**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

# Sommario

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'obiettivo 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 complesse, 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 (Figura 1.1). 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

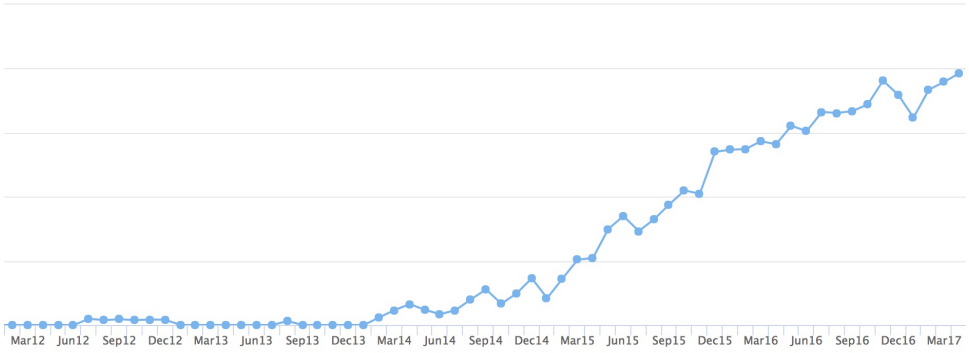


**Figura 1.1:** Grafico della popolarità di React (generato da hntrends.com).

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 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 (Figura 1.2).

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 (Figura 1.3).

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

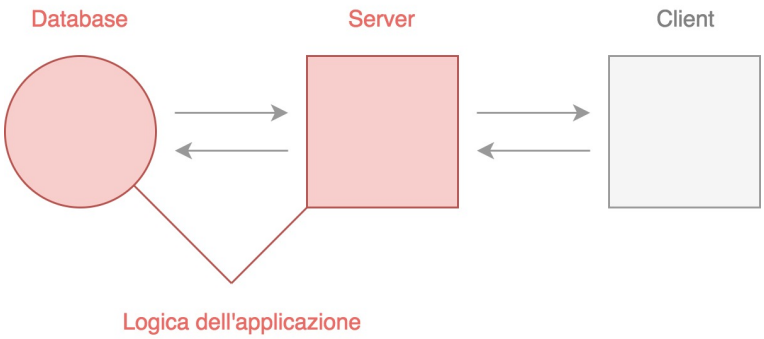


**Figura 1.2:** Esempio di una applicazione con la logica nel server.

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.

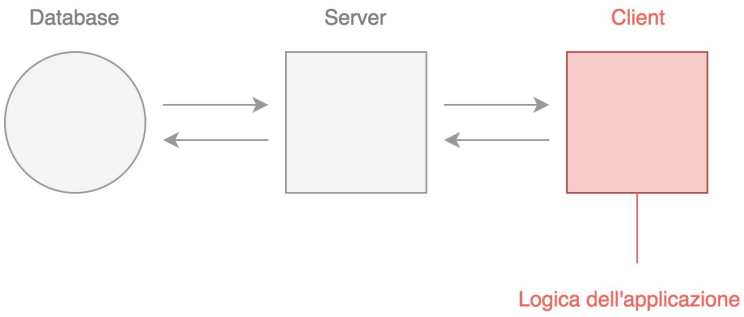


**Figura 1.3:** Esempio di una applicazione SPA.

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 dell'applicazione (Model), la presentazione di quest'ultimi (View) e la logica che funge da tramite (Controller), incaricata di gestire gli eventi innescati dall'utente. 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, e la maggior parte di questi tendono a sviluppare questa architettura in maniera leggermente differente, con i propri vantaggi e svantaggi a seconda dei cambiamenti che apportano e del tipo di applicazione che si deve costruire. Una di queste è MVP (Model - View - Presenter) utilizzata da *Backbone.js*[1]. In questa il Presenter, che sostituisce il Controller, ha una responsabilità maggiore di quest'ultimo e oltre che gestire gli eventi dell'utente si occupa di fornire i dati necessari alla generazione della View. 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 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.

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

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 venire 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 da uno standard nuovo e poco supportato ad ECMAScript5 (o inferiore) che è implementato 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 (Codice 2.1). E' bene notare che questi tipi di variabili sono legate al contesto (e quindi alla funzione) dove sono definite e non al blocco di codice `{ /* ... */ }` in cui sono dichiarate, a differenza di molti altri linguaggi di programmazione.

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,

[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*.

ossia il cui valore non può mutare attraverso una riassegnazione e non possono essere ridichiarate (Codice 2.2).

```
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*.

La keyword *const* verrà molto utilizzata nei successivi codici in quanto permette di utilizzare un concetto fondamentale dei linguaggi funzionali: l'immutabilità[3], oltre al fatto che utilizzare delle variabili non mutabili riduce il rischio di errori e semplifica il debugging del codice.

### 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*.

[3]Il concetto di immutabilità è un pilastro fondamentale della programmazione funzionale. 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.

Si consideri il Codice 2.3. Il problema illustrato si verifica 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 (Codice 2.4).

```
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* (Codice 2.5).

```
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 (Codice 2.6).

### 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 (Codice 2.7). Questa tecnica prende il nome di Prototypal Inheritance [10].

```
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.

```
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 (Codice 2.8).

```
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

Un modulo in Javascript corrisponde ad un'unità autonoma dell'applicazione che contiene tutto il necessario per eseguire una particolare e distinta funzionalità di questa. 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 (Codice 2.9). Questo comporta che per analizzare

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

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.

```
var a = true;

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

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

ES6 mette a disposizione un sistema di gestione dei moduli statico (Codice 2.10). 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. Come abbiamo detto nella Sezione 2.1.5, un modulo è un'unità autonoma dell'applicazione che ne rappresenta una funzionalità distinta. Un modulo può includere dentro di sé una o più librerie, ossia

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

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 (Figura 2.1).

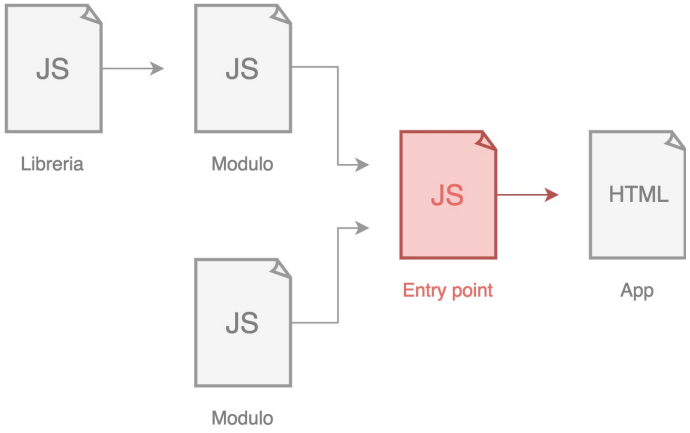


**Figura 2.1:** Rappresentazione del sistema di impacchettamento di Webpack.

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).

### 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*[6]) che si occupa di ottimizzare il codice Javascript.

[6]https://github.com/mishoo/UglifyJS

## 2.3 React

React è una libreria scritta ed utilizzata da Facebook per la creazione di interfacce utente interattive in maniera funzionale ed altamente scalabile che fa uso di diverse tecnologie all'avanguardia come il *Virtual DOM* ed il *Server-side Rendering* [11]. Si basa sul concetto di "componente" come elemento base fondamentale che permette di dividere l'interfaccia in pezzi autonomi e riutilizzabili. Un componente non è altro che una funzione Javascript che accetta delle proprietà arbitrarie (solitamente chiamate "props") e ritorna un elemento React generalmente descritto tramite codice JSX, un'astrazione sopra il normale DOM. Un componente può anche essere definito estendendo la classe `React.Component`, ottenendo diverse funzionalità aggiuntive come eventi sul suo ciclo di vita e gestione del suo stato interno. 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 (Codice 2.11).

```
// 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.

Un pattern molto utilizzato quando si scrive componenti React consiste nel suddividere questi in due categorie fondamentali: *Componenti di presentazione* e *Componenti contenitori*. I primi sono classificati come componenti senza stato interno (*Stateless components*), definiscono lo stile degli elementi, sono estremamente riutilizzabili e non dipendono direttamente dalla logica dell'applicazione ma dalle proprietà ottenute dai componenti superiori di cui fanno parte. Un esempio di componente di presentazione potrebbe essere un bottone, un campo di testo, o una lista. I componenti contenitori invece sono incaricati di gestire la logica dell'applicazione e forniscono i dati e gli eventi ai componenti di presentazione. Questo approccio migliora la separazione delle occupazioni all'interno dell'interfaccia definendo i componenti contenitori come dei supercomponenti con uno stato interno ed un ruolo attivo nell'architettura dell'applicazione che si compongono attraverso i vari componenti di presentazione autonomi.

### 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 (Figura 2.2). 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*[7] 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.


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

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.

### 2.3.2 Server-side rendering

React è una libreria *isomorfica*, cioè è in grado di essere eseguita sia lato client che lato server. Essa trae 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

[7] Il design pattern Observer si compone 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.

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)*[8] 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].

[8] 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. Il flusso bidirezionale dell'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" [16], 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 (Figura 3.1).

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 cambiano, 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.

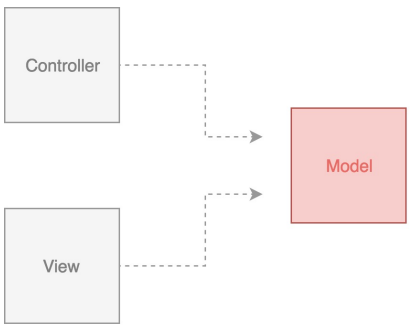


**Figura 3.1:** Architettura MVC classica.

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 sempre più frequente 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].

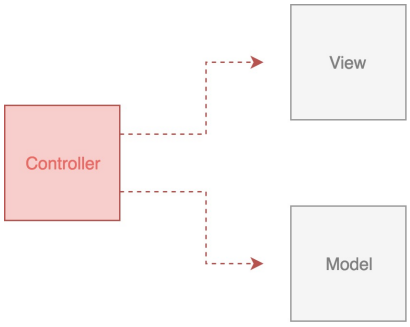


**Figura 3.2:** Architettura MVC recente.

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 (Figura 3.2). 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 sì 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, altri ancora 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' stata 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 sì 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).

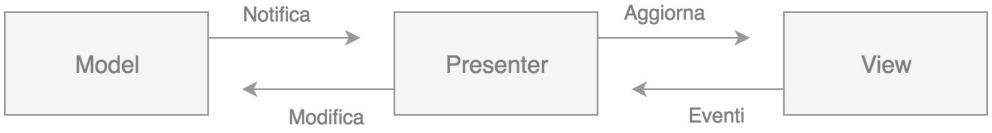


**Figura 3.3:** Architettura MVP.

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[1].

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).


**Figura 3.4:** Architettura MVVM.

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 (Figura 3.4) il quale si pone come livello passivo intermedio tra Model e

[1] http://knockoutjs.com

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].

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 (anche chiamati *Two-ways data bindings*) del ViewModel direttamente all'interno del DOM (Codice 3.1).

```
<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 dalla View, che 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 Analisi dell'applicazione

Per analizzare più in dettaglio 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[2] nella 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 (in coppia con il plugin *Env preset*[3], utilizzato per determinare automaticamente la versione di ES utilizzata) 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*[4] e *style-loader*[5]. Le uniche due dipendenze richieste da *Backbone.js* consistono in: *jQuery*[6], libreria per

[2] Codice completo su https://github.com/diegopq/unicam-thesis-examples.
[3] https://babeljs.io/docs/plugins/preset-env
[4] https://github.com/webpack-contrib/style-loader
[5] https://github.com/webpack-contrib/css-loader
[6] https://jquery.com/

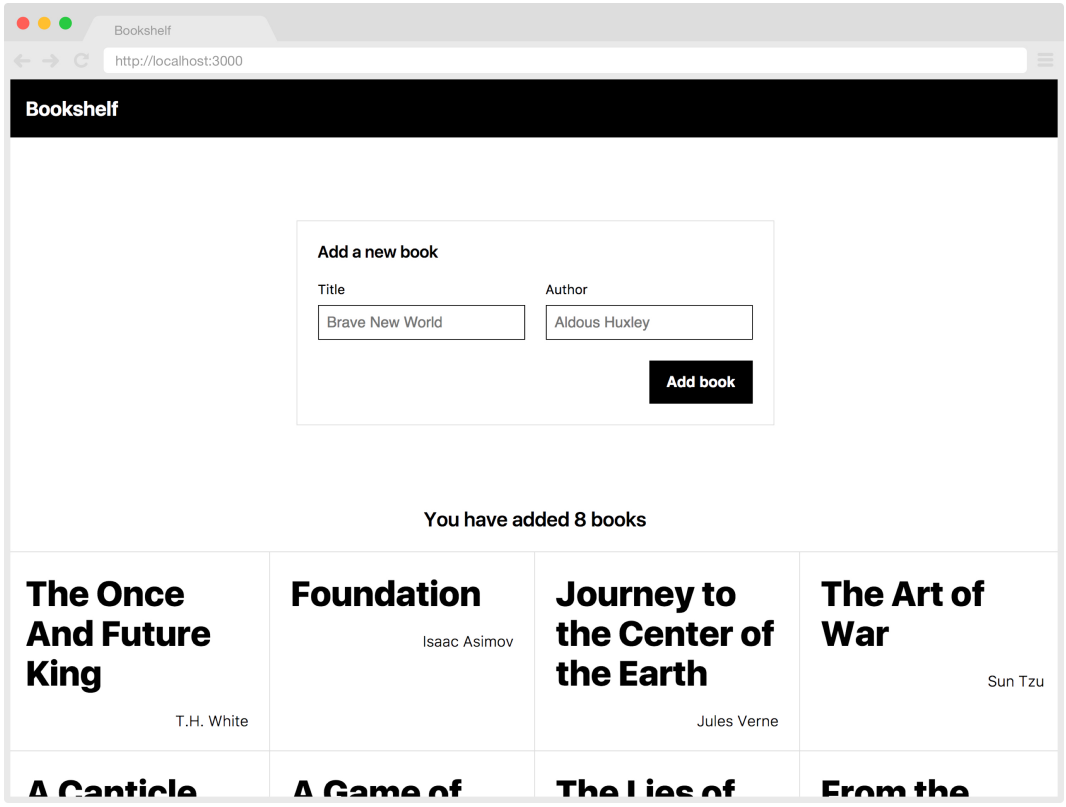


**Figura 3.5:** Applicatione esempio per un database di libri.

la manipolazione del DOM che mette a disposizione altre utili funzioni come animazioni, gestione degli eventi e chiamate Ajax; *Underscore.js*[7], una raccolta di metodi per gestione di oggetti, array e collezioni varie attraverso tecniche utilizzate nel paradigma di programmazione funzionale.

Di seguito verranno analizzate le sezioni fondamentali dell'applicazione con le relative parti di codice più importanti per capire come questo framework riesca a gestire la problematica del flusso di dati.

### Model

Il Model di una applicazione scritta con *Backbone.js* è un oggetto che estende `Backbone.Model` e che racchiude dentro di sé 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 (Codice 3.2).

Il modo definitivo 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 riconoscendo come istanza di `BookModel` qualsiasi elemento aggiunto ad essa (Codice 3.3).

La classe Collection è implementata come se fosse una normale collezione di elementi (o lista) e *Backbone.js* ci mette a disposizione tutti i metodi per interagire con essa.

[7] http://underscorejs.org/

```
// 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.

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

const BCInstance = new BookshelfCollection();
BCInstance.add(bookshelfDb);
```

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

Ogni qualvolta questa lista o un suo elemento viene modificato, la stessa si occupa di propagare il corrispettivo evento a tutte le View ad essa collegate in modo da permettere loro di aggiornarsi di conseguenza. Viene poi creata un'istanza di questa collezione, popolata con dei libri di default letti dal file JSON `db.json` ed esportata ottenendo così un Singleton[8] che rappresenta l'intero stato dell'applicazione condiviso globalmente tra tutti i componenti.

### View

Una View in *Backbone.js* corrisponde al codice HTML dell'applicazione (Codice 3.4). Oltre che gestire questo, tuttavia, permette di definire dei piccoli "modelli" all'interno di un tag `<script type='text/template'>` che verranno poi popolati in modo dinamico attraverso un *Template engine*, ossia una libreria che combina un template con un modello di dati, come ad esempio *Mustache*[9] o *EJS*[10].

In questa applicazione viene utilizzato *Underscore.js*, che è già una dipendenza di default dell'applicazione e mette a disposizione un piccolo sistema di templating basilare.

[8] Il "Singleton" è un design pattern che restringe ad uno il numero massimo di istanze che una classe può avere. E' utilizzato molto spesso quando c'è bisogno di avere un oggetto condiviso globalmente all'interno di una applicazione.

[9] https://mustache.github.io

[10] http://ejs.co

```
<!-- index.html -->
<script type="text/template" id="js-book-template">
  <div class="c-singlebook__title"><%- title %></div>
  <div class="c-singlebook__author u-txright u-mt3"><%- author %></div>
</script>

<script type="text/template" id="js-counter-template">
  You have added <%- count %> books
</script>
```

**Codice 3.4:** Templates dell'applicazione.

### Presenter

L'oggetto che estende `Backbone.View` è il componente centrale di Backbone.js e rappresenta il Presenter nell'architettura MVP. Ha un ruolo centrale sia all'interno dell'applicazione per le sue innumerevoli funzioni, sia nell'architettura in quanto si pone tra gli altri due componenti e li coordina in maniera attiva.

```
// CounterView.js
const CounterView = Backbone.View.extend({
  tagName: 'div',
  className: 'c-heading c-heading--1 u-txcenter',

  template: _.template($('#js-counter-template').html()),

  initialize: function() {
    this.listenTo(this.collection, 'change', this.render);
  },

  render: function() {
    this.$el.html(this.template({ count: this.collection.length }));
    return this;
  }
});
```

**Codice 3.5:** Presenter dell'applicazione.

La classe `Backbone.View` rappresenta, in pratica, un elemento che verrà poi aggiunto al DOM dell'applicazione tramite il metodo `render()`. Il tipo di elemento e le sue classi sono descritte attraverso le proprietà `tagName` e `className`. Il contenuto invece viene definito in principio all'interno della stessa funzione `render`, e nel caso del Codice 3.5 viene utilizzato il template engine messo a disposizione da *Underscore.js* per popolare il template selezionato. La gestione degli eventi è uno dei compiti più delicati del Presenter. Esistono due tipi di eventi da gestire:

- **Eventi dal Model**: sono quegli eventi che partono dal Model a seguito di una modifica di quest'ultimo (l'aggiunta o la modifica di un elemento all'interno di una collezione).

- **Eventi dalla View**: sono quegli eventi che vengono innescati da un certo comportamento dell'utente che utilizza la View dell'applicazione (il click di un pulsante o la scrittura all'interno di un campo di testo).

Gli eventi del primo tipo vengono gestiti all'interno del metodo `initialize()`, attraverso la funzione `View.listenTo()` passando ad essa il Model o la Collection da osservare (è possibile utilizzare `this.collection` o `this.model` a seconda del tipo di Model assegnato alla View al suo avvio), il tipo di evento e la relativa funzione che dovrà essere eseguita. Per la gestione degli eventi che provengono dalla View invece si utilizza l'oggetto `events` specificando il tipo di evento da controllare, il selettore per identificare l'elemento vincolato nel DOM e la funzione da chiamare all'innesco dell'evento (Codice 3.6). *Backbone.js* si occuperà poi di assegnare e gestire questi eventi a ciascun elemento del DOM selezionato, risolvendo anche il classico problema di assegnare eventi ad elementi creati successivamente in maniera dinamica.

```
// BookshelfView.js
const BookshelfView = Backbone.View.extend({
  // ...

  events: {
    'click .js-remove': 'removeBook'
  },

  // ...

  removeBook: function(e) {
    const cid = e.target.dataset.cid;
    this.collection.remove(cid);
  }
});
```

**Codice 3.6:** Presenter con la gestione degli eventi dalla View.

### 3.3.2 Effetto cascata

Come presentato nei capitoli precedenti, MVC è un'ottima architettura per strutturare il codice, tuttavia soffre di problemi non indifferenti quando utilizzata lato front-end ed in applicazioni di una certa complessità.

L'esempio riportato nella Figura 3.6 rappresenta il tallone d'Achille di questo paradigma (e dei suoi derivati). Un'azione scaturita dall'utente, innesca un evento nel Controller (o qualsiasi altro componente che si occupa di gestire gli eventi) il quale apporta le opportune modifiche al Model, che a sua volta notifica gli altri componenti dell'aggiornamento avvenuto facendo scaturire nuovi eventi e nuove modifiche. Questo comportamento viene chiamato "Cascading effect" e causa estrema difficolta nel debugging del codice specialmente per quanto riguarda l'analisi degli effetti che un evento causa all'applicazione.

Un pratico esempio di questo problema è spiegato dai programmatori di Facebook, e descrive la relazione tra chat e notifiche. Quando abbiamo una chat aperta ed arriva un

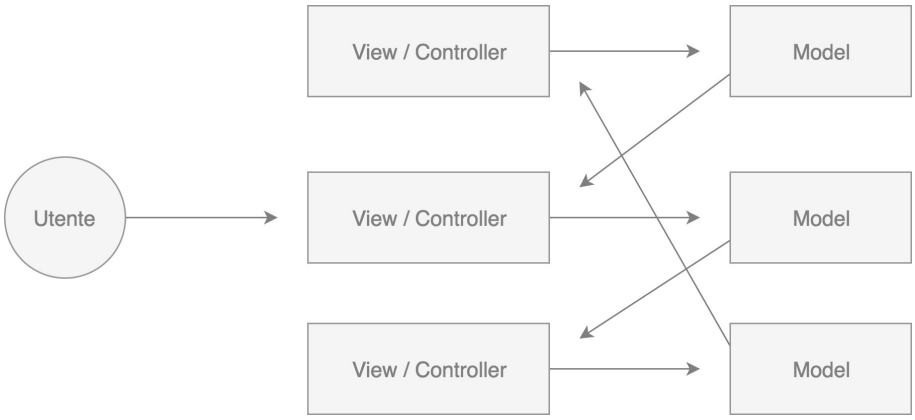


**Figura 3.6:** Rappresentazione del flusso di dati in un'architettura MVC.

ì

messaggio, questo viene riportato all'interno della chat, e sotto forma di nuova notifica. La chat e le notifiche sono due View separate che però prendono i dati da uno stesso Model. Quando si effettua un click sulla chat la notifica viene automaticamente letta, come anche vice versa se si clicca sulla notifica comparirà il simbolo di "messaggio visualizzato" nella chat. All'interno dei rispettivi Controller questi comportamenti devono essere sincronizzati ed applicati in maniera corretta tenendo presente lo stato di ciascun altro elemento e tutte le varie eccezioni che possono incorrere. Tutto ciò rende difficile scalare l'applicazione aggiungendo nuove features collegate [21].

# 4. Il flusso unidirezionale con Flux e Redux

E' stato spiegato come MVC gestisce la problematica del flusso dei dati attraverso la bidirezionalità, e tutti i problemi che quest'ultima causa nell'implementazione di una applicazione web complessa. Solo recentemente è stato utilizzato un nuovo tipo di approccio che cerca di ovviare a tutto ciò, ed è quello del "flusso unidirezionale". Due sono le architetture che verranno analizzate: Flux e Redux. Entrambe riescono a mantenere un alto livello di scalabilità a prescindere dalla complessità dell'applicazione e risolvono tutti i problemi legati al flusso bidirezionale ed MVC.

## 4.1 Architettura Flux

Flux è un'architettura recente creata da Facebook che struttura il front-end di una applicazione in modo che il flusso dei dati dall'innesco di un evento fino alle sue ripercussioni nell'interfaccia segua un'unica direzione. Questo pattern, essendo molto astratto, non ha delle vere e proprie dipendenze ed è possibile applicarlo a qualsiasi tipo di applicazione con qualsivoglia linguaggio. È tuttavia nato per strutturare applicazioni React e quindi perfezionato per tale libreria.

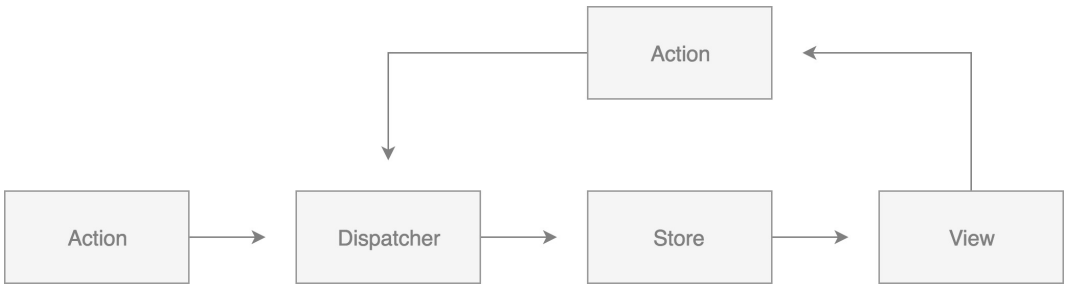


**Figura 4.1:** Rappresentazione del flusso di dati con Flux.

L'architettura descritta da Flux nella Figura 4.1 evidenzia dei componenti quasi completamente nuovi rispetto a quelli visti fino ad ora nei vari dialetti di MVC. Ci sono quattro elementi fondamentali: le *Action*, il *Dispatcher*, gli *Store* e le *View*. Le Action sono degli "eventi" che descrivono i vari comportamenti dell'applicazione e che le View trasmettono ai gestori dello stato chiamati Store, attraverso il Dispatcher, il quale è unico ed ha solamente il compito di indirizzatore. Gli Store a seconda del tipo di Action ricevuta modificano di conseguenza la parte di stato contenuto al loro interno emettendo infine una notifica alle View. Il percorso che i dati dell'applicazione seguono

in questo caso è ciclico, e ogni componente interagisce solamente con il successivo. La logica dell'applicazione è divisa in due parti: ciò che riguarda la gestione degli eventi e le modifiche all'interfaccia è compreso nelle View, in questo caso più propriamente classificabili come Controller-View; la logica che riguarda invece lo stato dell'applicazione è compresa negli Store, i quali modificano lo stato a seconda della Action ricevuta dal Dispatcher.

### 4.1.1 I componenti

Sono stati elencati i componenti fondamentali di questa architettura e come essi comunicano fra loro. Per capire appieno Flux e come questo implementi l'unidirezionalità dei dati tuttavia, è bene analizzare in dettaglio il lavoro di ciascuno di questi componenti.

#### Action

Il problema fondamentale di dividere la logica dell'interfaccia da quella della gestione dello stato, sta nel far comunicare le due in modo semplice e veloce. Le View riescono a notificare gli Store di un evento innescato dall'utente grazie alle Action, e questi dopo aver modificato la loro parte di stato, comunicano alle View l'avvenuto cambiamento. Le Action sono quindi la moneta di scambio dell'architettura e consistono generalmente in un oggetto composto da due elementi:

- **Nome** Rappresenta lo scopo dell'azione da effettuare, ed è una stringa come ad esempio `AddArticle` o `RemoveUser`.
- **Payload** È l'insieme di dati necessari a portare a termine l'azione. Se si stesse parlando di una azione `AddArticle` ad esempio, il payload sarebbe composto dal titolo e dal contenuto dell'articolo da aggiungere.

Tutti gli eventi esistenti che modificano lo stato dell'applicazione devono essere descritti sotto forma di Action. In Flux qualsiasi cosa non sia un'azione non può verificarsi.

#### Dispatcher

Il Dispatcher nell'architettura Flux è il componente che ha il compito di distribuire le Action agli Store. È sostanzialmente un registro di callback, e non ha una vera e propria logica sua fungendo quindi solo da trasmettitore. La sua particolarità è che qualsiasi azione riceva, viene trasmessa poi ad ogni callback registrata lasciando a quest'ultima il compito di analizzare l'azione e decidere se effettuare cambiamenti o meno (Codice 4.1).

Normalmente come Dispatcher viene utilizzato quello già implementato contenuto nella libreria Javascript *Flux*[1]. La particolarità di questo è che permette di gestire azioni con dipendenze, ossia azioni che dipendono dalla corretta esecuzione di altre azioni precedenti. Man mano che una applicazione cresce è molto probabile trovare dipendenze legate a Store differenti. Quando ad esempio lo "Store X" ha bisogno che lo "Store Y" si aggiorni prima di effettuare i propri cambiamenti, c'è bisogno che il Dispatcher sia in grado di sincronizzare i due eventi. Ciò avviene tramite il metodo

[1] https://github.com/facebook/flux/blob/master/src/Dispatcher.js

```
class Dispatcher {
  constructor() {
    this._callbacks = [];
  }

  register(callback) {
    this._callbacks.push(callback);
  }

  dispatch(action) {
    this._callbacks.forEach((callback) => {
      callback(action);
    });
  }
}
```

**Codice 4.1:** Esempio di un semplice Dispatcher.

`Dispatcher.waitFor()`, che blocca l'esecuzione della callback dove viene eseguito attendendo che tutte le altre callback passate a quest'ultimo come parametro vengano completate.

### Store

Lo Store dell'architettura Flux può essere paragonato al Model del paradigma MVC. Si occupa di mantenere parte dello stato dell'applicazione, di ottenere i dati, di emettere eventi e di fornire la callback per la gestione delle Action al Dispatcher (Figura 4.2). Questa callback non è altro che una funzione che prende in input l'Action emessa dal Dispatcher e con all'interno un costrutto `Switch` che determina cosa fare. Se l'Action è stata eseguita dallo Store, questo emetterà un evento e tutte le View in ascolto di questo particolare evento si aggiorneranno di conseguenza.

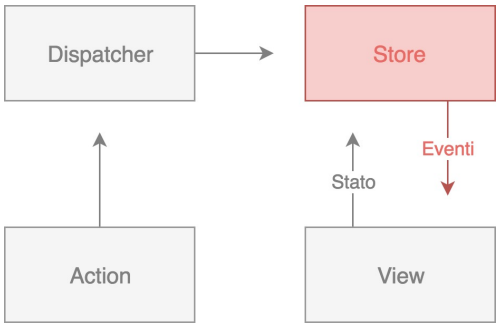


**Figura 4.2:** Compiti dello Store nell'architettura Flux.

Un punto fondamentale riguardo gli Stores è che essi non ricevono modifiche esterne. Tutta la logica relativa al cambiamento dello stato dell'applicazione è contenuta al loro

interno e questo semplifica enormemente il lavoro di debugging del codice e rafforza la consistenza dello stato.

#### View

La View è formata da componenti React strutturati presumibilmente con il pattern spiegato nella Sezione 2.3, dei componenti contenitori e di presentazione. Nel caso di Flux, i componenti contenitori sono posizionati al livello più alto ed interagiscono direttamente con gli Store, rimanendo in ascolto dei loro eventi e ottenendo da essi i dati che verranno poi passati ai componenti di presentazione figli. Utilizzando questa strategia si assicura un corretto flusso dei dati, in cui l'unico punto di ingresso è il componente contenitore all'inizio della catena.

### 4.1.2 Analisi dell'applicazione

Per avere un esempio pratico di questa architettura verrà implementata di nuovo l'applicazione presentata nella Sezione 3.3.1, questa volta attraverso React e Flux. In questo caso oltre che la normale sintassi di ES6 c'è anche bisogno di compilare il codice JSX all'interno di questa in modo da ottenere i componenti React desiderati, per questo motivo verrà utilizzato anche il plugin di Babel *React Preset*[2]. Vengono utilizzati due tipi di configurazioni per Webpack: `webpack.dev.js` in fase di development, più veloce ma che genera un codice non ottimizzato; `webpack.prod.js` in fase di produzione, più lenta che genera però un codice più leggero e veloce.

Il codice dell'applicazione è diviso in due parti principali: i componenti React nella cartella `flux/src/components` e gli elementi dell'architettura Flux nella cartella `flux/src/data`. Di seguito verranno spiegate in dettaglio queste due parti.

#### I componenti React

Il componente più ad alto livello in cui viene inizializzata l'applicazione e dove React viene aggiunto materialmente al DOM è situato in `flux/src/app.js`. Salta subito all'occhio che questo è un componente "di passaggio" ed utilizzato solo per inizializzare React. Il vero e proprio componente più ad alto livello, e che possiamo considerare come componente "container" è `flux/src/components/Bookshelf.js`.

Il Codice 4.2 rappresenta l'interazione tra il componente container Bookshelf e lo Store principale dell'applicazione. Questa interazione si divide in due parti:

1. Nel costruttore viene inizializzato lo stato del componente ottenendolo dallo Store tramite il metodo `BookshelfStore.getBooks()`.

2. Nel metodo `componentWillMount()`, che React esegue automaticamente durante l'inizializzazione del componente, viene iniziato l'ascolto dell'evento di aggiornamento dello Store attraverso il metodo `BookshelfStore.addChangeListener()`. Questo non fa altro che sottoscrivere una funzione della View, nel caso in esame il metodo `_onChange()`, all'evento `change` dello Store. La funzione sottoscritta ha il compito di sincronizzare lo stato del componente container con il nuovo stato dello Store.

[2]https://babeljs.io/docs/plugins/preset-react

```
// Bookshelf.js
class Bookshelf extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      books: BookshelfStore.getBooks()
    };
  }

  componentWillMount() {
    BookshelfStore.addChangeListener(this._onChange.bind(this));
  }

  _onChange() {
    // ...
  }

  // ...
}
```

**Codice 4.2:** Esempio di interazione tra componente container e Store.

Il componente container ha poi il compito di comporre l'interfaccia utilizzando gli altri componenti di presentazione a disposizione, e passare loro i dati e le funzioni per il corretto funzionamento dell'applicazione.

Nel Codice 4.3 viene utilizzato il componente `<AddBook />` definito all'interno del file `AddBook.js`, al quale viene passata la funzione `_handleAddBook()` che gestirà l'evento di aggiunta di un nuovo libro emanando l'Action relativa al Dispatcher. Tutti i componenti di presentazione utilizzati per comporre l'interfaccia sono dipendenti dal componente container `<Bookshelf />` e non comunicano direttamente con l'architettura Flux. Questo ha il compito di creare le funzioni per gestire i vari eventi dell'applicazione e passarle ai suoi componenti di presentazione. Un componente di presentazione è solitamente composto solamente da una funzione che ritorna codice JSX in base alle proprietà ricevute dal componente superiore.

Il componente rappresentato nel Codice 4.4 ad esempio, che rappresenta un semplice bottone, è riutilizzabile all'interno dell'applicazione in quanto a seconda del contesto possiamo cambiare il suo contenuto e comportamento in base delle proprietà a lui passate.

E' possibile creare un componente contenitore all'interno di un altro nel caso sia necessario gestire elementi dell'interfaccia piuttosto complessi. Questo è il caso del componente `<AddBook />`, che rappresenta il form di aggiunta di un nuovo libro, composto dai vari campi di testo ed il bottone di invio e mantiene al suo interno lo stato di questi vari elementi. È bene prestare attenzione nell'aggiungere componenti contenitori nidificati, evitando soprattutto di utilizzarli per interagire con l'architettura, compito che dovrebbe essere lasciato sempre al componente di più alto livello.

```
// Bookshelf.js
class Bookshelf extends React.Component {
  // ...

  render() {
    return (
      // ...
      <div className="u-mt1">
        <AddForm onAddBook={this._handleAddBook} />
      </div>
      // ...
    )
  }

  _handleAddBook(title, author) {
    BookshelfActions.addBook(title, author);
  }
}
```

**Codice 4.3:** Esempio di composizione tra componente container e di presentazione.

```
// Button.js
const Button = (props) => (
  <button
    className="c-button"
    onClick={props.onClick}>
    {props.children}
  </button>
);
```

**Codice 4.4:** Esempio di un semplice componente di presentazione.

### La gestione dei dati

L'architettura Flux è implementata attraverso i componenti presenti nella cartella `flux/src/data`. Il primo file da analizzare è `BookshelfActions.js`, una libreria che fornisce le Actions eseguibili all'interno dell'applicazione sotto forma di funzione, e le fa emettere direttamente dal Dispatcher. Questo tipo di utility è comunemente chiamata in ambito Flux *ActionCreator* ed è un pattern molto utilizzato in quanto permette di organizzare e gestire in maniera dinamica le Action, che normalmente sarebbero dei semplici oggetti.

Nell'esempio del Codice 4.5 viene riportata la funzione `Actions.addBook()` che prende in input il titolo e l'autore di un libro ed automatizza l'emissione dell'azione `ADD_BOOK` al Dispatcher tramite il metodo `BookshelfDispatcher.dispatch()`. Le funzioni implementate dall'ActionCreator vengono generalmente passate dal componente container ai componenti di presentazione per interagire in maniera corretta con lo Store.

```
// BookshelfActions.js
const Actions = {
  addBook: function(title, author) {
    BookshelfDispatcher.dispatch({
      type: 'ADD_BOOK',
      payload: {
        title: title,
        author: author
      }
    });
  },

  // ...
};
```

**Codice 4.5:** Esempio di ActionCreator dell'applicazione.

Le Action devono essere emesse dal Dispatcher. Quest'ultimo, nell'applicazione, viene implementato nel file `BookshelfDispatcher.js` utilizzando quello messo a disposizione automaticamente dalla libreria *Flux*. Questo Dispatcher, implementato ed attualmente utilizzato anche da Facebook, è una versione molto più potente del codice esempio esposto nella Sezione 4.1.1.

```
// BookshelfDispatcher.js
import { Dispatcher } from 'flux';

export default new Dispatcher();
```

**Codice 4.6:** Dispatcher dell'applicazione.

Nel Codice 4.6, che rappresenta il Dispatcher dell'applicazione in analisi, viene esportata direttamente l'istanza che ci fornisce un singleton. Questo perché il Dispatcher deve essere unico all'interno dell'applicazione e condiviso tra tutti i componenti che hanno bisogno di effettuare azioni.

L'ultimo componente che viene preso in esame è lo Store, implementato nel file `BookshelfStore.js`. Questo ha il compito di gestire lo stato dell'applicazione mantenendo una lista di libri in `this.state`, pre-popolata attraverso il file JSON `db.json`, e anche quello di fornire al Dispatcher la funzione incaricata di gestire le Action emesse.

Analizzando il Codice 4.7 è possibile notare che lo Store viene creato estendendo la classe `EventEmitter`, questo perché ogni Store ha la necessità di mandare eventi all'interno dell'applicazione ogni qualvolta lo stato viene modificato. Troviamo poi all'interno del costruttore della classe il metodo `BookshelfDispatcher.register()` che prende come parametro la funzione dello Store da registrare nel Dispatcher. La funzione che verrà aggiunta è `_registerActions()` la quale prende come parametro un'Action e, come da prassi, la getta in pasto ad un costrutto `Switch` che la eseguirà sullo Store corrente.

```
// BookshelfStore.js
class BookshelfStore extends EventEmitter {
  constructor() {
    // ...

    BookshelfDispatcher.register(
      this._registerActions.bind(this)
    );
  }

  _registerActions(action) {
    switch(action.type) {
      case 'ADD_BOOK':
        return this._addBook(
          action.payload.title,
          action.payload.author
        );

      // ...
    }
  }

  // ...
}
```

**Codice 4.7:** Registrazione dello Store nel Dispatcher

### 4.1.3 Flux e il flusso dei dati

Come possiamo notare dalla Sezione 4.1.2, l'architettura di Flux rispetto a quella MVC è molto più strutturata, ed i dati e gli eventi devono sottostare a delle regole e dei percorsi più rigidi, cosa che permette di avere un maggiore controllo sui comportamenti dell'applicazione e sul cambiamento dello stato. In Flux gli effetti collaterali derivati dall'aggiornamento dello stato sono molto facili da controllare, e grazie alla rigidità dell'architettura anche in numero molto inferiore. Quando uno Store riceve un'Action, ed il suo stato viene modificato, emette un evento grazie al quale le View dell'applicazione si modificheranno di conseguenza. Questo è l'unico, ed inevitabile, caso in cui la modifica dello stato causa un'ulteriore esecuzione di codice. Tuttavia il vantaggio di Flux non sta solo in questo, ma anche nel gestire in modo chiaro le dipendenze tra due Store differenti. In un'architettura MV* non esiste un vero e proprio approccio che permette di eseguire un cambiamento solo se ne è stato fatto un'altro in precedenza, se non in maniera imperativa e notificando manualmente il componente dipendente. In Flux questo problema è risolto semplicemente dichiarando la dipendenza all'interno dello Store tramite la funzione `Dispatcher.waitFor()` descritta nella Sezione 4.1.1. Un altro elemento a favore di Flux riguarda sempre lo Store e come questo integri in se stesso tutti i possibili cambiamenti che lo stato possa fare. Ciò fa sì che nel caso in cui si debba analizzare un certo cambiamento, la sua posizione all'interno del codice sia immediata. Questo impedisce la creazione di cambi di stato nascosti o non previsti,

classici durante lo sviluppo di applicazioni MV* [22].

Tutto ciò espresso fino ad ora è indubbiamente un punto a favore a livello di architettura, di gestione dei dati e degli eventi, tuttavia a livello implementativo possono sorgere dei difetti non trascurabili. In Flux lo Store notifica le View del cambiamento del proprio stato, senza specificare cosa è stato cambiato come ad esempio potrebbe capitare in altre architetture. Questo comporta un lavoro più oneroso da parte della View, la quale potrebbe dover aggiornare, anche inutilmente, più componenti del necessario. Proprio per questo motivo React è considerato un'ottima libreria per Flux in quanto riesce ad aggiornare solamente il DOM che viene effettivamente modificato. Un'altra problematica sempre riguardo gli Store è quando il loro numero e la loro complessità diventa elevata, e di conseguenza diventa molto difficile gestire le dipendenze delle Action nonostante la presenza del Dispatcher. Queste dipendenze possono diventare sempre più numerose, complesse e difficili da gestire all'aumentare del numero di features dell'applicazione. Un altro punto a sfavore di Flux riguarda le Action. Ogni cosa che accade all'interno dell'applicazione deve essere descritta con un'Action, ed è facile immaginare che una singola feature può dunque crearne un numero non irrilevante. Tenere traccia di ognuna di esse diventa quindi un compito non proprio semplice nonostante l'aiuto di una libreria ActionCreator.

## 4.2 Architettura Redux

Redux è una libreria scritta da Dan Abramov e Andrew Clark per la gestione dello stato in applicazioni web scritte con Javascript. Evolve l'idea dell'architettura Flux spiegata nella Sezione 4.1 semplificandola attraverso pattern e tecniche della programmazione funzionale. Il concetto base di Redux è che lo stato completo di una applicazione dovrebbe essere descritto attraverso un solo e semplice oggetto Javascript.

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

**Codice 4.8:** Stato di una applicazione scritta con Redux preso dal sito ufficiale.

Nel Codice 4.8 preso dal sito ufficiale di Redux, ad esempio, viene rappresentato lo stato completo di una applicazione per la gestione delle cose da fare (la classica "To-do application"), con anche dei filtri per visualizzare i compiti aggiunti. La struttura di questo è molto semplice ma allo stesso tempo ci da tutte le informazioni necessarie per capire l'applicazione. Il problema di avere lo stato in un unico oggetto risiede nella difficoltà che si ha nel gestirlo. Basti immaginare quanto possa essere intricata la struttura dell'oggetto che descrive un'interfaccia complessa come quella di Facebook ad esempio, e quanti livelli nidificati di stato si dovrebbero gestire. Proprio per questo l'approccio

utilizzato da Flux consiste nel dividerlo, distribuendolo su più Store e diminuendo drasticamente la complessità del lavoro. Redux invece risolve questo problema utilizzando la tecnica della composizione di funzioni, un classico della programmazione funzionale che in questo caso permette di generare lo stato completo dell'applicazione attraverso delle funzioni chiamate *Reducer* che ne gestiscono e ritornano piccoli pezzi.

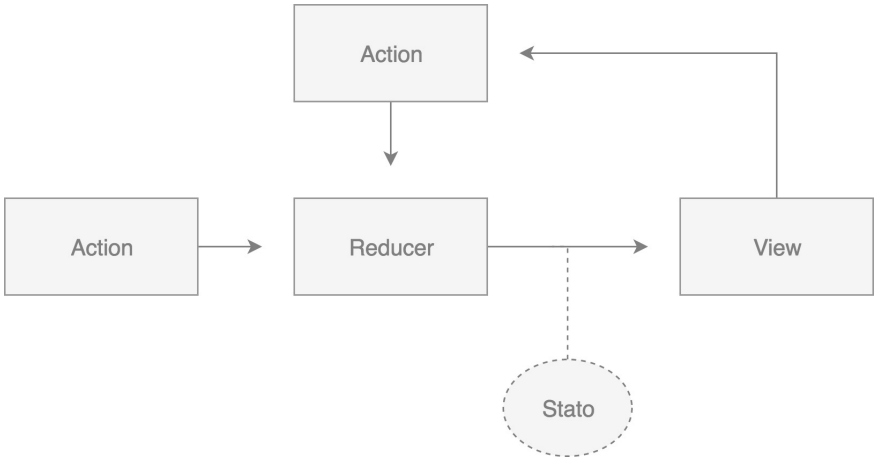


**Figura 4.3:** Architettura Redux

Come è possibile notare nella Figura 4.3, Redux è molto simile a Flux, ereditando cose come la completa gestione logica dello stato all'interno di un singolo componente, che in Flux era lo Store ma che in Redux è il Reducer. Un altro elemento in comune riguarda l'uso delle *Action*, le quali essendo input del Reducer permettono di modificare lo stato all'interno dello Store. Il Reducer quindi non è altro che una funzione che prende in input lo stato attuale ed un'Action, restituendo in output il nuovo stato da loro derivato. Al contrario di Flux tuttavia non esiste un Dispatcher in quanto i Reducers sono *funzioni pure*[3], facili da comporre e che non hanno quindi bisogno di un componente esterno che li gestisca. Questa può essere vista sia come una differenza di implementazione che come un miglioramento, in quanto la gestione dello stato di Flux è sempre descritta come una funzione che deriva il nuovo stato dal precedente e dall'azione eseguita su di esso `(state, action) => state`, descrivendo precisamente quello che è il Reducer in Redux [24].

### 4.2.1 I tre principi base di Redux

Per scrivere una applicazione che segue in modo preciso questa architettura è necessario seguire tre principi basilari che ne accertano il giusto utilizzo [24]:

1. **Una sola fonte di verità** Il primo principio indica che lo stato di una applicazione deve essere unico e deve essere memorizzato all'interno di un singolo Store. Questo facilita operazioni come la serializzazione dello stato attuale in modo da poterlo salvare, ad esempio in un database, e recuperarlo in seguito per ripristinare l'applicazione ad un momento precedente. Oppure implementare

[3]Una funzione pura è una funzione che, dati gli stessi input, restituisce lo stesso output e non ha nessun effetto collaterale, come ad esempio richieste al database, o mutazioni a variabili fuori dal suo scope [23].

funzionalità che normalmente sarebbero piuttosto complicate, come l'azione di annullare un'operazione effettuata. Avere a disposizione uno stato singolo per applicazione semplifica anche le operazioni di debugging e di ispezione.

2. **Lo stato in sola lettura** Il secondo principio di Redux, e comune anche a Flux, dice che lo stato dell'applicazione non può essere modificato se non per mezzo dell'emissione di un'Action al Reducer. Questo assicura che nessun componente possa inavvertitamente apportare modifiche dirette allo stato e generare quindi comportamenti inaspettati ed imprevedibili.

3. **Lo stato cambia solo attraverso funzioni pure** Il terzo ed ultimo principio, ma non a livello di importanza, riguarda l'obbligo di utilizzare solamente funzioni pure all'interno dei Reducer. Questo permette di comporre più Reducer specifici nell'unico Reducer che si occupa della gestione dello stato, e siccome questi sono solamente funzioni senza effetti collaterali è possibile controllare il loro ordine di esecuzione o creare Reducer generici per parti dello stato comuni.

### 4.2.2 I componenti

Due sono i cambiamenti fondamentali dei componenti di Redux rispetto a quelli di Flux descritti nella Sezione 4.1.1: il primo consiste nell'abbandono del Dispatcher; il secondo riguarda invece lo Store che questa volta è unico all'interno dell'applicazione e sfrutta la funzione Reducer per la gestione logica dello stato.

#### Action

Anche in Redux come nell'architettura Flux le Action sono semplici oggetti composti da un "nome" e da un "payload", e anche qui si utilizza una libreria ActionCreator per una loro gestione più facile e comprensibile. La differenza principale tuttavia è che in Flux l'ActionCreator eseguiva direttamente `Dispatcher.dispatch()` ed emettendo quindi direttamente l'Action, in Redux invece l'ActionCreator si occupa solamente di restituire l'oggetto Action desiderato. Astrarre l'Action in questo modo è utile perché la funzione `dispatch()`, che si occupa di emettere l'azione al Reducer, è possibile ottenerla sia dallo Store con `Store.dispatch()` sia dalla funzione `connect()` presente nella libreria *react-redux*[4]. Questa automatizza la creazione del componente container in React e gestisce tutti i suoi aggiornamenti in maniera molto più efficiente che facendolo a mano sottoscrivendolo allo Store.

#### Reducer

Il Reducer è il cuore dell'architettura Redux, nonché il componente che racchiude tutto il significato di quest'ultima ed il perché della sua potenza. Questo, come è stato precedentemente detto, è una funzione pura del tipo `(state, action) => state`, ossia che prende come parametro lo stato attuale dell'applicazione ed una Action, restituendo il nuovo stato. Il Reducer è comparabile in Flux alla funzione che dallo Store registriamo nel Dispatcher per gestire un'Action. Siccome lo stato, in questa architettura, è unico (deve essere rappresentato in un singolo oggetto Javascript) come è unico anche lo Store, e quindi anche la funzione Reducer, è necessario trovare un modo per organizzare quest'ultima in maniera comprensibile. Se si dovesse scrivere il Reducer per esteso si

[4]https://github.com/reactjs/react-redux

otterrebbe una funzione lunghissima e difficilmente gestibile anche per una applicazione non troppo grande, tuttavia utilizzando il pattern della composizione presente nella programmazione funzionale è possibile ovviare a questo problema in modo molto semplice.

```
const todosReducer = (state = [], action) => {
  // ...
};

const visibilityFilterReducer = (state = 'SHOW_ALL', action) => {
  // ...
};

// Questo è il Reducer composto
const todoAppReducer = combineReducers({
  todos: todosReducer,
  visibilityFilter: visibilityFilterReducer
});
```

**Codice 4.9:** Esempio di composizione fra Reducer.

Come rappresentato nel Codice 4.9 è possibile creare un Reducer isolato per ogni elemento dello stato da gestire, e comporli insieme per formare il Reducer principale attraverso la funzione `combineReducers()` messa a disposizione dalla libreria Redux. Questa funzione non fa altro che chiamare i Reducer creati con il pezzo di stato a loro assegnato, combinando tutti i risultati ottenuti in un singolo oggetto [24].

### Store

Lo Store nell'architettura Redux rappresenta il componente che integra sia il Reducer che le Action. Lo Store è incaricato di tenere traccia dello stato dell'applicazione, fornire lo informazioni sullo stato ai componenti della View, aggiornare lo stato attraverso la funzione `dispatch()` e sottoscrivere i componenti interessati all'aggiornamento dello stato in modo che siano notificati. Una volta creato il Reducer principale dell'applicazione, la costruzione dello Store è praticamente immediata utilizzando la funzione `createStore()` che prende come primo parametro il Reducer, e come secondo parametro opzionale uno stato iniziale.

### View

La View di Redux, usando React, è quasi del tutto simile ad una View dell'architettura Flux, con qualche differenza riguardo ai componenti container. Per interagire con l'architettura Redux dal componente container è possibile utilizzare due metodi diversi, uno manuale ed un altro automatizzato:

- **Metodo manuale** In questo caso la comunicazione con l'architettura Redux avviene attraverso la sottoscrizione del componente container allo Store attraverso la funzione `Store.subscribe()` in modo da poterne ottenere lo stato, e la gestione sempre manuale delle proprietà di tutti i componenti di presentazione collegati. Questo metodo è quello adottato anche nell'architettura Flux.

- **Metodo automatizzato** La libreria Redux mette a disposizione la funzione `connect()` che permette di creare in maniera automatica il componente contenitore. Questa riceve in input due funzioni chiamate: `mapStateToProps()` e `mapDispatchToProps()`. La prima permette di tradurre lo stato ottenuto dallo Store nelle proprietà da passare poi ai vari componenti di presentazione collegati. La seconda permette invece di definire i metodi che eseguono la funzione `dispatch()` per l'emissione delle Action.

Utilizzare la funzione `connect()` per la generazione del componente contenitore è estremamente preferibile in quanto apporta delle ottimizzazioni automatiche (come l'implementazione della funzione interna `shouldComponentUpdate()` [5] dei componenti React) per evitare di far effettuare aggiornamenti inutili ai componenti, ad ogni modifica dello stato nello Store.

Tutti i componenti container potrebbero aver bisogno di accedere allo Store, in qualsiasi posizione dell'applicazione essi siano, specialmente se la comunicazione con l'architettura avviene in maniera manuale. Un'opzione è quella di passarlo come proprietà ad ogni livello gerarchico della View, tuttavia potrebbe risultare un'azione ripetitiva e piuttosto tediosa specialmente se abbiamo dei componenti container nidificati. Un'altra utile feature di Redux riguarda il componente `<Provider />`. Utilizzandolo solo una volta ed al livello più alto dell'applicazione rende lo Store disponibile ad ogni componente senza doverlo passare esplicitamente [24].

### 4.2.3 Analisi dell'applicazione

Per l'analisi concreta di un'applicazione con Redux, verrà ripreso parte del codice dell'architettura Flux implementato nella Sezione 4.1.2. Le dipendenze sono sempre le stesse ad eccezione delle librerie dell'architettura, ed in questo caso dell'aggiunta di *react-redux*. Come le stesse sono anche le due configurazioni di Webpack.

Il codice è sempre diviso in due parti principali: i componenti React nella cartella `redux/src/components` e gli elementi dell'architettura Redux in questo caso, nella cartella `redux/src/data`. Di seguito verranno spiegate in dettaglio queste due parti e le principali differenze con l'architettura Flux.

#### I componenti React

Come è già stato detto in precedenza, l'unico componente React che cambia nell'architettura Redux è il componente container, il quale non è più gestito manualmente ma automatizzato attraverso la funzione `connect()`.

Nel Codice 4.10 è rappresentato il componente container principale dell'applicazione che viene creato utilizzando il metodo automatizzato descritto nella Sezione 4.2.2. Le funzioni `mapStateToProps()` e `mapDispatchToProps()` vengono connesse insieme al componente `<Bookshelf />` fornendogli le proprietà con all'interno gli elementi utili dello stato e le Action per interagire con lo Store da passare ai suoi componenti di presentazione.

[5] La funzione *shouldComponentUpdate* permette di modificare il sistema di aggiornamento del componente React dove essa è definita. Di default un componente si aggiorna ogni volta che le sue proprietà vengono modificate, e quindi la funzione ritorna sempre *true*. Se non vogliamo aggiornare un componente, o vogliamo che il suo aggiornamento avvenga solo in casi particolari è necessario modificare *shouldComponentUpdate* affinché ritorni *false* quando necessario.

```
// Bookshelf.js
const mapStateToProps = state => ({
  // ...
});

const mapDispatchToProps = dispatch => ({
  // ...
});

class Bookshelf extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    // ...
  }
}

const ConnectedBookshelf = connect(
  mapStateToProps,
  mapDispatchToProps
)(Bookshelf);
```

**Codice 4.10:** Componente container con Redux.

Il contenuto di queste funzioni utilizzate all'interno di `connect()` è presente nel Codice 4.11. La funzione `mapStateToProps()` prende in input lo stato dallo Store (che in questa applicazione è solamente un array di libri) e lo assegna ad una proprietà. Questo procedimento è l'equivalente di assegnare lo stato dello Store in maniera diretta: `<Bookshelf books={state.books} />`. Allo stesso modo, la funzione `mapDispatchToProps()` prende come parametro la funzione per l'emissione delle Action `dispatch()` e la assegna, utilizzando le funzioni create all'interno dell'ActionCreator, alle proprietà che serviranno per gestire gli eventi nei vari componenti di presentazione. La funzione `connect()` prende in input queste due funzioni e restituisce in output una nuova funzione *Higher-order component*[6], che passa ad un componente React tutte le proprietà precedentemente definite. In questo caso il componente React passato come parametro, come è possibile notare nel Codice 4.10, corrisponde solamente alla funzione `render()` del componente `<Bookshelf />` implementato nell'architettura Flux. Questo perché tutti i metodi creati all'interno in maniera manuale sono stati invece generati automaticamente da `connect()` e passati come proprietà.

[6]Un *Higher-order component*, o più semplicemente "HOC", è una funzione che prende in input e restituisce in output un componente React. Il suo nome deriva dalle "higher order function" che in matematica rappresentano funzioni che prendono come parametro una o più funzioni, e ritornano una o più nuove funzioni derivate da queste.

```
// Bookshelf.js
const mapStateToProps = state => ({
  books: state.books
});

const mapDispatchToProps = dispatch => ({
  handleAddBook: (title, author) => {
    dispatch(addBook(title, author));
  },

  handleRemoveBook: index => {
    dispatch(removeBook(index));
  }
});

// ...
```

**Codice 4.11:** Funzioni di gestione del componente container con Redux.

### La gestione dei dati

La vera e propria architettura Redux è implementata nei file contenuti in `redux/src/data` dove si trovano Reducer e Action, e nel file `redux/src/app.js` dove viene creato lo Store. Tecnicamente, in Redux abbiamo un solo componente che esiste concretamente, ed è lo Store. I Reducer e le Action (più precisamente l'ActionCreator in questa applicazione) servono a far funzionare nel modo corretto quest'ultimo. I primi gestiscono lo stato, le seconde permettono ai vari componenti di interagire con lo Store. Sotto questo punto di vista Redux semplifica parecchio l'architettura di Flux, concentrando tutto il lavoro in un unico componente invece di avere un numero indefinito di Store coordinati tramite il Dispatcher.

```
// BookshelfActions.js
const addBook = (title, author) => ({
  type: 'ADD_BOOK',
  payload: {
    title: title,
    author: author
  }
});

// ...
```

**Codice 4.12:** ActionCreator dell'applicazione Redux.

Nel Codice 4.12 è rappresentato l'ActionCreator che ritorna in maniera dinamica le varie Action da eseguire per mezzo della funzione `dispatch()`. Al contrario dell'architettura Flux ogni funzione definita nell'ActionCreator ritorna un normale oggetto invece che emettere direttamente l'azione allo Store. Redux non possiede un vero e

proprio standard per la struttura delle Action e viene quindi solitamente utilizzata la struttura definita da Flux, che prevede il nome dell'azione nel campo "type" e i dati utili per la sua esecuzione nel campo "payload".

```
// BookshelfReducer.js
const initialState = {
  books: []
};

function bookshelfReducer(state = initialState, action) {
  switch(action.type) {
    case 'ADD_BOOK':
      return Object.assign({}, state, {
        books: [].concat(
          action.payload,
          state.books
        )
      });

    // ...
  }
};
```

**Codice 4.13:** Reducer dell'applicazione Redux.

Il Reducer descritto nel Codice 4.13 descrive il cambiamento dello stato nello Store alla ricezione di determinate Action. La struttura della funzione del Reducer è molto simile a quella della funzione registrata all'interno del Dispatcher nell'architettura Flux, tuttavia il Reducer è una funzione pura. Lo stato non viene quindi mai modificato direttamente ma viene ritornato un nuovo stato a seconda dell'Action e dello stato precedente. Ad ogni caso dello `Switch` lo stato non viene mai modificato, bensì viene duplicato tramite `Object.assign()` modificando solamente le parti dovute. Utilizzare il costrutto `Switch` non è obbligatorio, ma è diventato in ogni caso lo standard di Redux riguardo il componente Reducer.

Nel Codice 4.14 è raffigurato il contenuto del file `app.js`, nel quale viene creato l'unico Store dell'applicazione. Questo viene creato attraverso la funzione `createStore()` la quale prende in input il Reducer e lo stato iniziale (il quale è rappresentato dalla lista di libri nel file JSON `db.json`) e restituisce in output uno Store che tiene traccia dello stato completo dell'applicazione. Lo Store non è posizionato in un file a parte, e non è un singleton questa volta, bensì viene passato all'interno dell'applicazione attraverso il componente `<Provider />` che permette il suo accesso a qualsiasi livello della gerarchia della View. Avere uno Store mobile all'interno dell'applicazione e non un singleton globale semplifica il lavoro sia in fase di testing, sia nel caso in cui si voglia utilizzare la tecnica del Server-side rendering, in quanto lato server è comunque necessario creare uno Store nuovo ad ogni richiesta ricevuta.

```
// app.js
// ...

const initialState = {
  books: defaultDB
};

let store = createStore(
  bookshelfReducer,
  initialState
);

ReactDOM.render(
  <Provider store={store}>
    <Bookshelf />
  </Provider>,
  document.getElementById('app')
);
```

**Codice 4.14:** Store dell'applicazione Redux.

### 4.2.4 Redux e il flusso dei dati

Il flusso dei dati nell'architettura Redux può essere riassunto semplicemente in un'Action emanata dalla View che viene data in pasto al Reducer nello Store, per generare il nuovo stato dell'applicazione. Come architettura Redux è decisamente meno complesso di Flux riuscendo a semplificare il lavoro degli Store e ad eliminare il Dispatcher. I vantaggi che offre sono comunque gli stessi, infatti la gestione del flusso unidirezionale è comunque perfetta, in aggiunta l'immutabilità del Reducer e lo stato in un singolo Store sono concetti che possono essere utili per risolvere problemi comuni tipicamente complessi.

React, come descritto nella Sezione 2.3.2, è una libreria isomorfica che è possibile utilizzare anche lato server per implementare il server-side rendering ed ottenere quindi prestazioni maggiori integrando dati iniziali ottenuti dal server già dalla prima visualizzazione dell'interfaccia, senza dover effettuare richieste aggiuntive. Con Flux tuttavia sincronizzare lo stato iniziale con il server non è semplice, in quanto lo stato è suddiviso in Store multipli ed ognuno di essi è un singleton. Redux è nativamente più efficiente sotto questo aspetto in quanto lo stato è unico e rappresentabile come un oggetto Javascript, e la sincronizzazione con il server risulta quindi immediata senza la necessità di utilizzare API esterne.

Un altro aspetto positivo di Redux è come sia possibile implementare in maniera semplice features che si basano sulla composizione dei Reducer. Per riutilizzare funzionalità su più Reducer ad esempio non è necessario utilizzare l'ereditarietà, ma basta creare un *Reducer Factory*[7] che genera altri Reducer a seconda delle esigenze. Sempre grazie a questa tecnica sono nate librerie come ad esempio *redux-undo*[8], la quale permette di

[7] Un *Reducer Factory* consiste in una funzione scritta secondo il design pattern *Factory*, ossia una funzione che ritorna un Reducer strutturato a seconda dei parametri che le passiamo in input.

[8] https://github.com/omnidan/redux-undo

implementare in maniera estremamente facile una feature complessa come quella del "undo/redo", che permette di tenere traccia del passato dello stato, e recuperarlo se necessario [9].

A livello di performance Redux soffre, forse in maniera peggiore, degli stessi problemi di Flux, specialmente al livello di rendering della View in quanto ad ogni azione l'intero stato viene aggiornato. Tuttavia tra le ottimizzazioni messe a disposizione da React, le ottimizzazioni di Redux sui componenti container (tramite la funzione `connect()`), e altre librerie come ad esempio *Reselect*[9], che ricalcola gli elementi dello stato solo quando sono effettivamente modificati, questo problema è quasi trascurabile.

[9]https://github.com/reactjs/reselect

# 5. Conclusione

Tutte le varie architetture e tecniche analizzate in questo documento hanno i loro pro e contro. Nello stato attuale delle cose, è sicuramente vero che architetture come Flux e Redux, ossia a flusso unidirezionale, sono molto più gestibili e scalabili di architetture a flusso bidirezionale come quelle derivate da MVC. Gran parte di ciò è vero però solo grazie all'incredibile innovazione apportata da React, o più in particolare dal Virtual DOM, al mondo delle applicazioni web ossia la capacità di modificare solamente gli elementi del DOM le cui proprietà sono effettivamente cambiate. Infatti sia Flux, che in particolare Redux, non sono in grado di fornire ai vari componenti informazioni su ciò che all'interno dello Store è stato cambiato bensì aggiornano blocchi di stato nel caso di Flux, e lo stato intero nel caso di Redux, causando una richiesta di modifica a molti più componenti di quelli che in realtà sono stati effettivamente cambiati. Senza il Virtual DOM le prestazioni di queste architetture calerebbero in maniera non indifferente mettendo forse in ombra anche tutti i benefici strutturali da loro apportati, e magari l'architettura MVC in fin dei conti sarebbe ancora la scelta migliore. Da questo punto di vista è possibile concludere che l'efficenza sia a livello di scalabilità, sia prestazionale di una architettura dipende dalle tecnologie presenti al tempo in cui essa viene adottata. Proprio per questo dire che Flux e Redux siano le architetture perfette per l'implementazione di applicazioni web complesse è sbagliato, è invece giusto dire che sono *attualmente* perfette, con le tecnologie presenti fino a questo momento.

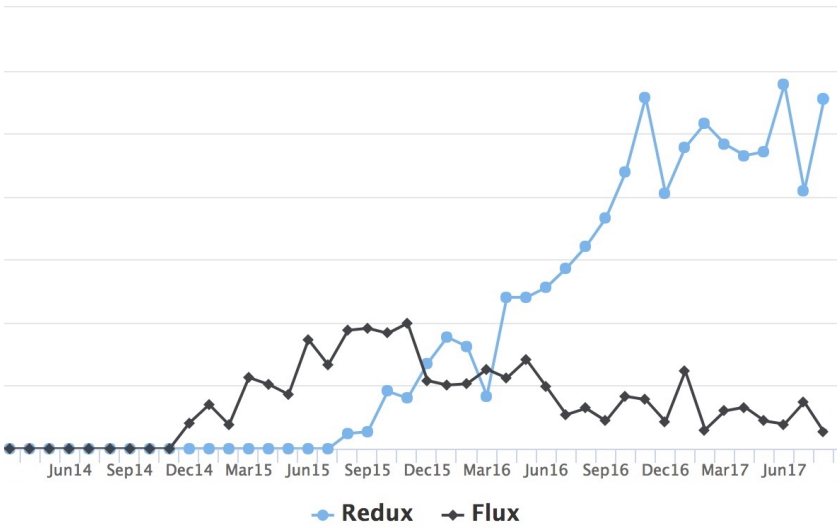


**Figura 5.1:** Grafico della popolarità di Redux e Flux (generato da hntrends.com).

Riguardo alla "lotta" tra Flux e Redux (che come è possibile notare dalla Figura 5.1 attualmente sembra essere a senso unico) il giudizio spetta al programmatore. Analizzando solo l'architettura che permettono di organizzare, il vincitore sembrerebbe essere Redux, in quanto tutto ciò che fa è semplificare Flux utilizzando metodi alternativi (in questo caso pattern presi dalla programmazione funzionale) per la gestione dei dati. Tuttavia in questo documento molti argomenti di entrambe le architetture non sono stati toccati per il semplice fatto che avrebbero portato il discorso troppo fuori tema. Uno di questi argomenti riguarda Redux, ed è la gestione di codice asincrono all'esecuzione di una Action. Come è stato più volte chiarito, il Reducer deve essere obbligatoriamente una funzione pura affinché l'architettura di Redux sia consistente, ciò pone ad esempio l'interrogativo del dove poter effettuare richieste HTTP ad una eventuale API. Questo problema viene brillantemente risolto da tecnologie come *redux-thunk*[1], *redux-promise*[2] o dal più recente *redux-saga*[3]. Queste librerie forniscono alternative differenti alla gestione di codice asincrono nell'architettura Redux. Tuttavia questo non basta ad una corretta gestione dell'applicazione. L'effettuare richieste asincrone causa un altro problema, questa volta relativo anche a Flux, ed è quello di cosa mostrare all'utente nel mentre della richiesta. L'analisi di queste librerie e la soluzione di questi problemi esulano dallo scopo di questo documento. La gestione del codice asincrono in Redux è un argomento vasto e complicato se analizzato in dettaglio e non apporta grossi vantaggi o svantaggi all'architettura in generale rispetto a quelli intrinsechi del paradigma. Il problema della gestione dell'interfaccia durante una richiesta asincrona invece è terreno comune ad entrambe le architetture come comuni sono le varie tecniche risolutive.

Al di là di questi problemi per prendere una decisione su quale architettura scegliere tra Flux e Redux sarebbe anche opportuno analizzare il loro ecosistema. Entrambe le architetture hanno attorno una community piuttosto attiva ed il numero di librerie e plugin implementati sia ufficialmente che dagli utenti è vasto. In ultima battuta si può concludere che la scelta concreta tra queste due eventuali architetture, nonostante tenda un po' più verso Redux è anche una questione di preferenza del programmatore. Se un approccio funzionale è quello favorito e lavorare con dati immutabili non arreca alcun problema allora l'utilizzo di Redux è la scelta più ideale, altrimenti Flux potrebbe essere comunque una ottima alternativa.

[1] https://github.com/gaearon/redux-thunk
[2] https://github.com/acdlite/redux-promise
[3] https://redux-saga.js.org/

# 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.*

[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/.

[21] Jing Chen. Hacker way: Rethinking web app development at facebook, 2014. URL https://www.youtube.com/watch?v=nYkdrAPrdcw&t=10m20s.

[22] Adam Boduch. *Flux Architecture*. Packt Publishing Ltd, 2016.

[23] Jack Franklin. Making your javascript pure, 2016. URL https://alistapart.com/article/making-your-javascript-pure.

[24] Redux documentation, 2017. URL http://redux.js.org/docs/.