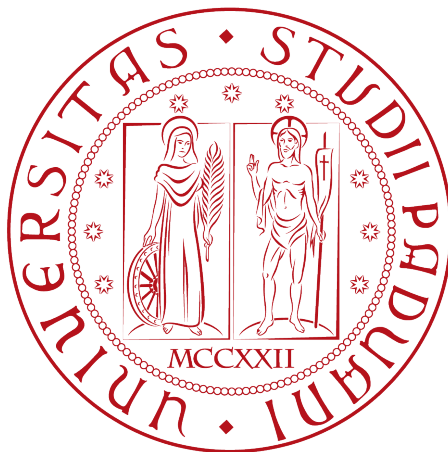


Università degli Studi di Padova

**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

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 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 integrare tali tecnologie nel prodotto **WorkWave Route Manager**. Quest'ultima applicazione offre ai propri clienti e permette di pianificare, dirigere, tracciare e analizzare le rotte dei propri veicoli in tempo reale.

Nello specifico, l'obiettivo principale è poter estrarre porzioni di interfaccia grafica dall'applicazione in una nuova pagina web autonoma, ma sincronizzata a livello di stato applicativo. L'integrazione ha avuto difatti lo scopo di permettere la visualizzazione della mappa Google Maps, ricca di rotte ed veicoli, su un monitor separato full-screen.

Lo sviluppo ha portato dunque alla nascita della libreria denominata **Stargate**, ispirato dall'omonima serie tv di fantascienza.

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. Maggiori informazioni sulle tecnologie sono disponibili in Appendice §A di questo documento.

Ringraziamenti

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

Vorrei inoltre dare un caloroso abbraccio ad Elisa per l'affetto datomi 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 decisioni.

Padova, Settembre 2018

Giovanni Jiayi Hu

Indice

Indice	iii
Elenco delle figure	v
Elenco delle tabelle	vi
1 L'azienda	1
1.1 Descrizione	1
1.2 Servizi offerti	1
1.3 Route Manager	2
2 Lo stage	4
2.1 Presentazione del progetto Stargate	4
2.2 Obiettivi	6
2.3 Pianificazione	7
2.4 Analisi preventiva dei rischi	9
2.5 Aspettative aziendali	9
2.6 Aspettative personali	10
3 Processi e metodologie	11
3.1 Accertamento di Qualità	11
3.1.1 Pull Request	11
3.2 Gestione della configurazione	12
3.2.1 Versionamento	12
3.2.2 Ambiente di verifica	12
3.2.3 Ambiente di rilascio	13
3.3 Gestione di Progetto	13
3.3.1 Standup	13
3.3.2 Ticketing	13
4 Architettura per la comunicazione cross-page	14
4.1 Architettura multi-processi	14
4.1.1 Cosa fa ogni processo?	15
4.1.2 Strategie multi-process	16
4.1.3 Come forzare l'uso di un nuovo processo	17

4.2	Comunicazione cross-page	18
4.2.1	postMessage	18
4.2.2	Eventi Storage	19
4.2.3	Cookies	19
4.2.4	Structured clone algorithm VS serialization	20
4.2.5	Conclusioni	21
4.3	Prima architettura Stargate	22
5	Architettura per computazione parallela	24
5.1	Stato derivato	24
5.2	Web Worker	26
5.3	BroadcastChannel	27
5.4	Evoluzione architettura Stargate	28
6	Diff & patch	30
6.1	Flusso diff & patch	30
6.2	Ottimizzazione per immutable state	32
6.2.1	Strutture dati persistenti	32
6.2.2	Ottimizzazione dell'algoritmo	33
7	Architettura per il Context	35
7.1	Remote Procedure Call (RPC)	36
7.1.1	Implementazione RPC tramite Proxy	37
8	Integrazione in Route Manager	39
8.1	Google Maps	39
8.1.1	Google Maps serialization	40
8.1.2	Google Maps diffing	40
8.1.3	ChannelStrategy	41
9	Valutazione retrospettiva	42
9.1	Raggiungimento degli obiettivi	42
9.2	Resoconto dell'analisi dei rischi	44
9.3	Conoscenze acquisite	44
9.3.1	Ricerca & Sviluppo	44
9.3.2	Tecnologie apprese	45
9.4	Valutazione personale	45
A	Tecnologie utilizzate	46
A.1	TypeScript	46
A.2	React	47
A.3	Redux	48
	Bibliografia	51

Elenco delle figure

11	Logo dell'azienda WorkWave	1
12	Logo WorkWave Route Manager	2
13	WorkWave Route Manager	3
14	WorkWave Route Manager - Scheduler degli ordini	3
21	Logo della libreria Stargate	4
22	WorkWave Route Manager con Google Maps in una nuova finestra attraverso Stargate	6
23	Diagramma Gantt della pianificazione	8
31	Flusso di una Pull Request	11
32	Jenkins	12
33	Amazon Cloudfront	13
34	CA Agile Central	13
41	Architettura multi-process in Chrome	15
42	Controllo per estrarre la mappa in una nuova finestra	22
43	Prima architettura Stargate	22
51	Esempio di composizione di selectors	25
52	Comunicazione tra la pagine web ed il Web Worker	26
53	Esempio di funzionamento del BroadcastChannel	28
54	Architettura Stargate evoluta con Web Worker e Broadcast-Channel	28
61	Sequenza delle chiamate per il flusso diff & patch	31
62	Albero di ys, ove le parti blu hanno nuovi riferimenti	33
71	Architettura per il Context	36
72	Sequenza per una chiamata RPC del context	37
73	Esempio del pattern Proxy per GoogleMaps	38
81	Flusso della serializzazione per Google Maps	40
82	Esempio di nodo LatLng sostituito con oggetto avente campi comparabili	41
83	Strategy Pattern per la configurazione di Stargate	41

A1	TypeScript	46
A2	React	47
A3	Redux	48
A4	Flusso unidirezionale in Redux	49

Elenco delle tabelle

21	Tabella degli obiettivi	7
22	Tabella della suddivisione delle ore	8
23	Tabella dell'analisi dei rischi	9
91	Tabella raggiungimento degli obiettivi	43
92	Tabella dell'analisi dei rischi	44

1 L'azienda

1.1 DESCRIZIONE

WorkWave, una divisione di IFS, è una società americana fondata nel 1984 che fornisce soluzioni di Field Service Management e che connette ogni aspetto del business attraverso le sue piattaforme unificate e di facile uso. L'insieme delle soluzioni della compagnia permettono facilmente ai professionisti del settore di assegnare ed automatizzare attività di vendita e marketing, migliorando l'efficienza ed incrementando il monitoraggio sulle operazioni sul campo attraverso soluzioni mobile.

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



Figura 11: Logo dell'azienda WorkWave

1.2 SERVIZI OFFERTI

WorkWave aiuta aziende nel campo Field Service Management, oltre che industrie di trasporti e logistica, a mitigare gli aspetti dolorosi che incontrano ogni giorno. Consente loro infatti 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 funzionalità con depositi, fatturazione ed altri servizi back-office.

WorkWave fornisce a tale scopo 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 prodotti correlati all'attività di stage:

- * *WorkWave Service*: è un servizio software che consente di 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 proprie 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.

1.3 ROUTE MANAGER



Figura 12: Logo WorkWave Route Manager

WorkWave Italy è la sede centrale europea di WorkWave, dove sono concentrati gli sviluppi dell'algoritmo di routing e del prodotto Route Manager, contesto di sviluppo delle attività descritte in questo documento.

WorkWave Route Manager invece automatizza la pianificazione delle rotte per migliorare l'efficienza e la comunicazione tra l'amministrazione ed i guidatori dei veicoli, in maniera completamente customizzabile tramite le sue API.

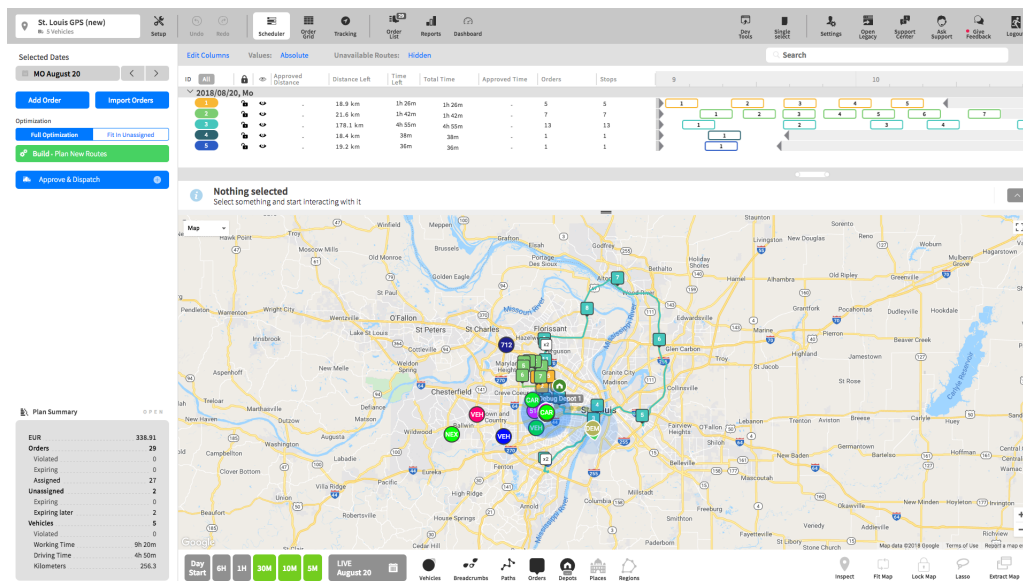


Figura 13: WorkWave Route Manager

In particolare, per quanto riguarda routing & scheduling, rende possibile navigare attraverso 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 delle modifiche sul numero di ordini possibili per i veicoli disponibili, il tempo stimato di completamento delle attività e comparare il costo per miglio.

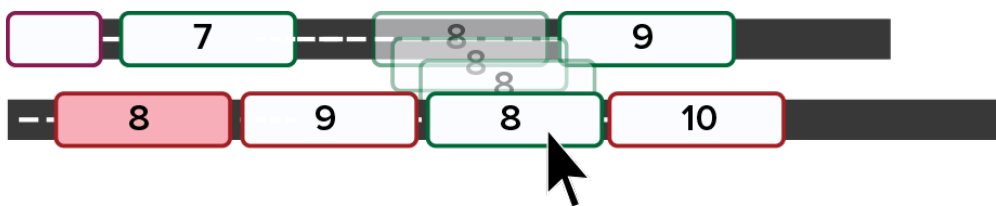


Figura 14: WorkWave Route Manager - Scheduler degli ordini

2 Lo stage

2.1 PRESENTAZIONE DEL PROGETTO STARGATE



Figura 21: Logo della libreria 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 vitro la possibilità di spostare un componente UI React dalla finestra *padre* verso una nuova finestra *figlia*, mantenendo lo stato consistente. Tali componenti UI sono identificati col termine *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 consistentemente. 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 la stessa classe istanziata molteplici volte ma indipendentemente 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 ed invece una singola rotta real-time su un'altra. Inoltre permette di usufruire 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 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 usufruire 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 widget, ciascuno di questi deve essere sincronizzato col rispettivo padre e non deve creare effetti con l'altra applicazione principale.

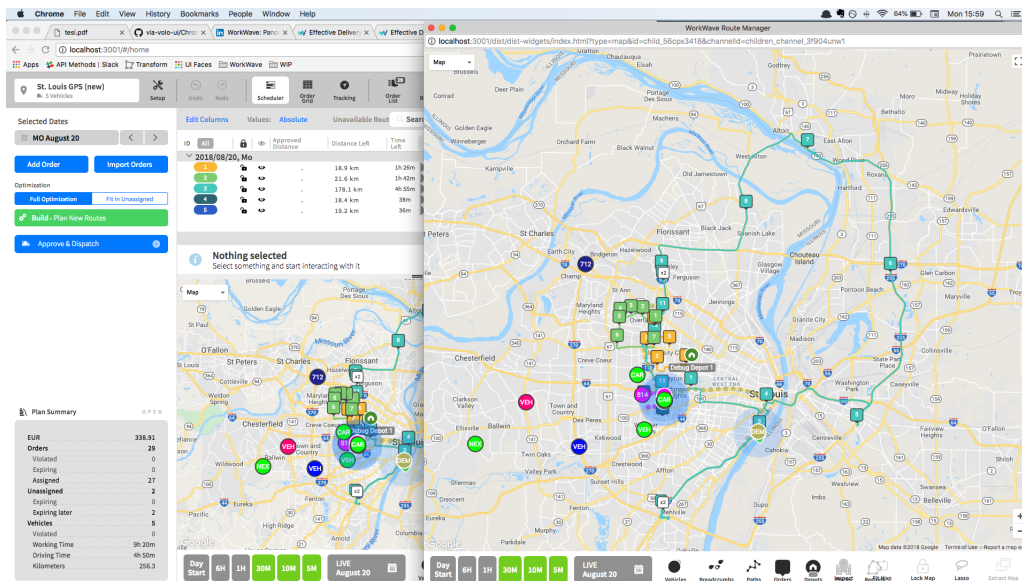


Figura 22: WorkWave Route Manager con Google Maps in una nuova finestra attraverso Stargate

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 <i>Route Manager</i> per supportare i widget
O07	Produzione della documentazione d'uso di <i>Stargate</i>
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 librerie JavaScript non React, ad esempio Angular 2
Facoltativi	
O04	Supporto per i browsers Internet Explorer v11 e Edge v12

Tabella 21: Tabella degli obiettivi

2.3 PIANIFICAZIONE

In accordo col tutor aziendale 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:** Implementazione dell'integrazione del widget Google Map in *Route Manager* ed evoluzione della libreria;
- * **Fase 4:** Validazione e stesura documentazione.

Non vi è stata invece 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 architetturale
104	Integrazione in <i>Route Manager</i> del widget Google Maps
24	Gestione configurazione per supportare widget
16	Validazione e Collaudo
8	Refactoring prima del rilascio in produzione
8	Stesura documentazione
Totale: 312 ore	

Tabella 22: Tabella della suddivisione delle ore

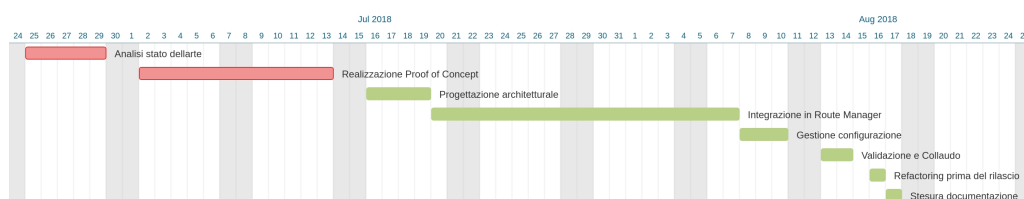


Figura 23: Diagramma Gantt della pianificazione

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 un'efficiente 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 <i>Route Manager</i> , 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 <i>Stargate</i> oppure in <i>Route Manager</i>	Occorrenza: Alta Pericolosità: Alta

Tabella 23: 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 mantenendo contemporaneamente 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 ed instaurare una conoscenza che potrà poi essere portata avanti tramite apertura di una posizione lavorativa.

2.6 ASPETTATIVE PERSONALI

Ho intrapreso questo stage con l'obiettivo primario di venire in contatto con un ambiente di lavoro focalizzato sul team e sulla qualità dei loro prodotti. Ritengo difatti fondamentale apprendere non solo conoscenze tecniche, ma anche skill sociali e sul modo di lavorare. Grazie a WorkWave ho avuto quindi l'opportunità di conoscere cosa significhi 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 [standup](#) e sono stati spiegati gli strumenti di pianificazione e comunicazione interni.

Ritengo inoltre indispensabile confrontarmi con colleghi molto più esperti e con maggiore esperienza, in particolare il tutor Matteo Ronchi, che ha condiviso pazientemente le ragioni dietro decisioni architetturali apprese grazie alla sua esperienza in progetti passati. Infine è molto istruttivo rendersi conto del 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.

Processi e metodologie

3

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

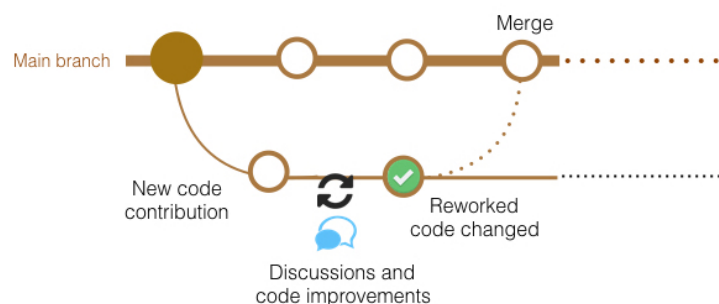


Figura 31: Flusso di una Pull Request

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.

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 fase 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 32: 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 33: 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 nei diversi Paesi del mondo.

3.3 GESTIONE DI PROGETTO

3.3.1 Standup

Il team si incontra quotidianamente per lo standup, 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 eventuali criticità, permettendo anche di trasmettere maggiore consapevolezza e conoscenza del progetto ai diversi partecipanti.

3.3.2 Ticketing

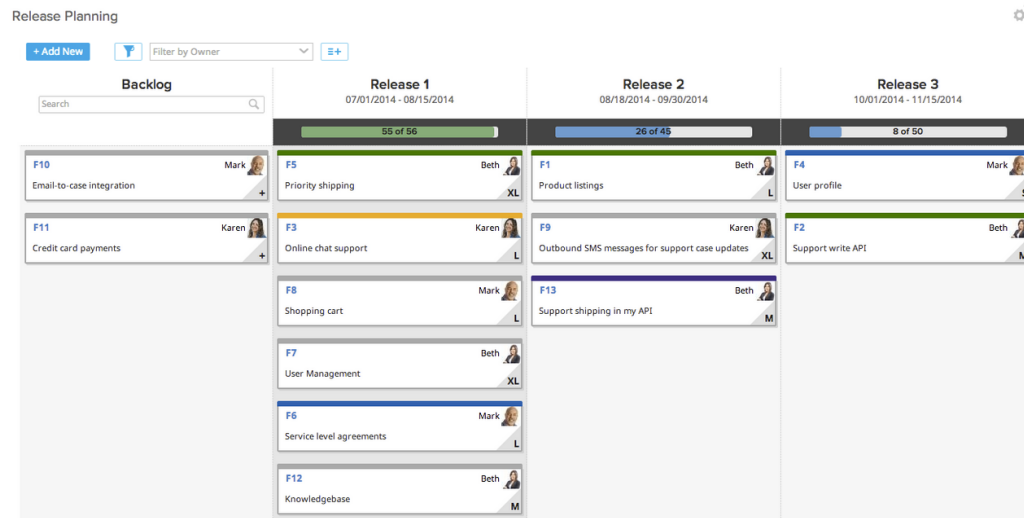


Figura 34: CA Agile Central

L'azienda utilizza lo strumento CA Agile Central ¹ per la gestione del ticketing, assegnando un ticket per ogni attività atomica. Ciò permette un'agevolazione nell'allineamento nel team sulle attività e gli obiettivi giorno per giorno.

Sono inoltre assegnate diverse priorità alle task al fine di garantire il completamento di quelle con maggior valore. Oltre a ciò è possibile avere un quadro complessito del progresso, delle dipendenze e dell'allineamento della pianificazione durante ogni **sprint**.

¹ <https://www.ca.com/it/products/ca-agile-central.html>

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 semplicistica della 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* apra 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 dinamico. Per tale motivo, i browsers 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 similmente a 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 browsers i quali pongano tutto il lavoro in un processo affrontano seri problemi di robustezza, responsività 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?

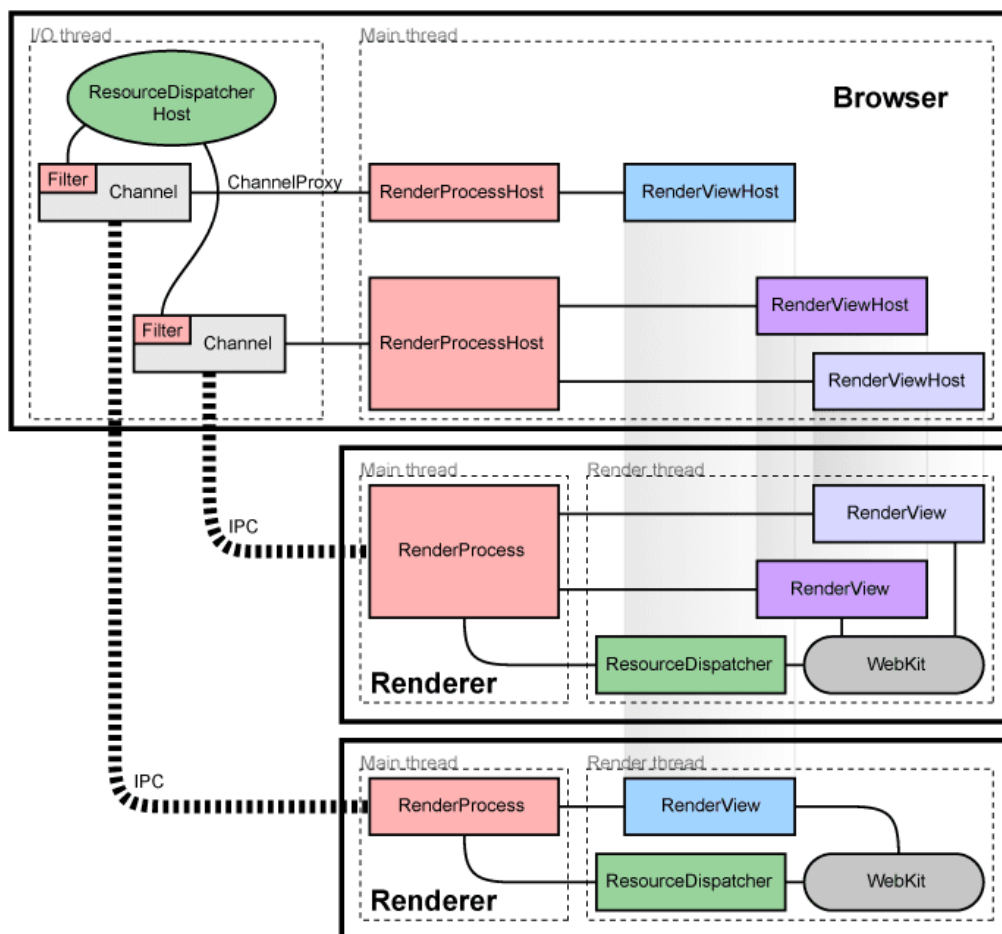


Figura 41: Architettura multi-process in Chrome

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 ed 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 come il famoso Adblock.

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 tabs di condividere 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 tabs correlati di comunicare via JavaScript e condividere la cache. Al contrario, se viene aperta una pagina di un sito diverso verrà riservato un nuovo processo.

Per essere precisi, si definisce un "sito" come un dominio registrato (ad esempio `google.com` 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 la prima apre la seconda programmaticamente. Invece se l'utente digita manualmente lo stesso indirizzo in due tabs diverse, vengono considerate due istanze separate con processi distinti.

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

Process-per-site-instance

Normalmente i browsers usano una strategia "Process-per-site-instance", ovvero lo stesso sito aperto in tabs diverse 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 sia 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 tabs. Di conseguenza può risultare in processi molto onerosi.

Process-per-tab

Esiste anche una strategia più semplice che dedica un processo renderer per ogni gruppo di tabs. 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 allocare 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 riferimenti è necessario per il corretto funzionamento di molte applicazioni multi-finestra, 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 proprio in quanto non si possiede più alcun riferimento JavaScript ad esse. 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 tabs, 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.

L'utilizzo più comune è `targetWindow.postMessage(message);`, ove:

- * **targetWindow** è il riferimento alla finestra che riceverà il messaggio. Alcuni esempi per ottenere tale riferimento sono:
 - `window.open()` crea una nuova finestra e ne ritorna il riferimento;
 - `window.opener` restituisce il riferimento alla finestra genitore che ha aperto la finestra corrente tramite il metodo precedente;
- * **message** sono i dati da inviare all'altra finestra. I dati sono copiati 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  
}
```

Nel caso di *Stargate* non sono disponibili i riferimenti alle nuove finestre, in quanto risiedono su nuovi processi, per cui non è possibile usare *postMessage* direttamente con essi. Il capitolo successivo §5 fornisce la soluzione a questo problema, usando lo stesso metodo con altre vie di comunicazione.

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 tabs 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 §4.2.4.

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 periodico 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 profonda di oggetti JavaScript complessi ed è utilizzato internamente 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 ($a \rightarrow a$) o indirettamente ($a \rightarrow b \rightarrow a$).

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 solo 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 di default solo un sotto-insieme dei tipi di oggetti, in particolare unicamente 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 \Leftrightarrow stringa JSON;
- * Overhead 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*, la quale sfrutta lo *Structured clone algorithm*. Difatti entrambi sono nati esattamente per lo scopo di comunicazione tra la pagina principale ed altre entità quali estensioni browser, altre finestre e Web Workers (spiegati in §5.2).

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. Possono 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 il controllo UI che permette di aprire la mappa Google Maps esternamente.



Figura 42: Controllo per estrarre la mappa in una nuova finestra

4.3 PRIMA ARCHITETTURA STARGATE

Alla luce delle precedenti informazioni, si illustra di seguito una prima architettura per la libreria *Stargate*. Tale architettura verrà ampliata passo per passo nei prossimi capitoli.

Con la strategia *postMessage* è possibile organizzare l'applicazione nel seguente modo, ove si instaura una comunicazione bilaterale in cui *Parent* trasferisce lo stato applicativo ed i vari *Widget* notificano gli eventi.

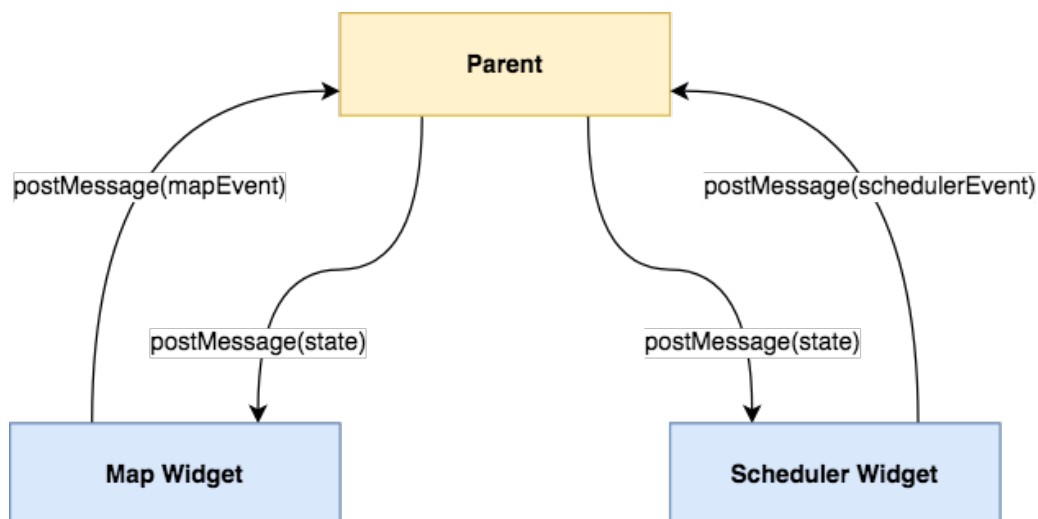


Figura 43: Prima architettura Stargate

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.

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 corretta esecuzione di tutte le sue funzionalità. La funzione che calcola tali dati, 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 pure¹, 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.

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

```
interface Purchase {  
    name: number  
    price: number  
}  
  
interface State {  
    purchases: Array<Purchase>  
}
```

¹https://en.wikipedia.org/wiki/Pure_function

```
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 matematiche $f \circ g$, 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*.

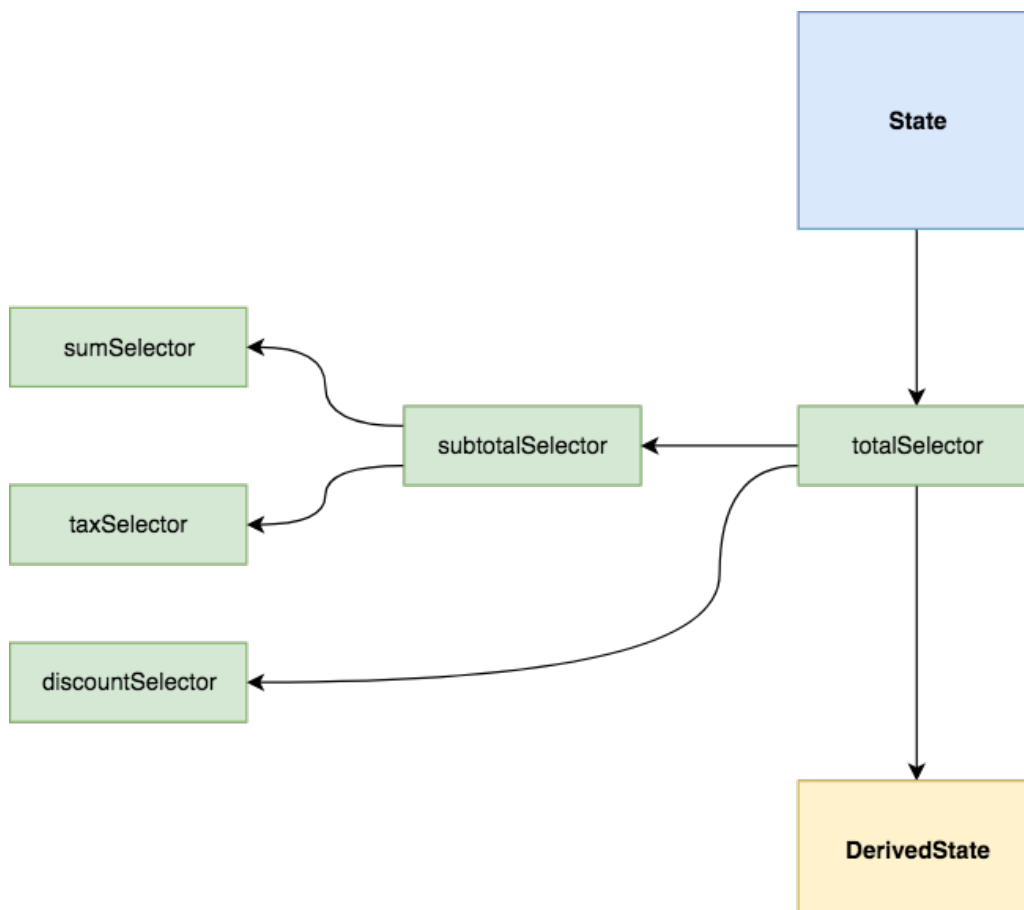


Figura 51: Esempio di composizione di selectors

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* sarebbe ripetuto pur essendo 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 una sola volta ad ogni modifica dello stato applicativo e propagare il risultato del calcolo ai widget, ad esempio a tutte le finestre contenenti la mappa. In tal modo l'onere computazionale è linearmente dipendente dal numero di **tipi di widget** attivi invece che dal numero totale di finestre. Tuttavia tale calcolo allo stesso tempo non può avere nella finestra padre per le stesse ragioni descritte sopra riguardo al eseguirlo in quelle figlie.

5.2 WEB WORKER

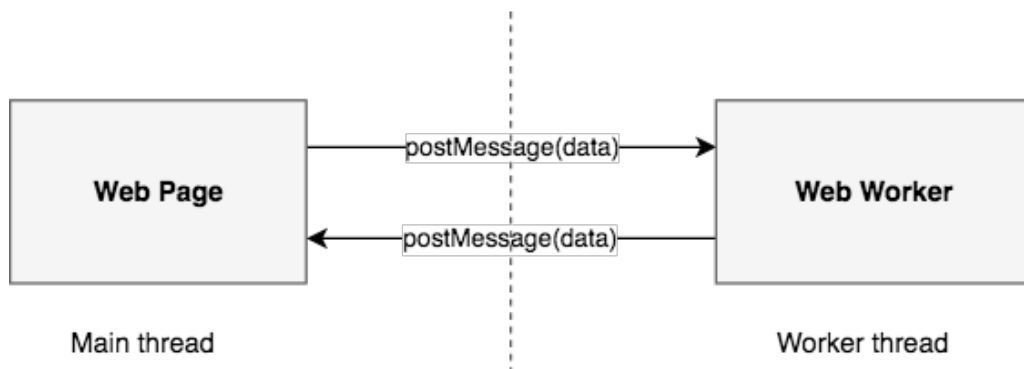


Figura 52: Comunicazione tra la pagine web ed il Web Worker

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 *Web 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 se fosse tra finestre. Infine i *Web Workers* possono attivare nuovi workers per delegare ulteriormente del lavoro in nuovi threads.

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 delle 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 quindi 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 (ad esempio finestre, 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 di comunicazione, si è in grado di inviare messaggi attraverso di esso usando `channel.postMessage(data)`, dunque 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 sottoscrivere al canale *BroadcastChannel* ed avere una comunicazione bi-direzionale (*full-duplex*) verso tutte le

altre.

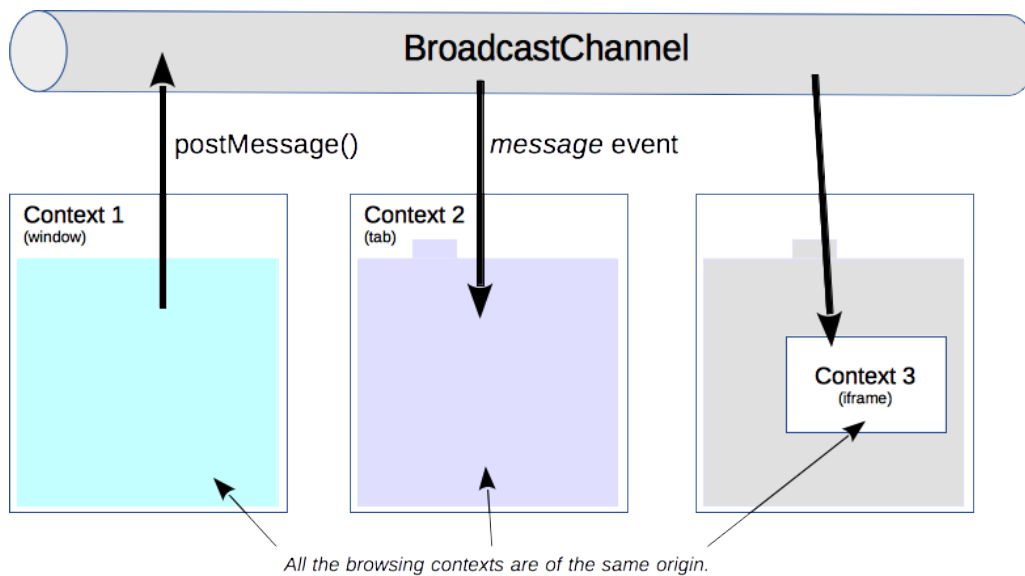


Figura 53: Esempio di funzionamento del BroadcastChannel

5.4 EVOLUZIONE ARCHITETTURA STARGATE

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

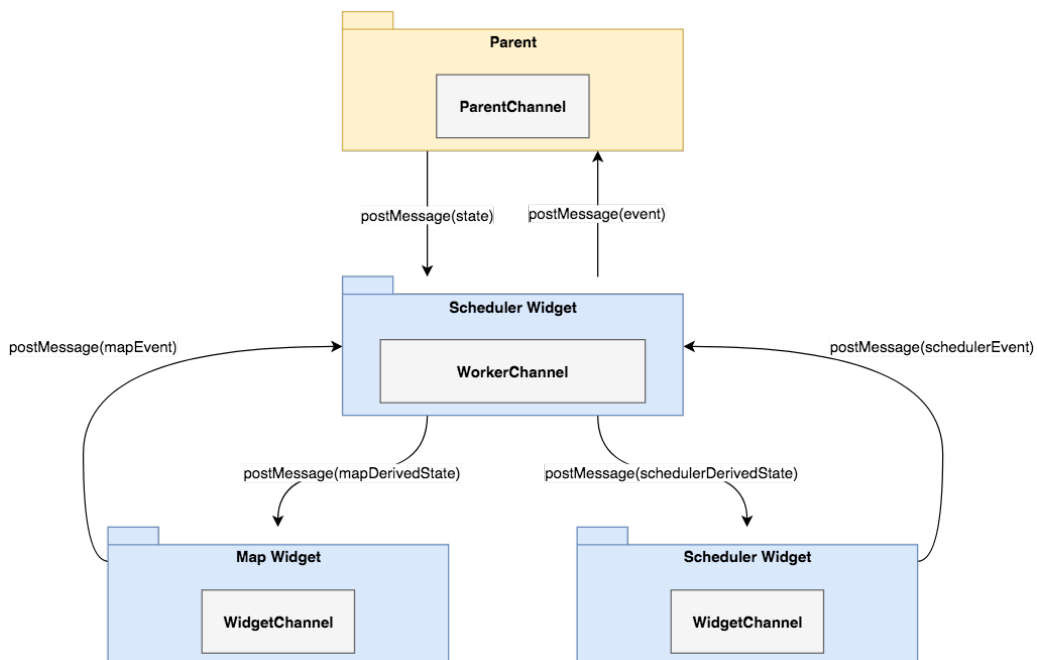


Figura 54: Architettura Stargate evoluta con Web Worker e BroadcastChannel

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, a differenza 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 *WidgetChannel* 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.

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 attraverso i *Web Workers* e tutte le operazioni dei widgets per via di 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 scambio di dati 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.

Un'interazione utente potrebbe richiedere un tempo sensibile per portare l'aggiornamento di tutte le finestre widget.

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 mancanti 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 solo alla loro apertura, ma i successivi aggiornamenti contengono solo i *delta*.

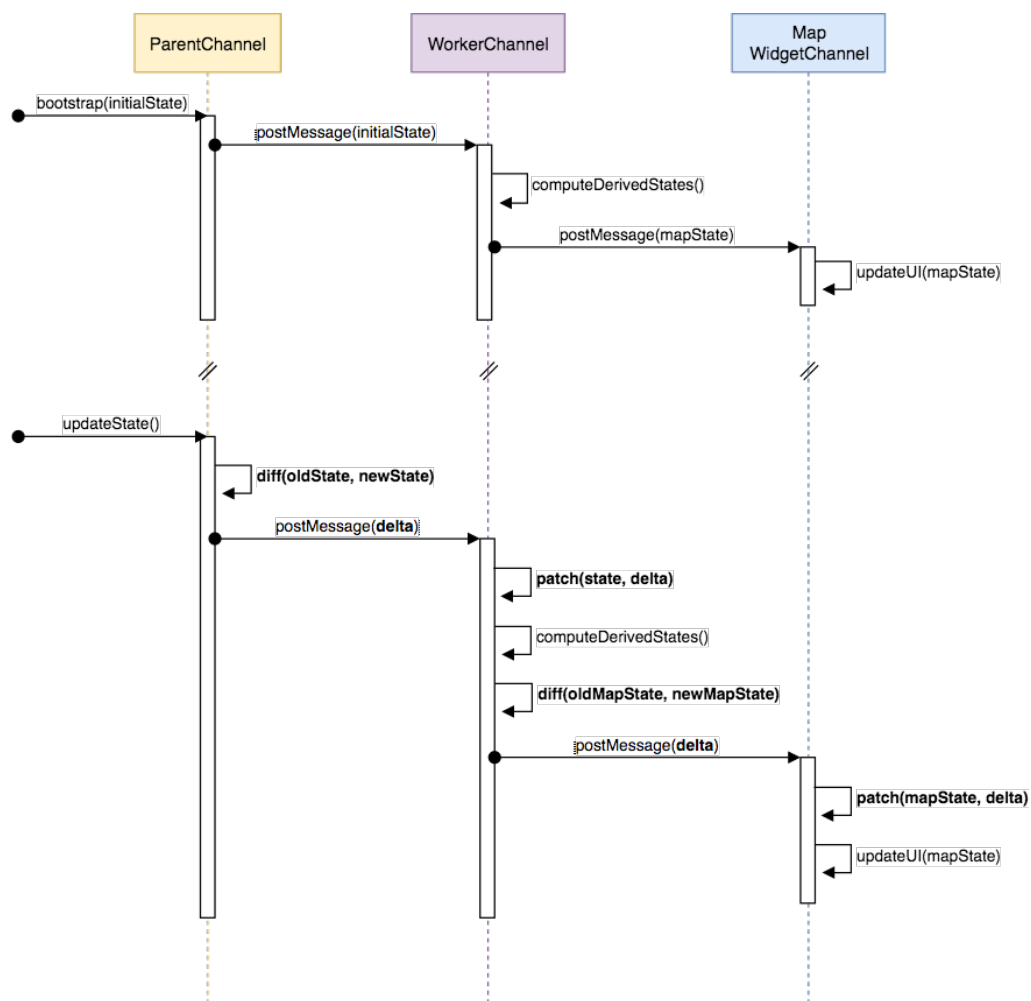


Figura 61: Sequenza delle chiamate per il flusso diff & patch

Si assicura in tal modo che le dimensioni del carico di trasmissione siano dipendenti unicamente dal numero di modifiche effettuate. Quest'ultime sono previste essere frequenti ma è ragionevole presupporre 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 fatte per riferimento, ma bensì ritornino una copia aggiornata avente un nuovo riferimento e le cui proprietà siano anch'esse nuove laddove 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'* sono stati modificati), che a vecchi (*b*, *c*, *h* sono invariati).

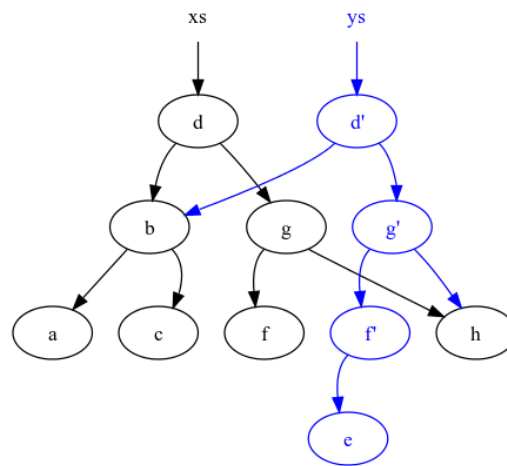


Figura 62: Albero di *ys*, ove le parti blu hanno nuovi riferimenti

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 mutano 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 divenute fondamentali anche per linguaggi non puramente funzionali come JavaScript in quanto portano a diversi vantaggi come la diminuzione 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* e

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

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 direttamente proporzionale $\Theta(n)$ al numero di campi annidati.

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'ipotetica 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*.

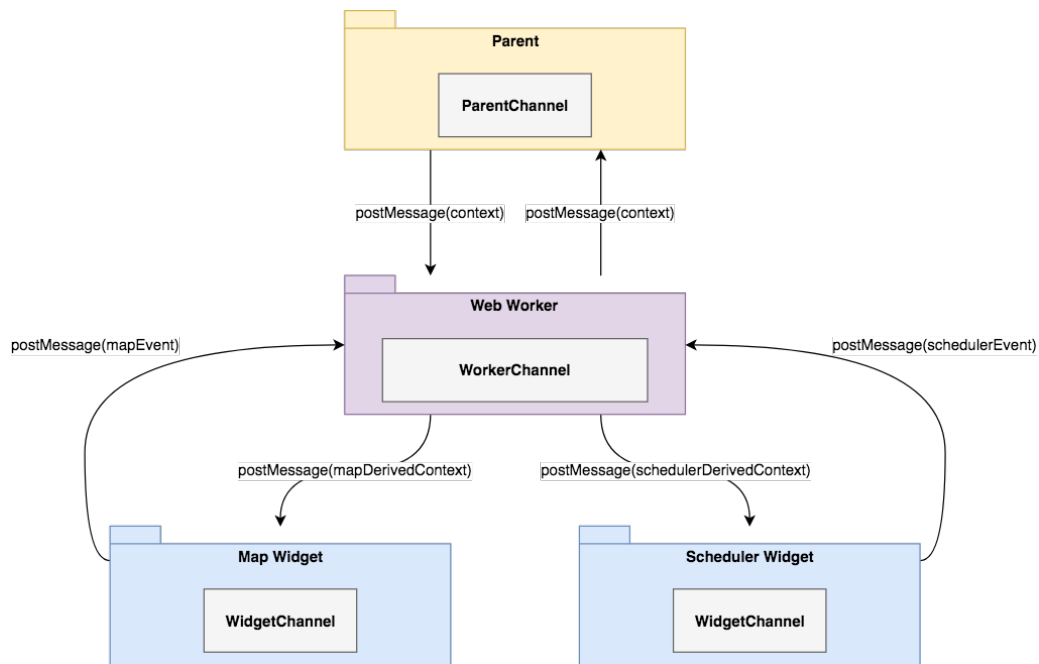


Figura 71: Architettura per il Context

7.1 REMOTE PROCEDURE CALL (RPC)

L'introduzione del *Context* porta però anche alla luce una nuova sfida: chiamate di metodi nella finestra padre da parte di widget. 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 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 forzatamente asincrone, anche qualora fossero originariamente sincrone.

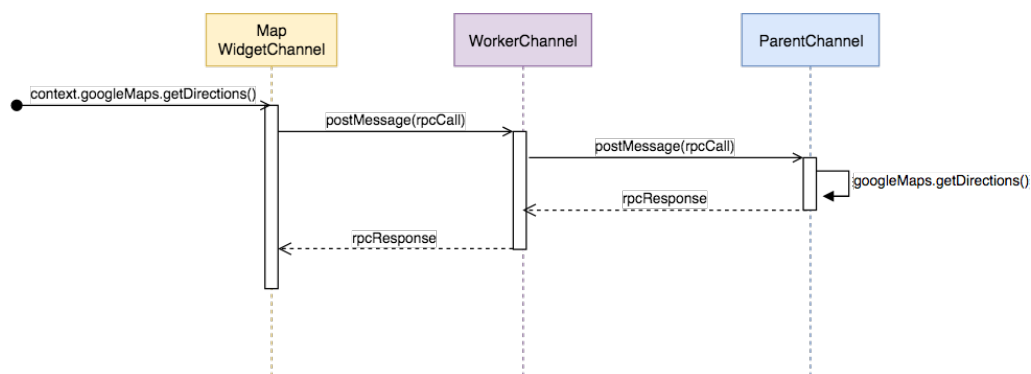


Figura 72: Sequenza per una chiamata RPC del context

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*.

Un *Proxy* è solitamente un contenitore che funziona come interfaccia di un altro oggetto, il cui scopo può essere semplicemente delegare a quest'ultimo o fornire logica extra per eseguire una connessione network, leggere da un file, ottenere risorse costose ed etc.

Nel caso della libreria, i *Proxy* degli oggetti nel *Context* funzionano esattamente come gli originali, ma racchiudono al loro interno la logica di comunicazione RPC descritta poco prima.

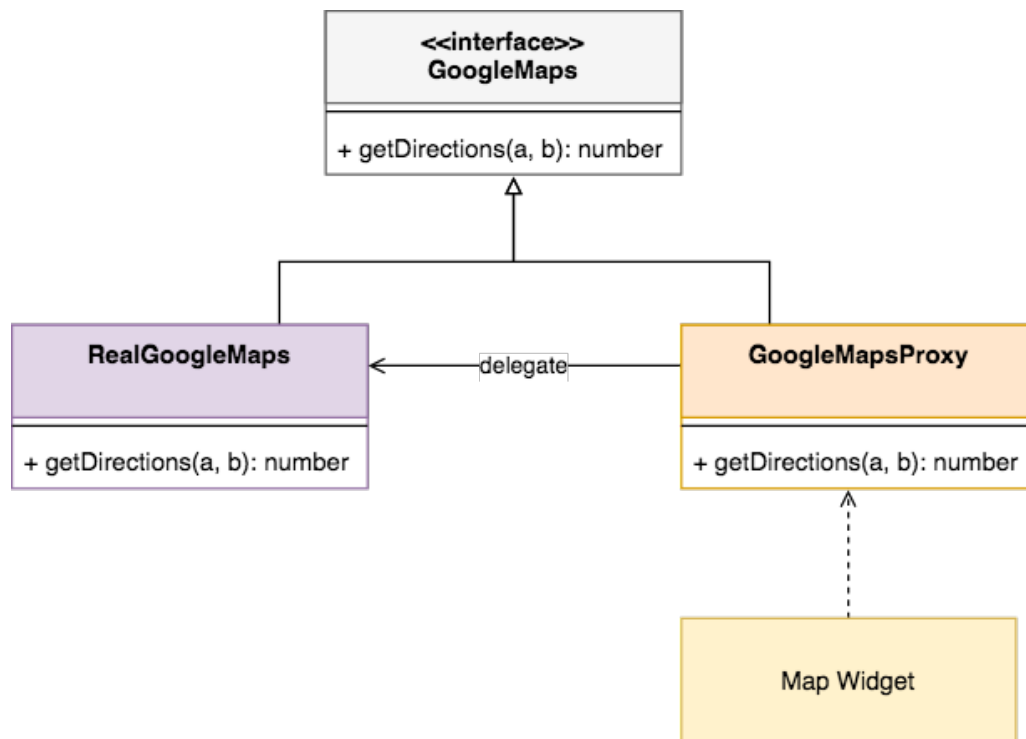


Figura 73: Esempio del pattern Proxy per GoogleMaps

Integrazione in Route Manager



Le precedenti scelte architettureali, in particolare l'evoluzione per il supporto al concetto di *Context* invece che stato applicativo, sono state dettate dalle esigenze dell'applicazione *WorkWave Route Manager*.

Tuttavia nonostante ciò l'integrazione ha richiesto anche un significativo lavoro all'interno del codice sorgente dell'applicazione, con aggiunta di logica necessaria puramente per il prodotto in questione per usufruire di *Stargate*.

Le modifiche architettureali alla libreria invece sono mirate e portare benefici a tutti i consumatori, non unicamente al prodotto *Route Manager*.

Di seguito sono presentate le principali sfide affrontate per l'integrazione.

8.1 GOOGLE MAPS

L'aspetto più doloroso dell'integrazione è stato Google Maps, tuttavia di vitale importanza per un'applicazione di gestione rotte. Google Maps difatti non è stato progettato per un uso moderno in *Web Workers* e per motivi di compatibilità mantiene comportamenti che lo rendono inadatto all'uso immediato con *Stargate*.

Alcune istanze, in particolari della classe `LatLng` che rappresenta una coordinata geografica, non possiedono campi pubblici ma bensì espongono solo metodi pubblici per poter accedere ai valori di latitudine/longitudine. Tuttavia esse devono essere presenti nello stato applicativo in quanto rappresentano dati fondamentali per l'applicazione e le istanze forniscono i metodi per lavorare agevolmente con tali dati.

Ciò causa però l'impossibilità di trasmetterli dall'applicazione principale verso il *Web Worker* o i widgets, in quanto è impossibile clonare una funzione. L'utilizzo di chiamate RPC §7.1 è invece proibitivo dato l'estremo alto uso di tali istanze, anche da parte di parti interni di Google Maps ove non è possibile riscrivere il codice affinché gestisca le chiamate RPC asincrone.

8.1.1 Google Maps serialization

Il problema ha richiesto di poter configurare la strategia di copia dei dati in *Stargate* durante la trasmissione tra le parti e soprattutto quindi di definirne una propria per Google Maps all'interno di *Route Manager*. In particolare, poiché Google Maps supporta la serializzazione verso il formato [JSON](#), è stato obbligatorio serializzare i dati prima della trasmissione di ogni messaggio e ripristinare alla ricezione.

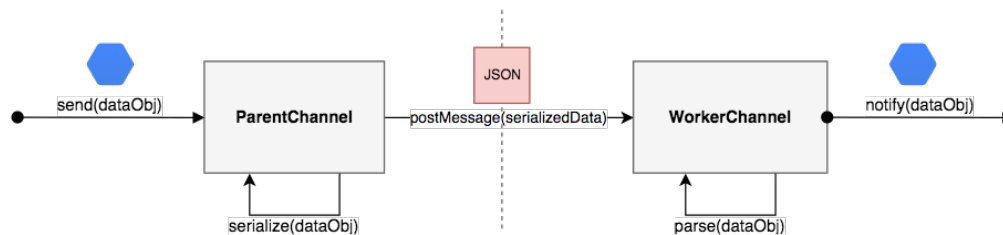


Figura 81: Flusso della serializzazione per Google Maps

A causa dell'overhead aggiuntivo, vi è stata una particolare attenzione per l'implementazione delle funzioni di serializzazione e parsing, che non puntano a soddisfare tutti i possibili casi d'uso di Google Maps, ma bensì solo quelli in uso nel *Route Manager* con la miglior performance possibile. Il risultato è stato soddisfacente e l'interazione utente con i widgets è fluida.

L'utilizzo della serializzazione invece della copia tramite *Structured clone algorithm* ha portato tuttavia ad altri problemi, in particolari quelli descritti come svantaggi di tale strategia alla sezione §4.2.4. Ciò ha richiesto del lavoro in più ma non vi è stato nulla di bloccante ed, essendo problemi minori, sono tralasciati da questo documento.

8.1.2 Google Maps diffing

Per motivi analoghi, non è possibile calcolare la differenza tra due oggetti `LatLng` in quanto hanno solo metodi e non campi dati. In questo caso, l'algoritmo di default per *diff & patch* è stato esteso in *Route Manager* per supportare tali oggetti.

In particolare l'algoritmo normalmente attraversa l'albero dell'oggetto, comparando ogni valore col suo precedente per calcolarne la differenza. In *Route Manager*, tale algoritmo consente di normalizzare i sotto-alberi prima che vengano attraversati per il calcolo e dando quindi la possibilità, nodo per nodo, di rimpiazzare le istanze `LatLng` con valori che siano comparabili.

Una volta calcolata la differenza, nell'oggetto *delta* vengono ripristinate le istanze originali di Google Maps ed il delta è pronto per la serializzazione

e trasmissione.

Poiché la normalizzazione avviene in contemporanea all'attraversamento dell'albero, il costo di performance aggiuntivo è costante in quanto non vi sono maggiori attraversamenti.

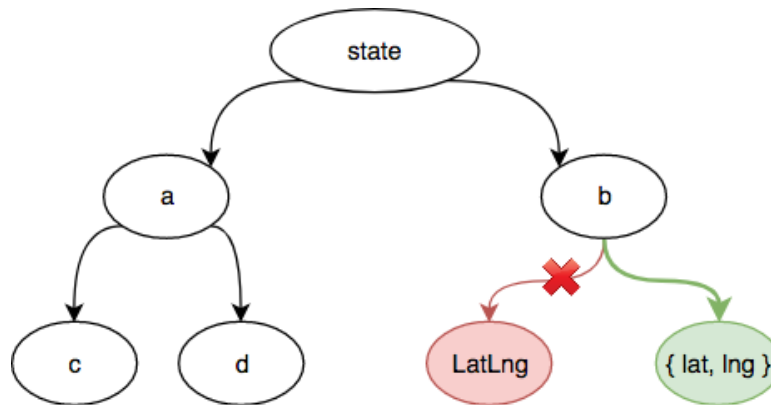


Figura 82: Esempio di nodo `LatLng` sostituito con oggetto avente campi comparabili

8.1.3 ChannelStrategy

Vista la precedente necessità di poter configurare alcuni aspetti del funzionamento, la libreria *Stargate* fornisce la possibilità di definire un'implementazione dell'interfaccia `ChannelStrategy`, utilizzata poi internamente per eseguire *diff & patch* e per sapere il formato dei messaggi.

Di seguito si presenta gli aspetti salienti di tale interfaccia, tralasciando alcuni campi minori non menzionati nel documento.

```
// Versione semplificata dell'interfaccia
interface ChannelStrategy<TransferFormat> {
  diff: (oldCtx: Context, newCtx: Context) => Delta | undefined
  patch: (ctx: Context, delta: Delta | undefined) => Context

  transfer: (message: Message) => TransferFormat
  receive: (message: TransferFormat) => Message
}
```

`diff/patch` definiscono le funzioni per l'omonimo algoritmo, mentre `transfer/receive` permettono di trasformare il messaggio dal tipo `Message` al tipo `TransferFormat` durante la trasmissione e l'inverso durante la ricezione.

La libreria fornisce già due implementazioni di tale interfaccia per un'uso immediato: `CloneStrategy` basato sullo *Structured clone algorithm* e `SerializeStrategy` basato sulla *serialization*. È tuttavia possibile per il

consumatore utilizzare una propria implementazione.

`ParentChannel`, `WorkerChannel` e `WidgetChannel` sono tutti configurabili con `ChannelStrategy` e solitamente viene usata la stessa per tutte e tre le classi, ma nulla vieta di usare diversi algoritmi tra le parti. È difatti possibile voler usare un algoritmo *diff & patch* estremamente ottimizzato per un `WidgetChannel` sapendone le necessità e limiti.

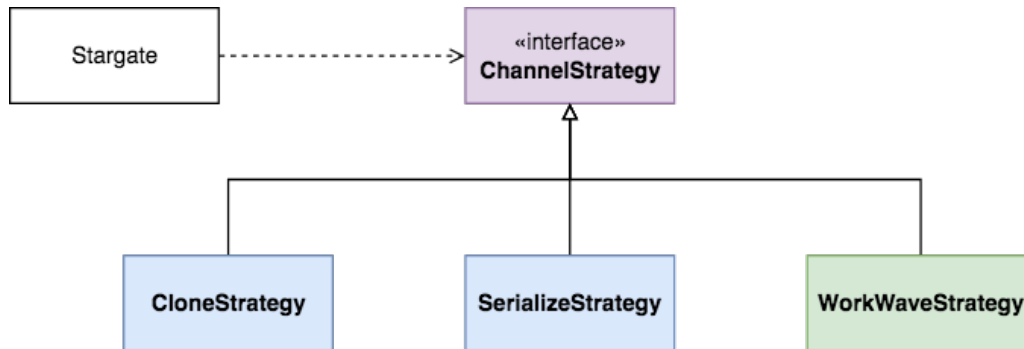


Figura 83: Strategy Pattern per la configurazione di `Stargate`

Valutazione retrospettiva

Si riporta di seguito una valutazione retrospettiva dell'esperienza di stage, al fine di valutare ciò che è stato fatto, le esperienze apprese e trarre insegnamenti per il futuro.

9.1 RAGGIUNGIMENTO DEGLI OBIETTIVI

Durante lo sviluppo è emerso che alcuni requisiti quali "005 - Supporto a multi-sessione" fossero estraneamente facili da implementare con le tecnologie descritte nei capitoli precedenti, mentre altre come "006 - Gestione della configurazione di Route Manager per supportare i widget" sono state difficoltose quanto previsto o addirittura di più.

Tuttavia sia il lavoro di ricerca che il successivo sviluppo sono proseguiti bene nonostante qualche imprevisto lungo la strada ed entrambi le parti, azienda WorkWave e sottoscritto, siamo soddisfatti del risultati. Difatti tutti gli obiettivi obbligatori sono stati raggiunti con successo e vi è stato sufficiente tempo per il completamento anche di quelli desiderabili.

Prima del termine dell'esperienza, la libreria *Stargate* è stata integrata correttamente nell'applicazione *Route Manager* ed è attualmente già in uso in produzione da parte di alcuni clienti del prodotto.

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

Tabella 91: Tabella raggiungimento degli obiettivi

9.2 RESOCONTO DELL'ANALISI DEI RISCHI

Si riporta di seguito anche il riscontro dei rischi analizzati ad inizio stage.

Descrizione	Verificato	Piano attuato
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.	NO	Nessuno, la scrupolosa analisi iniziale delle tecnologie è stata fruttuosa.
Difficoltà di integrazione: vi è il rischio che non sia possibile integrare le tecnologie adottate nel contesto dell'applicazione <i>Route Manager</i> , in quanto è già in sviluppo da oltre un anno e non progettata in partenza per essere multi-finestra.	SI	Grazie al supporto del tutor aziendale Matteo Ronchi sono state individuate soluzioni dal buon compromesso che hanno permesso l'integrazione. I dettagli sono elaborati nel capitolo §8.

Tabella 92: Tabella dell'analisi dei rischi

9.3 CONOSCENZE ACQUISITE

Il periodo di stage è stato ricco sia di insegnamenti a livello di conoscenze tecniche che a livello di esperienze personali. Di seguito si riportano le principali skill acquisite.

9.3.1 Ricerca & Sviluppo

Sebbene abbia già svolto attività R&D in passato, questa è stata la prima esperienza in tal senso della durata di due mesi a tempo pieno. Ho avuto la fortuna di poter analizzare con calma lo stato tecnologico e cercare di non perdere alcuna informazione. Sono state difatte più volte piccoli informazioni apparentemente non inerenti che hanno fatto la differenza nella direzione presa dal progetto e addirittura a determinarne il successo o fallimento.

Ho osservato quanto sia importante, nella fase di ricerca tecnologica, non precludersi da alcuna informazione sebbene sembri a primo impatto solo

in parte affina all'argomento d'interesse. Proprio data la natura sperimentale del progetto, è importante raccogliere maggiori informazioni possibili ed elaborarle per formare una strategia di sviluppo.

Esempio di questa esperienza è stato l'apprendimento del parametro di sicurezza `rel=noopener` per forzare l'uso di nuovi processi ad ogni finestra §4.1.3, dedotto per puro caso da un articolo in cui si accennava di esso unicamente per scopo di sicurezza senza menzionare riguardo a processi.

Inoltre ho appreso a mie spese che è bene non tralasciare di leggere il codice sorgente di alcuni progetti open-source, in quanto vi sono molte informazioni importanti assenti nella documentazione poiché magari ritenute dall'autore di utilità improbabile.

Infine è stato importante anche il tracciamento delle fonti e delle informazioni fin dal primo momento, a fine di documentare il processo di ricerca effettuato. Sarà possibile in tal modo capire come è stato raggiunto il risultato e quali altre possibili vie non sia state prese.

9.3.2 Tecnologie apprese

Tra le conoscenze puramente tecniche ho avuto modo di migliorare la mia conoscenza del *type-level programming*, ovvero dell'uso avanzato dei tipi statici in TypeScript. In particolare ho approfondito l'uso degli enums e soprattutto dei *conditional types* ¹.

Inoltre ovviamente ho migliorato le mie conoscenze per quanto riguarda la comunicazione asincrona per messaggi, avendo ad esempio usato ampiamente `postMessage` e implementato da zero le chiamate RPC §7.1.

Infine vi è stata anche l'opportunità di lavorare a livello avanzato con le API di Google Maps, quali ad esempio *markers*, *disegno sulla mappa*, *polylines* e *polygons*.

9.4 VALUTAZIONE PERSONALE

Questa esperienza in WorkWave mi ha permesso di conoscere in primis persone fantastiche, non solo a livello professionale, e di imparare ad introdurmi in un team dove vi sono dei veri valori condivisi dai singoli. Lavorando in team è infatti fondamentale il rispetto del lavoro altrui e ad allo stesso tempo la capacità di saper ricevere critiche e migliorare il risultato progressivamente.

¹<https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-8.html>

Inoltre la possibilità di condividere lo spirito verso nuove sfide e di risolvere un problema insieme ad un collega arricchisce ancor di più l'esperienza ed il rafforzamento dello spirito di appartenenza al team. Saper lavorare in team è indispensabile per perseguire un risultato di qualità, ma anche per poter continuare ad amare questo lavoro.

A livello tecnico ho invece potuto felicemente constatare l'aiuto fornito dalle nozioni apprese durante il corso di studi e comprendere l'importanza di una solida base per poter mirare in alto.

In conclusione l'esperienza in WorkWave è stata ricca sia a livello conoscitivo che umano e spero poter ripetere anche in futuro.

Tecnologie utilizzate



A.1 TYPESCRIPT



Figura A1: TypeScript

TypeScript è un linguaggio di programmazione open-source sviluppato e mantenuto da Microsoft. La sua sintassi è un super-set di JavaScript, ovvero qualsiasi sintassi di quest'ultima è valida anche nella prima ma non vice-versa. TypeScript aggiunge difatti la possibilità di avere tipi statici.

È inoltre progettato per lo sviluppo di grandi applicazioni e compila in JavaScript, per cui può essere usato sia per l'esecuzione lato client che server (Node.js).

Il linguaggio fornisce i tipi statici attraverso *type annotations* che attivano il controllo dei tipi a tempo di compilazione. La scelta è tuttavia opzionale ed è possibile scrivere normale codice JavaScript senza tipi.

```
// Esempio di funzione con tipi statici
function add(left: number, right: number): number {
  return left + right;
}
```

I tipi statici possono anche essere esportati dopo la compilazione in file di dichiarazione separati, in modo da fornire le informazioni sui tipi a chi deve utilizzare la libreria, la quale sarà già stata compilata in JavaScript.

Sia la libreria *Stargate* che l'applicazione *WorkWave Route Manager* sono scritte in TypeScript 3.0, rilasciato ad agosto 2018.

A.2 REACT

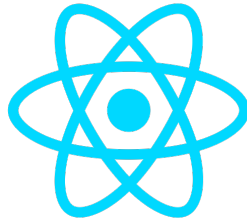


Figura A2: React

React è una libreria JavaScript per la realizzazione di applicazioni web, in particolare per la creazione di interfacce utenti. È mantenuta e sviluppata da Facebook, sebbene abbia anche una forte community open-source.

La seguente classe è un componente UI React, che accetta una proprietà `greeting`. Il metodo `ReactDOM.render` si occupa invece di creare un'istanza di tale componente, passando ad esso `"Hello World!"` come valore di `greeting`. Il risultato è renderizzato come figlio del nodo HTML avente id `"root"`.

```
class Greeter extends React.Component {  
  render() {  
    return <h1>{this.props.greeting}</h1>  
  }  
}  
  
ReactDOM.render(  
  <Greeter greeting="Hello World!" />,  
  document.getElementById('root')  
);
```

Di seguito sono invece elencate le caratteristiche principali di React, che hanno fortemente influenzato lo sviluppo moderno di web applications:

- * **One-way data binding con "props"**: tutte le informazioni esterne di cui ha bisogno il componente sono ricevute ed aggiornate tramite props dal componente genitore. Ciò assicura un flusso dei dati unidirezionale e predicibile. Inoltre le props sono da trattarsi come dati in sola lettura ed immutabili;
- * **Componenti Stateful**: oltre alle props, i componenti possono avere uno stato interno utilizzabile internamente o passabile come prop ai figli.

```
class ParentComponent extends React.Component {  
  state = { color: 'red' };  
  
  render() {  
    return (  
      <ChildComponent color={this.state.color} />  
    );  
  }  
}
```

- * **Virtual DOM:** il DOM è la rappresentazione in JavaScript della struttura HTML della pagina. React crea invece una propria rappresentazione virtuale del DOM dei propri componenti ed aggiorna efficientemente quello reale solo laddove necessario. Ciò consente al programmatore di descrivere il render dell'intera applicazione in base a props e state, mentre React si occupa delle effettive modifiche necessarie al DOM;
- * **JSX:** è un'estensione della sintassi JavaScript per poter scrivere un codice simile all'HTML, ma dinamicamente in JavaScript, invece che come stringa template statica.

React può essere usato come libreria di appoggio per lo sviluppo di applicazioni sia web che mobile e spesso è accompagnata da ulteriori librerie, quali *Redux*, per la gestione dello stato, delle rotte o delle richieste server.

A.3 REDUX



Figura A3: Redux

Redux è una libreria JavaScript per la gestione dello stato applicativo, che può essere visto come l'equivalente dello stato di un componente React ma relativo all'intera applicazione. La libreria implementa l'architettura *Flux*, un'alternativa al popolare Model-View-Controller e che si avvicina più ad architetture ad eventi/messaggi.

In Redux, oggetti chiamati *actions* hanno il compito di descrivere l'intento di modifica dello stato applicativo. Tale messaggio viene inviato allo *Store*, il quale gestisce la modifica dello stato applicativo in base all'*action* e notifica la view (React) del cambiamento. Nessuna classe, ad eccezione dello *Store*, ha la possibilità quindi di aggiornare lo stato applicativo, trattato come immutabile.

All'interno dello *Store*, l'aggiornamento effettivo dello stato applicativo avviene attraverso i *reducers*, funzioni pure aventi la firma (State, Action) => State e componibili tra di loro.

Si genera quindi un flusso unidirezionale, in cui la view riceve i dati e notifica di eventi tramite *actions*. Quest'ultimi sono passati ai *reducers* che producono un nuovo stato applicativo, passato dallo *Store* nuovamente alla view.

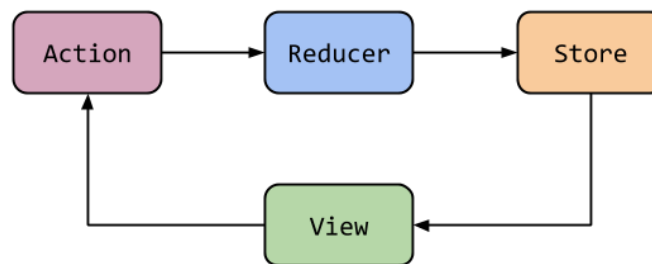


Figura A4: Flusso unidirezionale in Redux

Bibliografia

- [1] Chrome Multi-process architecture. URL:
<https://www.chromium.org/developers/design-documents/multi-process-architecture>
- [2] Chrome Multi-process strategies . URL: <https://www.chromium.org/developers/design-documents/process-models>
- [3] About rel=noopener . URL:
<https://mathiasbynens.github.io/rel-noopener/>
- [4] Deep-copying in JavaScript . URL:
<https://dassur.ma/things/deep-copy/>
- [5] BroadcastChannel API . URL: https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API
- [6] Crosstab. *A utility library for cross-tab communication using localStorage.*
URL: <https://github.com/tejacques/crosstab>
- [7] Using Web Workers . URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
- [8] Remote Procedure Call . URL:
https://en.wikipedia.org/wiki/Remote_procedure_call