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.

Optional. Max a few lines.

Acknowledgements

Optional. Max 1 page.

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.

- L'applicativo dovrà interfacciarsi con l'infrastruttura GraphQL già presente all'interno del progetto.

- 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 conte-

sto grafico. Ciò implica ovviamente che con l'evolversi della simulazione il contesto grafico debba essere aggiornato. [1]

- 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.

2.1.3 Architettura API GraphQL

Prima di analizzare il funzionamento principale dell'architettura server esistente è utile capire brevemente le motivazioni dietro l'utilizzo di GraphQL e il contesto per il quale nasce. GraphQL[2] 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 rete del necessario. Con questo linguaggio vengono risolti quindi i problemi di *over-fetching* e *under-fetching*.

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

[2] `https://graphql.org/foundation/`

- **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.

2.1.4 Web 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.

2.1.5 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. 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 simulazione che è in esecuzione.

2.1.6 Progetto multiplatform

Kotlin Multiplatform 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

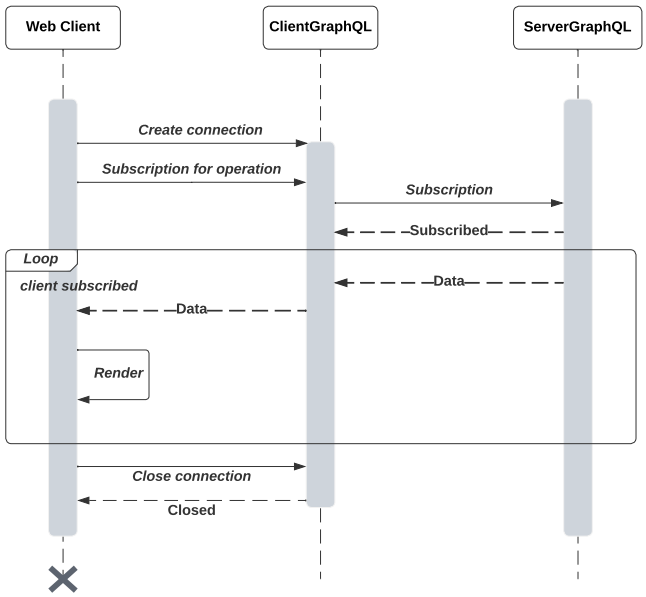

Figura 2.1: Diagramma di sequenza del rendering dei nodi a seguito di una *subscription*

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 specifico della 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 Virtual Machine (JVM), JavaScript o nativo. 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**: Codice destinato al target JavaScript. Qui viene definita la struttura e il comportamento dell'interfaccia grafica vera e propria.

2.2 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 client-web 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 Architettura generale client web

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 3.1 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.
- **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 recupe-

rare 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.

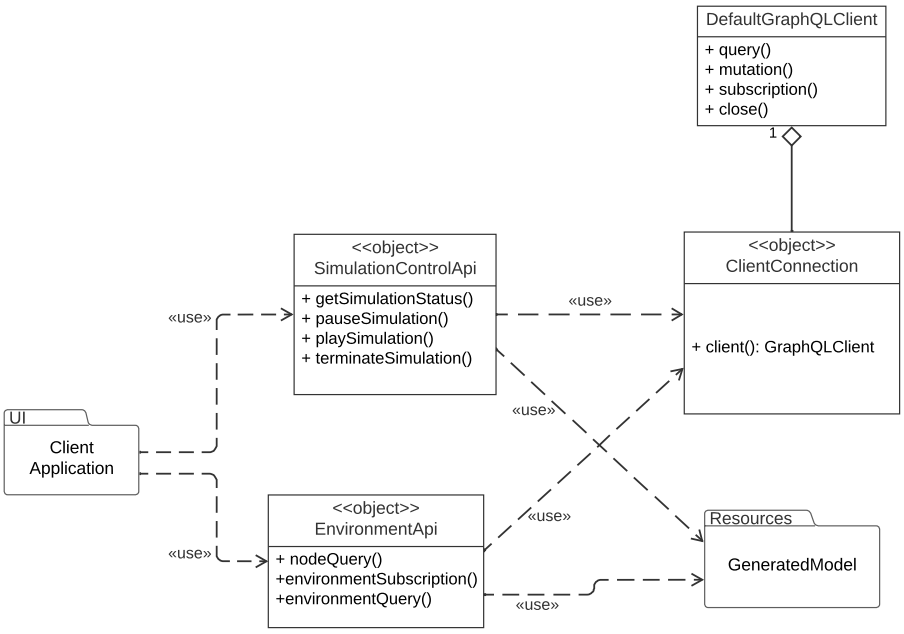

Figura 3.1: Architettura generale del client web

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

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

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.

3.2

3.3 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 3.2 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*.

- **Informazioni e controlli sul canvas**: in questa sezione vengono raccolte le principali informazioni riguardo al contesto come il fattore di *zoom* corrente, la differenza di traslazione rispetto all'origine e lo *slider* per il cambiamento del raggio dei nodi.

- **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 mappa che ha come chiavi le molecole e valori le relative concentrazioni), e le reazioni (Vedi **??**). Per le ultime tre categorie sono stati usati degli elementi che possono essere "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).

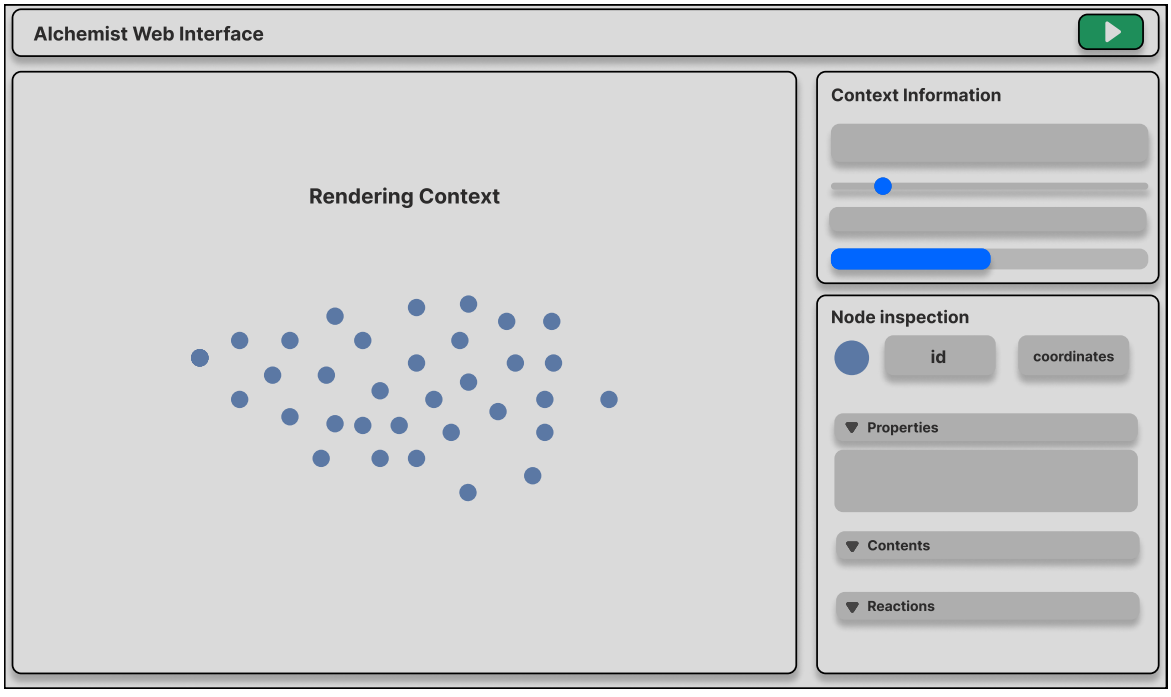

Figura 3.2: Mockup dell'interfaccia grafica

Capitolo 4

Implementazione e Verifica

4.1 Componenti grafici

4.2 Gestione dello stato

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.