**INTRODUZIONE E PRESENTAZIONE PRODOTTO**

Il nostro progetto è una web app che si propone l’idea di visualizzare dati a molte dimensioni in diversi tipi di grafico. Per ricevere i dati l’applicazione propone due modalità: tramite caricamento di file CSV o tramite database. Dopodiché l’utente può scegliere le dimensioni su cui vuole lavorare e può visualizzarle in cinque diversi tipi di grafico. Per evidenziare cluster o valori anomali è possibile modificare le preferenze.

**PERN STACK**

Lo scheletro del nostro progetto è composto dal PERN stack. Questo acronimo indica le tecnologie che sono state applicate, ovvero:

* **PostgreSQL**: che è il nostro database relazionale (struttura: la nostra applicazione vuole visualizzare più dataset, quindi non è una struttura complessa)
* **Express**: che utilizziamo lato server
* **React:** nel nostro progetto fornisce l’interfaccia utente
* **NodeJS** utilizzato per la codifica del back-end.

**Diagramma dei package piccolo**

La connessione tra client e server è cosí gestita: il server fornisce l’interfaccia per l’interrogazione del database attraverso delle API costruite con Express.js, mentre il client sfrutta Axios per la connessione al server. Grazie a delle funzioni richiamate nel viewmodel, vengono inviate le query di selezione al server che si occupa di interrogare il database, il quale restituisce il risultato dell’interrogazione.

**MOBX**

Per il client ci siamo subito interessati ad una architettura del tipo **mv\***, in particolare abbiamo ritenuto che il pattern **MVVM** fosse quello che faceva al caso nostro vista la sua ampia diffusione in ambito di web application ed il più consigliato da utilizzare con ReactJS : il framework da noi utilizzato per lo sviluppo dell'interfaccia utente.

Per implementare tale architettura abbiamo deciso di utilizzare la libreria Mobx, questa ci permette di implementare l’ observer pattern che di default non è presente in React.

L’idea di mobx è quella di fornire uno store, contenente lo stato dell’applicazione, alla parte di View attraverso un context e far si che questa si aggiorni ogni qualvolta lo stato cambi. Per fare ciò la libreria propone di marcare come observable oggetti e attributi e di utilizzare dei metodi marcati come computed per accedervi, mentre metodi marcati come action per modificarli. Ad ogni action su un oggetto observable viene inviata una reaction alla parte di View observer che quindi si aggiorna con lo stato corrente dello store.

**MVVM**

Questa libreria permetterebbe di costruire applicazioni con solamente Model e View, noi abbiamo però deciso di attuare il pattern MVVM, abbiamo quindi sviluppato un ViewModel per ogni View che necessita di interagire con lo store, e delegato a tali ViewModel la modifica dello store in base agli input dell'utente sulla Vista e anche la notifica del cambio di stato dello store alla View.

In questo modo siamo riusciti quindi a dividere la presentation logic dalla business logic utilizzando uno store, una vista composta da componenti react privi di logica e un viewmodel per ogni componente che si occupa di fare da tramite e contiene anche la logica della vista, eventualmente avremmo potuto suddividere ogni viewmodel in viewmodel e viewcontroller, ma abbiamo ritenuto che non fosse necessario in quanto l'applicazione non ha un'elevata logica di presentazione.

Per assicurare la SRP abbiamo deciso di utilizzare tre diversi store, che vengono forniti all’applicazione tramite un unico store padre **rootStore**, come suggerisce la documentazione di Mobx. Ogni ViewModel avrà il riferimento al rootStore ma utilizzerà solamente gli store di suo interesse, e notificherà quindi alla sua vista solamente i cambiamenti degli store che realmente utilizza.

**PACKAGE**

Da questo diagramma dei package si possono vedere bene le dipendenze dell’applicazione, la view ha dipendenza sul viewmodel e il viewmodel ha dipendenza sullo store e tutti e tre sono collegati a mobx per l’ observer pattern. La vista utilizza react, mentre il grosso delle dipendenze sono collegate al viewmodel che contiene tutta la logica dell’applicazione, vediamo quindi che sono collegati i services per la connessione al server, lo strategy pattern per la riduzione dimensionale e la libreria d3 per la costruzione dei grafici.

**DIAGRAMMA VISTA**

Per la vista il nostro punto d’ingresso è View, un componente react che viene utilizzato solamente per essere innestato nella pagina html e funziona da root per la vista, come consigliato dalla documentazione di React. Mentre i tre veri componenti principali sono Header, che è un componente statico privo di viewmodel, e Menu e Chart che gestiscono le due funzionalità principali dell'applicazione, il primo gestisce tutti i modali per il caricamento dei dati e per la riduzione dimensionale, oltre che la scelta di quale grafico visualizzare, mentre il secondo gestisce la renderizzazione del grafico e della relativa finestra per settarvi le preferenze. Più semplicemente dalla vista si possono vedere che ci sono diversi componenti figli di Menu, ma solamente uno potrà essere visualizzato in base a quale pulsante viene cliccato dall’utente, per visualizzarne un’ altro l'utente dovrà chiudere il componente visualizzato e cliccare su di un’ altro, lo stesso per Chart, in base alle preferenze dell’utente verrà mostrato un solo grafico e una sola finestra delle preferenze, Chart si occupa di cambiarle nel caso in cui le preferenze dell'utente cambino.

**DIAGRAMMA DI SEQUENZA**

Da questo diagramma infatti si può vedere come chart gestisce il rendering di un solo grafico alla volta, prima di tutto preleva dallo store il nome del grafico da visualizzare, e successivamente renderizza un componente per settare le preferenze ed il componente per il grafico, anche questi due componenti necessitano di prelevare le preferenze già settate dallo store per impostarle come valori di default e per fare ciò sfruttano il proprio ViewModel, in questo caso abbiamo assunto che il grafico da visualizzare sia il PLMA.

**DIAGRAMMA MODELLO**

Per quanto riguarda il modello, grazie all’utilizzo di MobX, è stato possibile creare 3 diversi store istanziati all’interno di un unico RootStore.

In particolare:

* **DatasetStore** che contiene i dati caricati e le dimensioni da visualizzare, fornendo i metodi per recuperarli e aggiornarli,
* **PreferencesStore** che contiene le preferenze di visualizzazione di tutti i grafici disponibili;
* **DistanceMatricesStore** che contiene le matrici delle distanze.

**Dimension** e **DistanceMatrix** sono invece i tipi che definiscono rispettivamente le dimensioni caricate e le matrici delle distanze create dall’utente attraverso le varie funzioni di calcolo della distanza.

**STRATEGY**

Per mostrare un esempio di funzionalità della web app, nella slide è mostrata la riduzione dimensionale sui dati.

Si può vedere come è stato utilizzato lo strategy pattern per l’implementazione dei vari algoritmi e come ora ogni componente della vista, il quale richieda la comunicazione con uno store, abbia il suo il viewmodel. In particolare...

**DimensionalReduction** è il componente React che si occupa solo di ritornare gli elementi HTML che compongono questa parte di vista;

**DimensionalReductionVM** è il rispettivo view-model che ne contiene tutta la logica. Questo prende i dati dal DatasetStore, crea i parametri da utilizzare a seconda dell'algoritmo scelto dall’utente e crea il contesto **DimReduction** chericeve i dati e i parametri per la riduzione dimensionale.

DimReduction si occupa solo di chiamare attraverso *executeStrategy()* il metodo *startDR()* sull’algoritmo corretto (impostato dal viewmodel settando l'attributo *strategy* del contesto con l'algoritmo scelto dall’utente). Quindi il contesto mantiene il riferimento all’algoritmo da utilizzare, senza però sapere effettivamente quale andrà a chiamare.

*startDR()* infine esegue la riduzione dimensionale utilizzando i metodi forniti dalla libreria Druid.js e ritorna le nuove dimensioni.

Riassumendo è stato utilizzato il design pattern observer (implementato con MobX) per l’aggiornamento della vista al cambiamento degli attributi observable negli store, e lo strategy per rendere l’implementazione degli algoritmi di riduzione dimensionale facilmente estendibile. L’utilizzo di altri design pattern è risultato invece a nostro parere forzato e quindi non effettivamente utile per la risoluzione dei problemi progettuali.