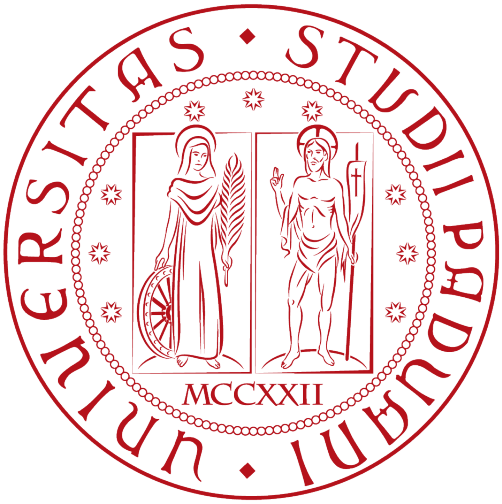
Università degli Studi
di Padova



A.A. 2021-2022

Progettazione e sviluppo di una componente frontend con JavaScript e React

Studente
Marco Mazzucato [Mat. 1193113]

Relatore e Tutor Interno
Prof. Paolo Baldan

Tutor Esterno
Ing. Luca Bizzaro

Ai miei genitori che, in modi completamente diversi,
mi hanno accompagnato in questo periodo
importante della mia vita.

Abstract

RiskApp S.r.l., con sede a Conselve, è un'azienda che si occupa di fornire un supporto ai player assicurativi e bancari che desiderano offrire soluzioni per la copertura assicurativa delle Piccole e Medie Imprese, sviluppando modelli di calcolo per l'analisi dei rischi che un'eventuale compagnia assicurativa vuole coprire. L'obiettivo di RiskApp è quello di fornire un quadro più completo possibile di tutti i rischi a cui possono andare incontro le compagnie assicurative prima di prendere una decisione: per questo offrono strumenti per valutare scenari sia positivi che negativi per ottenere velocemente informazioni sui possibili clienti di queste compagnie assicurative.

Il lavoro svolto durante lo stage presso l'azienda RiskApp S.r.l. è stato quello di creare una nuova feature da inserire nella piattaforma Riskapp. Con l'obiettivo di aumentare la conoscenza delle aziende analizzate, questa nuova funzionalita` permette di visualizzare graficamente la gerarchia interna di una determinata azienda, con relative informazioni aggiuntive riguardanti una singola persona o differenti societa` collegate.

In una prima fase si sono apprese le tecnologie necessarie per lo svolgimento del progetto, tra le quali JavaScript e React. Successivamente si è analizzato il problema, per decidere la libreria grafica più adatta per rappresentare i dati forniti. Infine si è proceduto a codificare l'applicazione, rendendola compatibile con il sistema già esistente.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Scopo dello stage

Il progetto svolto durante il periodo di stage ha previsto lo sviluppo di una componente frontend da inserire nella **piattaforma Riskapp**.

Con l'obiettivo di aumentare la conoscenza delle aziende analizzate, la nuova funzionalità permette di visualizzare graficamente la gerarchia interna di una determinata azienda, con relative informazioni aggiuntive riguardanti una singola persona o differenti società collegate. Lo svolgimento dello stesso ha previsto l'utilizzo di tecnologie molto usate in ambito di sviluppo software lato frontend come *JavaScript* e *React*, oltre ad altre di supporto per rendere il prodotto finale solido e consistente. È stato un progetto molto istruttivo dato che si trattava di sviluppare un prodotto da zero, il che mi ha aiutato a comprendere come nel mondo del lavoro viene sviluppato il software partendo dallo studio del problema fino ad arrivare alla soluzione finale, permettendomi di mettere in pratica gli insegnamenti appresi durante il corso di Ingegneria del Software.

1.2 L'azienda

RiskApp è il principale fornitore di tecnologia che possiede una piattaforma di **gestione del rischio** a tutto tondo per il settore assicurativo. L'idea di base è stata quella di supportare e migliorare le sottoscrizioni, i reclami, le vendite e le decisioni tecniche riguardanti le coperture assicurative: tutto questo offrendo dati e analisi dei rischi, consulenza e sviluppo di software per la valutazione dei rischi aziendali.

L'azienda possiede un **algoritmo proprietario** che stima il valore del rischio in base a dati raccolti da più fonti ed esprime le perdite economiche che possono derivare da tali rischi, confrontando i risultati con le migliori pratiche del settore. Il sistema offre supporto ai player assicurativi e bancari che desiderano offrire soluzioni per la copertura delle piccole e medie imprese, sviluppando modelli di calcolo per l'analisi dei rischi che un'eventuale compagnia assicurativa vuole coprire.

Il prodotto offre inoltre strumenti per aiutare le assicurazioni ad effettuare decisioni informate e ricevere un rapporto dettagliato e chiaro dei rischi, così da mitigarli o assicurarli.

La piattaforma viene principalmente utilizzata da professionisti del rischio, brokers e agenti assicurativi, che la utilizzano per reperire informazioni dettagliate sui i rischi assicurabili e per ricevere un preventivo in maniera istantanea di eventuali polizze.

1.3 Principali problematiche

Dopo un iniziale studio delle tecnologie obbligatorie da utilizzare, tra le quali *React* e *Redux*, è emerso il problema della scelta della **libreria grafica** adatta al tipo di applicazione da realizzare. Nella piattaforma i grafici esistenti sono stati realizzati con la libreria *HighChart*, che però non è adatta alla creazione di grafici dinamici come richiesto nel mio caso. Ho dovuto quindi studiare varie librerie grafiche disponibili per *JavaScript* per avere una panoramica generale e scegliere quindi la più adatta al mio progetto.

1.4 Soluzione scelta

Dopo essermi consultato con il tutor aziendale e aver creato un mockup$_G$ per avere un'idea di come dovrà risultare il prodotto finale ho optato per la libreria grafica ***D3***, una delle più utilizzate in ambito di manipolazione e rappresentazione di dati che permette un'elevata personalizzazione dei grafici. Nello specifico abbiamo concordato di utilizzare la libreria *d3-org-chart*, una sotto-libreria di *D3* specifica per la creazione di grafici gerarchici.

1.5 Strumenti utilizzati

Per la realizzazione del progetto di stage ho utilizzato le seguenti tecnologie, che verranno spiegate in dettaglio nel capitolo successivo:

- *React*, framework$_G$ per la creazione di una componente web riutilizzabile;
- *Redux*, framework per la gestione dello stato dell'applicazione;
- *Node.js*, ambiente run-time che include tutto ciò che serve per eseguire un programma scritto in JavaScript;
- *d3-org-chart*, libreria grafica JavaScript per rappresentare i dati i maniera gerarchica;
- *Axios*, libreria JavaScript per le chiamate asincrone al server.

1.6 Descrizione del prodotto ottenuto

Il prodotto finale consiste in un **grafico gerarchico** che mostra i proprietari e direttori di una determinata azienda e permette di muoversi nella mappa ed espandere o collassare i diversi livelli. Mette inoltre a disposizione una serie di bottoni che permettono di gestire più facilmente il grafico e servono per ingrandire/rimpicciolire il grafico, espandere/collasare la gerarchia e centrare il grafico nella finestra. Questo prodotto copre i requisiti obbligatori prefissati e verrà integrato nella piattaforma Riskapp.

1.7 Organizzazione del testo

Il secondo capitolo descrive le tecnologie studiate ed utilizzate durante il periodo di stage.

Il terzo capitolo descrive il processo che mi ha portato alla realizzazione dell'applicativo.

Il quarto capitolo contiene le valutazioni finali tra cui obiettivi raggiunti e conclusioni personali.

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

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

Capitolo 2

Tecnologie utilizzate

Dato che si tratta dello sviluppo di una componente frontend il pricipale linguaggio di programmazione utilizzato è stato *JavaScript*: linguaggio di scripting lato client utilizzato per rendere interattive le pagine web. *JavaScript*, insieme ad $HTML_G$ e CSS_G, è una delle tecnologie principali della programmazione web frontend, permette di gestire il comportamento degli elementi dinamici di un sito web, ovvero come il contenuto (*HTML*) e lo stile grafico (*CSS*) reagiscono e si modificano sulla pagina web. Questo linguaggio principale viene poi arricchito poi con le seguenti tecnologie per esigenze di progetto e per cercare di ottenere un prodotto quanto più performante e manutenibile.

2.1 React

React è probabilmente la prima libreria *JavaScript* che nasce con una vocazione specifica: diventare la soluzione definitiva per sviluppare applicazioni frontend su PC e su mobile, lo strumento che permette di costruire **interfacce utente dinamiche** e sempre più complesse rimanendo comunque semplice e intuitivo da utilizzare. Creata da *Facebook*, React è la colonna portante del social network più popolare del mondo e su di essa si basa l'interfaccia Web di *Instagram*.

2.1.1 L'approccio React

La creazione di applicazioni Web, indipendentemente dal framework scelto per lo sviluppo, coinvolge necessariamente i tre linguaggi fondamentali della piattaforma: *HTML* per la struttura, *CSS* per la presentazione e *JavaScript* per la logica applicativa. 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. Le pagine complete invece sono ridotte al minimo, anzi molto spesso a una sola, tanto che queste applicazioni prendono il nome di ***Single Page***

Applications$_G$ (SPA) e che servono da "contenitore" in cui creare e gestire l'interfaccia utente.

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, ad esempio un commento a un articolo, o la lista degli stessi commenti. Benché dichiarativa, la rappresentazione del componente in realtà si traduce in chiamate all'API$_G$ di *React* che intervengono, nel modo più veloce e performante possibile, sul DOM$_G$ della pagina per creare gli elementi necessari.

2.1.2 Sviluppo per componenti

React consente di creare interfacce utente complesse tramite la loro suddivisione in componenti. Ogni componente è in grado di definire il proprio **aspetto**, mantenere un proprio **stato** e racchiudere una parte di **logica**. Si può dire che questo non è molto diverso da quello adottato in altri ambienti dove l'interfaccia si esprime tramite l'uso di controlli, ovvero classi specializzate che eseguono il rendering$_G$ di una porzione dell'interfaccia complessiva, contengono campi per mantenere i valori che costituiscono lo stato e all'occorrenza metodi per l'implementazione di funzionalità usate internamente o richiamabili dall'esterno.

Ciascun componente incapsula tutto ciò di cui ha bisogno, ed è pertanto completamente **isolabile**, **riutilizzabile** e **testabile**, in rispetto ai principi di corretta suddivisione delle responsabilità. Tutto ciò può avvenire sul server, tramite il supporto a *Node.js*, oppure all'interno del browser, sul client, permettendo a interfacce utente anche molto complesse di reagire in tempo reale. Ogni componente rappresenta un modello replicabile di markup$_G$ che deve essere inserito nella pagina per fungere da vista per un singolo dato o un insieme di dati.

2.1.3 Proprietà e stato di un componente

Nella fase di creazione di un componente possiamo definire un insieme di proprietà per passare al componente valori da inserire all'interno del template *HTML* generato in fase di rendering, oppure per influenzarne la logica interna di funzionamento. I valori di tali proprietà sono contenuti nel campo **props**. Una delle prerogative di *React* è quella di poter generare gli elementi del DOM in base ai dati specificati e, qualora essi subiscano un cambiamento, "reagire" aggiornandoli per riflettere tali cambiamenti. Tuttavia, le proprietà sono **immutabili**, ovvero non è possibile modificare i valori delle proprietà impostati inizialmente: per queste esigenze è possibile ricorrere allo stato del componente.

Analogamente alle proprietà, lo stato di un componente è un oggetto (memorizzato nel campo **state**) che contiene valori utilizzabili durante il rendering e modificabili all'occorrenza attraverso funzioni specifiche del framework che consentono a *React* di occuparsi, se necessario, dell'aggiornamento degli elementi del DOM legati al nuovo stato, nel modo più veloce e performante possibile.

2.1.4 React lifecycle methods

Ogni componente in *React* attraversa un **ciclo di vita**, dalla nascita (*mounting*) passando per la crescita (*updating*) e arrivando alla sua morte (*unmounting*). I metodi più importanti che ho usato nel mio progetto di stage sono:

- **render:**

  Questo è il metodo del ciclo di vita più utilizzato. È l'unico metodo **obbligatorio** richiesto all'interno di una classe *React*. Come suggerisce il nome, gestisce il rendering del componente nell'interfaccia utente durante il montaggio e aggiornamento dello stesso. Questo metodo deve essere puro e senza effetti collaterali, ovvero deve restituire sempre lo stesso output quando vengono passati gli stessi input. All'interno di questo metodo non è quindi consentito modificare lo stato, ciò dovrebbe accadere negli altri metodi del ciclo di vita, mantenendolo quindi puro.

- **componentDidMount:**

  Questo metodo viene chiamato non appena il componente è montato e pronto, è un buon posto per **avviare le chiamate API**, come richiesto nel mio caso. A differenza del metodo `render()`, `componentDidMount()` consente di aggiornare lo stato e questo causerà un altro rendering, ma avverrà prima che il browser aggiorni l'interfaccia utente. Questo per garantire che l'utente non vedrà alcun aggiornamento dell'interfaccia utente con il doppio rendering. Nonostante sia concesso modificare lo stato, non è consigliabile farlo all'interno di questo metodo per motivi di prestazioni, la migliore pratica è assicurarsi che gli stati siano assegnati nel costruttore del componente.

- **componentDidUpdate:**

  Questo metodo del ciclo di vita viene richiamato non appena avviene un qualche tipo di aggiornamento. Il caso d'uso più comune per il metodo `componentDidUpdate()` è l'**aggiornamento del DOM** in risposta a modifiche di stato o proprietà. Anche qui è possibile modificare lo stato, ma come prima non consigliabile perchè se non si verifica la la presenza di modifiche di stato o proprietà dallo stato precedente può portare a un ciclo infinito.

- **componentWillUnmount:**

  Come suggerisce il nome, questo metodo del ciclo di vita viene chiamato appena prima che il componente venga smontato e distrutto. Se ci sono azioni di **pulizia da fare**, questo è il posto giusto. In questo metodo non è possibile modificare lo stato, dato che il componente in questione non verrà più renderizzato.

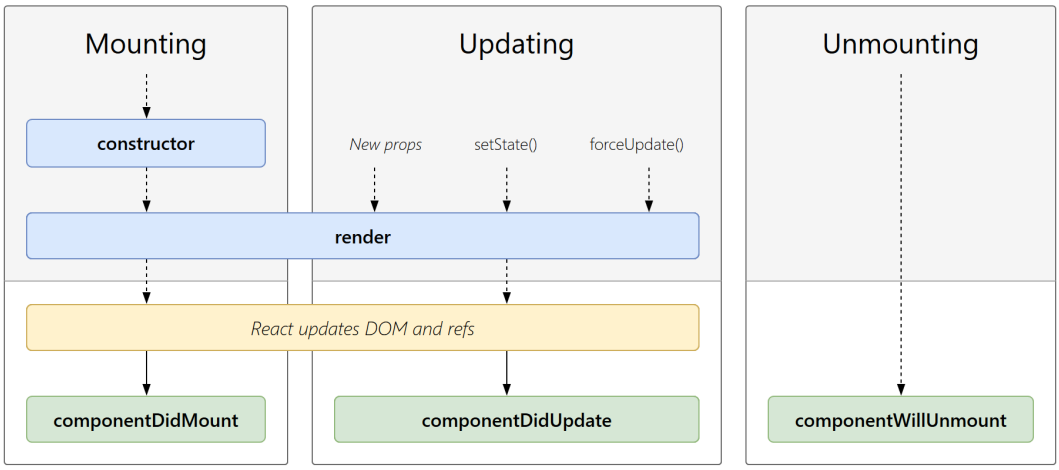


Figura 2.1: React lifecycle methods.

2.2 Redux

Redux è un contenitore di stato prevedibile per app *JavaScript*. Fornisce un modo semplice per **centralizzare lo stato e la logica di un'applicazione web**: per questo è ideale nella programmazione di applicazioni a pagina singola (SPA). Inoltre con *Redux* si possono scrivere applicazioni coerentemente eseguite in ambienti diversi (client, server e nativi), facilitando allo stesso tempo le attività di testing e debugging grazie ai *Redux DevTools*. Pur essendo un componente estremamente leggero, *Redux* dispone di un ampio ecosistema di componenti aggiuntivi. Può essere utilizzato insieme a *React* o con qualsiasi altra libreria *JavaScript* di visualizzazione.

2.2.1 La solidità di Redux

Redux si basa sul concetto di avere un **unico stato**, rappresentato da un oggetto JSON_G e conservato in uno **store**. Questo stato può mutare solo in seguito ad azioni, ma non direttamente; la modifica avviene tramite l'invocazione di una funzione pura denominata **reducer**. Redux si basa su tre principi fondamentali:

- **Single source of truth:** lo stato è l'unica fonte di verità a cui fa riferimento l'interfaccia utente;
- **State is read-only:** lo stato è in sola lettura e può mutare solo con un intento ben definito, in pratica a partire da una azione e attraverso un reducer;
- **Changes are made with pure functions:** i cambiamenti allo stato avvengono solo attraverso funzioni pure, cioè una funzione che dato un certo input restituisce sempre lo stesso output senza effetti collaterali. Questo rende le funzioni facilmente prevedibili e testabili, e vengono utilizzate per gestire i reducer.

Questi tre principi, per quanto possano sembrare limitanti, servono a garantire una corretto rispetto dei principi di **coesione** e **accoppiamento** nel nostro sofware.

2.2.2 Concetti fondamentali di Redux

In *Redux* si celano diversi termini e concetti importanti, nello specifico durante la mia esperienza di stage mi sono focalizzato sui seguenti:

- **State:** lo state è quello già citato in *React*, cioè lo stato che viene renderizzato nella UI_G. È buona pratica, e Redux serve proprio a questo, mantenere lo stato generale dell'applicazione nel modo più centralizzato possibile.
  Potrebbe sorgere il dubbio se un singolo componente può mantenere il

suo stato interno. La risposta è che si dovrebbe limitare l'uso di uno stato locale nel componente portandolo il più in alto possibile nell'albero dei componenti (*lifting state*).

Questo potrebbe bastare per una qualsiasi applicazione, ma allora, perché usare *Redux* e una gestione dello stato centralizzato? Dobbiamo farlo perché lo stato locale non scala bene all'aumentare degli sviluppatori, non tutti nel team applicheranno correttamente il *lifting state* e, quindi, avere una architettura ben definita con l'aiuto di *Redux* aiuta a mantenere una buona qualità del codice. La gestione locale dello stato è comunque una buona scelta nel caso in cui l'applicazione (o il team) non è di grandi dimensioni oppure si sta sviluppando una libreria di componenti;

- **Actions:** sono funzioni che fanno il *dispatch* (invio) di azioni, come ad esempio una invocazione ad un servizio esterno. Queste azioni possono contenere un body che potrebbe essere utilizzato, nei reducer, per eseguire la mutazione dello stato;

- **Reducer:** gestisce lo stato, in pratica è un grande *switch* che esegue una variazione dello stato in base all'azione, ed eventualmente, al suo contenuto. Lo stato impostato dovrebbe tornare sempre una nuova istanza immutabile dello stato. Nel caso in cui l'applicazione sia complessa e si vogliono organizzare i reducer in più file si può usare il metodo `combineReducers`;

- **Connect:** è una funzione invocata quando si esporta il componente, serve a collegarlo alle azioni e alla parte dello stato e delle azioni di cui ha bisogno. Questo vuol dire che troveremo gli elementi dello stato e le azioni che ci interessano nelle proprietà del componete e le proprietà relative allo stato saranno reimpostate quando questo cambierà.

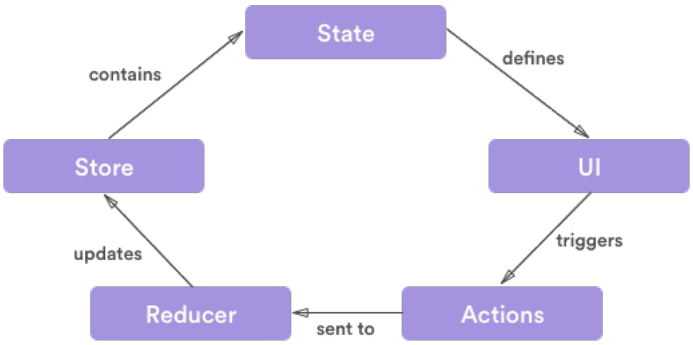


Figura 2.2: React flow — Leggerlo partendo da UI e da destra verso sinistra.

2.3 Node.js

Node.js è un software open source per l'esecuzione del codice *JavaScript* di proprietà di *Google Chrome*. Presenta una serie di moduli e permette agli sviluppatori di aggiungerne altri sempre in *JavaScript*.

Si tratta di un **runtime JavaScript**, per l'esattezza, che permette di eseguire il codice come per qualsiasi linguaggio di programmazione. La novità risiede nel fatto che questo linguaggio **nasce sul lato client** e, quindi, viene eseguito di norma solo all'interno di un browser. La particolarità di *Node.js* riguarda il fatto che ha estrapolato il linguaggio dal browser rendendolo programmabile come gli altri linguaggi conosciuti. Dunque può essere utilizzato per eseguire qualsiasi tipologia di programma e, proprio per questa sua **versatilità**, ha preso piede anche nel mondo dell'*Internet of Things*$_G$.

Tra gli ambiti di utilizzo al primo posto vi è la realizzazione di backend web per cui si sfrutta *JavaScript* per realizzare componenti server che interagiscano con qualsiasi sistema mobile o di web app. Come anticipato trova applicazione anche nell'*Internet of Things* e nella domotica$_G$ dal momento che la massima efficienza consente di realizzare applicazioni da integrare con i dispositivi IoT. Infine è utile per la creazione di applicazioni desktop e per la gestione del sistema operativo tramite le apposite righe di comando che possono essere modificate o integrate alla libreria di un progetto.

2.3.1 La libreria NPM

NPM (*Node Package Manager*) è la più grande libreria opensource al mondo di pacchetti *JavaScript*. È il **gestore di pacchetti ufficiale** che viene installato con la piattaforma *Node.js*.

Si tratta di un'**interfaccia a riga di comando** che aiuta nell'installazione dei pacchetti, nella gestione delle versioni e nella gestione delle dipendenze. NPM fa anche riferimento al registro, un grande database pubblico di applicazioni *JavaScript*, e al sito web che consente di scoprire pacchetti, impostare profili e gestire altri aspetti dell'esperienza NPM.

2.4 D3

D3 è una libreria *JavaScript* open source utilizzata per creare **visualizzazioni dinamiche ed interattive** partendo da dati organizzati, visibili attraverso un comune browser. Per fare ciò si serve largamente degli standard web: SVG_G, *HTML*, e *CSS*.

Questa libreria utilizza funzioni *JavaScript* prefatte per selezionare elementi del DOM, creare elementi SVG, aggiungergli uno **stile grafico, transizioni o effetti di movimento**. Questi oggetti possono essere largamente personalizzati utilizzando lo standard web *CSS*. In questo modo grandi collezioni di dati possono essere facilmente convertiti in oggetti SVG usando semplici funzioni di *D3* e così generare ricche rappresentazioni grafiche di numeri, testi, mappe e diagrammi. I dati utilizzati possono essere in diversi formati, i più comuni sono JSON e CSV_G, ma, se necessario, si possono scrivere funzioni *JavaScript* ad hoc per leggere dati in altri formati.

2.4.1 d3-org-chart

Nel mio caso, sapendo ciò che dovevo rappresentare, ho deciso di utilizzare *d3-org-chart*, una libreria **specifica per lo sviluppo di grafici gerarchici** basata su *D3*. Questa libreria permette di rappresentare i dati provenienti da un file in formato JSON o CSV in un grafico gerarchico, dopo aver assegnato le varie dipendenze per strutturare l'albero nel modo desiderato.

2.5 Axios

Axios è una libreria *JavaScript* che permette di connettersi con le API di backend e **gestire le richieste effettuate tramite il protocollo $HTTP_G$**. Il vantaggio di Axios risiede nel suo essere *promise-based*, permettendo quindi l'implementazione di codice asincrono. Il codice asincrono permette, in una pagina, di caricare più elementi contemporaneamente invece che in maniera sequenziale, snellendo sensibilmente i tempi di caricamento.

Il **Promise**, su cui si basa *Axios*, è un oggetto di *JavaScript* che permette di completare delle richieste in maniera asincrona facendole passare da tre stati: in sospeso, soddisfatta, rifiutata. *Axios* è isomorfo, ovvero può essere eseguito sia nel browser sia in *Node.js* con la stessa base di codice. Sul lato server (Node.js) utilizza il modulo HTTP, mentre sul client (browser) utilizza *XMLHttpRequests*.

Capitolo 3

Sviluppo del progetto

3.1 Fase di analisi

Dopo una prima parte di stage dove ho studiato le tecnologie esposte nel capitolo precedente, ad eccezione di *D3* che verrà scelta in seguito, ho iniziato effettivamente a pensare come realizzare il prodotto. Analizzando la richiesta, ciò che dovevo produrre era una **rappresentazione grafica dei proprietari e direttori di una determinata azienda**, trovando questi dati in un file JSON (diverso per ogni azienda) che conteneva un *report*.

In accordo con il tutor, e visto che è una prassi interna all'azienda, la prima cosa da fare era creare un'**anteprima grafica** per avere un'idea comune di ciò che sarebbe stato il risultato finale, e allo stesso tempo permettere di capire la libreria *JavaScript* più adatta per realizzarlo. Per realizzare questa anteprima ho utilizzato *Figma*, uno strumento per la progettazione di interfacce che oltre ed essere finalizzato allo sviluppo per il web, offre anche un sistema di collaborazione in real-time. Un altro aspetto importante di *Figma* è che tramite plugin$_G$ permette di leggere diversi formati di file tra cui JSON, il che mi ha permesso di risparmiare molto tempo dato che non ho dovuto inserire tutti i dati a mano.

Un'altra cosa che ho analizzato durante questa prima parte è stato il file JSON. Il mio tutor mi ha fornito quello di una singola azienda, e ho cercato quindi di capire i **dati da visualizzare** e altri invece da scartare. Di questo file che conteneva praticamente tutte le informazioni disponibili di un'azienda, dal fatturato alla metratura dell'immobile di appartenenza, ciò su cui mi sono focalizzato sono stati due **array**, quello che conteneva i **proprietari** e quello che conteneva i **direttori**. Analizzando questi due array per capire i dati da estrapolare e mostrare nel mio grafico, sono arrivato alla conclusione che i seguenti campi erano adatti.

Per quanto riguarda i **proprietari**:

- Nome e cognome;
- Codice fiscale;
- Indirizzo;
- Tipo di proprietario (persona o azienda);
- Percentuale detenuta;
- Quantità in euro della percentuale.

Per quanto riguarda i **direttori**:

- Nome e cognome;
- Codice fiscale;
- Indirizzo;
- Luogo e data di nascita;
- Posizioni ricoperte (anche più di una).

Poichè i dati da mostrare erano tanti, per rendere il grafico più **gestibile** ho pensato di mostrare inizialmente solo poche informazioni per non creare confusione, permettendo una volta selezionata una determinata persona di vedere le informazioni mancanti.

Una volta chiaro come sarebbe dovuto risultare graficamente il tutto, c'era da capire che **libreria grafica** utilizzare. Il fatto di voler creare un grafico dinamico e altamente personalizzato mi ha spinto a scegliere la libreria **D3** a discapito della libreria *HighChart* che veniva solitamente utilizzata in azienda. Quello che mi ha convinto a voler utilizzare questa libreria, oltre appunto al fatto che permette un'**elevata personalizzazione dei grafici**, è il fatto di avere avuto un minimo di esperienza avendola usata nel progetto universitario di *Ingegneria del Software*.

Studiando questa libreria e ricercando il metodo più adatto per creare un grafico il più possibile uguale al design prefissato, mi sono imbattuto nella libreria *d3-org-chart*, sviluppata tramite un progetto open-source e basata appunto su *D3*.

In questa prima fase il tutor mi ha reso disponibile un account per la piattaforma *Riskapp* oltre ad avermi dato accesso al codice della stessa. In questo modo ho pututo comprendere meglio il contesto in cui avrei inserito la mia applicazione e soprattutto ho potuto vedere come viene gestito il codice in progetti molto grandi come questo.

3.2 Architettura già esistente

Il software *RiskApp* si configura come una web-app. Il sito è **modulare** ovvero composto da un'insieme di servizi. A seconda della propria sottoscrizione, l'utente ha accesso a un sottoinsieme di questi servizi. Il sistema è composto da un **frontend** e un **backend**. Il backend di *RiskApp* è scritto in *Python 3*, supportato dal framework *Django* per un agevole sviluppo web. Il frontend è scritto appunto in *Javascript* che fa uso delle librerie sopracitate *React* e *Redux*, le quali consentono il **regolamento del flusso di esecuzione** e l'**organizzazione del codice** in piccole componenti riusabili. La comunicazione tra frontend e backend è costituita da chiamate **REST**$_G$ attraverso l'utilizzo di *Django REST framework*. Si utilizzano VirtualBox e Vagrant per consentire a tutti gli sviluppatori di lavorare sullo **stesso ambiente** ed evitare la maggior parte degli errori dovuti alle proprie personali macchine. VirtualBox ospita l'immagine del sistema operativo desiderato comune a tutti gli sviluppatori, mentre Vagrant coordina il proprio sistema locale con quello presente sulla macchina virtuale. L'ambiente di sviluppo tipicamente usato per lavorare è **PyCharm**. Questo IDE$_G$ è uno strumento pensato appositamente per lavorare in *Python*, includendo anche degli strumenti di sviluppo quali debugging$_G$ e deploying$_G$ su GitHub e risulta comodo anche lato frontend con *JavaScript*. Inoltre offre un assistente per la scrittura di codice e safe refactoring e facilita l'utilizzo di Vagrant per il quale la configurazione risulta semplice. Questo IDE non è stato imposto in modo vincolante: al suo posto è possibile utilizzare allo stesso modo **Visual Studio Code**, come nel mio caso.

3.3 Approccio alla codifica

Nonostante avrei dovuto codificare un componente da zero non mi è stato richiesto di produrre documentazione relativa al codice, ma solo di scriverlo in maniera pulita e commentandolo, dato che l'azienda punta ad un approccio **Agile** a discapito di quello a cascata.

3.3.1 Perchè usare il metodo Agile

Inizialmente, nel mondo informatico, veniva usato maggiormente il **metodo a cascata**, che vede le sue origini nell'industria manifatturiera e nel settore delle costruzioni, ambienti in cui il processo di lavoro è stato ed è tutt'ora per forza di cose lineare.

Quando lo sviluppo del software cominciò a essere concepito come attività industriale, questa metodologia venne inserita anche in questo settore. Inizialmente l'introduzione del modello a cascata fu una grande innovazione tra i sviluppatori di software, considerando che si arrivava da una situazione in cui l'approccio allo sviluppo era definito "code and fix", un modo molto artigianale di procedere per tentativi ed errori.

Attorno agli anni '80 questo metodo iniziò ad essere messo in **discussione**, i principali difetti emersero particolarmente nell'affrontare progetti molto grossi e i cui requisiti non erano ben chiari fin dal principio. La linearità del metodo non consentiva infatti di tornare facilmente (o agilmente) sui propri passi e ogni fase doveva essere conclusa prima di poter affrontare la successiva.

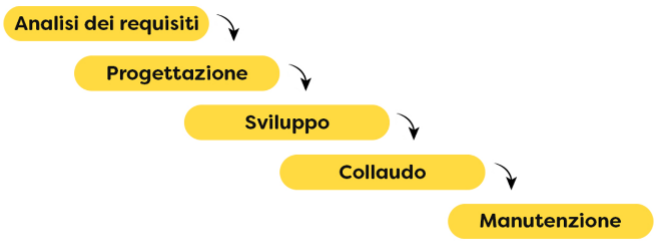


Figura 3.1: Fasi del modello a cascata.

Verso la metà degli anni '90 l'industria di sviluppo software iniziò a interrogarsi su come affrontare questa situazione e come **sfornare in tempi brevi software funzionante**, di qualità e che sapesse reggere nel mercato in continua evoluzione. Si iniziò a parlare di approcci "agili", snelli, iterativi e focalizzati sulle **reali necessità** dei committenti. Il processo di sviluppo non era più visto come una sequenza rigida e lineare, ma suddiviso in fasi ripetute dette sprint. In ogni sprint c'era infatti la possibilità di analizzare piccole parti di richieste e valutare il modo migliore di svilupparle, per poi passare direttamente a sviluppo, test e rilascio.

L'attenzione alle necessità, alle iterazioni, alle relazioni e alla collaborazione ha portato come ovvia conseguenza a dover rivedere il processo definito nel metodo a cascata, passando quindi da un processo rigido e lineare ad un processo appunto più agile, snello e adattabile basato su singole fasi di sviluppo iterative, i cosiddetti **Sprint**. Ma come funzionano queste fasi? Ogni Sprint consente di rilasciare una parte di applicativo contenente specifiche funzionalità concordate all'inizio dello sprint stesso. Il cliente ha quindi la **possibilità di utilizzare e testare il software**, restituendo preziosi feedback al team di sviluppo utili per continuare l'implementazione negli sprint seguenti, avvicinandosi di volta in volta ad un prodotto finale qualitativamente più alto e più vicino alle sue esigenze.

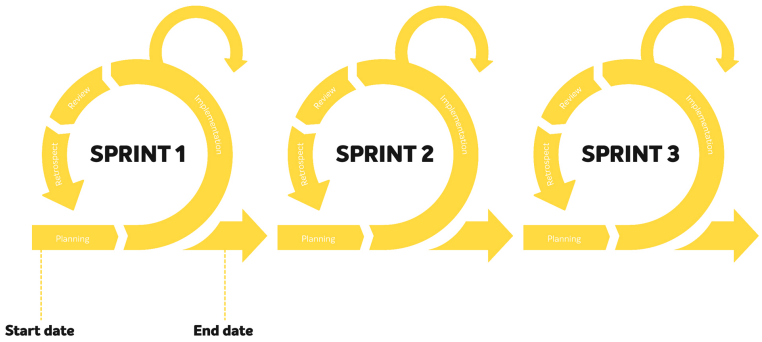


Figura 3.2: Fasi del modello Agile.

Nel mio caso lo sviluppo di questa applicazione è stato quindi uno sprint del metodo Agile.

3.4 Fase di codifica

Per lo sviluppo di questo applicativo ho utilizzato **Visual Studio Code**, un IDE cross-platform molto leggero e dinamico che permette di lavorare con una grande quantità di linguaggi. Permette inoltre di realizzare applicazioni *Node.js*: è veloce e leggero non soffrendo quindi delle problematiche in termini di performance che affliggono alcune app *JavaScript*.

3.4.1 Creazione del componente React

Come prima cosa dovevo importare nel file *JavaScript* del componente tutte le librerie e classi necessarie per far funzionare correttamente il codice. Oltre ad importare ovviamente la libreria *React* e la classe `OrgChart` dalla libreria *d3-org-chart*, ho importato le seguenti cose:

- La funzione `connect` dalla libreria *react-redux* e la funzione `bindActionCreators` dalla libreria *redux*, necessarie per collegare *React* allo store di *Redux*;
- Alcune componenti dalla libreria *react-bootstrap* e dalla libreria *antd* per realizzare le parti dell'interfaccia utente, tra cui bottoni e spinner per il caricamento;
- La funzione `fetch_companyReport` per richiedere il file JSON dal database tramite chiamata REST API;
- Alcune immagini png sempre per la parte grafica dell'applicazione.

```
1   import React, { Component } from "react";
2   import { connect } from "react-redux";
3   import { bindActionCreators } from "redux";
4   import { Button, ButtonGroup, Spinner} from "react-bootstrap";
5   import { Popover } from "antd";
6   import { OrgChart } from "d3-org-chart";
7   import { fetch_companyReport } from "../actions";
8   import user from "../icons/user.png";
9   import company from "../icons/company.png";
10  import arrow from "../icons/arrow.png";
```

Figura 3.3: Import nel file del componente.

In un secondo momento, dopo aver creato il costruttore impostando la variabile di stato `isLoading` uguale a `true` per gestire il possibile ritardo di ricezione dei dati dal server, ho scritto il metodo `componentDidMount()`. In questo metodo, adatto per le chiamate API, ho richiamato la funzione `fetch_companyReport()` per richiedere al server il file JSON contenente il report aziendale. Per ottenere il JSON desiderato, a questa funzione viene passato il parametro `cID`, che essendo una props e quindi gestito al livello superiore si riferisce alla giusta azienda. Una volta ricevuti questi dati la variabile di stato `isLoading` diventa `false` e viene invocata la funzione `createDiagram()` che si occupa di creare il grafico.

```
12  class OrgChartComponent extends Component {
13    constructor(props) {
14      super(props);
15      this.createDiagram = this.createDiagram.bind(this);
16      this.state = {
17        isLoading : true
18      }
19    }
20
21    componentDidMount() {
22      if (!this.chart) {
23        console.log(this.chart)
24        this.props.fetch_companyReport(this.props.cID)
25        .then((res) => {
26          if(res.data){
27            this.setState({isLoading: false})
28            this.createDiagram(res);
29        }});
30      }
31    }
```

Figura 3.4: Costruttore e metodo componentDidMount().

La funzione `createDiagram()` è la funzione principale di questo componente perchè oltre a creare i vari nodi del grafico in maniera corretta si occupa effettivamente di "disegnarlo". All'inizio di questa funzione ho dovuto comporre gli array che mi servivano, perchè come già spiegato il file JSON del report conteneva molti altri dati non rilevanti per ciò che dovevo fare. Ho quindi creato un array per i **proprietari** e uno per i **direttori**, assegnando ad ogni elemento di questi una chiave per identificare se fa parte di un gruppo o di un'altro. Dopodichè ho creato altri tre elementi, la radice dell'albero che contiene il nome dell'azienda, e le card con la scritta "PROPRIETARI" e quella con la scritta "DIRETTORI", che nel grafico saranno i nodi padre delle persone effettive. Ho inserito quindi tutte queste informazioni in un unico array.

```
39  createDiagram(res) {
40    //take the array of shareholders and directors from company report
41    let shareholdersArray = res.data.report.shareCapitalStructure.shareHolders;
42    shareholdersArray.forEach(item => item.key = "prop");
43    let directorsArray = res.data.report.directors.currentDirectors;
44    directorsArray.forEach(item => item.key = "dir");
45
46    //create the first 3 card, that are the same for each company
47    const azienda1 = {name: res.data.report.companySummary.businessName}; // name of the company
48    const shareholders1 = {name: "PROPRIETARI"};
49    const directors1 = {name: "DIRETTORI"};
50
51    //put into the same array shareholder, directors and the first 3 card
52    let array = [ azienda1, shareholders1, directors1, ...shareholdersArray, ...directorsArray];
```

Figura 3.5: Creazione degli array con i dati del JSON.

Per poter utilizzare correttamente la libreria *d3-org-chart*, ogni elemento dell'array deve avere un campo id proprio e un id del padre, ho quindi iterato sull'array per assegnare ad ogni elemento questi campi per poi ottenere la struttura desiderata. Nel mio caso qesti due campi si chiamano rispettivamente `nodeId` e `parentNodeId`.

```
54    //set the nodeId and parentNodeId for each element of new array
55    array.forEach((item, i) => {
56      item.nodeId= i+1;
57      if(i==0)
58        item.parentNodeId = null;
59      if(i==1 || i==2)
60        item.parentNodeId = 1;
61      if(item.key == "prop")
62        item.parentNodeId = 2;
63      if(item.key == "dir")
64        item.parentNodeId = 3;
65    })
```

Figura 3.6: Assegnazione campi nodeId e parentNodeId.

Una volta creati i nodi con le proprietà necessarie per essere visualizzati correttamente, ho potuto procedere con la creazione del grafico, utilizzando quindi la libreria *d3-org-chart*. Come per la classica libreria *D3* si crea un oggetto iniziale al quale vengono concatenate una serie di proprietà per poter personalizzare a proprio piacimento vari aspetti del grafico. Queste proprietà contengono ad esempio lo zoom iniziale o il set di dati dal quale prenderli.

```
131     if (!this.chart) {
132       this.chart =  new OrgChart();
133     }
134      this.chart.container(node)
135         .data(newData)
136         .svgWidth(788)
137         .initialZoom(0.6)
138         .nodeHeight((d) => {
```

Figura 3.7: Creazione del grafico OrgChart.

Un'importante proprietà per l'aspetto di layout è sicuramente `nodeContent`, ovvero ciò che ogni nodo dovrà contenere. Qui inserendo il codice *HTML* e *CSS* sotto forma di stringa è possibile personalizzare appunto il layout del nodo. Per ogni tipologia di nodo, che sia un proprietario o un direttore, che sia aperto o chiuso, ho scritto il relativo codice per renderlo uguale rispetto al mockup iniziale.

```
return `
  <div style="padding-top:30px;background-color:none;margin-left:1px;height:${d.height}px;border-radius:2px;overflow:visible">
    <div style="box-shadow:10px 10px 10px #C2C3C4;height:${d.height - 32}px;padding-top:0px;background-color:#F8F9FA;border-radius:3%">
      <img src=" ${user}" style="margin-top:-30px;margin-left:${d.width / 2 - 30}px;width:60px;height:60px;" />
        <div style="padding:10px; padding-top:10px;text-align:center">
          <div style="color:#5890f8;font-size:20px;font-weight:bold;"> ${d.data.name} </div>
          <div style="color:#404040;font-size:14px;margin-top:10px;margin-bottom:20px">${d.data.role1.positionName}</div>
      <img src="${arrow}" style="width:40px;height:25px;" />
    </div>
  </div>
`;})
```

Figura 3.8: Esempio di layout di un nodo.

In seguito c'era da scrivere il metodo `render`, obbligatorio per ogni componente *React*. In questo metodo, oltre ovviamente a inserire il riferimento al grafico, ho inserito una serie di bottoni utilizzando la libreria *react-bootstrap*, i quali permettono di gestire con più facilità le interazioni con il grafico. Uno di questi è un **popover**, un bottone che una volta cliccato farà apparire un testo con le informazioni per un corretto utilizzo del grafico.

```
<Popover
  title="Istruzioni all'uso"
  content={content}
  placement="bottomRight"
  trigger="focus"
  style={{width:"100px"}}>
  <Button
    variant="info"
    style={{margin:"5px"}}
    className="pull-right"
  >Info</Button>
</Popover>
```

Figura 3.9: Codice per il componente Popover.

Infine, all'esterno della classe componente ma rimanendo nello stesso file, ho scritto il codice che permette di collegarsi allo store di *Redux*. La funzione principale è connect, alla quale vengono passati i parametri `mapStateToProps` e `mapDispatchToProps`, che rispettivamente servono per ottenere i dati dallo store di *Redux* e inviare dati allo stesso.

```
360  function mapStateToProps() {
361    return {};
362  }
363
364  function mapDispatchToProps(dispatch) {
365    return bindActionCreators(
366      {
367        fetch_companyReport,
368      },
369      dispatch
370    );
371  }
372
373  OrgChartComponent = connect(mapStateToProps, mapDispatchToProps)(OrgChartComponent);
374
375  export { OrgChartComponent };
```

Figura 3.10: Connessione con lo store di Redux.

3.4.2 Integrazione nel sistema

Una volta terminato il componente richiesto, dovevo integrarlo all'interno del sistema già esistente. Dopo essermi confrontato con il mio tutor abbiamo deciso che il modo migliore per visualizzare il grafico fosse una modale sulla pagina principale, ovvero una **finestra popup** che viene visualizzata quando si preme un bottone. Ho quindi aggiunto il seguente codice nel file *JavaScript* relativo alla pagina principale e ho naturalmente importato il componente *React* precedentemente creato, passandogli come **props** l'identificativo dell'azienda corrente.

```
155   renderModalOrgChart(){
156     return (
157       <Modal
158         size="lg"
159         show={this.state.showModalOrgChart}
160         onHide={() => {
161           this.toggleModalOrgChart();
162         }}
163         scrollable
164       >
165         <Modal.Header closeButton>
166           <Modal.Title >Struttura societaria</Modal.Title>
167         </Modal.Header>
168         <Modal.Body style={{padding:"0px 5px 5px 5px"}}>
169           <OrgChartComponent cID={this.props.params.id}/>
170         </Modal.Body>
171       </Modal>
172     );
173   }
```

Figura 3.11: Codice per visualizzare e gestire la modale.

```
{this.renderModalOrgChart()}
{this.props.customer && (
<Card
style={{
  marginBottom: "2em",
  backgroundColor: "#00AB41",
  color: "#fff",
}}
cardClick={() => {
  this.setState({showModalOrgChart: true});
}}
>
  <div>
    <h5>
      <img
        src={orgIcon}
        style={{position:"relative", bottom:"3px", width:"15px", height:"15px"}}>
      </img>
      {" "}Mostra struttura societaria
    </h5>
  </div>
</Card>)}
```

Figura 3.12: Codice per creare la card che una volta cliccata mostra la modale.

Come ultima cosa mi restava da creare la chiamata che chiedesse il file necessario a creare il grafico direttamente al server aziendale. Per fare ciò è stato necessario usare *Axios*, passando come parametro alla funzione `axios.get` il corretto percorso fornitomi dal tutor.

```
9  export function fetch_companyReport(cID) {
10     return (dispatch) => {
11       return axios.get(`${ROOT_URL}/customers/${cID}/customerstructure/`)
12           .then((res) => {
13             if(httpSuccessTest(res.status)){
14               return res.data;
15             }
16           })
17           .catch((error) => {
18             console.log(error);
19             if(error !== UNAUTHORIZED_ERROR)
20               dispatch(notifSend({
21                 message: messageStatus(error),
22                 kind: 'danger',
23                 dismissAfter: 2000
24               }));
25             return {status: error.status};
26           });
27     }
28   }
```

Figura 3.13: Funzione per la connessione al server.

3.5 Prodotto ottenuto

Il prodotto finale è quindi una **modale**, che rispettivamente alla pagina dell'azienda in cui ci si trova, mostra la relativa **struttura societaria**, collegandosi con il backend tramite chiamata *Axios*, ricevendo un file JSON che viene elaborato e tramite la libreria *D3* crea un grafico gerarchico, offrendo diversi bottoni per una migliore navigazione dello stesso.

Le singole card di proprietari e direttori possono essere espanse per vedere ulteriori informazioni ma per motivi di privacy non vengono inseriti screenshot riguardanti questo. Graficamente il risultato è il seguente:

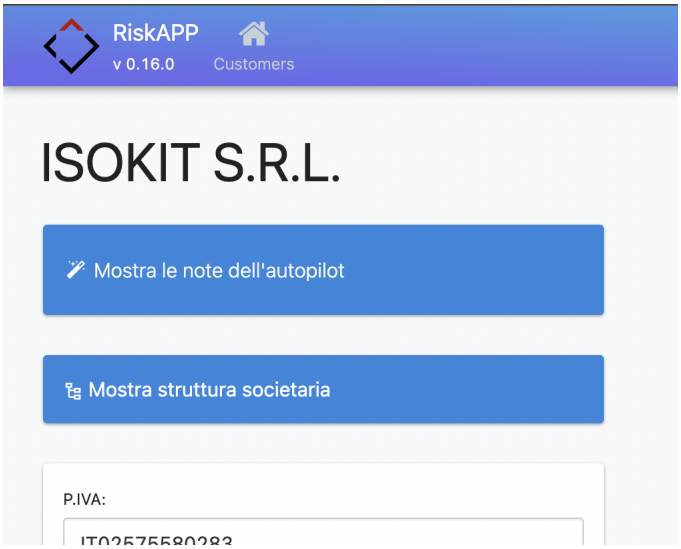


Figura 3.14: Bottone nella pagina principale che mostra la modale.

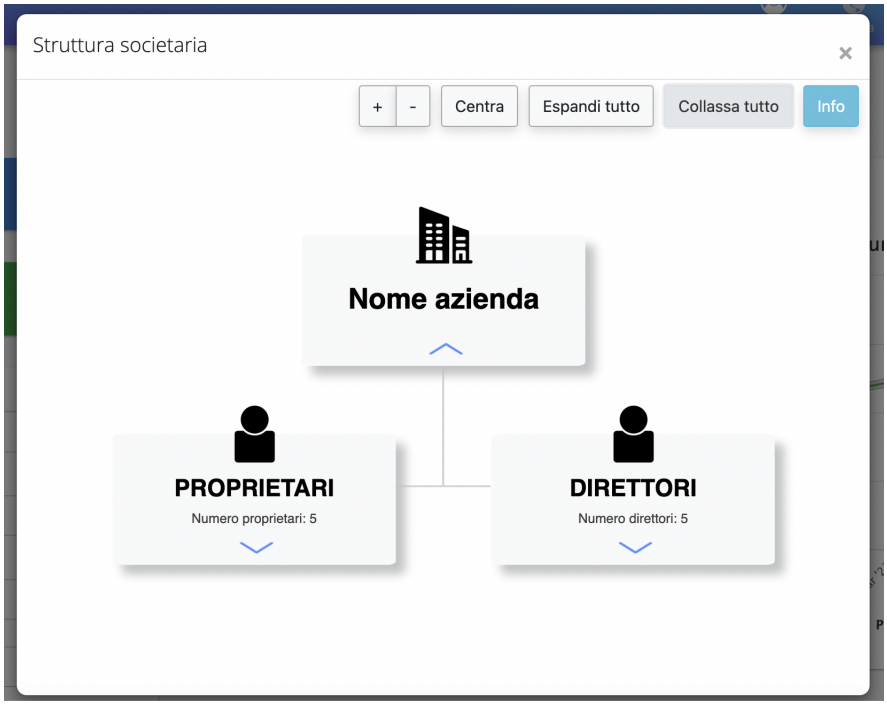


Figura 3.15: Grafico all'apertura della modale.

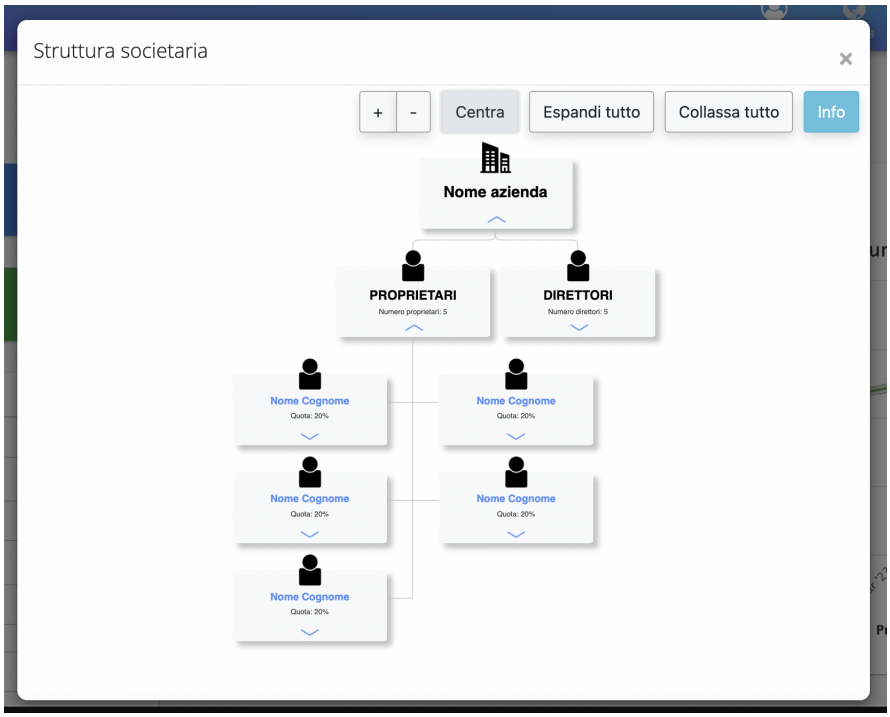


Figura 3.16: Esempio sola visualizzazione dei prorietari.

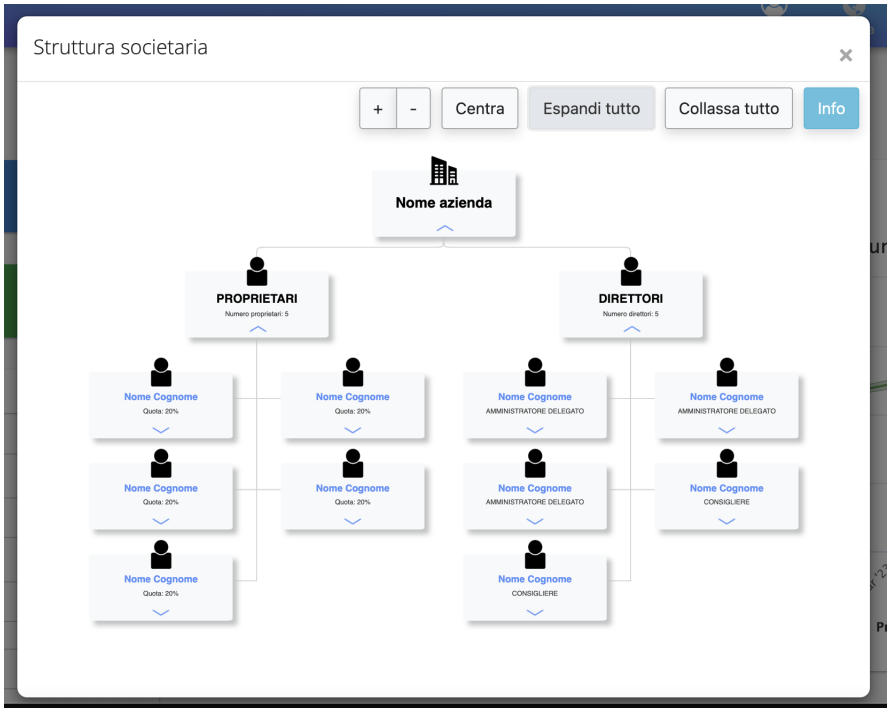


Figura 3.17: Espansione completa del grafico.

Figura 3.18: Popover del bottone Info che spiega come utilizzare il grafico.

Capitolo 4

Valutazione finale

4.1 Obiettivi di stage

Notazione

Si farà riferimento ai requisiti secondo le seguenti notazioni:

- *O* per i requisiti obbligatori, vincolanti in quanto obiettivo primario richiesto dal committente;
- *D* per i requisiti desiderabili, non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;
- *F* per i requisiti facoltativi, rappresentanti valore aggiunto non strettamente competitivo.

Le sigle precedentemente indicate saranno seguite da una coppia sequenziale di numeri, identificativo del requisito.

Obiettivi fissati

- Obbligatori
  - *O01*: Creazione del frontend per la rappresentazione dello schema societario;
  - *O02*: Interazione con i dati da backend via API REST sullo schema societario;
- Desiderabili
  - *D01*: Richiamare e rappresentare notizie riguardanti gli esponenti e le società collegate alla società analizzata, usando altri servizi API di Riskapp per la ricerca di notizie;
- Facoltativi
  - *F01*: Test a livello di frontend;

4.2 Considerazioni e valutazione

In seguito alle attività svolte, nelle 320 ore disponibili sono riuscito a portare a termine in maniera soddisfacente i requisiti obbligatori, non riuscendo però a coprire quelli desiderabili e facoltativi per motivi prettamente di tempistiche. Avendo avuto più tempo, avrei sicuramente migliorato il prodotto sotto il punto di vista della robustezza, dato che il tutor mi ha spiegato che molto probabilmente non tutti i file dei report sono uguali e avrei quindi dovuto gestire vari casi in base alla formattazione degli stessi. Una cosa che purtroppo non è stata fatta e che mi avrebbe dato la possibilità di affrontare argomenti nuovi e interessanti è l'obiettivo *D01*, il quale mi avrebbe permesso di scoprire meglio tutta l'infrastruttura Riskapp soprattutto lato backend. Sicuramente il prodotto finale è una base per l'aggiunta di funzionalità di qualsiasi tipo, dalla ricerca di eventuali altre informazioni correlate alla navigazione dentro l'applicativo stesso.

Al termine dello stage devo ritenermi soddisfatto rispetto alle aspettative iniziali di crescita personale. Questo progetto mi ha permesso di mettere in pratica buona parte delle conoscenze acquisite durante il mio percorso di studi ma soprattuto di arricchire la mia formazione con nuove tecnologie. Avendo creato in quasi completa autonomia questo prodotto, prendendo anche decisoni in fatto di tecnologie da utilizzare, sono riuscito ad affrontare tutte le varie fasi di uno sviluppo software, dall'ideazione alla realizzazione. Anche il fatto di aver riscontrato diversi problemi molto ricorrenti in fase di sviluppo ed essere riuscito a risolveri mi ha fatto acquisire delle soft skill molto utili da poter utilizzare nel mondo del lavoro. Un altro aspetto molto importante a mio avviso è che la componente che ho realizzato, dopo essere stata ovviamente rivista dal tutor e magari ottimizzata sotto certi aspetti, sarà sicuramente integrata nel loro sistema ed utilizzata.

In conclusione sono molto soddisfatto di aver sviluppato una componente frontend utilizzando linguaggi di programmazione come JavaScript e React, dato che sono tra i più utilizzati in questo ambito e quindi molto utili per il futuro. L'azienda mi ha dato un certo livello di autonomia e questo mi ha permesso di capire come organizzare il proprio lavoro al meglio, sentire il peso molto più marcato delle scelte fatte e rendermi responsabile di qualcosa di importante senza relegarmi a lavori secondari.

4.3 Conclusioni

Il compito dell'Università non è limitato alla formazione di uno studente su vari fronti, ma si estende alla sua preparazione per affrontare l'inserimento al mondo del lavoro, al termine del percorso di studio. Questo non significa che uno studente laureato debba essere già in grado di svolgere un lavoro, ma deve essere preparato a superare le problematiche che gli si presentano. Sulla base della mia esperienza, uno studente universitario, grazie alla preparazione ricevuta, è in grado di gestire in autonomia il lavoro assegnatogli, sapendo affrontare anche situazioni impreviste. Inoltre possiede un'ottima capacità di adattamento rispetto allo studio di nuove tecnologie e all'utilizzo di nuovi strumenti di lavoro. Nella visione generale del corso di laurea in Informatica ritengo molto utile e stimolante per gli studenti l'utilizzo di progetti didattici, sia individuali che di gruppo, a supporto di alcuni corsi di studio. Ciò permette di mettere subito in pratica quanto si apprende durante le lezioni frontali, avendo al tempo stesso un obiettivo da portare a termine entro i tempi stabiliti e con le risorse a disposizione. Un altro aspetto molto importante è che lo stage che si svolge al termine del percorso di studio, dopo il completamento di tutti gli esami previsti dal corso di studi. Collocare lo stage anticipatamente andrebbe a discapito dello studente, il quale potrebbe non avere le conoscenze necessarie ad affrontare il progetto. Inoltre la propedeuticità data dall'insegnamento di Ingegneria del Software fornisce allo studente l'esperienza necessaria allo svolgimento di un progetto software in autonomia.

Glossario

API

In un programma informatico, con application programming interface (API), in italiano "interfaccia di programmazione di una applicazione", si indica un insieme di procedure (in genere raggruppate per strumenti specifici) atte a risolvere uno specifico problema di comunicazione tra diversi computer o tra diversi software o tra diversi componenti di software; spesso tale termine designa le librerie software di un linguaggio di programmazione[2], sebbene più propriamente le API sono il metodo con cui le librerie vengono usate per sopperire ad uno specifico problema di scambio di informazioni.

CSS

Il CSS (sigla di Cascading Style Sheets, in italiano fogli di stile a cascata), in informatica, è un linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML, ad esempio i siti web e relative pagine web. Le regole per comporre il CSS sono contenute in un insieme di direttive (Recommendations) emanate a partire dal 1996 dal W3C.

CSV

Il comma-separated values (abbreviato in CSV) è un formato di file basato su file di testo utilizzato per l'importazione ed esportazione (ad esempio da fogli elettronici o database) di una tabella di dati. In questo formato, ogni riga della tabella (o record della base dati) è normalmente rappresentata da una linea di testo, che a sua volta è divisa in campi (le singole colonne) separati da un apposito carattere separatore, ciascuno dei quali rappresenta un valore.

Debugging

Il debugging (o semplicemente debug), in informatica, nell'ambito dello sviluppo software, indica l'attività che consiste nell'individuazione e correzione da parte del programmatore di uno o più errori (bug) rilevati nel software, direttamente in fase di programmazione oppure a seguito della fase di testing o dell'utilizzo finale del programma stesso. L'attività di debug è una delle operazioni più importanti e difficili per la messa a punto di un programma,

spesso estremamente complicata per la complessità dei software in uso e delicata per il pericolo di introdurre nuovi errori o comportamenti difformi da quelli desiderati nel tentativo di correggere quelli per cui si è svolta l'attività di debug.

Deploy

Deployment, o deploy, è un termine della lingua inglese utilizzato in informatica, in particolare nel software, con diverse accezioni specifiche ma con il concetto generico di effettuazione di una distribuzione software. Il deployment del software è l'insieme delle attività che rendono un sistema software disponibile per l'utilizzo.

DOM

In informatica il Document Object Model (spesso abbreviato come DOM), letteralmente modello a oggetti del documento, è una forma di rappresentazione dei documenti strutturati come modello orientato agli oggetti. È lo standard ufficiale del W3C per la rappresentazione di documenti strutturati in maniera da essere neutrali sia per la lingua che per la piattaforma. È inoltre la base per una vasta gamma di interfacce di programmazione delle applicazioni, alcune di esse standardizzate dal W3C.

Domotica

È la scienza interdisciplinare che si occupa dello studio delle tecnologie adatte a migliorare la qualità della vita nella casa e più in generale negli ambienti antropizzati. Questa area fortemente interdisciplinare richiede l'apporto di molte tecnologie e professionalità, tra le quali ingegneria edile, architettura, ingegneria energetica, ingegneria gestionale, automazione, elettrotecnica, elettronica, telecomunicazioni, informatica e design.

Framework

In informatica e specificamente nello sviluppo software, è un'architettura logica di supporto (spesso un'implementazione logica di un particolare design pattern) sulla quale un software può essere progettato e realizzato, spesso facilitandone lo sviluppo da parte del programmatore. Talora è usato come sinonimo di rack o piattaforma software, anche nel gergo informatico.

HTML

In informatica l'HyperText Markup Language (traduzione letterale: linguaggio a marcatori per ipertesti), comunemente noto con l'acronimo HTML, è un linguaggio di markup. Nato per la formattazione e impaginazione di documenti ipertestuali disponibili nel web 1.0, oggi è utilizzato principalmente per il

disaccoppiamento della struttura logica di una pagina web (definita appunto dal markup) e la sua rappresentazione.

HTTP

In telecomunicazioni e informatica l'Hypertext Transfer Protocol (HTTP) (in italiano: protocollo di trasferimento ipertesto) è un protocollo a livello applicativo usato come principale sistema per la trasmissione d'informazioni sul web ovvero in un'architettura tipica client-server. Le specifiche del protocollo sono gestite dal World Wide Web Consortium (W3C). Un server HTTP generalmente resta in ascolto delle richieste dei client sulla porta 80 usando il protocollo TCP a livello di trasporto.

IDE

Un ambiente di sviluppo integrato (in inglese integrated development environment ovvero IDE), in informatica, è un software che, in fase di programmazione, supporta i programmatori nello sviluppo e debugging del codice sorgente di un programma.Spesso l'IDE aiuta lo sviluppatore segnalando errori di sintassi del codice direttamente in fase di scrittura, oltre a tutta una serie di strumenti e funzionalità di supporto alla fase stessa di sviluppo e debugging.

Internet of Things

L'Internet delle cose (IdC), in inglese Internet of Things (IoT), è un neologismo utilizzato nel mondo delle telecomunicazioni e dell'informatica che fa riferimento all'estensione di internet al mondo degli oggetti e dei luoghi concreti, che acquisiscono una propria identità digitale in modo da poter comunicare con altri oggetti nella rete e poter fornire servizi agli utenti. Si tratta dell'evoluzione del web stesso, il 3.0, inteso come la generalizzazione del Web of Things (o WoT) e come parte anche del web semantico e degli altri tipi di web.

JSON

Acronimo di JavaScript Object Notation, è un formato adatto all'interscambio di dati fra applicazioni client/server. È basato sul linguaggio JavaScript Standard, ma ne è indipendente.

Markup

Insieme di regole che descrivono i meccanismi di rappresentazione (strutturali, semantici, presentazionali) o impaginazione di un testo; facendo uso di convenzioni rese standard, tali regole sono utilizzabili su più supporti.

Mockup

Un mockup, o mock-up, è una realizzazione a scopo illustrativo o meramente espositivo di un oggetto o un sistema, senza le complete funzioni dell'originale; un mockup può rappresentare la totalità o solo una parte dell'originale di riferimento (già esistente o in fase di progetto), essere in scala reale oppure variata.

Plugin

Il plugin in campo informatico è un programma non autonomo che interagisce con un altro programma per ampliarne o estenderne le funzionalità originarie (ad es. un plugin per un software di grafica permette l'utilizzo di nuove funzioni non presenti nel software principale): possono essere utilizzati non solo su software, ma anche su qualunque cosa che possa essere visitata da chiunque, quindi pubblica (ad es. i videogiochi online).

Rendering

Nella computer grafica, il rendering identifica il processo di resa, ovvero di generazione di un'immagine a partire da una descrizione matematica di una scena tridimensionale, interpretata da algoritmi che definiscono il colore di ogni punto dell'immagine digitale. In senso esteso (nel disegno), indica un'operazione atta a produrre una rappresentazione di qualità di un oggetto o di una architettura (progettata o rilevata).

REST

Representational state transfer (REST) è uno stile architetturale per sistemi distribuiti. Il termine REST rappresenta un sistema di trasmissione di dati su HTTP senza ulteriori livelli, quali ad esempio SOAP. I sistemi REST non prevedono il concetto di sessione, ovvero sono stateless. L'architettura REST si basa su HTTP. Il funzionamento prevede una struttura degli URL ben definita che identifica univocamente una risorsa o un insieme di risorse e l'utilizzo dei metodi HTTP specifici per il recupero di informazioni (GET), per la modifica (POST, PUT, PATCH, DELETE) e per altri scopi (OPTIONS, ecc.).

Single-page Applications

In informatica con Single-page application (in italiano: applicazione su singola pagina) o in sigla SPA si intende un'applicazione web o un sito web che può essere usato o consultato su una singola pagina web con l'obiettivo di fornire una esperienza utente più fluida e simile alle applicazioni desktop dei sistemi operativi tradizionali. In un'applicazione su singola pagina tutto il codice necessario (HTML, JavaScript e CSS) è recuperato in un singolo caricamento

della pagina o le risorse appropriate sono caricate dinamicamente e aggiunte alla pagina quando necessario, di solito in risposta ad azioni dell'utente.

SVG

Scalable Vector Graphics (formato in .svg), indica un particolare formato che è in grado di visualizzare oggetti di grafica vettoriale e quindi di salvare immagini in modo che siano ingrandibili e rimpicciolibili a piacere senza perdere in risoluzione grafica. In particolare, il formato svg lavora in XML, cioè di un'applicazione del metalinguaggio posto a base degli sviluppi del Web da parte del consorzio W3C, che si pone l'obiettivo di descrivere figure bidimensionali statiche e animate.

UI

L'interfaccia utente (anche conosciuta come UI, dall'inglese User Interface) è un'interfaccia uomo-macchina, ovvero ciò che si frappone tra una macchina e un utente, consentendone l'interazione reciproca: in generale può riferirsi ad una macchina di qualsiasi natura, tuttavia l'accezione più nota è in ambito informatico con l'interazione utente-computer.

Acronimi

API

Application Programming Interface

CSS

Cascade Style Sheet

DOM

Document Object Model

HTML

HyperText Markup Language

IoT

Internet of Things

REST

Representational State Transfer

SPA

Single Page Application

Bibliografia

[1] React documentation.

https://it.reactjs.org/

[2] React, la libreria dedicata al frontend.

https://www.html.it/pag/55052/react-introduzione/

[3] React lifecycle methods.

https://programmingwithmosh.com/javascript/react-lifecycle-methods/

[4] Redux documentation.

https://redux.js.org/

[5] Redux in parole semplici.

https://davidecerbo.medium.com/redux-in-parole-semplice-6fec4a207c

[6] Node.js.

https://www.tophost.it/blog/node-js-tutorial-guida-in-italiano/

[7] Libreria D3.

https://d3js.org/

[8] Vantaggi D3.

https://isolution.pro/it/t/d3js/d3js-introduction/d3-js-introduzione

[9] Metodologia Agile.

https://monade.io/it/lapproccio-agile-allo-sviluppo-del-software/