ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di un'Interfaccia Grafica per Software Simulativi Complessi mediante GraphQL e KotlinJS

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Dott. Danilo Pianini

Candidato
Tiziano Vuksan

Correlatore
Dott. Angelo Filaseta

IV Sessione di Laurea
Anno Accademico 2023-2024

Sommario

Max 2000 characters, strict.

Indice

Elenco delle figure

List of Listings

Capitolo 1

Introduzione

1.1 Contesto

1.1.1 Simulazione

1.1.2 Alchemist

1.2 Motivazione

1.3 Obiettivi

Capitolo 2

Analisi

2.1 Analisi dei requisiti

Lo scopo principale del progetto è la realizzazione di una interfaccia web (quindi interpretabile da un qualsiasi browser moderno) che permetta l'interazione con il sistema software di simulazione *Alchemist* in modo intuitivo e *user-friendly*. Il compito dell'applicativo sarà quindi quello di comunicare, attraverso apposite Application Program Interface (API), con l'infrastruttura server preesistente e presentare in seguito a cambiamenti della simulazione in corso o a richieste da parte dell'utente, un'interfaccia grafica che ne rappresenti i risultati.

2.1.1 Requisiti funzionali

- L'applicativo dovrà presentare un interfaccia grafica all'interno di un web browser.

- In una tipica simulazione di *Alchemist* (come discusso nel paragrafo) sono presenti dei nodi. L'applicativo quindi dovrà essere in grado di rappresentare in un piano bidimensionale la posizione di tali nodi all'interno di un contesto grafico. Ciò implica ovviamente che con l'evolversi della simulazione il contesto grafico debba essere aggiornato. [1]

[1] Date le diverse *incarnation* e i diversi possibili scenari che *Alchemist* può modellare non è detto che i nodi cambino di posizione.

- Ogni nodo contiene diverse proprietà, reazioni, concentrazioni etc. L'interfaccia dovrà permettere di ispezionare il contenuto di ciascun nodo.

- L'interfaccia dovrà controllare lo stato attuale della simulazione. Ciò vuol dire poterla eseguire o mettere in pausa.

2.1.2 Requisiti non funzionali

- Interagendo con l'interfaccia, non si devono verificare tempi di risposta eccessivi. Per esempio se l'utente decide di ispezionare un nodo, il recupero di tali informazioni deve essere presentato in tempi ragionevoli.

- L'applicativo deve essere compatibile con un ambiente multipiattaforma.

- L'architettura delle componenti grafiche deve essere estendibile e facilmente modificabile.

2.2 Requisiti implementativi

Per la realizzazione di questo progetto, è stato necessario tener conto di due requisiti implementativi, tra cui l'uso di KotlinJS come linguaggio di sviluppo dell'interfaccia grafica e l'utilizzo del modulo API GraphQL già esistente per instaurare una comunicazione con la simulazione.

2.2.1 Tecnologie per lo sviluppo web

Nel mondo dello sviluppo web esistono diverse tecnologie in grado di fornire gli strumenti necessari alla creazione di una User Interface (UI). Tecnologie che si evolvono costantemente per migliorare l'efficienza e la manutenibilità. È importante quindi analizzare con attenzione gli strumenti che verranno adoperati per la realizzazione di un applicativo, soprattutto nel caso questo debba essere integrato a un software costantemente controllato e aggiornato. In questa sezione esploreremo due linguaggi che stanno guadagnando sempre più popolarità per lo sviluppo di applicazioni frontend: **TypeScript** e **KotlinJS**.

TypeScript TypeScript[2] è un linguaggio di programmazione sviluppato da Microsoft nel 2012 che estende le funzionalità di JavaScript, rendendolo un linguaggio con tipizzazione statica, ovvero che il tipo di ogni variabile viene verificato in fase di compilazione. Non a caso, tra gli errori più comuni durante la scrittura di codice da parte dei programmatori vi è il cosiddetto *type error*. Quest'ultimo si verifica nel momento in cui si tenta di utilizzare un valore in un contesto dove il tipo di dato non è compatibile con quello richiesto. JavaScript non è un linguaggio tipizzato e nasce come un semplice linguaggio di scripting per aggiungere un livello di interattività basilare alle pagine web. Con gli anni è diventato poi il linguaggio di scelta sia per le applicazioni frontend che backend. Sebbene la dimensione e la complessità delle applicazioni scritte in questo linguaggio siano cresciute esponenzialmente, le capacità di JavaScript sono rimaste pressoché inalterate. Obiettivo di TypeScript è pertanto quello di imporre un maggiore rigore durante la scrittura di codice, assicurandosi che i tipi del programma siano corretti prima che il codice venga eseguito, migliorando robustezza e chiarezza del codice. Come risultato, i file sorgente scritti in TypeScript vengono tradotti in puro JavaScript.

KotlinJS KotlinJS[3] fornisce la possibilità di tradurre il codice Kotlin, insieme alla sua libreria standard e a qualsiasi libreria compatibile, in codice JavaScript. Ha origine come parte del progetto *Kotlin Multiplatform*, che mira allo sviluppo di applicazioni su diverse piattaforme utilizzando come unico linguaggio di programmazione Kotlin stesso. Possiamo pertanto elencare una serie di peculiarità:

- **Interoperabilità con JavaScript**: consente una facile integrazione con l'ecosistema JavaScript, anche utilizzandone librerie e framework tipiche (e.g. React).
- **Tipizzazione statica**: così come TypeScript, Kotlin è un linguaggio con typing statico.
- **Leggibilità**: Kotlin è noto per la sua sintassi chiara e concisa, che può rendere il codice più leggibile rispetto ad altri linguaggi.

[2] `https://www.typescriptlang.org/docs/handbook/intro.html`
[3] `https://kotlinlang.org/docs/js-overview.html`

Conclusioni Presi singolarmente, se si considerano i due aspetti principali di entrambi, ergo la tipizzazione statica e la diretta traduzione in linguaggio JavaScript, i due linguaggi offrono sostanzialmente gli stessi vantaggi. Da una parte Kotlin compilato per il target JavaScript è relativamente nuovo (marzo 2017)[4], il che non lo rende tanto maturo quanto TypeScript, che vanta risorse e community più ampie. Dall'altra parte invece Kotlin offre una sintassi più espressiva e concisa, riducendo la quantità di codice necessaria per compiti comuni. Gli aspetti decisivi che hanno portato alla scelta di KotlinJS rispetto all'utilizzo di TypeScript sono due:

1. **Compatibilità con progetti multipiattaforma**: KotlinJS offre la possibilità di condividere il codice con progetti Kotlin che mirano anche alla JVM, consentendo un riuso efficiente del codice tra frontend e backend.

2. **Codebase preesistente**: questo aspetto, decisivo, è dettato dalla necessità di garantire coerenza con l'ecosistema tecnologico esistente, considerando che l'attuale *codebase* del progetto *Alchemist* è per buona parte già scritto nel linguaggio Kotlin.

2.2.2 Modulo GraphQL

Come già anticipato, l'applicativo dovrà interfacciarsi con l'infrastruttura di API GraphQL preesistente all'interno del progetto. Prima di analizzare il funzionamento principale dell'architettura server è utile capire le motivazioni dietro all'utilizzo di GraphQL e il contesto per il quale nasce.

GraphQL GraphQL[5] è un linguaggio di interrogazione per le API che offre una sintassi flessibile e potente per recuperare dati da un server, creato da Facebook nel 2012 come alternativa all'esistente architettura Representational State Transfer (REST). Evidenziamo quindi i punti di forza più pertinenti:

- **Flessibilità nelle query**: i client possono richiedere esattamente i dati di cui hanno bisogno, evitando di occupare, nelle richieste di dati, più banda di

[4] `https://blog.jetbrains.com/kotlin/2017/03/kotlin-1-1/`
[5] `https://graphql.org/foundation/`

rete del necessario. Con questo linguaggio vengono risolti quindi i problemi di *over-fetching* e *under-fetching*.

- **Unica endpoint**: mentre nelle architetture di tipo REST i dati sono esposti tramite endpoint dedicati che corrispondono ciascuno a una risorsa specifica (identificati tramite un Uniform Resource Identifier (URL) univoco), in GraphQL l'interrogazione dei dati avviene tramite un unico *endpoint*.

- **Tipizzazione forte**: GraphQL offre una tipizzazione forte dei dati, consentendo ai client di conoscere in anticipo i tipi di dati che riceveranno in risposta alle loro query. Questo porta a un maggiore controllo e previsione durante lo sviluppo delle applicazioni.

È facile quindi notare come questo linguaggio semplifichi lo sviluppo di applicazioni frontend.

L'interazione con i dati avviene attraverso tre operazioni:

- **Query**: operazione di lettura per ottenere un tipo determinato di dato dal server.

- **Mutation**: operazione di scrittura per modificare uno o più dati sul server.

- **Subscription**: operazione per ricevere i cambiamenti di uno o più tipi di dati in tempo reale.

Server GraphQL

Al centro delle operazioni GraphQL c'è uno schema che definisce tutti i tipi di dati disponibili e le relazioni che ci sono tra di essi, oltre che alle operazioni che possono essere eseguite. La natura intrinseca dello schema garantisce che fra client e server ci sia un meccanismo di *type safety* che previene errori legati a richieste che non sono compatibili. Per questo, uno strumento molto utile messo a disposizione dal web server, accessibile tramite l'*endpoint* `/graphiql`, è il *playground* GraphiQL. Qui è possibile effettuare e verificare *ex-ante* il risultato delle operazioni che si intendono fare prima che queste vengano usate per generare le classi associate durante la fase di compilazione del progetto. Questo processo, per lo sviluppo

di un qualsiasi applicativo client che si appoggia su queste API, permette allo sviluppatore di validare ogni singola operazione che verrà utilizzata all'interno dell'applicativo che si intende sviluppare.

2.2.3 Rendering del contesto grafico

È importante sottolineare come le prestazioni siano un fattore decisivo nella scelta delle tecniche per rappresentare l'ambiente della simulazione. In questo contesto, prestazioni ottimali assicurano un'esperienza utente fluida e soddisfacente. È per questo motivo che la scelta di disegnare i nodi della simulazione di *Alchemist* direttamente all'interno di un elemento di tipo *canvas* HTML prevale rispetto alla rappresentazione tramite elementi Document Object Model (DOM). Nell'ipotesi in cui si decidesse di rappresentare ciascun nodo con un elemento del DOM (e.g. `div`), il *rendering* risulterebbe oneroso, perché ogni elemento DOM aggiunto alla pagina web richiederebbe risorse di sistema per essere gestito e disegnato dal browser. Con un considerevole numero di nodi, oltretutto aggiornati frequentemente, questo metodo causerebbe solo un deterioramento delle prestazioni. Inoltre, utilizzando un *canvas*, si ha maggiore flessibilità nel disegno dei nodi e nel loro comportamento. Si può disegnare senza nessun vincolo una qualsiasi forma o figura, applicare trasformazioni ed effetti visivi senza doversi occupare delle restrizioni del DOM. D'altro canto, agli elementi del DOM possono essere collegati dei *listener*, funzioni associate a un determinato tipo di evento, come un click o il movimento del mouse. Sebbene questo vantaggio, il *canvas* torna più utile in questa situazione perché offre la possibilità di implementare interazioni più complesse, come per esempio lo zoom del contesto (modifica della scala) o la traslazione dell'intera area di disegno. La realizzazione di queste funzionalità senza l'utilizzo del canvas implicherebbe il recupero degli oggetti dal DOM e la successiva manipolazione delle loro proprietà. Il processo di rendering dei nodi sul *canvas* è illustrato in figura fig. 2.1. Inizialmente, il sistema apre una connessione con il client GraphQL e si iscrive alla richiesta di invio dei nodi. Il server accetta la richiesta d'iscrizione e invia dati fino a quando la *subscription* non viene cancellata o completamente consumata. A ogni iterazione il web client ridisegna i nodi nel *canvas*. A ogni fase della simulazione i nodi possono cambiare di posizione o meno, in base al tipo di

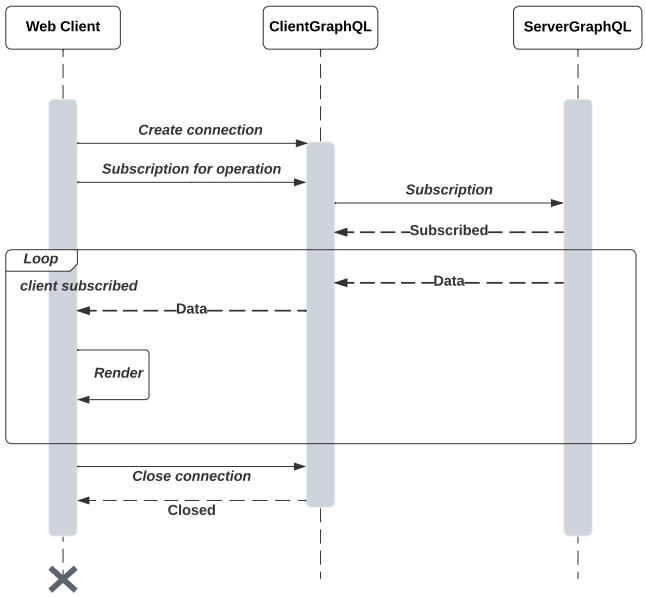

Figura 2.1: Diagramma di sequenza del rendering dei nodi a seguito di una *subscription*

simulazione che è in esecuzione.

2.2.4 Progetto multiplatform

Kotlin Multiplatform[6] è una tecnologia che permette lo sviluppo di codice Kotlin condivisibile tra diverse piattaforme, come per esempio Android, iOS, web e desktop. Questo significa che è possibile utilizzare lo stesso codice Kotlin per creare applicazioni native per diverse piattaforme, riducendo la necessità di scrivere e mantenere codice separato per ciascuna piattaforma. Caratteristica dei progetti multipiattaforma è che sono composti dai cosiddetti *source sets*: insiemi di codice sorgente specifici alla piattaforma a cui si riferiscono. Generalmente è sempre presente il modulo comune, detto "common code". Questo modulo contiene il codice condiviso che può essere utilizzato su tutte le piattaforme. Ad accompagnarlo quindi sono altri *source sets* aggiuntivi, compilati per target diversi come Java Vir-

[6] `https://kotlinlang.org/docs/multiplatform-discover-project.html`

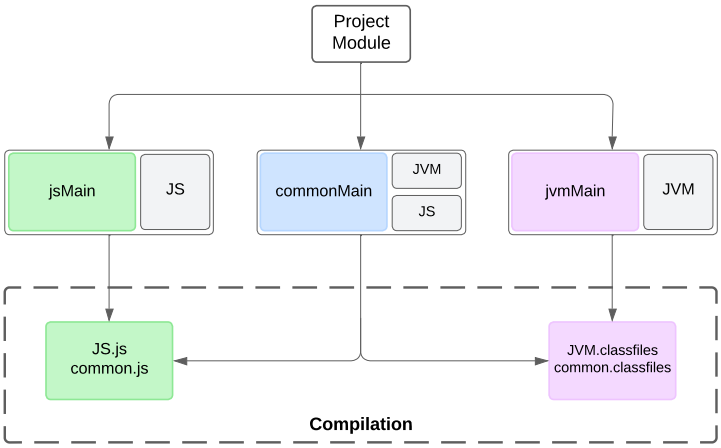

Figura 2.2: Struttura di un progetto multipiattaforma compilato per KotlinJS e KotlinJVM

tual Machine (JVM), JavaScript o nativo. Al momento della compilazione di un progetto multi-piattaforma per un target specifico, Kotlin raccoglie tutti i *source sets* contrassegnati con quel target e produce da essi i file binari (come file *.jar* per JVM, file *.js* per JavaScript, ecc.) che possono essere utilizzati nell'ambiente di destinazione corrispondente. Questo approccio consente una maggiore flessibilità e compatibilità nella creazione di applicazioni multi-piattaforma utilizzando Kotlin.

Nel caso dello sviluppo di una applicazione in-browser il progetto potrebbe includere i seguenti *source set*:

- **Common Source Set**: Codice condiviso che può essere utilizzato su tutte le piattaforme target.

- **JVM Source Set**: Codice destinato al target JVM. Da qui può essere gestita una componente server dal quale sarà accessibile l'interfaccia grafica.

- **JavaScript Source Set**: insieme di file sorgente destinato al target JavaScript. Il codice scritto in linguaggio Kotlin viene compilato per il target JavaScript (da qui la denominazione KotlinJS). In questo sottomodulo viene definita la struttura e il comportamento dell'interfaccia grafica vera e propria.

2.3 Analisi e Modello del Dominio

Uno dei requisti di questo applicativo verte sul bisogno di creare una piattaforma web tale da impiegare le API esposte dall'infrastruttura GraphQL fornita. Pertanto non esiste una comunicazione diretta tra il web client e la simulazione di *Alchemist*. La gestione dell'accesso e del recupero dei dati dal modello della simulazione è affidata alla componente server, che, al contempo, fornisce anche un punto di accesso ai client che richiedono tali dati. Il web client quindi comprende un modulo (nell'immagine ClientGraphQL) che si pone da interfaccia tra l'applicazione web vera e propria e l'*endpoint* sulla quale la componente server utilizzerà per ricevere richieste e mandare risposte (*endpoint* `/graphql`). Il compito del client-web quindi sarà quello di utilizzare le operazioni possibili (*query*, *mutation* e *subscription*) e utilizzare i risultati per rappresentarli graficamente.

Alchemist presenta già altri moduli che rappresentano l'ambiente di simulazione, implementati con tecnologie differenti a quelle che verranno proposte successivamente in questo elaborato. Il modulo specifico che viene avviato per rappresentare la simulazione dipende dalla configurazione con cui viene avviato l'intero software. Di conseguenza, è opportuno che l'interfaccia web e la componente server vengano avviate esclusivamente attraverso una specifica configurazione della simulazione.

Capitolo 3

Design

3.1 Layout dell'interfaccia

?? Il layout dell'interfaccia grafica è stato pensato per rappresentare nel modo più semplice ed intuitivo l'ambiente della simulazione. La figura 3.1 rappresenta un mockup utilizzato durante la fase di progettazione dell'interfaccia. Si possono individuare le seguenti sezioni:

- **Barra di navigazione**: nella parte alta dell'interfaccia è presente una barra di navigazione contenente il titolo e il pulsante per avviare o mettere in pausa la simulazione, ancorato all'estrema destra. Molte interfacce web moderne presentano questo tipo di elemento come *header* della pagina web principale, inteso come punto centrale dal quale è possibile accedere a tutte le sezioni e funzionalità. Questo fornisce all'interfaccia un punto di espandibilità dell'applicativo, come l'aggiunta di una barra di ricerca o di un menù detto ad "hamburger". Sarebbe stato possibile, per esempio, inserire una barra di ricerca per i nodi, filtrandoli per categorie di proprietà. Questo tipo di funzionalità è indirizzato a lavori futuri.

- **Canvas grafico**: la sezione principale di questa interfaccia. All'interno di un contesto grafico bidimensionale vengono rappresentati i nodi della simulazione. Ogni nodo è rappresentato come un cerchio pieno, avente centro le coordinate del nodo e raggio un valore variabile che può essere impostato

dall'utente nella sezione descritta successivamente. Lo spazio bidimensionale ha come sfondo una griglia, che fornisce un riferimento visivo e un aiuto all'orientamento. Funzionalità non banale di questa sezione è che l'utente può spostare il contesto visivo trascinando il cursore sullo schermo, oltre che a effettuare un ingrandimento o una diminuzione della scala. Per ottenere questo tipo di comportamenti sono stati adottati meccanismi ad hoc per il calcolo dello spostamento del *drag* e dello *zoom-in/zoom-out*. Infine, facendo click su un nodo è possibile selezionarlo, andando a riportare nella sezione di ispezione del nodo tutte le sue caratteristiche principali.

- **Informazioni e controlli sul canvas**: in questa sezione vengono raccolte le principali informazioni riguardo allo stato attuale del *canvas*, come il fattore di *zoom* corrente, la differenza di traslazione rispetto all'origine e la grandezza del raggio utilizzato per rappresentare i nodi. Il fattore di scala è riportato, oltre nella sua forma numerica anche tramite una barra di progresso, la cui lunghezza varia in base alla percentuale di zoom raggiunta rispetto al massimo. Altro oggetto con cui l'utente può interagire è uno *slider*, al variare del quale viene aggiornato il raggio utilizzato per disegnare i nodi nel *canvas*. I parametri legati al *rendering* (fattore minimo e massimo di scala, altezza e larghezza del canvas, numero di iterazioni di scala etc.) sono raggruppati in un unico oggetto e quindi aperti a modifiche.

- **Sezione di ispezione di un nodo**: qui vengono rappresentate tutte le informazioni riguardanti un nodo. Sono presenti quindi il codice identificativo, posizione nello spazio bidimensionale, proprietà, i contenuti (intesi come una lista di molecole alle quali vengono associate le relative concentrazioni), e le reazioni (Vedi **??**). Per le ultime tre categorie sono stati usati degli elementi grafici che possono essere espansi o "collassati" in quanto non è garantito che queste proprietà siano presenti (sempre per il fatto che *Alchemist* può rappresentare una certa gamma di simulazioni tra loro eterogenee).

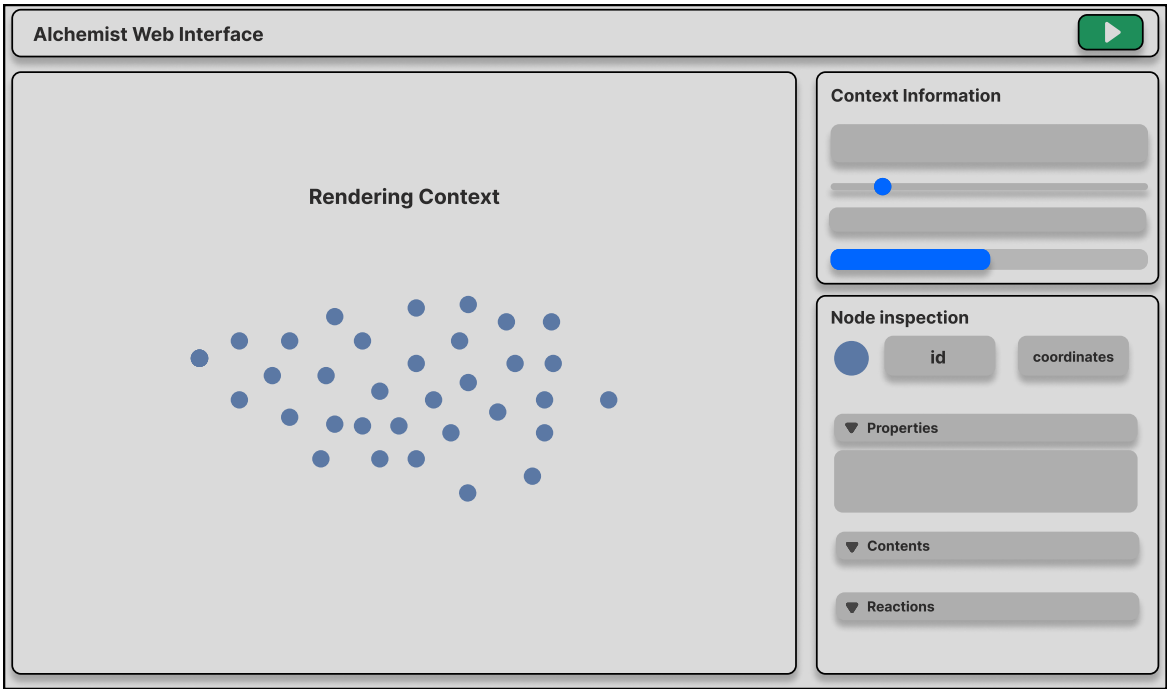

Figura 3.1: Mockup dell'interfaccia grafica

3.2 Connessione al server GraphQL

In questa sezione viene esplorato come l'interfaccia web interagisce con le API GraphQL per effettuare operazioni sul server e per poi usare i risultati di suddette operazioni per mostrarli graficamente. Nella figura 3.2 vengono mostrate le principali componenti protagoniste di questo meccanismo. Le descriviamo in questo modo:

- **Client Application**: questo *package* contiene tutte le componenti grafiche che vengono rappresentate all'interno della pagina principale. Ogni componente, una volta che l'applicativo viene avviato, è tradotto al browser in formato HTML.
- **ClientConnection**: punto di accesso attraverso il quale è possibile effettuare tutte le operazioni definite secondo lo schema GraphQL. All'interno di questo oggetto è dichiarata l'unica istanza per l'intero progetto che funge da punto di accesso per le operazioni sul server secondo lo schema definito.

- **SimulationControlApi**: questo oggetto contiene tutte le funzioni necessarie a controllare lo stato della simulazione e dipende strettamente dalla componente **ClientConnection**. Si parla quindi di funzioni utili all'avvio, alla sospensione e terminazione della simulazione. Notare come queste siano tutte operazioni di tipo *mutation*.

- **EnvironmentApi**: è l'oggetto utile a recuperare le informazioni riguardanti un nodo, lo stato *attuale* dell'*Environment*, ma soprattutto utile a recuperare la posizione dei nodi in tempo reale, quindi attraverso l'utilizzo di una *subscription*.

- **GeneratedModel**: questo pacchetto contiene tutte le risorse generate a partire dallo schema GraphQL esposto dal server. È utilizzato dagli oggetti **EnvironmentApi** e **SimulationControlApi** nell'utilizzo dei tipi di dato corretto durante l'utilizzo delle operazioni sul server.

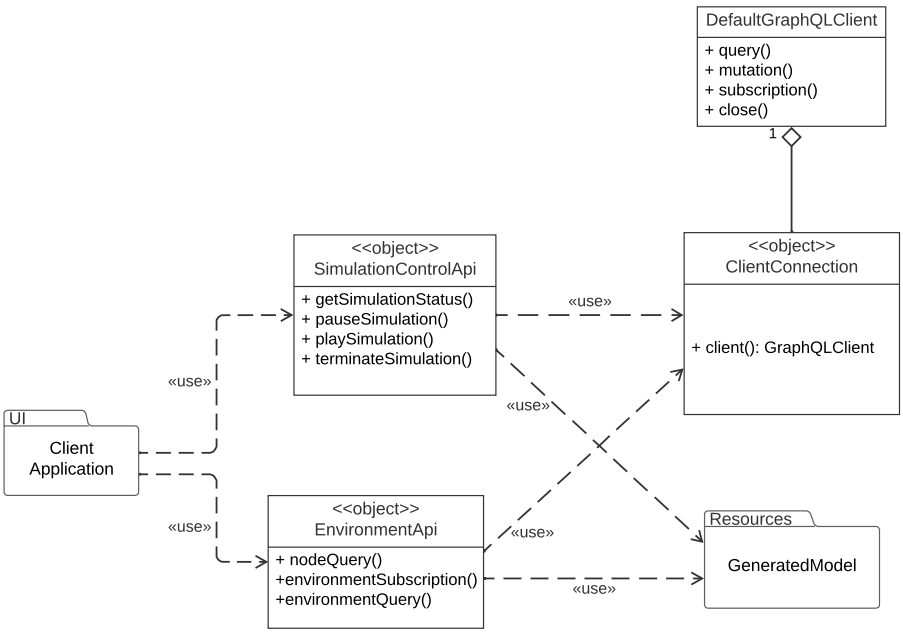

Figura 3.2: Architettura generale del client web

Gli oggetti **SimulationControlApi** e **EnvironmentApi** sono stati implementati attraverso il design pattern *Singleton* [EG94]. Sebbene quest'ultimo, se

abusato o implementato in modo non adeguato sia considerato di fatto un "anti-pattern" [1], in questa situazione risulta essere molto comodo, specialmente considerando la necessità di un unico punto di accesso comune al client che effettua le query sul server. Risulterebbe infatti inutile, per ogni componente grafico che ne necessita, istanziare un'altro client GraphQL dal quale effettuare query. Lo stesso vale anche nell'ipotesi in cui vengano utilizzate delle proprietà che fungono da parametri di configurazione dell'applicativo. Un *Singleton* può fornire un punto centralizzato per queste impostazioni.

3.3 Architettura generale client web

Il diagramma UML in figura mostra la soluzione adottata alla necessità di ospitare la pagina web finale all'interno di un browser web. Dallo schema è possibile individuare le seguenti sezioni:

- **OutputMonitor**: interfaccia di *Alchemist* che fornisce un modo flessibile per osservare la progressione delle simulazioni tramite l'esposizione di *hook standard*. Quest'ultimi si riferiscono a punti predefiniti all'interno del ciclo di vita della simulazione (avvio della simulazione, fine di ogni passo della simulazione, fine della simulazione) ai quali è possibile collegare meccanismi personalizzati. Questa interfaccia aderisce al design pattern *Observer* [EG94].

- **GraphQLServer**: implementazione dell'interfaccia `OutputMonitor`. Questa classe avvia il server GraphQL all'avvio della simulazione e si assicura che al termine della simulazione il server venga chiuso.

- **WebUIMonitor**: estensione della classe `GraphQLServer`, che avvia il server sul quale viene presentata la pagina web contenente l'interfaccia grafica esplorata in sezione **??**. Come per la classe da cui eredita, al momento della terminazione della simulazione, il server viene chiuso. L'estensione alla classe `GraphQLServer` permette che la simulazione venga configurata con

[1] `https://code.google.com/archive/p/google-singleton-detector/wikis/WhySingletonsAreControversial.wiki`

`WebUIMonitor` come `OutputMonitor`, che, in thread separati, avvia sia il server GraphQL che il server che ospita la pagina web che rappresenta l'interfaccia grafica. In questo contesto, è necessario configurare una nuova *route* nel server per caricare la pagina principale `index.html`. Per semplificare il processo, questa *route* verrà mappata per rispondere alle richieste GET alla radice ("`/`"), servendo la risorsa `index.html`. Ciò consente ai client di accedere direttamente alla pagina specificando l'indirizzo e la porta del server.

- **Generated Artifacts**: questo pacchetto include tutti i file necessari alla composizione di un unico file di output, con estensione `.js` (processo noto anche come *bundling*). Il file che ne risulterà verrà servito al server in modo statico. Il server può essere configurato per andare a recuperare tutti gli artefatti necessari al *bundling* a partire da un percorso remoto specificato. Ciò significa che nel caso fossero stati dichiarati dei file CSS o JavaScript separati questi sarebbero stati comunque coinvolti nella generazione del file di output e sarebbero stati accessibili tramite URL che iniziano dal percorso remoto (in questo caso "`/`"). Ad esempio, se il pacchetto base contiene un file chiamato "`styles.css`" e il percorso remoto è "`/static`", il file "`styles.css`" può essere accessibile all'URL "`static/styles.css`" nell'applicazione.

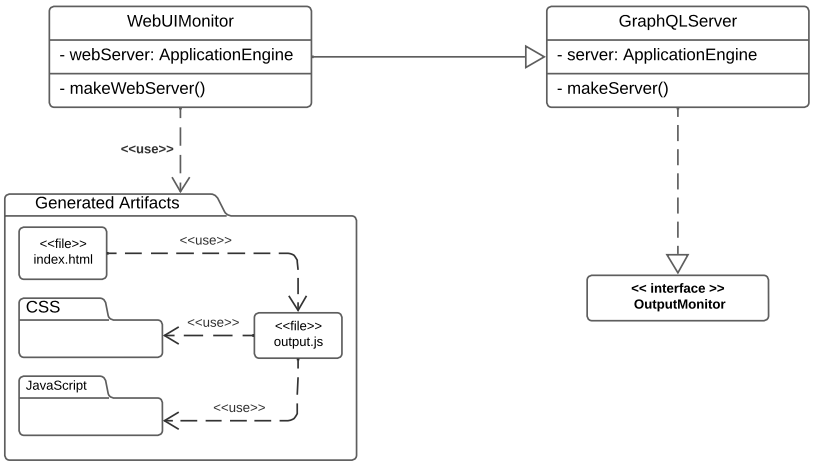

Figura 3.3: Architettura generale del client web

3.4 Struttura della pagina web

Capitolo 4

Implementazione e Verifica

4.1 Componenti grafici

4.2 Gestione dello stato

Il framework *KVision*, oltre a fornire strumenti e metodi di programmazione molto robusti e versatili, dona la possibilità di usare tutta la capacità della libreria di *Redux*[1], una libreria open-source per la gestione dello stato delle applicazioni JavaScript. Il fulcro di questa libreria consiste nel cosiddetto *store*, un archivio centralizzato per uno stato che deve essere condiviso in tutta l'applicazione, attraverso l'utilizzo di regole che garantiscono che lo stato possa essere aggiornato solamente in modo prevedibile. Analizziamo ora il funzionamento della gestione dello stato introducendo tutti i concetti chiave:

- **State**: "The source of truth that drives our app", in altre parole, dati o insieme di dati che influenzano il comportamento o l'aspetto dell'applicazione.
- **View**: una descrizione dichiarativa dell'interfaccia utente basata sullo stato attuale.
- **Actions**: usati per descrivere possibili cambiamenti dello stato. Sono oggetti, dotati di un campo che ne indica il tipo, incaricati a indicare l'azione

[1] `https://redux.js.org/tutorials/essentials/part-1-overview-concepts`

che deve essere eseguita sullo store per cambiarne lo stato. Si può pensare a questo tipo di oggetti come ad eventi che riportano un certo avvenimento nell'applicazione. Di solito vengono chiamati dopo un input, ovvero quando si verifica un evento specifico nell'applicazione, come un click del mouse o quando un pulsante viene premuto.

- **Store (archivio)**: l'oggetto centrale che contiene lo stato dell'applicazione in un determinato momento.

- **Reducers**: funzioni che descrivono esattamente come deve essere cambiato lo stato, in risposta alle azioni chiamate sullo store. Come parametri di ingresso accettano lo stato corrente e l'azione che si vuole eseguire, ritornando un nuovo stato. I reducer possono essere paragonati a dei *listener* che gestiscono gli eventi (*actions*) in base al loro tipo.

- **Dispatch**: metodo di cui lo store dispone. L'unico modo per aggiornare lo stato è invocando questa funzione, passando come parametro un'azione. A questo punto lo store può eseguire il suo *reducer* e salvare il nuovo stato al suo interno. Nel momento in cui un evento viene innescato (proveniente per esempio della UI), si vuole di conseguenza aggiornare il valore dello store.

- **Subscribe**: sono funzioni che permettono ai componenti grafici di "iscriversi" ai cambiamenti di stato dello store. Ogni volte che lo stato cambia, i componenti iscritti vengono avvisati, dando la possibilità di aggiornare la UI in base al nuovo cambio di stato.

Insieme agli elementi appena descritti, la figura fig. 4.1 riesce a dare una visione più completa. Dallo schema è possibile notare come sia presente una unica direzione che collega le varie entità. Questo è dovuto al fatto che la sequenza dei passi da compiere per l'aggiornamento dello stato segue il paradigma "one-way data flow". In particolare, per Redux si seguono quindi questi passaggi:

1. Qualcosa accade nell'applicazione, come un utente che fa clic su un pulsante.

2. Viene chiamata la funzione `dispatch`, specificando l'azione per modificare lo store Redux.

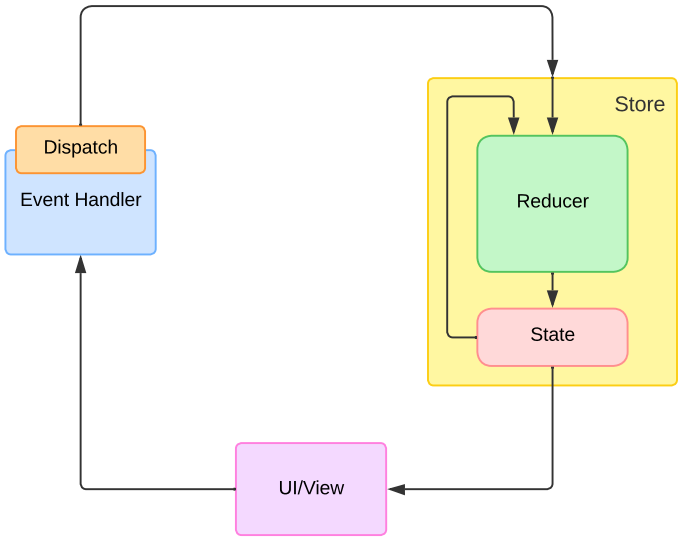

Figura 4.1: Architettura generale del client web

3. Lo store esegue la funzione reducer con lo stato precedente e l'azione corrente, e salva il valore restituito come nuovo stato.

4. Lo store notifica a tutte le parti dell'interfaccia utente che sono iscritte che lo store è stato aggiornato.

5. Ciascun componente dell'interfaccia utente che necessita dei dati dallo store controlla se le parti dello stato di cui hanno bisogno sono cambiate.

6. Ciascun componente che rileva che i suoi dati sono cambiati forza un nuovo rendering con i nuovi dati, così da poter aggiornare ciò che viene mostrato sullo schermo.

Per mantenere l'informazione ottenuto dalla query sullo stato di un nodo lo store che ne deriva viene dichiarato come in section 4.2.

```
1 val nodeStore = createTypedReduxStore(::nodeReducer, NodeState())
```

.

`createTypedReduxStore` è una funzione della libreria di *KVision* che permette la creazione

4.3 Integrazioni di operazioni GraphQL

4.4 Verifica

Capitolo 5

Conclusione

5.1 Lavori futuri

Bibliografia

[EG94] Ralph Johnson John M. Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.