Designing and implementing an observability module for TuSoW enabling the implementation of a Web-based monitoring (graphical) interface

Chiara Forresi
chiara.forresi@studio.unibo.it

Giacomo Montalti
giacomo.montalti3@studio.unibo.it

January 2019

Il progetto si propone l'obiettivo di fornire un modulo aggiuntivo per TuSoW in grado di interfacciarsi con esso, avente il compito specifico di implementare un servizio di monitoraggio attraverso un'interfaccia grafica. Fornirà la possibilità di dialogare con TuSoW tramite le primitive già implementate, operando all'interno di spazi di tuple, sarà presente la possibilità di osservare il flusso di richieste ed eventi e il loro sviluppo all'interno del webServer. Le funzionalità descritte verranno sviluppate tramite JavaScript, che gestirà l'invio di richieste direttamente al web service. Inoltre verrà adottato il pattern publish/subscribe (senza un'autenticazione) per poter dare la possibilità di ispezionare ciò che avviene all'interno degli spazi di tuple.

Contents

1 Goal/Requirements

1. Sviluppare una GUI chiara e semplice in grado di fornire tutte le informazioni richieste.
2. Definizione di una ReST API per l'osservabilità.
3. Fornire un servizio per il monitoraggio in grado di reagire in autonomia e tempi ragionevoli.
4. Fornire dei metodi rapidi e intuitivi per eseguire le principali operazioni sugli spazi di tuple.
5. Utilizzare in maniera corretta le interfacce fornite e instradare correttamente le richieste provenienti dall'esterno.
6. Usare correttamente i Test per sviluppare in maniera adeguata le funzionalità previste.

In Figura 1 viene mostrato il *diagramma dei casi d'uso* che mostra l'interazione tra Servizio Web e Server con Rest API.

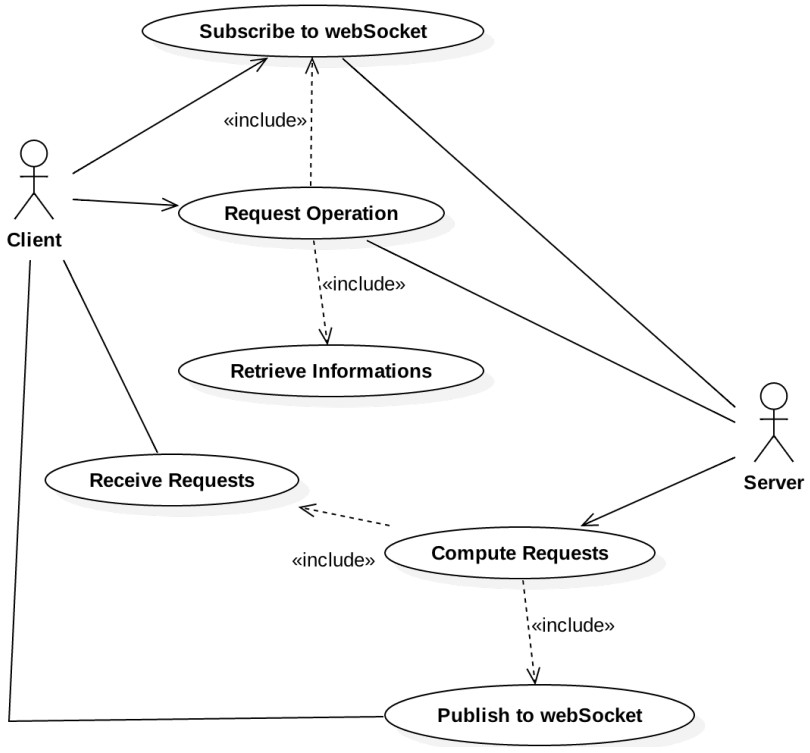

Figure 1: Diagramma dei casi d'uso.

1.1 Scenarios

L'utente si potrà collegare al servizio TuSoW in esecuzione attraverso un'interfaccia web, la quale interagirà con il sistema come mostrato in Figura 1.

Verrà sviluppata una libreria avente il compito specifico di relazionarsi con il servizio TuSoW e contenente tutte le dipendenze necessarie all'interfacciamento con esso, dovrà permettere all'utente di interagire con qualunque spazio di tuple logico, fornendo gli strumenti adatti a poter inviare qualunque tipo di richiesta possibile al servizio TuSoW. Infine, la libreria sviluppata dovrà dare accesso al flusso degli eventi derivanti dalle variazioni in atto nello spazio di tuple. Tramite l'utilizzo di una *WebSocket* le modifiche verranno visualizzate graficamente in real-time e non sarà necessario dover aggiornare manualmente la pagina.

1.2 Self-assessment policy

Il sistema verrà auto-valutato sia per quanto riguarda la *qualità* che l'*efficacia* dello stesso come segue.

- La *qualità* del *software prodotto* verrà valutata attraverso un approccio Test driven.
- L'*efficacia* dei risultati del progetto verrà valutata attraverso il superamento dei Test sviluppati, del funzionamento dell'interazioni tra le varie componenti software e nella capacità di generalizzazione del software in spazi di tuple non logici (stringhe).

2 Requirements Analysis

Il software che verrà sviluppato sarà un'estensione di TuSoW[1], per cui dovrà integrarsi con il progetto e rispettare la semantica delle primitive dello spazio di tuple. Esso si preoccuperà di fornire un'**interfaccia per l'ispezionabilità** nello spazio di tuple *logico*: se le modifiche apportate riusciranno ad *astrarre i concetti* base nel migliore modo possibile, sarà possibile in un secondo momento procedere con l'integrazione di queste anche all'interno di altre tipologie di spazi di tuple, come ad esempio quello basato su stringhe.

Il Server per poter fornire il servizio d'ispezionabilità al Client dovrà dotarsi di un **meccanismo di Publish/Subscribe** che permetta all'utente, attraverso il servizio web, di ricevere i messaggi provenienti dal Server relativi agli eventi che avvengono nello spazio di tuple e di reagire a questi nel modo desiderato.
La tecnologia che meglio rappresenta questo tipo di interfaccia in questo contesto specifico è la *WebSocket*.

Per rendere il tutto il più modulare e manutenibile possibile, verranno incapsulati tutti i metodi per l'interfacciamento con TuSoW all'interno di una libreria da utilizzare in ambito web. Questo permetterà, in un secondo momento, la possibilità di riprogettare una

[1] `https://app.swaggerhub.com/apis/lrizzato/TuSoW/1.0.0`

nuova interfaccia grafica senza dover riprogettare o migrare i metodi utilizzati all'interno di questo progetto.

3 Design

Analizzeremo il comportamento del sistema sotto tre dimensioni principali. In primis andremo ad analizzare quali sono i concetti e le entità rappresentate, questo è lo scopo della sottosezione 3.1. Secondariamente, nella sottosezione 3.2, si andrà ad analizzare come queste interagiscono. Infine, nella sottosezione 3.3, si andrà ad analizzare il comportamento delle singole entità.

3.1 Structure

All'interno di questa sezione verranno descritte le entità sviluppate per risolvere i problemi principali del progetto. In primo luogo verranno descritte le modifiche apportate al servizio TuSoW originario, successivamente verrà presentata l'interfaccia lato Client utilizzabile per implementare un applicativo web atto a monitorare e interagire con il nuovo servizio Server descritto nella sezione precedente.

3.1.1 Server

Avendo il progetto come obiettivo quello di integrare un modulo per l'ispezionabilità e, quindi, riuscire a intercettare qualsiasi evento che arrivi nello spazio di tuple, l'idea è stata quella di partire dall'implementazione esistente di spazio di tuple ed estenderla a un concetto di spazio di tuple ispezionabile. Si è deciso di prendere questa strada poiché integrare il concetto di ispezionabilità può essere considerato come integrare al concetto di spazio di tuple una serie di funzionalità che consentano di catturare cambiamenti all'interno dello stesso. Questi cambiamenti possono essere considerati come degli eventi per cui, come verrà descritto in seguito, si è dovuto creare un concetto di "evento" e di "ascoltatore di evento". In seguito verrà descritto a livello tecnico quanto detto.

Il diagramma delle classi in Figura 2 mostra i concetti di classi esistenti e come ad esse viene integrato il concetto di ispezionabilità. In sostanza la nostra implementazione dello spazio di tuple (`DeterministicLogicSpaceImpl`) è un'estensione di `InspectableLogicTupleSpace` che comprende tutte le operazioni che si possono fare in: `PredicativeTupleSpace`, `NegatedTupleSpace` e `BulkTupleSpace`.
`InspectableTupleSpace` non è altro che l'estensione di `TupleSpace` con tre metodi fondamentali:

- `operationInvoked`: restituisce un *EventSource* collegato alle operazioni invocate;
- `operationCompleted`: restituisce un *EventSource* collegato alle operazioni completate;
- `tupleSpaceChanged`: restituisce un *EventSource* collegato alle operazioni che provocano cambiamenti nello spazio di tuple.

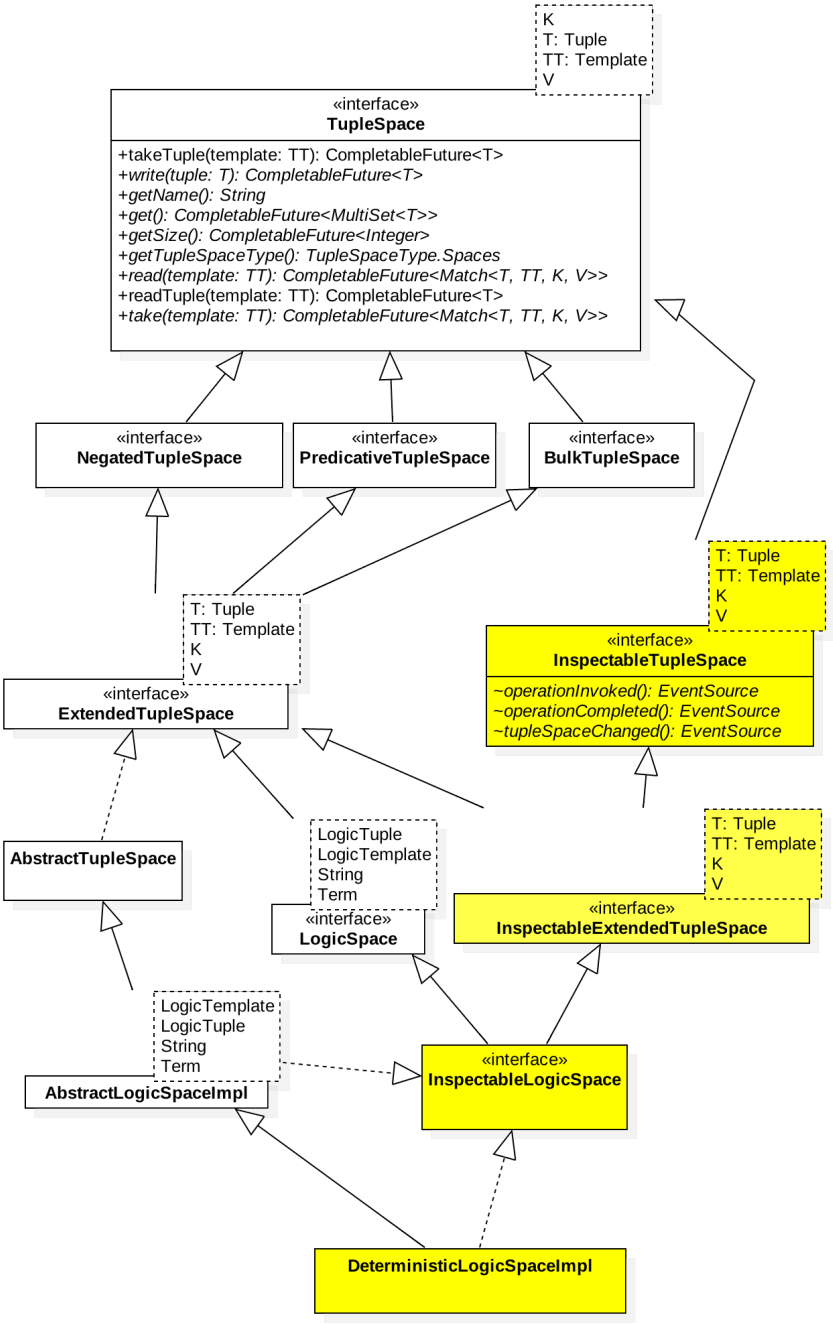

Figure 2: Diagramma delle classi che mostra come è stato sviluppato il concetto di spazio di tuple ispezionabile.

Il diagramma in Figura 3 mostra la struttura dell'interfaccia **EventSource**. Attraverso la gerarchia mostrata viene replicato il concetto di *evento* in Java. `EventSource` consente

di far legare (*bind*) o slegare (*unbind*) un `EventListener` che è un'*interfaccia funzionale* che effettua un'azione in corrispondenza di uno specifico evento.

Gli eventi modellati, come mostrato nel diagramma delle classi in Figura 4, sono di due tipi:

- `TupleEvent`: eventi relativi ai cambiamenti dello spazio di tuple;
- `OperationEvent`: eventi relativi a operazioni nello spazio di tuple, ovvero *invocazione* o *completamento* di una richiesta.

Queste due tipologie di eventi, venongo mappate alle tre operazioni sopracitate che è possibile svolgere in un `InspectableTupleSpace`.

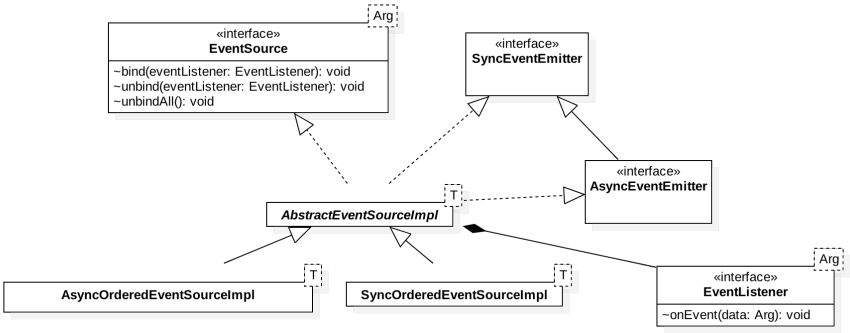

Figure 3: Diagramma delle classi relativo all'interfaccia EventSource.

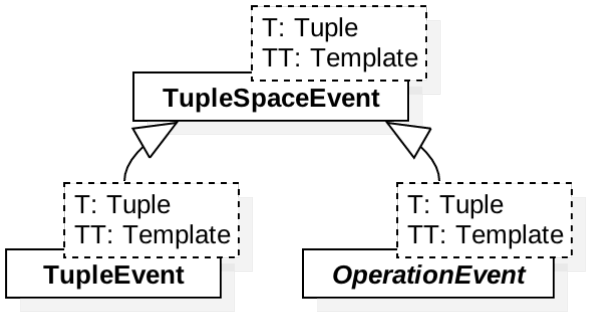

Figure 4: Diagramma delle classi relativo alle tipologie di eventi presenti.

Per quanto riguarda lo sviluppo del Servizio REST, viene esteso il concetto di *io.vertx.core*[2] `AbstractVerticle` con un concetto astratto di `Service`. La specializzazione di quest'ultimo, `InspectableLogicService`, usa il concetto di spazio di tuple logico ispezionabile.

[2] `https://vertx.io`

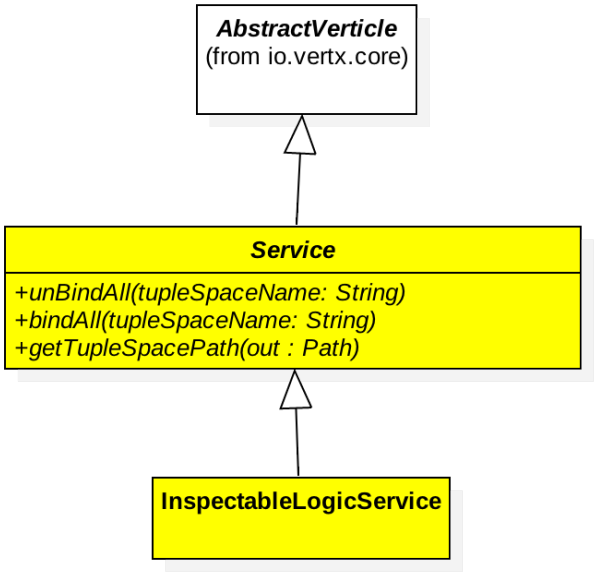

Figure 5: Diagramma delle classi relativo all'implementazione del Servizio REST.

Per quanto concerne la modellazione delle classi viene mantenuta una struttura *multi-project* che fornisce un'adeguata distinzione tra generalizzazione e specializzazione di quanto sviluppato. Il diagramma dei package in Figura 6 mostra tale struttura.

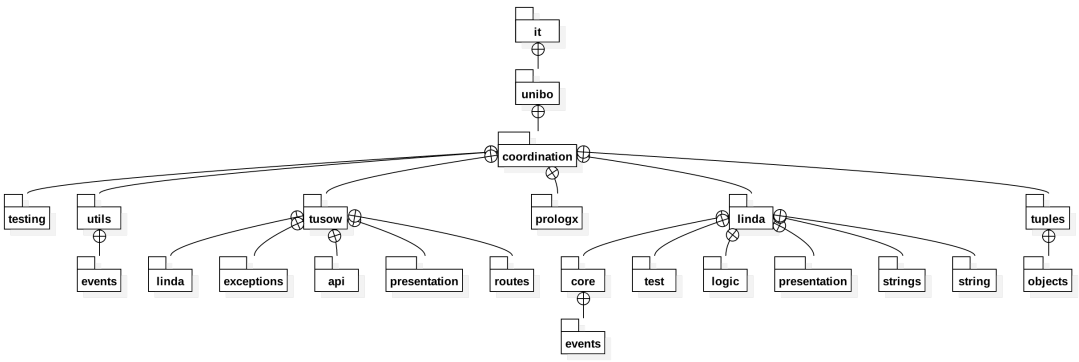

Figure 6: Diagramma dei Package.

3.1.2 Client

Dopo aver visto la parte relativa al servizio TuSoW, all'interno di questa sezione viene approfondita la parte di progetto relativa al Client. In particolare, vengono descritte nel

dettaglio l'interfaccia grafica e la libreria JavaScript prodotte.
Durante la fase di analisi è stata intrapresa la scelta di suddividere i compiti tra due entità separate:

- tutto ciò che riguarda la **view** e meri aspetti di grafica e visualizzazione sono stati inseriti all'interno di una pagina web;
- la **logica** per connettersi al servizio ispezionabilità di TuSoW è stato inserito all'interno di una libreria sviluppata.

Tale approccio ha comportato una quantità di lavoro maggiore, poiché oltre allo sviluppo di entrambe le entità si è reso necessario lavorare sulla comunicazione tra esse. Questa architettura ha di riflesso una serie di vantaggi importanti:

- separare i compiti permette di concentrarsi meglio sul singolo componente, scrivendo codice più pulito, ordinato e strutturato in modo indipendente dall'altro;
- favorisce la manutenibilità del progetto;
- l'aggiunta di funzionalità o la modifica di alcuni aspetti all'interno di un componente non intaccano per forza l'altro.

Si è sviluppata prima l'interfaccia `TusowInspectInterface` per l'interfacciamento con TuSoW e successivamente la sua specializzazione nella classe `TusowJS` in grado di adempiere alle specifiche fornite.
La Figura 7 rappresenta il diagramma delle classi relative a tale aspetto.

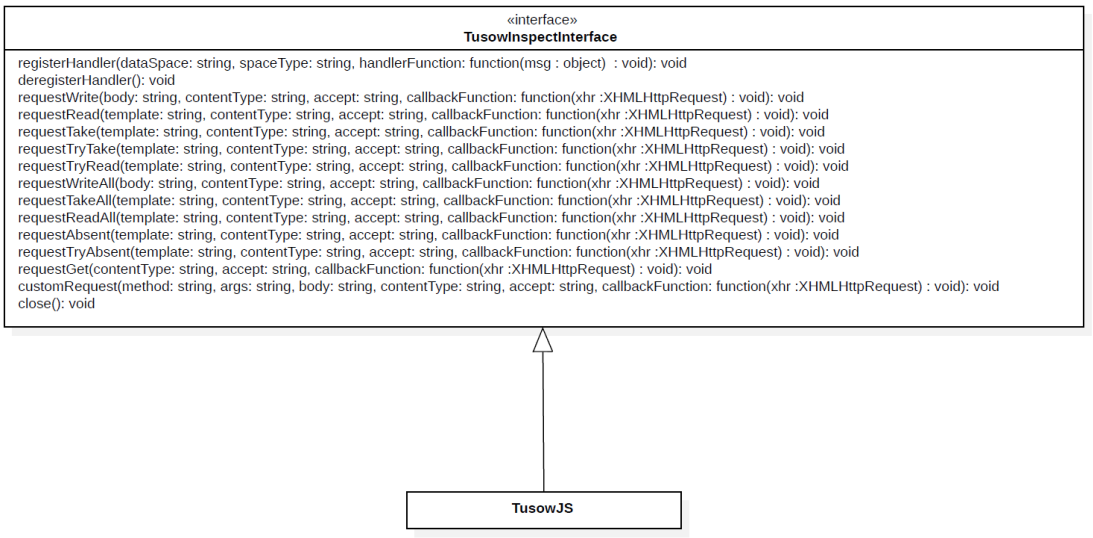

Figure 7: Diagramma delle classi che mostra l'interfaccia della libreria JavaScript per l'ispezionabilità e la classe che la specializza.

3.2 Behaviour

Il comportamento delle entità presenti nel progetto nasce dalla cooperazione tra quanto sviluppato lato Server con l'`InspectableLogicService`, tutti i concetti di `linda-logic` da cui esso dipende e le funzionalità presenti nella libreria sviluppata.
Come è già stato detto, le entità di questa architettura sono due: Client e Server. Se quest'ultima è rappresentata da TuSoW, la prima invece **non è rappresentata completamente** dalla libreria sviluppata e descritta nella sezione 3.1.2, in quanto con la sola libreria risulta impossibile a un ipotetico Client poter inviare richieste al Server. Per poter utilizzare la libreria è stata infatti sviluppata una semplice interfaccia grafica in grado di utilizzare nel modo corretto i metodi resi disponibili da `TusowJS`.
Scendendo più nel dettaglio del comportamento dei singoli componenti, la pagina web sviluppata si occuperà di raccogliere tutti i parametri necessari in base alla richiesta, mentre utilizzerà un'istanza di `TusowJS` per dialogare con il Server.
Il diagramma delle attività in Figura 8, mostra come l'interfaccia grafica si relaziona al Servizio REST per inviare una richiesta e il comportamento che esso ha.

3.3 Interaction

L'interazione tra il Servizio REST e la libreria indica il comportamento del sistema a regime. Il diagramma di sequenza in Figura 9 mostra come il Client apre una WebSocket e si interfaccia al Servizio REST **tramite un'istanza di TusowJS**, da questo momento in poi TuSoW inizierà ad inviare la sequenza di eventi che si susseguono nello spazio di tuple a cui esso afferisce.
Nel dettaglio l'apertura della WebSocket e la configurazione dei parametri iniziali per interfacciarsi con TuSoW vengono richiesti dall'interfaccia grafica nel momento in cui viene instaurata una connessione con uno spazio di tuple. L'invio di una richiesta dal Client comprensivo del settaggio di tutti i suoi parametri e della ricezione di una risposta dal Server è gestito dalla stessa interfaccia grafica nel momento in cui l'utente effettua una richiesta. Invece, la chiusura e la conseguente distruzione della linea di vita avviene quando il Client esce dal servizio.

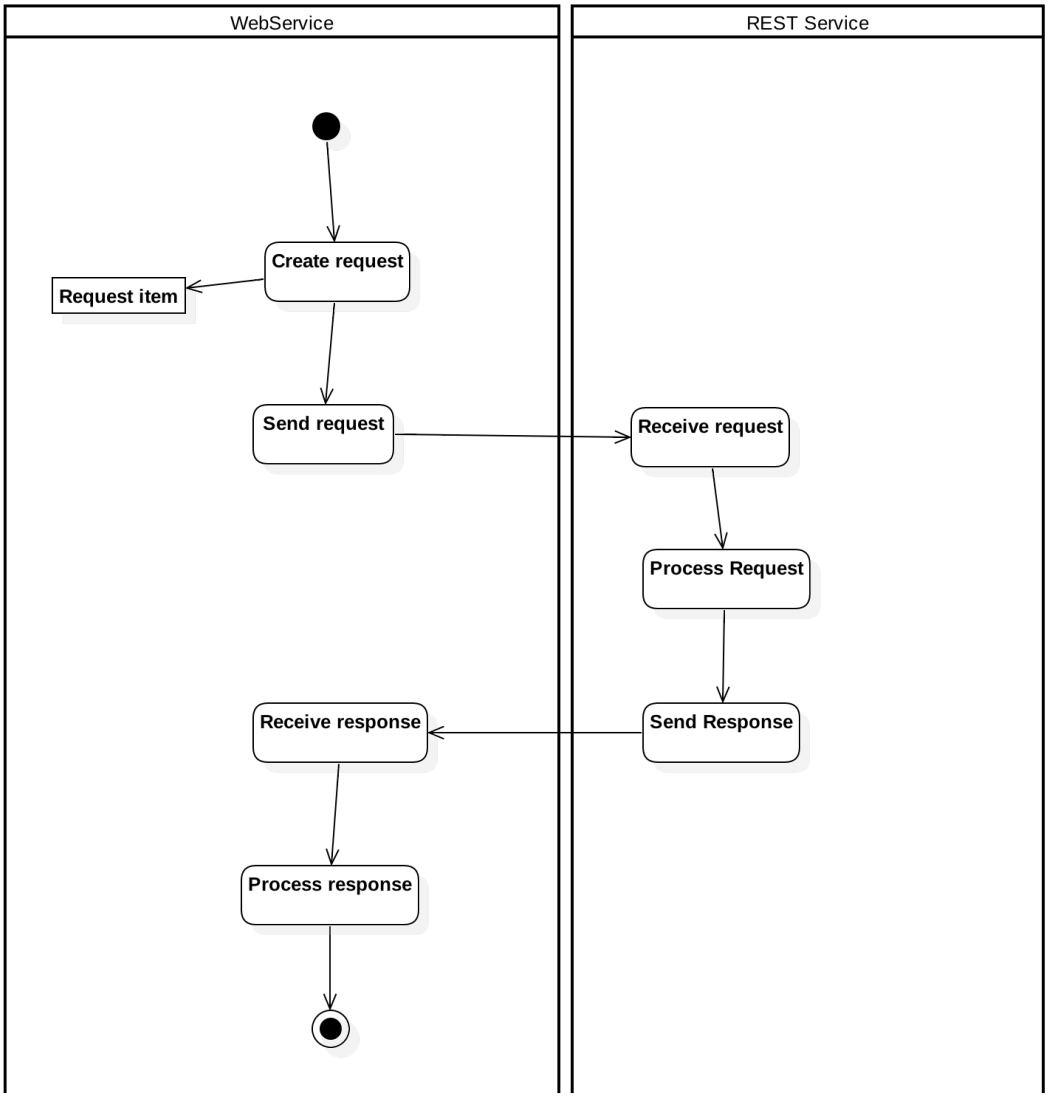

Figure 8: Diagramma delle attività che mostra come il Web Service invia una richiesta e come questa viene elaborata dal Servizio REST.

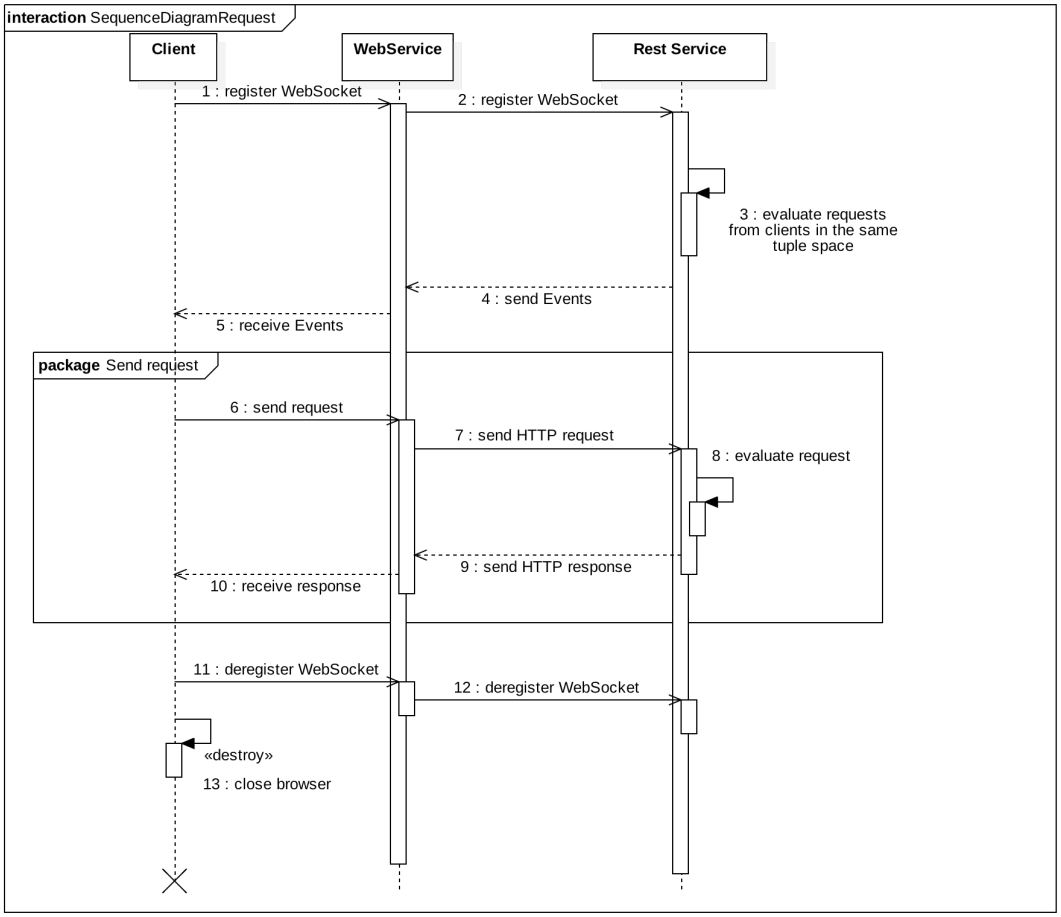

Figure 9: Diagramma di sequenza che mostra l'interazione tra Client, Web Service e REST Service.

4 Implementation Details

All'interno di questa sezione sono stati raccolti i **punti di scelta** incontrati durante lo sviluppo del progetto. I dettagli implementativi più significativi sono descritti di seguito.

4.1 Interfacciarsi al Servizio REST

Per interfacciarsi al servizio REST sono state redatte le seguenti specifiche swagger [3]. Si noti che, a differenza del progetto iniziale fornitoci, il parametro *template* nelle *GET* e nelle *DELETE* viene trattato come *parametro* e non come il campo *body* della richiesta:

[3] `https://app.swaggerhub.com/apis/chiara.forresi/TuSoW2/1.0.0`

questa modifica si è resa necessaria poichè per queste tipologie di richieste, AJAX non consente di utilizzare un campo body[4].

4.2 Comunicazione tra Servizio REST e Client JavaScript

Per implementare il concetto di comunicazione tra Client e Server, dunque la WebSocket, si è scelto di utilizzare una libreria in grado di fornire questa funzionalità: il componente *EventBus* disponibile all'interno della libreria *vertx*[5].
In primo luogo questo ci ha permesso di concentrarci maggiormente su altri aspetti del progetto, risparmiando tempo sull'implementazione di tale feature.
Abbiamo scelto di attuare un *commitment tecnologico* spinti sia dall'ampia popolarità di vertx, ma soprattutto perché parte del codice lato Server si basa proprio su questa famiglia di librerie. Questa scelta è avvenuta con la consapevolezza che ciò significa essere *dipendenti* dalla libreria e dalle scelte che verranno intraprese dagli sviluppatori di quest'ultima.

Le informazioni che viaggiano nella *WebSocket* sono stringhe informato JSON, in modo da rendere facile la loro deserializzazione in oggetti JSON in JavaScript.

4.3 Ricostruzione dello spazio di tuple

Per ricostruire lo stato di uno spazio di tuple **iniziale** è necessario che il Client Web effettui una *"get"* con parametro *Accept* di tipo JSON: senza questa operazione l'utente non sarà mai in grado di conoscere lo stato dello spazio di tuple prima della sua sottoscrizione allo stesso.
Una volta ricostruito lo spazio iniziale lo spazio di tuple viene ricostruito partendo dagli eventi che la WebSocket del servizio web riceve dal Server.
Ogni qual volta viene ricevuto un `OperationEvent` di completamento di una richiesta se essa è una *"out"* o *"out_all"* vengono aggiunte le relative tuple risultanti allo spazio di tuple, altrimenti se si tratta di una *"in"*, *"inp"* o *"take_all"* vengono rimosse dallo spazio di tuple le corrispondenti tuple risultanti. Nell'implementazione sviluppata la rimozione di una tupla avviene utilizzando la libreria JavaScript `Lodash` [6] che confronta in maniera *deep* oggetti JSON. Inizialmente si è cercato di sviluppare tale funzionalità manualmente, ma ci sono stati problemi dovuti alla tipizzazione che in JavaScript è completamente assente e non ci ha consentito di poter distinguere un oggetto JSON da un array nello sviluppo di un algoritmo ricorsivo di confronto tra due alberi.

Questa parte non è stata implementata all'interno della libreria poiché, ritenendo che i modi per ricostruire uno spazio di tuple possono essere molteplici, si è delegato questo aspetto al futuro utilizzatore di `TusowJS` in modo da non vincolarlo con la nostra scelta.

[4] `http://api.jquery.com/jquery.ajax/`
[5] `https://vertx.io/docs/vertx-core/java/#event_bus`
[6] `https://lodash.com/docs/4.17.11#isEqual`

4.4 Sviluppo libreria TusowJS

Per gestire l'insieme delle dipendenze necessarie alla redazione della libreria Client del modulo software sviluppato ci si è affidati a **Node.js** e al package manager **npm**, contenente un numero vastissimo di oggetti e librerie già pronte da usare come dipendenze della package prodotto. La libreria prodotta ha il nome di *TusowJS*. Se in un secondo momento sarà necessario ampliare la libreria `TusowJS` con ulteriori compiti e funzionalità, sarà estremamente semplice aggiungere dipendenze e gestire il *versioning* delle tecnologie utilizzate.

Node.js consente l'esecuzione di codice JavaScript lato Server e nativamente questo non ne permette l'esecuzione lato Client. A tal proposito è stato usato il tool *browserify* [7] per *impacchettare* tutte le dipendenze e le estensioni richieste al funzionamento della libreria, in modo da avere un file `.js` utilizzabile anche dai vari browser. Per poter creare agilmente questo file, è stato settato l'alias `npm run bundlify` che ne permette un utilizzo molto più immediato anche in ottica di automazione e *continuous integration*.

Per astrarre meglio i concetti e per aggiungere l'aspetto *di compilazione* più rigido rispetto a quello (non) presente nativamente in JavaScript, si è scelto di utilizzare il *superset* **Typescript**. Il codice in Typescript per essere usato lato Client viene compilato lato Server producendo un file JavaScript. La compilazione viene eseguita seguendo una specifica *ECMAScript*, nel nostro caso la `ECMA-262`, versione 5.1. A tal proposito è stato creato un comando eseguibile da linea di comando (`npm run build`) per eseguire la compilazione del file Typescript in codice eseguibile JavaScript.

Per migliorare la qualità del software prodotto, ogni tipologia di richiesta è stata associata a un elemento di classe `Requests`: questo permette di avere a disposizione in maniera statica ogni tipo di richiesta in qualunque parte del codice, oltre a poter associare a ogni richiesta una serie di parametri e campi al pari di un'enumerazione complessa, non presente in JavaScript nativamente. Inoltre, in questo modo tutte le informazioni sono contenute e organizzate in un unico punto, permettendo così una migliore **manutenibilità** sul lungo periodo e una migliore facilità nel trovare errori e modificare anche pesantemente la libreria.

Punto importante da menzionare per l'uso della libreria è che per poter chiudere la socket che viene aperta alla creazione di un oggetto `TusowJS`, non esiste altro modo che procedere manualmente tramite il metodo `close()`. Se usato in maniera scorretta, può causare **comportamenti non corretti o inaspettati**.

5 Self-assessment / Validation

Per testare e valutare il software prodotto si è usato un modello *Test-driven*.
I Test, scritti in vari linguaggi, puntano a controllare che ogni parte del progetto sia funzionante a prescindere dalle altre.
A tal fine sono stati quindi sviluppati Test specifici per:

1. le funzionalità dello `InspectableTupleSpace`;

[7] `http://browserify.org`

2. l'utilizzo delle WebSocket tramite `EventBus`;

3. la libreria *TusowJS*, sviluppata in JavaScript per interfacciarsi con il servizio REST.

Nelle seguenti sottosezioni verranno spiegati nel dettaglio i vari Test.

5.1 Testing of InspectableTupleSpace

La *classe astratta* `TestTupleSpaceInspectability` in `linda-test` si occupa di testare il concetto di `InspectableExtendedTupleSpace`. I Test si basano sui valori restituiti dalla classe `TupleTemplateFactory` che consente di ottenere dati di prova per effettuare le operazioni sullo spazio di tuple agevolmente.
La sua specializzazione `TestLogicSpaceInspectability` in `linda-logic` si occupa di testare l'implementazione di `InspectableLogicSpace`.
Di seguito verranno elencati i vari Test e cosa testano.

1. I Test `testReadInvocation` e `testTakeInvocation` si occupano di accertare la sola invocazione di un'operazione di *"rd"* e di *"in"* in uno spazio di tuple vuoto.

2. Il Test `testWrite` si occupa di verificare la corretta invocazione e completamento di un'operazione di *"out"*.

3. I Test `testReadCompletion1` e `testTakeCompletion1` eseguono un'operazione di *"out"* e poi rispettivamente una di *"rd"* e di *"in"*, controllando che la sequenze di eventi ci sia prima l'invocazione e il completamento della *"out"* e poi, in base al Test, i rispettive eventi di invocazione e completamento per la *"rd"* o la *"in"*.

4. I Test `testReadCompletion2` e `testTakeCompletion2` eseguono prima rispettivamente un'operazione di *"rd"* e di *"in"*, poi un'operazione di *"out"* attendendo il suo completamento, in seguito attendono il completamento della prima operazione invocata. In questo caso l'ordine degli eventi non è come nel caso precedente e non si possono fare asserzioni troppo restrittive sullo stesso. L'unica asserzione dal punto di vista di ordine è che la prima operazione viene invocata prima della seconda.

5. I Test `testTryReadFail` e `testTrTakeFail` si occupano di testare che l'invocazione di una *"rdp"* e di una *"inp"* su uno spazio di tuple vuoto venga effettivamente completata.

6. I Test `testTryReadSuccess` e `testTryTakeSuccess` si occupano, invece, di verificare il successo del completamento di una *"rdp"* e di una *"inp"* dopo aver invocato e completato una *"out"*.

7. Il Test `testWriteAll` si occupa di testare l'effettivo completamento di un'operazione di *"out_all"*.

8. Il Test `testGet` si occupa di testare l'effettivo completamento di un'operazione di *"get"* dopo aver effettuato e completato una *"out_all"*.

9. I Test `testReadAllCompletation1` e `testTakeAllCompletation1` si occupano di testare l'effettivo completamento di un'operazione di *"rd_all"* e *"take_all"* dopo aver effettuato e completato una *"out_all"*.

10. I Test `testReadAllCompletation2` e `testTakeAllCompletation2` eseguono prima rispettivamente un'operazione di *"rd_all"* e di *"take_all"*, poi un'operazione di *"out_all"* attendendo il suo completamento, in seguito attendono il completamento della prima operazione invocata. In questo caso l'ordine degli eventi non è come nel caso precedente e non si possono fare asserzioni troppo restrittive sullo stesso. Oltre alle asserzioni fatte sull'ordine delle invocazioni, come nei Test `testReadCompletion2` e `testTakeCompletion2`, è possibile fare asserzioni sui `TupleEvent` invocati che possono essere una qualsiasi combinazione di TupleEvent della prima operazione invocata, per fare questo viene usato il concetto di `PowerSet` ovvero tutte le disposizioni degli eventi possibili.

11. Il Test `testAbsentInvocation` dopo aver effettuato e completato una *"out"*, testa il fallimento di un'operazione di *"no"* con come argomento il template della tupla con cui è stata effettuata la *"out"*.

12. Il Test `testAbsentCompletation` testa il completamento di un'operazione di *"no"* su un tuple space vuoto.

13. Il Test `testAbsentInvocationAndCompletation`, dopo aver invocato e completato una *"out"*, invoca un'operazione di *"no"*, in seguito invoca e completa una *"in"* e attende il completamento della *"no"*. In questo modo viene testato il successo della *"no"* dopo aver effettuato la *"in"*.

14. Il Test `testTryAbsentSuccess`, testa il completamento di un'operazione di *"nop"* su un tuple space vuoto.

15. Il Test `testTryAbsentFail`, dopo aver effettuato una *"out"*, testa il fallimento di un'operazione di *"nop"* con come argomento il template della tupla con cui è stata effettuata la *"out"*.

5.2 Testing of EventBus

La *classe astratta* `TestInspectableService` in `tusow` si occupa di testare il funzionamento dell'`EventBus` applicato a un Client Java e al Servizio REST sviluppato. I Test si basano sui valori restituiti dalla classe `TupleTemplateFactory` che consente di ottenere dati di prova per effettuare le operazioni sullo spazio di tuple agevolmente. La sua specializzazione `TestLogicInspectableService` testa l'implementazione nello spazio di tuple logico ispezionabile. I Test sono molto simili a quelli della sezione precedente, ma puntano a testare che la WebSocket riceva nel formato JSON i `TupleSpaceEvent` inviati dal Servizio e si testa la serializzazione e deserializzazione dei dati che vengono inviati e ricevuti per ogni richiesta al Server.

5.3 Testing of JavaScript library for interact with Tusow

All'interno di questo gruppo di Test si è verificata la corretta funzionalità delle richieste, basandosi sulla libreria `TusowJS`.
I passaggi necessari all'esecuzione di questo Test sono:

1. Eseguire il servizio TuSoW **manualmente** tramite `./gradlew run`.
2. Aspettare l'effettivo avvio del servizio, cioè la visualizzazione della stringa "`INFO: Service listening on port: XXXX`" sulla console lato Server.
3. Utilizzare il comando `./gradlew npmTest` per eseguire i Test.

Sfortunatamente il Test, a differenza degli altri in Java, **non è automatizzato**, in quanto non è possibile eseguire l'ultimo comando solo e soltanto **dopo** l'effettivo avvio del servizio, poiché non esiste un modo per intercettare questo evento.

Per effettuare il controllo sull'effettiva correttezza all'interno di ogni Test si è reso necessario eseguire un controllo sul campo `ResponseText`, verificando l'eventuale uguaglianza con una stringa desiderata. Poiché le risposte JSON del Servizio TuSoW non seguono un'ordine preciso dei dati, questo fa si che **non ci si possa limitare** a un semplice controllo sulla stringa ricevuta, ma al contrario si debba eseguire una serializzazione in oggetto JSON. In questo modo, benché se comparate come stringhe {`"A":"value", "B":"val"`} e {`"B":"val", "A":"value"`} risultino differenti, è possibile validarle come identiche, secondo un procedimento simile a quello menzionato nella Sezione 4.3.

6 Deployment Instructions

Per eseguire il progetto, dopo aver clonato la repository da GitLab, è necessario:

1. assicurarsi di aver installato sia *Node.js* (v11.10.0) che *npm* (6.7.0)
2. eseguire `./gradlew run` dalla directory principale del progetto per lanciare il servizio e una volta che il servizio sarà in ascolto, proseguire con il punto successivo;
3. andare su `http://localhost:8080/tusow/v1/tuple-spaces/logic` dal Browser (i.e. Google Chrome) per accedere all'interfaccia Client e interagire con la stessa e con il servizio.

7 Usage Examples

In questa sezione viene fornito un esempio di utilizzo dell'interfaccia grafica per poter comprendere il funzionamento di `TusowJS`.

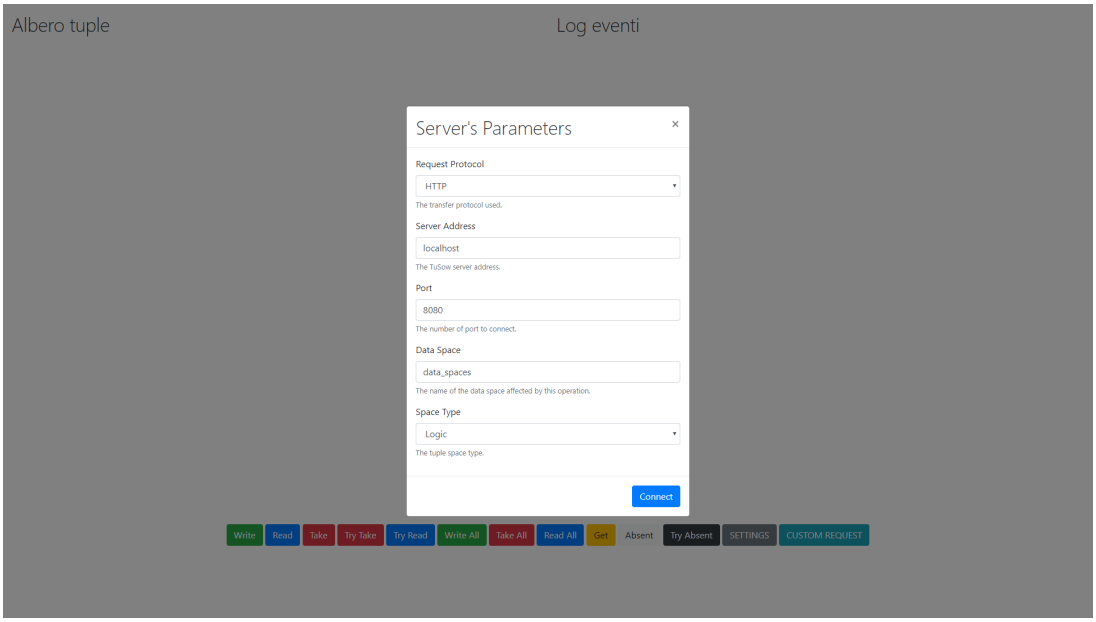

Figure 10: Primo schermata visibile, scelta dei parametri per il collegamento al servizio TuSoW desiderato.

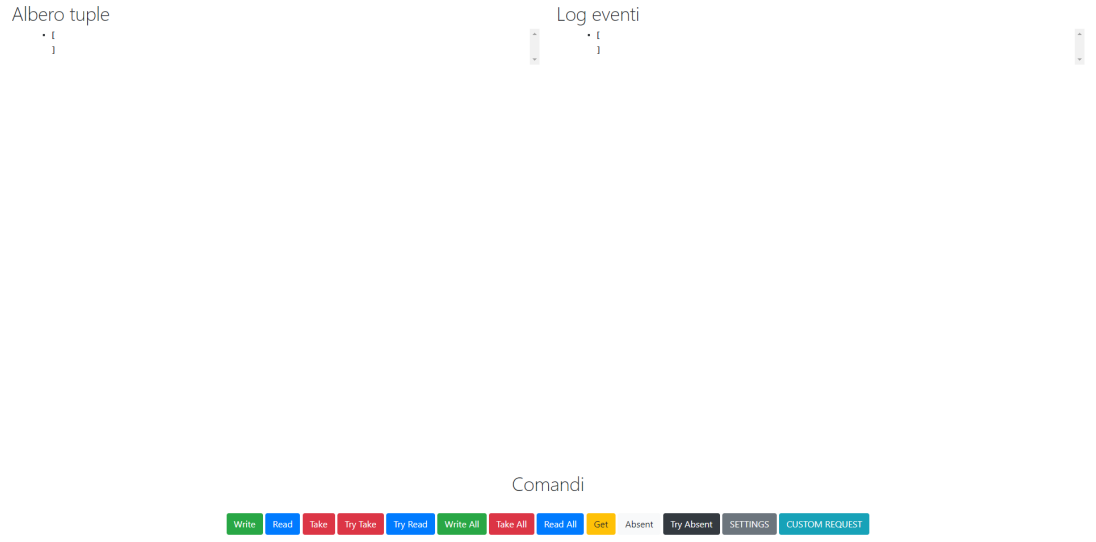

Figure 11: Main page dell'interfaccia grafica, stato iniziale, senza tuple ed eventi, con serie di pulsanti per ogni richiesta utilizzabile.

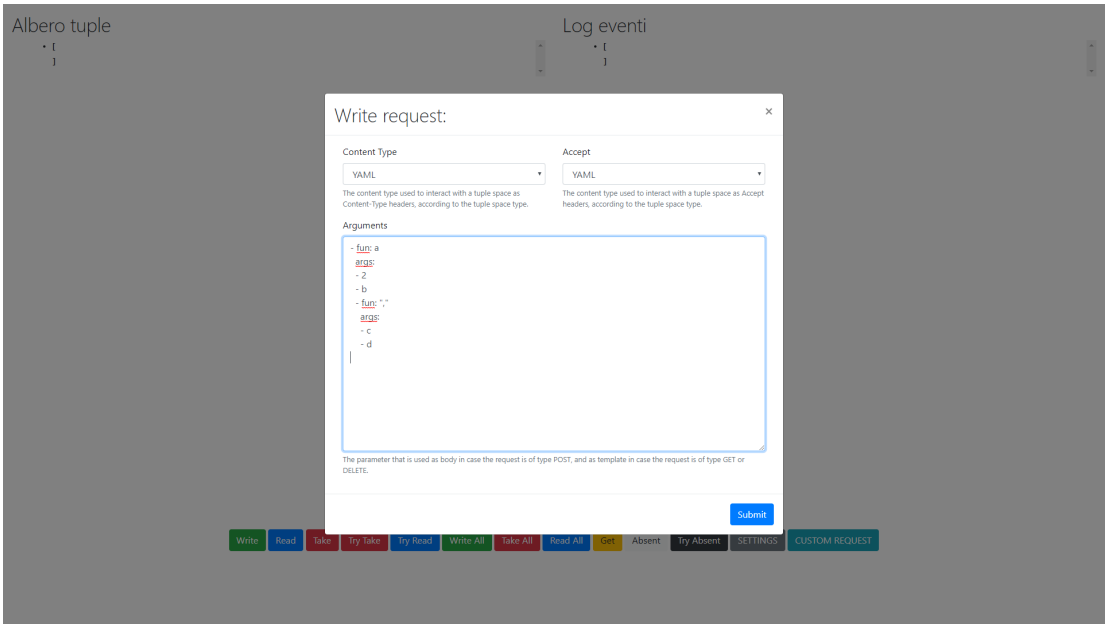

Figure 12: Modal richiamabile scegliendo la tipologia di richiesta da inviare, con parametri e argomenti a piacere.

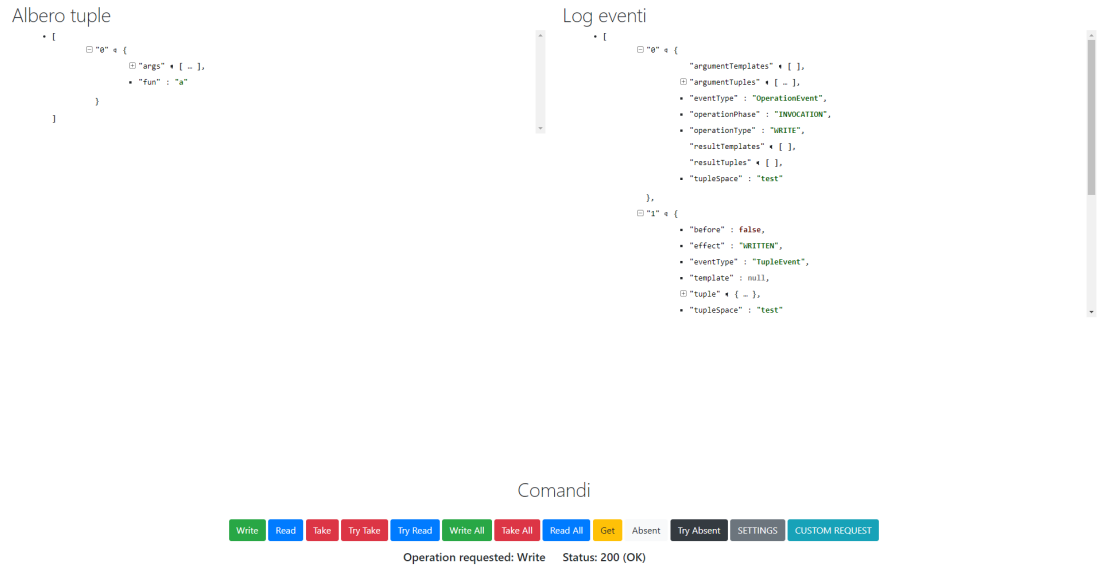

Figure 13: Reazione dell'interfaccia grafica alla ricezione da parte del Server di una richiesta: aggiornamento albero tuple, log eventi e riepilogo status richiesta (sotto alla riga pulsanti).

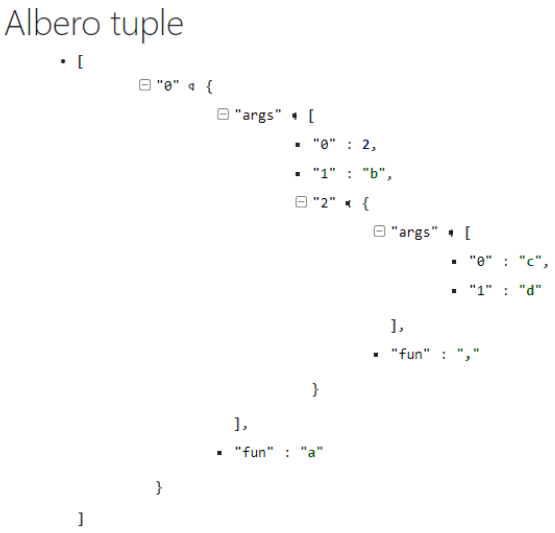

Figure 14: Dettaglio del box tuple, con possibilità di espandere e comprimere l'albero della struttura.

Figure 15: Dettaglio del box eventi, con possibilità di espandere e comprimere l'albero della struttura.

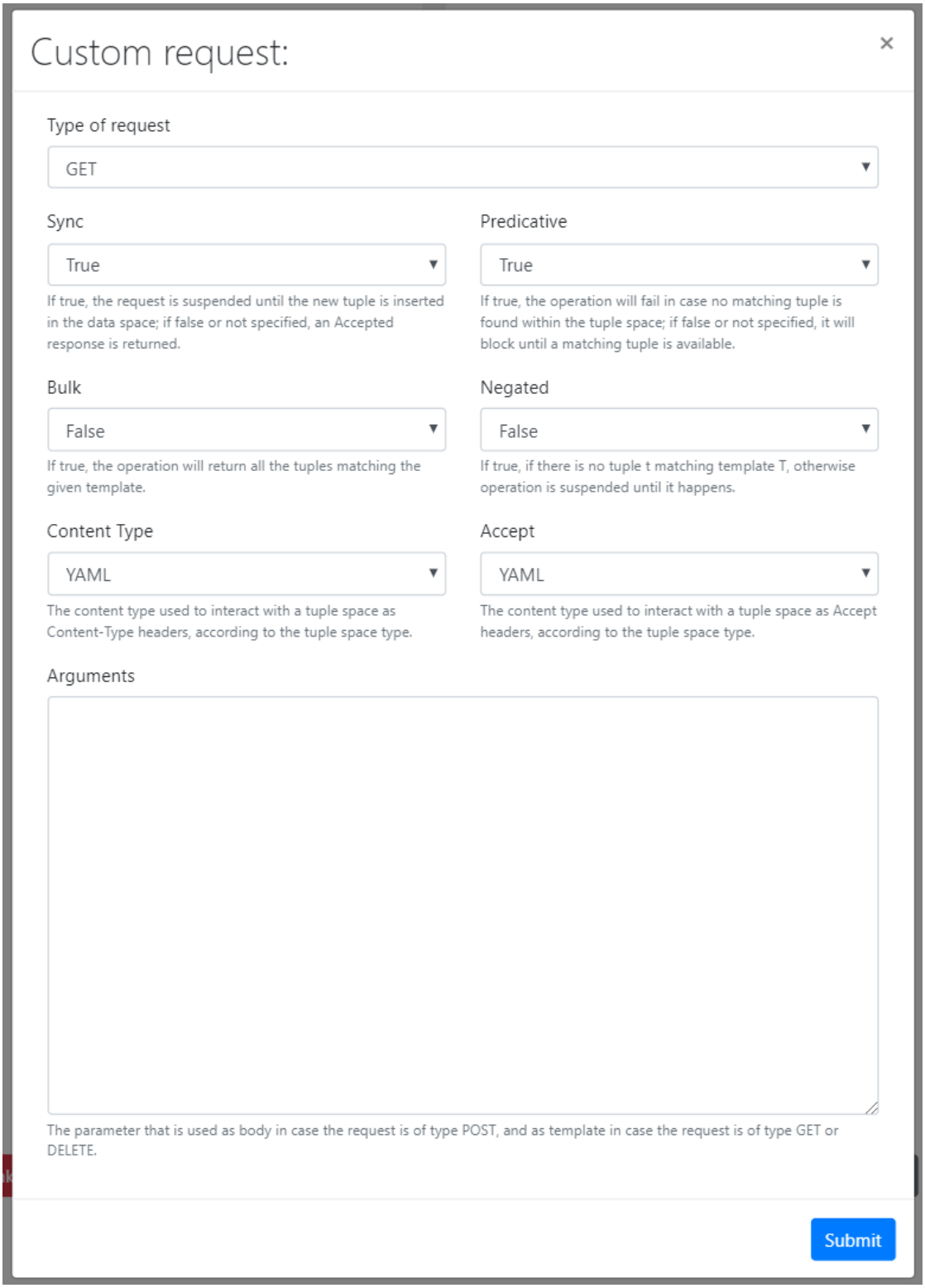

Figure 16: Dettaglio della modal richiamabile per l'invio di una richiesta totalmente personalizzabile.

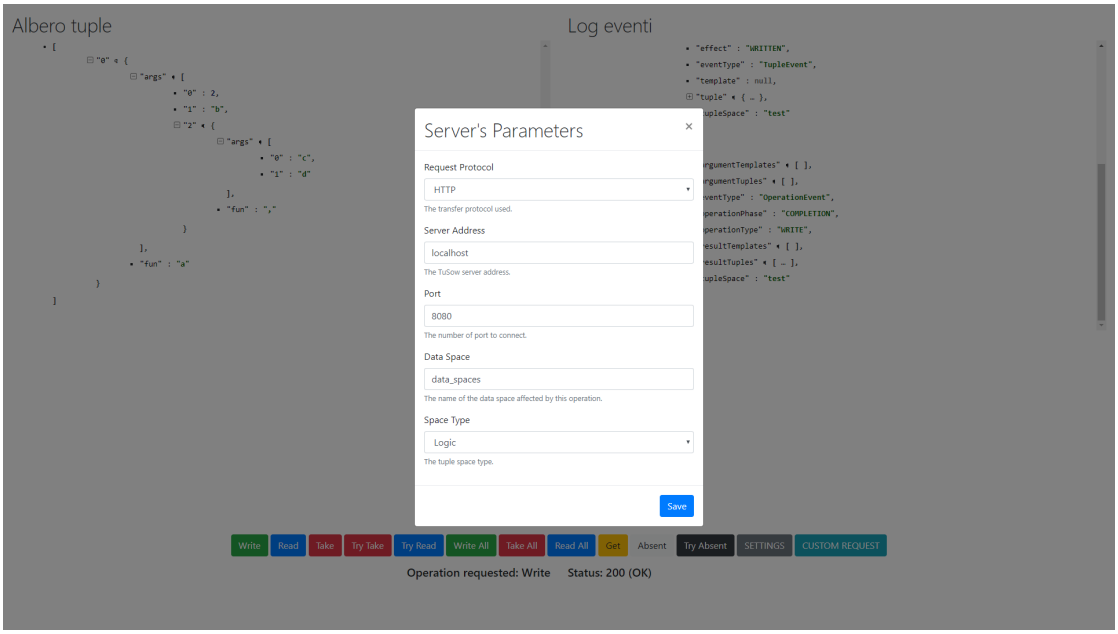

Figure 17: Dettaglio della modal richiamabile per il cambio dei parametri anche a connessione già avvenuta e con scambio di eventi.

8 Conclusions

In conclusione, il progetto ha in prima battuta esteso il concetto di spazio di tuple presente, aggiungendo il concetto di ispezionabilità, modellando gli eventi e integrando quanto sviluppato in un'API REST. Si sono andati a testare, per quanto concerne lo spazio di tuple logico, sia il concetto di spazio di tuple ispezionabile che il concetto di Servizio e di EventBus per monitorare gli eventi. Lato JavaScript, il progetto ha sviluppato una libreria per interagire con il servizio sul web e un'interfaccia grafica in grado di utilizzarla. La GUI sviluppata risulta pulita e semplice per fare richieste, monitorare il flusso di eventi e ricostruire partendo da esso lo spazio di tuple.

8.1 Future Works

In futuro si potrebbe far in modo che il servizio supporti anche lo spazio di tuple delle stringhe, attraverso piccole modifiche. Inoltre, si potrebbe supportare la serializzazione delle informazioni dello spazio di tuple lato JavaScript e far in modo che lo stato iniziale dello spazio di tuple possa essere conosciuto da un Client senza dover inviare una richiesta *GET* preliminare. Per quanto riguarda la libreria, sicuramente si potrebbe pensare a una gestione migliorata, evitando di delegare all'utente compiti che potrebbe non fare (come chiudere la socket o gestire in maniera corretta l'istanza della libreria). Per quanto riguarda l'interfaccia web le migliorie potrebbero essere svariate: svilupparne

una completamente *mobile responsive*, migliorare la visualizzazione di eventi e tuple, aggiungere il supporto a più dataspace, permettere di poter inviare più richieste contemporaneamente e gestirne storico e dettagli, rendendo graficamente più utilizzabile la visualizzazione grafica della risposta.

8.2 What did we learned

Attraverso questo progetto abbiamo imparato a utilizzare agevolmente la libreria *vertx* ed effettuare *testing* con la stessa, conoscere meglio quanto accade nello spazio di tuple integrando il concetto di *ispezionabilità* e, di conseguenza, di *evento*. Abbiamo capito come utilizzare quest'ultimo per ricostruire uno spazio di tuple. Un'altra cosa fondamentale che abbiamo imparato è stato pensare in modo tale da rendere i concetti astratti e modularizzabili per poi poterli riusare in altri contesti. Infine abbiamo imparato a sviluppare una libreria JavaScript e pertanto è stato approfondito l'uso delle tecnologie *Node.js*, *npm*, *Typescript* e *gradle* per l'automazione di compilazione e testing.

La suddivisione del lavoro scelta ha portato che un membro del gruppo si dedicasse di più al lato Server (Forresi) e l'altro a tutta la parte Client (Montalti), questo ci ha consentito di essere abbastanza indipendenti nel lavoro in Team e comunicare all'altro aspetti relativi a un modulo specifico del software prodotto.