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

Abstract

In questa tesi viene trattata la problematica relativa alla gestione del flusso dei dati e degli eventi che modificano direttamente o indirettamente l'interfaccia di una applicazione web moderna. Avere un adeguato controllo su questo flusso è di primaria importanza al fine di ottenere un servizio il cui codice sia chiaro e con uno stato che cambi in maniera comprensibile e deterministica, dove sia semplice riprodurre eventuali errori ed aggiungere nuovi elementi.

In questo documento verranno messe a confronto architetture innovative come *Flux* e *Redux* che implementano un flusso di dati unidirezionale, con altre più datate come *MVC* e derivate che invece scelgono un flusso bidirezionale. L'analisi sarà approfondita con degli esempi di codice per ognuno di questi paradigmi in modo da effettuare una dimostrazione pratica delle loro principali differenze.

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. Per fare un esempio semplicistico della questione, il banale cliccare su un bottone in un qualsiasi servizio online moderno, fa scaturire un evento che porta con sé dei dati che andranno ad apportare dei cambiamenti nello stato dell'applicazione. Ciascun elemento dell'interfaccia deve successivamente tener conto di questo cambiamento e modificarsi di conseguenza se necessario. Più questo flusso di dati è intenso e disorganizzato, più diventa complicato gestire i cambi di stato e la loro propagazione attraverso tutti i vari componenti.

Un'interfaccia utente quindi mette a disposizione dell'utilizzatore una immensa 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.

L'obbietto principale di questa tesi consiste nel fornire una presentazione delle varie architetture e dei vari approcci utilizzati attualmente per risolvere il problema della gestione del flusso di dati in applicazioni web con una codebase solitamente grande e complessa, comparando i loro relativi vantaggi e svantaggi. Verrà effettuata un'analisi approfondita del paradigma *MVC* (Model - View - Controller) lato front-end, spiegando le grosse differenze che si vengono a creare rispetto al suo utilizzo lato back-end e alla sua scarsa scalabilità derivata dall'uso di un flusso di dati bidirezionale tra Model e View. Verranno poi discusse le architetture di *Flux* e di *Redux* che implementano un flusso di dati unidirezionale per ovviare ai problemi del paradigma MVC. L'analisi coprirà le differenze di implementazione delle due librerie, di come Flux si basi su una struttura complessa ma estremamente funzionale oltre che versatile, e di come Redux la semplifichi attraverso pattern della programmazione funzionale per ottenere lo stesso livello di scalabilità con una semplicità decisamente maggiore.

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* ma sopratutto con la crescita smisurata della complessità dei servizi online. 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

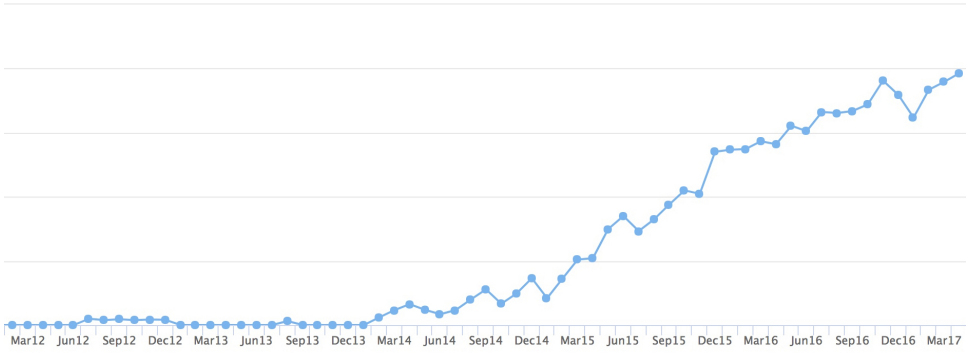

Figura 1.1: Grafico della popolarità di React (generato da hntrends.com).

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 verranno discusse le alternative attualmente più gettonate come quella 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 [1]. Stiamo tuttavia parlando di una applicazione statica costruita solamente in HTML dove gli unici input dell'utente erano limitati al navigare i documenti. Più avanti con l'evoluzione di HTML si iniziarono a vedere i primi elementi interattivi come bottoni e campi di testo. Tuttavia la svolta vera e propria avvenne il 5 maggio del 1995 con l'avvento di Javascript [2], il linguaggio di scripting che portò un primo accenno di dinamicità all'interno delle pagine web e che anche adesso è alla base di tutte le tecnologie front-end più nuove e potenti. Da qui in poi l'evoluzione andò 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 application

Andando avanti con gli anni si sono tentati diversi approcci per la creazione di applicazioni web. Un problema fondamentale è stato quello di dove posizionare la logica del servizio che fino a quel momento è stata sempre considerata come una parte del server specialmente per la mancanza di tecnologie adeguate lato client.

Con la maturazione degli strumenti client-side si è aperta una porta ad un nuovo tipo di applicazioni: le *Single-page application* (SPA). Una Single-page application è una applicazione web contenuta in una sola pagina e che sviluppa tutta la sua logica nel client.

Non è da poco che questo tipo di applicazioni hanno preso piede, tuttavia intorno agli anni 2000 non era Javascript la prima scelta come linguaggio di programmazione

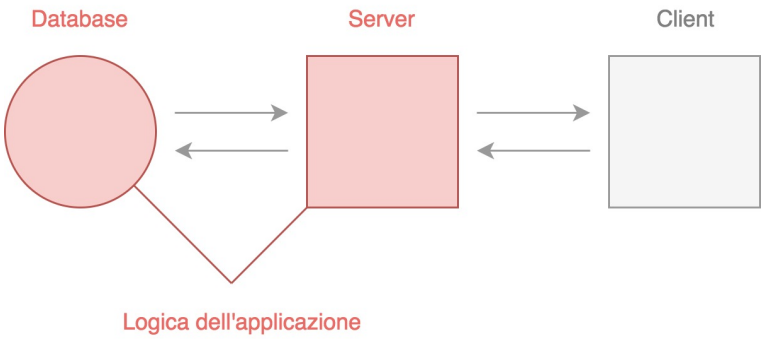

Figura 1.2: Esempio di una applicazione con la logica nel server.

ma Flash e Java Applets. Solo più avanti è riuscito a diventare competitivo abbastanza da deprecare entrambi i suoi predecessori per diversi motivi [3]:

- **Velocità di esecuzione** Javascript non ha bisogno di alcun ambiente esterno per essere eseguito consentendo di eliminare un livello di complessità all'applicazione.
- **Nessuna dipendenza esterna** Gli utenti non hanno bisogno di scaricare nessuna dipendenza esterna e quindi nessun plugin per eseguire il servizio.
- **Controllo sulla pagina web** Javascript ha pieno controllo sulla pagina web dove viene eseguito in quanto è tutt'uno con essa. Al contrario, Flash e Java, vengono inseriti in maniera "embedded" nella pagina e non hanno la possibilità di interagire con essa in maniera diretta causando un'esperienza meno fluida ed interattiva per l'utente.

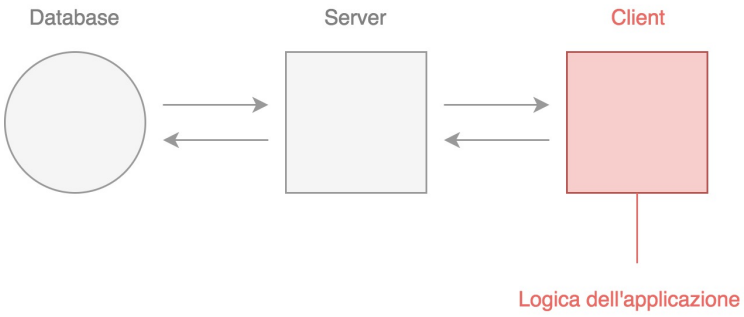

Figura 1.3: Esempio di una applicazione SPA.

La problematica della gestione del flusso dei dati è diventata estremamente rilevante con la comparsa delle SPA in 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.

> 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. [4]

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 (Model), la presentazione di quest'ultimi (View) e la logica che funge da tramite (Controller), incaricata di gestire le richieste degli utenti e gli eventi che accadono all'interno dell'applicazione fornendo i dati necessari alla costruzione della 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 [5].

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*[1]. 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 *Knockout*[2], 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 [6].

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 è la comunicazione bidirezionale: una View, tramite il Controller, modifica diversi Model i quali a loro volta aggiornano le relative View. Tenendo presente che stiamo parlando di applicazioni complesse e quindi con un grande numero di componenti, questo genera un effetto cascata di modifiche tra i vari Model e View 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 [7].

1.2.3 Framework ed architetture a flusso unidirezionale

Un ulteriore passo avanti nella storia delle applicazioni web è stata React[3], libreria per la creazione di interfacce utente sviluppata da Facebook che si basa su diverse tecnologie all'avanguardia e che permette grazie alla sua versatilità di strutturare architetture

[1] http://backbonejs.org
[2] http://knockoutjs.com
[3] https://facebook.github.io/react

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 [8]. 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*[4] 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. Questo tipo di approccio di permette di avere un flusso di dati facilmente analizzabile in quanto sappiamo esattamente dove arrivano in input e dove escono in output i dati di ogni singolo strato dell'architettura ma sopratutto ci permette di gestire in maniera chiara e pulita la sincronizzazione tra stato ed ogni singolo componente dell'interfaccia utente.

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*[5], 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 [9]. 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.

[4] https://facebook.github.io/flux
[5] http://redux.js.org

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

Quando si parla di applicazioni web, e più in particolare del loro front-end, ci si riferisce tuttavia a quel codice Javascript che viene scaricato ed interpretato dal browser dell'utente. Sorge quindi il serio problema di far eseguire un codice su una macchina il cui interprete potrebbe non supportarlo a dovere, ad esempio nel caso in cui si utilizzi ES6 in un browser che supporta solo uno standard più vecchio. Proprio per questo si utilizzano strumenti come *Babel*[2] che hanno la funzione di *transpiler*, ossia di compilare codice sorgente Javascript dallo standard ECMAScript6 a ECMAScript5 che è supportato dalla stragrande maggioranza dei browser.

Le funzionalità introdotte da ECMAScript6 sono molte, tuttavia qui parleremo solo di quelle che risulteranno propedeutiche per capire il codice dei capitoli successivi.

2.1.1 Let e Const

Una delle features introdotte, che probabilmente è anche una delle più incisive, riguarda l'assegnazione delle variabili. In ES5 la dichiarazione avveniva tramite la keyword *var* ed il loro scope era relativo alla funzione direttamente loro superiore.

In ES6 a questa si aggiungono anche let e *const* il cui scope è relativo al blocco in cui sono posizionate (e non alla funzione, come in *var*). La prima non ha nulla di particolare oltre ciò che abbiamo già detto, la seconda invece dichiara variabili costanti, ossia il cui valore non può mutare.

La keyword *const* verrà molto utilizzata nei successivi codici in quanto permette di utilizzare un concetto fondamentale dei linguaggi funzionali: l'immutabilità[3].

[1] https://www.ecma-international.org/ecma-262/6.0

[2] https://babeljs.io

```
function func() {
  var y = 2;
}

console.log(y); // y non esiste in questo scope
```

Codice 2.1: Esempio della dichiarazione di una variabile con *var*.

```
let x = 1;
{ let x = 2; } // x fuori da questo scope è comunque 1

const y = 2;
y = 'foo'; // Questo è un errore, abbiamo dichiarato una costante
```

Codice 2.2: Esempio della dichiarazione di variabili con *let* e *const*.

2.1.2 Arrow Function

In Javascript la parola chiave *this* si riferisce al contesto dove viene eseguita. Può essere in parte paragonata al modo in cui nella programmazione ad oggetti ci si riferisce all'istanza su cui stiamo lavorando. La grossa differenza tuttavia è che in Javascript cambiare il contesto e quindi avere un valore del *this* inaspettato è estremamente semplice e comporta non poche problematiche nell'analisi di un codice.

```
function Foo() {
  this.answer = 42;

  setInterval(function() {
    // Dentro questa callback il contesto è cambiato
    // e this.answer non esiste più
    console.log(this.answer);
  }, 1000);
}

var bar = new Foo(); // Il risultato sarà "undefined"
```

Codice 2.3: Esempio di comportamento inaspettato di *this*.

Questo capita perché il contesto dove viene eseguito il *this* non è più quello dell'oggetto *Foo*, ma un contesto separato ristretto alla funzione *setInterval*. Per evitare problemi di questo genere (che ovviamente non sono relativi solo alla funzione *setInterval* ma ad un vasto numero di altri casi particolari) ES6 mette a disposizione le *Arrow Function*, ossia funzioni anonime sintatticamente più corte che non sovrascrivono il *this* del contesto precedente permettendo la scrittura di un codice ad oggetti più sicuro e senza comportamenti inaspettati.

[3]Il concetto di immutabilità è un pilastro fondamentale della programmazione funzionale, e rappresenta un oggetto il cui stato non può essere modificato in alcun modo. Avere un elemento immutabile significa che l'unico modo per modificarlo è quello di crearne uno nuovo con le modifiche volute e modificare la referenza a quest'ultimo.

```
function Foo() {
  this.answer = 42;

  setInterval(() => {
    // Dentro questa callback il contesto NON è cambiato
    console.log(this.answer);
  }, 1000);
}

var bar = new Foo(); // Il risultato sarà "42"
```

Codice 2.4: Esempio di *Arrow Function*.

2.1.3 Parametri predefiniti e ad oggetti

ES6 permette di utilizzare valori predefiniti per i parametri delle funzioni che creiamo. Questi valori di default vengono assegnati quando i normali parametri sono *undefined*.

```
function func(foo = true, bar = 'bar') {
  if(!foo) return;
  console.log(bar);
}

func(); // Il risultato sarà "bar"
```

Codice 2.5: Esempio di utilizzo dei parametri di default.

Un altro miglioramento apportato da ES6 riguarda gli oggetti passati come parametri. E' ora possibile destrutturare un oggetto direttamente dai parametri durante la definizione di una funzione.

```
function func({ foo = true, bar = 'bar' }) {
  if(!foo) return;
  console.log(bar);
}

func({ foo: true, bar: 'cat' }); // Il risultato sarà "cat"
```

Codice 2.6: Esempio di destrutturazione di un oggetto passato come parametro.

2.1.4 Classi

In Javascript non esistono classi ma esistono invece oggetti con proprietà particolari che ci permettono di simulare ereditarietà e riusabilità. ES5 non possiede una vera e propria parola chiave per definire una classe, e quindi per creare un oggetto che si comporti come tale partiamo dal costruttore per poi aggiungere i metodi voluti. Questa tecnica prende il nome di Prototypal Inheritance [10].

```
function MyClass(value) {
  this.aProperty = value; // Questa è una proprietà dinamica della classe
}

MyClass.prototype.aMethod = function() {
  // Questo è un metodo della classe
};

var Instance = new MyClass('value');
```

Codice 2.7: Esempio di una classe in ES5.

ES6 mette a disposizione una sintassi molto più comprensibile e versatile per la gestione delle classi che rimane però solamente un abbellimento sopra il concetto di Prototypal Inheritance.

```
class MyClass {
  constructor(value) {
    this.aProperty = value; // Questa è una proprietà della classe
  }

  aMethod() {
    // Questo è un metodo della classe
  };
}

const Instance = new MyClass('value');
```

Codice 2.8: Esempio di una classe in ES6.

2.1.5 Struttura statica dei moduli

La struttura dei moduli di ES5, usando ad esempio la sintassi di *CommonJS*[4], è dinamica ciò significa che l'importazione e l'esportazione è mutabile a seconda dell'esecuzione del codice ed avviene quindi a run-time.

```
var a = true;

if (a) {
  dynamicLib = require('foo');
} else {
  dynamicLib = require('bar');
}
```

Codice 2.9: Esempio di importazione dinamica di un modulo in ES5.

[4]http://requirejs.org/docs/commonjs.html

Questo comporta che per analizzare le dipendenze di un progetto ed eliminare eventuali elementi non utilizzati, sia necessario eseguire il codice e monitorarne il comportamento in quanto non sappiamo a priori quali importazioni ed esportazioni verranno eseguite.

ES6 mette a disposizione un sistema di gestione dei moduli statico. Questo comporta la possibilità di analizzare le dipendenze di un codice a compile-time in quanto non sono presenti elementi dinamici che obbligano l'esecuzione del codice, con la conseguenza quindi di poter eliminare tutto ciò che non viene effettivamente utilizzato senza eseguire una singola riga di Javascript. Ovviamente in questo caso non è più possibile posizionare l'importazione o l'esportazione di un modulo all'interno di un costrutto o utilizzare variabili.

```
// library.js
export function foo() {
  console.log('foo');
}

// main.js
import { foo } from './library.js';
foo(); // Il risultato sarà "foo"
```

Codice 2.10: Esempio di importazione statica di un modulo in ES6.

2.2 Webpack

Abbiamo parlato nella sezione precedente di Webpack[5] e di come è in grado di trasformare ECMAScript 6 nella sua precedente versione supportata da quasi tutti i browser attuali. Tuttavia questa è solo una piccola caratteristica rispetto a quello che è veramente. Nel sito ufficiale Webpack è descritto come ≪A module bundler for modern JavaScript applications≫. In pratica si occupa di ricercare tutte le dipendenze dell'applicazione e raggrupparle in file aggregati. Per capire appieno questo concetto è bene analizzare la struttura di una applicazione Javascript moderna che normalmente consideriamo divisa in due parti ben distinte: il codice sorgente di base e i moduli (sia propri che di terzi) che implementano le varie funzioni. Un modulo è una unità dell'applicazione che contiene tutto il necessario per eseguire un aspetto o una particolare funzionalità di essa. Un modulo può includere dentro di sé una o più librerie, ossia delle collezioni di funzioni e metodi per risolvere dei particolari problemi. Quello che fa Webpack è analizzare il file Javascript relativo alla nostra applicazione, chiamato "Entry point", e creare un pacchetto con tutti i moduli e le librerie richieste affinché il servizio possa funzionare in maniera corretta.

Con Webpack diventa estremamente facile suddividere l'applicazione in file di dipendenze multipli che possono includere da codici sorgenti come moduli Javascript o CSS, fino ad immagini e font. E' anche possibile utilizzare *Loader*, ossia dei middleware, che prendono in input delle dipendenze specifiche e le trasformano a seconda di ciò che abbiamo bisogno (Il transpiler da ES6 ad ES5 fa esattamente questo prendendo in input ogni file Javascript).

[5] https://webpack.js.org

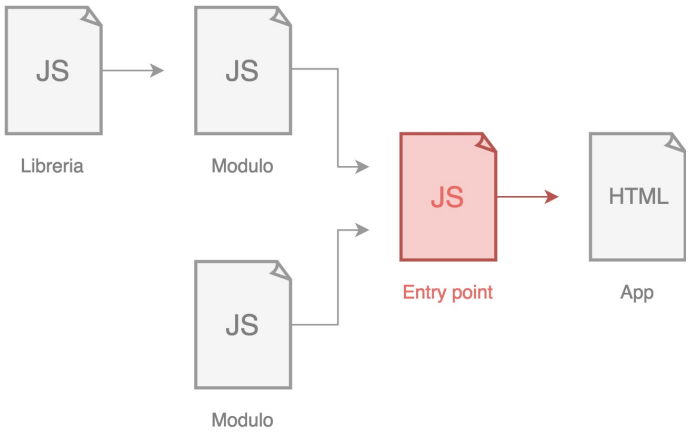

Figura 2.1: Rappresentazione del sistema di impacchettamento di Webpack.

2.2.1 Hot Module Replacement

Un aspetto molto interessante di Webpack riguarda l'*Hot Module Replacement* (HMR) che si occupa di aggiungere o rimuovere i pacchetti di dipendenze generati nel run-time dell'applicazione senza un aggiornamento completo della pagina. Questo è molto utile in fase di development in quanto consente di mantenere lo stato di una applicazione anche dopo aver effettuato modifiche al sorgente ed avere le nuove caratteristiche disponibili in maniera molto più veloce del normale aggiornando solo ciò che è necessario.

2.2.2 Tree Shaking

La tecnica del *Tree Shaking* permette di eliminare il codice inutilizzato all'interno della codebase. Quello che fa Webpack è andare ad analizzare la struttura di *Import* ed *Export* del sorgente che, come abbiamo detto precedentemente nella sezione riguardante ES6, è statica ossia è possibile analizzarla a *compile-time* senza la necessità di eseguire il codice. Una volta trovati gli elementi che non vengono utilizzati essi vengono classificati come "codice morto" e vengono marcati attraverso adeguati commenti. Webpack non si occupa di eliminare questi elementi ma affida il compito ad un eventuale *Minifier* (come ad esempio *UglifyJS*) che si occupa di ottimizzare il codice Javascript.

2.3 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* [11].

```
// Componente HelloWorld
const HelloWorld = (props) => {
  return (
    <h1>Hello World!</h1>
  );
};

// Componente che si compone con HelloWorld
const CompositedHelloWorld = (props) => {
  return (
    <div>
      <HelloWorld />
      <i>This is an HelloWorld component</i>
    </div>
  );
}
```

Codice 2.11: Esempio di composizione tra componenti React.

2.3.1 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 [12].

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*[6] 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 [13].

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

[6] 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.

Figura 2.2: Posizione del Virtual DOM all'interno di una applicazione React.

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.

2.3.2 Server-side rendering

React è una libreria *isomorfica*, cioè è in grado di essere eseguita sia lato client che lato server. Essa tra vantaggio da NodeJS ed dal 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 SPA 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 altre 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 SPA già avviata e fornita di tutti i dati di cui avrebbe normalmente bisogno. Esistono tuttavia delle problematiche non proprio da sottovalutare:

- L'utente non può comunque effettuare alcun tipo di interazione con l'applicazione finché React non viene scaricato anche dal client;
- Il primo *TTFB (Time To First Byte)*[7] è generalmente più lento del normale in quanto ci sono più computazioni lato server da effettuare per compilare il codice React;
- Le computazioni server-side potrebbero non essere veloci come quelle effettuate client-side causando un considerevole calo di prestazioni [15].

[7] Il Time To First Byte è solitamente utilizzato come misura di quanto veloce un server web risponde ad una richiesta, ed è in pratica la durata che intercorre tra la richiesta effettuata dall'utente e il primo byte di risposta del server [14].

3. Architettura MVC

L'architettura MVC, per la prima volta presentata nel 1988 da Glenn E. Krasner e Stephen T. Pope nel loro articolo "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", rappresenta un caposaldo della programmazione e una delle architetture più utilizzate sia lato front-end che back-end.

L'elemento base su cui si fonda MVC è la modularità del codice e la portabilità dei vari componenti che formano l'applicazione. Per ottenere ciò si è teorizzato di dividere quest'ultima in tre parti ben distinte: le parti che ne rappresentano la struttura astratta, il modo in cui queste vengono presentate e il modo con cui l'utente ci interagisce. Questa divisione permette l'isolamento delle unità funzionali dell'applicazione in modo da facilitarne il debugging e la scalabilità, oltre al fatto di migliorare la riusabilità dei componenti creati seguendo questo pattern [16].

3.1 Divisione dei compiti

La divisione dei compiti nell'architettura MVC, come è stato detto precedentemente, avviene attraverso Model, View e Controller. Questi tre elementi fondamentali dell'applicazione sono interconnessi e ognuno ha un compito specifico, isolato dagli altri, a cui deve attenersi.

Il Model rappresenta i dati che formano tutta o una parte dello stato dell'applicazione e come essi mutano. E' possibile immaginare questo componente come un oggetto che identifica un elemento specifico del servizio e ne descrive come il suo stato muta ad una eventuale azione. Il Model è "cieco", nel senso che tutto ciò che fa è mantenere dati e descrivere come essi mutano non avendo in alcun modo né la possibilità di conoscere in che modo influenzano l'applicazione né la capacità autonoma di effettuare modifiche su di essi se non sotto specifica istruzione del Controller.

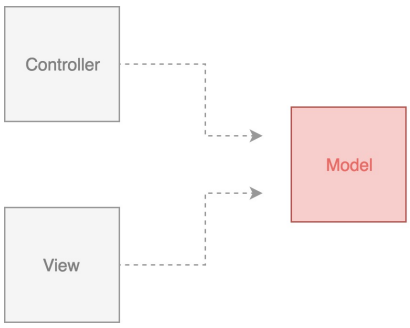

Figura 3.1: Architettura MVC classica.

I componenti che compongono la View hanno l'obiettivo di rappresentare i dati del Model in una forma facilmente comprensibile all'utente finale. Nell'architettura standard MVC il ruolo della View era piuttosto dinamico e organizzava i suoi aggiornamenti discutendo direttamente con il Model ed ottenendo i dati da esso, lasciando al Controller il compito di gestire gli input dell'utente. In questo modo l'architettura era organizzata con il Model come componente centrale togliendo importanza al Controller.

Andando avanti nel tempo e con la comparsa di sempre più framework ispirati a questo pattern, l'evoluzione ha cambiato leggermente le regole standard dettando che la View dovrebbe avere il meno possibile un ruolo "attivo" di lettura degli aggiornamenti del Model, e che la gran parte delle informazioni dovrebbe passare attraverso il Controller che diventa così l'elemento principale addetto alla logica [17].

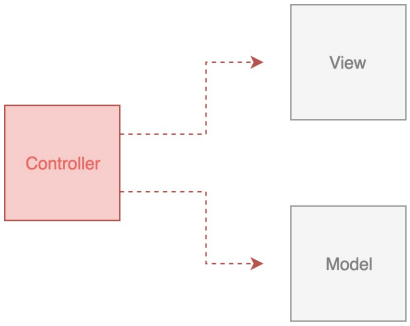

Figura 3.2: Architettura MVC recente.

Il Controller, nella versione classica dell'architettura MVC, si occupa solo ed esclusivamente di gestire gli input forniti dall'utente, trasmettendo le informazioni ricevute al Model. E' bene tenere presente che ogni Controller viene eseguito solamente a seguito di una azione che l'utente ha intrapreso dalla View e che quest'ultima ha un collegamento diretto con il Model comunicando attraverso il pattern Observer per adeguarsi ai vari cambiamenti dello stato. Nelle versioni più recenti dell'architettura MVC invece, il Controller assume sempre di più la parte centrale incaricata di organizzare e collegare le altre due. Non solo si occupa di gestire i vari eventi ed input generati dall'utente, ma si occupa anche di fornire e filtrare i dati del Model alla View, sottolineando la loro natura in questo caso più statica.

3.2 MVC nel front-end

L'architettura MVC è nata per risolvere il problema dell'organizzazione e della struttura del codice in applicazioni desktop scritte in Smalltalk-80. La sua versatilità e potenza ha tuttavia fatto si che anche tutt'ora questo paradigma venga utilizzato nella stragrande maggioranza delle applicazioni, specialmente lato back-end. Con la nascita delle SPA e con lo spostamento della logica sempre più verso il front-end tuttavia si sono presentati diversi problemi con l'architettura MVC classica i quali hanno spinto quest'ultima a modificarsi ed adattarsi al nuovo ambiente.

Lato front-end l'architettura MVC assume diverse forme a seconda del tipo di caratteristiche di cui abbiamo bisogno. Molti framework in Javascript tentano di includere il lavoro del Controller all'interno della View, altri al contrario rendono quest'ultima completamente passiva spostando tutta la logica al Controller, aggiungono semplicemente nuovi livelli di divisione. Queste differenti architetture derivate vengono associate ad una macro-categoria chiamata *MV** (l'asterisco al posto della "C" di Controller sta a significare che questo è l'elemento che viene solitamente sostituito o modificato) dalla quale è possibile estrapolare due tra i più famosi ed utilizzati paradigmi lato front-end: MVP e MVVM [18].

3.2.1 MVP

L'architettura MVP nasceva nel 1990 nell'azienda Talingent e veniva utilizzata per lo sviluppo di applicazioni in C++ e Java [19]. E' stato successivamente ripresa da vari framework Javascript tra cui uno dei più famosi è sicuramente Backbone.js. Mentre nella versione classica di MVC, la View osservava il Model e si modificava di conseguenza, l'evoluzione di questo pattern ha fatto si che il ruolo di osservatore passasse sempre

più al Controller, ora chiamato Presenter, che costituisce un vero e proprio elemento centrale ed intermediario, eliminando quindi completamente la logica dalla View.

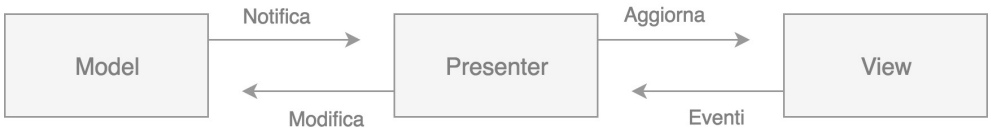

Figura 3.3: Architettura MVP.

MVP è un paradigma utilizzato specialmente per applicazioni a livello enterprise, con View complesse e grandi quantità di interazioni da parte dell'utente in quanto tutta la logica risiede nel Presenter rendendo l'applicazione semplice da testare e staccando la View dal resto rendendola particolarmente riutilizzabile all'interno del codice. Tuttavia le differenze tra le architetture MVC e MVP sono solo a livello semantico, interpretando cioè diversamente un componente comune ad entrambe. Molti problemi più radicati che troviamo nella prima quindi si vengono comunque a presentare nella seconda [18].

3.2.2 MVVM

L'architettura MVVM è stata per prima definita da Microsoft in un post del 2005 di John Grossman [20]. E' nata come una architettura per la creazione di applicazioni con Windows Presentation Foundation (WPF) ed è stata successivamente implementata con un discreto successo in framework Javascript come Knockout.js.

Questo paradigma si basa completamente sulla sincronizzazione della View attraverso il pattern Observer. La logica dell'applicazione quindi questa volta è posizionata quasi interamente nella View e, mentre il Model si comporta come da norma solamente da "memoria", questa ne osserva i cambiamenti e si aggiorna di conseguenza. La vera differenza è che in questo caso le azioni dell'utente sono gestite direttamente dalla View, in quanto il vincolo posto sui dati è bidirezionale (modificando l'elemento in input nella View viene automaticamente modificato il valore referenziato). La View tuttavia non può leggere direttamente i dati di cui ha bisogno dal Model, i quali nella maggior parte dei casi sono grezzi e hanno bisogno di essere filtrati e lavorati. Per questo è stato creato un nuovo componente, che sostituisce il Controller, chiamato ViewModel il quale si pone come livello passivo intermedio tra Model e View, ed estrapola i dati utili dal primo e li rende disponibili in maniera comprensibile al secondo, oltre anche a fornire funzioni utili alla gestione dello stato e degli eventi [18].

Figura 3.4: Architettura MVVM.

Uno dei punti forti dell'architettura MVVM implementata dai più recenti framework Javascript riguarda l'astrazione della View per l'utilizzo dei vincoli di dati bidirezionali (Two-ways Data Bindings) del ViewModel direttamente all'interno del DOM.

```
<input data-bind="value: contactName, valueUpdate: 'keyup'" />
```

Codice 3.1: Esempio di vincolo di dati nel DOM.

Da una parte questi facilitano enormemente l'implementazione dell'interfaccia e consentono di ridurre in maniera considerevole la logica da implementare a parte all'interno del ViewModel, dall'altra sono considerati un tipo di programmazione obsoleta che mischia la logica con il linguaggio di markup.

3.3 Il flusso dei dati

MVC, insieme alle sue architetture derivate, è un paradigma nel quale il flusso di dati avviene in maniera bidirezionale. L'utente esegue un'azione da uno specifico componente della View, la quale viene interpretata ed eseguita dalla relativa sezione dove risiede la logica dell'interfaccia (ad esempio quindi dal Controller se si parla di MVC classico o dal Presenter nel MVP) la quale apporterà le dovute modifiche ai dati presenti all'interno del Model causando una notifica da parte di quest'ultimo a tutti gli altri componenti della View segnalando l'avvenuto cambiamento. Il flusso, come possiamo vedere nell'immagine 3.3, avviene quindi sia dal Model alla View che vice versa.

3.3.1 Applicatione esempio

Per l'analisi più in dettaglio di ciò che è stato appena descritto, verrà utilizzata una applicazione esempio fine a se stessa costruita utilizzando il framework *Backbone.js* e quindi l'architettura MVP. Tutte le affermazioni che verranno fatte tuttavia non riguarderanno solo ed esclusivamente questa architettura, ma si estenderanno anche alle altre derivate da MVC, in quando descriveranno problematiche radicate nel paradigma.

L'applicazione (figura 3.5) rappresenta un semplice database di libri con la possibilità di aggiungerne di nuovi tramite il form in alto, ed eliminare quelli presenti andando con il mouse sopra un libro e cliccando alla comparsa del simbolo "×". Il codice è scritto utilizzando la sintassi e il sistema di moduli di ES6 usando Webpack con Babel per creare il file Javascript compilato e completo di dipendenze. Anche il CSS per comodità è stato aggiunto nel pacchetto ed automaticamente applicato alla pagina dell'applicazione attraverso i loaders *css-loader*[1] e *style-loader*[2]. Le uniche due dipendenze richieste da *Backbone.js* consistono in: *jQuery*[3], libreria per la manipolazione del DOM e altre utili funzioni come animazioni, gestione degli eventi e chiamate Ajax; *Underscore.js*[4], una raccolta di metodi per gestione di oggetti, array e collezioni varie attraverso tecniche utilizzate nel paradigma di programmazione funzionale.

Non ci sono collegamenti espliciti ad un eventuale back-end in quanto tale aspetto non è direttamente collegato al discorso di questo documento, tuttavia alcuni riferimenti riguardo come comunicare nel modo più giusto con esso attraverso *Backbone.js* verranno fatti in seguito.

[1] https://github.com/webpack-contrib/style-loader
[2] https://github.com/webpack-contrib/css-loader
[3] https://jquery.com/
[4] http://underscorejs.org/

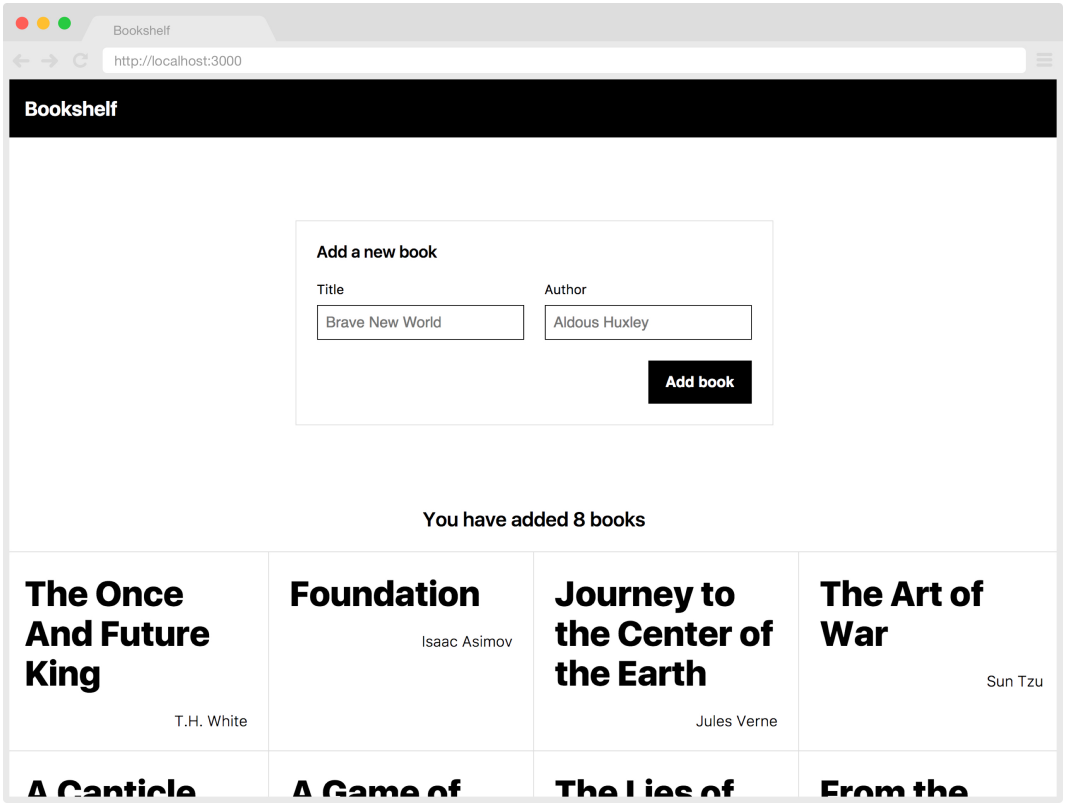

Figura 3.5: Applicatione esempio per un database di libri.

3.3.2 Model

Il Model di una applicazione scritta con *Backbone.js* è un oggetto che estende `Backbone.Model` e che racchiude dentro di se dei dati strutturati. Nel caso dell'applicazione presentata, il Model più basilare descrive lo stato di un singolo libro che è composto da due parametri: "title" e "author", dando loro un valore di default.

```
// BookModel.js
const BookModel = Backbone.Model.extend({
  defaults: function() {
    return {
      title: 'No title',
      author: 'No author'
    };
  }
});
```

Codice 3.2: Model dell'applicazione relativo ad un libro.

Il modo migliore per descrivere lo stato dell'applicazione tuttavia è attraverso una lista di libri, è necessario quindi creare un tipo di Model che rappresenti questo concetto in maniera opportuna. *Backbone.js* mette a disposizione la classe `Backbone.Collection` che rappresenta esattamente una "collezione di Model" e gestisce in maniera automatica tutti gli elementi al suo interno.

```
// BookshelfCollection.js
const BookshelfCollection = Backbone.Collection.extend({
  model: BookModel,
});

// We need a pre-populated singleton
const BCInstance = new BookshelfCollection();
BCInstance.add(bookshelfDb);
```

Codice 3.3: Model dell'applicazione relativo ad una lista di libri.

La classe Collection è implementata come se fosse una normale lista ed ogni qualvolta lei o un suo elemento viene modificato, si occupa di propagare il corrispettivo evento a tutte le View ad essa collegate in modo da permettere loro di aggiornarsi di conseguenza.

Bibliografia

[1] Dino Grandoni. World's first website, created by tim berners-lee in 1991, is still up and running on 21st birthday, 2012. URL http://www.huffingtonpost.com/2012/08/06/worlds-first-website_n_1747476.html.

[2] W3C. A short history of javascript, 2012. URL https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript.

[3] M.S. Mikowski and J.C. Powell. *Single Page Web Applications: JavaScript End-to-end*. Manning, 2013. ISBN 9781617290756. URL https://books.google.it/books?id=JqNuMAEACAAJ.

[4] Mikito Takada. Single page apps in depth, 2013. URL http://singlepageappbook.com/.

[5] Bill Venners. The importance of model-view separation, a conversation with terence parr, 2008. URL http://www.artima.com/lejava/articles/stringtemplate.html.

[6] Shailendra Chauhan. Understanding mvc, mvp and mvvm design patterns, 2016. URL http://www.dotnettricks.com/learn/designpatterns/understanding-mvc-mvp-and-mvvm-design-patterns.

[7] Amir Salihefendic. Flux vs. mvc (design patterns), 2015. URL https://medium.com/hacking-and-gonzo/flux-vs-mvc-design-patterns-57b28c0f71b7.

[8] Samer Buna. Yes, react is taking over front-end development. the question is why., 2015. URL https://medium.freecodecamp.org/yes-react-is-taking-over-front-end-development-the-question-is-why-40837af8ab76.

[9] Dan Abramov. Risposta su stackoverflow alla domanda "why use redux over facebook flux", 2015. URL https://stackoverflow.com/a/32920459.

[10] Cory Rylan. Javascript prototypal inheritance, 2016. URL https://coryrylan.com/blog/javascript-prototypal-inheritance.

[11] Ken Wheeler. Learning react.js: Getting started and concepts, 2014. URL https://scotch.io/tutorials/learning-react-getting-started-and-concepts.

[12] Jonathan Robie Philippe Le Hégaret, Lauren Wood. What is the document object model?, 2000. URL https://www.w3.org/TR/DOM-Level-2-Core/introduction.html.

[13] Rupesh Mishra. Virtual dom in reactjs, 2017. URL https://hackernoon.com/virtual-dom-in-reactjs-43a3fdb1d130.

[14] John Graham-Cumming. Stop worrying about time to first byte (ttfb), 2012. URL https://blog.cloudflare.com/ttfb-time-to-first-byte-considered-meaningles/.

[15] Alex Grigoryan. The benefits of server side rendering over client side rendering, 2017. URL https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8.

[16] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.* URL http://dl.acm.org/citation.cfm?id=50757.50759.

[17] Callum Hopkins. The mvc pattern and php, part 1, 2013. URL https://www.sitepoint.com/the-mvc-pattern-and-php-1/.

[18] Addy Osmani. Journey through the javascript mvc jungle, 2012. URL https://www.smashingmagazine.com/2012/07/journey-through-the-javascript-mvc-jungle/.

[19] Mike Potel. Mvp: Model-view-presenter the taligent programming model for c++ and java. *Taligent Inc*, 1996.

[20] John Gossman. Introduction to model/view/viewmodel pattern for building wpf apps, 2005. URL https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/.