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 2022-2023

Sommario

Max 2000 characters, strict.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

Nell'era digitale, il software pervade ogni aspetto della vita quotidiana. L'elemento che ha un impatto maggiore sull'esperienza di utilizzo è quasi sempre l'interfaccia utente. Riveste un ruolo chiave nella *user experience*, influenzando in modo preponderante il giudizio complessivo sul prodotto finale. In un certo senso, è un aspetto che rende univoco qualsiasi software, determinando la sua efficacia e fruibilità. Rappresenta il punto di incontro tra l'utente e le funzionalità del sistema stesso. Una User Interface (UI) fluida e accessibile permette all'utente di esplorare tutte le funzionalità del software e di sfruttarne al meglio il potenziale. Essa non è solo un insieme di elementi estetici, ma un vero e proprio linguaggio visivo che comunica con l'utente e lo guida all'utilizzo.
Per la conservazione di una esperienza utente soddisfacente, è fondamentale stare al passo delle tecnologie moderne. In un mondo in costante evoluzione, gli utenti si aspettano interfacce grafiche piacevoli e intuitive. Da questa situazione emerge l'esigenza della creazione di una interfaccia web per il simulatore *Alchemist*, in grado di affidarsi alla nuova infrastruttura Application Program Interface (API), sviluppata per l'accesso e controllo dei dati delle simulazioni. L'interfaccia grafica esaminata all'interno di questo elaborato, è destinata a procurare un punto di partenza per descrivere il meta-modello di un sistema complesso come quello di *Alchemist*, mediante l'utilizzo di uno stile moderno e facile da usare.

1.1 Contesto

Il progresso tecnologico degli ultimi vent'anni converge sempre più alla totale integrazione di dispositivi connessi a Internet nella vita quotidiana. La società odierna dipende sempre di più dalla tecnologia per soddisfare le proprie esigenze, spesso ricorrendo all'utilizzo di dispositivi e servizi tra loro eterogenei in hardware, ma soprattutto in software. Smartphone, *wearables*, dispositivi embedded, TV, elettrodomestici, e una vasta gamma di altre tecnologie contribuiscono a creare un ecosistema digitale interconnesso. In questo contesto nasce il concetto di *Pervasive Computing*), un modello informatico proposto negli anni '80, quando il ricercatore Mark Weiser introdusse il concetto di "computing ubiquo" (da qui anche la denominazione *Ubiquitous Computing* [Wei02]. L'obiettivo principale di questo paradigma è quindi quello di rendere la tecnologia meno intrusiva e più adattabile al contesto dell'utente. Per raggiungere questo obiettivo, è necessario un coordinamento efficace di dispositivi, che deve garantire un'integrazione armoniosa e una comunicazione fluida tra di essi per fornire un'esperienza priva di discontinuità. Un sistema di questo tipo deve presentare caratteristiche di adattabilità e auto-organizzazione. L'ingegneria di questi sistemi si concentra sulla coordinazione di agenti mobili e interconnessi che collaborano attraverso lo scambio di informazioni.

1.1.1 Simulazione

Il ricorso a strumenti simulativi per studiare sistemi composti da entità in grado di auto-coordinarsi e scambiare informazioni con l'ambiente circostante risulta essere essenziale per validare modelli che rappresentano scenari diversi. Una simulazione consente di eseguire una serie di test in un ambiente controllato per garantire che i modelli producano risultati attendibili. Inoltre, è possibile valutare l'impatto delle prestazioni del sistema in situazioni con un'alta densità di agenti, nonché variazioni nella topologia di interconnessione o cambiamenti nei pattern di interazione. Una simulazione precede l'implementazione reale del sistema in esame, esponendo punti di forza e punti deboli in anticipo, riducendo così il rischio di errori potenzialmente costosi in risorse, tempo e denaro.

1.1.2 Alchemist

Alchemist [PMV13] è un framework di simulazione open-source progettato e sviluppato dall'Università di Bologna per supportare lo studio e l'analisi di sistemi pervasivi. Il suo scopo principale è quello di fornire agli sviluppatori uno strumento avanzato per descrivere e simulare interazioni complesse tra individui autonomi in ambienti dinamici e distribuiti. Gli agenti autonomi fanno parte di un ecosistema che evolve nel tempo in modo autogestito attraverso leggi di base (chiamate *eco-laws*) che definiscono meccanismi di coordinamento, comunicazione e interazione. L'approccio intrapreso da *Alchemist* per le interazioni tra i nodi (agenti) si ispira a reazioni chimiche, la cui natura aleatoria deriva dall'implementazione di un Stochastic Simulation Algorithm (SSA) su misura [PMV11]. Ne consegue un comportamento indipendente da parte degli agenti all'interno del sistema, che rispecchiano le peculiarità mutevoli e intricate dei contesti pervasivi. L'architettura del simulatore consente di modellare gli *autonomous-agents* come entità che spaziano in diversi campi, tra cui il *pervasive*, l'*aggregate* e il *nature-inspired computing*.

Il meta-modello

Con meta-modello ci si riferisce alla struttura concettuale che definisce le entità e le loro relazioni all'interno dell'ambiente di simulazione. Serve per delineare gli elementi fondamentali e le loro interazioni, fornendo una base per la costruzione ed esecuzione di simulazioni. In generale, è utile per definire e comprendere i componenti e le dinamiche del mondo simulato in Alchemist. Gli elementi principali

- **Molecule**: Nome di un determinato dato.
- **Concentration**: Valore associato a una particolare molecola.
- **Node**: Contenitore di molecole e reazioni, situato all'interno dell'ambiente di simulazione.
- **Environment**: Rappresenta lo spazio di simulazione. È un contenitore di nodi e può fornire informazioni su:

1. La posizione dei nodi nell'ambiente.
2. La distanza tra due nodi.
3. Il supporto per lo spostamento dei nodi.

- **Linking rule**: Funzione dello stato corrente dell'ambiente che associa a ciascun nodo un vicinato.

- **Neighborhood**: Entità definita da un nodo centrale e un insieme di nodi vicini a esso.

- **Reaction**: Qualsiasi evento che può cambiare lo stato dell'ambiente. Ogni nodo ha un insieme di reaction, che può anche essere vuoto. Ogni reaction è definita da una lista di condizioni, una o più azioni e una distribuzione temporale. La frequenza con cui la reaction accade dipende da:

  1. Un parametro di frequenza statico.
  2. Il valore di ogni condition.
  3. Una "equazione di frequenza" che combina il parametro di frequenza statico con il valore delle condition, fornendo come risultato una valore di "frequenza istantanea".
  4. Una valore di distribuzione temporale.

- **Condition**: Una funzione che prende in input lo stato corrente dell'ambiente e restituisce un booleano e un numero. Se la condizione non è soddisfatta (ovvero il suo output corrente è falso), la reazione a cui è associata non può essere eseguita. In base alla reaction e alla distribuzione temporale, il numero in output ha la possibilità di influenzare o meno la velocità della reaction.

- **Action**: Modella un cambiamento nell'ambiente.

La fig. 1.1 illustra il modello in esame.

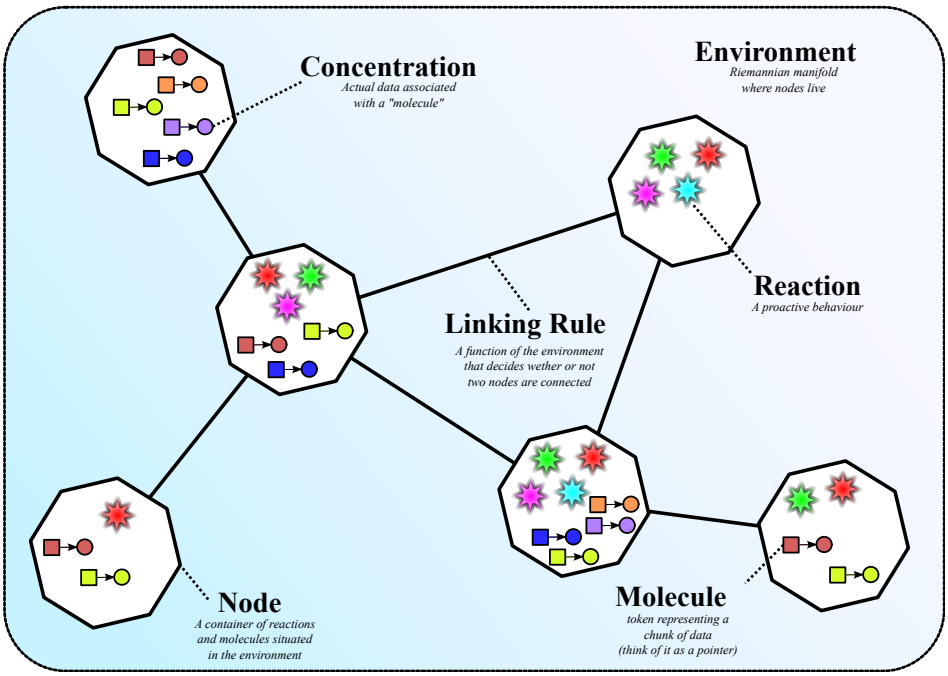

Figura 1.1: Rappresentazione del meta-modello di Alchemist

Incarnation

Inizialmente *Alchemist* è stato concepito solamente come un motore di simulazione stocastico orientato alle reazioni chimiche, dando mobilità ai nodi e mantenendo alte prestazioni. La duttilità di questo simulatore risiede nella concezione generale di **molecule** e **concentration**. In *Alchemist* quest'ultimi sono definiti, rispettivamente, come un generico codice identificativo e un dato di un certo tipo. Una *incarnation* definisce pertanto il tipo delle **concentration**, il set delle **condition** specifiche, le **actions**, **environment** e **reaction** che possono operare su tali tipi. In altre parole una incarnation è un'istanza concreta del meta-modello di *Alchemist* ed è per questo motivo che è possibile modellare scenari completamente diversi tra loro. Al momento, le incarnation possibili sono le seguenti:

1. **Protelis incarnation**[1]: progettato per semplificare la creazione di una rete composta da dispositivi potenzialmente mobili e diversi tra loro. Aderisce al

[1] `https://protelis.github.io/`

paradigma di *aggregate computing*, che applica la filosofia *divide et impera* mediante l'utilizzo di una rete di sensori e computer distribuiti.

2. **SAPERE incarnation** [ZOA+15]: progettato per la simulazione di sistemi compositi di servizi pervasivi.

3. **Biochemistry incarnation**[2]: consente di modellare reazioni chimiche o fenomeni biochimici all'interno dell'**environment**.

4. **Scafi incarnation**[3]: progettato anch'esso per la simulazione di sistemi compositi di servizi pervasivi.

1.2 Motivazione

1.3 Obiettivi

[2] `http://archive.today/nhGQy`
[3] `https://protelis.github.io/`

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 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 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.

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 e verifica delle operazioni

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 server GraphQL, accessibile tramite l'*endpoint* `/graphiql`, è il *playground* GraphiQL[6]. 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

[6] `https://github.com/graphql/graphiql`

allo sviluppatore di validare ogni singola operazione che verrà utilizzata all'interno dell'applicativo che si intende sviluppare. Solo successivamente potranno essere eseguite le operazioni sul server GraphQL.

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

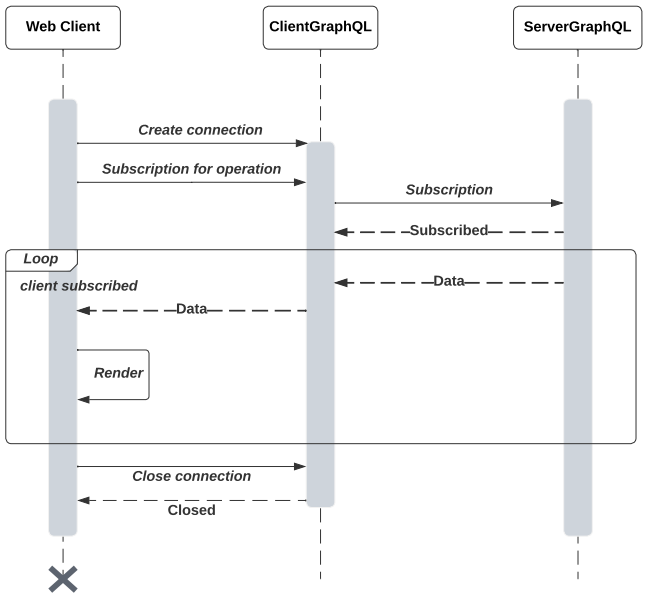

Figura 2.1: Diagramma di sequenza del rendering dei nodi a seguito di una *subscription*

simulazione che è in esecuzione.

2.2.4 Progetto multiplatform

Kotlin Multiplatform[7] 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-

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

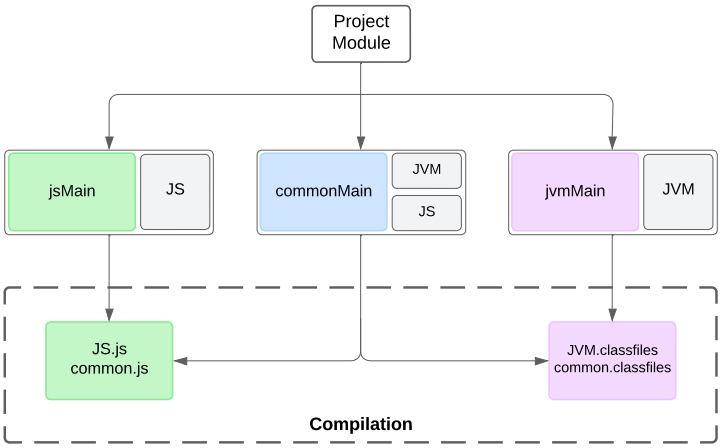

Figura 2.2: Struttura di un progetto multipiattaforma compilato per KotlinJS e KotlinJVM

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 raffigurano 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).

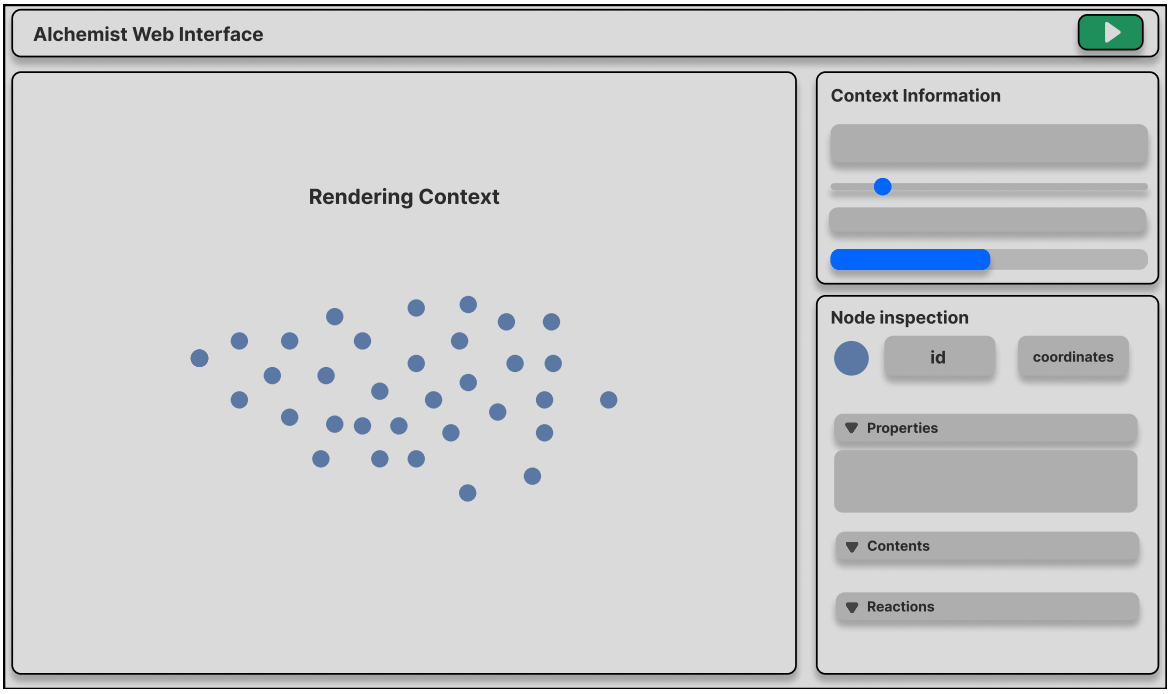

Figura 3.1: Mockup dell'interfaccia grafica

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*.
- **GeneratedSources**: 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 la composizione delle operazioni sul server.

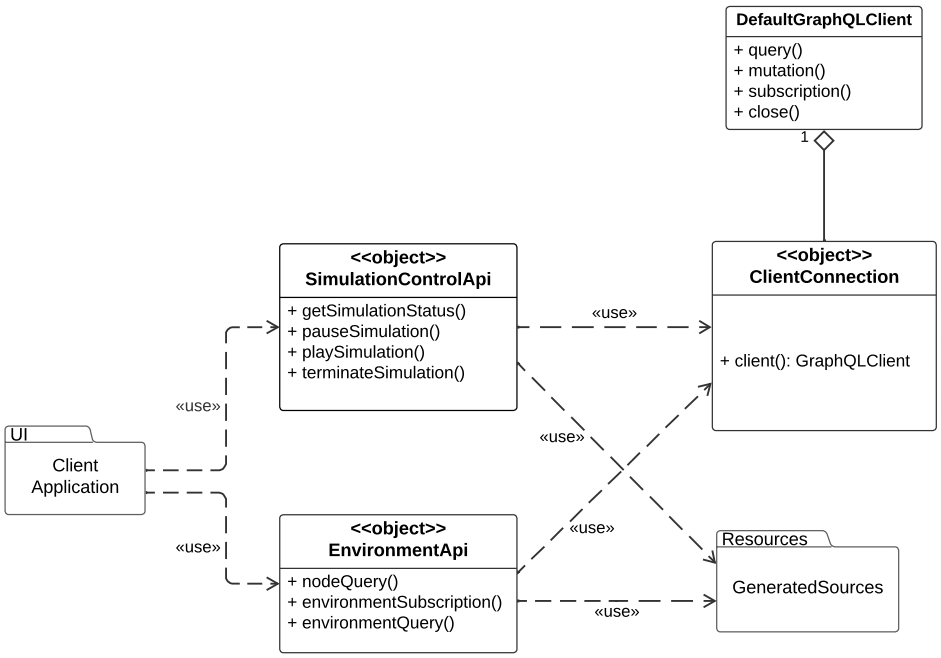

Figura 3.2: Utilizzo della connessione al client GraphQL

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 in fig. 3.3 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 3.1. 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 avvia il server GraphQL e 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.

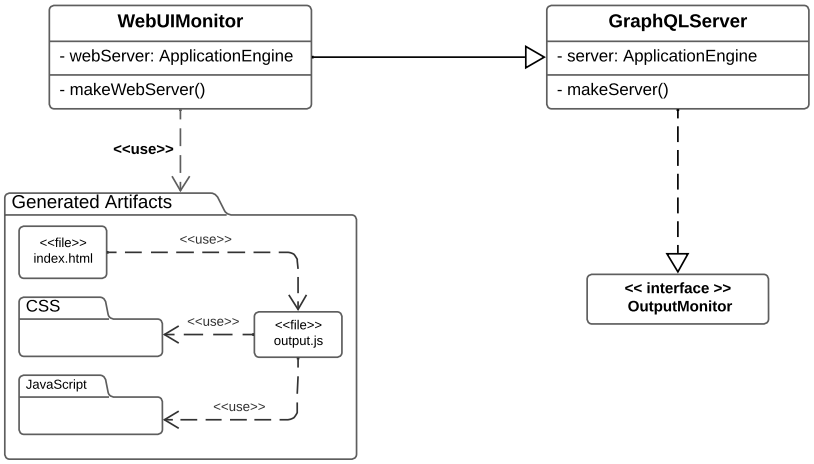

Figura 3.3: Architettura generale del client web

3.4 Struttura della pagina web

Si vuole definire ora la struttura della pagina che si presenterà nel momento in cui gli utenti si collegheranno al client web. La struttura finale si presenta come una gerarchia di componenti grafiche. Ogni componente è raggruppata all'interno di contenitori logici per una gestione efficiente e una navigazione chiara dell'interfaccia. Questo è ottenuto attraverso l'utilizzo di oggetti che dispongono i componenti figli secondo un layout prestabilito. Sono stati creati contenitori modulari e riutilizzabili per facilitare lo sviluppo e la manutenzione dell'interfaccia, oltre che a fornire un punto di scalabilità. Di conseguenza, ciò consente di comporre e combinare diverse componenti per soddisfare le esigenze specifiche delle diverse sezioni dell'applicazione. La figura 3.4 descrive la suddetta struttura.

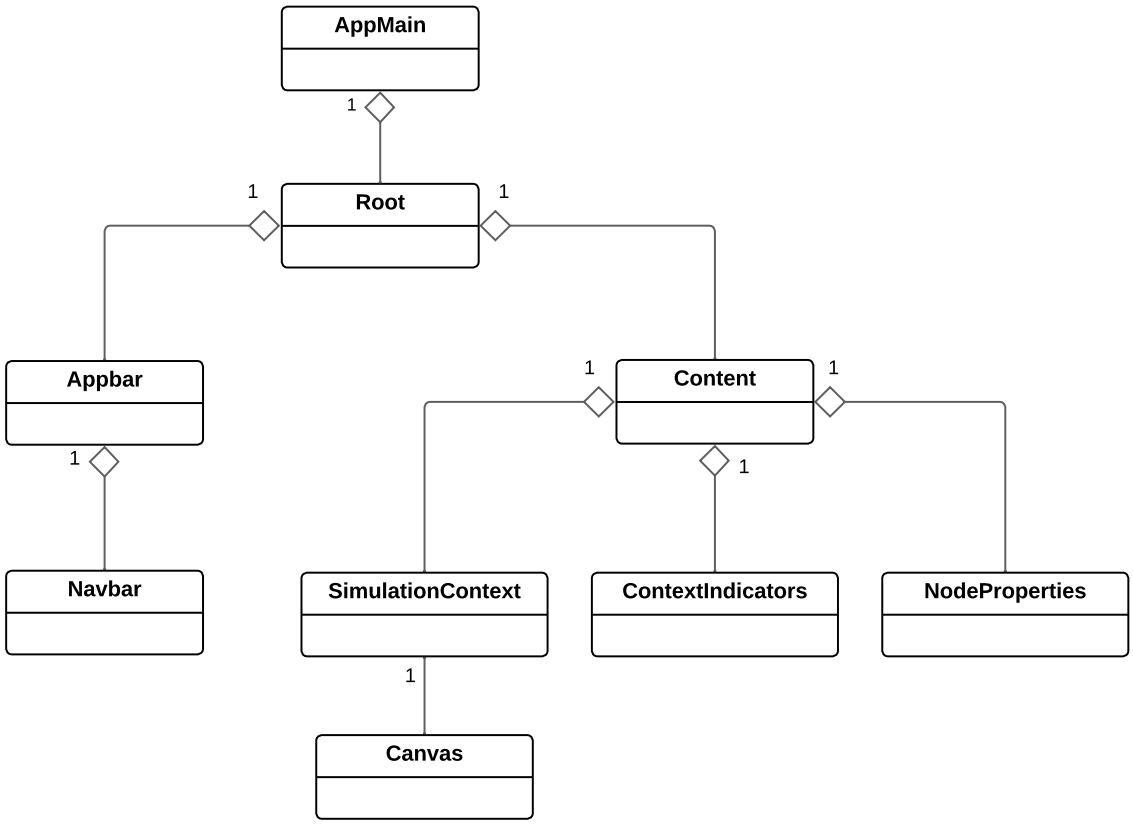

Figura 3.4: Diagramm UML delle classi per le componenti della pagina web

L'applicazione parte dalla classe `AppMain` che effettua una query sul DOM per la ricerca dell'elemento padre con identificativo "root". Una volta ottenuto, vengono aggiunti, come componenti figli, le classi `Appbar` e `Content`, che, rispettivamente, danno forma alla barra di navigazione e alla parte corposa della pagina (il contenuto principale). In particolare per quest'ultimo, si può osservare l'esistenza di una corrispondenza uno a uno tra le sezioni descritte in 3.1:

- `SimulationContext`: classe padre contenente il canvas di disegno.
- `ContextIndicators`: contiene tutte le singole entità grafiche che provvedono a dare informazioni riguardo al canvas.
- `NodeProperties`: fornisce in un layout riassuntivo, le principali informazioni riguardanti un nodo selezionato dal canvas.

Come per gli elementi appena citati, è importante sottolineare la correlazione tra i macro-elementi che hanno un ruolo all'interno dell'interfaccia utente e le classi principali che compongono la struttura finale della pagina HTML. Il mantenimento di questa relazione biunivoca è di aiuto allo sviluppatore e mantiene ordinato e scalabile il codice sorgente.

Capitolo 4

Implementazione e Verifica

4.1 Componenti grafici

Lo sviluppo web moderno trae benefici significativi da framework che semplificano la creazione di applicazioni web. Tutte le componenti grafiche all'interno di questo progetto sono state sviluppate attraverso l'utilizzo del framework open-source *KVision*[1], che permette agli sviluppatori di costruire interfacce web moderne senza utilizzare HTML, CSS o JavaScript. Le interfacce vengono assemblate attraverso la composizione di oggetti pronti all'uso, seguendo un paradigma paragonabile a quello dichiarativo. Il risultato sono gerarchie di componenti che possono essere usate come blocchi costituenti dell'intera interfaccia. In aggiunta, *KVision* presenta supporto integrato per gli store Redux (esplorati in sezione 4.3), per le icone Font Awesome, per Bootstrap e molto altro. Questa libreria sfrutta puramente le capacità del linguaggio Kotlin (compilato per target JS), specialmente con l'utilizzo di *type safe builders*, implementati attraverso *extension functions*. Quello offerto da *KVision* è di fatto un DSL, di cui si è fatto ampiamente uso. Nel listato 4.1 è illustrata, a scopo esemplificativo, una versione riadattata della definizione della struttura del contenuto principale della pagina web.

Per la maggioranza degli elementi dichiarabili in HTML, esiste una classe corrispondente offerta da *KVision*. Da qui deriva la natura dichiarativa nello sviluppo delle componenti grafiche. Questo avviene grazie all'utilizzo di *extension func-*

[1] `https://kvision.gitbook.io/kvision-guide/`

Listing 4.1: DSL KVision per la struttura del contenuto principale della pagina

```
hPanel {
    add(
        SimulationContext("simulation-context").apply {
            width = 1400.px
            height = 900.px
        }
    )
    vPanel {
        width = 520.px
        height = 95.perc
        add(
            SimulationIndicators("simulation-indicators").apply {
                width = 100.perc
                height = 100.perc
            }
        )
        add(
            NodeProperties("node-properties").apply {
                width = 100.perc
                height = 100.perc
            }
        )
    }
}
```

tions, funzionalità caratteristica del linguaggio Kotlin che permette di aggiungere nuovi comportamenti a classi esistenti senza dover ereditare da esse o modificarle direttamente. In questo caso, ciò significa che per un elemento del DOM come `div`, il framework fornisce la classe `Div` alla qual è stata aggiunta l'*extension function* `div()` che funge da *builder*. L'invocazione di quest'ultima evita la memorizzazione dell'istanza della componente in una variabile separata, e la successiva aggiunta manuale alla lista di figli della classe padre che li contiene. Nel listato 4.1 sono stati creati componenti custom che estendono la classe `SimplePanel` (classe base per un semplice pannello rappresentabile nella pagina). Si parla di `SimulationContext`, `SimulationIndicators` e `NodeProperties`, come indicato nella sezione 3.4. In

questo caso sono stati aggiunti manualmente come componenti figli delle rispettive classi padre, tramite il metodo `add()`. Sarebbe stato possibile aggiungere a queste classi una *extension function* per creare un builder. Ogni volta in cui sarebbe stato necessario dichiarare più volte questo tipo di oggetto in diverse parti dell'applicativo, sarebbe bastata una invocazione come nel listato 4.2. In questo caso non era strettamente necessario, perché queste componenti vengono istanziate una volta sola.

Listing 4.2: Uso di *type-safe* builders

```
vPanel {
    width = 520.px
    height = 95.perc

    simulationIndicators {
        className = "simulation-indicators"
        width = 100.perc
        height = 100.perc
    }

    nodeProperties {
        className = "node-properties"
        width = 100.perc
        height = 100.perc
    }
}
```

La manipolazione degli elementi all'interno di un contenitore in una pagina web risulta essere molto onerosa se si fa uso di HTML e CSS puro. La convenienza di utilizzo di questo framework risalta anche nell'esistenza di elementi pronti all'uso come `hPanel` e `vPanel`, due costrutti che dispongono, rispettivamente, in orizzontale e in verticale gli elementi aggiunti al loro interno. La disposizione degli oggetti all'interno di questi layout è facilitata perché quest'ultimi rispettano le specifiche della recommendation *CSS Flexible Box Layout* [2].

[2] `https://www.w3.org/TR/css-flexbox/`

Inutile dire che la libreria utilizzata offre una vasta gamma di costrutti e metodi che semplificano notevolmente lo sviluppo di interfacce utente, contribuendo così a migliorare l'efficienza complessiva del processo di sviluppo.

4.2 Integrazioni di operazioni GraphQL

Nella sezione 3.2 sono stati illustrati i motivi dietro all'utilizzo di un unico punto di accesso per l'esecuzione delle *query*, *mutation* e *subscription* all'interno dell'applicativo. A questo scopo i *Singleton* `EnvironmentApi` e `SimulationControlApi` raggruppano al loro interno funzioni che avviano operazioni sul modello fra loro correlate. Come suggerisce il nome, il primo presenta tutte le operazioni effettuabili sull'ambiente della simulazione, mentre il secondo contiene quelle legate al controllo della simulazione. Per illustrare meglio come avvengono le operazioni sul server GraphQL, viene presentata l'implementazione della query che reperisce lo stato corrente della simulazione.

1. È stata definita l'operazione voluta, come riportato nel listato 4.3.

Listing 4.3: Query GraphQL per il recupero dello stato corrente della simulazione

```
query SimulationStatus {
    simulationStatus
}
```

2. Al momento della compilazione del progetto, viene generata dalla libreria *Apollo* [3] la classe Kotlin `SimulationStatusQuery` dentro al modulo `graphql` (pacchetto denominato come *GeneratedSources*, vedi 3.2).

3. La funzione `getSimulationStatus()` del listato 4.4, esegue la query tramite l'avvio di un thread asincrono, in modo che questa chiamata non influisca sul thread di rendering principale della UI.

4. La query può essere chiamata in qualsiasi momento tramite la funzione `callGetStatus()` (listato 4.5).

[3] `https://github.com/apollographql/apollo-kotlin`

Listing 4.4: Esecuzione asincrona della query SimulationStatus

```
suspend fun getSimulationStatus(): Deferred<SimulationStatusQuery.Data?> =
  coroutineScope {
    async {
      ClientConnection.client
      .query(SimulationStatusQuery())
      .execute()
      .data
    }
  }
```

Listing 4.5: Chiamata per il prelievo dello status della simulazione

```
fun callGetStatus() {
  MainScope().launch {
    val result = SimulationControlApi.getSimulationStatus().await()

    simulationStore.dispatch(SimulationAction.SetSimulation(result))
  }
}
```

La natura e il tipo di ritorno di queste funzioni varia in base al tipo di operazioni effettuate, ovvero:

- **query** e **mutation**: per questo tipo di operazioni ci si aspetta un unico valore di ritorno. L'esecuzione di queste procedure avviene in modo asincrono attraverso l'utilizzo del costrutto `async()`, servito dalla libreria Kotlin *kotlinx.coroutines*[4]. Il risultato viene restituito dal server una volta che la richiesta ha avuto successo. L'attesa del completamento dell'esecuzione della funzione che esegue la query avviene invocando `await()`. Non a caso, questo tipo di funzioni possono essere sospese. Da qui il motivo per il modificatore d'accesso `suspend`.

- **subscription**: diversamente da quanto descritto per *query* e *mutation*, l'esecuzione di una *subscription* fornisce come risultato un oggetto di tipo `Flow`, di cui si esamina il comportamento in sezione 4.3.1.

[4] `https://kotlinlang.org/docs/coroutines-guide.html`

4.3 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*[5], 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 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.

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

- **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 offre una visione più completa del funzionamento della gestione dello stato. Dallo schema è possibile notare

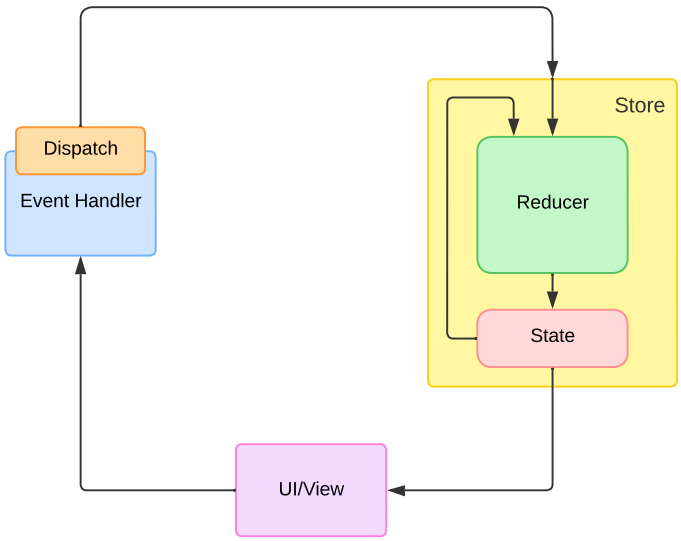

Figura 4.1: Funzionamento di uno store Redux

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. Accade un evento 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.

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.

Stato del nodo selezionato All'interno dell'interfaccia grafica è possibile ottenere le informazioni di un nodo effettuando una apposita query al server GraphQL. Queste informazioni possono essere utili all'interno di tutta l'interfaccia grafica e sono appositamente memorizzate in uno store. *KVision* offre un'implementazione ad-hoc per gli store Redux. La creazione avviene come riportato in listato 4.6. `createTypedReduxStore` è una funzione della libreria di *KVision* che permette la

Listing 4.6: Creazione dello store Redux `nodeStore` nel framework KVision

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

creazione di uno store alla quale, da definizione, vengono passati il relativo reducer e lo stato iniziale.

Lo stato dell'applicazione è memorizzato all'interno dello store Redux. Non può essere cambiato direttamente dall'esterno, questo perché vige una regola di immutabilità. Lo stato corrente è sempre un oggetto immutabile, del quale non si possono cambiare i contenuti. Per aggiornare i valori in modo immutabile, devono essere eseguite copie degli oggetti esistenti e solo dopo modificare le copie. Dal listato 4.7 si può notare come la classe è di tipo `data`. Dopotutto lo stato deve "solo" contenere informazioni.

Listing 4.7: Classe `NodeState` che modella lo state

```
data class NodeState(val node: NodeQuery.Data?)
```

Per questo store pertanto, le azioni sono definite come classi che possono contenere informazioni aggiuntive. La classe dichiarata è di tipo `sealed`, il che permette di usare l'espressione `when` in modo esaustivo in fase di definizione delle operazioni per tipo. Al suo interno, vi sono le sottoclassi che contengono le azioni vere e proprie. Il listato 4.8 mostra come `NodeAction` è stata implementata.

Listing 4.8: Classe action per lo store `NodeStore`

```
sealed class NodeStateAction: RAction {

    data class SetNode(val node: NodeQuery.Data?): NodeStateAction()

}
```

Infine, il listato 4.9 mostra come è stata definita la funzione reducer, che è l'unico modo per modificare lo stato. Di solito, viene chiamata dopo che un'azione è stata inviata allo store.

Listing 4.9: Funzione reducer per lo store `NodeStore`

```
fun nodeReducer(state: NodeState, action: NodeStateAction):
    NodeState {
    when (action) {
        is NodeStateAction.SetNode -> {
            return state.copy(node = action.node)
        }
    }
}
```

Facendo click sul canvas sopra a un nodo, viene quindi chiamata la funzione `dispatch()` per aggiornare lo stato corrente (listato 4.10).

Listing 4.10: Chiamata alla query `nodeQuery` con l'`id` recuperato dall'evento click

```
fun nodeById(nodeId: Int = 0) {
    MainScope().launch {
        val result = EnvironmentApi.nodeQuery(nodeId).await()

        nodeStore.dispatch(NodeStateAction.SetNode(result))
    }
}
```

Aggiornamento della componente grafica Il framework *KVision* viene di nuovo in aiuto per l'aggiornamento di una componente grafica in base al cambiamento di stato di uno store. Viene utile a questo scopo la concezione di *state binding*. Si riferisce al processo di collegamento che avviene tra uno stato dell'applicazione e lo stato di una componente della UI. Questo permette al componente di reagire dinamicamente ai cambiamenti nello stato, aggiornando automaticamente la visualizzazione quando lo stato cambia. In *KVision*, questo meccanismo è accessibile in diversi modi tramite funzioni specifiche fornite dalla libreria, che consentono di associare direttamente il valore dello stato a un componente UI senza dover gestire manualmente gli aggiornamenti della visualizzazione. Il listato 4.11 mostra lo state binding tra una parte (semplificata) della componente grafica `NodeProperties` (sezione 3.1) e lo store `NodeStore` descritto in listato 4.6. I due contenitori HTML `div` riportano le proprietà del nodo selezionato, rispettivamente il codice identificativo `id` e la coppia di coordinate (x, y) nel canvas. La funzione `bind()` esegue il processo descritto fino ad'ora. In alternativa, sarebbe stato comunque possibile aggiornare la componente grafica invocando la funzione `subscribe()` offerta da `NodeStore`.

4.3.1 Rappresentazione dei nodi

Nella sezione 2.2.3 abbiamo analizzato come la scelta di un canvas sia stato l'approccio più adatto per motivi di performance. Per questo motivo è stato impiegato l'oggetto *canvas* di HTML, o meglio, si è usata la sua implementazione da parte di *KVision*. All'interno del *canvas* è possibile accedere al suo contesto tramite la proprietà `context2D`, di tipo `CanvasRenderingContext2D`, che fornisce la ca-

Listing 4.11: Binding tra i componenti `div` e `NodeStore`

```
div {
    ...
}.bind(NodeStore.nodeStore) {
    +"id: ${it.node?.environment?.nodeById?.id}"
}

div {
    ...
}.bind(NodeStore.nodeStore) {
    +"X: ${it.node?.nodePosition?.coordinates?.get(0)}"
    +"Y: ${it.node?.nodePosition?.coordinates?.get(1)}"
}
```

pacità di disegnare qualsiasi cosa in un piano bidimensionale, combinando linee, archi, curve e altro. `CanvasRenderingContext2D` sfrutta le capacità della libreria *WebGL* [6], un API JavaScript per il rendering di grafiche 2D (e 3D) all'interno di un qualsiasi browser. Questo standard aderisce strettamente a quello OpenGL ES 2.0, dando la possibilità a queste API di trattare completo vantaggio dall'utilizzo dell'accelerazione grafica del sistema. Ora è possibile fornire lo schema in fig. 4.2, una versione più dettagliata di quello in sezione 2.2.3.

Seguendo lo schema, viene effettuata una operazione di tipo **subscription** al server GraphQL per il recupero delle posizioni dei nodi. Il listato 4.12 mostra

Listing 4.12: Chiamata alla subscription sulla posizione dei nodi

```
fun environmentSubscription(): Flow<...> {
    return ClientConnection.client
            .subscription(EnvironmentSubscription())
            .toFlow()
}
```

la chiamata alla funzione che esegue l'operazione di *subscription*. La funzione restituisce il risultato della query come un oggetto di tipo `Flow`. Quest'ultima è una classe di Kotlin che rappresenta una sequenza di valori che vengono prodotti in

[6] `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API`

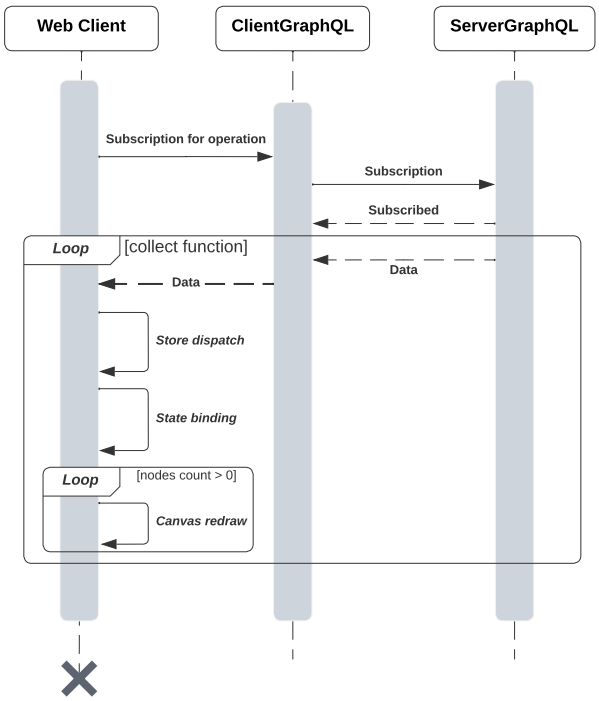

Figura 4.2: Diagramma di sequenza per il disegno dei nodi nel canvas

modo asincrono. Più nello specifico, viene restituito un *cold* `Flow`. Questo significa che i dati non vengono forniti fino a quando il flusso non viene consumato. I valori possono essere consumati in un secondo momento, come avviene nel listato 4.13. Notare come in questo caso il *triggering event* per l'aggiornamento dello store legato alla posizione dei nodi è proprio la funzione `collect()`.

Listing 4.13: Consumazione della subscription e aggiornamento dello store dei nodi

```
EnvironmentApi.environmentSubscription().collect {
    EnvironmentStore.store.dispatch(
        EnvironmentStateAction.AddAllNodes(
            it.data?.environment?.nodeToPos!!.entries
        )
    )
}
```

A questo punto, è necessario iscriversi ai cambiamenti di stato e richiamare la

funzione *redraw* del canvas, come nel listato 4.14. Il meccanismo di state binding in questo caso è reso possibile attraverso l'utlizzo della funzione `subscribe()` dello store `EnvironmentStore.store`.

Listing 4.14: Iscrizione agli aggiornamenti dello store e richiamo della funzione `redrawNodes()`

```
1  //Environment state subscription
2  EnvironmentStore.store.subscribe { state ->
3      ...
4          canvasCtxt.redrawNodes(state.toListOfPairs())
5      ...
6  }
7
8  //Canvas redraw function
9  fun CanvasRenderingContext2D.redrawNodes(
10     nodes: List<Pair<Double, Double>>
11 ) {
12     ...
13     nodes.forEach {
14         drawNode(Pair(it.first, it.second))
15     }
16     ...
17 }
```

4.4 Funzionamento client web

Il processo di *bundling* menzionato in sezione 3.3 avviene grazie all'utilizzo di *Webpack*[7]. Webpack è uno strumento che si occupa di organizzare e quindi combinare le risorse (JavaScript, CSS, icone, font) in un numero minore di file (in questo caso solo uno, di estensione `.js`). In questo contesto non sono stati creati file CSS per la descrizione dell'aspetto delle varie componenti, dato che *Kvision* fornisce già un certo livello di stile CSS. Inoltre, si è fatto uso di elementi che beneficiano nativamente del framework *Boostrap*. A prescindere da ciò, il file in output dal processo di *bundling* può essere incluso nella pagina `index.html`, ottenendo

[7] `https://webpack.js.org/concepts/`

il medesimo risultato che si sarebbe ottenuto includendo manualmente ogni singola risorsa. L'esecuzione della simulazione può essere configurata appositamente con il monitor `WebUIMonitor`. Questa classe permette, in thread separati, l'avvio contemporaneo del server GraphQL e il server che ospita la pagina web che rappresenta l'interfaccia grafica. Facendo anche riferimento alla fig. 3.3, per servire la pagina principale in un browser web si è fatto utilizzo di un server *Ktor* [8]. È stata configurata la *route* "`/`" per rispondere alle richieste di tipo GET con la pagina principale `index.html`, alla quale viene impostata come risorsa statica l'artefatto `.js` ottenuto dal processo di *bundling* precedentemente descritto.

4.5 Verifica

La verifica del software attraverso l'utilizzo di test automatizzati per interfacce utente, o per alcune delle loro componenti, risulta essere spesso un procedimento problematico in quanto queste tendono a essere soggette a frequenti aggiornamenti durante il loro sviluppo. In aggiunta, le UI sono spesso soggette a interazioni in tempo reale con l'utente, come eventi del mouse, input da tastiera e aggiornamenti dinamici. Questi aspetti rendono difficile simulare completamente il comportamento della UI in un ambiente di test isolato, poiché è difficile riprodurre accuratamente le interazioni dell'utente durante l'esecuzione dei test. L'unica verifica che è stata effettuata avviene in tempo reale: si è interagito direttamente e in modo esaustivo con l'interfaccia durante il processo di sviluppo. Questa pratica informale ha consentito di individuare e risolvere rapidamente eventuali problemi di usabilità e comportamentali, senza la necessità di test automatizzati specifici. Per questi motivi, l'automatizzazione dei test sulla UI può essere complessa e richiedere risorse significative, con risultati talvolta limitati o poco pratici rispetto al testing manuale in tempo reale.

D'altro canto, per garantire la corretta rappresentazione dei nodi e l'integrità del contenuto di ciascuno di essi, sono state eseguite diverse simulazioni, nello specifico con le *incarnation protelis* e *sapere*.

[8] `https://ktor.io/docs/create-server.html`

Si è fatto ampio ricorso al *playground* GraphiQL, uno strumento che consente di consultare tutta la struttura del modello di *Alchemist* e le operazioni che si possono effettuare su di esso. Questo riconduce sempre al concetto di *schema* delle operazioni descritto in 2.2.2. Dopo aver individuato il tipo di dati desiderati, sono state definite all'interno del *playground* le operazioni client per verificarne l'effettivo risultato che si ottiene una volta eseguite. A seguito, sono state incluse all'interno delle risorse client del modulo `graphql`. Operando secondo questi passaggi, ogni operazione aggiunta all'insieme delle operazioni client è risultata convalidata sullo *schema* ancora prima della fase di compilazione, garantendo l'assenza totale di errori di sintassi durante l'esecuzione di query o ricevimento dei dati.

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.

[PMV11] Danilo Pianini, Sara Montagna, and Mirko Viroli. A chemical inspired simulation framework for pervasive services ecosystems. pages 667–674, 01 2011.

[PMV13] D Pianini, S Montagna, and M Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, August 2013.

[Wei02] M. Weiser. The computer for the 21st century. *IEEE Pervasive Computing*, 1(1):19–25, January 2002.

[ZOA+15] Franco Zambonelli, Andrea Omicini, Bernhard Anzengruber, Gabriella Castelli, Francesco L. De Angelis, Giovanna Di Marzo Serugendo, Simon Dobson, Jose Luis Fernandez-Marquez, Alois Ferscha, Marco Mamei, Stefano Mariani, Ambra Molesini, Sara Montagna, Jussi Nieminen, Danilo Pianini, Matteo Risoldi, Alberto Rosi, Graeme Stevenson, Mirko Viroli, and Juan Ye. Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing*, 17:236–252, February 2015.