Università degli Studi di Camerino

Scuola di Scienze e Tecnologie
Corso di Laurea in Informatica (Classe L-31)

La gestione del flusso dei dati nel front-end di applicazioni web complesse

Laureando
Diego Pasquali
Matricola 093341

Relatore
Prof. Luca Tesei

A.A. 2016/2017

Indice

1. Introduzione

Il *Flusso dei dati* nel front-end di una applicazione web rappresenta tutti gli input e gli eventi che si muovono attraverso i suoi vari livelli logici. Più questo flusso è intenso e disorganizzato più diventa complicato gestire lo stato dell'applicazione.

Un'interfaccia utente mette a disposizione dell'utilizzatore una grande quantità di interazioni, sia volontarie che involontarie, che cambiano in continuazione lo stato dell'applicazione e che devono essere opportunamente gestite e sincronizzate. La struttura del codice diventa quindi prioritaria al fine di ottenere un prodotto che sia soddisfacente a livello di prestazioni e che riesca a mantenere un adeguato livello di scalabilità.

1.1 Background

La gestione del flusso dei dati all'interno di una applicazione web è un argomento molto discusso dopo l'avvento di tecnologie front-end sempre più complesse e potenti come *React* o *Angular*. La causa di ciò è la necessità di avere un codice che sia il più possibile scalabile ed il più facilmente testabile a prescindere dal numero di features che verranno successivamente aggiunte. Codebase vaste come potrebbero essere quelle di Facebook, Twitter o YouTube necessitano di una architettura di fondo che sia altamente chiara e comprensibile per evitare confusione tra i vari servizi. Come vedremo successivamente, architetture datate come l'MVC pur essendo molto efficienti lato back-end non rendono allo stesso modo lato front-end, dove c'è una quantità maggiore di azioni che l'utente può intraprendere e che possono avere ripercussioni differenti su più componenti diversi all'interno di una View. In questo documento verrà discussa l'alternativa attualmente più gettonata che è quella dell'architettura a flusso unidirezionale, implementata in prima battuta da Flux e successivamente ottimizzata da Redux.

1.2 Lo stato dell'arte

Possiamo paragonare la creazione della prima applicazione web con la messa online del primo sito da parte di Tim Berners-Lee nel 1991 dal Cern di Ginevra [Gra12]. Stiamo tuttavia parlando di una applicazione statica costruita solamente in HTML. La svolta avvenne il 5 maggio del 1995 con l'avvento di Javascript [W3C12], il linguaggio che anche adesso è alla base di tutte le tecnologie web più nuove e potenti. Da qui in poi l'evoluzione è andata avanti in maniera esponenziale partendo da un utilizzo banale del linguaggio fino a giungere alla situazione attuale con framework ed architetture complesse.

1.2.1 Single-page web application

La problematica della gestione del flusso dei dati è diventata estremamente rilevante con la comparsa delle così dette *Single-page web application* (SPA), ossia applicazioni organizzate all'interno di un'unica pagina web e gestite interamente tramite Javascript. Nel loro stadio iniziale queste non erano costruite sopra una struttura ben definita e la gestione del flusso dei dati avveniva in maniera disordinata gestendo ogni azione dell'utente in maniera diretta.

Andando avanti con il tempo e con l'aumentare della complessità di queste applicazioni, si è sentito il bisogno di creare un'architettura ben definita che aiutasse a gestire in maniera più consistente lo stato del servizio e tutti i suoi eventuali cambiamenti.

1.2.2 I primi framework basati su MVC

La necessità di avere struttura più solida lato front-end, specialmente per applicazioni più esigenti, ha portato alla nascita dei primi framework basati sull'architettura MVC (Model - View - Controller).

> Single page apps are distinguished by their ability to redraw any part of the UI without requiring a server roundtrip to retrieve HTML. This is achieved by separating the data from the presentation of data by having a model layer that handles data and a view layer that reads from the models. [Tak13]

Nel modello MVC per front-end (e come anche in quello back-end), che riprenderemo in dettaglio più avanti nel documento, abbiamo una netta distinzione tra i dati e la presentazione di quest'ultimi (ossia tra Model e View). Questo ci permette di avere un controllo maggiore sullo stato globale e di ogni singolo componente dell'interfaccia oltre ad un codice più robusto e facile da modificare nel tempo [Ven08].

Javascript ha a disposizione un numero considerevolmente alto di framework MVC. Uno dei più famosi è sicuramente *AngularJS* (parliamo della versione 1) mantenuto da Google e dalla vasta community formatasi intorno. L'architettura MVC lato front-end non è considerata però ottimale. Col passare del tempo si sono venute a creare delle strutture derivate da questa che tendono a sviluppare la gestione dell'interfaccia utente in maniera differente, con i propri vantaggi e svantaggi. Una di queste è MVP (Model - View - Presenter) utilizzata da *Backbone.js*. In questa il Presenter, che sostituisce il Controller, ha una responsabilità minore di quest'ultimo e si occupa solamente di passare dati alla View la quale deciderà cosa e come mostrare. Un'altra architettura derivata da MVC è MVVM (Model - View - ViewModel), utilizzata da framework come *Ember.js* e *Knockout*, in cui il ViewModel si occupa di mantenere i dati del Model (che sono grezzi) nella forma richiesta dalla View, ed espone a quest'ultima metodi e funzioni per la gestione dello stato dell'applicazione. [Cha16].

Ancora una volta tuttavia ci troviamo di fronte ad un muro, dove le architetture venutesi a creare sono tante ma tutte derivate da MVC traendone i difetti. Il problema più grande di questo paradigma, specialmente nei suoi derivati MVVM e MVP, è la comunicazione bidirezionale: la View ha la possibilità di modificare direttamente o per mezzo di un livello intermedio il Model. Tenendo presente che stiamo parlando di applicazioni complesse e quindi con un grande numero di features, questo genera un effetto cascata tra i vari componenti dell'interfaccia che sono direttamente o indirettamente collegati a quel Model causando una codebase molto difficile da gestire ed analizzare.

Per la prima volta a questo punto si inizia a discutere di flusso di dati in maniera attiva e diretta riconoscendolo come difetto principale del modello MVC e trovando in React la libreria perfetta per risolverlo [Sal15].

1.2.3 Framework ed architetture a flusso unidirezionale

Un ulteriore passo avanti nella storia delle applicazioni web è stata React, libreria per la creazione di interfacce utente sviluppata da Facebook che si basa su diverse tecnologie all'avanguardia e che ci permette grazie alla sua versatilità di strutturare architetture più complesse ed efficienti. React è una libreria e non un framework, ciò significa che il suo lavoro è solamente quello di creare interfacce utente [Bun15]. Non è quindi una soluzione finale ma un componente fondamentale che unito ad altri ci permette di scrivere applicazioni web in maniera molto più dinamica.

Per sfruttare al massimo una libreria come React, Facebook mette anche a disposizione una struttura che sembra risolvere il problema relativo alla comunicazione bidirezionale riscontrato nel paradigma MVC. *Flux* è un'architettura complessa per la costruzione di interfacce utente che si basa su un flusso di dati unidirezionale facilmente gestibile ed altamente scalabile. In Flux una View non modifica mai in maniera diretta lo stato di una applicazione, bensì propaga azioni che vengono gestite da un *Dispatcher* e che hanno delle ripercussioni sul Model di questo paradigma che si chiama *Store*. Infine, quest'ultimo si occupa di propagare a sua volta un evento a tutti i componenti delle View che permette loro di aggiornarsi di conseguenza. In questo modo ogni cambiamento dello stato viene notificato ad ogni componente dell'interfaccia in maniera chiara e pulita.

1.2.4 Un approccio funzionale al flusso unidirezionale: Redux

Flux riesce a risolvere a pieno tutti i problemi fino ad ora descritti causati dalla comunicazione bidirezionale delle architetture MVC e derivate. Tuttavia la complessità di tale paradigma si fa sentire già da subito sia per il gran numero di passaggi che un'azione deve effettuare prima di essere effettivamente eseguita, sia per la difficoltà nell'implementazione anche in applicazioni di medio-piccole dimensioni.

Un'alternativa che si basa sempre su Flux è *Redux*, una libreria scritta da Dan Abramov e Andrew Clark che si identifica come semplice "Gestore dello stato" di una applicazione. Redux riesce a semplificare l'architettura di Flux utilizzando dei pattern della programmazione funzionale come la composizione e l'immutabilità per ottenere una struttura a flusso unidirezionale eliminando il Dispatcher e altre complessità [Abr15].

Vedremo nei prossimi capitoli come sia semplice ed elegante gestire il flusso dei dati e lo stato di un'applicazione che utilizza Redux e React, e come questi risolvono in dettaglio le principali problematiche delle architetture descritte precedentemente.

2. Strumenti

Per analizzare le varie architetture presentate dobbiamo prima fare un discorso sugli strumenti utilizzati per costruire una applicazione web e che useremo per gli esempi di codice dei capitoli successivi.

2.1 ECMAScript 2015

Anche conosciuto come *ECMAScript6*, è una standardizzazione del linguaggio Javascript creata da Ecma International. Questa versione in particolare mette a disposizione features molto utili per scrivere codice funzionale. Possiamo classificare ECMAScript come un linguaggio a se e differente da Javascript, che in principio doveva essere utilizzato solamente come linguaggio di scripting lato web, ma che ora viene utilizzato come vero e proprio linguaggio di programmazione su ambienti e scale differenti [Ecm15].

In ambito web non tutte le features di ES6 sono disponibili, per questo si utilizzano strumenti come *Babel* o *Webpack* che hanno la funzione di *transpiler*, ossia di compilare codice sorgente da ECMAScript6 a ECMAScript5 che è supportato dalla stragrande maggioranza dei browser.

2.2 React

React è una libreria scritta da Facebook per la creazione di interfacce utente interattive in maniera funzionale ed altamente scalabile. Si basa sul concetto di "componente" come elemento base fondamentale, ossia un pezzo di interfaccia che ha uno stato proprio ed è riutilizzabile all'interno del servizio. Il concetto funzionale di composizione si adatta benissimo a React: un componente complesso dovrebbe essere formato da componenti più piccoli e agnostici che possono quindi essere riutilizzati in altri componenti complessi.

Viene utilizzato in produzione sia da Facebook che da Instagram e fa uso di diverse tecnologie all'avanguardia come il *Virtual DOM* ed il *Server-side Rendering* [Whe14].

2.2.1 Virtual DOM

Il *Document Object Model* (DOM) è una API che definisce la struttura di un documento HTML e come essa viene acceduta e manipolata. E' una rappresentazione ad oggetti di una pagina web la quale può essere modificata con un linguaggio di scripting come Javascript. Per fare un esempio pratico, lo standard DOM stabilisce che l'interfaccia *Document* rappresenti l'intera pagina HTML e che concettualmente sia il nodo root. L'interfaccia *Node* rappresenta invece l'elemento base ossia il singolo nodo all'interno

di un documento e l'implementazione di questa richiede la creazione dei metodi per la gestione del nodo stesso e dei propri figli [PLH00].

Quando parliamo di Virtual DOM parliamo di un'astrazione sopra l'astrazione del DOM. Modificare quest'ultimo non è particolarmente dispendioso (si tratta solamente di modificare un oggetto Javascript) è tuttavia il processo di lettura e di "ridisegno" della pagina da parte del browser il vero problema. Questa tecnologia riesce a risolvere il suddetto problema mantenendo in memoria una rappresentazione del DOM reale che utilizza il design pattern *Observer*[1]per capire quale particolare nodo è stato modificato generando successivamente un nuovo albero derivato dal precedente ma con il nuovo stato. A questo punto effettua complessi algoritmi di differenza per trovare il numero minimo di passaggi per aggiornare il DOM reale per poter infine effettuare la riconciliazione [Mis17].

Il concetto che permette al Virtual DOM di garantire prestazioni maggiori sul DOM reale consiste nell'aggiornamento aggregato. Tutti i cambiamenti effettuati da un evento (che sono i passaggi trovati dall'algoritmo di differenza) vengono aggregati ed il DOM viene ridisegnato solamente una volta.

React implementa il Virtual DOM attraverso *JSX*, una estensione di ECMAScript simile ad XML che permette di scrivere elementi di markup con una sintassi simile all'HTML all'interno dei componenti dell'interfaccia. Portare l'HTML all'interno del codice sorgente Javascript offre dei vantaggi non banali come ad esempio il debugging compile-time degli errori di sintassi durante la costruzione del DOM, la versatilità di avere un linguaggio di scripting per effettuare composizione ed altre azioni dinamiche e sopratutto avere una perfetta separazione tra componenti differenti.

2.2.2 Server-side rendering

React è una libreria *isomorfica*, è in grado di essere eseguita sia lato client che lato server traendo vantaggio da NodeJS ed il fatto che il principale linguaggio di programmazione per entrambi gli ambienti sia sempre Javascript. Il vantaggio di eseguire React anche lato server risiede nella prima visualizzazione. In una Single Page Application normale durante il primo caricamento vengono scaricati gli elementi base per la sua esecuzione e successivamente viene eseguito il codice Javascript per il rendering del suo stato iniziale. La seconda fase può essere ulteriormente pesante, basti pensare che possono essere effettuate ulteriori richieste per soddisfare il normale fabbisogno dell'applicazione.

La tecnica del Server-side rendering permette di semplificare questa prima visualizzazione interpretando i componenti React lato server e restituendo una pagina iniziale con la Single Page Application già avviata e fornita di tutti i dati di cui avrebbe normalmente bisogno.

[1]Il design pattern Observer si struttura di un oggetto chiamato "Subject" che mantiene una lista di altri oggetti dipendenti chiamati "Observers" e li notifica ogni qualvolta il suo stato viene modificato.

Bibliografia

[Abr15] Dan Abramov. Risposta su stackoverflow alla domanda “why use redux over facebook flux”, 2015.

[Bun15] Samer Buna. Yes, react is taking over front-end development. the question is why., 2015.

[Cha16] Shailendra Chauhan. Understanding mvc, mvp and mvvm design patterns, 2016.

[Ecm15] EcmaInternational. Ecmascript® 2015 language specification, 2015.

[Gra12] Dino Grandoni. World’s first website, created by tim berners-lee in 1991, is still up and running on 21st birthday, 2012.

[Mis17] Rupesh Mishra. Virtual dom in reactjs, 2017.

[PLH00] Jonathan Robie Philippe Le Hégaret, Lauren Wood. What is the document object model?, 2000.

[Sal15] Amir Salihefendic. Flux vs. mvc (design patterns), 2015.

[Tak13] Mikito Takada. Single page apps in depth, 2013.

[Ven08] Bill Venners. The importance of model-view separation, a conversation with terence parr, 2008.

[W3C12] W3C. A short history of javascript, 2012.

[Whe14] Ken Wheeler. Learning react.js: Getting started and concepts, 2014.