "filosofia" di Laravel, in modo modulare e con una precisa logica.

4.2.1 Rotte

Come di buona norma, la prima parte sviluppata è stata quella riguardante le rotte. Aperto il file *web.php* della directory *routes*, è stata creata un'unica rotta. Questo perché, essendo una single page applications, che ha come scopo quello di produrre un unico componente front-end, non necessita di ulteriori rotte, al di fuori di quella, che ci permetterà di visualizzarlo. La rotta definita è quella in Figura 18.

```
Route::view('/', 'home')
        ->name('home');
```

Figura 18. Rotta dell'applicazione

Nello specifico la rotta in questione è stata definita tramite la facade *Route::* e l'helper (funzioni PHP globali) *view* che ritorna un'istanza di una vista chiamata *home.* La rotta in questione è stata nominata anch'essa *home* grazie all'helper *name*. Pertanto, all'atto della richiesta dell'utente, tramite il browser, all'URL seguente:

http://localhost/React/public/

l'applicazione ritornerà la vista contenuta nel file *home.blade.php.*

4.2.2 Viste

Come detto, la vista che fa riferimento alla rotta della nostra applicazione, prende il nome di *home* e si trova nella directory *resources/views.* All'interno

di *home,* abbiamo fatto uso di due direttive blade come si può notare in Figura 19:

```
@extends('layouts.app')

@section('content')

    <div id= "simulatore"> </div>

@endsection
```

Figura 19. Contenuto del file *home.blade.php*

- @extends: serve per far ereditare alla nostra vista il layout che prende il nome di *app*.
- @section: definisce una sezione denominata *content*, che verrà iniettata in un layout, la quale ha un semplice in cui noi concentreremo tutta la nostra applicazione.

Dunque, oltre alla vista, abbiamo creato anche un layout (*app.blade.php*), in cui troviamo tutto il codice html necessario per strutturare la nostra pagina web. In questo file abbiamo messo all'interno di <html>, un tag <*head*> in cui abbiamo collocato dei metadati che non vengono visualizzati dal browser: il titolo del documento, l'inclusione di script, font e fogli di stile. Fondamentale nella dichiarazione degli script, troviamo il file *js/app.js,* tramite il quale verranno richiesti i nostri componenti React. Un altro tag in *app.blade.php,* che merita di essere citato è <title>. Quest'ultimo è stato valorizzato grazie all'helper *config* nel seguente modo:

{{ config('app.name') }}

Le parentesi graffe servono per effettuare la cosiddetta sanificazione, ossia per evitare l'inserimento di codice malevolo all'interno dei tag html. Di seguito, l'helper andrà a prelevare dal file *app.php,* nella cartella *config,* il nome dell'applicazione nella variabile d'ambiente *"APP_NAME",* che poi verrà visualizzato nella barra della pagina nel browser. Essenziale nel layout è il tag

<body>, nel quale abbiamo messo una direttiva blade, come mostra la Figura 20.

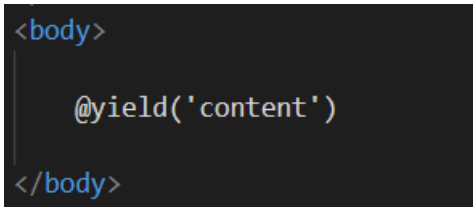

Figura 20. Tag <body> del layout contenuto in app.blade.php

Questa prende il nome di @yield e serve per visualizzare il contenuto della sezione *content,* che abbiamo definito nella vista *home,* nel body. In generale, le direttive usate sono utili nelle applicazioni che sono fornite di numerose viste, ma che condividono lo stesso layout. Nell'applicazione trattata, avendo una sola vista, non era necessario questo tipo di implementazione, ma è stata realizzata comunque in ottica di una futura espansione del software.

4.2.3 App.js

Come abbiamo detto nel file *resources/js/app.js*, vengono inclusi i componenti React della nostra applicazione (il codice JavaScript una volta compilato verrà in genere posizionato nella directory *public/js*). L'inclusione viene realizzata tramite l'istruzione JavaScript:

import 'path_componente';

Nel nostro caso è stato importato un unico componente, il quale fungerà da genitore per tutti gli altri, che prende il nome di *Simulatore*. In esso verranno iniettati tutti i componenti figli che comporranno la nostra interfaccia. Pertanto, nella fase iniziale in *app.js* importeremo solamente *Simulatore*, per poi successivamente includere i suoi figli nel momento in cui definiremo *Simulatore* stesso. Nel file si può notare anche, che vi è un ulteriore importazione:

import './bootstrap';

App.js caricherà anche il file *resources/js/bootstrap.js,* che in generale avvia e configura React, e tutte le altre dipendenze JavaScript.

4.2.4 Componenti React

Tutte le nostre componenti React verranno collocate in *resources/js/components.* Questa directory, per una migliore fruizione è stata organizzata in questo modo:

- Un file *Simulatore.js* in cui è stato definito il componente genitore *<Simulatore />*.
- Una cartella chiamata *pieces* che ospiterà tutti i componenti figli.

4.2.4.1 Simulatore

Come detto in precedenza, il componente genitore che incapsula tutti i suoi figli è <Simulatore />. Il metodo che permetterà la "renderizzazione" di Simulatore nel browser è:

ReactDOM.render(<Simulatore />, document.getElementById('simulatore'));

Esso è fornito dal package ReactDOM che è stato importato nel file in questione tramite l'istruzione JavaScript:

import ReactDOM from 'react-dom';

Grazie a *ReactDOM.render(),* il componente *<Simulatore />* verrà "renderizzato" nel della vista *home,* che aveva come *id* la stringa "simulatore". Entrando nello specifico di *Simulatore.js*, per definire il componente è stata definita una funzione *Simulatore()* preceduta dall'istruzione *export. Export* viene utilizzata durante la creazione di moduli JavaScript per esportare collegamenti diretti a funzioni, oggetti o valori primitivi dal modulo in modo che possano essere utilizzati da altre parti del programma tramite l'istruzione *import.* All'interno di *Simulatore()* troviamo una funzione *return()*,

la quale ritorna un con dentro una serie di componenti che ora verranno illustrati. Quelli che avranno uno scopo prettamente estetico sono <Grid /> e <Paper />. Nello specifico *Grid* è stato usato per una miglior disposizione dei componenti nella pagina del browser. É stato importato da una libreria esterna che va sotto il nome di *@material-ui/core/Grid*, e consente agli elementi di una pagina di condividere lo spazio disponibile in modo automatico. Con <Grid /> si può anche impostare la dimensione di un elemento e gli altri si ridimensioneranno automaticamente attorno ad esso[10]. La libreria dalla quale sono stati importati <Grid /> e <Paper>, è stata installata grazie al comando npm:

npm i @material-ui/core

L'effetto che si vuole ottenere nel progetto è quello mostrato in Figura 21:

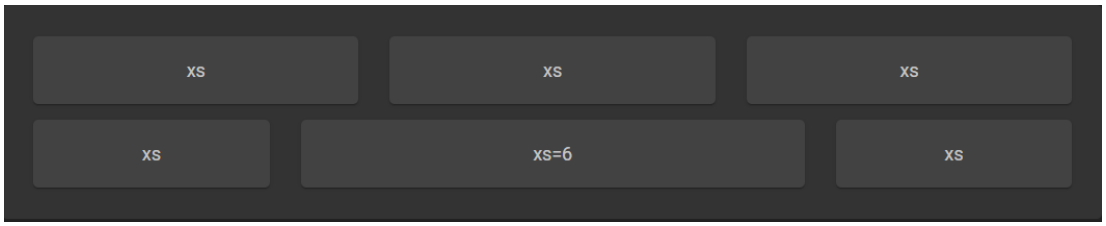

Figura 21. Disposizione usata nella nostra interfaccia suddivisa in sei componenti <Grid />

A livello di codice tale disposizione è stata realizzata tramite due <Grid container />. La props container indica che il nostro componente sarà un "contenitore" di altri elementi. Di questi contenitori ne sono stati disposti, uno superiormente e uno inferiormente. In ognuno di essi ci sono tre ulteriori <Grid item /> (in totale ci sono sei <Grid item />). Particolarità del <Grid item /> in basso e al centro è l'assegnazione della props xs={6} che permette così l'impostazione manuale della larghezza dell'elemento (gli altri <Grid item /> hanno xs con un valore di default). In ognuno di questi <Grid item /> è stato inserito un <Paper />; un componente prettamente estetico e che aiuta a marcare visivamente la disposizione dei componenti. Esso infatti visualizza, attorno all'oggetto contenuto, una sorta di "foglio", con la possibilità di specificare anche regole di stile come ad esempio colore, ombreggiatura ecc.…

Non a caso, ad entrambi i tipi di componenti sono stati associati degli stili tramite una funzione *makeStyle()*, anche essa importata da una libreria esterna chiamata *@material-ui/core/styles*. La funzione *makeStyle()* consente di creare delle regole di stile che poi verranno utilizzate da alcune delle componenti React del progetto. *MakeStyle()* in gergo prende il nome di hook: una speciale funzione che permette di utilizzare lo stato e altre funzionalità di React senza scrivere una classe. Nel caso in questione il foglio di stile è *theme*, il quale è già stato predefinito. *MakeStyle()* restituisce la funzione *useStyles()*, che viene quindi utilizzata per utilizzare gli stili. Dalla Figura 22, si nota che in essa ci sono due oggetti:

- *root*: a cui abbiamo associato una sola regola di stile ossia "*flexGrow: 3*". Questa regola css ci permette di impostare il fattore di crescita di un elemento flessibile.
- *paper*: in esso abbiamo messo svariate regole css:

  "padding: theme.spacing(2),
  textAlign: 'center',
  color: theme.palette.text.secondary,
  height: '93%' "

```
const useStyles = makeStyles((theme) => ({
    root: {
      flexGrow: 3,
    },
    paper: {
      padding: theme.spacing(2),
      textAlign: 'center',
      color: theme.palette.text.secondary,
      height: '93%',
    },

  }));
```

Figura 22. Funzione useStyles() in Simulatore.js

Successivamente all'interno della funzione che definisce *Simulatore* è stata inizializzata una variabile nel seguente modo:

const classes = useStyles();

Pertanto, le regole di stile verranno estratte da *classes* e saranno utilizzate come valore dell'attributo *className* di alcuni elementi. Nello specifico abbiamo usato queste regole in due occasioni:

- : il quale è stato usato come contenitore di tutto ciò che si trova all'interno del *return()* di *Simulatore.js.*
- <Paper className= "classes.paper" />: usato per dare le regole di stile ad ogni <Paper />.

Ultima passo è stato quello di mettere tutti i componenti del nostro Simulatore dentro ogni Paper. Alla fine, la gerarchia che si è creata all'interno del *return()* è la seguente: esternamente vi sono due <Grid container /> (superiore e inferiore) dentro i quali abbiamo tre <Grid item />. Ciascuno di quest'ultimi contiene un <Paper />, all'interno del quale c'è un componente React del sistema (Figura 23).

```
return (
    <div className={classes.root}>
  <Grid container spacing={3}>
    <Grid item xs>
      <Paper className={classes.paper}><p className="title" >Controller</p>
            <div><p>ON/OFF</p><Toggle  isOn={value} handleToggle={() => setValue(!value)} onColor="#3f51b5" /></div>
            <div><p>STEP</p><Select options={step}/> </div>
      </Paper>
    </Grid>

    <Grid item xs>
      <Paper className={classes.paper}><p className="title" >Diagram</p>
            <OrgChart tree={initechOrg} NodeComponent={MyNodeComponent} />
      </Paper>
    </Grid>

    <Grid item xs>
      <Paper className={classes.paper}><p className="title" > Messages</p>
            <Messaggi/>
      </Paper>
    </Grid>

  </Grid>
```

Figura 23. Parte della funzione *Return()* di Simulatore.js

4.2.4.2 Controller

Il componente Controller è composto da due elementi, racchiusi nel primo <Paper /> del Simulatore come illustra la Figura 24.

```
<Paper className={classes.paper}><p >Controller</p>
    <div><p>ON/OFF</p><Toggle  isOn={value} handleToggle={() => setValue(!value)} onColor="#3f51b5" /></div>
    <div><p>STEP</p><Select options={step}/> </div>
</Paper>
```

Figura 24. Componenti relativi al Controller.

Come si nota, ci sono due contenenti rispettivamente <Toggle /> e <Select />. Il primo è stato definito nel file *components/pieces/Toggle.js* e ha lo scopo di visualizzare il bottone On/Off citato nel capitolo 2.2.1. <Toggle /> nella Figura 24 inoltre ricevere tre props:

- isOn: prop che ha lo scopo di tenere traccia se il bottone è acceso o spento. Essa viene valorizzata con una variabile booleana *value.* Questa variabile è stata inizializzata all'inizio della funzione Simulatore, tramite un hook: *useState()* visibile in Figura 25.

```
function Simulatore() {

    const [value, setValue] = useState(false);
```

Figura 25. Uso dell'Hook *useState()* per valorizzare la variabile *value*

UseState() è un hook che permette di aggiungere lo stato nei componenti React dichiarati attraverso una funzione. Quest'ultimo è stato importato tramite l'istruzione JavaScript *import,* all'inizio del file. Nel dettaglio, quello che succede è che *useState()* dichiara una variabile stato (*value*), che non deve essere un oggetto (a differenza del caso in cui abbiamo una classe). La *useState()* ritorna la coppia di valori: *value* e *setValue,* la funzione che aggiornerà la variabile. Questa scelta implementativa è stata fatta perché inizialmente Simulatore è stato definito con una funzione. Non essendo una classe quindi, il

classico metodo *setState()* del capitolo 3.2.3.3, non può essere implementato (non possiamo accedere a *this)* [11]. Le altre due props di <Toggle /> sono:

- handleToggle: servirà per gestire l'evento dell'accensione e dello spegnimento, in modo da far cambiare il colore di sfondo di entrambi i bottoni. Viene valorizzata con la funzione *setValue()* citata precedentemente, alla quale in ingresso viene dato *{!value}*.
- onColor: inizializzata con il colore di sfondo del bottone in stato di On.

Osservando il file Toggle.js, si può notare il codice relativo al componente *<Toggle />*, definito da una funzione, che come è stato anticipato, prende in input i tre parametri suddetti. La funzione ritornerà al chiamante: un <input> e una <label> contenente uno . Tramite fogli di stile importati esternamente, questi elementi prenderanno l'aspetto grafico desiderato e l'animazione per la transizione dello stato del bottone. Riguardo la gestione del cambio di colore, durante l'accensione e lo spegnimento, la sua implementazione è avvenuta nel seguente modo:

```
<input checked={isOn} onChange={handleToggle}
className="react-switch-checkbox" id={`react-switch-new`} type="checkbox"/>

<label style={{ background: isOn && onColor }}
className="react-switch-label" htmlFor={`react-switch-new`} >
  <span className={`react-switch-button`} />
</label>
```

Figura 26. Elementi ritornati dal componente in *Toggle.js*

Osservando la Figura 26, che visualizza gli elementi all'interno del *return()* della funzione *Toggle(),* si nota che <input>, di tipo checkbox (checked = false inizialmente), ha la funzione *handleToggle()* come valore di *onChange.* In sostanza cliccando il bottone, selezioniamo e deselezioniamo un checkbox che viene nascosto tramite la regola di stile *visibility:hidden* contenuta in toggle.css (che analizzeremo in seguito). Al cambio di valore viene attivata la funzione passata nell'inizializzazione del <Toggle />, *setValue({!value}),* la quale assegna alla variabile di stato *value* il suo valore negato, che a sua volta verrà

mandato in *isOn*. Tramite questi cambi di valore, dando le due variabili *isOn* e *onColor* all'attributo di stile *background* di <label>, verrà visualizzato il colore definito in *onColor* solo se *isOn* sarà impostata a *true*. Ultimo aspetto da citare nel componente <Toggle /> è l'istruzione:

export default Toggle;

Questa istruzione serve durante la creazione di moduli JavaScript per esportare associazioni dirette a funzioni, oggetti o valori primitivi del modulo, in modo che possano essere utilizzati da altre parti dell'applicazione con l'istruzione *import* [12]. In *Simulatore,js* abbiamo fatto questo per permettere di inizializzare <Toggle />:

import Toggle from './pieces/Toggle.js'

Secondo ed ultimo componente del Controller usato nel primo <Paper /> del *Simulatore* è <Select /> . Questo è stato importato direttamente nel file Simulatore.js dalla libreria 'react-select', la quale è stata installata grazie al comando di npm:

npm i react-select

Il componente Select riceve come unica prop *options*. È facile intuire che in options andranno le opzioni che la nostra select mostrerà in fase di visualizzazione. Nello specifico abbiamo valorizzato *options* con la variabile *step. Step* non è altro che un array di oggetti già inizializzati, dove per ogni oggetto abbiamo due proprietà:

- *label*: in cui metteremo ciò che vorremmo visualizzare sulla singola opzione della select.
- *value*: il valore che verrà associato ad ognuna di queste opzioni.

Come è stato detto più volte, non essendo ancora definito il back-end dell'applicazione, è stato deciso di valorizzate *label* e *value* con un valore identico (una semplice stringa) per ogni opzione. Ogni volta che verrà fatto un click su un'opzione, la libreria assegnerà alla variabile predefinita *value* (associata sia alle props che allo stato), l'oggetto selezionato:

value: {label: "Step1", value: "Step1"}

4.2.4.3 Subsystems

Il secondo componente del sistema che è stato sviluppato è <Subsystems />, che come al solito è stato importato in *Simulatore.js* con il comando *import*. Essendo che nella finestra di visualizzazione del browser, il sottosistema *Subsytems* deve essere disposto sotto il Controller, nel file *Simulatore.js*, esso è stato collocato all'interno del <Grid container /> inferiore, per poi essere messo all'interno di un ulteriore <Grid item /> e un <Paper />. Il file in cui è stato definito è *components/pieces/Subsystems.js* e ora verrà analizzato. Questo componente è stato realizzato ad-hoc per l'interfaccia. Esso è costituito da una funzione *Switches()* e una classe *Subsystems*. La funzione *Switches()* ha lo scopo di ritornare quattro bottoni On/Off, che dovranno svolgere la stessa funzionalità di <Toggle /> del sottosistema Controller. Per differenziare esteticamente questi bottoni da quello del Controller, è stato importato un nuovo componente <Switch /> dalla libreria @material-ui/core/Switch. Pertanto, la funzione *Switches* restituirà in un *return()* quattro <Switch /> , nei quali abbiamo "passato" alcune props come viene mostrato in Figura 27.

```
SHA0 <Switch
checked={state.checkedA}
onChange={handleChange}
color="primary"
name="checkedA"
disabled={disabledComp}
/>
```

Figura 27. Uno dei quattro <Switch /> nel *return()* dalla funzione *Switches()*

Le props in questione sono:

- *checked*: valorizzata nel primo <Switch /> con la variabile di stato *checkedA*. Essa è definita in un hook che inizializza quattro variabili di stato booleane: *checkedA, checkedB, checkedC* e *checkedD* tramite *useState()*. Per aggiornarle durante l'uso dell'applicazione si userà la funzione *setState()* (Figura 28).

```
const [state, setState] = React.useState({
  checkedA: true,
  checkedB: true,
  checkedC: false,
  checkedD: false,
 });
```

Figura 28. Inizializzazione degli elementi di Switches

Nella pratica queste variabili servono preliminarmente per definire se il bottone sarà accesso o spento al lancio dell'applicazione. Verranno anche utilizzate per cambiarne il valore durante l'interazione con l'interfaccia. Le altre tre props ricevute dalle variabili di stato *checkedA, checkedB, checkedC* e *checkedD* sono:

- *name*: stringa valorizzata con il nome della variabile di stato associata al bottone (es: name="checkedX") perché ci ritornerà utile durante la fase di gestione del click sul bottone.
- *onChange*: prop per la gestione del cambio di valore del bottone, alla quale abbiamo associato la funzione *handleChange* (Figura 29). La *handleChange* prende in ingresso l'evento generato dal cambio di valore del bottone, così che nella variabile *event.target* ci sia il riferimento allo <Switch /> che è stato cliccato. Il principio di funzionamento è simile a quello usato per il toggle del Controller. Dunque, il singolo <Switch /> nasconde un checkbox che al momento del click determina il cambiamento di *event.target*.checked. Pertanto, richiamando la *setState()* viene cambiato il valore della variabile di stato *checkedX*

relativa al bottone *X* (che coincide con il name dello <Switch/>, quindi con *event.target.name*).

```
const handleChange = (event) => {
  setState({[event.target.name]: event.target.checked });
};
```

Figura 29. Funzione handleChange in Switches per cambiare il valore delle variabili di stato *checkedX*

- *disabled*: prop predefinita della libreria che ci servirà per disabilitare o abilitare gli <Switch /> in base alla modalità cliccata ("Set" o "Show") come citato nel capitolo 2.2.2. A *disabled* è stato dato il valore della variabile globale *disableComp* inizialmente a *false*. Questo serve per abilitare tutti gli <Switch /> al primo caricamento della pagina. Successivamente verrà illustrato come, cliccando il bottone "Show" *disableComp* verrà messo a *true* disabilitando i quattro <Switch /> e le due <Select />. La variabile *disableComp* è globale perché verrà poi usata dalla classe *Subsystems.*

Prima di entrare nello specifico della classe *Subsystems* è utile osservare cosa ritornerà tale classe al *Simulatore.* Preliminarmente bisogna far presente che per definire un componente React, tramite una classe, bisogna estendere la classe *React.Component.* Questa modalità di definire un componente obbliga l'uso di *render()* all'interno della classe. Questo metodo ha il compito di esaminare sia *this.state* che *this.props,* e restituire elementi React . Nel nostro caso nel metodo *render()* abbiamo messo un *return()*. Come si può notare dalla Figura 30, ciò che verrà ritornato, sarà immagazzinato in un <Grid />. Questo semplicemente per disporre meglio i vari elementi del componente sul <Paper /> del *Simulatore*. Proseguendo ed evitando i tag html, si notano cinque componenti React:

```
return (
  <Grid>
  <p>Subsystems</p>
  <div>
    <Button text = "Set" style= {this.state.style[0]}
      onChangeStyle={this.handleButtonS1}/>

    <Button text = "Show" style= {this.state.style[1]}
      onChangeStyle={this.handleButtonS2}/>

    <hr></hr>
    FCU0 <Select options={options1} isDisabled={disabledComp} />
    <br></br>
    Win0 <Select options={options1}  isDisabled={disabledComp} />

    <Switches/>

  </div>
  </Grid>
);
```

Figura 30. *Return()* all'interno di *render()* della classe *Subsystems*

- 2 <Button /> : rispettivamente associati alla modalità "Set" e "Show". Si può notare che i <Button /> ricevono come props il testo che verrà visualizzato all'interno del bottone (*text*) e lo stile che prenderanno da un array dello stato (*style*). Infine, entrambi ricevono la prop *OnChangeStyle,* alla quale sono associati rispettivamente i metodi che cambieranno i colori dei bottoni al click e disabiliteranno o abiliteranno gli elementi sottostanti (due <Select /> e i quattro <Switches />).
- 2 <Select /> : stesse <Select /> utilizzate dal Controller; alle quali abbiamo "passato" le solite opzioni e una prop predefinita dalla libreria: *isDisabled.* Quest'ultima, come abbiamo detto in precedenza, permette di disabilitare un componente se messa a true (abilitarlo a false o non dichiarandola).
- <Switches /> : il componente descritto dalla funzione *Switches()*, che abbiamo trattato all'inizio del capitolo. Esso restituirà quattro <Switch />.

Di queste componenti ritornate dalla classe *Subsystems* non abbiamo illustrato ancora nel dettaglio <Button />. <Button /> è stato concepito interamente da zero e senza l'uso di librerie esterne. Esso è definito nel file *components/pieces/Button.js* in cui vi è un'unica classe *Button* (vale lo stesso discorso delle classi React in *Subsystems*). <Button /> essendo un componente figlio di *Subsystems,* erediterà le variabili di stato di *Subsystems.* Anche in questo caso abbiamo il solito metodo *render()*, all'interno del quale inizializziamo due const JavaScript: *style* e *text,* associandole rispettivamente alle props *style* e *text* (Figura 31). Queste props sono state inizializzate in *Subsystems* al momento dell'introduzione dei <Button /> come abbiamo visto in Figura 30.

```
constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
}

handleChange(){
  this.props.onChangeStyle();
}

render() {
  const style = this.props.style;
  const text = this.props.text;

  return (  <button className={style}
            onClick={this.handleChange}>  {text} </button>);
}
}
```

Figura 31. Contenuto della classe *Button*

Infine, la classe nel *return()* ritorna un semplice <button> html, il quale ha come attributi:

- *className*: con il valore della const *style* (quindi della prop *style*)
- *onClick:* valorizzato con la funzione handleChange, che a sua volta fa riferimento alla prop *onChangeStyle* che abbiamo visto in Figura 30. Le *onChangeStyle* dei due <Button />, dichiarati da *Subsystems,* lanceranno

rispettivamente le funzioni *handleButtonS1* e *handeButtonS2* definite sempre in *Subsystems* (Figura 33).

Ritornando alla classe *Subsystems,* si può notare la presenza di due variabili di stato, dichiarate e inizializzate nel costruttore. Questo ha evitato l'utilizzo degli hooks come è avvenuto nelle precedenti componenti. Le variabili di stato si chiamano *modalità e style* e sono state definite come mostra la Figura 32 tramite *this.state*.

```
this.state = {
                modalità: "Set",
                style: ["buttonOn" , "buttonOff"],
               };
}
```

Figura 32. Dichiarazione delle variabili di stato del componente Subsystems

La variabile *modalità* servirà per tenere traccia della modalità corrente di *Subsystems* (essendo inizializzata con "Set" al primo caricamento della pagina gli elementi sottostanti devono essere utilizzabili). *Style* invece è un array composto da due stringhe, grazie al quale potremo associare due classi di stile differenti a ciascun bottone (contenute in *style.css*) :

- *buttonOn*: che da uno sfondo blu al nostro bottone.
- *buttonOff*: metterà a *null* il colore dello sfondo.

Queste variabili sono state usate principalmente all'interno delle funzioni *handleButtonS1() e handleButtonS2().* Essendo le funzioni identiche, eccezion fatta per gli argomenti, ci si concentrerà principalmente su una delle due. La *handleButtonS1()* di Figura 33, viene lanciata al click del bottone "Set". Inizialmente, verifica tramite un if, se la variabile di stato *modalità* sia diversa dalla stringa "Set". Questo perché, nel caso in cui il bottone "Set" fosse attivo, il click sullo stesso non dovrebbe comportare nessun cambiamento. Ma se il bottone acceso è quello associato a "Show" si entra nel corpo della funzione.

Per prima cosa viene dichiarata ed inizializzata una const *style*; ad essa abbiamo dato il valore dell'array di stato *style.* Subito dopo, ci sono due operazioni:

- style[0]= "buttonOn"
- style[1]= "buttonOff"

Questo servirà per il cambiamento dello sfondo del bottone "Set", mettendolo blu. Quest'ultimo, infatti, ritornando sulla Figura 30 fa riferimento a *this.state.style[0]* per l'attributo *style*. Il bottone "Show" invece prenderà le regole di stile associate a *buttonOff* quindi avrà uno sfondo incolore. Per attivare questo cambiamento però sarà necessaria la *this.setState()*. All'interno di essa prima cambieremo *modalità* e poi assegneremo come nuovo valore di *style* il nuovo array creato appositamente nel corpo della funzione. Ultimo passo è quello di mettere a false la variabile *disableComp*, che come abbiamo visto valorizzerà le props *isDisabled e disabled,* rispettivamente delle <Select /> e degli <Switch />, permettendone così l'abilitazione (visto che partivamo dalla situazione in cui *modalità*= "Show").

```
handleButtonS1(){
  if(this.state.modalità != "Set"){
      const style = this.state.style;
      style[0] = 'buttonOn';
      style[1] = 'buttonOff';
      disabledComp = false;
    this.setState({modalità: "Set" , style : style
                  });
  }

}
```

Figura 33. Funzione *handleButtonS1()* nella classe *Subsystems* in *Subsystems.js*

Molto sinteticamente *handleButtonS2()* invece prima verificherà se la modalità corrente è "Show". In questo caso effettuerà la creazione dell'array invertendone i valori rispetto *handleButtonS1()* e poi imposterà a *true*

disableComp. Ancora una volta verrà richiamata la *this.setState()* per mettere *modalità* a "Show" e riaggiornare *style*.

4.2.4.4 Diagram

Nel secondo <Grid item /> del primo <Grid container /> di *Simulatore.js,* è contenuto all'interno del solito <Paper /> il componente che rappresenta il sottosistema Diagram. L'implementazione di quest'ultimo è stata fatta utilizzando la libreria *'react-orgchart',* installata sempre tramite npm:

npm i react-orgchart

Il componente importato prende il nome di <OrgChart />. Come si osserva in Figura 34, nella sua dichiarazione, contenuta nella classe Simulatore, ha ricevuto i valori per due props predefinite nel componente:

```
<Grid item xs>
  <Paper className={classes.paper}><p>Diagram</p>
          <OrgChart tree={initechOrg} NodeComponent={MyNodeComponent} />
  </Paper>
</Grid>
```

Figura 34. Componente <OrgChart /> in *Simulatore.js*

- *tree*: prop nella quale passeremo un array di oggetti. Esso ci permetterà di definire la gerarchia dei nodi raffigurati nell'organigramma e i relativi attributi.
- *NodeComponent:* prop che definisce l'informazione associata a ciascun nodo. Essa prende in input una funzione che si chiama *MyNodeComponent* che verrà illustrata in seguito.

La variabile che prende in ingresso *tree* si chiama *initechOrg,* ed è stata dichiarata all'interno della funzione di *Simulatore,* come *MyNodeComponent.* Come preannunciato, è un array di oggetti che contiene tre proprietà per ogni nodo:

- *name:* prende in input una stringa, che in seguito verrà visualizzata al centro del riquadro rappresentante il nodo.
- *description:* una breve descrizione che sarà utile all'utente una volta cliccato il nodo per leggerne delle informazioni.
- *children:* in essa vengono specificati i figli del nodo su cui è definita. Dunque, riceve a sua volta un array di oggetti con le stesse proprietà appena citate per ogni figlio.

Nel nostro caso *initechOrg* è quella rappresentata in Figura 35.

```
const initechOrg = {
    name:"KPI",
    description: "KPI1 information",
    children: [
      {
        name: "KPI1",
        description: "KPI1 information",
        children: [
          {
            name: "KPI1.1",
            description: "KPI1.1 information"
          } ,  {
            name: "KPI1.2",
            description: "KPI1.2 information"
          }

        ]
      },
      {
        name: "KPI2",
        description: "KPI2 information"
      },
      {
        name: "KPI3",
        description: "KPI3 information"
      },
    ]
  };
```

Figura 35. Gerarchia e attributi dei nodi di <OrgChart /> in *initechOrg*

Rimane da spiegare la prop *NodeComponent.* Per fare ciò verrà illustrata la funzione che prende come ingresso: *MyNodeComponent.* Questa funzione prende in ingresso l'oggetto predefinito dalla libreria *{node}*, grazie al quale si possono estrarre tutti gli attributi che abbiamo dichiarato in *initechOrg.* Come primo passo, la funzione ha un *return()* nel quale ritorna un . Il tag contiene all'interno *{node.name}*, dunque nel riquadro di ogni nodo ci sarà il *name* relativo. Come attributi troviamo:

- *id:* ci servirà per identificare il nodo nella funzione di gestione per l'evento click (assegnato il valore di {*node.name*}).
- *className:* ad essa abbiamo dato una classe di stile, per far sì che il nostro prenda la forma di un nodo di un organigramma (uguagliata a "initechNode" definita in *graph.css* dove per altro ci sono le altre regole di stile dell'organigramma).
- *onClick:* inizializzata con la funzione *HandleNode* che è descritta in seguito.

La funzione *HandleNode* si trova anch'essa all'interno di *MyNodeComponent.* Essa come prima cosa lancia un *alert()* JavaScript, grazie al quale l'utente può visualizzare le informazioni del nodo cliccato ({*node.description*}). In seguito, esegue un'istruzione JQuery che seleziona tutti i con className="*initechNodeClick*":

$('div.initechNodeClick').removeClass('initechNodeClick');

La classe di stile appena citata è definita in *graph.css*. L'unica regola di *initechNodeClick* fa riferimento all'ultimo nodo cliccato, al quale viene impostato il colore di sfondo blu. Pertanto, il selettore JQuery prende il nodo che era stato selezionato nel precedente click. A quest'ultimo esegue la funzione *removeClass("initechNodeClick")*, che rimuoverà la classe di stile. Infine, verrà prelevato l'elemento cliccato tramite l'*id* del (grazie alla *document.getElementById()*), e gli verrà aggiunta la classe "*initechNodeClick*". Tutto questo è raffigurato in Figura 36.

```
const MyNodeComponent = ({node}) => {
  return (
    <div id={node.name} className="initechNode" onClick={HandleNode}>{ node.name }</div>
    );

  function HandleNode(){
    alert(node.description);
    $('div.initechNodeClick').removeClass('initechNodeClick');
    document.getElementById(node.name).classList.add('initechNodeClick');
  };
};
```

Figura 36. Funzione *MyNodeComponent,* utilizzata da <OrgChart />, in *Simulatore.js*

4.2.4.5 KPI

Il componente relativo al KPI è stato messo nella <Grid xs={6}/> all'interno del secondo <Grid container />, quindi in basso e al centro nella disposizione di Figura 21. Anche <KPI /> come gli altri è stato messo all'interno di un <Paper />. Il componente è stato definito nel file *components/pieces/KPI.js,* attraverso una classe. Nel costruttore della classe abbiamo inizializzato lo stato del componente con la sola variabile *selectedOption* (Figura 37).

```
constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
  this.state = {
  selectedOption: { value: 0},
};
}
```

Figura 37. Costruttore della classe KPI dentro *components/pieces/KPI.js*

Quest'ultima è un array di oggetti con una sola proprietà (poteva essere usata una sola variabile numerica, ma questa scelta rende più intuitivo il meccanismo):

- *value*: in questa proprietà, verrà definito il valore visualizzato nei tachimetri.

Andando direttamente nel metodo *render()* della classe, troviamo due variabili: *selectedOptionCurrentValue* e *selectedOptionExpectedValue.* Le variabili prendono rispettivamente, il *value* della variabile di stato, e sempre il *value* di *selectedOption* moltiplicato per 2 (Figura 38):

```
const  selectedOptionCurrentValue  = this.state.selectedOption.value;
const  selectedOptionExpectedValue  = (this.state.selectedOption.value)*2;
```

Figura 38. Dichiarazione e assegnamento di *selectedOptionCurrentValue* e *selectedOptionExpectedValue*

Non vi è stata nessuna ragione, inerente al sistema, dietro questa assegnazione a *selectedOptionCurrentValue* e *selectedOptionExpectedValue*. Infine, è stato collocato un *return()* contenente ciò che viene mostrato in Figura 39.

```
return (
  <div >
    <Select placeholder="Select KPI..." onChange={this.handleChange} options={options_kpi}/>
    <br></br><hr></hr>
    <Grid container>
                            
     <Paper elevation={0} ><p id="label">Current Value</p>
                <ReactSpeedometer  width={250} height={150} maxValue={500}  value={selectedOptionCurrentValue}
                needleColor="Orange" startColor="red" segments={5} endColor="green" />
     </Paper>
           
     <Paper elevation={0} ><p id="label">Expected Value</p>
                <ReactSpeedometer  width={250} height={150} maxValue={500}  value={selectedOptionExpectedValue}
                needleColor="Orange" startColor="red" segments={5} endColor="green" />
     </Paper>
    </Grid>
  </div>
);
```

Figura 39. *Return()* della classe KPI in *KPI.js*

Dentro un abbiamo messo tutte le parti che comporranno il nostro <*KPI*/>. Il primo oggetto è una <*Select* />, analoga a quelle già viste in precedenza. Le sue props sono:

- *options*: predefinita, a cui abbiamo passato *options_kpi* ossia un array di oggetti con le proprietà *label* e *value* (Figura 40). La proprietà *label* definisce un'etichetta che verrà visualizzata nella <Select /> sopra al tachimetro. *Value* invece conterrà il valore associato ad ogni opzione della <Select />.

```
const options_kpi = [
  { label: 'KPI1', value: 50 },
  { label: 'KPI2', value: 100 },
  { label: 'KPI3', value: 200 },
];
```

Figura 40. Dichiarazione e inizializzazione di *options_kpi* nella classe KPI

- *onChange*: solita prop predefinita, associata al cambio di valore della <Select />, posta uguale alla funzione *handleChange* della classe che tramite il metodo *this.setState(),* aggiorna la variabile di stato *selectedOption* ad ogni cambio di valore nella <Select /> (Figura 36).

```
handleChange = selectedOption => {

  this.setState({ selectedOption });

};
```

Figura 41. Funzione *handleChange()* della classe KPI in *KPI.js*

Di seguito alla <Select /> troviamo una <Grid container /> nella quale metteremo i nostri tachimetri. La prop *container* è stata messa esclusivamente per permettere di affiancare i due tachimetri. Questi due tachimetri a loro volta sono stati messi ciascuno dentro un apposito <Paper elevation={0} />. In questo caso, a <Paper /> è stata passata la prop elevation={0} per non visualizzarne il bordo, che troviamo evidente nella configurazione di default. Dentro ogni <Paper elevation={0} />, abbiamo messo il componente <ReactSpeedometer /> importato dalla libreria esterna 'react-d3-speedometer' installata con il solito comando:

npm i react-d3-speedometer

Al primo <ReactSpeedometer /> abbiamo passato delle props, evidenziate sempre in Figura 39:

- *maxValue:* il valore massimo a cui può arrivare il tachimetro.
- *value:* il valore corrente sui cui viene posizionata la lancetta. Ad esso abbiamo dato il valore *{selectedOptionCurrentValue}*. In questo modo avremo il valore dell'opzione selezionata nella <Select />, avendolo preso dalla variabile di stato.
- *startColor:* colore del primo segmento del tachimetro (impostato a verde).
- *endColor:* colore dell'ultimo segmento del tachimetro (impostato a rosso).

- *needleColor:* colore del segmento centrale del tachimetro (impostato ad arancione). In base ad esso, a *startColor* e *endColor,* viene creata una scala cromatica in relazione ai segmenti.
- *segments:* numero di segmenti visualizzati nel tachimetro. Nel nostro caso abbiamo dato cinque segmenti.
- *weight e height:* usati per ridimensionare i tachimetri.

Per quanto riguarda il secondo <ReactSpeedometer />, le props sono le stesse. Unica differenza è nella valorizzazione di *value.* Ad esso è stato assegnato *{selectedOptionExpectedValue}.* Perciò, il tachimetro di destra sarà impostato sempre al valore doppio di quello di sinistra.

4.2.4.6 History

A fianco di KPI troviamo il componente relativo al componente History. Anche in questa circostanza, nel *return()* della funzione *Simulatore()* in *Simulatore.js,* abbiamo disposto il componente <History /> in un <Paper />, il quale a sua volta è contenuto nell'ultimo <Grid item /> del secondo <Grid container />. <History /> è stato sviluppato all'interno del file *components/pieces/History.js.* Anche questo componente è associato ad una classe, ovviamente preceduta dall'istruzione *export* per permetterne l'importazione in *Simulatore.js* (procedura eseguita in tutte le classi con componenti React da esportare). Entrando nel corpo della classe History, troviamo un costruttore molto simile a quello descritto nel componente relativo a KPI. Dalla Figura 42 infatti si nota proprio questa somiglianza.

```
constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
  this.state = {
  selectedChart: chart_options[0]
}
}
```

Figura 42. Costruttore della classe History in *components/pieces/History.js*

Anche in questo caso abbiamo un'unica variabile di stato chiamata *selectedChart.* Questa variabile dovrà realizzare la stessa funzionalità implementata in KPI. In sostanza questa variabile permetterà, con l'ausilio di una <Select />, la visualizzazione di un grafico selezionato tra le varie opzioni. La *selectedChart* nella prima fase è inizializzata con il valore del primo elemento dell'array di oggetti *chart_options.* Quest'ultimo, eccezion fatta per i suoi valori, è identico nella forma a *options_kpi,* di cui abbiamo discusso nel capitolo 4.2.4.4. Entrando nel dettaglio di *chart_options,* troviamo le due solite proprietà (Figura 43):

- *label:* etichetta che verrà visualizzata nella <Select /> che verrà introdotta di seguito.
- *value:* valore associato ad ogni opzione della <Select />.

```
const chart_options = [
  { label: 'KPI1', value: KPI1 },
  { label: 'KPI2', value: KPI2 },
  { label: 'KPI3', value: KPI3 },
];
```

Figura 43. Array di oggetti *chart_options* definite in *History.js*

Da Figura 43 si nota che ad ognuna delle tre proprietà *value* è stata assegnata una variabile *KPIX*. Prendendo come esempio la prima, *KPI1* è stata definita all'inizio del file e anche essa è un array di oggetti. *KPI1*, come anche *KPI2* e *KPI3*, avranno lo scopo di contenere tutti quei dati che serviranno per tracciare la funzione nel piano. Nello specifico *KPI1* ha due proprietà. Dalla Figura 44 si nota che la prima prende il nome di *labels. Labels* è un array in cui sono state messe un determinato numero di etichette (pari al numero di punti su cui si baserà il grafico). Quest'ultime quindi, verranno visualizzate in ordine nell'asse delle ascisse del piano cartesiano che conterrà il grafico. La seconda proprietà invece è *datasets,* un array di oggetti. Questi parametri predefiniti sono stati trovati nella documentazione della libreria del componente <Line />: 'react-

chartjs-2'. Anche per importare questa libreria è stato usato il comando del gestore dei pacchetti npm:

npm i react-chartjs-2

```
const KPI1 = {
  labels: ['A', 'B', 'C', ],
  datasets: [
    {
      label: 'KPI1',
      fill: false,
      lineTension: 0.4,
      backgroundColor: 'rgba(75,192,192,0.4)',
      borderColor: 'rgba(75,192,192,1)',
      borderDash: [],
      borderJoinStyle: 'miter',
      pointBorderColor: 'rgba(75,192,192,1)',
      pointBackgroundColor: 'rgba(75,192,192,1)',
      pointBorderWidth: 1,
      pointHoverRadius: 5,
      pointHoverBackgroundColor: 'rgba(75,192,192,1)',
      pointHoverBorderColor: 'rgba(220,220,220,1)',
      pointHoverBorderWidth: 2,
      pointRadius: 1,
      pointHitRadius: 10,
      data: [65, 59, 80,]
    }
  ]
};
```

Figura 44. Variabile KPI1 definita in *History.js*

Le proprietà associate ad ogni *datasets* sono:

- *label*: etichetta che verrà visualizzata sopra il grafico, vicino ad un rettangolino dello stesso colore del grafico associato (citata nel capitolo 2.2.6)
- *fill*: se messo a *true* permette di colorare la parte di piano sottostante al grafico. Nel nostro caso è stato messo a false.
- *lineTension:* parametro per la tensione della curva. Se impostato da 0 ci permette di ottenere linee rette.
- *backgroundColor:* specifica il colore di riempimento della funzione.
- *borderColor:* specifica il colore della del bordo della funzione.

- *borderDash:* se valorizzato permette di tratteggiare la linea. Nel nostro caso non è stato specificato.
- *borderJoinStyle:* definisce lo stile con cui si uniscono le linee. Nel caso in questione, ad esso è stato dato il valore di 'miter'. Grazie a questo, i segmenti collegati vengono uniti estendendo i loro bordi esterni per connettersi in un unico punto, con l'effetto di riempire un'area aggiuntiva a forma di rombo.
- *pointBorderColor:* il colore del bordo dei punti.
- *pointBackgroundColor:* il colore del riempimento dei punti.
- *pointBorderWidth:* specifica la larghezza dei punti.
- *pointHoverRadius:* specifica il raggio dei punti quando si passa sopra ad essi con il puntatore.
- *pointHoverBorderColor:* colore dei punti quando si passa sopra ad essi con il puntatore.
- *pointHoverBorderWidth:* larghezza dei punti quando si passa sopra ad essi con il puntatore.
- *pointRadius:* raggio dei punti del grafico.
- *pointHitRadius:* la dimensione del raggio dell'area attorno al punto che intercetta l'evento onMouseOver.
- *data:* proprietà per passare il set di dati del grafico. Può supportare diversi formati, ma nel nostro esempio è stata utilizzata una matrice di numeri che identificano le ordinate dei punti.

L'ultima parte da citare nel componente <History /> è il suo metodo *render()*. In esso inizialmente è stata definita ed inizializzata una variabile *selectedChartValue*. Rifacendosi alla struttura di <KPI />, in *selectedChartValue* metteremo il valore della variabile di stato nel seguente modo:

const selectedChartValue= this.state.selectedChart.value;

Infine, avremo il *return()* di Figura 45 nel quale ritorneremo due componenti. Il primo sarà una classica <Select /> con le seguenti props:

- *onChange:* prop già citata nelle <Select /> dei precedenti componenti alla quale assegniamo la funzione *handleChange*. Quest'ultima è identica a quella di Figura 41, tranne che come argomento riceverà la variabile di stato *selectedChart.*
- *value:* al quale diamo il valore della variabile di stato *selectedChart*. Questo ci permette di avere preimpostato alla prima apertura di applicazione KPI1 nella <Select />.
- *options:* anche qua abbiamo la solita prop predefinita per le opzioni della <Select /> in cui abbiamo iniettato il valore dell'array di oggetti *chart_options*.

Il secondo invece sarà il componente importato dalla libreria <Line />. Ad esso è stato passata un'unica prop: *data*. Quest'ultima è una prop predefinita dalla libreria del componente la quale permette di inoltrare i dati necessari per rappresentare il grafico. Pertanto, *data* riceve in ingresso la variabile che abbiamo definito nel *render()*: *selectedChartValue.* Facendo così ogni volta che verrà selezionato uno delle tre opzioni della <Select /> in *data* finirà l'intero contenuto della variabile KPI1, KPI2 o KPI3. Ciò permetterà a <Line /> di prelevare i dati dalle proprietà: *labels* e *datasets.*

```
return (
  <div>
    <Select onChange={this.handleChange}
            value={this.state.selectedChart} options={chart_options} />

    <Line data={selectedChartValue} />
  </div>
);
```

Figura 45. *Return ()* della classe History in *History.js*

4.2.4.7 Messages

L'ultima parte dell'applicazione rimasta da commentare è Messages. Il componente relativo è <Messaggi /> ed è stato sviluppato nel file *components/pieces/Messaggi.js*. Essendo Messages disposto nella parte alta a destra dell'interfaccia, il componente è stato messo nel primo <Grid container /> di *Simulatore.js.* All'interno del primo <Grid container />, <Messaggi /> è stato inserito nel <Paper /> del terzo <Grid item />, allo stesso modo in cui abbiamo fatto per gli altri. In *Messaggi.js* sono state implementate due funzioni:

- *Messaggi()*
- *renderRow()*

La prima funzione è quella che definisce il componente <Messaggi /> che abbiamo utilizzato in Simulatore. Proprio per questo, allo stesso modo delle altre componenti, abbiamo utilizzato, nella sua definizione, la funzione con l'istruzione *export* per poterne permettere l'importazione in *Simulatore.js.*

```
export default function Messaggi() {
  return (
    <form>
      <FixedSizeList style={border} height={150} width="auto" itemSize={30} itemCount={10}>
        {renderRow}
      </FixedSizeList>

      <Button text = "Apply" style= "buttonMessages" />
    </form>
  );
}
```

Figura 46. Funzione *Messaggi()* in *components/pieces/Messaggi.js*

In *Messaggi()* è stato implementato un *return()* come in Figura 46 che contiene una semplice <form> html. Nella <form> non sono stati specificati attributi come *action*, visto che sarà compito di chi svilupperà il back-end implementarli. All'interno della form abbiamo introdotto due componenti React:

- *<FixedSizeList />*

- *<Button />*

<FixedSizeList /> è stata importata dalla libreria *'react-window',* installata con la solita procedura tramite terminale:

npm i react-window

<FixedSizeList /> realizza una lista di elementi, affiancati da una barra di scorrimento. Dalla documentazione del componente, si è osservato che *<FixedSizeList />* necessita delle seguenti props [13]:

- *height:* specifica l'altezza della lista. Nel caso delle liste orientate in verticale, questa prop deve essere valorizzata con un dato di tipo numerico. Essa, pertanto, influisce sul numero di righe che verranno renderizzate (e visualizzate) in un dato momento.
- *width*: definisce la larghezza della lista. Per le liste orientate in verticale, questa prop può essere di tipo numerico ma anche stringa (es: "50%"). Ovviamente questa non incide sul numero di righe in fase di visualizzazione. Per evitare problemi durante il ridimensionamento della finestra del browser, width è stata messa ad "auto".
- *itemSize*: indica la dimensione di un elemento nella direzione in cui viene visualizzata la finestra. Per gli elenchi verticali, questa è l'altezza di ciascuna riga. Facendo la divisione tra *height* e *itemSize,* si ottiene il numero di righe che vengono visualizzate dall'elenco alla volta.
- *itemCount*: rappresenta il numero totale di elementi nell'elenco. Da tener sempre presente è il fatto che verranno renderizzati e visualizzati solo alcuni elementi alla volta.

Una volta iniettate tutte queste props in *<FixedSizeList />* è stata inserita al suo interno una funzione denominata *renderRow(),* che sarà descritta in seguito. Concludendo, nella *return()* di *Messaggi()* si può notare l'ausilio del

componente <Button />. <Button /> è stato importato da *components/pieces/Button.js* tramite l'istruzione:

import {Button} from './Button.js'

Ad esso sono state date le due props:

- *text*: valorizzato con "Apply", semplicemente per visualizzare la stringa all'interno del bottone.
- *style*: valorizzato con "buttonMessages", una classe di stili css.

Passando alla funzione *renderRow()* si nota che questa in ingresso riceve delle *props*. Queste *props* vengono iniettate automaticamente dalla libreria. In Figura 47 , si nota l'inizializzazione delle *props* per il primo elemento della lista. Le proprietà che verranno utilizzate sono:

- *index*: per dare ad ogni elemento della lista un relativo indice.
- *style*: regole di stile associate ad ogni elemento della lista.

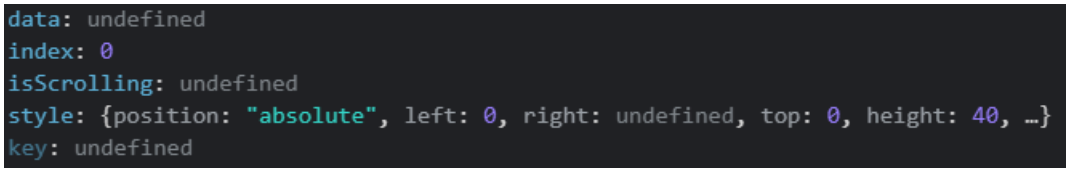

Figura 47. Definizione di *props* del primo elemento della lista

Di seguito, sempre in *renderRow()*, queste *props* sono state "estratte" tramite l'istruzione:

const { index, style } = props

Questa funzionalità viene chiamata assegnazione destrutturante, ed è un modo migliore per recuperare valori da un oggetto o da un array. In questo modo abbiamo creato due oggetti: *{ index, style }* nei quali verranno iniettate le omonime *props* di Figura 47. Andando avanti in *renderRow(),* è stato definito un array denominato *messages*. In *messages,* sono stati inizializzati un numero di elementi pari al valore di *itemCount*. Ognuno di questi contiene una stringa

che verrà poi visualizzata nella lista dei messaggi. Ultimo aspetto da citare nella *renderRow()* è la funzione di *return()* di Figura 48.

```
return (

  <ListItem button style={style} >
    <ListItemText primary={messages[index]}/> <input type="radio"  name="radio" id={index} />
  </ListItem>

);
```

Figura 48. *Return()* della funzione *renderRow()* in *Messaggi.js*

Return() ogni volta che viene lanciata ritorna un <ListItem />. Questo componente è stato importato nella parte iniziale del file:

import ListItem from '@material-ui/core/ListItem';

<ListItem /> rappresenta un elemento della lista e riceve due props: *button* e *style.* La prima, permette ad ogni elemento della lista di essere cliccabile (è stata introdotta per uno scopo esclusivamente grafico nel momento in cui si passa sopra l'elemento con il puntatore). *Style* invece è una prop, anch'essa predefinita della libreria, che è stata valorizzata con l'oggetto *style* discusso poco sopra. Questo permetterà ad ognuno degli elementi della lista di ricevere quelle regole di stile definite nella *props style* di Figura 47. Successivamente, all'interno del di <ListItem /> è stato messo il componente <ListItemText /> e un <input> html. Partendo dal primo, come nel caso precedente abbiamo effettuato l'importazione dalla libreria tramite l'istruzione JavaScript:

import ListItemText from '@material-ui/core/ListItemText';

<ListItemText /> come è facile intuire dal nome, permette di definire il contenuto testuale dei nostri elementi della lista. Tale contenuto viene inoltrato tramite la prop *primary.* Questa racchiude il contenuto principale dell'elemento, visto che si può definire anche un elemento secondario tramite la prop *secondary. Primary* nell'applicazione prende come valore *{messages[index]}*. In questo modo ogni elemento della lista con indice *index,* scorrerà l'array

messages e preleverà il contenuto relativo all'indice *index.* Questo permette di diversificare il contenuto di *primary* di ogni <ListItemText /> , dunque di ogni messaggio in ogni elemento della lista. Infine, come è stato accennato, dentro <ListItem /> è stato messo un:

<input type="radio" name="radio" id={index} />

Questo permetterà la visualizzazione di un *radio button* al fianco di ogni messaggio, per ogni elemento. Per garantire la mutua esclusione nella selezione dei bottoni, è stato attributo il medesimo *name* ad ogni bottone. Inoltre, è stato associato ad ogni botte un *id,* al quale è stato passato il valore di *{index}.*

4.3 Classi di stile

In più occasioni sono entrate in gioco, nei componenti React e non solo, delle regole di stile css. Nella nostro progetto abbiamo definito tre file css all'interno della directory *public/css:*

- graph.css
- style.css
- toggle.css

4.3.1 graph.css

All'interno del componente graph.css sono state messe tutte le regole css relative a <OrgChart />. Nel dettaglio le prime regole sono state prelevate dalla documentazione della libreria *'react-orgchart'* e sono quelle mostrate dalla Figura 49.

```
.reactOrgChart .orgNodeChildGroup .nodeGroupCell {
    vertical-align: top;
}

.reactOrgChart .orgNodeChildGroup .nodeGroupLineVerticalMiddle {
    height: 25px;
    width: 50%;
    border-right: 2px solid #000;

}

.reactOrgChart .nodeLineBorderTop {
    border-top: solid 2px #000;
}

.reactOrgChart table {
    border-collapse: collapse;
    border: none;
    margin: 0 auto;
}

.reactOrgChart td {
    padding: 0;
}

.reactOrgChart table.nodeLineTable {
    width: 100%;
}
```

Figura 49. Una parte delle regole di stile di graph.css

Evitando di descrivere il foglio di stile intero, si può dire che la logica dietro l'aspetto dell'organigramma è la seguente. La libreria forma l'intero organigramma da una normale tabella html alla quale sono stati "annullati" i bordi e i margini (ha come attributo class= "orgNodeChildGroup"). In sostanza, tra un nodo e suo figlio ci sono ben quattro righe dove: nella prima e nell'ultima verranno messi i nodi veri e propri (i del capitolo 4.2.4.3) mentre nelle altre due ci saranno le linee orizzontali e verticali per i collegamenti. Grazie anche all'ausilio dell'attributo *colspan,* ogni insieme di figli di un determinato nodo, viene racchiuso in un certo numero di colonne che la libreria ha prefissato per noi.

4.3.2 style.css

Style.css definisce le regole di stile per i bottoni di <Subsystems /> e <Messages /> (Figura 50). Quest'ultimo si occupa anche di ogni "titolo" dei componenti e delle etichette dei tachimetri.

```css
button {
    padding: 7px 9px;
    font-size: 16px;
    margin: 4px 2px;
    cursor: pointer;
    height: fit-content;
    width: fit-content;
    border-width: 2px;
  }

.buttonOn {
  background-color: #3f51b5;
  border-radius: 10px;
} /* Blue */

.buttonOff {
  background-color: null;
  border-radius: 10px;
}

.buttonMessages {
  background-color: null;
  border-radius: 10px;
  margin-top: 10px;

}
```

Figura 50. Regole css dei bottoni dell'applicazione in *style.css*

4.3.3 toggle.css

Anche questo foglio di stile è stato prelevato da una documentazione esterna relativa a <Toggle />. Come nel caso di graph.css anche qua toggle.css sarà illustrato in modo sintetico senza entrare troppo nel merito del codice. Il funzionamento del bottone deriva da un *<input type= "checkbox">* che però è stato nascosto da toggle.css. In seguito, la *<label>* comporrà la parte in sottofondo del bottone (quella colorata di grigio) attraverso uno **. Questo tramite delle regole, formerà un cerchio bianco al quale sarà associata anche la transizione orizzontale relativa all'accensione e allo spegnimento (Figura 51).

```css
.react-switch-checkbox {
  height: 100;
  width: 0;
  visibility: hidden;
}

.react-switch-label {
  display: flex;
  align-items: center;
  justify-content: space-between;
  cursor: pointer;
  width: 60px;
  height: 35px;
  background: grey;
  border-radius: 100px;
  position: relative;
  transition: background-color .2s;

}

.react-switch-label .react-switch-button {
  content: '';
  position: absolute;
  top: 2px;
  left: 2px;
  width: 35px;
  height: 30px;
  border-radius: 45px;
  transition: 0.2s;
  background: #fff;
  box-shadow: 0 0 2px 0 rgba(10, 10, 10, 0.29);
}

.react-switch-checkbox:checked +
.react-switch-label .react-switch-button {
  left: calc(100% - 2px);
  transform: translateX(-100%);

}

.react-switch-label:active .react-switch-button {
  width: 60px;
}
```

Figura 51. Regole di stile di *toggle.css*

4.4 Testing dell'applicazione

Tutto la parte di sviluppo dell'applicazione ovviamente è stata affiancata da numerose operazioni di verifica. Nello specifico è stata usata molto frequentemente l'estensione di Chrome: React Developer Tools (descritta nel capitolo 3.2.3.5). Grazie ad essa è stato possibile verificare le variabili di stato e le props dei componenti, ogni qual volta sorgeva un problema legato al loro contenuto o ad altri aspetti. Un'altra operazione che è stata utile relativamente alla fase di debug è l'attivazione del metodo *console.log()*. Come nel caso di React Developer Tools, questo metodo è stato usato per la verifica di variabili che però non sono inerenti allo stato o alle props. Ovviamente, sono stati usati anche gli strumenti per sviluppatori forniti dal browser, per quanto riguarda codice html, css ecc….

CAPITOLO 5:
CONCLUSIONI E SVILUPPI FUTURI

5.1 Risultato ottenuto e possibili migliorie

Alla fine del processo di sviluppo il risultato ottenuto è molto fedele a ciò che è stato richiesto, così come le funzionalità demandate ai componenti, le quali sono tutte funzionanti. La disposizione dei componenti tramite <Grid /> e <Paper /> ha ottenuto il risultato desiderato, forse anche sopra le aspettative. L'interfaccia realizzata è quella che si osserva in Figura 52.

Figura 52. Risultato finale dell'applicazione

Come citato più volte l'ausilio di librerie esterne è stato fondamentale in alcune situazioni. Tuttavia, questo uso di librerie non deve essere un aspetto negativo in merito al contributo effettivo del programmatore. Ormai la maggior parte delle applicazioni sfrutta elementi che sono già stati creati in precedenza per poi adattarli alle proprie esigenze. Questo permette vantaggi per quanto riguarda il

tempo impiegato ma anche per i risvolti sul prodotto finale. Nonostante ciò, componenti come <Subsystems /> o altri elementi, sono stati creati totalmente da zero. Questo ha avuto il vantaggio di permettere anche un miglior apprendimento di React. Naturalmente la soluzione proposta certamente può essere migliorata. Come prima cosa, è completamente assente il back-end, dunque, questo è un aspetto che sicuramente dovrà essere sviluppato nel futuro. Ciò permetterebbe l'implementazione di vere e proprie funzionalità all'interno dell'applicazione ma anche un interfacciamento con una base di dati. Una funzionalità inerente al back-end può essere quella riguardante Messages. Come è stato detto all'interno della tesi, Messages deve poter comportare delle conseguenze sui tachimetri di KPI, in base al messaggio ma anche dal fatto se viene selezionato o applicato. Anche riguardo al front-end, ci potrebbero essere delle migliorie da poter implementare. Un esempio potrebbe essere l'alert che compare al click di un nodo in Diagram. Si potrebbe creare un ulteriore componente React per creare una sorta di pop-up che sicuramente avrebbe un effetto visivo molto più piacevole. Oltre a questo, si potrebbero effettuare altre piccole modifiche anche se si può dire che il risultato ottenuto è di buona fattura. Tramite lo sviluppo di questo progetto ho avuto modo di apprendere un nuova tecnologia come React. Contemporaneamente, mi sono confrontato con tecnologie che avevo già incontrato durante gli studi universitari come HTML, Javascript e CSS ma con le quali ho acquisito una maggiore dimestichezza.

BIBLIOGRAFIA

[1] Mary Ercolini - "Introduzione alle Applicazioni Web" consultato a luglio 2021 *http://www.isa.cnr.it/dacierno/tesipdf/pomidoro.pdf*

[2] https://www.dotitsrl.it/sviluppo-siti-web/front-end-back-end-differenze/ consultato ad agosto 2021

[3] Andrea Azteni e Marco Vallini, Politecnico di Torino – "Introduzione a XAMPP" *https://security.polito.it/~lioy/01nbe/xampp.pdf*

[4] Università degli Studi di Padova – "Design Pattern Model-View-Controller" *https://www.math.unipd.it/~rcardin/pdf/Design%20Pattern%20-%20Model%20View%20Controller_4x4.pdf*

[5] *https://laravel.com/docs/8.x/structure#introduction* consultato ad agosto 2021

[6] *https://www.w3.org/TR/WD-DOM/introduction.html* consultato ad agosto 2021

[7] *https://it.reactjs.org/* consultato ad agosto 2021

[8] *https://laravel.com/docs/8.x/mix* consultato ad agosto 2021

[9] *https://docs.npmjs.com/about-npm* consultato ad agosto 2021

[10] *https://material-ui.com/components/grid/* consultato ad agosto 2021

[11] *https://it.reactjs.org/docs/hooks-state.html* consultato ad agosto 2021

[12] *https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Statements/export?retiredLocale=it* consultato ad agosto 2021

[13] *https://react-window.vercel.app/#/api/FixedSizeList* consultato a settembre 2021