ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Accesso e Controllo Efficiente di Sistemi Software Complessi tramite GraphQL

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Dott. Danilo Pianini

Candidato
Stefano Furi

Correlatore
Dott. Angelo Filaseta

III Sessione di Laurea
Anno Accademico 2022-2023

Sommario

All'interno dei sistemi software moderni è di cruciale importanza la possibilità di monitorare e controllare un'intero sistema nella maniera più efficiente possibile. I principali meccanismi di controllo e interrogazione dei sistemi possono comprendere il design e l'implementazione di *Application Programming Interface* (API), le quali costituiscono un solido punto di appoggio per le comunicazioni tra le diverse componenti software di un sistema. All'interno di questo elaborato, viene presentata la realizzazione di un servizio di API all'interno di Alchemist, un simulatore stocastico realizzato all'interno della *Java Virtual Machine* (JVM).

Il servizio di API mira ad esporre verso l'esterno un insieme di informazioni inerenti una simulazione all'interno di Alchemist, e allo stesso tempo fornisce meccanismi di controllo della stessa, mantenendo un'implementazione che non sia dipendente dalla piattaforma utilizzata mediante l'uso di Kotlin Mutliplatform.

All'interno di questo documento, verranno illustrate le strategie di design e architetturali per la realizzazione di un tale sistema attraverso il paradigma GraphQL, il quale fornisce un *Query Language* in grado di stabilire con esattezza la struttura dei dati che un client deve ricevere dal sistema di API. Sono quindi approfondite le motivazioni dietro la scelta di tale paradigma, e vengono definite le operazioni di massima che il sistema deve fornire. Successivamente verrano coperte estensivamente le sfide implementative e di compatibilità causate dalla natura complessa di Alchemist, e la rigida e semplice struttura di uno schema GraphQL. Infine, sono illustrare le future estensioni possibili del sistema costruito, illustrandone scopi e benefici che possono apportare al sistema, richiedendo pochi o nessun cambiamento all'architettura proposta.

Indice

Elenco delle figure

Elenco dei Listati

Capitolo 1

Panoramica

1.1 Introduzione

Le API rappresentano una componente fondamentale nei sistemi software moderni. Da decenni, esse sono state il cardine dell'interoperabilità tra differenti software e piattaforme, permettendo la creazione di ecosistemi software integrati e coesi, in cui applicazioni di diversa natura possono scambiare dati e funzionalità in modo efficace ed efficiente. Nell'architettura di un sistema software, le API fungono da interfacce contrattuali, definendo i metodi e i protocolli con cui le diverse componenti software interagiscono. Questa caratteristica promuove la modularità, il riuso e la scalabilità dell'intero sistema. In particolare, con l'evoluzione delle architetture distribuite, il ruolo delle API diventa ancora più cruciale. Esse non solo consentono la comunicazione tra diversi servizi, ma spesso rappresentano il principale punto di contatto e di interazione tra un servizio e i suoi consumatori, sia essi altre applicazioni o utenti finali.

In questo scenario si pone il progetto discusso nel presente documento: l'analisi e l'implementazione di un architettura software che interagisce con il simulatore Alchemist, esponendo funzionalità e servizi di controllo essenziali per lo sviluppo di applicativi attraverso le API progettate. Tramite questo nuovo modulo, si mira a fornire un accesso standardizzato ed efficiente alle componenti di un sistema software complesso come Alchemist, facilitando così la manipolazione e il controllo delle sue funzionalità.

1.2 Contesto

La crescente integrazione di dispositivi di comunicazioni avanzati, come laptop, smartphone o sensori di *Internet of Things* (IoT) all'interno di ogni aspetto della vita quotidiana, ha provocato lo sviluppo di un modello di computazione definito come *Pervasive Computing*. In questo scenario, le infrastrutture software accrescono in complessità per via della decentralizzazione delle unità di elaborazione, in concomitanza con la natura eterogenea dei dispositivi stessi. La programmazione pervasiva è quindi concepita come risposta alle sfide presentate da tali scenari, pensata per gestire e ridurre la complessità di sistemi che richiedono adattabilità dinamica al contesto e coordinazione autonoma, provvedendo allo sviluppo di formalismi e strategie per l'implementazione di ambienti all'interno dei quali un insieme di agenti provvisti di un certo grado di autonomia, si coordinano per produrre, consumare e scambiare informazioni all'interno del sistema pervasivo.

1.2.1 Simulazione

La difficoltà principale all'interno di questo tipo di sistemi, risiede nella definizione di approcci per la gestione della coordinazione, auto-adattamento e autogestione degli agenti distribuiti nello spazio dell'ambiente. La simulazione emerge come uno strumento indispensabile per affrontare queste sfide in modo controllato ed efficiente. Essa consente di modellare e testare le dinamiche degli agenti in una varietà di scenari, valutando il loro comportamento in risposta a diverse condizioni ambientali.

Nella sua connotazione più generale, una *simulazione* è una riproduzione del modo di operare di un sistema o un processo del mondo reale, in un arco temporale discreto [BC86]. La rappresentazione del sistema, sia essa una rappresentazione matematica, descrittiva o logica, definisce il *modello* della simulazione. Il sistema può assumere un insieme finito di *stati*, ovvero un insieme di variabili contenenti tutte le informazioni necessarie alla descrizione del sistema in un determinato momento. Questo quindi definisce lo stato di una simulazione come una funzione legata al tempo, il quale viene rappresentato da una variabile definita *clock*. Il modello così costruito permette la rappresentazione delle *entità* che compongono

il sistema, le quali ricoprono il ruolo di oggetti e componenti (e.g. prodotti, dispositivi, parti, ...) alle quali possono essere associati uno o più attributi. Durante l'esecuzione della simulazione, la riproduzione del sistema attraverso le entità viene aggiornata ad ogni passo dell'esecuzione in base ad un insieme di *attività*. Queste sono caratterizzate da una durata finita nel tempo, la quale può essere prestabilita o casuale. In generale però, la terminazione di un'attività e la sua durata possono dipendere da un insieme articolato di *condizioni* sullo stato del sistema.

L'insieme di questi componenti può formare una tipologia di modello definito come *Discrete-Event Simulation model* (DES), all'interno del quale lo stato del sistema subisce un cambiamento solo all'interno di un certo insieme di punti temporali denominati *event times*. In questo tipo di simulazione quindi il tempo viene incrementato secondo regole prestabilite, e lo stato della simulazione non può cambiare tra un *event point* e il suo successivo. Al cambiamento della simulazione all'interno di un *event point*, viene generato ciò che viene definito *snapshot* della simulazione corrente, il quale verrà modificato solo al prossimo *event point*. In questo modo la simulazione procede di *snapshot* in *snapshot* fino a quando non termina.

Durante il processo di esecuzione della simulazione, è possibile effettuare osservazioni del sistema simulato, e collezionare misurazioni d'interesse. Mediante queste misurazioni è possibile per esempio effettuare una valutazione sulle prestazioni generali del modello, oppure estrapolare dati in grado di fornire una maggiore comprensione dei fenomeni e comportamenti che regolano il sistema simulato, fornendo la possibilità di analizzare, sperimentare ed ottimizzare un insieme di modelli che possano interagire con il sistema stesso, utilizzando la simulazione come un ambiente di sviluppo e testing.

1.2.2 Alchemist

Alchemist [PMVb] è un simulatore stocastico general-purpose, estensibile ed open-source sviluppato dall'Università di Bologna, progettato per affrontare le complessità della programmazione pervasiva. Questo simulatore si focalizza sulla necessità di definire meccanismi avanzati per modellare interazioni tra agenti autonomi in ambienti dinamici e distribuiti. La sua architettura, composta da un insieme di

nodi comunicanti, ciascuno potenzialmente eterogeneo, consente la simulazione di scenari di *pervasive*, *aggregate* e *nature-inspired computing*.

Alchemist offre un ambiente ampiamente estensibile, dove le interazioni tra agenti sono esposte attraverso reazioni ispirate alla chimica. Queste reazioni consentono la definizione di comportamenti autonomi degli agenti all'interno del sistema, riflettendo fedelmente la dinamicità e le complessità intrinseche dei contesti pervasivi. La flessibilità di Alchemist si estende alla definizione di sofisticate regole comportamentali all'interno dell'ambiente, fornendo agli sviluppatori uno strumento potente per esplorare e sperimentare con strategie di coordinazione, auto-adattamento e autogestione nel contesto di sistemi pervasivi e multi-agente.

Meta-modello Il meta-modello di Alchemist rappresenta il fondamento concettuale che guida la progettazione e l'implementazione delle simulazioni. Questo schema astratto definisce gli elementi chiave e le relazioni che costituiscono la struttura fondamentale del simulatore, offrendo una base teorica solida per la modellazione e la simulazione di scenari complessi.

Gli elementi principali che compongono il modello di Alchemist sono[1]:

- **Molecule**: denominazione assegnata ad un dato specifico.
- **Concentration**: il valore assegnato ad una particolare molecola.
- **Node**: contenitore al cui interno risiedono molecole e reazioni, contenuto a sua volta nell'ambiente.
- **Environment**: astrazione del concetto di spazio all'interno di Alchemist. Contiene tutti i nodi ed è in grado di comunicare:
  1. La posizione nello spazio dei nodi.
  2. La distanza tra due nodi.
  3. Muovere all'interno dello spazio i nodi.
- **Linking Rule**: funzione dello stato attuale dell'ambiente che associa ogni nodo ad un vicinato.

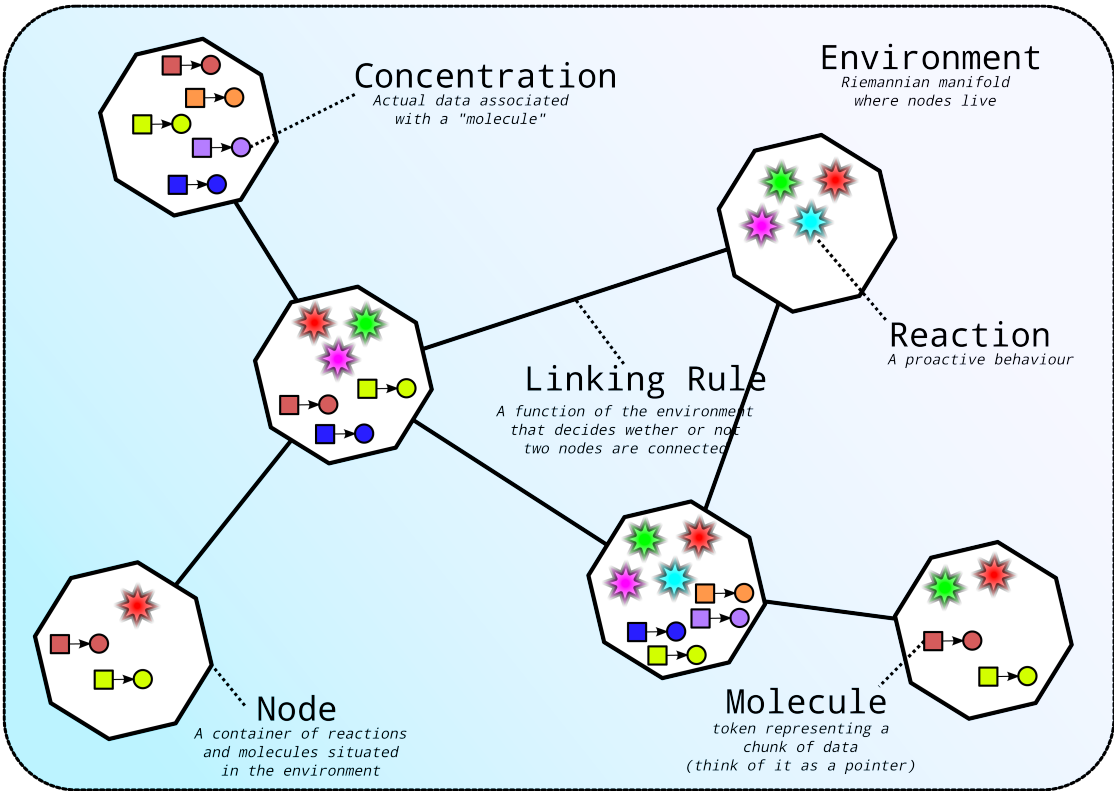
[1] https://alchemistsimulator.github.io/

- **Neighborhood**: entità composta da un nodo (centro) e un insieme di altri nodi (vicini).
- **Reaction**: rappresenta ogni evento che può cambiare lo stato dell'ambiente. È definita da un insieme, potenzialmente vuoto di condizioni, una o più azioni e una distribuzione temporale. La frequenza con cui può accadere dipende da:
  1. Un parametro statico di frequenza.
  2. Il valore di ogni condizione.
  3. Una equazione di frequenza, che combina la frequenza statica e il valore delle condizioni, restituendo una "frequenza istantanea".
- **Condition**: funzione che prende in input l'ambiente corrente e restituisce un valore booleano e un numero: se la condizione non è verificata, la reazione associatagli non verrà eseguita. Il numero restituito in output può influenzare la velocità della reazione, in base alla reazione e alla sua distribuzione temporale.
- **Action**: modella un cambiamento all'interno dell'ambiente.

Il meta-modello di Alchemist può essere riassunto nella Figura 1.1

Incarnation La chiave dell'estensibilità di Alchemist risiede nella rappresentazione astratta di molecole e concentrazioni. All'interno di Alchemist una *Incarnation* include una definizione per il tipo delle concentrazioni, un eventuale insieme di specifiche condizioni, azioni e (raramente) un ambiente e reazioni che operano su quei determinati tipi. In altre parole, un'incarnazione è una istanza concreta del meta-modello di Alchemist. Diverse *Incarnation* possono modellare universi completamente differenti. Allo stato attuale, Alchemist viene distribuito con le seguenti incarnazioni:

- **SAPERE Incarnation** [PMVa]
- **Protelis Incarnation** [PVB]



Source: `https://alchemistsimulator.github.io/explanation/metamodel/`

Figura 1.1: Meta-modello di Alchemist

- **Biochemistry Incarnation**[2]
- **SCAFI Incarnation** [CVAP]

Applicazioni di Alchemist Alchemist è largamente utilizzato all'interno della comunità scientifica che ricopre aree inerenti lo studio di sistemi distribuiti su larga scala e *pervasive computing*. Un esempio dell'applicazione di Alchemist in combinazione con *SCAFI Incarnation* è illustrato dall'articolo *"Dynamic Decentralization Domains for the Internet of Things"* [ACPV22], nel quale viene esposto come l'impiego di Alchemist e il *Domain-Specific Language* (DSL) Scala di *SCAFI* abbia reso possibile lo sviluppo di un astrazione di design basata sulla "decentralizzazione dinamica dei domini", ovvero regioni di spazio opportunisticamente

[2]https://alchemistsimulator.github.io/explanation/biochemistry/

formate per favorire il riconoscimento e l'azione contestuale, dimostrandone il funzionamento attraverso un caso di studio riguardante il monitoraggio ambientale: `FloodWatch`. Esso prevede l'impiego di pluviometri disposti in una città, comunicanti tra loro. Lo scopo di questo sistema è il monitorare la progressione di una tempesta sulla città. L'obiettivo è sfruttare il raggruppamento di aree simili per ottenere un tracciamento più accurato del fenomeno, comprendere la sua struttura spaziale e utilizzare queste informazioni per adottare una possibile contromisura da adottare nelle zone definite a più alto rischio in caso di alluvione.

La capacità di espansione di Alchemist può condurre alla creazione di nuovi moduli capaci di modellare scenari diversi da quelli originariamente concepiti, come evidenziato nello studio "*A Collective Adaptive Approach to Decentralised k-Coverage in Multi-robot Systems*" [PPCE22]. L'articolo tratta della problematica del *Online Multi-Object k-Coverage problem* (OMOkC) dove un insieme di robot mobili devono rilevare un obiettivo in movimento attraverso k punti di vista, capaci di coordinarsi in maniera possibilmente decentralizzata e scalabile. L'obiettivo non consiste nel definire nuovi algoritmi risolutivi, ma applicare paradigmi di *aggregate computing* in grado di stabilire direttamente il comportamento globale dell'intero complesso di dispositivi. In questo contesto è stato scelto tra tutte le alternative il simulatore Alchemist, poiché anche se non disponendo di strumenti per la modellazione di robot dotati di sensori visivi, esso offre un potente motore a supporto di scenari di *aggregate computing* (specialmente in contesti che prevedono un numero molto elevato di dispositivi), in combinazione con la sua estensibilità e personalizzazione del modello. Grazie a queste caratteristiche, sono stati integrati all'interno di Alchemist due ulteriori moduli in grado di estenderne le capacità, in particolare essi si occupano di (i) modellare le interazioni fisiche tra oggetti, arricchendo il concetto di nodo mediante la definizione di area di percezione (i.e. campo visivo) e (ii) fornire supporto alle azioni dei robot come il loro controllo, movimento e rotazione della loro telecamera.

1.2.3 Interazione coi Simulatori

L'interazione con le simulazioni avviene principalmente mediante lo sviluppo di codice all'interno della JVM che comunichi direttamente con il modello di Alche-

mist. Questo porta allo sviluppo di moduli aggiuntivi coerenti con l'architettura generale del sistema, ma è evidente come la loro implementazione non può astrarre al di là delle interfacce fornite dal simulatore. Prendendo come esempio il modulo per la realizzazione di un interfaccia grafica, l'intero modulo è a conoscenza e deve adattarsi al dominio applicativo, implicandone la profonda conoscenza e comprensione dell'interazione di tutte le componenti.

1.3 Motivazioni e Obiettivi

Quanto discusso nella sezione precedente mette in luce come risulti di grande aiuto il controllo e la gestione dell'intero simulatore attraverso interfacce ad alto livello che astraggano dalla simulazione e le sue relazioni interne. In questo modo, lo sviluppo di moduli aggiuntivi è supportato dal loro utilizzo, provocando sperabilmente un aumento di efficienza nella scrittura di codice che abbia la necessità di interagire con una simulazione.

1.3.1 Interazione Efficiente tra Piattaforme Diverse

Mediante lo sviluppo di API al di sopra del modello di Alchemist, è possibile costruire meccanismi che permettano di interagire con le simulazioni in modo standardizzato per qualsiasi piattaforma che ne richieda i servizi. In particolare, questo provvede a definire due tipi di standardizzazione: una dal punto di vista delle interfacce contrattuali esposte dalle API, perciò valide all'interno di un qualsiasi sistema software (e.g. sistema *web-based* all'infuori della JVM), e un'altra per quanto riguarda le strategie impiegate per il recupero dei dati necessari o l'invio di comunicazioni alle componenti interne. Proprio grazie ad un buon design di queste ultime, si può provvedere ad esporre metodologie di accesso o controllo della simulazione nella maniera più efficiente e personalizzabile possibile.

1.3.2 Obiettivi

Obiettivo principale di questo lavoro è quello di progettare e implementare un'architettura software in grado di accedere alle componenti del simulatore Alchemist,

e rendere disponibili all'esterno servizi e funzionalità di quest'ultimo, attraverso un'insieme di API standardizzate ed efficienti, compatibili con sistemi potenzialmente all'infuori della JVM. In questo modo, si fornisce un sistema di accesso e controllo efficiente delle simulazioni eseguite all'interno di Alchemist, in grado di porsi come canale comunicativo tra applicativi software terzi e il simulatore stesso.

In generale il software prodotto deve focalizzarsi sulla produzione di un architettura generica ed estendibile in grado di adattarsi alla natura mutevole di Alchemist, fornendo allo stesso tempo un punto d'accesso consistente, flessibile ed efficiente. In Figura 1.2 sono raffigurati gli scenari di casi d'uso per questo genere di sistema di API. In generale, un utente, il quale può essere sia un utente terzo

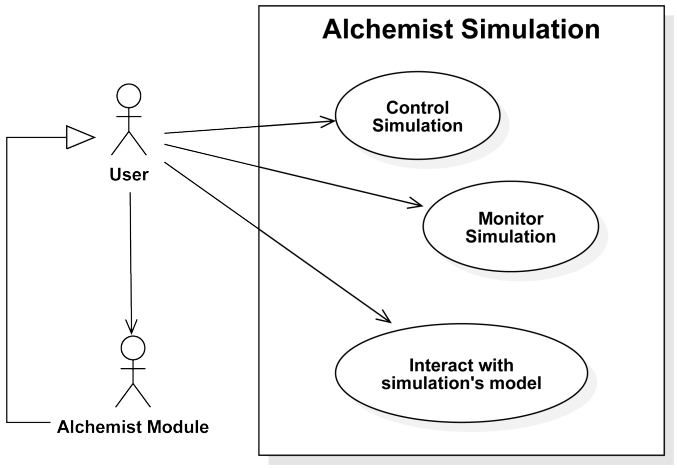


Figura 1.2: Diagramma dei casi d'uso del sistema di API per Alchemist

al sistema, sia un modulo di Alchemist, possono interagire con la simulazione attraverso un insieme di interfacce comuni, capaci di rendere disponibili informazioni riguardo la simulazione e i suoi elementi del dominio, controllare la simulazione (i.e. lanciare la simulazione, metterla in pausa e terminarla) e manipolare e interagire con gli elementi del sistema corrente. È quindi evidente che un utente generico può sia interagire con la simulazione attraverso l'impiego diretto delle API, sia attraverso un modulo che le utilizza per fornire il servizio, sfruttando il vantaggio del sistema per cui non vi sono differenze nel fornire servizi e informazioni, poiché esse sono standardizzate per ogni tipo di utilizzatore e piattaforma.

Capitolo 2

Analisi

2.1 Requisiti

L'applicativo si pone come obiettivo la realizzazione di un infrastruttura server all'interno della JVM, in grado di esporre verso l'esterno funzionalità e dati riguardanti simulazioni eseguite attraverso il simulatore Alchemist. Con infrastruttura server si intende una componente software in grado di ricevere, elaborare e rispondere attivamente a richieste da parte di client esterni al sistema, o quantomeno in un livello d'astrazione superiore al simulatore stesso (e.g. un web renderer che sfrutti il server costruito per visualizzare all'interno di un browser dati riguardanti la simulazione).

Requisiti Funzionali Innanzitutto, il suddetto server si dovrà occupare di esporre un interfaccia comune per l'instaurazione di comunicazioni. Una volta connessi, agli agenti esterni è necessario specificare quali tipi di operazioni sono permesse, e in quali modalità le risposte verranno consegnate. Principalmente devono essere esposte tre diversi tipi di operazioni:

- Recupero dati: operazioni cosiddette *one-shot* per richiedere dati riguardo la simulazione corrente.
- Recupero dati persistente: operazioni di recupero continuativo nel tempo di dati riguardo la simulazione. I dati verranno spediti al richiedente a fronte di cambiamenti di elementi d'interesse nel dominio applicativo.

- Modifica: operazioni con lo scopo di effettuare modifiche al modello applicativo, in grado di provocare *side effect* all'interno della simulazione corrente.

La comunicazione quindi terminerà con il verificarsi di uno dei seguenti eventi:

- L'operazione di recupero dati o di modifica ha avuto successo o fallisce.
- L'operazione di recupero dati persistente raggiunge una condizione di terminazione (e.g. l'oggetto osservato viene eliminato).
- La simulazione termina.

Essendo il simulatore Alchemist in grado di eseguire più simulazioni simultaneamente (*simulation batch*), il server deve essere in grado di rispondere a più richieste nello stesso istante, potenzialmente anche da client distinti. Inoltre, per via della natura variable delle diverse simulazioni, le operazioni esposte devono essere in grado di adattarsi dinamicamente in base al tipo dell'incarnazione.

Requisiti non Funzionali

- Dovendo condividere le risorse con il simulatore stesso, il server non deve costituire un eventuale collo di bottiglia che abbia un impatto negativo sulle prestazioni del sistema complessivo.
- L'applicativo sviluppato deve essere compatibile con un ambiente multi piattaforma.
- Sarebbe auspicabile che il server implementi meccanismi di estensione per agevolare l'integrazione con possibili cambiamenti del dominio applicativo.

2.2 Analisi dei requisiti

I requisiti illustrati nella sezione precedente, mettono in luce alcune caratteristiche interessanti che il sistema deve soddisfare. Primo su tutti, il vincolo sull'utilizzo della JVM per lo sviluppo di un server, il quale necessiterà di meccanismi in grado di instaurare connessioni con diversi attori e sia in grado di rispondere adeguatamente ad ogni richiesta. Queste richieste dovranno successivamente essere

elaborate e provocare una risposta, sia essa di successo o fallimento. Per la gestione di questo tipo di richieste è necessario quindi valutare le alternative architetturali definite negli anni che siano al contempo compatibili con i vincoli implementativi.

2.2.1 Analisi delle Architetture API

Nel corso degli anni sono state proposte diverse alternative di pattern architetturali in grado di guidare gli sviluppatori a costruire un software distribuito in rete in grado di sopperire al numero sempre maggiore di richieste e risorse. In questa analisi verranno presi in considerazione i due principi architetturali più utilizzati: **REST** e **GraphQL**.

REST Intorno alla fine degli anni novanta e inizio anni duemila, con la diffusione sempre maggiore di internet e di siti per uso generale, nasce l'esigenza di definire uno standard architetturale di software basati sulla rete: *REpresentational State Transfer* (REST) [FT00]. REST definisce un insieme di principi architetturali mirati alla semplicità e alla scalabilità del software. Al centro di questi principi c'è l'idea di considerare tutto come una *risorsa*, a cui si accede attraverso URL standardizzati ed univoci. Le API REST operano attraverso l'utilizzo dei metodi standard HTTP per eseguire le cosiddette operazioni CRUD (*Create, Read, Update, Delete*) sulle risorse; in particolare utilizza:

- **GET** per leggere o recuperare informazioni;
- **POST** per creare nuove risorse;
- **PUT** o **PATCH** per aggiornare risorse presenti;
- **DELETE** per eliminare risorse.

Di norma, si dice che REST è "*Stateless*", ovvero che ogni richiesta da client a server deve contenere tutte le informazioni necessarie per comprenderla e processarla. Questo significa che il server non conserva alcuna informazione sullo stato del client, e che ogni richiesta da parte del client è da considerarsi come una nuova richiesta indipendentemente dalle precedenti effettuate.

GraphQL GraphQL[1] viene sviluppato a partire dal 2012 da Facebook, e si pone come obiettivo la definizione di specifiche e principi architetturali per affrontare sfide incontrate con le tradizionali architetture software basate su REST. La principale motivazione dietro GraphQL nasce dal contesto del social network Facebook, dove molto spesso la larghezza di banda e risorse computazionali sono molto limitate. GraphQL quindi delinea linee guida per la realizzazione di architetture che permettano al client la flessibilità di poter richiedere esattamente i dati di cui necessitano. Questo avviene mediante l'impiego di un linguaggio ad hoc per la definizione di *query*, le quali specificano esattamente quali dati recuperare dal singolo *endpoint* delle API esposte dal server: in GraphQL infatti non si deve specificare un URL per ogni risorsa, ma queste vengono recuperate a partire da un singolo URL univoco (e.g. `/graphql`) in combinazione con una query GraphQL. Le operazioni all'interno di GraphQL sono:

- **Query** per leggere determinati dati;
- **Mutation** per modificare uno o più dati;
- **Subscription** per sottoscriversi ai cambiamenti di un dato, attraverso aggiornamenti in tempo reale (discussa in Sezione 2.2.2).

Non esistendo un vincolo ad una specifica base di dati, per poter operare sui dati GraphQL necessita di lavorare su uno *schema*, definito attraverso uno *Schema Definition Language* (SDL) che descrive la struttura dei dati attraverso i loro tipi e campi all'interno dello schema. Grazie a ciò, ogni operazione viene dapprima validata sullo schema, e successivamente eseguita sul server fornendo così un meccanismo di *type safety* in grado di bloccare in anticipo richieste GraphQL malformate.

Conclusioni Oggi queste due architetture convivono all'interno di molti sistemi, poiché ognuna offre i relativi pro e contro. Primo su tutti, REST risulta più semplice da un punto di vista implementativo, perché non richiede la definizione e manutenzione di uno schema che descriva i dati. Allo stesso tempo grazie a quest'ultimo GraphQL è in grado di fornire meccanismi che prevengano il fenomeno

[1] http://facebook.github.io/graphql/

di *over-fetching*, del quale soffrono invece sistemi RESTful. Per quanto riguarda l'efficienza effettiva di questi sistemi, non è possibile determinare a priori uno sull'altro, poiché per quanto GraphQL riesca a privilegiare scenari dove la scarsa banda è notevole (e.g. applicazioni per dispositivi mobili, dispositivi IoT), in alcuni sistemi risulta più efficiente un'architettura che possa sfruttare al meglio il *caching* dei dati, tipico dei sistemi REST [AR21].

In sintesi, l'adozione di una delle due architetture dipende fortemente dalla natura dell'applicativo, rendendo necessaria la conoscenza dei pro e contro di un sistema maturo e largamente diffuso come REST, oppure un paradigma relativamente nuovo e ancora in via di sviluppo, ma dalle grandi potenzialità.

2.2.2 Dati in Tempo Reale

Il sistema deve fornire le informazioni riguardo le simulazioni sperabilmente in modo asincrono ed efficiente, sia lato server sia lato client. In sistemi software dove avviene una costante richiesta da parte di un client di dati in tempo reale, può essere effettuata nei casi più semplici attraverso ciò che viene definito "*long polling*". In questo scenario, come raffigurato in Figura 2.1, viene instaurata una comunicazione per ogni volta che il client necessita di un dato, potenzialmente provocandone il blocco in attesa della risposta. È da subito evidente come se le comunicazioni sono frequenti, l'*overhead* causato dalla instaurazione della comunicazione e attesa di ricezione dei dati possono risultare in un sistema ben poco efficiente.

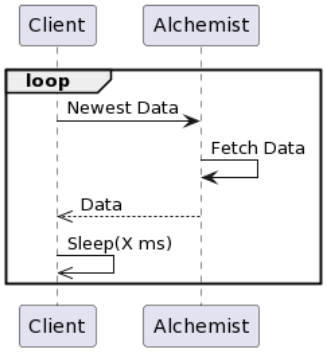


Figura 2.1: Richiesta di dati in tempo reale attraverso *long polling*.

Un sistema più efficiente è costituito dal modello *publish-subscribe* [BJ87]. Mediante l'utilizzo di questo pattern, gli utenti interessati ad un certo dato si "sottoscrivono" ad un determinato *topic*. Una volta che viene registrato un cambiamento o viene attivato un evento riguardante quel *topic*, il server ha il compito di "pubblicare" le informazioni aggiornate: a questo punto tutti i client che si sono iscritti per quel determinato *topic* riceveranno il dato aggiornato, senza bisogno di provocare ulteriori connessioni con il server stesso. In questo modo viene creata una connessione persistente tra client e il buffer di spedizione, sollevando il server dall'incarico di informare tutti i possibili client.

Per l'instaurazione di queste comunicazioni persistenti e allo stesso tempo bidirezionali, è spesso utilizzato il protocollo *WebSocket* [FM11]. Nato come soluzione per superare le limitazioni delle connessioni HTTP tradizionali, WebSocket opera su una singola connessione aperta, riducendo la latenza e garantendo trasmissioni in tempo reale. Questo protocollo e il pattern *Publish-Subscribe* aiutano a delineare una possibile configurazione di come possa essere strutturata l'architettura di messaggistica in tempo reale tra un client e il server Alchemist. In Figura 2.2 è mostrato qual è il comportamento desiderabile per il server Alchemist, il quale dovrebbe essere in grado di rispondere in maniera asincrona a tutti i possibili client che si sottoscrivono per ottenere dati in tempo reale riguardo gli elementi del dominio.

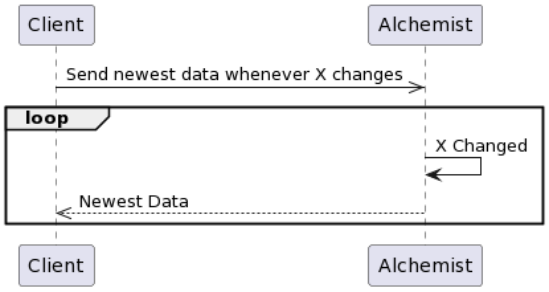
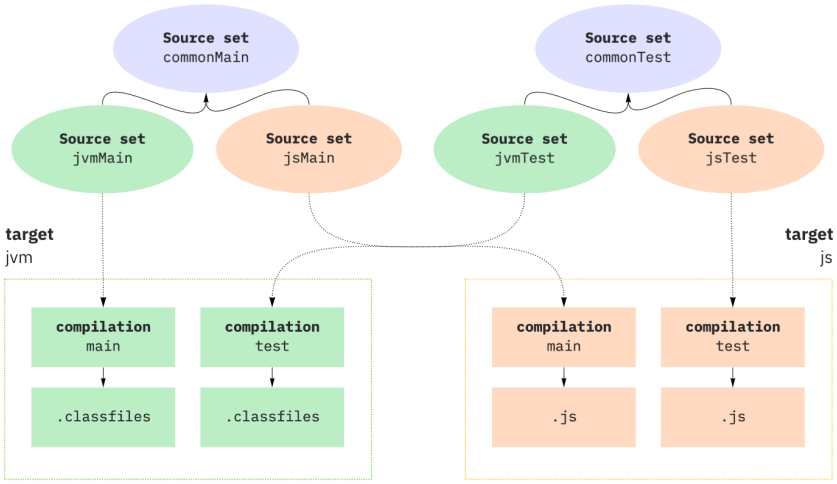


Figura 2.2: Richiesta dati in tempo reale sfruttando connessioni persistenti.

2.2.3 Multiplatform

Uno dei principali benefici del linguaggio Kotlin è il supporto alla programmazione di applicazioni multi piattaforma[2] (e.g. Android, iOS, web). Questa tecnologia permette di ridurre drasticamente lo sviluppo e il mantenimento dello stesso codice per differenti piattaforme, mantenendo la flessibilità e i benefici dello sviluppo nativo con Kotlin.

Un progetto multiplatform si compone di una serie di moduli, detti *source sets*, ognuno dei quali contiene al suo interno le componenti che hanno come target una specifica piattaforma, come illustrato in Figura 2.3. Tutti i moduli condividono il *source set Common Kotlin*, il quale mantiene le librerie fondamentali per il linguaggio Kotlin e tutto il codice portabile nelle diverse piattaforme.



Source: `https://kotlinlang.org/docs/multiplatform-discover-project.html`

Figura 2.3: Struttura di un progetto Kotlin Multiplatform per due piattaforme: *Kotlin/JVM* e *Kotlin/JS*

Kotlin mutliplatform può essere impiegato in molte applicazioni, ma prevalentemente si ricordano:

- **Applicazioni Android, iOS e web**: in questo contesto, la logica dell'applicativo si sviluppa all'interno del modulo comune a tutte le piattaforme, men-

[2]https://kotlinlang.org/docs/multiplatform.html

tre per esempio le specifiche implementazioni del front-end vengono delegate al relativo modulo. In questo modo, il codice specifico per ogni piattaforma viene ridotto notevolmente.

- **Applicazioni Web Full-stack**: in questo caso i moduli previsti possono consistere, oltre al modulo comune, un modulo *Kotlin/JVM* per una componente server e un modulo *Kotlin/JS* per delineare i comportamenti lato front-end. All'interno del modulo comune possono essere condivisi elementi del modello e logiche che possono essere condivisibili tra client e server.
- **Librerie Multiplatfom**: creando librerie con codice comune e implementazioni specifiche per le piattaforme, come JVM o web, è possibile utilizzarle come dipendenze in altri progetti multi piattaforma.

2.3 Analisi e Modello del Dominio

L'architettura software proposta dovrà operare al di sopra del simulatore Alchemist, il quale modello è stato ampiamente esplorato all'interno di Sezione 1.2.2. In questo contesto, è quindi dapprima necessario fornire un livello comunicativo tra il modello del dominio di Alchemist, e le componenti del sistema server, agendo quindi come *controller* all'interno del paradigma architetturale *Model-View-Controller* (MVC) [ERRJ]. Il server sarà quindi responsabile dell'accesso e recupero dati nel contesto di una simulazione, e allo stesso tempo fornire un punto di accesso per le comunicazioni con i client. Queste comunicazioni devono permette il transito sia di informazioni, sia di comandi specifici che il server deve far attuare all'interno di una simulazione. D'altra parte, il client deve essere in grado di stabilire e inviare correttamente questo tipo di operazioni al server.

In figura Figura 2.4 è illustrato il diagramma UML delle componenti del modello dell'applicativo. La componente server è composta a sua volta da un livello comunicativo con ciò che rappresenta una simulazione Alchemist, e l'esposizione di un interfaccia comune su cui i client possono stabilire una connessione. All'interno di questa interfaccia vengono stabilite tutte le operazioni eseguibili all'interno di una simulazione. Il client comprende un modulo principale incaricato di interporsi tra le applicazioni sviluppate potenzialmente in ambienti multi piattaforma, e il

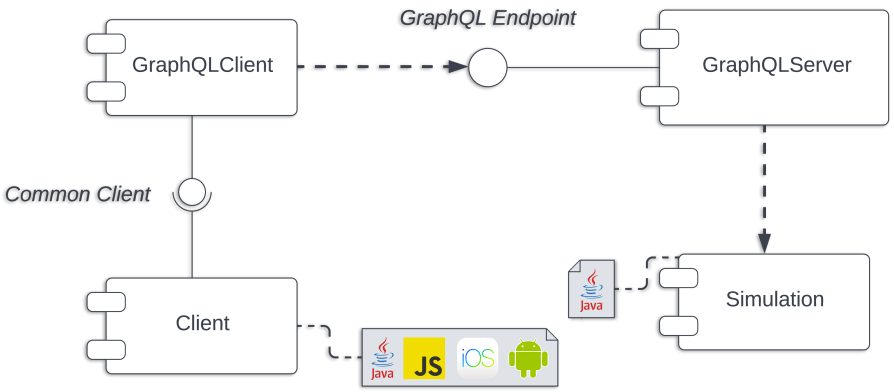


Figura 2.4: Diagramma UML dei componenti dell'architettura di alto livello

punto di accesso fornito dal server (*endpoint*). Ciò che l'applicativo client vero e proprio esegue, è contenuto all'interno della componente `Client`.

Siccome Alchemist prevede diverse modalità operative, come per esempio all'interno di una GUI o all'interno di un browser, la componente server deve poter essere in grado di essere lanciata solo attraverso specifica configurazione all'interno della simulazione. In questo modo il modulo verrà eseguito solo se richiesto esplicitamente.

Capitolo 3

Design

3.1 Architettura ad Alto Livello

L'analisi dell'applicativo può essere scomposta in due sezioni principali, rispettivamente per il server e il client. Per quanto il primo sia il fulcro di questo elaborato, il client sarà necessario per dimostrare il funzionamento e l'impiego delle API.

Una rappresentazione di alto livello dell'architettura è raffigurata in Figura 2.4, mostrando il funzionamento generale dell'applicazione considerando anche le interazioni dei vari componenti. Possiamo definire i ruoli come segue:

- **Simulation**: punto di accesso delle API Alchemist. È responsabilità di questo modulo gestire una simulazione e fornire un interfaccia di accesso all'ambiente e a tutte le sue componenti. Questo componente sarebbe il *model* nel pattern architetturale MVC. Questa componente è già implementata all'interno del codice di Alchemist.

- **GraphQLServer**: possiamo definire due ruoli principali:

  - Gestione del modello surrogato di Alchemist
  - Esposizione del servizio di API GraphQL

  Questa componente ricopre il ruolo di *controller* dell'intero applicativo, gestendo le richieste in entrata dai client, risolvendole interagendo con il modello di Alchemist esposto dalla componente *Simulation*.

- **GraphQLClient**: insieme di elementi in grado di rendere possibile l'instaurazione di una connessione ad un server, ed effettuare su di esso operazioni GraphQL. Questo modulo fa parte del codice *common*, rendendolo quindi indipendente dalla piattaforma.
- **Client**: Versione *platform specific* di un applicativo che sfrutti il client GraphQL comune. Questa componente mantiene tutta la logica di sistema di un servizio lato client, il quale sfrutta le API GraphQL di Alchemist.

3.2 Server

Il server ha lo scopo di gestione delle comunicazioni con i client e risoluzione delle richieste attraverso opportune interazioni con il modello di Alchemist. Di seguito verranno illustrate le componenti fondamentali e l'architettura nel suo complesso.

3.2.1 Architettura

Il diagramma UML in Figura 3.1 rappresenta l'architettura della componente server. Possiamo quindi descrivere il server come segue:

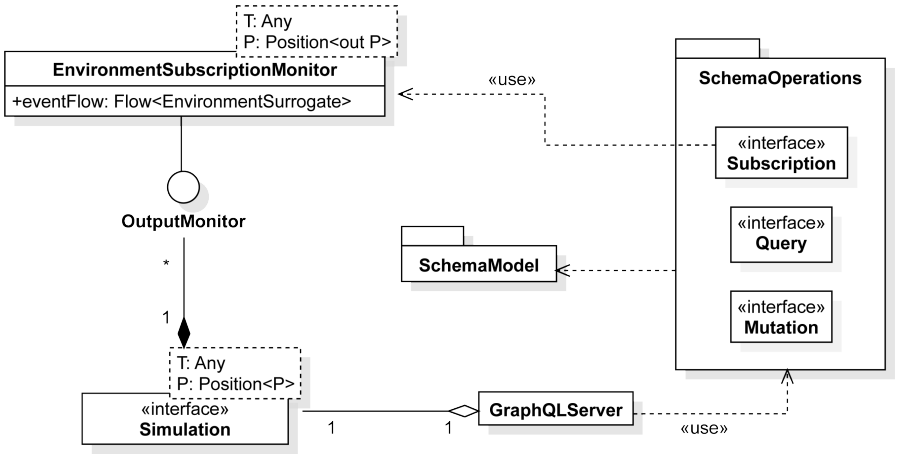


Figura 3.1: Architettura della componente server

- **GraphQLServer**: è la componente principale dell'intera architettura, si occupa di esporre un insieme di API GraphQL all'interno di una simulazione.

Questa componente mantiene quindi un riferimento alla simulazione corrente, attraverso la quale produce le istanze delle operazioni all'interno dello schema.

- **SchemaOperations**: è uno dei due moduli che contribuiscono alla creazione dello schema GraphQL. Al suo interno risiedono tutte le operazioni dei tre tipi *query*, *mutation* e *subscription*.

- **SchemaModel**: modulo che contiene tutte le componenti del dominio di Alchemist adattate per la rappresentazione all'interno di uno schema GraphQL. Le motivazioni dietro la necessità di questo modulo saranno illustrate in Sezione 3.2.3, e più in dettaglio in Sezione 4.1.1.

- **OutputMonitor**: interfaccia di Alchemist che mira al monitoraggio e all'osservazione sullo stato di una simulazione attraverso il design pattern *Observer* [ERRJ].

- **EnvironmentSubscriptionMonitor**: Realizzazione dell'interfaccia `OutputMonitor`, la quale ha il compito di fornire una versione aggiornata dell'ambiente a tutte le operazioni di tipo *subscription* presenti nello schema. In questo modo viene implementato il trasferimento di dati in tempo reale ad ogni aggiornamento della simulazione. Il suo uso e funzionamento sono descritti in dettaglio all'interno della Sezione 4.2.2.

3.2.2 API GraphQL

Come detto in precedenza, le API GraphQL costruite mirano a fornire una visuale ad alto livello di una o più simulazioni attraverso operazioni di tipo *query*, modificare lo stato della simulazione o elementi del dominio attraverso *mutation* e sottoscriversi a cambiamenti degli stessi attraverso *subscription*. L'esecuzione di queste operazioni deve avvenire in modo asincrono, affinché non venga interrotto il funzionamento del simulatore per la risoluzione delle richieste.

A differenza dei sistemi RESTful, i quali espongono i servizi attraverso differenti *endpoint* per l'esecuzione di operazioni, GraphQL opera esponendo un singolo end-

point (tipicamente `/graphql`) sul quale inviare l'operazione GraphQL desiderata secondo lo schema:

- **query**: operazione GraphQL desiderata.
- **variables**: variabili in input all'operazione specificata nel campo "query".
- **operationName**: nome dell'operazione da eseguire (opzionale).

Essendo GraphQL agnostico rispetto al protocollo utilizzato, le specifiche non definiscono un protocollo di trasporto standard, ma è ampiamente adottato e *de-facto* standard l'impiego di HTTP. All'interno di questo contesto, gran parte dei sistemi GraphQL supportano sia il metodo GET, sia il metodo POST di HTTP. Il primo prevede l'inserimento dell'operazione GraphQL (formattata secondo lo schema sopra descritto) direttamente nell'URI dell'endpoint GraphQL. È evidente come però se la rappresentazione in stringa dell'operazione sia abbastanza lunga, si incorre nell'errore `414 URI Too Long`. Proprio per questo motivo è largamente diffuso inserire nel *payload* del metodo POST l'operazione GraphQL per evitare questo genere di problematiche.

Dopo aver definito le metodologie di comunicazione tra client e server, è necessaria la definizione dei *resolver* per le operazioni GraphQL. Il resolver ha il compito di effettuare il *parsing* dell'operazione e successivamente la validazione della stessa sullo schema GraphQL. Se la validazione ha esito positivo, verrà eseguita la relativa operazione di recupero dei dati (*fetching*) e inviata la risposta al client. È importante sottolineare come la creazione di tale architettura può avvenire mediante due approcci:

- *schema-first*: viene dapprima creato lo schema GraphQL in base agli elementi del modello, e successivamente per ogni elemento dello schema si definisce un *resolver*.
- *code-first*: lo schema e i resolver vengono generati con la compilazione del codice sorgente contenente gli elementi del modello e le operazioni GraphQL.

Nel caso di Alchemist, è sicuramente preferibile il secondo approccio per evitare che un cambiamento nel dominio applicativo debba essere propagato all'interno

dello schema e tutti i relativi resolver. Per quanto questo approccio porti vantaggi riguardo l'astrazione dallo schema e dalla risoluzione delle operazioni, le specifiche GraphQL non permettono una rappresentazione uno a uno con il modello di Alchemist.

3.2.3 Adattamento del Modello

GraphQL impone delle specifiche che mirano alla definizione di uno schema in grado di esprimere esaustivamente gli elementi del modello e le operazioni che possono essere effettuate su di essi, mantenendo allo stesso tempo un elevato livello di semplicità. Per quanto queste specifiche siano efficaci per sistemi con un modello piuttosto semplice, all'interno di Alchemist il modello presenta molti elementi incompatibili con le suddette specifiche. In particolare, consideriamo le seguenti:

- "*Interfaces are never valid inputs.*"[1]. Questo significa che ogni interfaccia definita nel modello di Alchemist, non può figurare come input o parametro di una funzione o un metodo all'interno dello schema GraphQL.
- I tipi generici non sono ammessi.

Per la natura di Alchemist, l'estendibilità e la genericità sono due componenti fondamentali. Questo implica che all'interno del modello viene fatto un ampio uso di interfacce e polimorfismo attraverso l'impiego di generici, rendendo impossibile una rappresentazione diretta di questi elementi all'interno di uno schema GraphQL. Risulta quindi necessario definire uno strato intermedio di oggetti che siano conformi alle specifiche, i quali siano facilmente riconducibili ad oggetti Alchemist. Queste classi sono quindi rappresentabili attraverso il pattern architetturale *Adapter* [ERRJ], nella sua declinazione di *Object Adapter*. Un esempio di come viene applicato questo pattern è raffigurato in Figura 3.2, dove l'oggetto `Node` di Alchemist viene esposto dal server attraverso il suo surrogato, `NodeSurrogate`. Ogni surrogato estenderà la classe aperta (`open class`) `GraphQLSurrogate`, specificando come template l'oggetto del modello di Alchemist incapsulato. In questo modo tutti gli oggetti che figurano all'interno di Query, Mutation o Subscription

[1] http://spec.graphql.org/draft/#sec-Interfaces

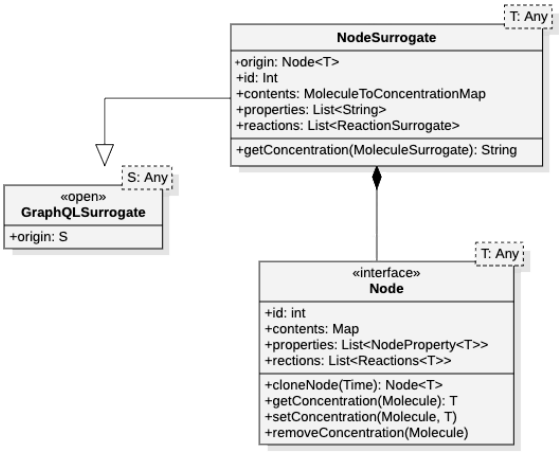


Figura 3.2: Applicazione del pattern *adapter* per la classe `node`.

saranno solo surrogati, i quali (lato server) delegano la risoluzione dell'operazione all'oggetto Alchemist sottostante.

Modello Risultante Attraverso l'impiego di classi surrogate, il modello che viene a crearsi è rappresentato in Figura 3.3

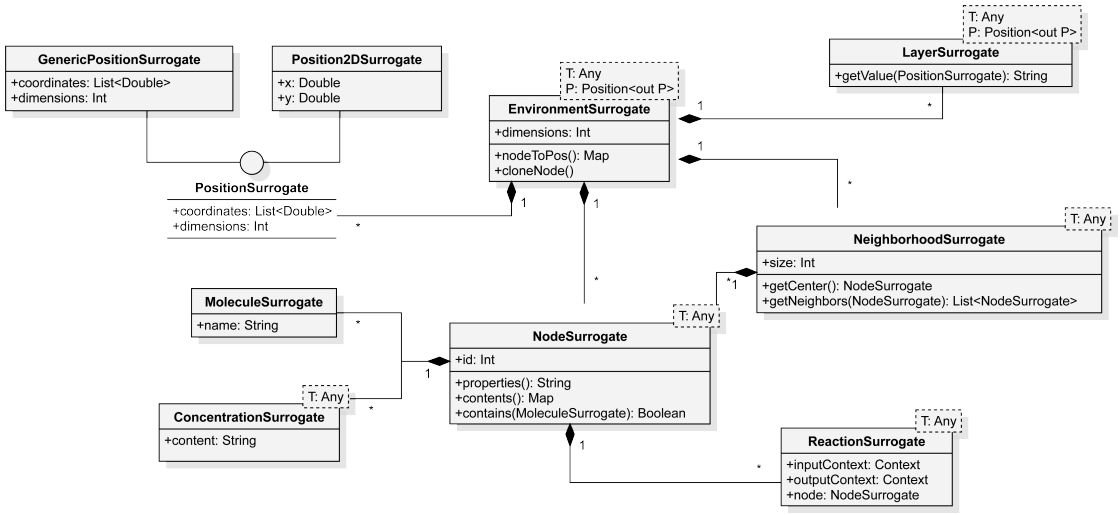


Figura 3.3: Modello di Alchemist rappresentato attraverso Surrogati GraphQL

Questo modello risulta una versione semplificata del modello di Alchemist implementato all'interno della JVM, il quale a sua volta è la realizzazione del metamodello descritto in Sezione 1.2.2.

L'elemento fondamentale del dominio è `EnvironmentSurrogate`, il quale mantiene al suo interno i nodi, associandogli la loro posizione, i vicinati composti dai nodi e gli eventuali `LayerSurrogate`. Questi ultimi sono in grado di calcolare la concentrazione in un determinato punto dello spazio che occupano. I nodi, rappresentati da `NodeSurrogate` e identificati da un `id` univoco, hanno al loro interno un insieme di molecole a cui è associata una concentrazione, un insieme di proprietà del nodo stesso e mantengono un riferimento alle reazioni a lui associate, descritte da `ReactionSurrogte`. Un insieme di nodi forma un `NeighborhoodSurrogate`, il quale è caratterizzato da un nodo che ne rappresenta il centro. La posizione di un nodo nello spazio è rappresentata attraverso l'interfaccia `PositionSurorgate`, la quale può essere del tipo specifico `Position2DSurrogate`, o nel caso più generico, una `GenericPositionSurrogate`.

È importante specificare che per la realizzazione del sistema software, è stato scelto un sottoinsieme di elementi del dominio di Alchemist, i quali possano essere

in grado di fornire una visione ad alto livello della simulazione corrente. In ogni caso, il software è costruito in modo tale che sia possibile una futura integrazione di ulteriori elementi che vengano reputati utili ai fini dell'impiego delle API GraphQL, come illustrato nei lavori futuri in Sezione 5.1.

3.3 Client

Considerando il contesto *Multiplatform* del sistema, ci si concentrerà su tutte quelle componenti che saranno condivise tra tutte le tipologie di client che verranno sviluppate in base alla piattaforma, ovvero tutti i moduli contenuti all'interno del *source-set commonMain*. Per poter effettuare operazioni GraphQL, un client deve essere in grado di (i) instaurare una connessione con un server GraphQL, (ii) recuperare lo schema attraverso un'operazione detta di *schema introspection*, (iii) effettuare un operazione GraphQL, (iv) attendere, recuperare e interpretare il risultato ottenuto.

3.3.1 Architettura generale

L'architettura del client è illustrata in figura Figura 3.4.

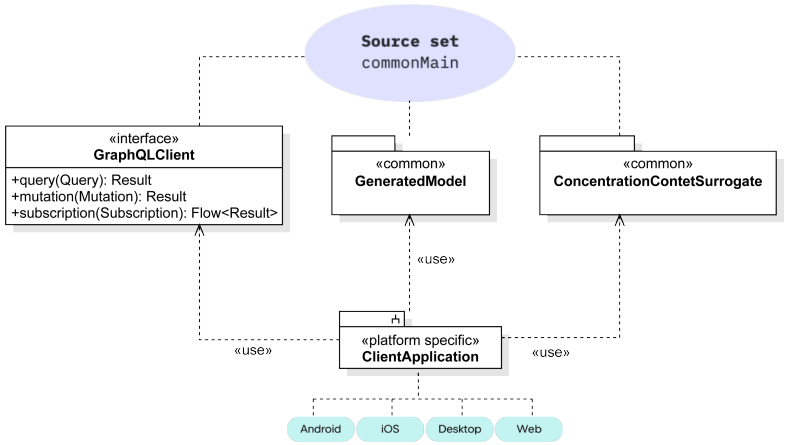


Figura 3.4: Architettura della componente client

Possiamo distinguere le seguenti componenti:

- **GraphQLClient**: client GraphQL in grado di stabilire una connessione con il server, ed effettuare su di esso *query*, *mutation* e *subscription*. Per via delle varie componenti che possono essere aggiunte al client, come per esempio il modulo di *WebSocket* per effettuare *subscription*, la scelta del protocollo di trasporto di GraphQL o l'aggiunta di un modulo di intercettamento dei pacchetti HTTP, il client GraphQL viene costruito mediante il pattern creazionale *Factory* [ERRJ]. Questa componente è all'interno del *source set* comune (*commonMain*).

- **GeneratedModel**: modulo contenente il modello di Alchemist generato a partire dallo schema GraphQL esposto dal server, utilizzato da tutte le operazioni definite all'interno del client.

- **ConcentrationContentSurrogate**: modulo speciale per definire, attraverso surrogati, i contenuti complessi delle concentrazioni, definendone anche le strategie di serializzazione e deserializzazione. Il loro scopo è meglio descritto all'interno della Sezione 4.1.2. Insieme al client e al modello generato, questi risiedono all'interno del *source set commonMain*.

- **ClientApplication**: questo sottosistema è l'implementazione specifica per una determinata piattaforma, la quale esponga un servizio che faccia uso del client GraphQL per l'esecuzione di API.

3.3.2 Effettuazione operazioni GraphQL

Se il server genera lo schema GraphQL a partire dal codice del modello, il client genera il codice del modello grazie allo schema. Il client infatti necessita a tempo di compilazione dello schema GraphQL del server, affinché sia possibile la validazione delle operazioni richieste nel codice già a tempo di compilazione. Questo provvede a garantire una *type safety* già nel momento in cui il codice client viene scritto, evitando preventivamente possibili operazioni GraphQL malformate. Come anticipato nella sezione precedente, lo schema può essere recuperato mediante un'operazione chiamata *introspection*. Questa operazione permette di interrogare uno speciale *endpoint* del server GraphQL, generalmente `/sdl`, in grado di for-

nire al richiedente lo schema completo del modello, insieme a tutte le operazioni eseguibili.

Una volta che il codice del modello è stato generato a partire dallo schema, il client può fornire un insieme di operazioni eseguibili sul server attraverso un insieme di file con estensione ".graphql" dove sono contenuti i dettagli di quali dati specifici si vuole ricavare da un operazione esposta dal server.

In figura Figura 3.5 è riassunto mediante un diagramma di flusso il funzionamento di un client GraphQL.

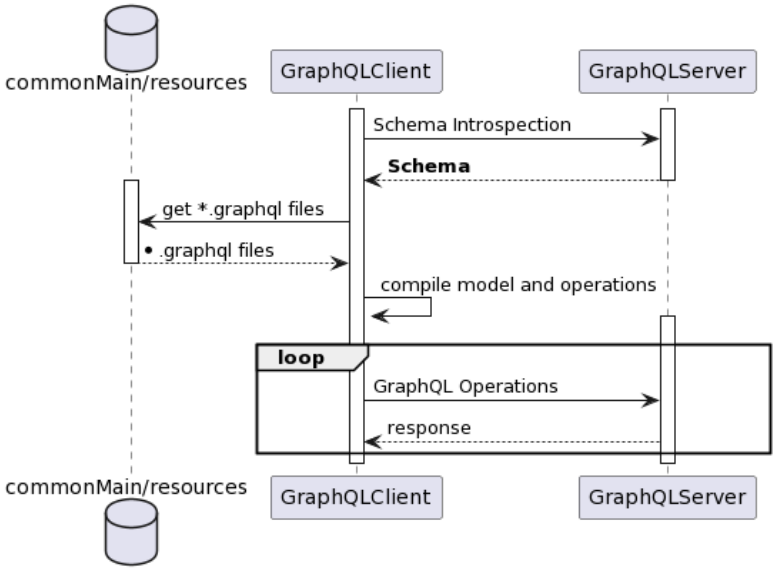


Figura 3.5: Utilizzo del server GraphQL lato client

Di seguito vengono elencate le funzionalità basilari che il client deve poter essere in grado di esguire:

- **Query**: le query sono tutte quelle operazioni che prevedono il recupero di un dato dal server. Per una versione dimostrativa del software un client deve essere in grado di effettuare query per recuperare:
  - Stato della simulazione.
  - Elenco dei nodi e loro posizioni.
  - Data una posizione e un `Layer`, conoscere il valore in quel punto.

  - Concentrazione di una molecola all'interno di nodo.
  - Dato un nodo, avere l'elenco dei nodi suoi vicini (`Neighborhood`).

- **Mutation**: queste operazioni effettuano modifiche all'interno della simulazione, e possiamo comprendere tra queste sia operazioni in grado di modificare lo stato della simulazione (*play*, *pause* e *terminate*), sia operazioni che modificano l'ambiente (e.g. aggiunta di un nodo, aggiunta di un layer, clonazione di un nodo, ...).

- **Subscription**: la raccolta di informazioni in tempo reale riguardo un elemento del dominio è una delle operazioni fondamentali per i requisiti del sistema. Queste operazioni possono comprendere tutti gli elementi del dominio che hanno una natura mutevole (e.g. i nodi, le concentrazioni).

3.3.3 Client *Kotlin/JS*

Come dimostrazione del funzionamento dell'intero sistema software costruito, si provvede un'implementazione di un client all'interno di un ambiente web, sviluppato grazie al supporto di Kotlin Multiplatform per piattaforme Kotlin/JS. All'interno di questo modulo, deve essere resa disponibile un'interfaccia grafica in grado di riassumere le operazioni chiave eseguibili sul server GraphQL. In questo modo un utente deve essere in grado di:

- Gestione della simulazione corrente attraverso l'uso delle rispettive *mutation* GraphQL.

- Visualizzare a schermo dati e informazioni riguardo elementi del dominio.

All'interno di questo contesto è necessario definire una nuova *route* all'interno del server che carichi la pagina HTML per l'esecuzione di tali operazioni, e per semplicità verrà mappata la rotta "`/`" all'indirizzo del server. In questo un client può visitare questa pagina collegandosi direttamente all'indirizzo e host del server. Attraverso questa pagina sarà quindi possibile l'esecuzione di operazioni di gestione della simulazione attraverso una serie di bottoni, mentre per la visualizzazione di dati e informazioni può essere implementata attraverso una libreria grafica per la creazione di grafici.

Capitolo 4

Implementazione e Verifica

4.1 Superamento delle specifiche GraphQL

Come illustrato nella sezione Sezione 3.2.2, l'approccio desiderabile di sviluppo del codice per lo schema GraphQL è un approccio di tipo *code-first*. La libreria GraphQL Kotlin [1] offre proprio questo tipo di strategia, permettendo di astrarre il più possibile dalla implementazione dello schema. Questo però comporta inevitabilmente un conflitto tra il modello di Alchemist e le specifiche GraphQL, le quali non sono in grado di fornire una rappresentazione uno a uno del modello. In questo contesto nasce la necessità di definire dei surrogati per adattare il modello di Alchemist ad uno schema che sia conforme alle specifiche, e allo stesso tempo garantire la genericità dipendente dalle diverse *incarnation*.

4.1.1 Surrogati (*Object Adapter Design Pattern*)

I surrogati per il modello di Alchemist hanno il compito di adattare elementi complessi del dominio con versioni meno complesse e compatibili con uno schema GraphQL. Nel contesto dell'architettura client-server, oltre a ricoprire questo ruolo, i surrogati hanno anche la funzione di *Data Transfer Object* (DTO), poiché solo essi compariranno all'interno delle comunicazioni in rete.

Siccome la generazione dello schema GraphQL è delegata alla libreria impiegata, è importante notare come i tipi di Kotlin siano interpretati e adattati allo

[1] https://opensource.expediagroup.com/graphql-kotlin/

schema GraphQL. Di seguito sono riportate le trasformazioni tra tipi di Kotlin e tipi GraphQL:

- **Tipi Primitivi**: i tipi primitivi di Kotlin come `Int`, `Float` e `Double`, `String` e `Boolean` sono nativamente supportati da GraphQL, permettendo quindi un adattamento immediato tra codice Kotlin e schema GraphQL.
- **Classi**: tutte le classi definite in Kotlin, vengono interpretate all'interno dello schema come di tipo `type`. Un `type` può comparire come valore di ritorno di un campo dello schema GraphQL, e può estendere un'interfaccia.
- **Metodi e Proprietà**: i metodi e proprietà vengono trasposti in uno schema GraphQL come campi di una struttura di tipo `type`, solo se essi risultano pubblici e non annotati con `@GraphQLIgnore` .
- **Input**: ogni classe che figura come parametro all'interno di un metodo (se pubblico e non annotato con `@GraphQLIgnore`) viene interpretato all'interno dello schema GraphQL come di tipo `input`. Un tipo `input` può contenere a sua volta dei campi, ma non può estendere un'interfaccia.
- **Interfacce, Classi Astratte e Classi *Sealed***: tutte le interfacce, classi astratte e *sealed* definite tali all'interno del codice Kotlin, vengono interpretate come tipo `interface` all'interno dello schema GraphQL. Un'interfaccia si compone di campi all'interno dello schema, può figurare come valore di ritorno di un metodo o di una proprietà ma *non* può essere un parametro di un metodo.

Durante la fase di generazione dello schema a partire dal codice Kotlin quindi, viene stabilita la corretta relazione tra tipo Kotlin e tipo GraphQL, e in caso in caso di incompatibilità la generazione dello schema fallisce. Se considerassimo per esempio il metodo `Node<T>.getConcentration(m: Molecule): T` , è da subito evidente che è incompatibile con uno schema GraphQL, sia per l'utilizzo dell'interfaccia `Molecule` come parametro, sia per l'impiego del generico `<T>` (il tipo di `Conc␣entration`).

Il modello di Alchemist, facendo pesantemente utilizzo del polimorfismo attraverso interfacce e classi astratte, risulta quindi immediatamente incompatibile con

uno schema GraphQL senza l'intervento di un livello ulteriore di astrazione. In questo scenario si interpongono i surrogati delle classi di Alchemist, definiti come in Elenco 1. Ogni surrogato è generico per il tipo `<S>`, ovvero l'oggetto Alchemist per il quale viene costruito l'*adapter*. In questo modo, possiamo definire un modello surrogato come illustrato nel diagramma delle classi in Figura 3.3.

```
1 @GraphQLIgnore
2 open class GraphQLSurrogate<S> {
3     @GraphQLIgnore open val origin: S,
4 }
```

Listato 1: Definizione di un Surrogato Generico GraphQL per il modello di Alchemist.

Surrogati e Comunicazioni da Client a Server Costruendo i surrogati in questo modo, è possibile mappare un elemento del modello di Alchemist ad un surrogato, ma non viceversa. Per tutte le operazioni che prevedono l'invio di un surrogato dal client infatti, non sono presenti alcuni riferimenti all'oggetto originario (poiché si ricorda essere annotato con `@GraphQLIgnore open val origin: S`), portando il client ad un'impossibilità di creare e spedire un surrogato Alchemist così definito. A questo punto quindi, per tutte le operazioni che prevedono l'impiego di un surrogato in input (principalmente *mutation*) è necessario definire una nuova classe istanziabile dal client e presente nello schema GraphQL con il tipo `input`. In questo modo il client è in grado di definire surrogati e utilizzarli come input di operazioni, e riprendendo l'esempio di prima, il metodo per recuperare una concentrazione data una molecola sarà mappato in termini di surrogati in `NodeSurrogate<T>.getConcentration(m: MoleculeInput): String`, dove `MoleculeInput` è l'oggetto in input istanziabile dal client. A questo punto sarà il server a definire le strategie di conversione tra oggetti di tipo `input` e il l'oggetto Alchemist corrispondente.

Per evitare la modifica del codice sorgente delle API del modello di Alchemist, grazie alle *extension functions* di Kotlin è possibile definire funzioni del tipo `to`

`XxxGraphQLSurrogate()` in grado di aggiungere un nuovo metodo ad un oggetto senza la necessità di ereditarietà o l'impiego di design patterns. In Elenco 2 è mostrata la funzione di conversione tra un nodo Alchemist e il relativo surrogato GraphQL, la quale non ha argomenti in input e crea un `NodeSurrogate` fornendo un riferimento a se stesso e l'`id` stesso del nodo.

```
1 fun <T> Node<T>.toGraphQLNodeSurrogate() = NodeSurrogate(this, id)
```

Listato 2: Conversione da `Node` al suo relativo surrogato GraphQL

4.1.2 Gestione del Polimorfismo

Come definito in Sezione 3.2.3, le principali sfide da affrontare per l'adattamento del modello di Alchemist con uno schema che sia compatibile con le specifiche GraphQL sono principalmente l'impossibilità di utilizzare interfacce come input, e la mancanza di supporto per i tipi generici. Per quanto riguarda la prima, è risolvibile mediante i surrogati, mentre per la seconda è necessario un ragionamento più approfondito riguardo le motivazioni dietro l'uso di tali tipi generici.

Position In Alchemist, una posizione è definita come `Position<P : Position<P>>` , utilizzando il cosiddetto "*recursive bound*". Questa tecnica è principalmente utilizzata per imporre una struttura gerarchica specifica sul tipo delle posizioni. Ciò significa che ogni classe che implementa questa interfaccia definisce un tipo di `Position` raffinandone la definizione attraverso l'ereditarietà, consentendo una modellazione più dettagliata, adattata al dominio di uno specifico contesto (ad esempio, `Position2D` per modellazioni di spazi bidimensionali).

Poiché la gerarchia delle posizioni è piuttosto limitata e ben definibile, per poter convertire una generica `Position` è sufficiente fornire allo schema GraphQL un surrogato per una posizione generica, e almeno una sua realizzazione concreta. Nel sistema raffigurato in Figura 3.3 è mostrato come si forniscano due realizzazioni dell'interfaccia `PositionSurrogate`: `GenericPositionSurrogate` e `Position2DSurrogate`. In questo modo, data una generica istanza di una

```
fun toPositionSurrogate(position: Position<*>) =
    when (position.dimensions) {
        2 -> Position2DSurrogate(
            position.coordinates[0],
            position.coordinates[1],
        )
        else -> GenericPositionSurrogate(
            position.coordinates.toList(),
            position.dimensions,
        )
    }
```

Listato 3: Conversione di una generica `Position` Alchemist, nel suo relativo surrogato.

`Position` all'interno di Alchemist, possiamo mappare il corretto surrogato GraphQL come mostrato in Elenco 3 a partire dal numero di dimensioni dello spazio della posizione. Questo esempio mira a delineare che, per ogni tipo di posizione, sarà sufficiente definire un surrogato basilare e una strategia di conversione per adattare un nuovo tipo di posizione (e.g. `GeoPosition` composto da latitudine e longitudine).

Il vantaggio principale nell'impiego di un'interfaccia all'interno di GraphQL, oltre alla definizione di un super-tipo per una serie di oggetti, risiede nella possibilità di utilizzare gli "*inline fragments*". Questo operatore, utilizzabile all'interno di operazioni GraphQL, viene impiegato quando il valore di ritorno di un campo è un tipo composito (i.e. `interface` e `union`) e permette di specificare selezioni di campi differenti in base al tipo tornato a *runtime*. In Elenco 4 è mostrato l'utilizzo di *inline fragments* per le diverse possibili implementazioni di `Position`.

Concentration Il tipo delle concentrazioni in Alchemist, dipende esclusivamente dal tipo di *Incarnation* selezionta, come descritto in Sezione 1.2.2. Diversamente da quanto accade per le posizioni, questo tipo non ha una interfaccia comune da

```
query NodePosition($nodeId: Int!) {
    nodePosition(nodeId: $nodeId) {
        dimensions
        coordinates
        ... on Position2DSurrogate {
            x
            y
        }
    }
}
```

Listato 4: Utilizzo di *inline fragments* per il tipo `PositionSurrogate`

cui tutte le classi ereditano, e perciò risulta più complessa la sua rappresentazione all'interno di uno schema GraphQL.

Per ogni tipo non facilmente definibile all'interno di uno schema GraphQL, è possibile impiegare uno speciale tipo di dato chiamato *Custom Scalar*[2]. Questo dato può non essere conforme alle specifiche, ma è necessario definirne strategie di interpretazione lato server prima di essere spedito, e allo stesso tempo definire lato ricevente come interpretare il *custom scalar* ricevuto.

Solitamente per inserire un custom scalar all'interno di un comunicazione GraphQL, esso viene serializzato come oggetto JSON, in modo tale da avere una rappresentazione univoca lato ricezione. In questo contesto si inserisce la libreria `kotlinx.serialization` in grado di fornire le funzionalità di serializzazione e deserializzazione anche in ambienti *multiplatform*. `kotlinx.serialization` offre implementazioni di default per i tipi primitivi di Kotlin, comprendendo anche gran parte delle *Collections* del linguaggio, risultando quindi una valida scelta per tutti quei casi dove il tipo delle concentrazioni è semplice. Per tutti i casi dove invece non è possibile effettuare una serializzazione con il *serializer* di default, è necessario definirne una esplicita implementazione.

L'idea di fondo è quella di fornire un surrogato nel *source-set commonMain*

[2]http://spec.graphql.org/draft/#sec-Scalars.Custom-Scalars

per il tipo `T` aderente alle specifiche GraphQL, e allo stesso tempo includere un *serializer* di `kotlinx.serialization`: questo può essere fatto semplicemente includendo nella dichiarazione della classe l'annotazione `@Serializable` . Definendo così il surrogato del contenuto della concentrazione, si è in grado di includere, spedire e ricevere correttamente un tipo *custom* di GraphQL, serializzato in formato JSON per motivi di compatibilità con i protocolli impiegati dalle librerie. Allo stato attuale, se un *serializer* non viene fornito per il tipo `T`, allora viene spedita la sua rappresentazione in stringa. In Elenco 5 è mostrato come venga effettuata la serializzazione del contenuto di una concentrazione. Attraverso la

```
1 fun <T : Any> encodeConcentration(content: T): String =
2     runCatching {
3         Json.Default
4         .encodeToString(serializer(content::class.java), content)
5     }.getOrElse {
6         content.toString()
7     }
```

Listato 5: Serializzazione del contenuto di una concentrazione in formato JSON

chiamata a funzione `serializer(content::class.java)` , si cerca di utilizzare il `serializer` per la classe del tipo `T`, e in caso non venga trovato, verrà lanciata una `SerializationException`, la quale verrà catturata per poter fornire una rappresentazione in stringa del tipo.

Per motivi di semplicità e leggerezza dello schema e della notazione, siccome il contenuto delle concentrazioni è sempre serializzato in formato JSON sotto forma di stringa, non è necessario definire un nuovo *custom scalar* per rappresentarlo, ma è sufficiente considerarlo come una semplice stringa, opportunamente dettagliando la documentazione riguardo che cosa è rappresentato all'interno della stringa stessa.

Deserializzazione lato Client Il vantaggio di porre il surrogato per il contenuto della concentrazione all'interno del package *commonMain* risiede nel poter

riutilizzare il `serializer` sia per effettuare l'*ecnoding* in formato JSON all'interno del codice JVM del server, sia per deserializzare il JSON ricevuto lato client all'interno dell'implementazione per una specifica piattaforma. In questo modo lato client viene ricevuta una concentrazione serializzata in JSON di cui ne si conosce il tipo, ed è quindi possibile deserializzarla attraverso una chiamata a funzione del metodo `Json.decodeFromString<T>(data: String)` , dove `<T>` è il tipo effettivo dell'oggetto da deserializzare. È importante notare come è necessaria una profonda conoscenza del tipo in arrivo lato client, effettuata consultando la documentazione relativa all'operazione chiamata all'interno dello Schema GraphQL e in base alla *incarnation* per la simulazione in esecuzione all'interno di Alchemist.

4.2 Funzionamento

4.2.1 Web Server

Il servizio di API è reso disponibile all'esterno attraverso un web server implementato mediante l'uso di *Ktor* [3]. Questa libreria offre un *framework* leggero e altamente configurabile per al creazione di servizi web e, più in generale, di applicazioni *backend*. Generalmente un servizio Ktor si compone di un insieme di moduli, ognuno dei quali solitamente ricopre ruoli specifici per la messa in funzione di un servizio web. Il *backend* per l'esposizione di API GraphQL comprende 2 moduli essenziali: un modulo di *routing* e un modulo per la configurazione di GraphQL. Il primo modulo ha il compito di esporre i tre *endpoint* principali su cui opera il server GraphQL:

- `/graphql`: *endpoint* dal quale arrivano le richieste GraphQL di tipo *query* e *mutation*. Le richieste devono essere effettuate tramite il protocollo HTTP, preferibilmente attraverso il metodo POST includendo nel payload l'operazione richiesta.

- `/sdl`: *endpoint* per l'introspezione del SDL, ovvero lo schema GraphQL esposto dal server, contenente il modello e le operazioni fornite.

[3] https://ktor.io/

- `/subscriptions`: *endpoint* speciale per l'esecuzione di operazioni del tipo *subscription*. A differenza degli altri, questo è interrogabile solo attraverso il protocollo *WebSocket*, affinché sia possibile mantenere una connessione persistente in modo efficace.

È all'interno di questo modulo che le richieste client arrivano e vengono approvate (i.e. verifica se le richieste sono ben formate). Successivamente queste vengono inviate al modulo responsabile dell'esecuzione delle operazioni GraphQL, il secondo modulo di cui si compone il web server. Il suo compito principale è quello di validare le richieste al di sopra dello schema, e in caso di esito positivo, provvedere al recupero dei dati richiesti e infine l'invio della risposta al client richiedente. All'interno di questo modulo risiede il cuore dell'architettura del server, le cui funzioni stanno alla base del successo dell'impiego delle API GraphQL, come esposto nella sezione seguente.

4.2.2 Esecuzione e Interrogazione delle API

Le sezioni precedenti hanno illustrato il processo di ideazione, design e implementazione del sistema software. In questa sezione verrà coperto il funzionamento generale dell'intera architettura, sia lato server, sia l'interrogazione da parte del client e l'interpretazione dei risultati ottenuti.

Server Procedendo attraverso un approccio *code-first*, il server prima di essere messo in funzione effettua un'operazione di generazione dello schema, grazie ad una componente software denominata `SchemaGenerator` implementata attraverso la libreria GraphQL Kotlin. Il compito di questa componente è di generare a *runtime* tutte le componenti del modello indicate all'interno del *build* file in modo tale che siano utilizzabili all'interno di un architettura GraphQL. Verrà quindi dapprima effettuato un controllo delle componenti sulle specifiche GraphQL, e se il controllo va a buon fine, vengono definiti i relativi *resolver* e *fetcher* per ogni componente elaborata. Il primo di questi ha il compito di interpretare le richieste GraphQL in arrivo che riguardano quel determinato elemento del dominio, effettuando quindi un *parsing* della richiesta stessa e dopo aver interpretato il suo contenuto ridireziona la richiesta al *fetcher*, il quale invece ha il compito di recu-

perare i dati richiesti dall'operazione all'interno del sistema, secondo la struttura definita all'interno della *query* inviatagli dal client. In caso invece la richiesta venga validata negativamente, un errore verrà spedito, in modo tale che sia possibile un'interpretazione dell'errore anche lato client.

Questo è quanto accade con operazioni del tipo *query* e *mutation*. Per tutte le operazioni di *subscription*, è necessario stabilire l'evento che scaturisce l'invio di un dato aggiornato. Come precedentemente anticipato in Sezione 3.2.1, mediante `EnvironmentSubscriptionMonitor` è possibile ottenere una versione dell'ambiente aggiornata ad ogni step che la simulazione esegue. Questo viene effettuato mediante una chiamata periodica da parte della simulazione ad ogni `OutputMonitor` ad essa collegato, del metodo `stepDone` , appunto ogni volta che uno step viene effettuato. In Figura 4.1 è illustrato un diagramma di sequenza dettagliato per ogni operazione di *subscription* effettuata dal client.

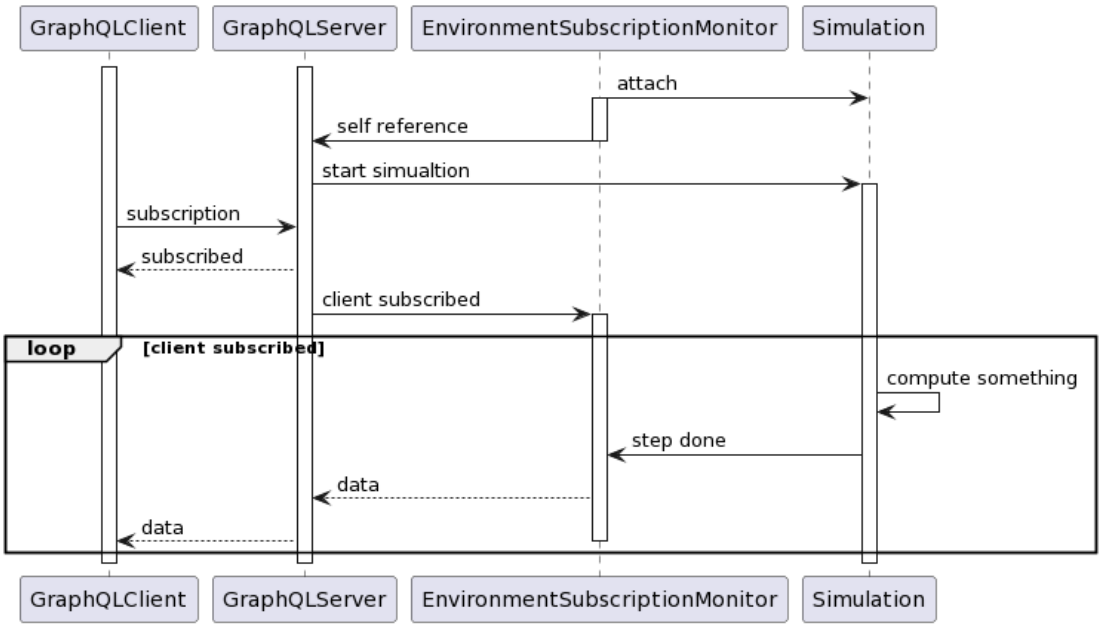


Figura 4.1: Diagramma di sequenza del funzionamento di una *subscription* mediante l'uso di `EnvironmentSubscriptionMonitor`

Prima della messa in funzione del server, l'istanza di `EnvironmentSubscrip` ⌋ `tionMonitor` viene collegata alla simulazione e preserva un riferimento all'interno

del server GraphQL. Una volta che la simulazione viene azionata, è possibile effettuare *subscription*. Queste ultime prevedono l'uso di strutture in grado di emettere una serie di valori in modo asincrono e duraturo nel tempo: i `Flow` Kotlin. La natura dei `Flow` è di essere strutture dette *cold*, ovvero che l'invio vero e proprio di valori avviene solamente nel momento in cui esso inizia ad essere collezionato. Una volta che esso viene collezionato, viene anche consumato, risultando in un impossibilità di condivisione dello stesso. Nel caso di un server GraphQL, il quale necessita di rispondere ad un numero esiguo di client con lo stesso tipo di informazioni, è necessaria una struttura che sia condivisibile cioè che non venga consumata nel momento in cui viene collezionata. All'interno di questo contesto si inseriscono i `MutableSharedFlow`, i quali estendono le funzionalità di un `Flow`; in particolare essi forniscono un supporto di condivisione del flusso di dati, senza che venga consumato dai collettori, ma essendo una struttura mutabile, l'invio è detto *hot*, cioè che il flusso di dati viene spedito nonostante ci sia qualche collettore dall'altra parte. Per quanto questa caratteristica sia in alcuni casi non voluta, poiché il server invierà sempre dati nel flusso anche se nessun client ha effettuato la sottoscrizione, essa risulta utile per tutti quei casi dove si può manifestare *backpressure*, ovvero che il client non riesce a stare al passo con i dati inviati dal server. In questo contesto infatti, siccome gli elementi nel flusso vengono inviati sempre e in modo costante, il server non verrà bloccato da client che richiedono molto tempo per l'elaborazione dei dati stessi, lasciando il compito di gestire il *backpressure* lato client, come descritto in Sezione 5.1.

Client Come già illustrato, il funzionamento del client è in linea con quanto riassunto all'interno dello schema UML in Figura 3.5. Operazioni preliminare è lo *schema introspection*, eseguibile solo una volta che il server è in esecuzione e in ascolto su un determinato host e porta. Successivamente è possibile scrivere e inserire all'interno del package *commonMain* operazioni GraphQL in base a ciò che espone il server grazie allo schema, potendo validare sullo stesso a tempo di compilazione, anche la struttura e correttezza delle operazioni scritte. Queste operazioni sono all'interno di file con estensione "`.graphql`", il quale è scritto attraverso il *Query Language* di GraphQL in grado di stabilire lato client la struttura desiderata dei dati che saranno spediti richiamando una specifica operazione del

server. In Elenco 6 sono rappresentati (per fini illustrativi all'interno dello stesso file), una parte della dichiarazione dello schema di un elemento del dominio e le possibili operazioni di tipo `Query`, e una *query* lato client. All'interno dello schema viene definita l'operazione `nodeById(nodeId: Int!)` di tipo *query*, la quale dato un `id` ritorna un oggetto di tipo `NodeSurrogate` corrispondente. All'infuori dello schema viene definita una *query* lato client in grado di richiamare l'operazione esposta dal server, la quale accetta in input un intero rappresentante l'`id` del nodo, e ritorna solo alcuni dei campi dell'oggetto `NodeSurrogate`. Questo è il grande vantaggio nell'utilizzo di GraphQL, poiché si cerca di limitare al massimo il numero di informazioni non necessarie che transitano in rete da client a server, evitando il fenomeno dell'*over-fetching*.

Una volta che lo schema e le operazioni GraphQL sono all'interno delle risorse di *commonMain*, è possibile generare i sorgenti derivanti dallo schema e dalle operazioni definite, attraverso Apollo Kotlin[4]. In questo modo è possibile già a tempo di compilazione avere un *type checking* all'interno del codice Kotlin scritto lato client che faccia uso delle operazioni definite. Infine, i sorgenti generati saranno utilizzati per poter far effettuare ad Apollo Kotlin, a runtime, il mapping tra oggetto definito nello schema e in arrivo dopo aver chiamato un'API, e il relativo oggetto generato in fase di compilazione.

All'interno del client quindi, il lavoro di instaurazione delle comunicazioni, *parsing* delle operazioni e successiva validazione e interpretazione del risultato ottenuto dopo l'esecuzione di un'operazione GraphQL, è completamente delegato alla libreria Apollo Kotlin, lasciando il programmatore ad un alto livello di astrazione.

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

```
# Schema.graphqls
"A node in the environment"
type NodeSurrogate {
  contains(molecule: MoleculeInput!): Boolean!
  contents: MoleculeToConcentrationMap!
  getConcentration(molecule: MoleculeInput!): String
  id: Int!
  moleculeCount: Int!
  properties: [String!]!
  reactions: [ReactionSurrogate!]!
}

type Query {
    "Returns a node with the given id"
    nodeById(nodeId: Int!): NodeSurrogate!
}

#-------------------------------------------------
# NodeQuery.graphql
query NodeQuery($nodeId: Int!) {
    nodeById(nodeId: $nodeId) {
        id
        moleculeCount
        properties
    }
}
```

Listato 6: Definizione di `NodeSurrogate` all'interno dello schema GraphQL e relative definizioni lato server e client della *query* che ne fa uso.

4.3 Verifica

Per la verifica del software sono stati effettuati Unit test per tutti gli elementi surrogati del modello. In particolare, i test hanno coperto la correttezza dell'effettiva produzione di oggetti surrogati, e successivamente sono stati valutati comportamenti di tali oggetti in relazione all'oggetto Alchemist incapsulato. Questo è stato fatto utilizzando un insieme di simulazioni di test, le quali producono un determinato modello di Alchemist. A partire da questo modello, vengono generati i relativi surrogati attraverso le *extension function*. A questo punto sono state comparate le caratteristiche del modello surrogato e quello originario in fase iniziale, e in seguito a modifiche all'interno dell'ambiente.

Per quanto riguarda le operazioni GraphQL, si è scelto un approccio meno automatizzato, poiché le librerie per l'esecuzione di un client e server GraphQL seppur disponendo di framework di testing, richiedono molto lavoro ulteriore per la creazione di *Mock Server* per simulare la risposta o l'esecuzione di operazioni GraphQL. Le tempistiche di progetto hanno portato l'implementazione di queste funzionalità aggiuntive di testing per lavori futuri. L'approccio che si è utilizzato per il testing di queste operazioni è effettuato attraverso l'impiego del *playground* GraphiQL[5], il quale permette di collegarsi al server GraphQL in esecuzione ed effettuare *introspection* dello schema, per poter consultare tutta la struttura del modello e le operazioni fornite. In un secondo momento è possibile effettuare operazioni GraphQL lato client richiamando *query*, *mutation* e *subscription* esposte dal server, specificandone quindi l'insieme di campi che si desiderano ricevere. Oltre al testing effettivo delle operazioni, GraphiQL risulta molto utile in fase di sviluppo delle operazioni client. È buona norma infatti definire all'interno del *playground* un'operazione client, verificarne l'effettivo risultato che si ottiene una volta eseguita ed esportare l'operazione all'interno delle risorse client del progetto. Seguendo questo approccio, ogni operazione aggiunta all'insieme delle operazioni client è già in qualche modo validata sullo schema da parte dello sviluppatore, fornendo inoltre una visione complessiva del risultato ottenuto richiamando quell'operazione.

[5]https://github.com/graphql/graphiql

Capitolo 5

Conclusioni

L'insieme di quanto discusso all'interno di questo elaborato hanno portato alla costruzione di un sistema software in grado di fornire meccanismi efficienti per l'accesso e il controllo di una simulazione all'interno di Alchemist. Questo è stato reso possibile grazie ad un attento studio delle architetture per lo sviluppo e interrogazione di API, prediligendo un approccio fortemente strutturato grazie a GraphQL. Il suo impiego ha reso necessaria un'analisi approfondita del dominio applicativo di Alchemist, grazie al quale sono state definite linee guida per la loro rappresentazione all'interno di uno schema conforme alle specifiche GraphQL.

Allo stato attuale, il progetto mira a fornire una solida componente server in grado di effettuare operazioni basilari all'interno di una simulazione, la quale mira a facilitare l'aggiunta di ulteriori operazioni e funzionalità, riducendo al minimo le modifiche all'architettura esistente. Alcune di queste funzionalità aggiuntive sono descritte all'interno della Sezione 5.1.

Come dimostrazione del funzionamento dell'architettura software prodotta, viene presentata una simulazione presa dal tutorial di Alchemist[1], al cui interno è presenta una griglia di nodi in grado di giocare a *dodgeball*. Il funzionamento è piuttosto semplice: il programma inizia la simulazione con una palla, e l'obiettivo è quello di lanciarla ad un vicino in modo casuale. Ogni nodo che viene colpito, acquisisce un punto, incrementa il proprio punteggio e lancia nuovamente la palla.

L'applicativo mostrato in Figura 5.1 mostra l'interfaccia all'interno del web

[1] https://alchemistsimulator.github.io/tutorials/basics/index.html

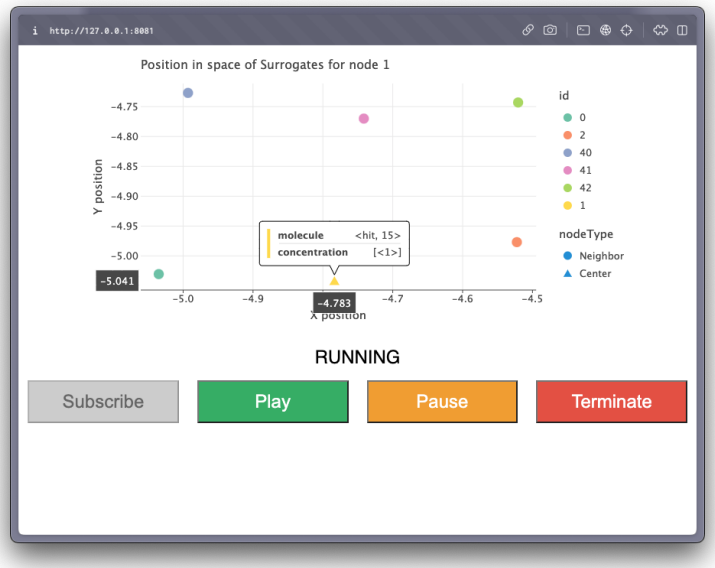


Figura 5.1: Applicativo client per la visualizzazione di statistiche riguardo la simulazione *Dodgeball*

browser composta da un pannello di controllo contenente 4 bottoni e uno spazio adibito ad un Grafico che mostri delle caratteristiche della simulazione corrente. Il grafico è generato attraverso l'uso della libreria Lets-Plot Kotlin[2], la quale è stata approfondita estensivamente all'interno dell'attività di tirocinio curricolare[3]. Di seguito una breve descrizione delle funzionalità proposte: al caricamento della pagina, è possibile sottoscriversi in un qualsiasi momento tramite il pulsante *subscribe*, oppure avviare la simulazione mediante *play*. Una volta che la simulazione sarà fatta partire e ci si sottoscrive, all'interno del grafico verranno visualizzate le posizioni di un vicinato, in questo caso del vicinato al quale appartiene il nodo con `id` pari a 1. La simulazione non prevede cambiamenti di posizioni durante l'esecuzione, ma muovendo il cursore al di sopra di un nodo è possibile monitorare lo stato della molecola e concentrazione ad esso associate, dove nella prima è contenuto il numero di *hit* effettuati fino a quel momento. Infine, i restanti due pulsanti hanno rispettivamente la funzione di mettere in pausa e terminare definitivamente

[2]https://github.com/JetBrains/lets-plot-kotlin
[3]https://s-furi.github.io/uni-internship

la simulazione. In caso di errori durante la simulazione, come per esempio la richiesta di avviamento della simulazione dopo che è stata fatta terminare, verrano visualizzati a schermo all'utente.

5.1 Sviluppi Futuri

L'obiettivo di questo elaborato è stato quello di dimostrare la fattibilità ed implementare un'architettura software per l'esposizione di API al di sopra del simulatore Alchemist, provvedendo a garantire meccanismi di estensibilità sia da un punto di vista delle operazioni eseguibili, sia per quanto riguarda il modello di Alchemist. Attraverso questo sistema software così costruito, è possibile operare su di esso per effettuare miglioramenti ed estensioni delle funzionalità, riassunte nei seguenti punti:

- **N+1 *problem***: nei sistemi GraphQL si può verificare durante l'esecuzione di una *query* che questa richieda una risorsa, e per ogni record restituito, viene effettuata un'ulteriore query per recuperare informazioni correlate, portando a un numero esponenziale di chiamate al sottosistema per il recupero dei dati, provocando per una operazione, ulteriori N query, causando ciò che viene appunto definito $N + 1$ *problem*. All'interno di GraphQL Kotlin è possibile mitigare questa problematica mediante appositi *data loader*, i quali permettono di effettuare *batching di operazioni* e offrire inoltre un livello di *caching* per le operazioni più frequenti. Questo può quindi comportare un ampio beneficio in termini di prestazioni generali del sottosistema, sia in termini di latenza di risposte, sia per quanto riguarda l'*overhead* generale nella risoluzione delle operazioni.
- **Gestione del *backpressure***: nei casi in cui il tasso di invio delle informazioni dal server al client sia maggiore di quanto il client stesso riesca a gestire, è necessario fornire quest'ultimo di strategie in grado di gestire il fenomeno del *backpressure*. Questo può avvenire mediante politiche ad-hoc al momento della ricezione del `Flow`, come per esempio la scelta dell'elemento più recente all'interno del canale comunicativo, oppure prenderne solo alcuni campioni. La strategia adottata dipenderà da sistema a sistema.

- **Miglioramento strategie di Serializzazione**: per come è stato implementata la logica di serializzazione di una concentrazione, essa funziona con i tipi primitivi di Kotlin e le classi annotate attraverso l'annotazione `@Serializable`. Per quanto questa soluzione sia efficace nei casi più semplici, esiste un certo numero di situazioni dove questo approccio risulti non viabile per via di alcuni fattori, come per esempio la presenza di ereditarietà tra gli elementi (necessitando di un `PolymorphicSerializer`) o anche l'impiego di istanze specifiche delle *collection* di Kotlin (e.g. `ArrayList` non è serializzabile direttamente ma `List` lo è). All'interno di questo contesto è necessaria la valutazione di strategie più avanzate di serializzazione, le quali non hanno trovato il tempo di essere approfondite durante l'elaborazione di questo progetto.
- **Ampliamento del modello**: allo stato attuale, l'architettura GraphQL implementata comprende una semplice sottoparte dell'intero modello di Alchemist. L'aggiunta di ulteriori elementi come *linking rule*, *action* e *reaction* renderanno il sistema in grado di rappresentare l'intero meta-modello di Alchemist con la corretta interpretazione in termini di specifiche GraphQL.

Bibliografia

[ACPV22] Gianluca Aguzzi, Roberto Casadei, Danilo Pianini, and Mirko Viroli. Dynamic decentralization domains for the internet of things. *IEEE Internet Computing*, 26(6):16–23, 2022.

[AR21] Tobias Andersson and Håkan Reinholdsson. *REST API vs GraphQL: A literature and experimental study*. PhD thesis, 2021.

[BC86] Jerry Banks and John S. Carson. Introduction to discrete-event simulation. In *Proceedings of the 18th conference on Winter simulation - WSC '86*, WSC '86. ACM Press, 1986.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles - SOSP '87*, pages 123–138. ACM Press, 1987.

[CVAP] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala dsl and toolkit for aggregate programming. 20:101248.

[ERRJ] Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 107–116, 139–150, 293–303. Addison-Wesley.

[FM11] I. Fette and A. Melnikov. The WebSocket protocol. Technical report, December 2011.

[FT00] Roy Thomas Fielding and Richard N. Taylor. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, 2000. AAI9980887.

[PMVa] Danilo Pianini, Sara Montagna, and Mirko Viroli. A chemical inspired simulation framework for pervasive services ecosystems. In *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 667–674.

[PMVb] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. 7(3):202–215.

[PPCE22] Danilo Pianini, Federico Pettinari, Roberto Casadei, and Lukas Esterle. A collective adaptive approach to decentralised k-coverage in multi-robot systems. *ACM Trans. Auton. Adapt. Syst.*, 17(1–2), sep 2022.

[PVB] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: Practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1846–1853. Association for Computing Machinery.