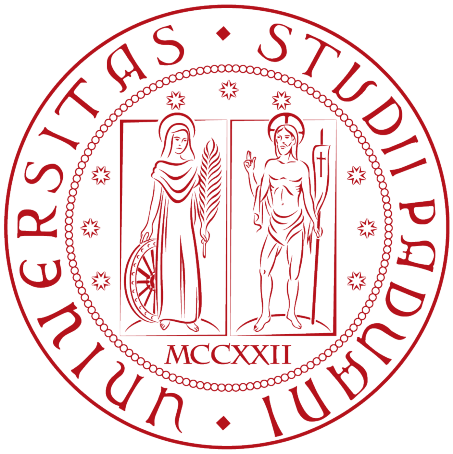
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Stato applicativo multi-finestra in JavaScript

Tesi di laurea triennale

Relatore

Prof.Gilberto Filè

Laureando

Giovanni Jiayi Hu

Anno Accademico 2017-2018

A rock pile ceases to be a rock pile the moment a single man contemplates it, bearing within him the image of a cathedral.

— Antoine de Saint-Exupéry, The Little Prince

Sommario

Il presente documento è la relazione finale del lavoro svolto durante il periodo di stage del laureando Giovanni Jiayi Hu presso l'azienda WorkWave Italy Srl, della durata di 312 ore.

Lo scopo dello stage è stato l'esplorazione e l'apprendimento delle più recenti tecnologie web per la realizzazione di applicazioni web multi-finestra. A tale scopo è stato necessario eseguire un'attività di Ricerca & Sviluppo (R&D), testarne la loro maturità e realizzare un Proof of Concept che sfrutti tali tecnologie per poter estrarre porzioni di interfaccia grafica dall'applicazione principale in una nuova pagina autonoma, ma sincronizzata a livello di stato applicativo.

La prima fase delle attività ha portato dunque alla nascita della prima versione di una libreria battezzata col nome *Stargate*, ispirato dall'ononima serie tv di fantascienza.

In secondo luogo è stato richiesta l'evoluzione e l'integrazione di Stargate nell'applicazione web in via di sviluppo denominata *WorkWave Route Manager*. Quest'ultima è la nuova versione di uno dei prodotti principali che l'azienda offre ai propri clienti e permette di pianificare, dirigere, tracciare e analizzare le rotte dei propri veicoli in tempo reale.
L'integrazione ha avuto difatti l'obiettivo di permettere la visualizzazione della mappa Google Maps, ricca di rotte ed veicoli, su un monitor separato full-screen.

Sia *Stargate* che *Route Manager* sono basati su TypeScript, un linguaggio tipizzato che compila in JavaScript, ed utilizzano le librerie React 16 e Redux 4. In particolare *Stargate* è usufruibile su qualsiasi applicazione web JavaScript, ma fornisce già le integrazioni per agevolarne l'uso con React e Redux.

Organizzazione del testo

Il secondo capitolo descrive ...

Il terzo capitolo approfondisce ...

Il quarto capitolo approfondisce ...

Il quinto capitolo approfondisce ...

Il sesto capitolo approfondisce ...

Nel settimo capitolo descrive ...

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

* gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
* per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*[g];
* i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Gilberto Filè, relatore della mia tesi, per la disponibilità che mi ha offerto durante il periodo di stage e per i preziosi insegnamenti durante questi tre anni.

Un ringraziamento speciale a Cesare d'Amico e a tutti i colleghi di WorkWave per la calorosa accoglienza fin dai primi giorni di stage. E naturalmente un sentito grazie a Matteo Ronchi, che ha saputo sempre guidarmi saggiamente durante questa esperienza e condividere pazientemente le sue preziose conoscenze.

Vorrei inoltre dare un caloroso abbraccio ad Elisa per l'affetto e per avermi insegnato ad amare i dettagli della vita. Un sentito ringraziamento invece per tutti gli amici, sia di università che di vita, i quali mi hanno sopportato e tenuto compagnia in questo percorso.

Desidero infine ringraziare con affetto i miei genitori per il costante sostegno ed incitamento nei miei progetti e nelle mie decisioni.

Padova, Settembre 2018 Giovanni Jiayi Hu

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

L'azienda

1.1 Descrizione

WorkWave, una divisione di IFS, è una società americana fondata nel 1984 con anche sede in Italia che fornisce soluzioni di Field Service Management e che connette ogni aspetto di un business attraverso le sue piattaforme unificate e di facile uso. L'insieme delle soluzioni della compagnia permettono ai professionisti di servizi ultimo-miglio di facilmente assegnare ed automatizzare attività di vendita e marketing, migliorando l'efficienza ed incrementando la visibilità delle operazioni sul campo attraverso le soluzioni mobile.

Le piattaforme di WorkWave forniscono ad oltre 8 mila clienti un livello senza precedenti di analisi del business, permettendolo loro di aumentare l'efficienza, il guadagno e garantendo un'eccezionale customer experience.

Figura 1.1: Logo dell'azienda WorkWave

1.2 Servizi offerti

WorkWave aiuta aziende nel campo Field Service Management ed industrie di trasporti e logistica mitigare gli aspetti dolorosi che incontrano ogni giorno, consentendo loro di salvare tempo, spese e migliorando il servizio agli utenti. Per Field Service Management si intendono risorse impiegate per intradare verso i domicili dei clienti, quali localizzazione dei veicoli, gestione delle attività degli operatori, pianificazione ed impiego delle attività, garanzia della sicurezza dei conducenti ed integrazione di tali servizi con depositi, fatturazione ed altri servizi back-office.

WorkWave fornisce sia soluzioni per l'installazione e manutenzione hardware che servizi software per l'aiuto della gestione di tali attività.

La suite dei servizi software cloud, mobile e marketing permette a compagnie di ogni dimensione di facilmente stimare attività, pianificare ed dirigere operatori mobili con facilità. Di seguito si elencano i principali correlati all'attività di stage:

* *WorkWave Service*: è un servizio software che consente ti velocemente schedulare rotte efficienti, visionare la produttività in tempo reale degli operatori, visualizzare stime e gestire pagamenti;

* **WorkWave Route Manager**: è un set di servizi per la gestione dei veicoli al fine di migliorare l'efficienza e la scalabilità attraverso pianificazione dinamica e miglioramenti intelligenti delle rotte. L'algoritmo proprietario del Route Manager garantisce che le migliori rotte per gli operatori siano usate per salvare tempo, costi e migliorare la soddisfazione dei clienti;

* **WorkWave GPS**: fornisce un'intuitiva panoramica dei propri veicoli e delle propie risorse con un potente servizio GPS che cattura le azioni dell'operatore e le posizioni in tempo reale dei veicoli. Permette inoltre di migliorare la sicurezza dei guidatori, segnalare comportamenti errati e riportare incidenti per velocità, frenata improvvisa, curve o altro.

1.3 Route Manager

Figura 1.2: Logo WorkWave Route Manager

WorkWave Italy è la sede italiana di WorkWave dove sono concentrati gli sviluppi dell'algoritmo di routing e del Route Manager nella nuova versione denominata Unified UI, contesto di sviluppo delle attività dello stage descritte in questo documento. WorkWave Route Manager automatizza la pianificazione delle rotte per migliorarne l'efficienza e la comunicazione tra l'amministrazione e i guidatori dei veicoli, completamente customizzabile tramite le sue API.

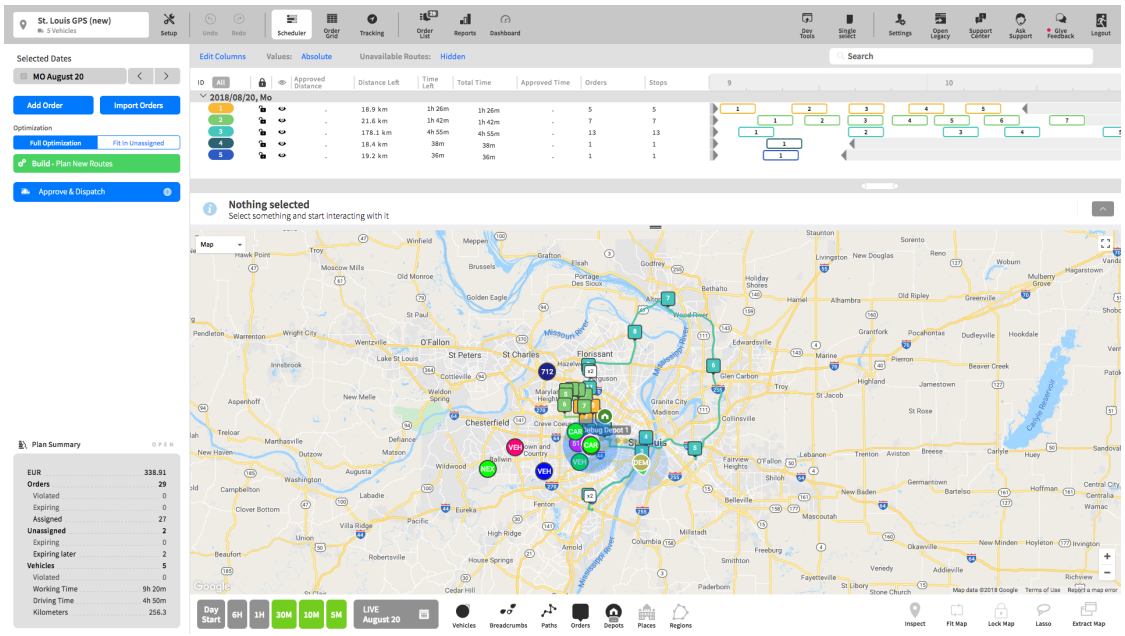


Figura 1.3: WorkWave Route Manager - Unified UI

In particolare, per quanto riguarda Routing & Scheduling, rende possibile navigare attraverso le richieste dei clienti per gli orari di ricezione, schedulare le attività dei guidatori giornalmente ed eseguire report sulle performance. Attraverso le impostazioni, il software fornisce rotte ottimali in base ai propri vincoli stradali.

È inoltre possibile fare aggiustamenti manuali alle rotte via drag&drop, approvare i piani e mandarli in esecuzione agli operatori sui veicoli. Oltre a ciò consente di visualizzare istantaneamente gli effetti sul numero di ordini possibili per i veicoli disponibili, il tempo stimato di completamento delle attività e comparare il costo per miglio.

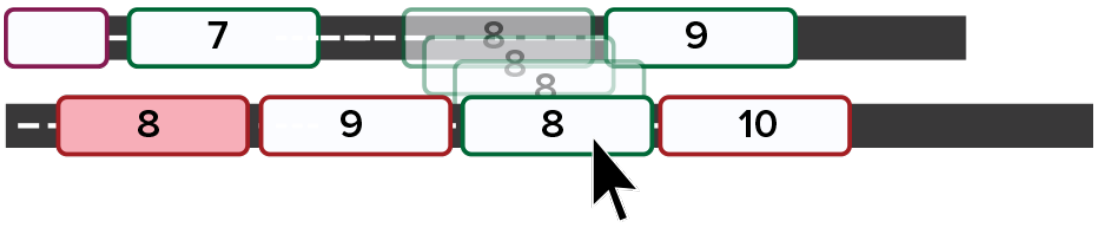


Figura 1.4: WorkWave Route Manager - Scheduler degli ordini

Capitolo 2

Lo stage

2.1 Presentazione del progetto Stargate

Il progetto dello stage consiste nella realizzazione di una libreria TypeScript per applicazioni web, che permetta di aprire qualsiasi componente di User Interface (UI) in una nuova pagina mantenendo lo stato sincronizzato con l'applicazione padre. Tale libreria è denominata **Stargate**, ispirandosi all'omonima serie televisiva di fantascienza incentrata su portali spaziali nelle diverse galassie. Idealmente la libreria dovrebbe funzionare per qualsiasi applicazione web scritta in JavaScript, tuttavia il requisito obbligatorio minimale è che si integri con librerie React e Redux col minor sforzo possibile.

La prima fase dello stage richiede dunque la realizzazione di un Proof of Concept che verifichi in vitreo la possibilità di spostare un componente UI React dalla finestra principale (*padre*) verso una nuova *figlia*, mantendo lo stato consistente. Tali componenti UI sono identificati col nome di *widget*, qualora siano mostrati su una pagina separata.
Un widget deve continuare a visualizzare i dati provenienti dal padre e, se questi si aggiornano, anche il componente nella finestra figlia deve aggiornarsi consistemente. Viceversa le interazioni utente nel componente devono essere propagate alla pagina principale. In sostanza, il componente deve esibire lo stesso comportamento di quando si trovi nell'applicazione principale, sebbene fisicamente si trovi su una diversa pagina del browser.
Infine non vi devono essere vincoli sul numero di componenti aperte in contemporanea su pagine diverse, sia che siano componenti di tipologia diversa sia che siano lo stesso componente istanziato molteplici volte ma indipendenti tra loro.

In secondo luogo, è necessario integrare *Stargate* all'interno del prodotto *Route Manager* implementando la possibilità di estrarre la mappa Google Maps su una nuova pagina. L'obiettivo è permettere all'utente di visualizzare le informazioni sulla mappa secondo molteplici prospettive a sua discrezione, ad esempio mostrandolo tutti i veicoli su una pagina, solo una singola rotta real-time su un'altra. Inoltre permette di usufuire della mappa su schermi multipli, funzionalità fortemente desiderata in quanto la mole delle informazioni a schermo è elevata e l'applicazione

principale mostra difficoltà nel visualizzarle tutte in una sola pagina web.

Lo sviluppo di *Stargate* dovrà dunque avere le seguenti caratteristiche:

* Supporto multi-finestra di componenti React;
* Supporto per grandi moli di dati, ad esempio geospaziali, in continuo aggiornamento;
* Possibilità di eseguire il componente in una nuova pagina che risieda su un processo separato del sistema operativo, affinché alleggerisca il carico computazione dell'applicazione principale. In particolare questa caratteristica è utile per evitare che i calcoli geospaziali vengano eseguiti dal processo padre;
* Supporto a molteplici finestre in contemporanea, tutte sincronizzate rispetto all'applicazione padre;
* Supporto obbligatorio unicamente per i browser moderni Google Chrome e Firefox. Gli utenti che utilizzino browser non compatibili con *Stargate*, potranno usuguire della normale esperienza utente ma senza la possibilità di aprire nuove finestre;
* Supporto a multi-sessione. Qualora l'utente apra due istanze dell'applicazione padre ed in ognuna crei una nuova finestra per un componente, ciascuno di questi deve essere sincronizzato col rispettivo padre e non deve creare conflitti con l'altra applicazione principale.

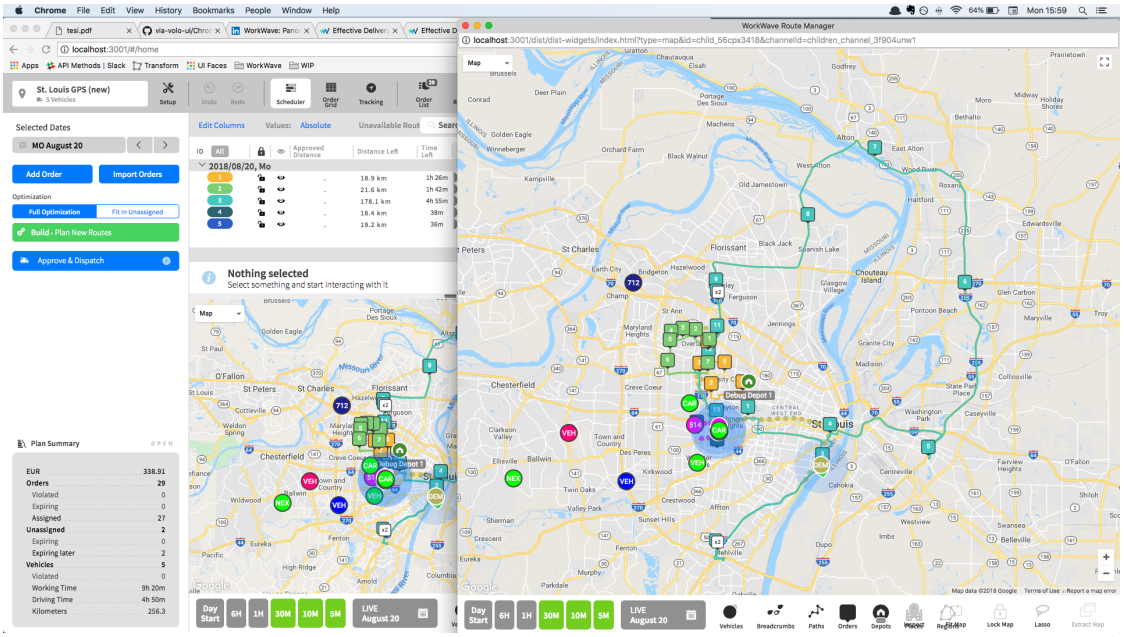


Figura 2.1: WorkWave Route Manager con Google Maps in una nuova finestra attraverso Stargate

2.2 Obiettivi

Si farà riferimento agli obiettivi secondo le seguenti notazioni:

* **O** per i requisiti obbligatori, vincolanti in quanto obiettivo primario richiesto dal committente;

* **D** per i requisiti desiderabili, non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;

* **F** per i requisiti facoltativi, rappresentanti valore aggiunto non strettamente competitivo.

Le sigle precedentemente indicate saranno seguite da una coppia sequenziale di numeri, identificativo del requisito. Si quindi prevede lo svolgimento dei seguenti obiettivi:

ID	Descrizione
Obbligatori	
O01	Supporto multi-finestra di componenti React
O02	Supporto per grandi moli di dati in continuo aggiornamento, dell'ordine di alcuni MByte
O03	Supporto a molteplici finestre in contemporanea, tutte sincronizzate rispetto all'applicazione padre
O04	Supporto per i browser moderni Google Chrome v56 e Firefox v38
O05	Supporto a multi-sessione
O06	Gestione della configurazione di *Route Manager* per supportare i widget
O07	Produzione della documentazione d'uso di *Stargate*
O08	Utilizzo del linguaggio TypeScript v3
Desiderabili	
D01	Possibilità di eseguire il componente in una nuova pagina che risieda su un processo separato del sistema operativo
D02	Supporto prestazionale fino ad almeno 5 tabs simultanee
D03	Supporto componenti JavaScript non React, ad esempio Angular 2
Facoltativi	
O04	Supporto per i browser Internet Explorer v11 e Edge v12

Tabella 2.1: Tabella degli obiettivi

2.3 Pianificazione

In accordo col tutor Matteo Ronchi, l'attività dello stage è stata suddivisa nelle seguenti fasi:

* **Fase 1**: Analisi dello stato dell'arte per la comunicazione cross-page in JavaScript
* **Fase 2**: Realizzazione Proof of Concept in React e Redux
* **Fase 3**: Progettazione per integrazione in *Route Manager* del widget Google Map. Implementazione dell'integrazione ed evoluzione della libreria.
* **Fase 4**: Validazione e stesura documentazione

Non vi è stata necessità di un periodo iniziale di formazione sulle tecnologie TypeScript, React e Redux io quanto già in mio possesso. Mi sono anche trovato subito a mio agio con i processi di sviluppo aziendali, già incontrati da me in altre occasioni.
Essendo inoltre un'attività di Ricerca e Sviluppo, vi è stata una continua interazione col tutor aziendale per la definizione dei successivi step, ma a monte è stata pianificata un'ipotetica suddivisione delle ore nel seguente modo:

Ore	**Descrizione dell'attività**
40	Analisi dello stato dell'arte tecnologico
80	Realizzazione Proof of Concept in React e Redux
32	Progettazione architteturale
104	Integrazione in *Route Manager* del widget Google Map
24	Gestione configurazione per supportare widget
16	Validazione e Collaudo
8	Refactoring prima del rilascio in produzione
8	Stesura documentazione
Totale: 312 ore	

Tabella 2.2: Tabella della suddivisione delle ore

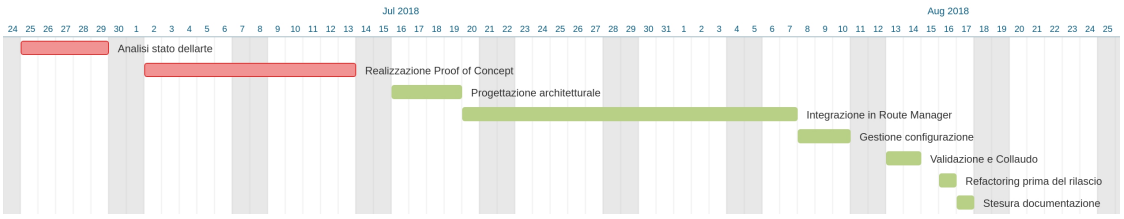


Figura 2.2: Diagramma Gantt della pianificazione

2.4 Analisi preventiva dei rischi

Durante la fase di analisi iniziale sono stati individuati alcuni possibili rischi a cui si potrà andare incontro. Si è quindi proceduto a elaborare delle possibili soluzioni per far fronte a tali rischi.

Descrizione	**Piano di emergenza**	**Rischio**
Difficoltà tecnologica: vi è il rischio che lo stato dell'arte dello sviluppo web non consenta di aprire finestra come processi separati o che non sia possibile effettuare la comunicazione tra pagine diverse.	È importante effettuare un'attenta attività di ricerca iniziale, al fine di comprendere lo stato dell'arte per quanto riguarda la comunicazione cross-page in applicazioni web. In caso di difficoltà nella ricerca della soluzione tecnica ideale, si adotterà quella con miglior compromesso di affidabilità-performance tra quelle disponibili.	Occorrenza: Alta Pericolosità: Alta
Difficoltà di integrazione: vi è il rischio che non sia possibile integrare le tecnologie adottate nel contesto dell'applicazione *Route Manager*, in quanto è già in sviluppo da oltre un anno e non progettata in partenza per essere multi-finestra.	In caso di verifica del rischio, si analizzerà il problema col tutor Matteo Ronchi per trovare la miglior soluzione da adottare in *Stargate* oppure in *Route Manager*	Occorrenza: Alta Pericolosità: Alta

Tabella 2.3: Tabella dell'analisi dei rischi

2.5 Aspettative aziendali

L'azienda WorkWave spera grazie allo stage di poter implementare una funzionalità fortemente desiderata nei loro prodotti software, in particolare in *Route Manager*, con la possibilità di mostrare su pagine diverse qualsiasi componente dell'interfaccia. Ciò apre difatti le porte per una serie di possibili nuove interazioni da parte

dell'utente.

La prima immediata conseguenza è il poter mostrare la mappa Google Maps su un monitor esterno ed in full-screen, estremamente utile sia durante i meeting che negli uffici del clienti del *Route Manager* per tenere sotto costante monitoraggio le rotte ed i veicoli. Il tutto mantendo contemporeaneamente aperta l'applicazione principale su un'altra pagina, ove poter fare le modifiche alle pianificazioni ed altre attività.

Un ulteriore caso d'uso è la possibilità di aprire qualsiasi componente in una nuova pagina, permettendo agli utenti di customizzare i propri flussi di lavoro in modo da tenere sempre aperte alcuni widget durante la navigazione all'interno dell'applicazione.

Infine per l'azienda è anche un'opportunità di conoscere il tirocinante e valutarlo per una possibile futura assunzione.

2.6 Aspettative personali

Ho intrapreso questo stage con l'obiettivo primario di venire in contatto con un ambiente di lavoro focalizzato sul lavoro in team e sulla qualità dei loro prodotti. Ritengo difatti fondamentale apprendere non solo conoscenze tecniche, ma anche quelle sociali e sul modo di lavorare. Grazie a WorkWave ho avuto quindi l'opportunità di conoscere cosa significa lavorare in squadra, fare Pair Programming per pensare insieme alla risoluzione di un problema e condividere le esperienze nella realizzazione ed evoluzione di un prodotto proprio dell'azienda.

È altresì importante professionalmente il contatto con i processi lavorativi di un'azienda che opera a livello world-wide, ma cercando al coltempo di mantenersi agile ed efficiente. È stata difatti concordata la possibilità di partecipare agli stand-up e sono stati spiegati gli strumenti di pianificazione e comunicazione interni.

Ritengo inoltre indispensabile confrontarmi con colleghi molto più esperta e con maggiore esperienza, in particolare il tutor Matteo Ronchi, che ha condiviso pazientemente le ragioni dietro decisioni architteturale apprese grazie alla sua esperienza in progetti passati. Infine è molto istruttivo realizzare il livello delle conoscenze richieste per realizzare ed evolvere un prodotto complesso come *Route Manager*, sia a livello di Design/User Experience che di sviluppo tecnico.

Capitolo 3

Processi e metodologie

Durante il periodo di stage, ho avuto l'opportunità di entrare in contatto con i processi aziendali e diversi strumenti a supporto del mio lavoro, di seguito illustrati.

3.1 Accertamento di Qualità

Il processo di Accertamento di qualità provvede a garantire che il prodotto software sia conforme alle aspettative di qualità desiderate. Nello specifico, durante il mio periodo di stage, sono venuto a contatto con le seguenti pratiche di sviluppo.

3.1.1 Pull Request

Una Pull Request è una proposta di modifica al repository effettuata su Github. Essa è obbligatoria per qualsiasi modifica e deve essere sempre realizzata tramite un git branch dedicato, avente un nome univoco e semantico rispetto alle modifiche proposte.

Lo scopo della Pull Request in WorkWave è favorire la discussione delle modifiche da parte del team e permetterne un'attenta ispezione prima ritenerla valida. Tuttavia essa è anche un'opportunità di apprendimento sia per chi esegue la review che per chi la riceve, in quanto entrambi hanno modo di apprendere diversi approcci allo stesso problema.

È stato inoltre spiegato che essa permette anche, nel lungo termine, di venire a conoscenza di problematiche nel processo di sviluppo e poterle migliorare. Ad esempio la continua segnalazione di norme di sintassi è un indice della necessità di introdurre uno strumento automatico per la formattazione del codice.

3.2 Gestione della configurazione

3.2.1 Versionamento

L'azienda WorkWave organizza il proprio codice sorgente all'interno di diverse repository Git raggruppate sotto l'organizzazione GitHub dell'azienda. In particolare

è stato creato un repository dedicato al versionamento della prima versione della libreria *Stargate* assieme al Proof of Concept. Successivamente il codice sorgente della libreria è stato direttamente integrato nel repository dell'applicazione *Route Manager*.

3.2.2 Ambiente di verifica

Figura 3.1: Jenkins

Il processo di verifica è il più automatizzato possibile tramite tools eseguiti automaticamente con Jenkins https://jenkins.io/. Lo stesso procedimento avviene ad ogni Pull Request proibendone l'accettazione se le verifiche non sono superate.
Inoltre sono presenti script automatici che permettono di rilasciare in ambiente di sviluppo, demo, testing e produzione attraverso l'interfaccia grafica dashboard di Jenkins.

3.2.3 Ambiente di rilascio

Figura 3.2: Amazon Cloudfront

Il rilascio automatico eseguito da Jenkins porta al caricamento dell'applicazione *Route Manager* su Amazon Cloudfront https://aws.amazon.com/it/cloudfront/, un servizio di Content Delivery Network (CDN) che permette di distribuire l'applicazione con latenza minima nelle diversi Paesi del mondo.

3.3 Gestione di Progetto

3.3.1 Stand-up

Il team si incontra quotidianamente per lo stand-up, un incontro informale senza durata prefissata, che permette ai vari membri di allinearsi reciprocamente sullo stato di avanzamento ed eventuali problematiche. In particolare, nel caso di WorkWave tale attività è indispensabile in quanto vi sono alcuni del team che lavorano in remoto o negli Stati Uniti.
Tutti i membri, non solo i programmatori, sono invitati a partecipare e ad esporre su cosa stiano lavorando ed potenziali criticità, permettendo anche di trasmettere maggiore consapevolezza e conoscenza del progetto ai diversi partecipanti.

3.3.2 Ticketing

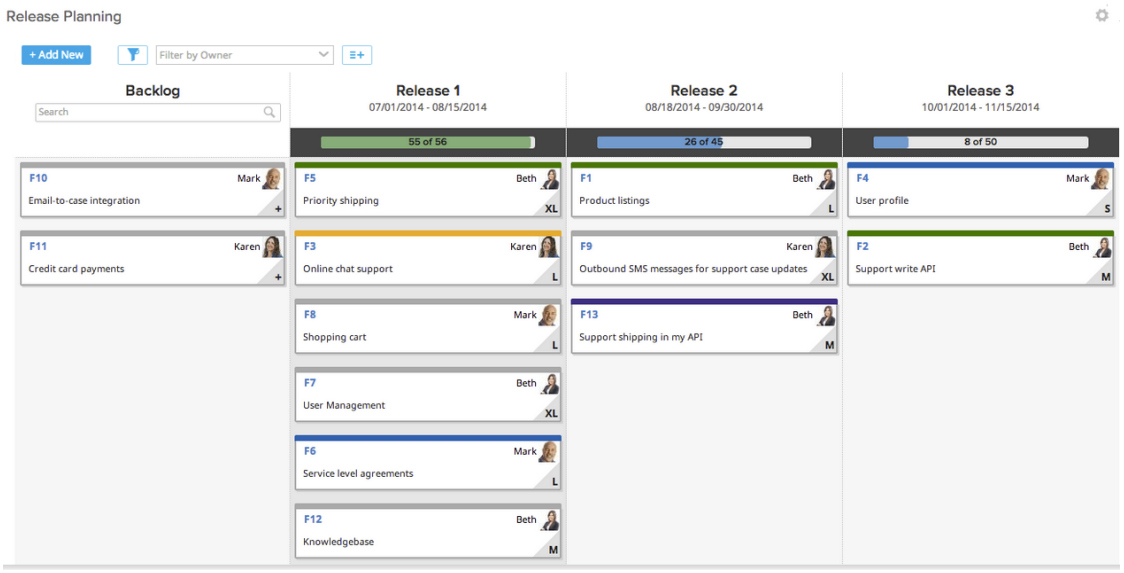


Figura 3.3: CA Agile Central

L'azienda utilizza lo strumento CA Agile Central `https://www.ca.com/it/products/ca-agile-central.html` per la gestione delle attività.

Capitolo 4

Architettura per la comunicazione cross-page

In questo capitolo viene presentata prima l'architettura multi-process dei browser, in particolare di Chrome, per la gestione delle pagine web. Uno degli obiettivi è difatti la possibilità di eseguire i componenti nelle finestre figlie come processi separati, in modo da migliorare e non inficiare il processo dell'applicazione principale.
In seguito si illustra invece lo stato dell'arte delle diverse soluzioni per la comunicazione cross-page in JavaScript tra pagine su tab diverse.

Entrambe le sezioni fungono da spiegazione del contesto tecnologico in cui è stata sviluppata la soluzione *Stargate* ed, al termine di esse, viene presentata una prima architettura naive per la libreria.

4.1 Architettura multi-processi

By design, JavaScript ha un modello di esecuzione single-threaded, ovvero tutte le sue istruzioni sono eseguite da un unico thread invece di averne diversi concorrenti. Ciò ha determinato la natura fortemente asincrona delle sue API, ove si cerca sempre di liberare il thread per i calcoli successivi appena possibile.
Qualora invece sia eccessivo il lavoro computazionale di una parte dell'applicazione, il risultato porta ad un'interfaccia bloccata e non responsiva fino al termine del calcolo. Questo ha un pessimo effetto sull'utente, in quanto l'applicazione non risponde alle sue interazioni e sembra anzi congelata. Per tale motivo è essenziale in primis che *Stargate* esegua le nuove finestre su processi separati, al fine di non appesantire l'applicazione padre.

Quando la maggior parte dei browser moderni fu progettata inizialmente, le pagine web erano semplici e avevano poco o nessun codice attivo. Per tale motivo, i browser renderizzano tutte le pagine usando lo stesso processo, al fine di mantenere basso l'utilizzo delle risorse.

Tuttavia, le pagine web odierne sono decisamente più attive a partire da siti statici ma con tanto uso di JavaScript fino a vere e proprie applicazioni web come Gmail.

Grosse parti di queste applicazioni girano all'interno del browser, così come le normali applicazioni eseguono in un sistema operativo e, proprio come questi, il browser deve dunque tenere le applicazioni separate tra di loro.

Oltre a ciò, le parti del browser che renderizzano HTML, JavaScript e CSS sono diventate straordinariamente complesse nel corso del tempo. Diventa perciò palese che browser i quali pongono tutto il lavoro in un processo affrontano seri problemi di rubustezza, responsitività e sicurezza.

Se un'applicazione web causasse un crash nel rendering engine, porterebbe la terminazione anche delle altre pagine web aperte. Le applicazioni web inoltre competono reciprocamente per l'uso della CPU ed ognuna di esse è single-thread per design di JavaScript, per cui rischierebbero di diventare non responsive alle interazioni utente. Infine anche la sicurezza è un fattore in rischio poiché una pagina web potrebbe sfruttare vulnerabilità del browser per accedere a dati delle altre pagine nello stesso processo.

4.1.1 Cosa fa ogni processo?

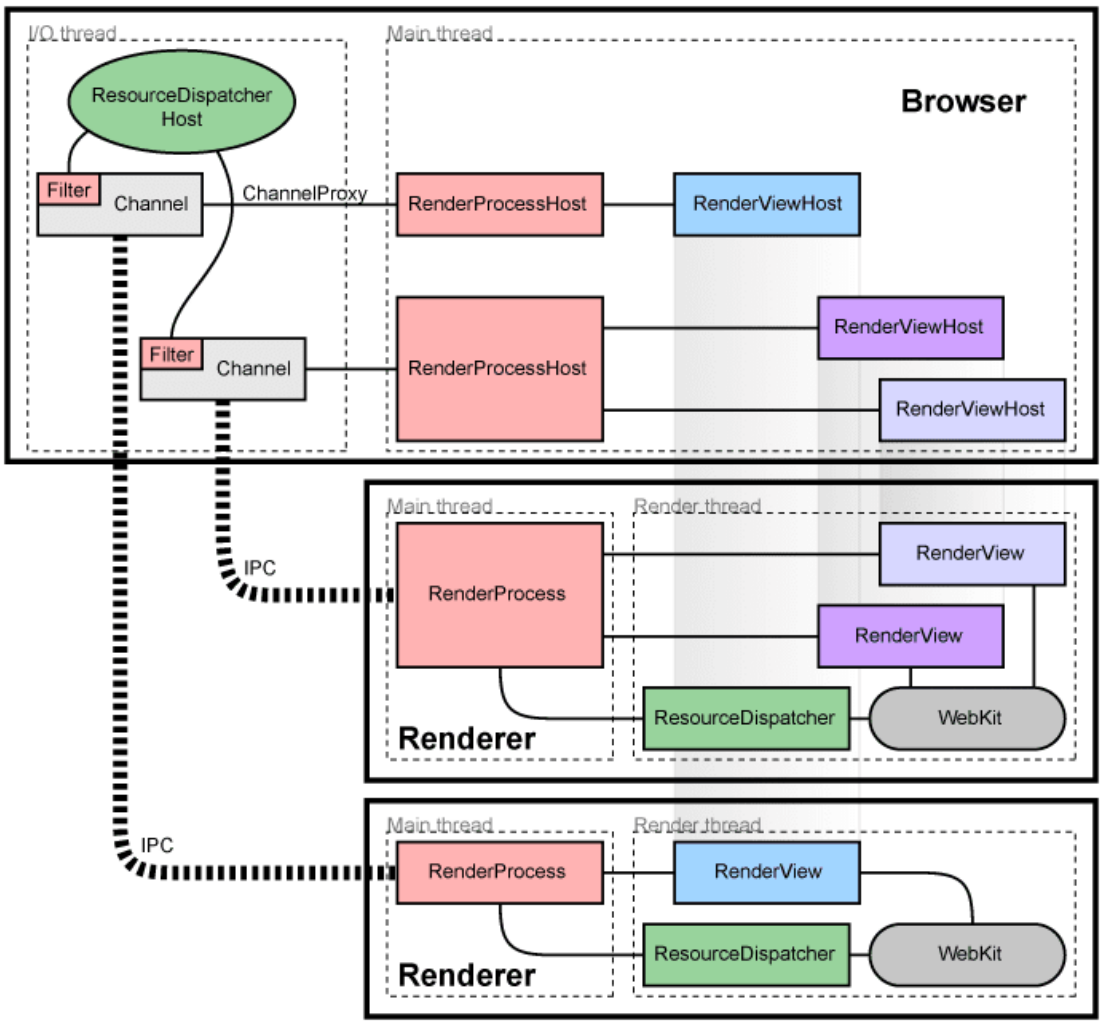


Figura 4.1: Architettura multi-process in Chrome

Il browser crea tre differenti tipi di processi: browser, renderer ed estensioni.

* **Browser**: esiste un unico processo browser, il quale gestisce i tab, le finestre e il browser stesso. Gestisce inoltre tutti i collegamenti delle pagine con il file system, la rete, input utente etc. ma non esegue alcun contenuto delle pagine;

* **Renderer**: il processo browser crea molteplici processi renderer, ognuno responsabile per la visualizzazione di una pagina web. I processi renderer contengono la complessa logica per la gestione di HTML, CSS, JavaScript, immagini e così via. Google Chrome, Safari ed altri utilizzano un rendering engine basato sul progetto open-source WebKit, mentre Firefox ed Edge hanno il proprio;

* **Estensioni**: il processo browser crea anche un processo per ogni estensione

4.1.2 Strategie multi-process

Una volta che il browser ha creato il processo omonimo, crea un processo renderer per ogni istanza di pagina web visitata dall'utente. Può essere pensato come un processo separato per ogni tab del browser, ma con l'eccezione di consentire a due tab di convidere lo stesso processo qualora siano collegati tra di loro e mostrino lo stesso sito.

Per esempio, se un tab ne apre un altro usando JavaScript o se viene aperto un link verso lo stesso sito in un nuovo tab, questi condivideranno lo stesso processo renderer. Si permette così ai tab correlati di comunicare via JavaScript e condividere la cache. Al contrario, se viene aperta una pagina di un sito diverso verrò riservato un nuovo processo.

Per essere precisi, si definisce un "sito" come un dominio registrato (ad esempio google.come o bbc.co.uk) e racchiude anche i sotto-domini (mail.google.com) e le porte (google.com:8080). Un' "istanza di sito" è invece un'insieme di pagine collegate provenienti dallo stesso sito. Due pagine sono considerate connesse se vi sono riferimenti reciproci in codice JavaScript, ad esempio una apre la seconda programmaticamente. Mentre se l'utente digita manualmente lo stesso indirizzo in due tab diverse, vengono considerate due istanze diverse con processi distinti.

Di seguito si illustrano nel dettaglio le diverse strategie multi-processi adottati dai browser moderni.

Process-per-site-instance

Normalmente i browser usano una strategia "Process-per-site-instance", ovvero lo stesso sito aperto in tab diversi con riferimenti reciproci saranno renderizzati dallo stesso processo. A volte è difatti necessario o desiderabile condividere il processo, quando per esempio un'applicazione web apre una nuova finestra con cui si aspetta di comunicare in maniera sincrona.

In generale invece, ogni nuova finestra o tab che non siano lo stesso sito possiede un nuovo processo.

Process-per-site

Raggruppa tutte le pagine dello stesso sito nello stesso processo, indipendentemente dalla presenza di riferimenti reciproci. Questa strategia è basata esclusivamente sul dominio del contenuto e non sulle relazioni tra le tab. Di conseguenza può risultare in processi molto onerosi.

Process-per-tab

Esiste anche una strategie più semplice che dedica un processo renderer per ogni gruppo di tab. Per ovvi motivi è estremamente inefficiente.

4.1.3 Come forzare l'uso di un nuovo processo

Dalla descrizione precedente della strategia *Process-per-site-instance*, sembrerebbe non sia possibile ottenere l'effetto desiderato per il progetto *Stargate*. Si desidera difatti alloccare un processo dedicato ad ogni finestra, sebbene appartengano allo stesso sito e quindi contrariamente al comportamento della strategia *Process-per-site-instance*.

In seguito a diverse ricerche è tuttavia emerso che è possibile ottenere un processo dedicato come effetto collaterale di un parametro di sicurezza per la creazione delle nuove finestre. Nei browsers moderni è difatti possibile specificare un parametro `rel=noopener`, che evita exploits di sicurezza in cui la finestra padre è capace di accedere a riferimenti contenuti nella finestra figlia e/o viceversa.
Quest'ultimo comportamento di condivisione dei riferimento è necessario per il corretto funzionamento di molte applicazioni cross-page, ma pone gli utenti a rischio qualora le nuove finestre siano di dominio diverso e potenzialmente maligne. Tramite il parametro `rel=noopener` è invece possibile evitare qualsiasi riferimento tra le parti e per, salvaguardare la sicurezza della memoria, ogni finestra creata con tale parametro ha un proprio processo dedicato.

Tramite quindi questa funzionalità, la libreria *Stargate* è in grado di forzare un processo per ogni nuova finestra widget. L'altra faccia della medaglia, tuttavia, è la maggior difficoltà di comunicazione tra le finestre in quanto non si possiede più alcun riferimento JavaScript alle finestre. Per tale motivo si sono studiate diverse strategie di comunicazione cross-page, descritte nella prossima sezione.

4.2 Comunicazione cross-page

Nel corso degli anni vi sono state diverse strategie per la comunicazione cross-page tra pagine web in diversi tab, per soddisfare casi d'uso quali fare il check-out in una nuova pagina protetta di PayPal e notificare la pagina principale del risultato. Un altro esempio d'uso è la possibilità di avere una chat in una pagina separata, che tuttavia scambi informazioni con l'applicazione principale. Ed infine ovviamente il caso d'uso in questione, ovvero estrarre componenti UI per i monitor multipli.

4.2.1 postMessage

Il metodo `window.postMessage(message)` permette la comunicazione sicura attraverso istanze di finestre ed altri tipi di oggetti cross-page, ad esempio tra una pagina ed il popup che ha creato.
`targetWindow.postMessage(message);`

* **targetWindow**: riferimento alla finestra che riceverà il messaggio. Alcuni esempi per ottenere tale riferimento sono:
  - `window.open()` crea una nuova finestra;

  - `window.opener` restituisce il riferimento alla finestra genitore che ha aperto la finestra corrente tramite il metodo precendete;

* **message**: dati da inviare all'altra finestra. I dati sono serializzati usando lo *structured clone algorithm* descritto successivamente, ma implica la possibilità di trasmettere una vasta varietà di oggetti in maniera safe senza doverli serializzare;

La finestra che riceve il messaggio può rimanere in ascolto attraverso la proprietà JavaScript `self.onmessage` che assegna alla propria pagina una funzione da invocare ogni volta che arriva un messaggio inviato da *postMessage*.

```
self.onmessage = function(message) {
    // Do something with message
}
```

4.2.2 Eventi Storage

Laddove l'API dedicata alla comunicazione cross-page *postMessage* non si possa utilizzare, ad esempio su browsers meno moderni o perché non è possibile ottenere un riferimento alla finestra, è possibile usare gli eventi dello `Storage`. I browsers forniscono difatti delle API per la lettura e scrittura di dati persistenti anche dopo la chiusura della pagina. Inoltre lo storage è condiviso tra tutte le pagine aventi lo stesso dominio.

È quindi dunque possibile usare tali API per comunicare tra le finestre.

Esempio di invio di dati:

```
// Salva nel proprio storage, che e' tuttavia condiviso tra
// tab dello stesso dominio
window.localStorage.setItem('stargate-msg', message)
```

Esempio di ascolto di dati, ove si rimane in ascolto dell'evento di modifica dello Storage:

```
window.addEventListener('storage', function(event) {
  const message = window.localStorage.getItem('stargate-msg')
})
```

È dunque palese che questo metodo sia un trick e non un'API nata allo scopo della comunicazione cross-page, tuttavia è stato utilizzato per anni anche a questo scopo. Un ulteriore aspetto inconveniente di questo approccio è il fatto che sia obbligatorio, a differenza di *postMessage*, serializzare i dati in formato JSON. Un confronto tra *structured clone algorithm* e serializzazione è fornito in seguito in questo capitolo.

4.2.3 Cookies

È infine possibile utilizzare i cookies per la comunicazione tra finestre, laddove nemmeno gli eventi `Storage` siano possibili. La finestra che invia il messaggio lo

serializza come stringa e lo scrive all'interno di un cookie del browser. La finestra ricevente è invece in ascolto tramite un timer che ogni tot millisecondi legge il cookie per capire se è stato cambiato.

Per diversi motivi quali limiti di dimensione dei cookie, difficoltà di lettura/scrittura, performance e scalabilità è una soluzione assolutamente sconsigliata.

Fortunatamente gli obiettivi obbligatori del progetto di stage richiedono il supporto solo di Google Chrome e Firefox, i quali supportano *postMessage*. I browser invece supportati dal prodotto *Route Manager*, ovvero fino ad Internet Explorer 11, supportano almeno gli eventi dello `Storage`, utilizzabili quindi come fallback.

4.2.4 Structured clone algorithm VS Serialization

Poiché la comunicazione è multi-finestra tra processi separati, è necessario purtroppo che i dati siano copiati da una finestra e l'altra per mantenersi sincronizzati.

Lo "Structured clone algorithm" è un algoritmo per la copia di oggetti JavaScript complessi ed è utilizzato internamento per il trasferimento di dati attraverso *postMessage* ed altre API JavaScript. Ciò che fa è realizzare una copia analizzando ricorsivamente l'oggetto input, ma mantenendo una mappa dei riferimenti visitati precedentemente per evitare di attraversare infinitamente strutture circolari.
Una struttura dati circolare consiste in un campo dell'oggetto il cui valore è il riferimento all'oggetto stesso in maniera diretta (`a -> a`) o indirettamente (`a -> b -> a`).

Vantaggi dello *Structured clone algorithm*:

* Supporto per strutture circolari;
* Supporto nativo per la copia di quasi qualsiasi tipo di oggetto JavaScript;
* Non vi è bisogno di convertire in un formato e ricostruire l'oggetto originale da tale formato, essendo quindi sia più performante che evitando di rischiare la perdita di informazioni durante la conversione.

Svantaggi dello *Structured clone algorithm*:

* Non è possibile clonare oggetti `Error` e `Function`;
* Supportato dai browsers che supportano *postMessage*

La serialization è invece una tecnica che consiste nel tradurre un oggetto JavaScript in un formato adatto per la trasmissione nelle rete o per il salvataggio. In JavaScript tale formato è una stringa JSON, comune anche ad altri linguaggi lato server.
Vantaggi della *Serialization*:

* Adatto per la trasmissione nella rete;

* Supportato da qualsiasi browser ed utilizza un formato comune ad altri linguaggi.

Svantaggi della *Serialization*:

* Non supporta strutture circolari di default;
* Supporta solo un sotto-insieme dei tipi di oggetti, in particolare solo i formati definiti dallo standard JSON: numeri, stringhe, booleani, oggetti letterali ed array. Non supporta ad esempio strutture dati quali `Map, Set, Date, ArrayBuffer`, etc.;
* Perdita di informazioni nella conversione oggetto <=> stringa JSON;
* Calcolo computazionale per conversione e ricostruzione dell'oggetto originale.

4.2.5 Conclusioni

Confrontando le diverse strategie per la comunicazione cross-page e la copia dei dati, è chiaro che la migliore sia l'utilizzo di *postMessage*, il quale sfrutta lo *Structured clone algorithm*. Difatti entrambi sono nati esattamente per uno scopo di comunicazione tra la pagina principale ed altre entità quali estensioni, altre finestre e Web Workers (spiegati in seguito).

È invece possibile utilizzare la tecnica degli *Eventi Storage* assieme alla *Serialization* come fallback della libreria *Stargate* qualora il browser di esecuzione non supporti la strategia *postMessage*. Tuttavia a causa delle nette differenze comportamentali tra *Structured clone algorithm* e *Serialization*, gli utilizzatori della libreria sono avvertiti delle possibili complicazioni. Posso quindi decidere se affidarsi al fallback, qualora non utilizzino alcuna struttura dati non supportata dalla *Serialization*, oppure disabilitare l'utilizzo di *Stargate* se il browser non è moderno.

Nel caso dell'azienda WorkWave per il prodotto *Route Manager*, è stata decisa proprio la seconda opzione e disattivare l'interfaccia UI che permette di aprire la mappa Google Maps esternamente.

4.3 Prima architettura Stargate

Alla luce delle precedenti informazioni, si illustra di seguito una prima architettura per la libreria *Stargate* e verrà ampliata passo per passo nei prossimi capitoli. Con la strategia *postMessage* è difatti possibile organizzare l'applicazione nel seguente modo, ove si instraura una comunicazione bilaterale in cui *Parent* trasferisce lo stato applicativo ed i vari *Widget* notificano di eventi.

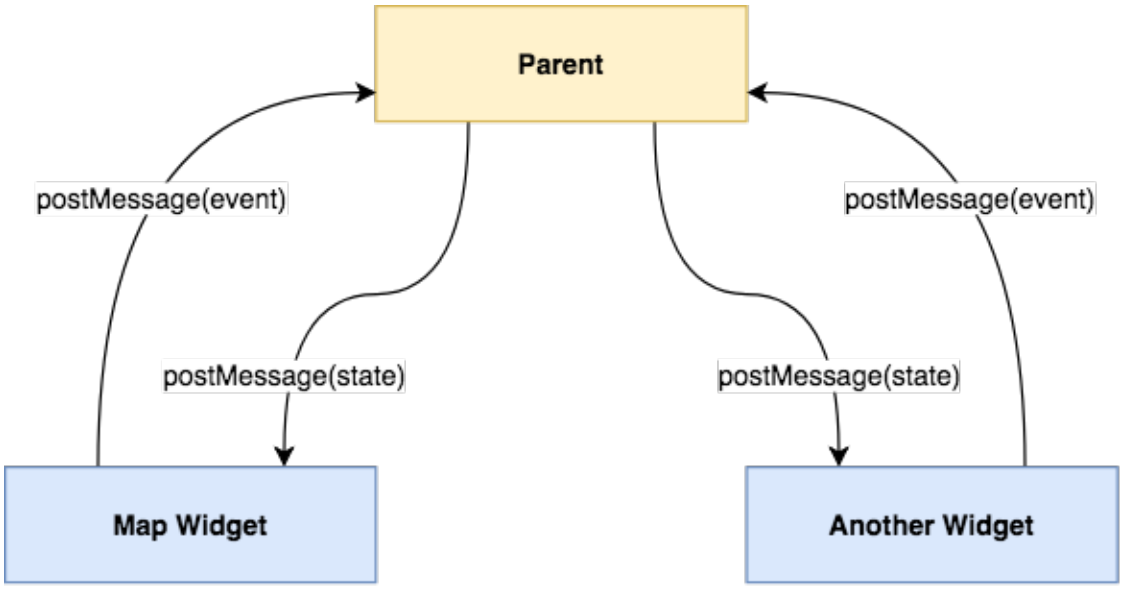


Figura 4.2: Prima architettura Stargate

Parent

* vive nella finestra principale ed è unica per sessione;
* si occupa della creazione e comunicazione con le finestre widget, mandando lo stato applicativo dell'applicazione contenente i dati necessari ai vari widget;
* gestisce gli eventi dei widget che necessitano di modificare lo stato applicativo dell'applicazione.

Widget

* Vive in una finestra widget, creata dal *Parent*
* Riceve lo stato applicativo ed utilizza i dati in essa per la corretta rappresentazione UI. Ogni volta che lo stato cambia, si aggiorna automaticamente anche la UI;
* Comunica al *Parent* qualsiasi evento che abbia effetti sullo stato applicativo, ad esempio un'interazione utente.

Capitolo 5

Architettura per computazione parallela

La precedente architettura risponde all'esigenza di stabilire una comunicazione cross-page, ma lascia irrisolti diversi obiettivi del progetto. In questo capitolo viene invece presentata un'evoluzione di tale architettura, al fine di poter migliorare le performance di *Stargate* ed offrire un sistema di gestione dei widget più indipendente. Dal punto di vista delle performance, non è difatti sufficiente eseguire le nuove finestre in un processo separato. Sarebbe infatti più ottimale che il calcolo dello **stato derivato** venga effettuato una volta sola per tutti i widget della stessa tipologia.

5.1 Stato derivato

Per *stato derivato* si intende l'insieme dei dati, calcolati a partire dallo stato applicativo dell'applicazione, necessari al widget per la correttezza esecuzione di tutte le sue funzionalità. La funzione, avente per input lo stato applicativo e per output lo stato derivato, viene convenzionalmente denominata **selector** ed ha una firma di tipo `State => DerivedState`.
Tutte le funzioni *selectors* devono inoltre essere pure, ovvero ritornare lo stesso risultato a parità di input e non avere effetti collaterali (*side-effects*), quali accesso/modifica a variabili non locali alla funzione, modifiche per riferimenti, accesso a I/O etc. Maggior informazioni sulle funzioni pure sono reperibili al seguente link: https://en.wikipedia.org/wiki/Pure_function.

Nel seguente esempio lo stato applicativo rappresenta un'applicazione che gestisce una lista di acquisti. Si immagini quindi di avere un widget UI che mostri il totale delle spese e che quindi necessiti di tale stato derivato.

```
interface Purchase {
    name: number
    price: number
}

interface State {
```

```
    purchases: Array<Purchase>
}

type DerivedState = number

function sumSelector(state: State): DerivedState {
    let sum: number = 0;

    for (let i = 0; i < state.purchases.length; i++) {
        sum += state.purchases[i].price;
    }

    return sum
}
```

Essendo funzioni pure, è possibile comporre *selectors* nella stessa maniera in cui si compongono funzioni matematica `f . g`, ottenendo *selectors* più complessi ma comunque modulari. Il *selector* finale di un componente UI può essere il risultato di decine di sotto-selectors, a loro volte composti da altri selectors.

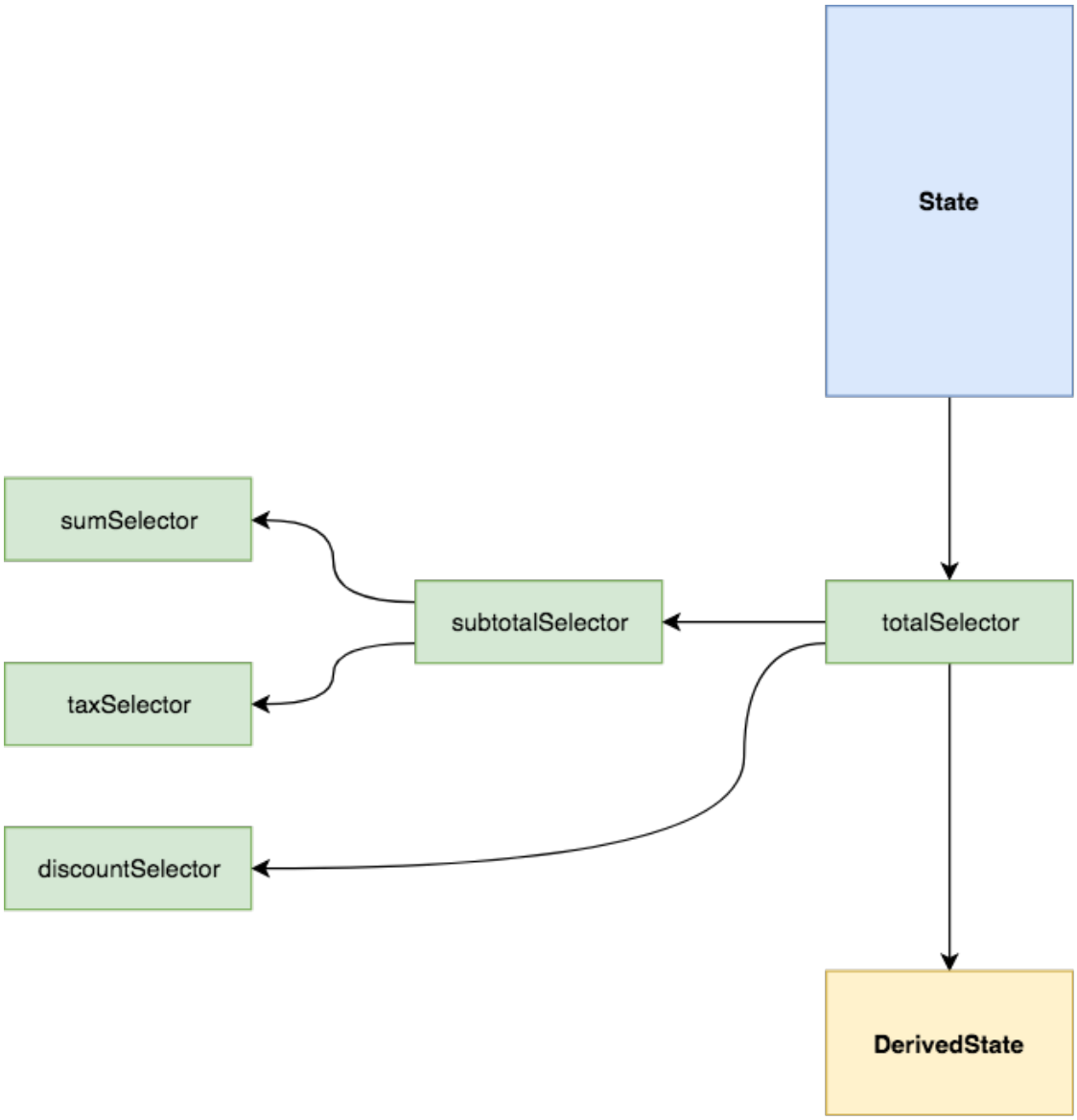


Figura 5.1: Esempio di composizione di selectors

È facile quindi immaginare che in un componente UI complesso quale una mappa dei veicoli e rotte, tali *selectors* siano molto complessi e richiedano un tempo di computazione non indifferente.

In particolare il codice JavaScript dell'applicazione principale è eseguito in un unico thread, per cui calcolare nell'applicazione padre lo stato derivato di diverse finestre con mappe potrebbe bloccare l'applicazione e renderla incapace di rispondere alle interazioni utenti fino al termine dei calcoli nei *selectors*.

Una possibile soluzione potrebbe essere delegare tale calcolo nelle finestre figlie, poiché vivono su processi dedicati. Sebbene sia accettabile, non è ottimale in quanto il calcolo dei *selectors* è ripetuto anche se è identico per due finestre aventi entrambe la stessa tipologia di widget, per cui esiste un'alternativa migliore.

L'ideale è difatti calcolare lo stato derivato per ciascun tipo di widget, ad esempio per tutte le finestre contenenti la mappa, una sola volta ad ogni modifica dello stato applicativo e propagare il risultato del calcolo ai widget. In tal modo l'onere computazionale è linearmente dipendente dal numero di **tipi di widget** attivi invece che dal numero totale di widget. Tuttavia tale calcolo allo stesso tempo non può avere nella finestra padre per le ragioni descritte sopra.

5.2 Web Worker

Un *Web Worker* è un modo per una pagina web di eseguire del codice in un thread background, in grado di eseguire attività senza interferire con l'interfaccia utente. Nello specifico è un oggetto della classe `Worker`, creato passando come parametro il file che contiene il codice da eseguire nel thread separato. Tale thread non avrà alcun riferimento di memoria in comune con il main thread della pagina e viene considerato come un contesto di esecuzione separato.

È possibile eseguire qualsiasi tipo di codice all'interno del thread worker, con alcune eccezioni tuttavia. Ad esempio non è possibile manipolare direttamente i nodi HTML della pagina web o accedere ad API CSS/HTML. In generale un *Worker* è da considerare un thread di calcolo computazione e non di manipolazione della pagina.

Un aspetto positivo è invece il sistema di comunicazione tra il worker and il thread principale dell'applicazione web, in quanto avviene anch'esso via *postMessage* come tra finestre. Infine i *Web Workers* possono attivare nuovi workers per delegare ulteriormente del lavoro in nuovi thread.
È quindi chiaro che un *Web Worker* è il candidato ideale in *Stargate* per l'esecuzioni dei *selectors*. Nello specifico si desidera calcolare in un thread apposito lo stato derivato per ciascuna tipologia di widget, salvarla in cache ed inviarla a tutti i widget attivi di tale tipologia. Salvando in cache, è possibile renderizzare istantaneamente un widget alla sua apertura in quanto lo stato derivato necessario è stato già calcolato precedentemente.
Inoltre, poiché i calcoli avvengono in un thread di background, non vi sono impatti negativi nelle performance sia dell'applicazione principale che nelle finestre widget.

Purtroppo l'utilizzo del *Web Worker* genera un nuovo problema: le nuove finestre devono essere aperte comunque dalla pagina principale in quanto le relative API non sono disponibili nel Worker, ma non è possibile per questi ottenerne i riferimenti per poter usare `widgetWindow.postMessage(derivedState)`.

5.3 BroadcastChannel

Un *BroadcastChannel* è un canale di comunicazione broadcast tra diversi "contesti" del browser (ovvero finestra, tabs o workers) provenienti dallo stesso sito. È difatti disponibile anche da parte dei *Web Workers*.

Creando un *BroadcastChannel*, il quale rimane in ascolto del sottostante canale, si è in grado di inviare messaggi attraverso di esso usando `channel.postMessage(data)`, quindi usando il riferimento all'istanza di *BroadcastChannel* invece che della finestra. Allo stesso tempo è possibile rimanere in ascolto di tutti i messaggi inviati attraverso il canale ed ogni messaggio è inviato a tutti coloro in ascolto, ovvero in broadcast.

È possibile quindi comunicare tra le parti senza riferimenti reciproci. Ogni parte dell'architettura è in grado di sottoscriversi al canale *BroadcastChannel* ed avere una comunicazione bi-direzionale (*full-duplex*) verso tutte le altre.

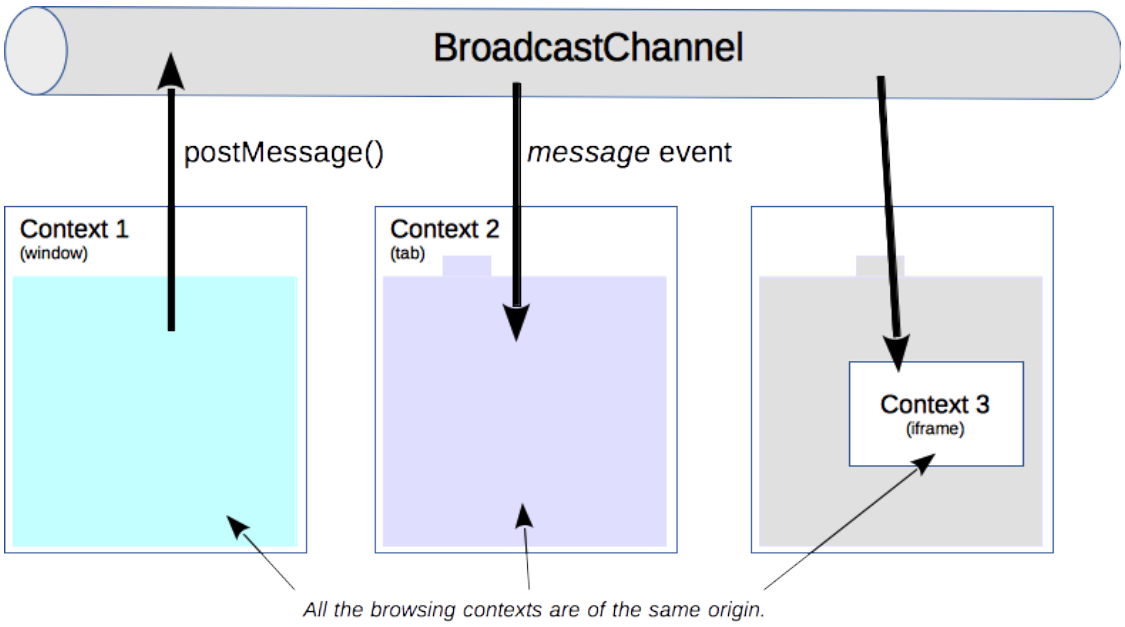


Figura 5.2: Esempio di funzionamento del BroadcastChannel

5.4 Evoluzione architettura Stargate

Alla luce delle conoscenze sui *Web Workers* e sul *BroadcastChannel*, si presenta la nuova architettura del progetto *Stargate*.

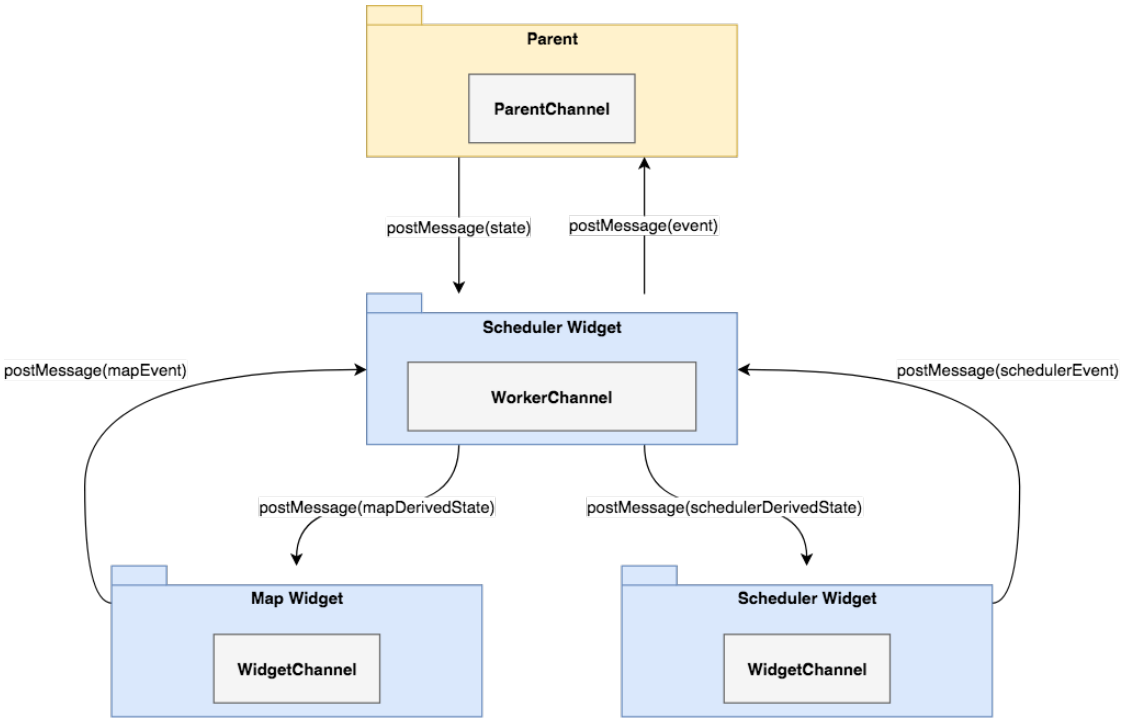


Figura 5.3: Architettura Stargate evoluta con Web Worker e BroadcastChannel

Le classi *Parent* e *Widget* sono divenute ora packages, in quanto contengono altre classi sottostanti tra cui quelle che gestiscono la comunicazione tramite *BroadcastChannel*.

ParentChannel

* vive nella finestra principale ed è unica per sessione;
* gestisce il canale di comunicazione *BroadcastChannel* verso il *Web Worker*
* si occupa della creazione delle finestre widget, ma manda lo stato applicativo dell'applicazione al *Web Worker* e **non** conosce le esigenze dei widget rispetto all'architettura precedente;
* gestisce gli eventi dei widget che necessitano di modificare lo stato applicativo dell'applicazione.

WorkerChannel

* Vive nel *Web Worker*, creato dal *ParentChannel*;
* Fa da intermediario per la comunicazione tra il *ParentChannel* e tutti i diversi *WidgetChannel*;
* Riceve lo stato applicativo dal *ParentChannel* e fornisce gli stati derivati a ciascun widget. Lo stato derivato, ad ogni modifica dello stato applicativo, è calcolato una volta sola per ogni tipologia di widget come descritto nelle sezioni precedenti;

* Propaga gli eventi dai *WidgetChannels* verso il *ParentChannel*.

WidgetChannel

* Vive in una finestra widget, creata dal *Parent*
* Riceve lo stato applicativo dal *Web Worker* ed utilizza i dati in essa per la corretta rappresentazione UI. Ogni volta che lo stato cambia, si aggiorna automaticamente anche la UI;
* Comunica al *Web Worker* (e non direttamente al Parent) qualsiasi evento che abbia effetti sullo stato applicativo, ad esempio un'interazione utente.

Capitolo 6

Diff & patch

Uno degli obiettivi primari del progetto *Stargate* è il supporto prestazionale a grosse mole di dati, dell'ordine di diversi MByte, in continuo aggiornamento ed uso. Finora l'architettura ottimizza il calcolo dello stato derivato all'interno dei *Web Workers* e tutte le operazioni dei widgets attraverso un processo dedicato del Sistema Operativo.

Tuttavia entrambe le soluzioni non sono in grado di ottimizzare un potenziale collo-di-bottiglia delle performance: la trasmissione di grosse quantità di dati. Attualmente, ad ogni modifica dello stato applicativo, questi va inviato interamente al *Web Worker* affinché possa calcolare gli stati derivati dei widgets, che a loro volta sono poi inviati ai widgets appunto.

Se da un lato tale flusso non abbia impatti negativi sulla pagina principale grazie all'uso di *Web Worker* e processi dedicati, dall'altro potrebbe causare un ritardo nei tempi di aggiornamento dell'Interfaccia Utente nelle finestre widget. Ciò causerebbe una cattiva percezione dell'utente nei confronti della fluidità d'uso dei componenti aperti nelle nuove finestre.

D'altra parte è inutilmente dispendioso il continuo invio di tutto lo stato applicativo, che deve essere copiato/serializzato da una parte all'altra. Per tale motivo è stato introdotto un sistema di *diff & patch* dello stato applicativo.

6.1 Flusso diff & patch

1. All'avvio dell'applicazione, viene inviato lo stato iniziale completo. Per completo si intende che non vi possono essere campi mancati rispetto alla sua interfaccia, sebbene possano avere come valore `null`;

2. Ad ogni successiva modifica dello stato applicativo, viene calcolata la differenza (ovvero il **delta** Δ) tra il nuovo stato applicativo e quello precedente. Questa operazione viene chiamata **diffing**;

3. Il *delta* contiene tutte le informazioni per ricostruire il nuovo stato a partire da quello vecchio. Tale *delta* viene quindi inviato al *Web Worker*;

4. Il *Web Worker* applica il *delta* sullo stato applicativo che ha in memoria, ottenendo il nuovo stato;

5. Vengono ricalcolati gli stati derivati dei widgets ed un simile processo di *diff & patch* viene effettuato per essi. Difatti anche i widgets ricevono lo stato derivato intero alla loro apertura, ma i successivi aggiornamenti contengono solo i *delta*.

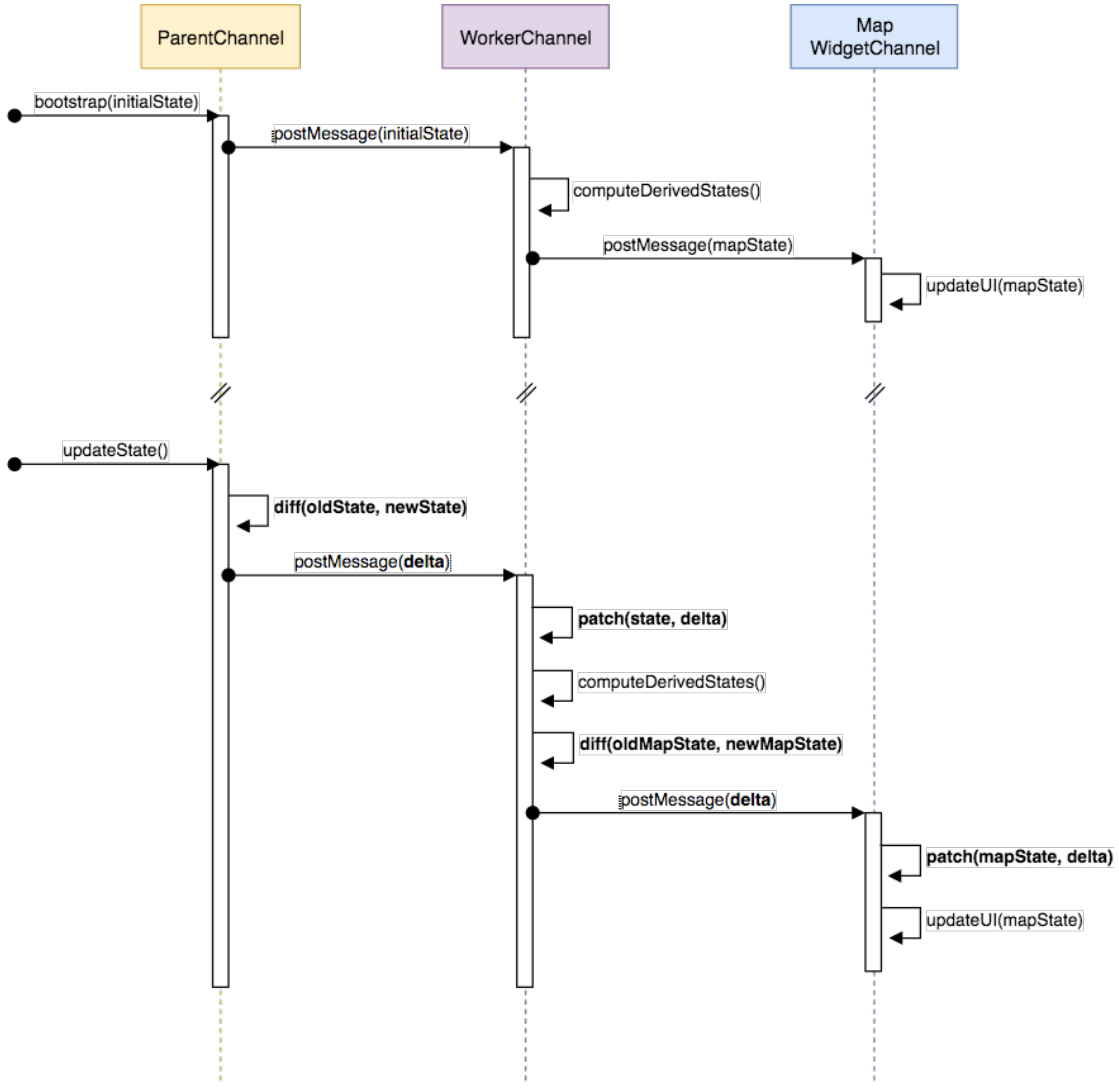


Figura 6.1: Sequenza delle chiamate per il flusso diff & patch

Si assicura in tal modo che le dimensioni del carico di trasmissione siano dipendenti unicamente dal numero di modifiche effettuate. Quest'ultime saranno frequenti ma è ragionevole presuporre che ogni modifica statisticamente cambi solo una piccola porzione dello stato applicativo, risultando quindi in piccoli *delta*.

6.2 Ottimizzazione per immutable state

Essendo inoltre *Stargate* rivolto verso applicazioni web moderne, in particolare in React & Redux, l'algoritmo interno di *diff & patch* assume anche che modifiche allo oggetto stato applicativo non vengano fate per riferimento, ma bensì ritornino una copia avente un nuovo riferimento e le cui proprietà sono anch'esse nuove laddove siano modificate.

6.2.1 Strutture dati persistenti

Una modifica per riferimento è una mutazione diretta ad una struttura dati (un oggetto) esistente senza crearne una copia. Una modifica **immutable** invece si assicura preventivamente di fare una copia, non profonda, dell'oggetto in qualsiasi caso in cui una proprietà cambi.

Ad esempio si assuma di voler modificare la proprietà `xs.d.g.f` sostituendo 1 con `e: 1` .

```
const xs = {
  d: {
    b: {
      a: 1,
      c: 1
    },
    g: {
      f: 1, // <== da sostituire con { e: 1 }
      h: 1
    }
  }
}
```

Modificarlo per riferimento sarebbe eseguire l'istruzione JavaScript `xs.d.g.f = e: 1` , in quanto viene modificato il campo annidato `f`, ma sia `xs` che i suoi sotto-oggetti `d, g, f` mantengono lo stesso riferimento rispetto a prima.
Una modifica immutabile invece ritorna un nuovo oggetto `ys`, ove sia `ys` che i suoi sotto-oggetti `d', g', f'` hanno nuovi riferimenti mentre `b` è rimasto invariato poiché non modificato.

```
const ys = {          // <== nuovo riferimento ys
  d: {                // <== nuovo riferimento d
    b: {
      a: 1,
      c: 1
    },
    g: {              // <== nuovo riferimento g
      f: { e: 1 },    // <== nuovo riferimento f
      h: 1
    }
  }
}
```

Una possibile rappresentazione della precedente modifica *immutable* è il seguente albero, ove il nuovo oggetto `ys` possiede sia riferimenti a nuovi oggetti (`d', g', f'` modificati), che a vecchi (`b, c, , h` invariati).

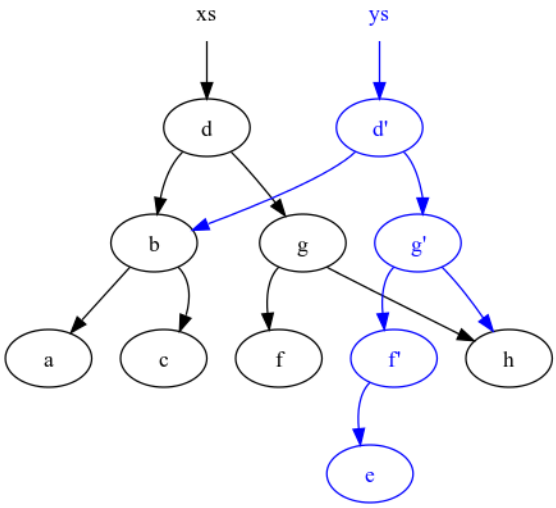


Figura 6.2: Albero di `ys`

Strutture dati che preservano sempre la propria precedente versione in caso di modifica si chiamano **strutture dati persistenti** [1] e sono immutabili in quanto le loro operazioni non aggiornano direttamente la struttura, ma bensì generano sempre una nuova aggiornata.
Tali strutture dati sono fondamentali nei linguaggi di programmazione funzionali, ma nei recenti anni sono divenuti fondamentali anche per linguaggio non puramente funzionali come JavaScript in quanto portano a diversi vantaggi come la diminuizione di *side-effects*.

6.2.2 Ottimizzazione dell'algoritmo

Nel caso specifico dell'algoritmo *diff & patch*, le strutture immutabili permettono di ottimizzare il calcolo del delta in quanto è sufficiente controllare per riferimento se un campo è cambiato rispetto a prima, senza dover controllare profondamente i valori. Ad esempio confrontando `xs, ys` dall'esempio precedente, l'algoritmo può evitare di proseguire il calcolo del delta per l'intero sotto-albero `b` in quanto il riferimento non è cambiato. Se invece viene rilevato un diverso riferimento, l'algoritmo continua lavorando sul sotto-albero.

Per uno stato applicativo di notevoli dimensioni, questa ottimizzazione permette di migliorare notevolmente i tempi di *diffing* dell'algoritmo, rendendo il costo linearmente dipendente al numero di modifiche invece che di dimensioni della struttura. Un controllo di riferimento per sapere se un sotto-albero è cambiato ha difatti tempo costante $O(1)$, altrimenti sarebbe direttamenteproporzionale $\Theta(n)$ al numero di campi annidati.

[1] https://en.wikipedia.org/wiki/Persistent_data_structure

Capitolo 7

Architettura per il Context

L'ultimo step di evoluzione del progetto *Stargate* consiste nell'astrazione **da stato applicativo a Context**, ovvero contesto di esecuzione dei componenti UI. Sebbene difatti lo stato applicativo rappresenti nella maggioranza dei casi tutto ciò di cui un widget ha bisogno da parte dell'applicazione principale, in alcuni casi è necessario poter accedere anche ad altri oggetti ed addirittura richiamare metodi dalla finestra padre.

Si supponga difatti che il widget mappa utilizzi un'istanza singleton della classe `GoogleMaps`, responsabile dei calcoli geografici attraverso le API di Google Maps e condivisa a molteplici componenti UI oltre alla mappa.
In questo caso, il widget mappa ha dunque bisogno del seguente "contesto" per poter correttamente funzionare anche al di fuori della pagina principale:

```
interface MapContext {
  googleMaps: GoogleMaps
  state: State
}
```

Si può notare come il precedente stato applicativo `State`, sia ora una delle proprietà del *Context* della mappa e sicuramente la più importante. Tuttavia il passaggio da stato applicativo ad un concetto più generale di "contesto", permette anche l'aggiunta all'interfaccia di un campo per l'istanza singleton di `GoogleMaps`, senza dover inserire questi nello stato applicativo di cui non fa parte logicamente.
A livello di architettura fortunatamente vi è poco da modificare, in quanto si tratta solamente di ampliare il concetto di stato applicativo a *Context* e quindi di rinominare i termini. Si parla quindi di **Context e DerivedContext** invece che *State e DerivedState*.

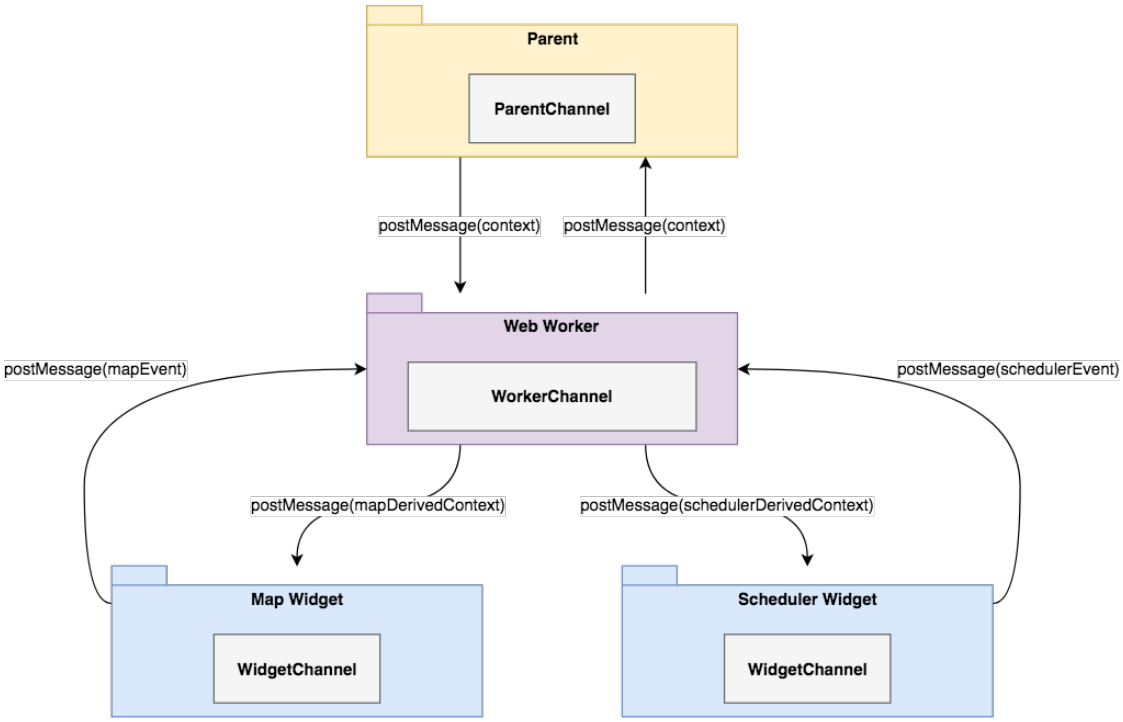


Figura 7.1: Architettura per il Context

7.1 Remote Procedure Call (RPC)

L'introduzione del *Context* porta alla luce una nuova sfida: chiamate di metodi nella finestra padre da parte di widgets. Lo *State* è infatti un'insieme di dati di sola lettura, ma l'istanza `GoogleMaps` viene invece utilizzata richiamandone metodi con passaggio di parametri e valori di ritorno attesi.

Tuttavia l'applicazione principale, il *Web Worker* ed i widget non hanno alcuna condivisione di memoria per cui apparentemente sembrerebbe impossibile per una finestra figlia richiamare un metodo di un oggetto esistente solo nel padre. Nel caso di istanze singleton come *GoogleMaps* non è possibile che ogni widget abbia la sua copia dell'oggetto, poiché violerebbe proprio la definizione di singleton.

In questi casi è necessario utilizzare **Remote Procedure Call (RPC)**, ovvero chiamate chiamate di procedure che avvengono in uno spazio di memoria diverso (solitamente un altro dispositivo della rete), ma in maniera trasparente come normali chiamate locali a procedure/metodi. È desiderabile infatti che il chiamante del *metodo RPC* non conosca i dettagli implementativi della chiamata remota sottostante.
In Java vi è un'implementazione delle chiamate RPC attraverso le *Remote method invocations (RMI)*.

Nel caso del progetto *Stargate*, quando il widget esegue una chiamata di un metodo del *Context*, ad esempio `context.googleMaps.getDistance(a, b)`, in realtà succede il seguente:

1. Viene creato un messaggio contenente informazioni sulla chiamata, quali la proprietà del *Context*, il nome del metodo ed i parametri;

2. Il messaggio viene inviato al *Web Worker*;

3. Il messaggio viene ritrasmesso al *ParentChannel*;

4. Viene invocato l'effettivo metodo nella finestra padre, passando i parametri provenienti dal widget;

5. Viene generato un messaggio contenente le informazioni della chiamata con l'eventuale valore di ritorno;

6. Il messaggio di risposta viene inviato al *Web Worker* e ritrasmesso al *WidgetChannel*;

7. Il chiamante riceve il valore di ritorno e può proseguire con il resto della procedura

Per ovvi motivi di performance, una volta chiamato il metodo, la finestra widget mette in pausa la procedura asincrona e prosegue con altre attività evitando di rimanere bloccata in attesa della risposta. L'inconveniente è difatti che tutte le chiamate di metodi nel *Context* diventano asincrone, anche qualora fossero originariamente sincrone.

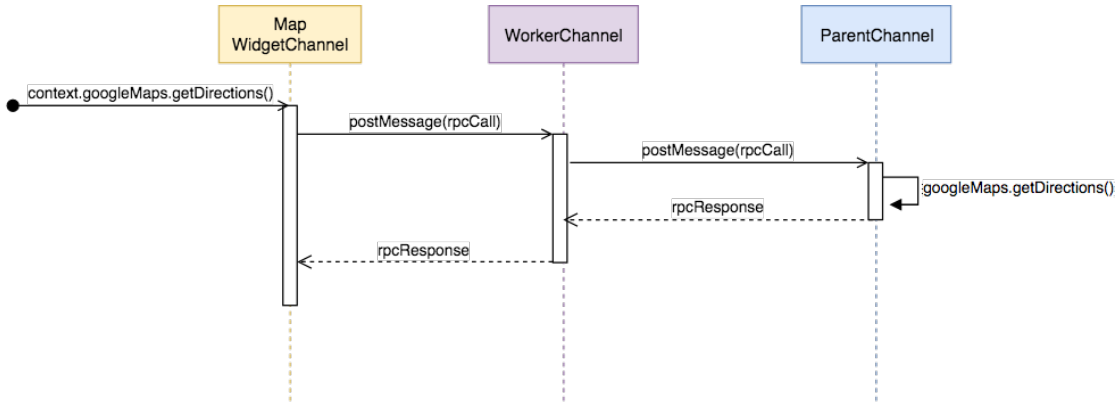


Figura 7.2: Sequenza per una chiamata RPC del context

7.1.1 Implementazione RPC tramite Proxy

Ai fini di rendere la chiamata RPC trasparente nei confronti dell'utente, gli oggetti al primo livello di annidamento del *Context* sono rimpiazzati da equivalenti *Proxy*, che funzionano esattamente come gli originali ma racchiudono al loro interno la logica di comunicazione descritta poco prima.

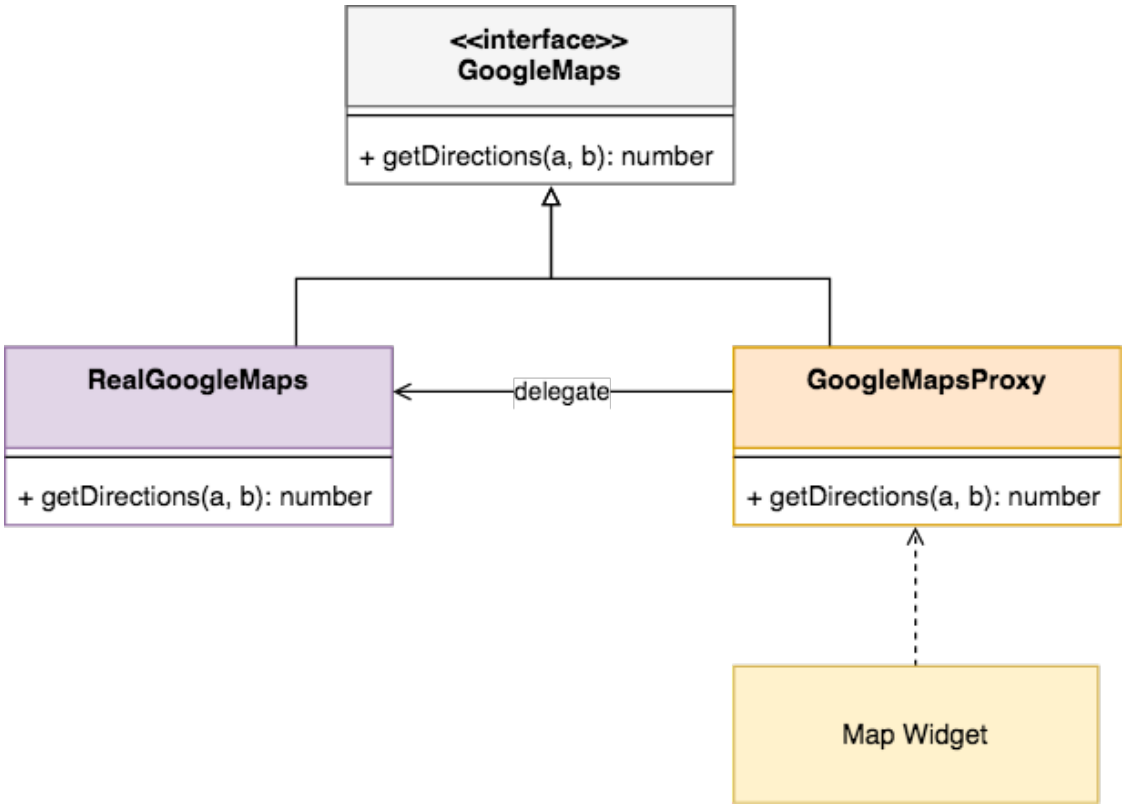


Figura 7.3: Esempio di Proxy per GoogleMaps

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia

* Multi-process architecture https://blog.chromium.org/2008/09/multi-process-architect
html