

# UNIDIRECTIONAL CIRCULAR DATAFLOW ARCHITECTURE FOR REAL-TIME UPDATES

Esdras Portilho Araujo Ferreira

Thesis Submitted for the Award of MSc


School of Computing  
Dublin City University

Supervised by Mr. Ray WALSHE and Dr. Kevin CASEY

August 2019

# Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Masters of Science is entirely my own work, and that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

**Signed:**  **ID No.:** 16211619 **Date:** 05th of August 2019

# Contents

<b>Declaration</b>	<b>I</b>
<b>List of Figures</b>	<b>IV</b>
<b>Abstract</b>	<b>V</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Literature Overview</b>	<b>4</b>
2.1 Web Applications . . . . .	4
2.1.1 Why do Frameworks Exist? . . . . .	5
2.2 Real-Time Systems . . . . .	8
2.2.1 The Reactivity of Real-Time Systems . . . . .	9
2.2.2 The Predictability of Real-Time Systems . . . . .	9
2.3 Real-Time Web Applications . . . . .	10
2.3.1 Addressing Reactiveness . . . . .	11
2.3.2 Addressing Predictability . . . . .	15
2.4 Proposed Approach . . . . .	20
<b>3 Architecture and Implementation</b>	<b>22</b>
3.1 The Unidirectional Flow . . . . .	22
3.2 Using Firebase for Reactiveness . . . . .	24
3.2.1 Structure . . . . .	24
3.2.2 Reference (ref object) . . . . .	25

3.2.3	Reading Data	25
3.2.4	Writing Data	26
3.3	Using Redux for Predictability	26
3.3.1	Actions	27
3.3.2	Reducers	27
3.3.3	connect()	28
3.3.4	React Component	30
3.3.5	Using Firebase with Redux	32
4	Discussion	39
4.1	Bi-directional vs. Unidirectional	39
4.1.1	MVC	39
4.1.2	Comparison	43
4.1.3	Real-Time Engine	55
5	Conclusion	58
5.1	Opinions	59
5.1.1	State Managers	59
5.1.2	Helpers	59
5.1.3	Building the View	60
5.1.4	Support	60
5.2	Recommendations	60
5.2.1	More Comparisons	60
5.2.2	Server-side Unidirectional Flow	60
5.2.3	Convert Bi-directional to Unidirectional Flow	60
5.3	Unidirectional Flow is the Natural Step to Handle Real-Time Events.	61

# List of Figures

2.1 Major Front-end Frameworks.	7
2.2 Real-Time System.	9
2.3 Real-Time Web System.	11
2.4 Real-Time System with WebSocket.	13
2.5 Real-Time System with State Controller.	16
2.6 MobX Flow.	17
2.7 Flux Flow.	18
2.8 Redux Flow.	20
2.9 Proposed Approach.	21
3.1 MVC Flow.	22
3.2 Data Flow Using Model Binding.	23
3.3 Unidirectional Flow.	24
3.4 Update Event Sequence Diagram.	35
3.5 Duplicate Data.	38
3.6 Removing Data.	38
4.1 MVP Flow.	40
4.2 MVVM Flow.	40
4.3 Server MVC.	41
4.4 Client MVC.	41
4.5 Scale Complexity of MVC.	42
4.6 Model of Unidirectional Data Flow for First Stage.	45

4.7 Model of Bi-directional Data Flow for First Stage.	46
4.8 Model of Unidirectional Data Flow for Second Stage.	47
4.9 Model of Bi-directional Data Flow for Second Stage.	48
4.10 Model of Unidirectional Data Flow for Third Stage.	49
4.11 Model of Bi-directional Data Flow for Third Stage.	50
4.12 Total Number of Lines Comparison.	53
4.13 Maintainability Index Comparison.	53
4.14 Total Number of Statements Executed Comparison.	55

## **Abstract**

Esdras Portilho Araujo Ferreira

# Unidirectional Circular Dataflow Architecture for Real-Time Updates

The capability to present data in real-time is an essential feature to be considered when architecting modern web applications. Examination of the origin, progress and obstacles of web applications reveals how it influenced the current state of how web applications are implemented to deliver data in real-time. This research aims to investigate an alternative and reproducible approach to address real-time, moving away from outdated requisites and not extending existing implementations. When considering real-time systems, how to build an architecture that satisfies its requirements?

Based on the review of the literature, understanding the current context of web applications and exploring available libraries and frameworks, an architecture is proposed broken down in layers that are responsible for ensuring each of the real-time system's requisites. An open source library is implemented developed to wrap and abstract the implementation of the unidirectional flow. The results show that not only the unidirectional data flow architecture is theoretically superior addressing such requisites, but also more maintainable. Further research is needed to run more comparisons, also to explore the usage of this approach outside the scope of the web applications.

# Chapter 1

## Introduction

An application, or app for short, is a computer software intended to operate a group of coordinated purposes, tasks, or actions for the benefit of the user. [Christensson, 2008] Its architecture is defined by its functional and business requirements, and the goal of a given architecture is to ensure that the application is scalable, reliable, available, and manageable. [Spewak and Hill, 1993] Thus, the chosen architecture can influence—or be influenced by—the system’s design and implementation. [Eden and Kazman, 2003]

For the first applications that used the client-server model, in which the first computer (client) sends a request for some information and the second computer (server) responds to the request, the processing load was shared between the code on the server and the code installed locally on each client. This meant that an application had its own compiled client program that worked as an interface and could be independently installed on each user’s computer. Changes to the server-side system would typically also demand an upgrade for the client-side version. [W3C, 2001]

Later, web applications were introduced, where each web page was sent to the client as a static document in a standard format. Web apps are supported by a variety of web browsers using standard procedures such as HTTP. Client web software updates occurred each time a web page was visited, and the browser handled the interpretation and display of the pages, acting as the universal client for any web application. [Rauschmayer, 2014]

Elements embedded in the page could deliver interactive features for user inputs. However, for considerable changes in the page, the client had to request a new page from the server that was subsequently loaded.

Further down the line, in 1995, JavaScript was introduced by Netscape. [Newswire, 1995] JavaScript is a scripting language that is executed on the client and allows the addition of dynamic elements to the web interface. Hence, using JavaScript, no data has to be sent down to the server for tasks such as validation and iterations—the necessary code that executes these processes is already part of the initial bundle downloaded with the page.

Early implementations of JavaScript were quite basic, they were not oriented towards experienced developers, but instead mostly limited to input validation. Yet the language evolved over time, and client-side scripting introduced a set of techniques within



it to build asynchronous web applications: Ajax, or Asynchronous JavaScript and XML. Ajax essentially enables clients to asynchronously request and transmit data to servers. In the beginning, it helped to decouple the data layer from the presentation layer because applications could contact the server without loading a completely new page. [Ullman and Dykes, 2007]

As Ajax and browsers became increasingly more powerful and popular, it was soon possible to build complex applications using heavy client-side scripting. The new era of web applications that are more and more interactive began with the likes of Gmail and Google Maps. Complex applications such as these demanded better structure, and that was addressed using model-view-controller, or MVC. MVC is a design pattern that divides an application into three parts: the data and how it is structured (model); the presentation layer and how it behaves (view); and the user interaction layer and logic (controller), which is commonly used to develop and implement such web applications adequately. [Rauschmayer, 2014]

When considering web applications at this stage, there was no method that servers could trigger to inform clients that new data was available to be downloaded, and so live updates could not yet be pushed to clients. Several solutions were tested. After the first load, requesting the data again and again within a given interval was the most basic solution, called polling. It solved, to a certain extent, the problem with real-time perception. However, polling adds performance and latency problems, as servers then have to deal with many connections at the same time as well as their overheads. [Lengstorf and Leggetter, 2013]

Subsequently, more high-level transports were developed under the umbrella term Comet. These techniques consist of iframes (forever frames), xhr-multipart, html-files, and long polling. With long polling, a client opens an XMLHttpRequest (XHR) connection to a server that never closes, leaving the client hanging. When the server has new data, it is sent down the connection, which then closes. The entire process then restarts, primarily supporting server push. However, every connection to the server contains a full set of HTTP headers—this means high latency. [Lengstorf and Leggetter, 2013]

To solve these drawbacks, WebSocket was introduced as part of the HTML5 specification in 2011. WebSocket provides bi-directional, full-duplex sockets over TCP. It has meaningful improvements over the prior server push transports because it is full-duplex, does not take place over HTTP, and persists once opened. [Lengstorf and Leggetter, 2013]

Unlike polling, where each new request has the overhead of HTTP headers and handshakes to open a connection and then close it, WebSocket considerably decreases bandwidth usage and increases performance by keeping a persistent connection opened, and now, servers can push updates as soon as they are acknowledged. It is the same for clients, which can send data down to the servers. [Lengstorf and Leggetter, 2013]

Several new tools were introduced with the support of WebSocket. Between them are the real-time databases (e.g., Google's Firebase, Facebook's Parse, Meteor, or RethinkDB): databases hosted on the cloud, where every time the data is updated, it is synchronised on every connected client in real time. For example, returning to the topic of MVC, it was initially introduced into Smalltalk-80 in the 1970s. [Reenskaug, 1979]

Since then, it has evolved and become more popular after its introduction of NeXT's WebObjects in 1996. [Overview, 2001] However, at that time, most of the communication between the server and the client side was made using the request-response (client-server) model.

When discussing real-time systems, it is essential to consider their reactivity (continuous interaction between server and client) and predictability (a system's behaviour has to be predicted before its execution). [Farines et al., 2000] Thus, with the introduction of WebSocket and all the other tools it powers such as real-time databases, is MVC (or its derivations) the right architectural pattern for handling the synchronisation of data when it is continuously pushed from the server to the client? This research aims to answer this question and discuss a better approach to managing real-time applications by comparing the benefits, performance, manageability, and maintainability of the source code for apps written using MVC and unidirectional flow.

Initially, the research covers the current complexity of web apps and their growth. Then, it analyses real-time systems and their constraints. Later, by combining knowledge gained through the two previous topics, the researcher studies the current tools to build a real-time web app. Afterwards, with the help of the aforementioned tools, the application's architecture is created while the concepts of unidirectional data flow are explored and implemented. Lastly, the current architecture (MVC) is compared with unidirectional architecture.

# Chapter 2

## Literature Overview

### 2.1 Web Applications

Web applications refer to applications that are executed in the browser. Initially, web applications were quite simple; just HTML pages with CSS. At that stage pages were static, meaning there was no way for users to interact with content on them. In order for the page to react to a user's action, such as input via a mouse click, it had to contain some logic, and this logic had to run on the client's machine to maintain acceptable responsiveness. [Rauschmayer, 2014]

In the early 1990s, there were two browsers that were fairly popular: Netscape Navigator and Internet Explorer. Netscape was the first browser to develop a programming language that allowed web pages to exhibit interactivity—called LiveScript, it was embedded in the browser, which interpreted the commands directly without expecting the code to be compiled and without the need for other tools. [Rauschmayer, 2014] Although it was developed under the name Mocha, the language was formally dubbed LiveScript during the beta release of Netscape Navigator 2.0 in September 1995. However, it was later rebranded as JavaScript. [Newswire, 1995]

Internet Explorer already supported its own scripting language, VBScript, but at this point, adoption was much lower for it than JavaScript, so Microsoft added support for JScript. JScript and JavaScript were very similar, and in fact, if the programmer wrote the code carefully, it would be possible to make it fully compilable with both JScript and JavaScript. Thus, it would work on Internet Explorer and Netscape Navigator. [Rauschmayer, 2014] As the trend in popularity for JavaScript and Netscape Navigator continued, Microsoft developed a strategy of frequently updating JScript to be more and more similar to JavaScript, so applications written for Netscape would continue to work on Internet Explorer. [Rauschmayer, 2014]

Therefore, writing applications using JavaScript meant that they would work on both browsers, and this is why web applications are usually still written in JavaScript today. [Deitel and Deitel, 2011] Additionally, conventional native software applications require installation, making web applications an attractive substitute. Because the application runs in the browser environment, differences that are platform-specific are abstracted away. Although browsers have minor variations, the pace of development is regarded

as more rapid.

The difference between a website and a web application is somewhat subjective. One method of classification argues that user interactions define web applications while content defines websites, i.e., the two are differentiated by the respective emphasis on creation and modification versus consumption. This definition also has exceptions, for instance, web applications where users only consume content, but it may be seen as a fair enough generalisation. Facebook is one example of a web application. [Deitel and Deitel, 2011]

### 2.1.1 Why do Frameworks Exist?

One way to measure the growth and increasing complexity of web applications is by looking into the growth of web frameworks.

JavaScript development has changed considerably from when it was invented. The language and its engine have improved immensely since its early days; robust engines are available now such as Google Chrome's V8 and Safari's JavaScriptCore.

Consequently, as JavaScript engines can be so powerful, writing sophisticated applications that run in the browser is something common and well disseminated. Email clients and navigation websites are examples of web applications that take full advantage of JavaScript's evolution. Companies are hiring full-time JavaScript developers. JavaScript is no longer just a sublanguage used to build simple scripts; rather, it is now a standalone language, achieving its full potential. [MacCaw, 2011a]

Because HTML pages were static and required each page to be drawn from the server for every click of a link, they loaded slower. However, when Ajax (Asynchronous JavaScript and XML) arrived, the environment changed. Ajax is a group of methods that allows web applications to create fast, incremental updating of the user interface with no need to reload the whole browser page. [Chaniotis et al., 2015]

Websites that use Ajax in gathering data load take advantage of the incremental updates and decrease the quantity of data transferred over the network, resulting in better network utilisation and application responsiveness. At the start, only some website elements used Ajax in gathering data. However, after a while, single-page applications started to rise. Asynchronous JavaScript is used in these applications for downloading required data and eliminating the need for the user to refresh the browser page. Due to the reduced time in moving between application sections, the user experience was significantly improved. [Dhote and Sarate, 2013]

This sharp increase in popularity can explain the development of several new JavaScript applications. [MacCaw, 2011a]

As services move the processing from the server to the user's browser (client), navigation and other operations are performed on the client side. The server only executes extra data load/work required in the background. [Wood et al., 2014]

Unfortunately, and possibly related to the language's early days, countless applications are assembled quite poorly. For some reason, when taking into consideration the

JavaScript development environment, acknowledged principles and rules are not well considered. Developers ignore architectural models and assemble the applications into a disordered mixture of HTML and JavaScript. [MacCaw, 2011a]

But what is the proper way to manage the data flow from the server to the client? And how can we handle updates in cases of data alterations? It requires improvement in the applications' structure. Developers responded to this need as they started to make frameworks for better organisation of applications. [Chaniotis et al., 2015] Web frameworks (or web application frameworks) are methods of defining a standard procedure to build tools such as APIs, web applications, and web pages [Enache, 2017] while avoiding the pitfalls of writing and maintaining software. Frameworks offer an integrated set of software artefacts (such as objects, classes, and components) that cooperate to provide a reusable structure by guiding application developers through the steps required to ensure the creation and deployment of complex software. Frameworks often are maintained by a large community, so they end up implementing several well-tested solutions and integrations out of the box, allowing developers to focus on utilising features and writing functionality. Thus, frameworks have provided a more effective way of producing quality software. [Douglas C. Schmidt and Natarajan, 2004]

In conclusion, since frameworks arose from the need to solve problems related to building complex applications, it is fair to suggest that their increase in use is correlated with the increased usage and complexity of web applications.

## Major FE frameworks, share of registry

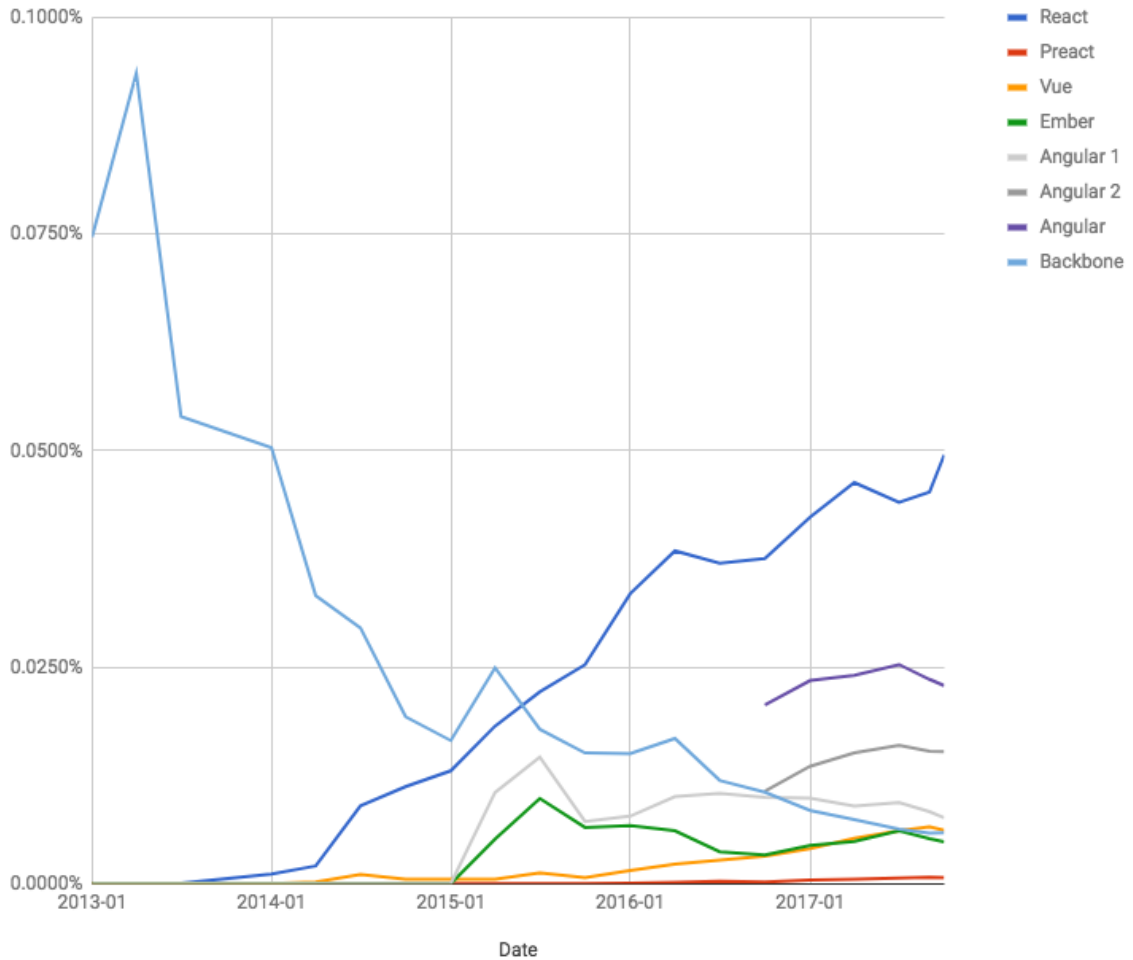


Figure 2.1: Major Front-end Frameworks.

Voss, 2018

The graph shows the popularity of the main frameworks in the *npm* community.

*npm* is the main package manager for JavaScript and the default package manager for Node.js. Currently, it is estimated that *npm* is used by around 75% of all JavaScript programmers.

By looking at *npm*'s raw statistics, it is possible to determine the number of contributions and downloads for a given package. This determination can be understood as the package's popularity. However, before considering popularity, it is more important to understand the rate of growth.

*npm*'s package rarely experiences a decrease in usage; this is because once installed, it is used as a dependency and coupled to an application, which means the package is rarely uninstalled. Thus, the number of usages will typically only increase. Therefore, the variation that makes sense to observe is the rate of growth.

However, measuring growth is difficult because everything in this space is becoming popular. It is natural to see increasing growth in every package. Therefore, measuring growth alone is not enough. Instead, one must measure the relative growth compared

against other packages in the *npm* registry at a presented interval. [Voss, 2018]

After considering this, the decline of Backbone is quite obvious. Backbone was one of the first web frameworks for sophisticated web applications. As it arrived earlier in this environment, it dominated and was very popular. However, for the past five years, the usage growth has been sharply decreasing.

The other packages that are increasing in usage are React Preact, Angular, Ember, and Vue.

What all of them have in common is the ability to provide a more straightforward way to implement applications that offer quick feedback when visitors interact with them. The biggest problem with building applications using a simple and basic JavaScript approach is updating the UI (User Interface) for every state change. Every time the state is updated, there is a lot of code required to update the UI. Modern web frameworks have implemented many built-in features and improvements since the days of bare JavaScript/Ajax coding. It means better code maintainability and overall performance.

In conclusion, the growth of frameworks suggests that an increasing number of web applications are being developed and used.

## 2.2 Real-Time Systems

As computer systems become more popular in society, the use of applications that require real-time solutions becomes more common. These applications vary in terms of complexity and the necessity to meet interval restrictions.

A system is considered real-time if the precision of an operation depends not only upon its logical accuracy, but also the time in which it is performed. [G. Shin and Ramanathan, 1994] [Davis et al., 2016] Real-time systems, as well as their deadlines, are classified by the outcome of missing a deadline:

- Hard: missing a deadline results in an entire system failure.
- Firm: the service quality is decreased by missed deadlines, but it still can tolerate it. The result is not considered if it misses its deadline.
- Soft: the result could not be valuable after its deadline, which means the quality of a system service is negatively impacted.

Among simple systems, there are smart controllers embedded into domestic utilities such as washing machines and media players. On the other extreme, there are military defence systems, industrial plants, and aerospace and railway control systems. Hard real-time systems, for instance, are the systems responsible for monitoring patients in a hospital and auto-pilot controls. Examples of firm and soft real-time systems are video games, internet teleconference applications, and media-related applications. All these systems and applications that are subject to time constraints are identified as

real-time systems. [Farines et al., 2000], [Audsley, 1993], [Young, 1982], [Stankovic and Ramamritham, 1989], [Stankovic and Ramamritham, 1990], [Joseph, 1991], [Kopetz, 1997], [Lann, 1990], [Buttazzo, 2011] and [Stankovic, 1988].

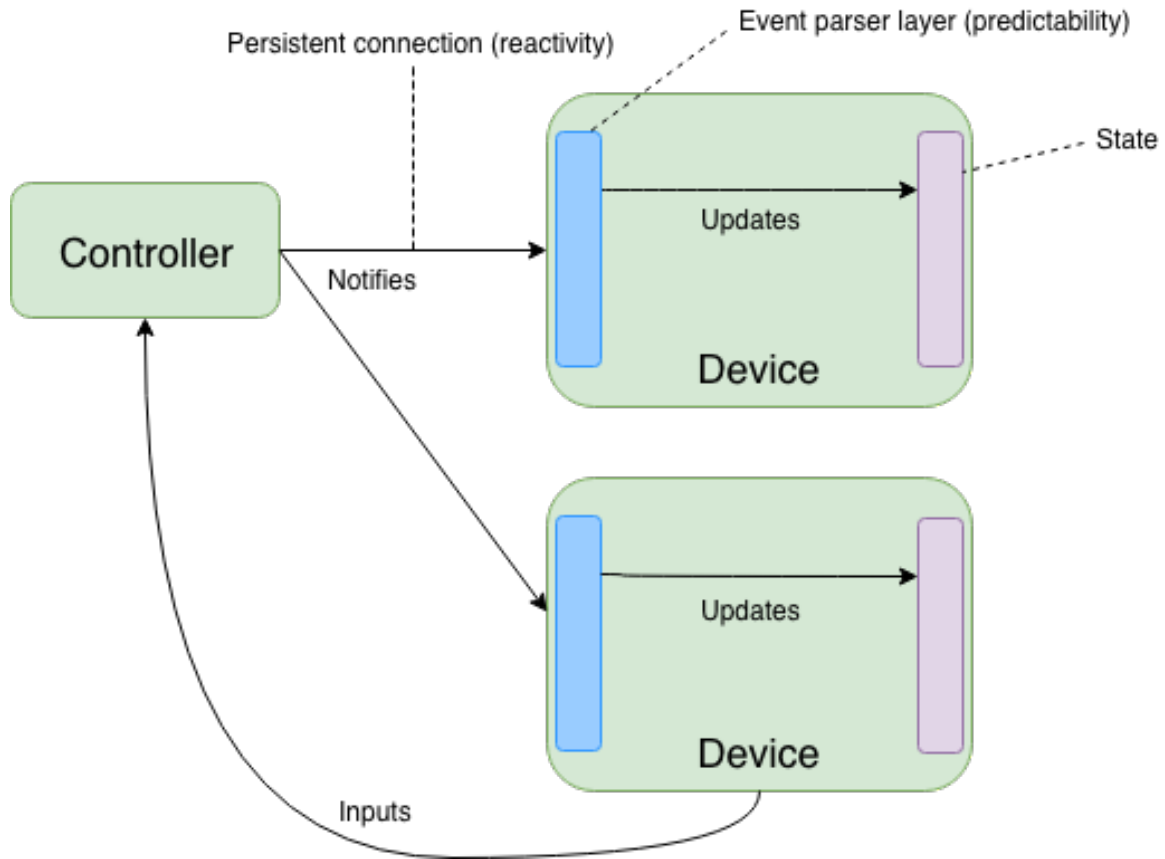


Figure 2.2: Real-Time System.

### 2.2.1 The Reactivity of Real-Time Systems

Apart from computer systems generally identified as Transformational Systems, where the process finishes when it computes an output from an input (for example, compilers and mathematical equations), there are other types that interact permanently with their environments. Among them are the so-called Reactive Systems that operate by sending responses continuously to inputs coming from the environment. [Farines et al., 2000] Real-time systems, in general, fit into this concept of reactive systems:

- A real-time system is a system that reacts to data originating from its environment and has specific deadlines. [Farines et al., 2000]

### 2.2.2 The Predictability of Real-Time Systems

One common assumption is that real-time can be achieved by increasing the computing speed alone. The rate in the calculation can improve the performance and reduce the response time for a group of tasks. However, a short response time does not guarantee that the deadlines for each task will be met. The average performance does not have



importance in the behaviour of a system with several tasks with time constraints. More than performance, the concept of predictability is essential for a real-time system. [Farines et al., 2000]

Looking at the concept of Quality of Service, it is essential to guarantee a high level of correctness for the output of a given event entering the system; this is done on the application layer. [Kyriazis et al., 2018]. This means that the developer of an application needs to make sure to parse the event correctly in order to deliver the Quality of Service for a given system. Thus, when speaking of Predictability, it also implies delivering Quality of Service in the form of correct changes to the state of the system.

A system is considered "predictable", in the logic and the temporal domain, when independent of variations occurring at the hardware level, load, and failures, the behaviour of the system can be predicted before its execution. To predict the progression of a real-time system and ensure it progresses within the specified limits and their time constraints, it is necessary to define a set of hypotheses regarding the behaviour and external environment concerning loads and failures. [Farines et al., 2000]

- Load hypothesis: it determines what corresponds to the maximum computation load generated by the system in the minimum time interval between each of the system's interactions. [Farines et al., 2000]

- Failure hypothesis: it describes the types and frequencies of failures that the system should deal within execution time while maintaining their functional and time requisites. [Farines et al., 2000]

Hence, from a rigorous point of view, to assume predictability of a real-time system, it is required to know in advance the behaviour of the system, considering the load and the potential failures. [Farines et al., 2000]

Additionally, as an optimization for critical modern systems, it is also vital to address priority assignment when processing events. When given a large number of events, it is critical to implement preference to crucial events. [Davis et al., 2016]. This is one more reason to have a strong predictability layer implemented on the system in order to perform optimizations. [Zhao et al., 2018]

## 2.3 Real-Time Web Applications

When analysing Real-Time Systems, there are two points to focus on: the ability of the system to be reactive, and the ability of the system to be predictable.

A Real-Time Web App is defined by its ability to have a continuous connection stream with the server so that the client can stay up-to-date with any changes on the server (frequently caused by other clients) in real-time.

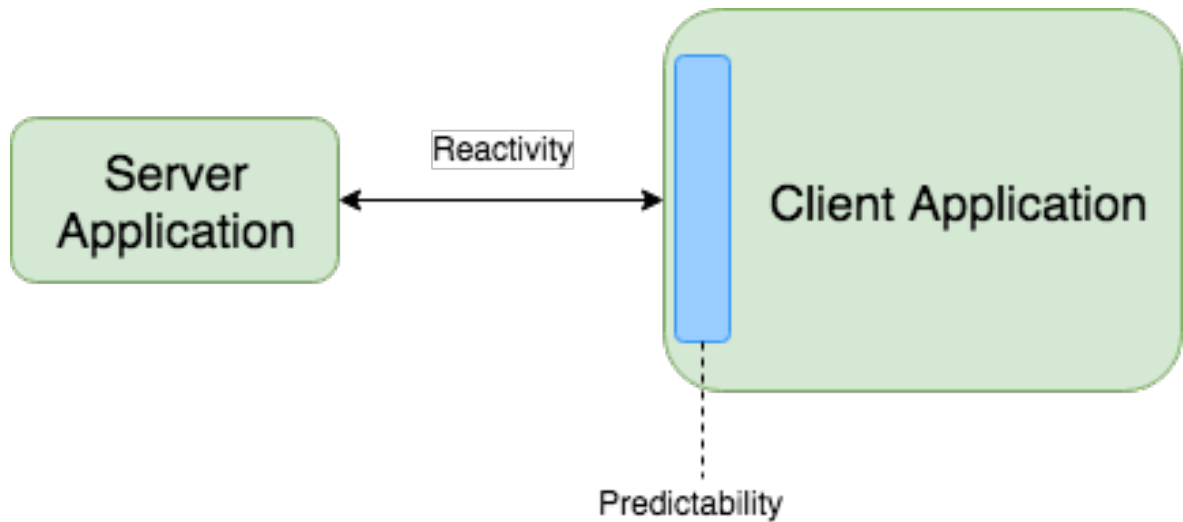


Figure 2.3: Real-Time Web System.

How can these two points be addressed in the scope of web applications? What are the options that can tackle Reactiveness and Predictability in the web space?

### 2.3.1 Addressing Reactiveness

When addressing reactivity, it is essential to remember that the system needs to be continuously responding to the environment.

#### The Pursuit of Real-Time

The earliest web design was known as the request-response model, where the client requests something from the server and the server, during its turn, responds with some data — it could be the requested information, an error, or another type of response. Then, as previously discussed, to improve the responsiveness and dynamism of web content, Ajax was introduced. With this, the client could request data from the server in the background and render it, making the experience smoother for the user. However, there was no way for the server to reach clients and inform them of the availability of new (live) data. [MacCaw, 2011b](#)

As explained, there were many solutions developed to overcome this, the simplest being polling, which means the client keeps sending requests to the server over and over again. Eventually, when the server has new data, it would be requested by the client, and the process would continue. [Loreto et al., 2011](#)

The polling solution provided the perception of real-time. However, it introduced latency and performance problems because servers had to process a large number of connections per second with both TCP handshakes and HTTP overheads. Although polling is still used, it is not ideal.

Later, more high-level transports were conceived under the generic term Comet. These systems consisted of iframes (forever frames), XHR-multipart, HTML files and long

polling. Utilising long polling, a client opens an XMLHttpRequest (XHR) connection and requests new data from the server. If new data is available, the server sends it down through the connection, which then closes. Regardless of whether new data was available, the entire process then starts again, basically emulating real-time.

Comet techniques had browser compatibility problems as well as performance issues. Each connection to the server included a full set of HTTP headers, thereby increasing latency.

Java and Adobe Flash offered TCP socket connections that would enable the page to perform in real-time. Yet applications that depended on Java and Adobe Flash were not guaranteed to work, since those languages were not built into the browsers but available as plug-ins. They also created obstacles on many business networks because of security concerns. [MacCaw, 2011b](#)

## WebSocket

WebSocket is a communication protocol that provides bi-directional, full-duplex communication channels over a single TCP connection. The IETF regulated the WebSocket protocol as RFC 6455 in 2011, and the World Wide Web Consortium (W3C) standardised the WebSocket API in Web IDL.

With WebSocket, the clients can get live data without long polling, as the connection is persistent. Additionally, since WebSocket is within the HTML5 specification recommended by the W3C, there is no need for browser-plugins. W3C defines the standards for the World Wide Web, so browsers that follow the W3C's specifications (e.g., Chrome, Opera, Firefox, and Safari) have built-in support.

Unlike HTTP, WebSocket maintains the connection once opened; it performs the handshake with a server once. Afterward, there is no need for HTTP headers with each piece of information exchanged, which improves performance since the use of headers with each poll increases bandwidth usage. Plus, because the clients can send data to the server, the server can push data to the clients immediately after new data is made available. [Wang et al., 2013](#)

The first operation necessary to start a WebSocket connection is execution of an HTTP "upgrade" request. The server then responds with the handshake. Next, the two-way channel protocol is established. [Fette and Melnikov, 2011](#)

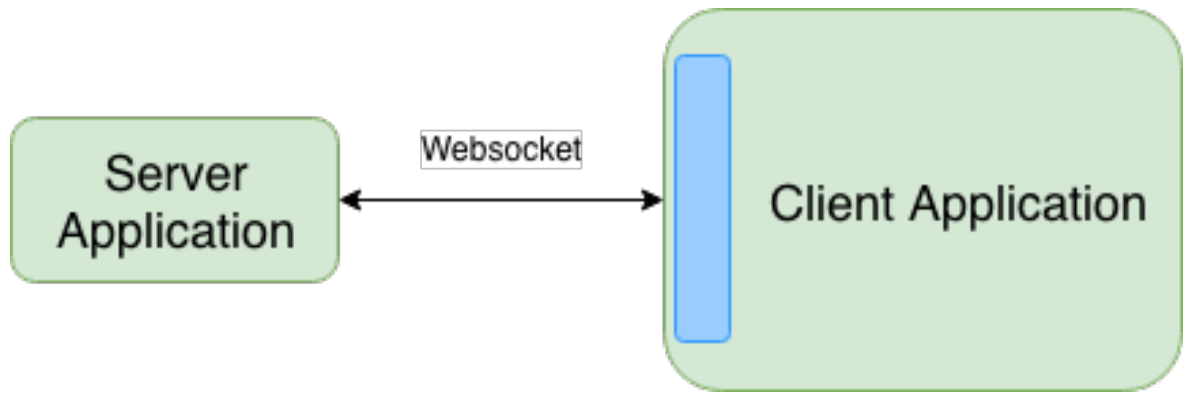


Figure 2.4: Real-Time System with WebSocket.

## Real-time Databases

Acknowledging the demand for reactivity on the database side, a new class of database systems have arisen that are natively push-oriented and, thus, ensure promotion of the development of reactive web applications.

Generally, push-oriented architecture is a type of web communication where the server does not end a connection after the response data has been delivered to the client. The server keeps the connection open so that a potential update can be sent out immediately; the publisher or the server launches the request for this operation. It differs from the pull-oriented (request-response) access pattern where the request for the transmission of data is launched by the client. [Thomson et al., 2016](#)

This event stream is supported by WebSocket that makes it possible to keep the connection between the client and database running and responsive.

The systems that use push-oriented architecture are often called real-time databases because they keep data at the client side synced with the current database state “in real-time” by proactively throwing changes. Furthermore, they decrease complexity and provide increased efficiency of development, i.e., immediately following the change. (Note that the term real-time database is overloaded; this paper does not refer to databases that produce an output with strict time constraints.)

There are fundamentally two distinct kinds of real-time queries:

1. *Self-maintaining queries* follow the same definition as common ad hoc queries, but offer a comprehensive and updated result whenever there is a change.
2. *Event stream queries* provide fine-grained information on events as they occur. Consequently, they expect some knowledge of the underlying mechanics—for instance, what types of events there are and what properties they have.

In real-time databases, the most prominent representatives are Meteor, RethinkDB, and Firebase.

## Meteor

Meteor [\[Meteor, 2018\]](#) is a set of tools that has reactive apps and websites written in JavaScript as the primary audience. It uses the MongoDB database and consequently exposes through its own API queries that mimic MongoDB queries combining event stream queries on top of it. Meteor offers two different implementations to identify relevant changes on the documents so they can be broadcast to listeners:

1. *Change monitoring + Poll-and-Diff*: This approach is only used as a fallback. Change monitoring means that each Meteor app server controls the write operations it receives to identify and manage state changes that are relevant for running real-time queries. If a recently inserted object matches a real-time query, the object is sent to all clients listening to this particular query. However, in the case of multi-server implementation, it does not work as expected because inserting an object in server A will not trigger the query for clients listening to server B. Therefore, Meteor employs the Poll-and-Diff: "poll" means that Meteor will periodically run the query, and it will compare the result with the previous result. Then it will make a "diff" of the results, the diff being the portion that was updated since it was last run. Finally, it will push the diff to the client so the client can show the most up-to-date data while the polling operation continuously proceeds.
2. *Oplog Tailing*: This takes advantage of how MongoDB implements writing scalability: It distributes the data in order to make redundant copies by first writing in the primary partition log and then moving to write on the additional partitions. Meteor watches each primary log and, by doing so, never misses an update, making Poll-and-Diff obsolete.

## RethinkDB

RethinkDB [\[RethinkDB, 2018\]](#) is an open-source database built for real-time web applications. It utilises a persistent connection to continuously push updated query results to applications in real-time instead of polling for changes like traditional request-response architecture.

To implement real-time push architecture in an efficient way, the RethinkDB company created most of its own database elements, including the query execution engine, the distributed system, the caching subsystem, and the storage engine. Since the architecture affects every part of the database, RethinkDB is implemented in C++. It was built with the help of hundreds of contributors from around the world.

RethinkDB provides drivers for JavaScript/Node.js, Ruby, Python, and Java. There are other drivers made by the community for other languages, including .NET, Go, and PHP.

It also can be written to support request-response paradigm applications. For instance, the following code would return the query once:

```
1 r.table('userA').get('todos').run();
```

This other example shows how to subscribe to a stream of updates any time the record changes:

```
1 r.table('userB').get('todos').changes().run();
```

In addition to the real-time push architecture, RethinkDB offers the following:

- A query language that supports table joins, subqueries, and parallelised distributed computation.
- Operations and monitoring API that integrate with the query language.
- An administration UI to shard and replicate. Additionally offers online documentation and query language suggestions.

## Firestore

Firestore Real-time Database [\[Firestore, 2018\]](#) is a cloud-hosted database. It uses format JSON to store the data and exposes it directly to connected clients, which can read it and receive updates as soon as the data is updated.

The connection between clients and Firestore Real-time Database is based on Web-Socket.

Firestore Real-time Database's instances are not a collection of data stored in JSON. Instead, it is a single JSON document. Clients can access each level of the JSON document by specifying the sequence of keys. It is possible to access and keep listening to any level of the document, so every time some data changes on any point of the tree below the key to which a given client is connected, the client will be informed and updated.

The database's APIs enable applications to access the data directly, with no mediator or server side. Clients connect straight to the data. Security and data access rules are applied for read and write operation and for each specific level of the document tree.

### 2.3.2 Addressing Predictability

When addressing the predictability aspect of real-time applications, it is necessary to keep in mind that the "predictability" of something (for instance, the time when an action completes) is the degree to which it can be known in advance. Therefore, there must be a method or tool available for measuring that degree.

The so-called state controllers are tools that help applications behave consistently; they organise the application state by "owning," storing, and updating it.

The application state is the local data that describes the current status of the application. The derived data from the application state is what is rendered and what dictates the behaviour of the application.

Moreover, a state controller is also necessary to prevent the state from being changed directly by forcing the input (this could be a result of a user action or a response from the server) to describe the changes, thereby making state control predictable. The same actions in the same order lead the application to end up in the same state. All the expected types of events (and their payloads) entering the app are mapped and have their specific and dedicated handlers.

Another problem that state controllers solve is determining a method to detect data mutation and handle its side effects. Side effects of data mutation include actions such as:

- Syncing the interface with the new data.
- Re-calculating computed variables.
- Navigating through a different route.

As data mutation using an assignment operator (such as `user.name = "John Doe"`) does not provide a mechanism to detect the change (`Object.observe()` has been deprecated), there is a requirement for state management libraries.

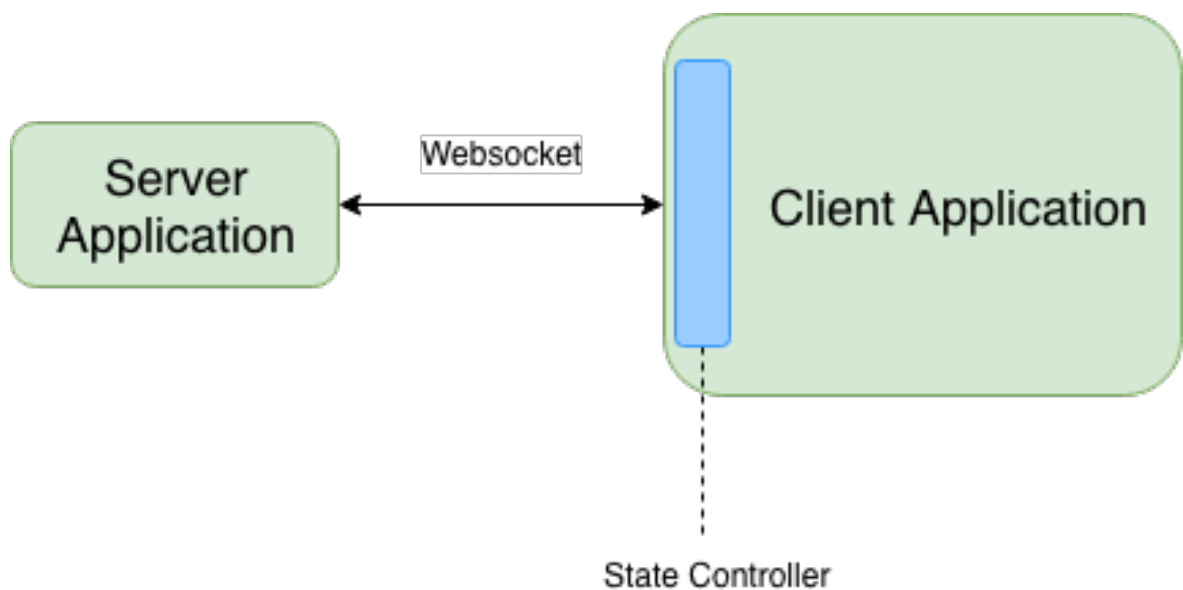


Figure 2.5: Real-Time System with State Controller.

## MobX

MobX [\[MobX, 2018\]](#) is a library that offers state management by applying functional reactive programming. The philosophy behind MobX is: "Anything that can be derived from the application state should be derived. Automatically." This includes the UI, data serialisation, server communication, and so on. MobX has only a few core concepts:

1. **Observable state:** MobX adds observable capabilities to data structures such as objects, arrays, and class instances. This can be done by annotating the

class properties with the `@observable` decorator (ES.Next). Using an observable decorator is similar to turning the property of an object into a spreadsheet cell. However, unlike spreadsheets, these values can not only be primitive values but also references, objects, and arrays.

2. **Computed values:** MobX can be used to set values that will be derived automatically when the relevant data is modified. Any value can be computed by applying a method that purely operates on other observable values. Computed values can range from the concatenation of a few strings up to deriving complex object graphs and visualisations. Since computed values are observable themselves, even the complete rendering of an interface can be obtained from the observable state. Computed values might decide either lazily or in response to state changes.
3. **Reactions:** A reaction is similar to a computed value. However, rather than creating a new value, it produces a side effect. Reactions link reactive and imperative programming for tasks such as printing to the console, making network requests, incrementally updating a component tree to patch the DOM, and so on.
4. **Actions:** Actions are the basic means to transform the state. Actions are not reactions to state changes, but take sources of change, such as user events or incoming web-socket connections, to modify the observable state. Actions will batch mutations and only notify computed values and reactions after the actions have been completed. This guarantees that intermediate or unfinished values generated during an action are not visible to the rest of the application until the action has been terminated.

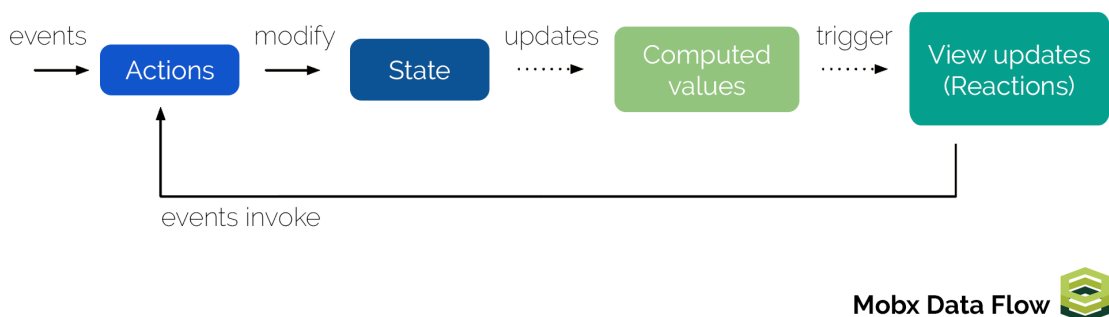


Figure 2.6: MobX Flow.

[MobX, 2018]

## Flux

Flux [Facebook, 2018a] is the application architecture built and utilised by Facebook for developing client-side web applications. It complements React's components by utilising a unidirectional data flow. It is seen as more of a pattern rather than a framework, and it is possible to start using Flux quickly without generating a lot of new code.



Applications based on Flux have three principal sections: the dispatcher, the stores, and the actions and views (React components).

A unidirectional data flow is fundamental to the Flux pattern. The dispatcher, stores, and views are independent nodes with distinct inputs and outputs. The actions are simple objects containing the new data and an identifying type property.

1. **Actions:** Actions are a mechanism to update the state in a given store. The Dispatcher exposes a method that can trigger actions. Actions should be the only alternative to change the data in the store.
2. **Dispatcher:** The Dispatcher is a simple tool that allows the creation of actions to interact with the store. It has no particular logic on what an action should update; it simply dispatches the actions.
3. **Stores:** The Stores hold the state of the application; they also hold the logic on how the data is updated when an action is dispatched. By using a switch, they identify what type of action was dispatched, and respond accordingly. Therefore, all the expected actions should be registered in the store as the algorithms to update the state with a given action type/payload. Every time the store is updated, it emits an event broadcasting the existence of the new state.

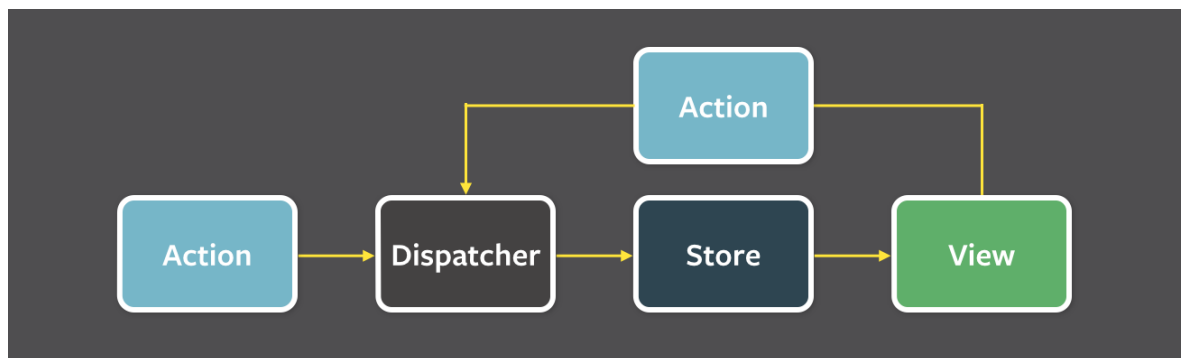


Figure 2.7: Flux Flow.

[Facebook, 2018a]

## Redux

Redux [Abramov, 2018] is a predictable state container. This means that it is easy to identify the source of the state change. Redux uses a single store, and it is often called "single source of truth," as only one place can access the state: the store. As applications grow more complex, it is necessary to maintain more states.

Redux has the same architecture as Flux. However, Redux subtracts some complexity by using a functional composition while Flux uses callback registration. There is not a fundamental difference, but it makes some abstractions more natural or, at least, possible to implement. [Caspers, 2017]

Redux makes state changes predictable. It accomplishes this by placing certain rules.

- The state must be immutable.
- Actions should contain the data describing the state change.
- Reducers perform the state change, and reducers should be pure functions.

The Redux library fulfils these policies by designating a store as the single source of truth, actions that represent the event that has happened, and reducers that specify how the application's state changes in response to an action.

1. Store: The store contains the entire state in the application. The state structure should be implemented as a tree-shaped hierarchy of plain JavaScript objects, referred to as the state tree. The store provides a `getState` method returning the state and a `dispatch` method for dispatching actions. It also provides a `subscribe` method allowing components to listen to their state change in the store.
2. Actions: Actions are objects that represent the description and payload to change the state. Action objects must contain all the data that is going to the store. They are required to have a `type` field to make them identifiable by reducers (see subsection: Reducers). The data contain additional fields; a common pattern is to use a `payload` field to hold the data of an action. The actions are dispatched via the store's `dispatch` method. Storing actions is similar to using the event sourcing pattern because actions are descriptions of change. Actions can be dispatched from anywhere, including callbacks.
3. Reducers: A reducer is a pure function that uses the previous state and action data as parameters to decide the next state according to that action. It can either produce a new state or return the current state unmodified. An application can have more than one reducer. However, since Redux's `createStore` function takes a single function as a parameter, it is necessary to use Redux's `combineReducers` function to return such a function. It takes an object as its parameter. As the name suggests, it maps the sub-state that a reducer operates on as a field name to that reducer. Note that all reducers will be called by every action [26].

The organization of modules within the Redux layer is shown below.

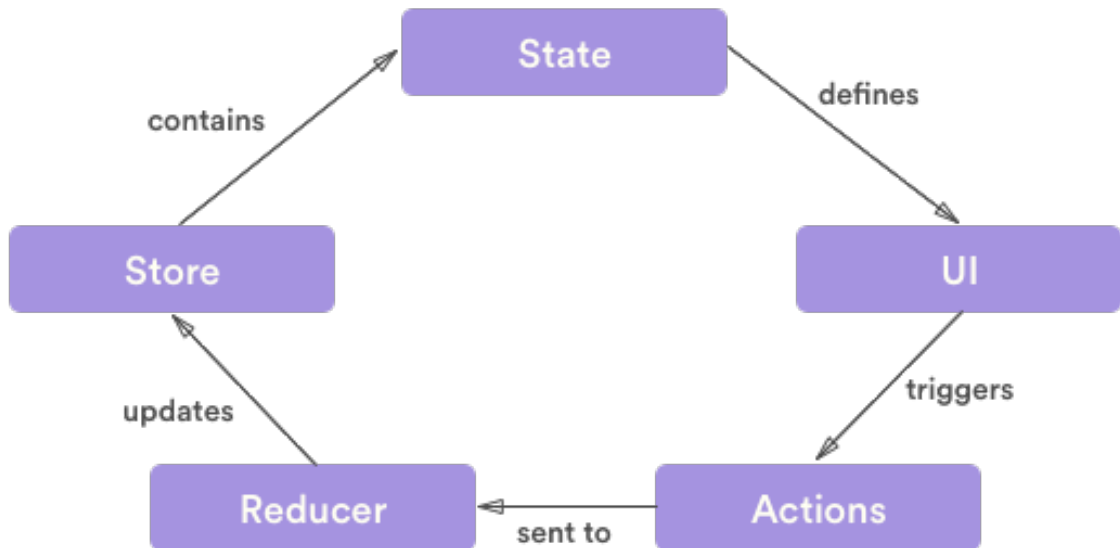


Figure 2.8: Redux Flow.

[Abramov, 2018](#)

## 2.4 Proposed Approach

From this point, to facilitate understanding and simplify the task of describing the concept, a combination of specific tools will be used to demonstrate the unidirectional flow: React, Redux, and Firebase. The main point of using these particular tools is their excellent documentation and simplicity.

The proposed architecture has three parts, and each part has one clear responsibility.

- The first part is View: rendering data to the user and then re-rendering it once the data changes. This will be handled by React, which is a Facebook-developed JavaScript library that was initially a simple internal project until it became open-sourced in 2013. The primary purpose of this library is to build user interfaces that have data that changes over time. [Fedosejev, 2015](#))
- The second part is the state container; it controls the state and the data flow. In other words, it mainly takes care of the predictability of the application. Redux handles it.
- The third part is the persistent connection handler that keeps the stream of communication open, guaranteeing the reactivity in the system. Firebase manages this task.

It is important to emphasise that although the tools chosen have particular implementations and specific APIs, the unidirectional flow is not tool-dependent and can be achieved with the right architecture, since it provides a way to cover reactivity and predictability.

Many benefits can be accomplished only by using the correct web application architecture. The first benefit is to increase maintainability—this would allow for easy addition of new features, as well as discovery of bugs. The next goal is reducing complexity, as this would allow for straightforward reasoning with regard to different application parts and understanding how the pieces communicate with one another. [Dulimarta, 2017](#)

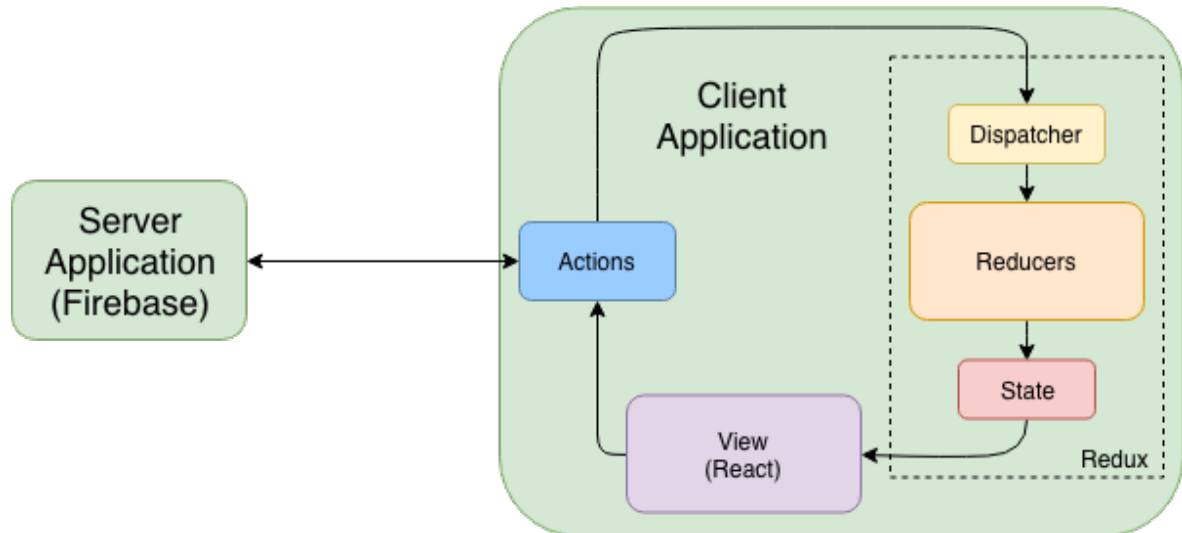


Figure 2.9: Proposed Approach.

# Chapter 3

## Architecture and Implementation

### 3.1 The Unidirectional Flow

Unidirectional data flow is a technique where the data in the application flows through in a single direction with better control over the application state, thereby ensuring a clean data flow architecture. Before one can understand the benefits of unidirectional data flow, it is necessary to analyse the inherent problems with the alternative, bi-directional data flow. This data model is typically a type of model binding.

The model-view-controller, or MVC, pattern, where a model is responsible for the storage and retrieval of application data, has become the default application architecture. The "view" is the presentational piece of the application and what the user interacts with. The controller is in charge of converting data from a model so that the view can display the data. [Street, 2015](#)

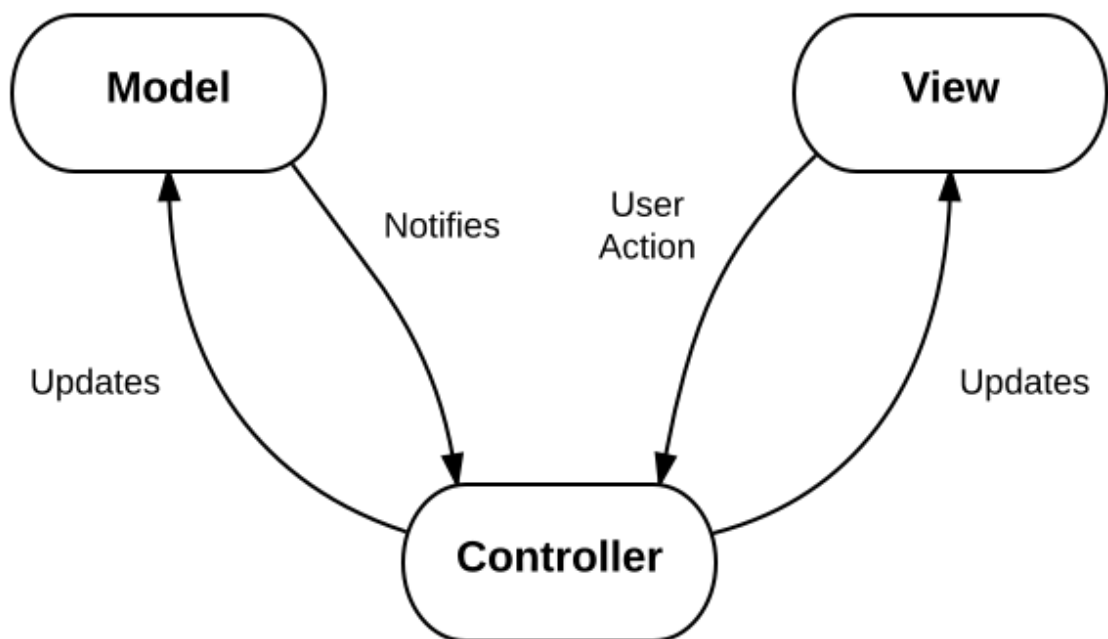


Figure 3.1: MVC Flow.

MVC enforces the Single Responsibility Principle: The view does not handle the controller logic, the controller does not model the data, and the model does not factor in how the data is displayed. This separation of concerns is great for simple applications, but it has problems scaling to fit complex solutions. As more models and views are added to the system, the controller becomes busier. In an attempt to decentralise the processing from the controller, the information in the model is bound to the view whenever changes in the model are replicated in the view layer, and vice versa. It relieves the controller but makes the state unpredictable, as the controller is ignored and two ends are managing the data. [Street, 2015]

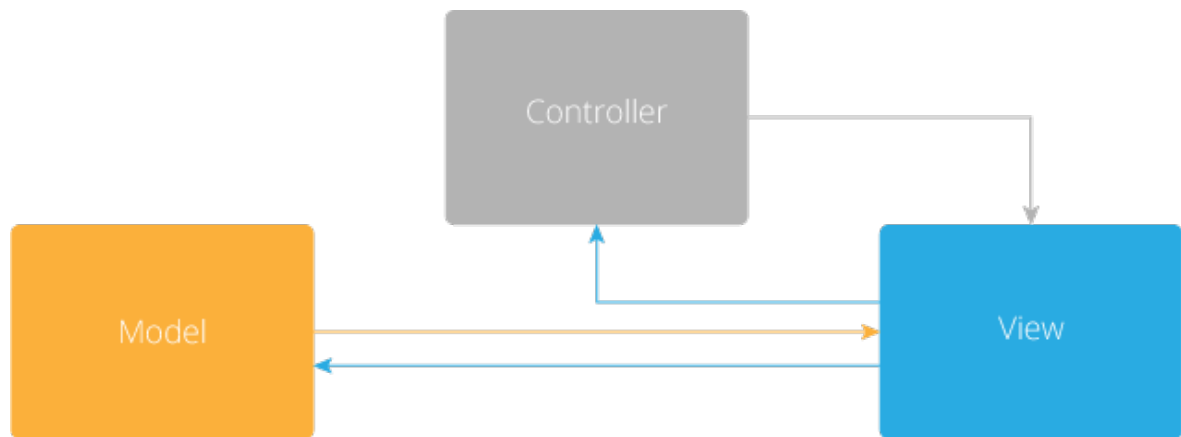


Figure 3.2: Data Flow Using Model Binding.

Alternatively, the unidirectional data flow approach provides a predictable application state. In a typical unidirectional application structure, changes in an application view layer trigger an action in the data layer. These changes are later sent back to the view. It is essential to see here that the view does not directly modify the application data. [Street, 2015]

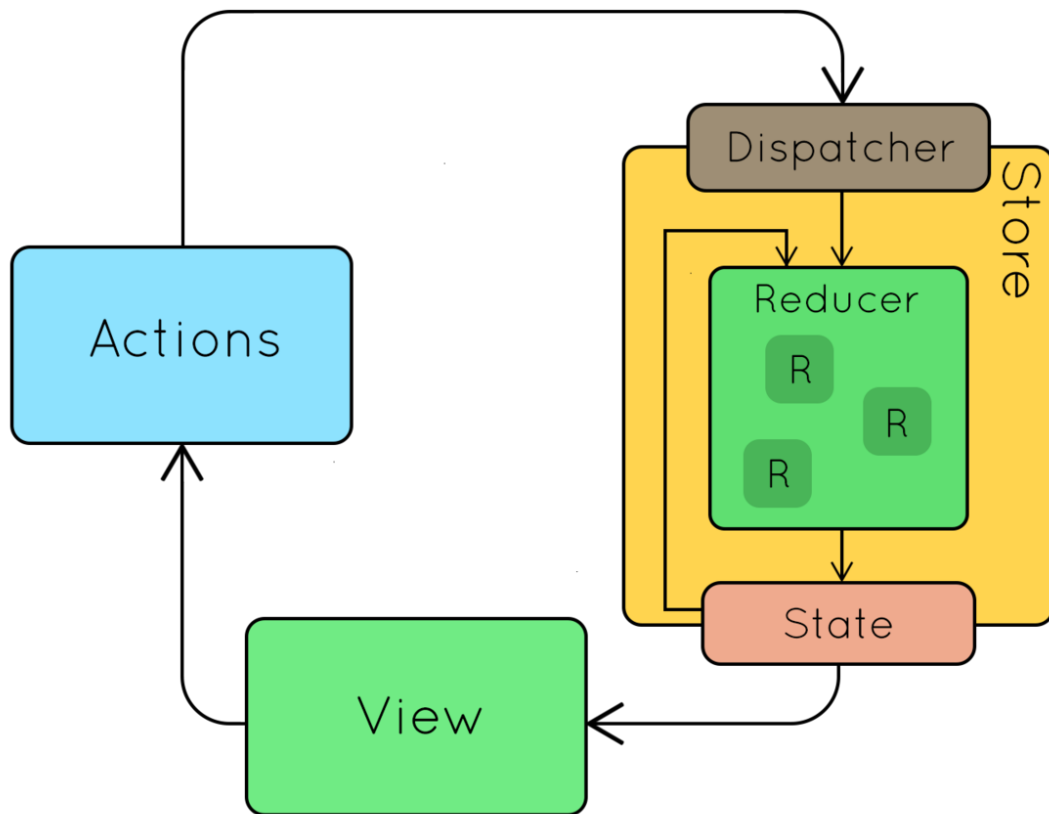


Figure 3.3: Unidirectional Flow.

Using frameworks such as React, the view is just a function of the application data. If provided the same set of values, the view will produce the same output. This is typically described as a pure function, and it is an excellent way to manage unidirectional data flow with a predictable application state. One pure function only works on a given set of inputs, returns a value, and does not create any side effects. While pure functions are not the exclusive means of performing a unidirectional data flow, they can be the easiest to argue in favour of and much more straightforward to test. The absence of side effects in a unidirectional flow of data addresses the biggest issue with a bidirectional flow. [Street, 2015]

By implementing a solution that flows in one direction, the application state is predictable. The data transformation is centralised as parts of the pipeline, which makes handling updates and even errors predictable. The use of pure functions means increased reusability and extendibility. [Street, 2015]

## 3.2 Using Firebase for Reactiveness

### 3.2.1 Structure

As discussed previously, Firebase stores data as a JSON document; the data can be written or read by specifying its key structure that is used as an address.

Although it is possible to nest the data in the document up to 32 levels deep, it is usually not a safe approach to do so. Because access rules are applied to every branch under a given key, maintaining deeply nested structures under a node means that whoever has access to this node also has access to all of the structures under it.

### 3.2.2 Reference (ref object)

Each node in the tree is named a Reference (*ref*), and each ref has its own URL. A reference is used for reading and writing operations, as it expresses the path in the structure where the data is located.

The root or child reference of the database can be retrieved by calling:

```
1 const rootRef = firebase.database().ref();
2 const childRef = firebase.database().ref("child/path");
```

A *ref* can be updated or queried like a regular database. Those queries or updates can occur from the front end. Firebase provides an SDK for iOS apps, Android apps, and web applications (JavaScript SDK).

### 3.2.3 Reading Data

A *ref* can emit events based on the data changes at the server by attaching an asynchronous listener. When it is created, it is triggered immediately with the existing data on the given *ref*, and it is triggered again whenever data under that path changes.

To read data on a path and listen for modifications, it uses the *on()* or *once()* methods.

```
1 const todoNameRef = firebase.database().ref('todos/' + todoId + '/name');
2 todoNameRef.on('value', (snapshot) => {
3   updateTodoName(snapshot.val());
4 });
```

The *on()* method receives a callback function as an argument. The callback function is invoked every time data nested under the path changes. It is invoked passing a snapshot as a parameter. The snapshot holds all the data as it is after the change that triggered the method. The snapshot then exposes a function *val()* that returns its data.

*once()* has a similar structure in that it receives a callback function, but it also has a very different behaviour: It does not subscribe to changes. The callback function is only triggered once, and this behaviour is useful for situations where the data does not need to be updated in real-time.

```
1 const userId = firebase.auth().currentUser.uid;
2
3 return firebase.database()
4   .ref('/users/' + userId)
5   .once('value')
6   .then(function(snapshot) {
```



```

7     const username = (snapshot.val() && snapshot.val().username) || '
    Anonymous';
8  });

```

### 3.2.4 Writing Data

*set()* is used for simple write operations - it just overwrites all the data under the reference in case the data already exists.

```

1 function writeUserData(userId, name, email) {
2   firebase.database().ref('users/' + userId).set({
3     username: name,
4     email: email,
5   });
6 }

```

The *updated()* method is used to concurrently write to specific children of a node without overwriting other child nodes. *updated()* updates lower-level child values by defining a path for the key.

```

1 function writeNewTodo(uid, userId, title, text) {
2   // A todo entry.
3   const postData = {
4     userId: userId,
5     uid: uid,
6     text: text,
7     title: title,
8   };
9
10  // Get a key for a new Todo.
11  const newTodoKey = firebase.database().ref().child('todos').push().
    key;
12
13  // Write the new todo's data simultaneously in the todos list and
    the user's todo list.
14  const updates = {};
15  updates['/todos/' + newTodoKey] = postData;
16  updates['/user-todos/' + uid + '/' + newTodoKey] = postData;
17
18  return firebase.database().ref().update(updates);
19 }

```

## 3.3 Using Redux for Predictability

Within the scope of applications, data arriving in from a WebSocket connection typically modifies the state of the application and eventually needs to be reflected in the GUI in some form.

The Redux store holds and manages all the states for the application and can also dispatch actions. The Redux store should be unique for each application. The data coming from a WebSocket connection, as an example, would trigger the store to dispatch an action. Actions, then, end up handled by reducers, which compose the data

and merge it with the current application state; they then trigger a render and are reflected in the GUI.

Following is a detailed explanation of actions, reducers, and how they are connected to the state and GUI.

### 3.3.1 Actions

The entry point to manipulate the state on the store is through actions. Whether discussing UI interactions, a response from the server, or snapshots from Firebase, they all need to be dispatched to the store as actions using *store.dispatch()*.

Actions dispatched by the store are JavaScript objects. Every action has to contain a specific field: *type*. The *type* is usually constant and is used by the reducers to identify and manipulate the payload.

```
1 {  
2   type: ADD_TODO,  
3   text: 'First To-Do item'  
4 }
```

This explains practically how the predictability of the system is addressed since, for coherence, there is a finite amount of action types to be defined. Thus, there is a limited number of ways that the store can be manipulated. Furthermore, by handling any source of data with an action, the system is aware of the data that is entered into the store, and it controls how this data is added to the application state.

Action creators are functions that create actions. It is easy to conflate the terms “action” and “action creator”—hence, it is necessary to be careful when using the terms.

```
1 function addTodo(text) {  
2   return {  
3     type: ADD_TODO,  
4     text  
5   }  
6 }  
7  
8 store.dispatch(addTodo(text));
```

### 3.3.2 Reducers

The store needs to manipulate the actions’ payload and reduce into the state. For this, it uses reducers. Reducers are a basic concept of functional programming. Developers implement a specific logic to handle each payload based on each action’s type. The return result is merged into the state.

The first task is to define the minimal data representation of the application’s state as an object. For instance:

```
1 {  
2   visibilityFilter: 'SHOW_ALL',
```

```

3   todos: [
4     {
5       text: 'Consider using Redux',
6       completed: true,
7     },
8     {
9       text: 'Keep all-state in a single tree',
10      completed: false
11    }
12  ]
13 }

```

After it is decided what the state object looks like, it is ready to write a reducer for the same. A reducer is a pure function that uses the previous state and an action to return the next state.

```

1 (previousState, action) => newState

```

Given the same arguments, as pure functions, reducers should compute the next state and return it with no side effects or asynchronous operations—just a calculation.

```

1 const initialState = {
2   visibilityFilter: VisibilityFilters.SHOW_ALL,
3   todos: []
4 }
5
6 function todoAppReducer(state = initialState, action) {
7   switch (action.type) {
8     case SET_VISIBILITY_FILTER:
9       return Object.assign({}, state, {
10         visibilityFilter: action.filter
11       })
12     default:
13       return state
14   }
15 }

```

It is important to note that when the application starts, Redux sends an action with an undefined type. This is for setting up the initial state.

From the beginning, it is necessary to understand that Redux has no specific relation to React. It is possible to use Redux with React, Angular, Ember, jQuery, or vanilla JavaScript.

That said, Redux operates particularly well with libraries such as React and Deku because Redux emits state updates in response to actions and they allow the representation of the UI as a function of the state.

### 3.3.3 connect()

Redux transfers states from the store to React components by using a higher-order component. The npm package of React-Redux provides a higher-order component called *connect*.

A higher-order component (HOC) is a method in React for reusing component logic.

HOCs are not part of React per se, but simply patterns that emerge from React's compositional character.

```
1 const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Basically, HOCs are wrappers that pass data and functions down to components. An HOC is a function that receives a component as one of its parameters and enhances that component in a certain way.

*connect* receives a React component as an argument and decorates it without modifying it.

```
1 connect(  
2   mapStateToProps,  
3   mapDispatchToProps,  
4   mergeProps,  
5   options,  
6 )(component);
```

As seen from the example, there are four parameters sent to the *connect* function. One, *mapStateToProps*; two, *mapDispatchToProps*; three, *mergeProps*; and four, *options*. All of them are optional.

*connect* returns a function that receives one more argument: the wrapped component.

**mapStateToProps** needs to be a function. This function does what its name suggests: maps a state from the store to the component passing as props.

```
1 const mapStateToProps = ({ reducer1, reducer2 }) => ({reducer1,  
   reducer2 });
```

There is no need to import the reducers into the file, since they are provided as dependency injected in the *mapStateToProps* function. The first argument is the whole Redux store's state. Derivating data from the reducer before passing the process of mapping it to the component is encouraged by its authors.

When *mapStateToProps* is specified, it is called on every state change and will be passed to the new state.

The results of *mapStateToProps* must be a plain object that will be merged into the component's props. It allows for the possibility of accessing the reducer state objects and values from within the React components. The object returned has the keys that will be the props names in the component.

If the *mapStateToProps* function is declared as receiving two parameters, it will be invoked with the store state as the first parameter, as previously explained, and the props passed to the connected component, as the second parameter, will be recalled whenever new props are sent to the connected component. This is determined by shallow comparisons. (The second parameter is usually referred to as *ownProps* by convention.)

```
1 const mapStateToProps = ({ reducer1, reducer2 }, ownProps) => ({  
2   reducer1,  
3   reducer2,  
4   ...ownProps,  
5 });
```

**mapDispatchToProps** should be declared as an object or a function. If it is an object, each key of the object should represent an action, and Connect will automatically call each of the actions using dispatch. However, when it is created as a function, the dispatch method is injected as the first parameter, so the developer can create custom logic and wrap the action creators with the dispatch method.

```
1 import { actionCreator1, actionCreator2 } from '../actions';
2
3 const mapDispatchToProps = (dispatch) =>
4   ({
5     actionCreator1 : () => dispatch(actionCreator1()),
6     actionCreator2 : (value) => dispatch(actionCreator2(value)),
7   });
```

Two important things are happening in the code above. The first is that it is setting up props that hold the action creators. The names of the props are kept the same as the name of the action creator. However, this is not compulsory.

The second important thing is binding the action creators to dispatch. Without this binding firing, an action creator will do nothing. As with `mapStateToProps`, there are also dependencies being injected in this method. The dispatch method is injected by *connect* in this function.

If the *mapDispatchToProps* function is declared as taking two parameters, it will be invoked with the first parameter being dispatch, and the second parameter (commonly referred to as *ownProps* by convention) is the props passed to the connected component. It will be re-invoked whenever the connected component receives new props.

**mergeProps** is a function. If specified, it receives the result of *mapStateToProps()*, *mapDispatchToProps()*, and the parent props. It is used to modify the object that will be passed as props to the wrapped component. Usually, it is not specified and when omitted, the default behaviour is a returning of the plain object.

```
1 (Object.assign({}, ownProps, stateProps, dispatchProps));
```

**options** is an object; it is used to specify further customisation to the behaviour of the connector.

### 3.3.4 React Component

React bindings for Redux adopt the concept of classifying presentational and container components.

Presentational components:

- Implement how components look.
- Often outputs DOM (document object model), but can also delegate to other components.
- Work with no dependencies and should be always reusable.

- Do not implement how data should be stored or implement data transformation.
- Can only receive data through props.
- Rarely have their state, when they do, it is regarding data presentation.
- Include the following examples: header, button, card, userphoto, and table.

Container components:

- Coordinate how components operate.
- Can compose presentational and other container components.
- Compute data and behaviour for presentational and other container components.
- Usually, have their state.
- Are usually produced through third-party libraries like Redux, Relay, or Flux.
- Include the following examples: UserProfile, ConnectedPeopleList, and CardsToRemoveContainer.

The code below is an implementation of the Todo List component - it is a presentational component and is not able to filter todos. It only renders the list that is given to it, not refining or mutating it in any way.

```

1 // omitting imports
2
3 // extracting todos and onTodoClick from props
4 const TodoList = ({ todos, onTodoClick }) => (
5   <ul>
6     {todos.map((todo, index) => (
7       <Todo
8         key={index}
9         {...todo}
10        onClick={() => onTodoClick(index)}
11      />
12    ))}
13   </ul>
14 );

```

Then, as in the Todo List container example, it has access to the application store using Connect and maps all the relevant branches of the application state to the presentational component above. It decides the behaviour and filters the list of todos.

```

1 // Omitting imports
2
3 // handler to filter todos by selected filter
4 const getVisibleTodos = (todos, filter) => {
5   switch (filter) {
6     case 'SHOW_ALL':
7       return todos;
8     case 'SHOW_COMPLETED':
9       return todos.filter(t => t.completed);
10    case 'SHOW_ACTIVE':

```

```

11     return todos.filter(t => !t.completed);
12   }
13 };
14
15 // mapping the filtered todos to the presentational component.
16 const mapStateToProps = state => {
17   return {
18     todos: getVisibleTodos(state.todos, state.visibilityFilter)
19   };
20 };
21
22 // mapping the onTodoClick function that dispatches the toggleTodo
    action creator.
23 const mapDispatchToProps = dispatch => {
24   return {
25     onTodoClick: id => {
26       dispatch(toggleTodo(id));
27     };
28   };
29 };
30
31 // wrapping the presentational component
32 const VisibleTodoList = connect(
33   mapStateToProps,
34   mapDispatchToProps
35 )(TodoList);

```

### 3.3.5 Using Firebase with Redux

We will use the process of building a very simple and basic micro-blog application (written using React, Redux, and Firebase) as an example, where the user can post some text in one channel (or room) and which other users following that channel can read.

The first instinct is to subscribe to the data in the database, thereby building the listener which is connected straight to the view layer. This way, every new change coming through the wire will be reflected in the user immediately. However, this approach jeopardises the predictability of the application: It is not possible to handle the changes appropriately. The transformation of the state is out of the control of the application, and it will make the maintainability difficult when trying to operate and derive the data from several sources.

When architecting a real-time system, it is important to remember that the application should have predictability. Handling predictability using Redux brings the need to keep in mind that the store is the "source of truth" for the application. Therefore, when a page on the application first loads, it needs to request the data from the store.

To request data from the store, the view component needs to be connected to it.

```

1
2 // Omitting imports
3
4 class PostsList extends Component {
5   // When the component is about to be rendered on the screen, it
    dispatches an action to fetch the posts

```

```

6   componentWillMount() {
7     this.props.fetchPosts();
8   }
9
10  // each mapped post is rendered
11  renderPosts() {
12    return _.map(this.props.posts, (post, key) => <div> post </div>);
13  }
14
15  // rendering the list of posts
16  render() {
17    return (
18      <div>
19        <ul className="list-group">
20          {this.renderPosts()}
21        </ul>
22      </div>
23    );
24  }
25 }
26
27 // mapping the posts from the store as props to be rendered by the
   component
28 const mapStateToProps = (state) => {
29   const { items } = state.posts;
30   return { posts: items };
31 };
32
33 // mapping the dispatch function to fetch as props to be called by the
   component
34 const mapDispatchToProps = dispatch => ({
35   fetchPosts: () => dispatch(fetchPosts()),
36 });
37
38 // exporting the component while connecting it to the store using
   Connect
39 export default connect(mapStateToProps, mapDispatchToProps)(PostsList)
   ;

```

It is curious to note that to receive the posts, the method `fetchPosts()` is invoked only once in the lifetime of the component. This happens because when the action is dispatched, it opens the persistent connection and forwards the data coming from the server as payload to the store:

```

1  // Omitting imports
2
3  // creating Firebase reference
4  const Posts = new Firebase('https://gdg-posts.firebaseio.com/');
5
6  export function fetchPosts() {
7    return (dispatch) => {
8      // Opening a listener on the reference
9      Posts.on('value', (snapshot) => {
10        // Every time the listener is triggered, it dispatches an action
           with the posts
11        dispatch({
12          type: RECEIVED_POSTS,
13          payload: snapshot.val(),
14        });
15      });
16    };
17  }

```



```
16   };
17 }
```

Then the reducer manages the payload based on the action type and puts it into the application state:

```
1 // Omitting imports
2
3 export default function (state, action) {
4   switch (action.type) {
5     // Updating the store with the action payload
6     case RECEIVED_POSTS: {
7       const items = action.payload || {};
8       return { items };
9     }
10
11     default:
12       return state;
13   }
14 }
```

Since this branch (post list) of the store is mapped into the component, every new update coming from the server will cause a re-render of the list with the most up-to-date information. To summarise what is happening:

1. The component is connected to the store and, when rendered, calls the `fetchPosts()` method.
2. The `fetchPosts()` method opens a persistent connection with the database by subscribing to changes in a given reference.
3. Every time there is a change, it triggers a dispatch action with the updated data.
4. The action goes through the predictability layer (reducer) and gets derivated into the store.
5. The store passes the updated list as props, which causes a re-render in the component.

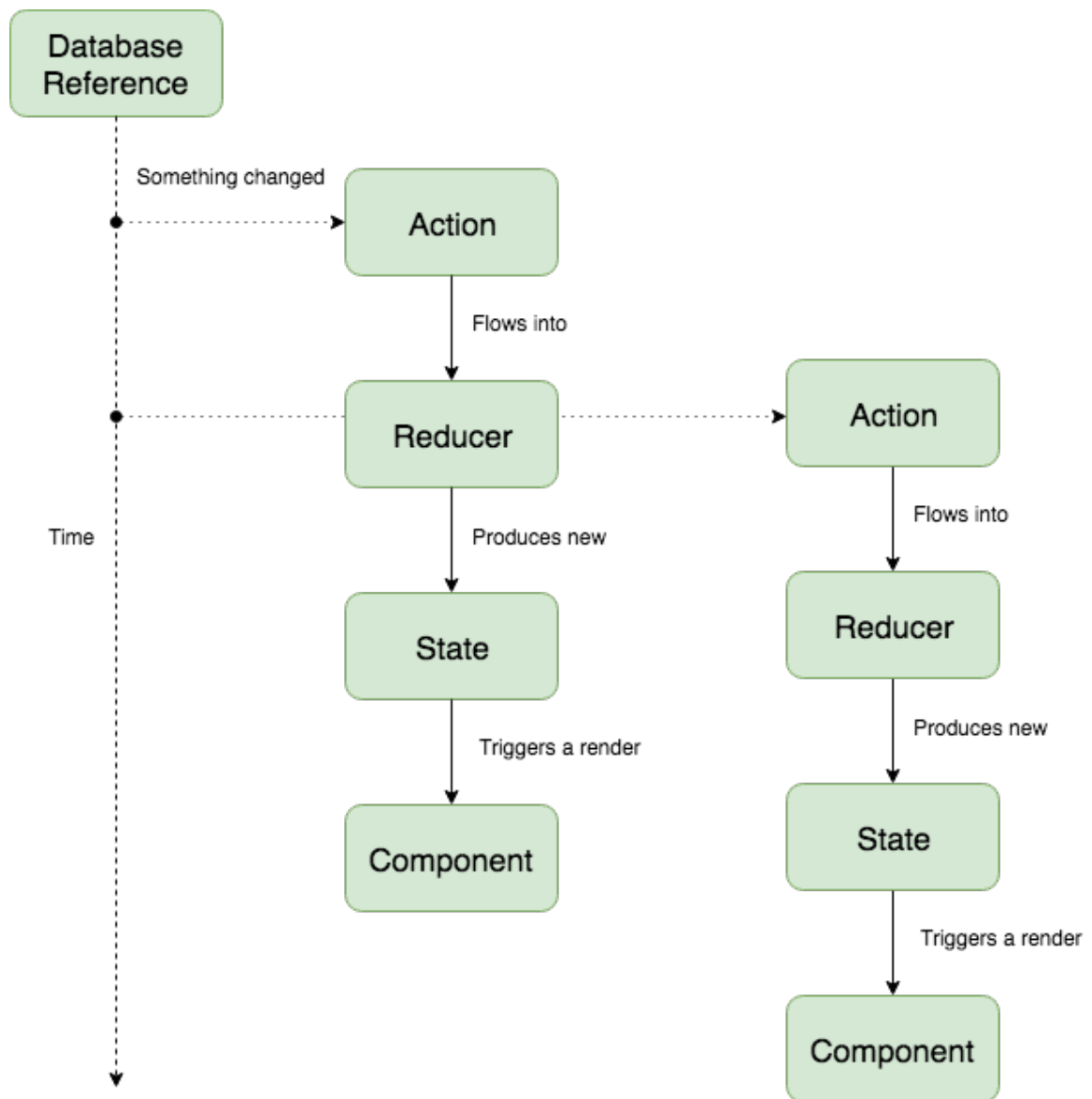


Figure 3.4: Update Event Sequence Diagram.

It is important to consider what approach to use when updating the UI.

## Updating UI

There are two main approaches to consider when handling UI updates, and they are Optimistically Update or Pessimistically Update.

Optimistic UI is a pattern used to simulate the outcome of an iteration and update the UI even before obtaining a response from the server. Once the response is received, the optimistic result is substituted with the actual result.

Optimistic UI provides a way to cause the UI to feel faster while ensuring that the data becomes consistent with the actual response when it arrives.

Example:

1. The user clicks a button.
2. The visual state of the button is changed into success mode instantly.
3. The call is sent to the server.
4. The response from the server is received by the application.
5. If the response is a success, we do not need to bother the user.
6. Only in the case of a failed request will the UI speak up.

Alternatively, Pessimistic UI is a pattern where the UI waits to update only after the server responds to the request.

Example:

1. The user clicks a button.
2. The button changes into a disabled state and a spinner is shown on the button to indicate that the system is working.
3. A call is sent to the server.
4. The response from the server is received by the application.
5. The visual state of the button and the page are updated according to the response status.

For non-critical transactions (feed actions, likes on social media, shares, and so on), it is recommended to use Optimistic UI. In the case of a failure, the user can be notified and the UI can be updated to reflect the state of the response. In a few situations, the updates might fail, and the user may or may not see the follow-up notification. However, this is generally acceptable as the action performed is not critical.

For critical transactions (e.g., bank transfers, airline check-ins, tax submissions), Pessimistic UI is recommended because a false result can be very consequential to the user.

Unidirectional flow has the advantage of making both flows easy to implement and easy to be re-factored into one another, thereby increasing maintainability.

If the UI is being Optimistically Updated:

1. The user interacts with the UI.
2. The success action is dispatched.
3. The application state changes and causes the UI to re-render into the success state.
4. The call is sent to the server.

5. The response from the server is received by the application.
6. If the response is a success, no further action is required.
7. In the case of a failure, the failure action is dispatched.
8. The application state changes and causes the UI to re-render into the failure state.

If the UI is being Pessimistically Updated:

1. The user interacts with the UI.
2. The loading action is dispatched.
3. The application state changes and causes the UI to re-render into the loading state.
4. The call is sent to the server.
5. The response from the server is received by the application.
6. If the response is a success, the success action is dispatched.
7. The application state changes and causes the UI to re-render into the success state.
8. In the case of a failure, the failure action is dispatched.
9. The application state changes and causes the UI to re-render into the failure state.

When handling the creation of posts in this application, we regarded the Pessimistic UI pattern. It is important to consider that the source of the events (responses from the server) comes from a persistent connection. There is a problem if the flow is divided in two. One flow internally updates the store, which immediately updates the view, while the other flow saves the changes on the server. This causes duplication of data and brings problems such as possible data discrepancies between the client and the server.

Additionally, due to the open listener, this approach is not necessary. When the value added is saved in the database, the listener is going to be triggered while passing the updated data, including the post that was just added and consequently, as seen, causing the re-rendering with the list of all the posts. It is also valid for updates done on different devices. Once a user creates a post, every connection on that channel will be updated and will follow the same flow as the user's own created post.

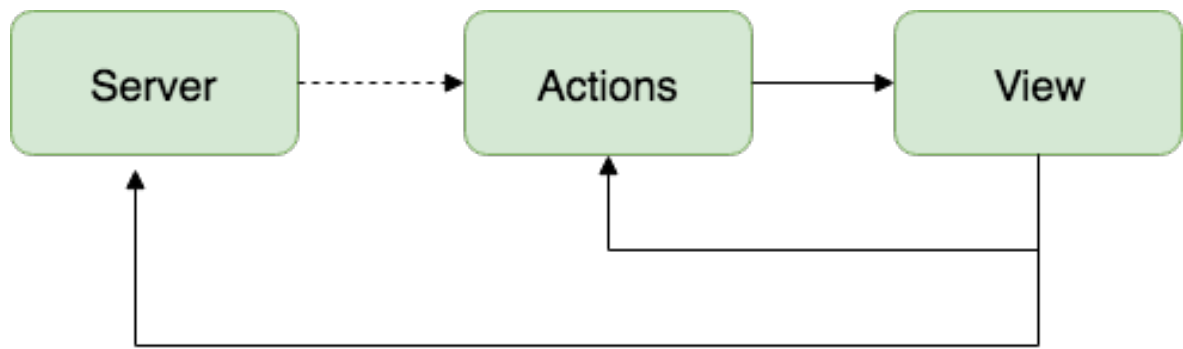


Figure 3.5: Duplicate Data.

This simple insert will trigger all the flow to take care of the update:

```

1 const Posts = new Firebase('https://gdg-posts.firebaseio.com/');
2
3 export function createPost(post) {
4   // Pushing the post
5   return () => Posts.push(post);
6   // return { type: CREATE_POST, payload: post }; < no need to
   // dispatch this action
7 }
  
```

The delete operation follows the same rationale as creating post flow; it will change the data and cause an update the same way, again. Removing it manually from the store is not necessary but will be handled by the unidirectional flow.

```

1 const Posts = new Firebase('https://gdg-posts.firebaseio.com/');
2
3 export function deletePost(key) {
4   // Pushing the post
5   return () => Posts.child(key).remove();
6   // return { type: DELETE_POST, payload: key }; <<< no need to
   // dispatch this action
7 }
  
```



Figure 3.6: Removing Data.

This graph shows the correct implementation for a maintainable circular flow.

# Chapter 4

## Discussion

### 4.1 Bi-directional vs. Unidirectional

To discuss the benefits of the unidirectional flow, it is necessary to first discuss the main ideas behind the bi-directional approach.

As previously explained, with the addition of more complex web applications, simply manipulating the DOM in traditional websites was not enough to keep the code scalable and maintainable while keeping track of the changes.

Furthermore, taking the concept of a web application seriously brought the discussion of front-end vs back-end architecture. This was when frameworks such as Backbone and Knockout started to become widespread (Section 2.1.1). They adopted practices that were already well-built for server-side architecture during a time when all the popular server-side frameworks (Rails, CakePHP, Spring, and Django) implemented at least some of the classic MVC model (also known as MV\* due to its variations).

#### 4.1.1 MVC

MVC is a pattern that has been used for various projects since the '70s, when it was introduced in Smalltalk-80. [Reenskaug, 1979]

MVC splits the application into three different parts:

- **Model** is the domain-specific representation of the data on which the system operates.
- **View** is partly responsible for rendering the UI elements and handling interaction with them.
- **Controller** processes and reacts to events, typically user actions, and handles changes on the model and the view.

The Figure 3.1 shows the data flow and control flow for the strict MVC pattern.

MVC also has some variations:

- **MVP** (Model-View-Presenter) pattern, where the view is passive and only responds to data emitted by the presenter, which comes from the model.

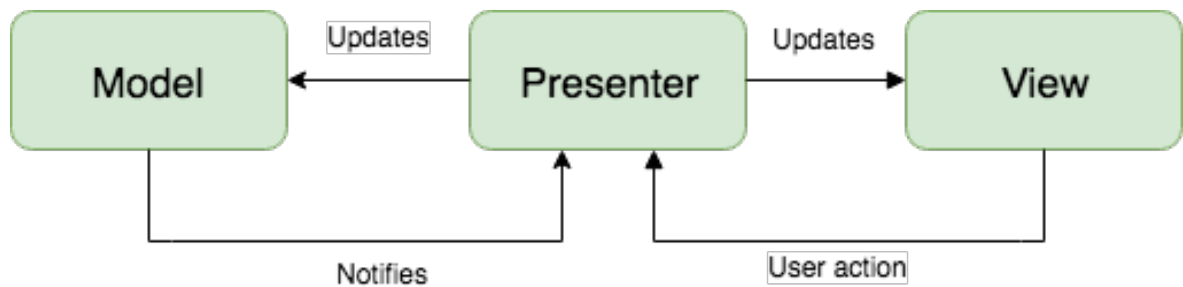


Figure 4.1: MVP Flow.

- **MVVM** (Model-View-ViewModel) pattern has a viewmodel that is responsible for abstracting all view inputs/outputs while implementing behaviour at the same time. It clarifies the testing of an application built with this pattern, as the viewmodel possesses the lion's share of data and behaviour, and it is separated from the view.

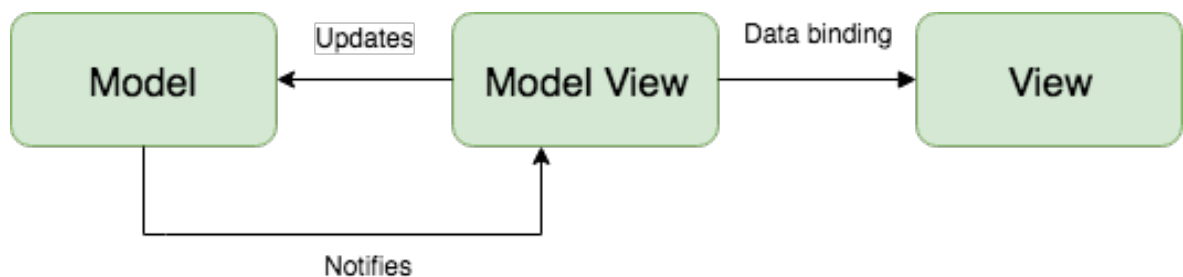


Figure 4.2: MVVM Flow.

As shown in the diagrams above, data can flow two ways. For instance, when a button in the view receives a press event, it submits the event to the controller, which processes it and updates the relevant model. The model, then, is updated and returns the result to the controller, so the controller updates the view. MVC also enforces a good separation of concerns: The separation of logical, representational, and visual layers makes it maintainable.

Below is a figure of how the view and the controller are co-operating on the server. There are only two contact points between them, both crossing the border between the client and the server.

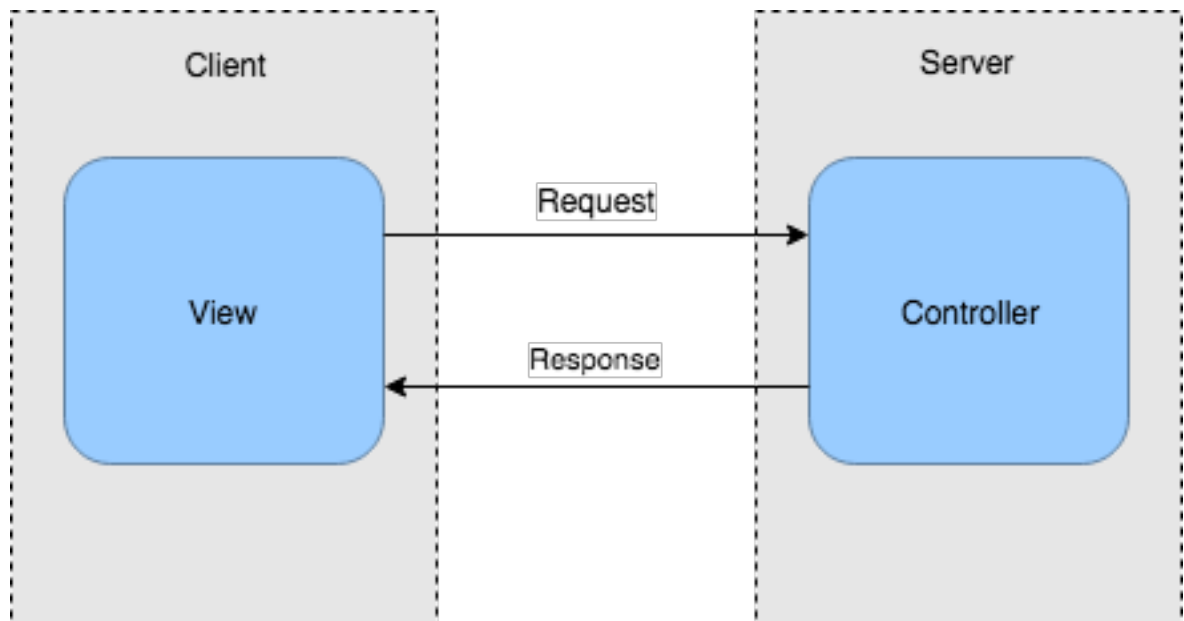


Figure 4.3: Server MVC.

The concept of MVC on the client side brings certain problems. Controllers are deeply coupled to the view.

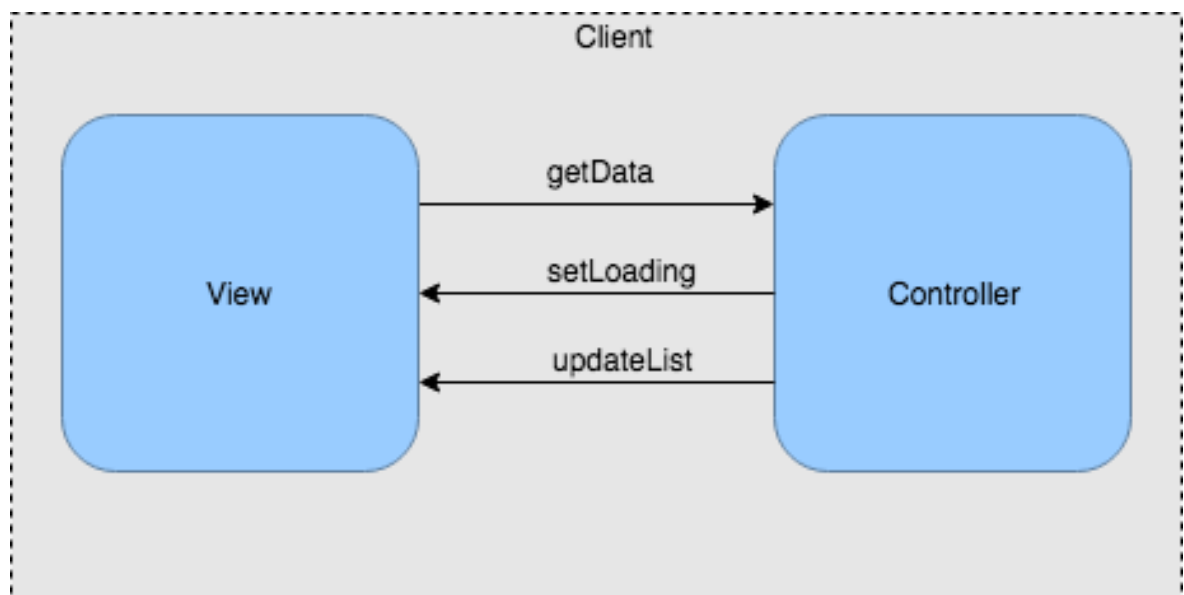


Figure 4.4: Client MVC.

Additionally, when considering the Single Responsibility Principle, this breaks the rules. The client controller code deals with both event handling and business logic, at a certain level. This complexity can explode when several models and their corresponding views are added to a system, as shown in the following model:



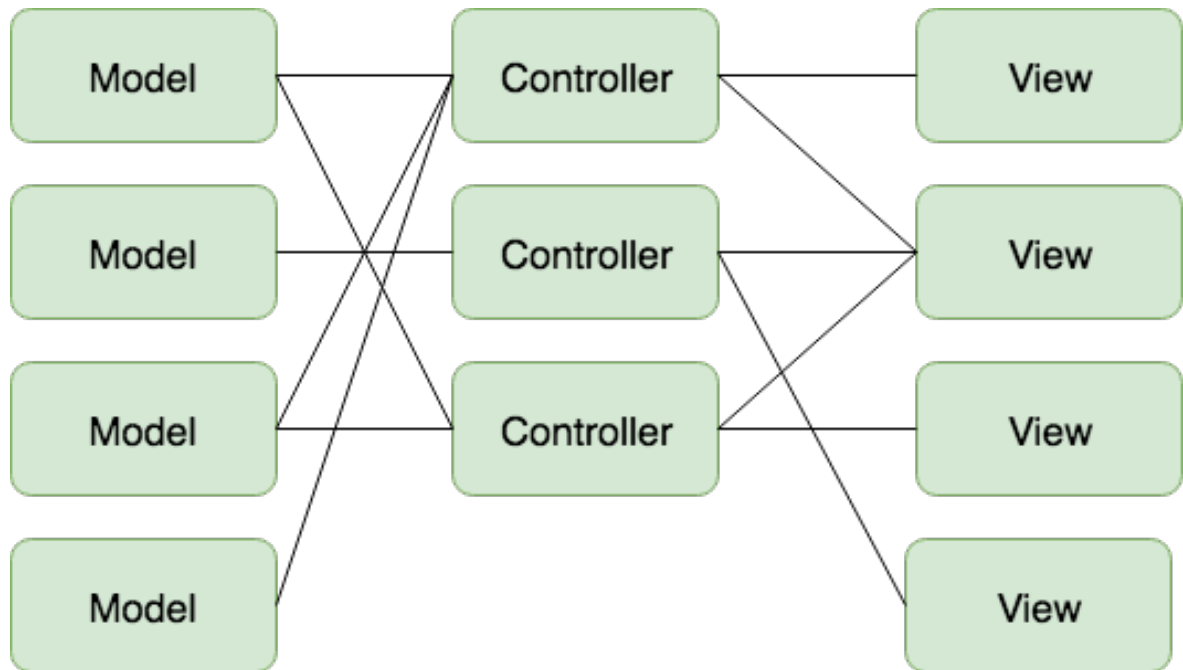


Figure 4.5: Scale Complexity of MVC.

The bi-directional communication can also loop back and have cascading effects across the codebase.

1. When introducing views that consume information from more than one model or models that provide data for more than one view, it is hard to reuse existing simple controllers to manage the new relationship. Thus, more controllers need to be introduced to handle this specific new connection, but often overlapping controllers diverge among them about the state of a model or a view.
2. Asynchronous network calls to retrieve data add the uncertainty of when the model will be changed or modified, such as a user changing the UI (section 3.3.5, Updating UI), while a callback from an asynchronous call comes back. Then, we will have a “non-deterministic” status of the UI.
3. The change state/model has another layer of complexity, which is the mutation. How should the mutation be recognised? If the application is collaborative in nature (for instance, Google Docs), where groups of data change are happening in real-time, it will be necessary to build extra tools to help recognise the mutation.
4. There’s no way to do ”undo” (travel back in time) easily without adding extra code.
5. Most of the controller functions are optimistic. To give the impression of real-time, the data changes are sent to the stores and the backend at the same time so that the UI remains snappy and does not need to wait for the backend to have stored all the data. (section 3.3.5, Updating UI)

The above can be summarised as: “There is no single source of truth for the applications at any given time.” This creates an unpredictable UI, and predictability is a core fundamental of a real-time system.

The unidirectional flow solves several problems:

1. Pure functions make the output deterministic and the UI predictable.
2. Time Travelling: Since the state changes through actions that are pure JavaScript objects, this means that it can re-create the state from scratch if the system keeps a log of those actions. Hence, the time-travelling problem is solved.
3. Logging actions: It establishes who modifies the state, and when.
4. Collaborative programs such as Google Docs can be easily achieved by sending actions on the wire and re-creating them there.
5. Easy debugging: By logging all actions on production, we can re-create the whole situation.
6. Unit testing is easier because unidirectional flow tests pure functions for UI and state mutation.

### 4.1.2 Comparison

To highlight the above points, a second application using MVC was developed to compare against the first application using unidirectional data flow built-in [chapter 3](#).

Both applications use React to create the views, and Firebase to keep the persistent connection with the database.

They have the same features and user interface (they are identical from an external point-of-view). They consist of a page where any user can write a short piece of text; the text is visible to every other user logged into the system. All the texts of each user form a "timeline" ordered by date-time. It can be conceptually broken down into three main features:

1. Create and publish posts to a timeline that other users logged in the system would be able to read and vice versa. The perception of the timeline should be real-time; as soon as a user publishes a post, it should be rendered on other users' screens with no need for a refresh.
2. Users can friend each other. A user can see if their friends are online in a dedicated sidebar.
3. Users can filter the timeline to see only the posts of selected users using a filter dialogue.

Additionally, the second application was prototyped to give not only insight into theoretical problems in achieving a real-time system using MVC, but also to further analyse and compare the code's complexity and the performance of both applications.

Since the source code for both applications is very different from each other, as they use different architectures, comparing the code side by side is not a logical choice.

A set of different exercises and experiments were performed to compare the different codebases. They are:

1. Model structure, the entities, and relationships between them: This comparison was executed in three stages to measure scalability. The first stage will include only the first feature of the application (create and publish posts to a timeline), the second stage will include the second feature (users can friend each other), and finally, the third stage will also cover the last main feature (users can filter the timeline).
2. Maintainability index. This comparison will measure how maintainable the codebase is, as in the first comparison. To add an element of scalability, it will include three stages where each stage adds a new feature to the application.
3. Runtime code coverage analysis. To measure how many lines of code and commands are executed for each essential interaction of creating a post, filtering the timeline, and others.

When considering the proposed timeline application, all the mentioned exercises and experiments performed produce static results - this means the results will not vary depending on the environment or the tool used to measure them.

## First Comparison: Models

As explained above, this comparison was made in three separate stages, with a new feature added in each stage.

### First Stage

The first stage consists of the timeline feature.

#### *Unidirectional*

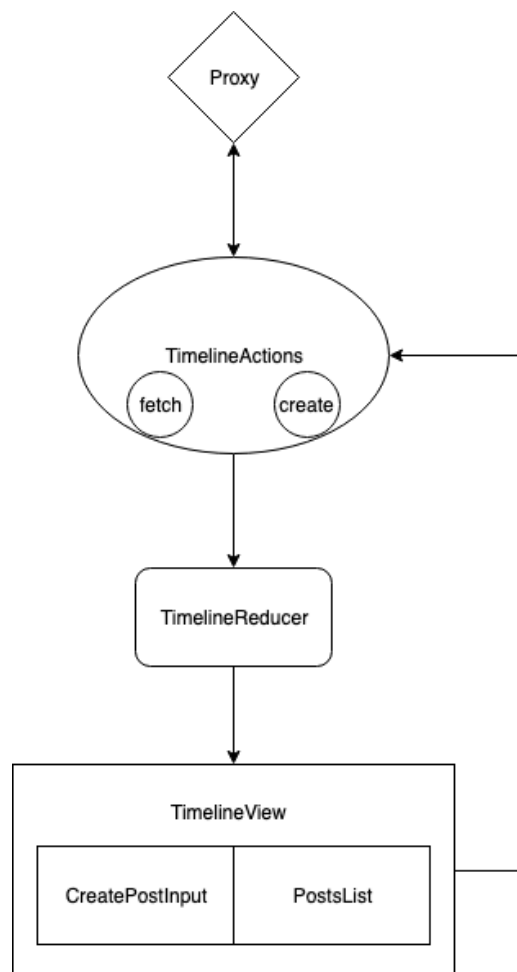


Figure 4.6: Model of Unidirectional Data Flow for First Stage.

Both the view and the proxy can dispatch actions; therefore, any new event was reduced to a state and was eventually rendered on the user screen.

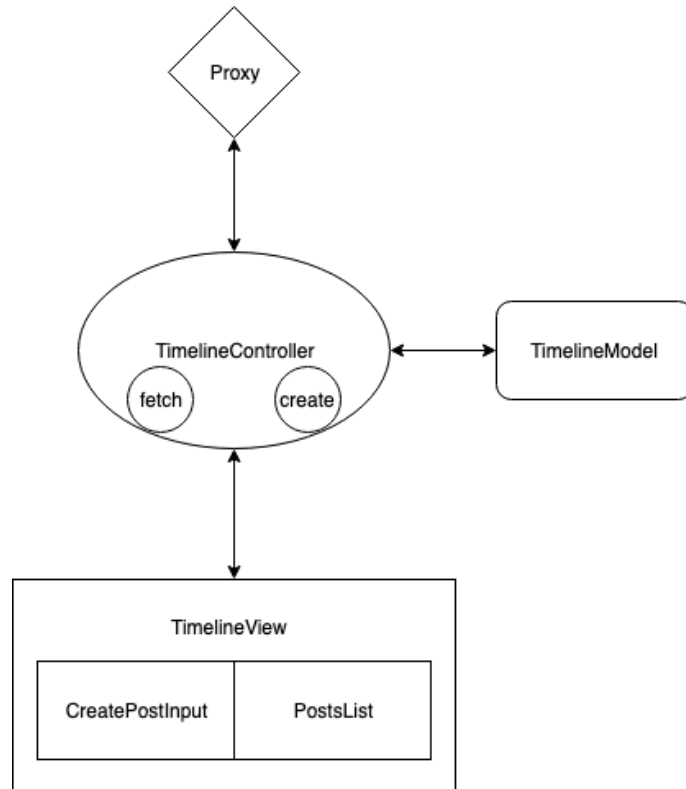


Figure 4.7: Model of Bi-directional Data Flow for First Stage.

The MVC design here is quite simple: The controller takes care of fetching data, then instantiates the models that transform the raw data from the API. The result is passed back to the UI, and the controller is already quite busy. An alternative solution would be to implement model binding, but in this case, the controller would not be accomplishing its primary task: controlling. This is particularly worrisome when addressing real-time, as there would be no simple way to manipulate merges and implement prioritization when the model is mutated from the UI as a new update is fetched from the API. Thus, the state becomes unpredictable.

## Second Stage

In the second stage, besides the timeline feature, the user feed feature was added.

### *Unidirectional*

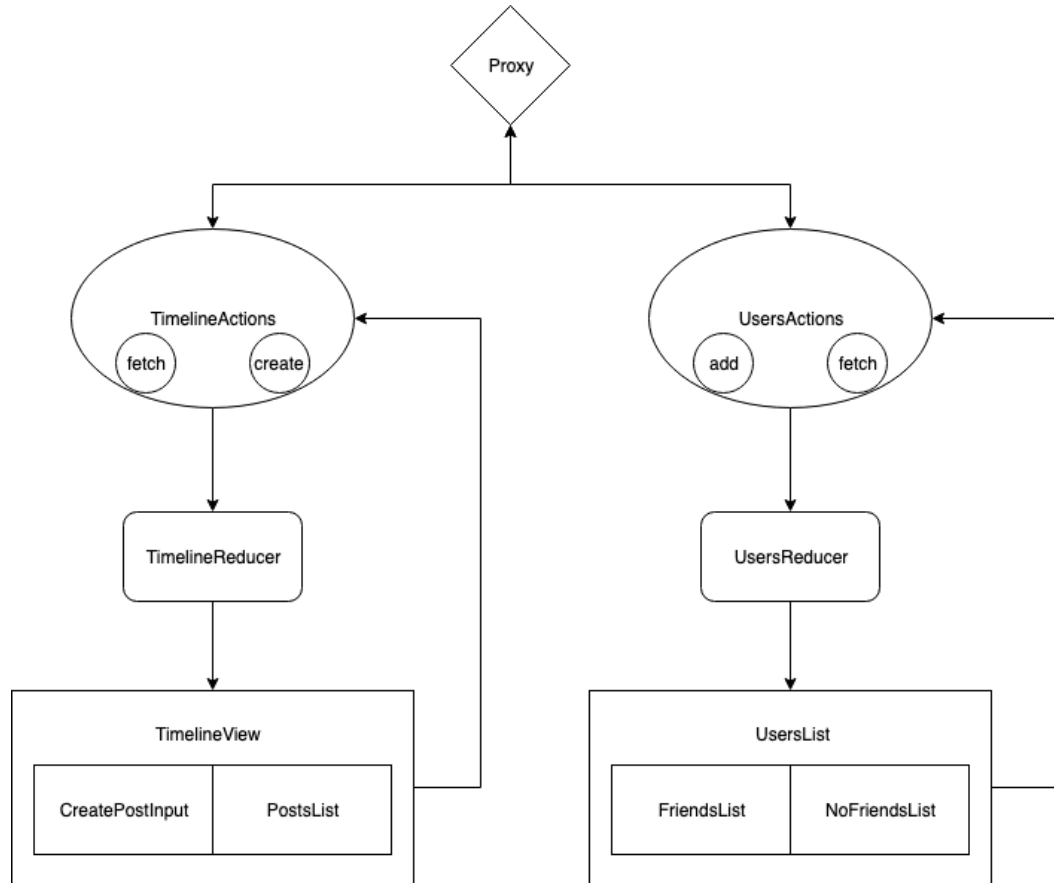


Figure 4.8: Model of Unidirectional Data Flow for Second Stage.

As can be seen, the second feature is logically isolated from the first feature. Although they share space on the DOM, there is no real rational interaction between them.

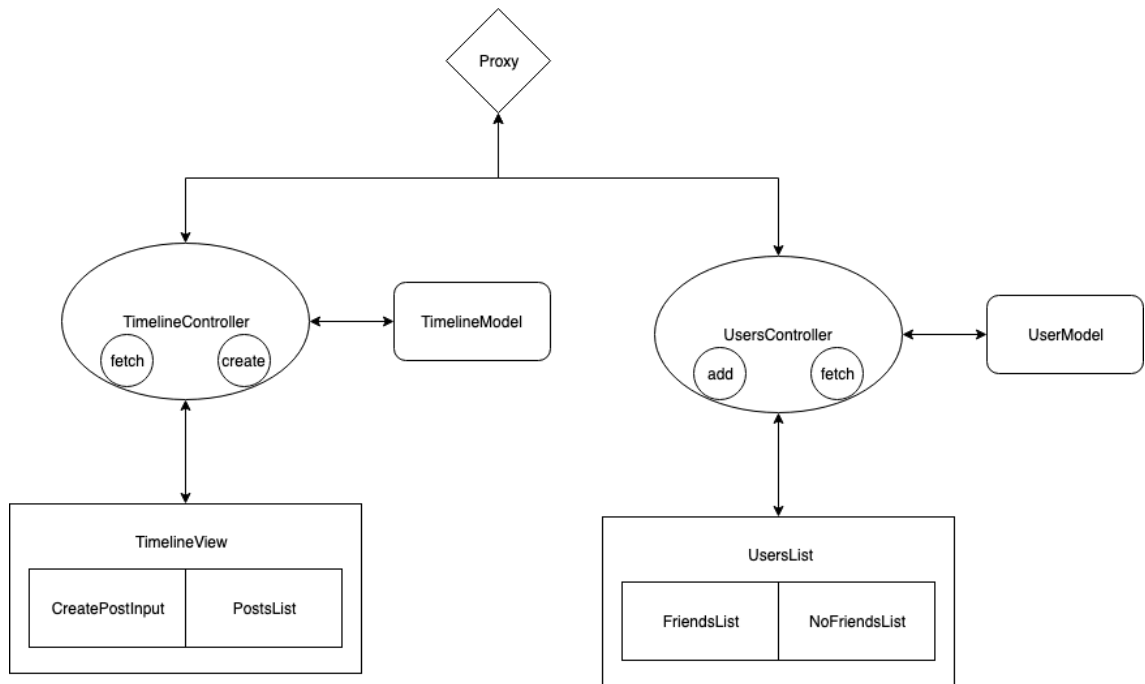


Figure 4.9: Model of Bi-directional Data Flow for Second Stage.

As in the unidirectional approach, the features are decoupled from one another. However, the addition of a new MVC entity adds to the already existing problems related to managing real-time features.

### Third Stage

The third stage brings the feature of filtering the timeline based on selected users.

#### *Unidirectional*

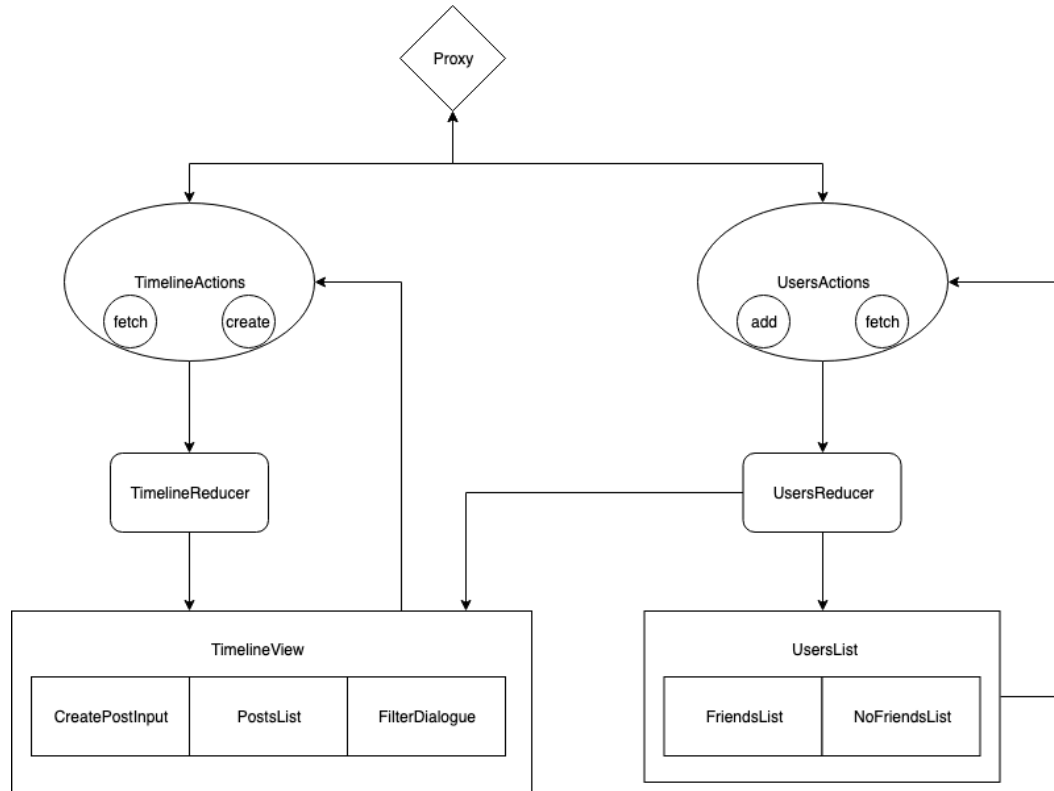


Figure 4.10: Model of Unidirectional Data Flow for Third Stage.

To deal with the relationship between the two, now linked, features, a new view component is introduced. Its primary responsibility is to derive the data from the existing reducers and filter out now irrelevant posts based on its internal state, which contains the filter options.



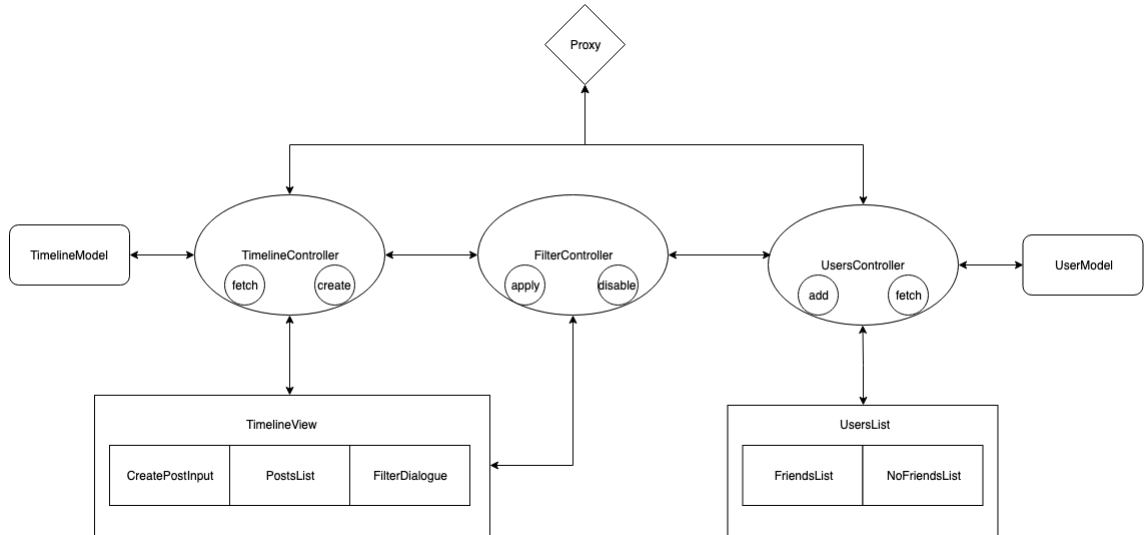


Figure 4.11: Model of Bi-directional Data Flow for Third Stage.

A new controller was added with an increased number of connections between controllers. The option to add a new controller was preferred over the second alternative: connecting the view with the two controllers. The main reason behind this choice is that the view would also be very coupled with the UsersController, as this controller would have to be extended to allow the view to get the list of users.

## Second Comparison: Complexity and Maintainability

Complexity is the characteristic of consisting of several interrelated elements. When a software consists of multiple interrelated elements, it is harder to reason within it; often this means that the software is more prone to the introduction of bugs compared to a software that is simplistic.

Every problem space holds some level of inherent complexity, which is shared by all potential solutions. Nevertheless, programmers can decrease the complexity of solutions by narrowing the interrelatedness of their components. This is commonly referred to as choosing cohesion over coupling and forms the basis on which premises such as the Single Responsibility Principle are established.

According to the IEEE standard definition [IEEE, 1990], software maintainability is the facility inbuilt within a given software system, or a component that can be altered to correct faults, improve performance and other properties, or adapt to a modified environment. The maintainability is so important that it is one of the main characteristics that describe the model of software quality on the ISO 9126 [ISO/IEC, 2001].

## Maintainability Index

It is easy to see the relationship between code complexity and code maintainability, thereby analysing the code's cyclomatic complexity through lines of code to calculate the Maintainability Index. The Maintainability Index is an empirical equation. It was

constructed by a model based on observation and adaptation. [Coleman et al., 1994]  
The metric originally was calculated as follows:

$$\begin{aligned} MaintainabilityIndex &= 171 - 5.2 * \ln(HalsteadVolume) \\ &\quad - 0.23 * (CyclomaticComplexity) \\ &\quad - 16.2 * \ln(LinesOfCode) \end{aligned}$$

Microsoft's Visual Studio [Naboulsi, 2011] calibrated the calculation to lie between 0 and 100:

$$\begin{aligned} MaintainabilityIndex &= 171 - 5.2 * \ln(HalsteadVolume) \\ &\quad - 0.23 * (CyclomaticComplexity) \\ &\quad - 16.2 * \ln(LinesOfCode) \end{aligned}$$

$$CalibratedMaintainabilityIndex = MAX(0, ((MaintainabilityIndex * 100)/171))$$

The higher the value, the better the maintainability. Values between 20 and 100 indicate that the code has good maintainability, values between 10 and 19 indicate that the code has moderate maintainability, and values between 0 and 9 indicate low maintainability.

All comparisons made from this point on will use Microsoft's Visual Studio-calibrated version of the Maintainability Index.

## First Stage

This is the comparison between the two codebases explained above: Each application has a page where any user can write a short text, and the text is visible to every other user logged into the system. All the texts of each user form a "timeline" ordered by date-time.

	Unidirectional	MVC (Bidirectional)
Total number of lines	<b>188</b>	<b>216</b>
Maintainability Index	<b>87.14</b>	<b>84.23</b>

It is observed that the differences are small, both in the number of lines and in the Maintainability Index.

However, one of the main points against MVC is how the complexity of the system grows exponentially every time a new feature is added, making the code fragile and unpredictable.

## Second Stage

This comparison was made after adding a new feature to the application: allowing users to follow other users. Besides the blog timeline, there is now a section with all the followed users along with an indicator of which users are currently online.

	Unidirectional	MVC (Bidirectional)
Total number of lines	<b>267</b>	<b>404</b>
Maintainability Index	<b>86.96</b>	<b>81.27</b>

There was growth in the number of lines in both applications, although the growth was higher on the MVC application.

The Maintainability Index was largely the same for the unidirectional application and a slight decrease was observed in the MVC application.

One can still argue that by adding more features, those numbers would stabilise and be very similar as the application grows.

## Third Stage

A third and final comparison was made after adding one more feature: a filter. Users can now select one or more users and see only their posts on the timeline.

	Unidirectional	MVC (Bidirectional)
Total number of lines	<b>337</b>	<b>587</b>
Maintainability Index	<b>86.70</b>	<b>76.45</b>

The number of lines keeps growing on the MVC application since this feature requires one more controller for filtered posts and another one to select users. The unidirectional code did not grow much since it allows better code reusability.

The Maintainability Index remained stable for the unidirectional application and had another slight decrease in the MVC application.

## Overall Maintainability Index Comparison

### Total number of lines

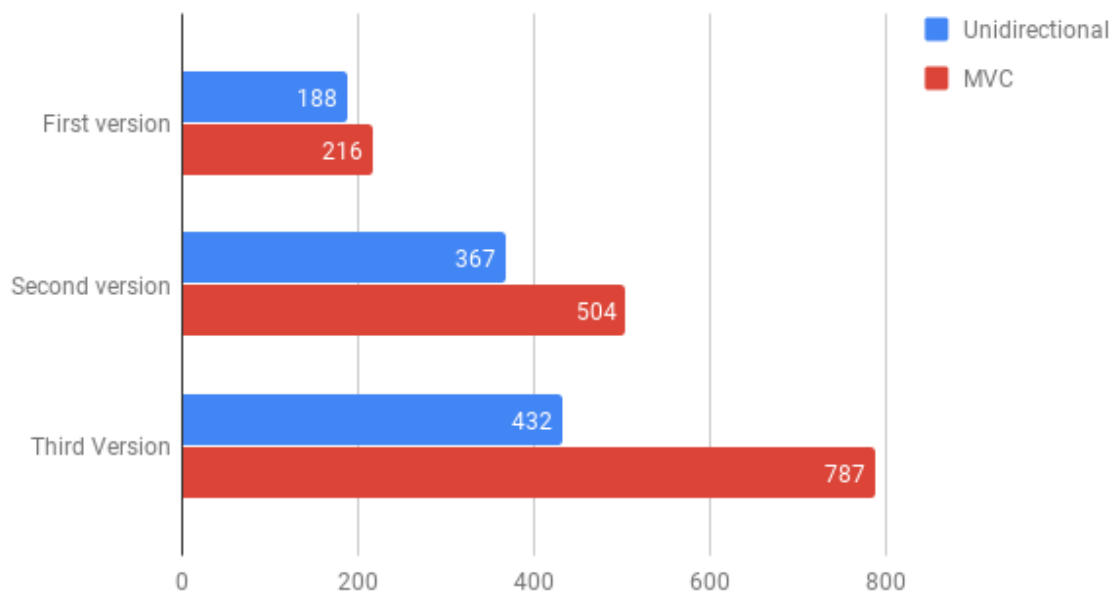


Figure 4.12: Total Number of Lines Comparison.

### Maintainability Index

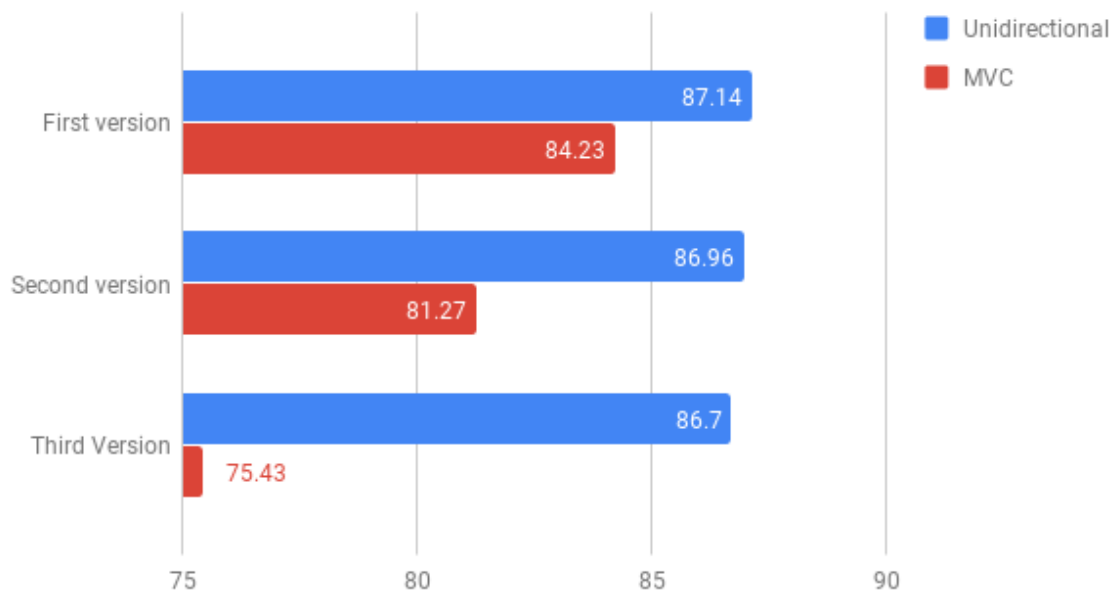


Figure 4.13: Maintainability Index Comparison.

The graph highlights the benefits of unidirectional flow with regard to maintainability as the application grows and becomes more feature-rich. It requires less code when a new feature is added due to less boilerplate code. Additionally, reusability is improved.

It is more maintainable because it has predictable behaviour as well as the benefits cited in the previous sections.

### Third Comparison: Runtime Code Coverage

Measuring code coverage is useful for determining which sections of the code are executed for a particular method, process, or action. This is commonly used in unit testing to measure how much of the code was executed after running a given test suite in order to conclude what percentage of the code is tested. However, it can also be used outside of the scope of testing. The Google Chrome browser, for example, supports exposing this information by providing embedded tools developers can use to analyse. [Gruber, 2017](#)

It is important to note that the tool Chrome provides runs against the JavaScript code transpiled to ES5 specification—JavaScript code is commonly written in its latest specification, but it is not guaranteed that a user's browsers are up-to-date and will be ready to execute it. For this reason, developers transpile the code to an earlier specification of JavaScript, and the widely supported ES5 is generally the target.

The code coverage data is broken down into:

- Branches: has each path of a logic control structure been executed?
- Statements: the percentage of statements executed. Some lines of code have more than one statement.
- Functions: has all the functions and methods been executed?

For this comparison, only the number of statements will be analysed, as the code can be so distinct between the codebases that analysing functions and branches would not be informative.

Also, this comparison will not take into consideration third-party dependencies libraries. It will only check the coverage on the code written by the author. This boundary was decided as the main two libraries used by the application are shared between them. On one side, React renders data; at the other, Firebase SDK retrieves and saves data.

The numbers were collected using Google Chrome's Coverage Report Development tool on four separated operations:

1. Fetching posts: After the application is loaded and painted in the DOM and the fetch post function is triggered, how many statements are executed to fetch the posts, parse them, and finally, paint them in the screen?
2. Creating a post: After the application is loaded and painted in the DOM and the create post function is triggered, how many statements are executed to build the request using the text input? This flow also includes the statements of parsing the response and adding it to the state to trigger the addition of the new post on the rendered list in order to highlight the benefits of the unidirectional flow.

3. Adding a friend: Similar to creating a post, this counts the statement only after the action is triggered and includes the journey back to updating the rendered element.
4. Filtering timeline: The number of statements executed to include the filter parameters on the state and update the timeline accordingly.

The graph below displays the results.

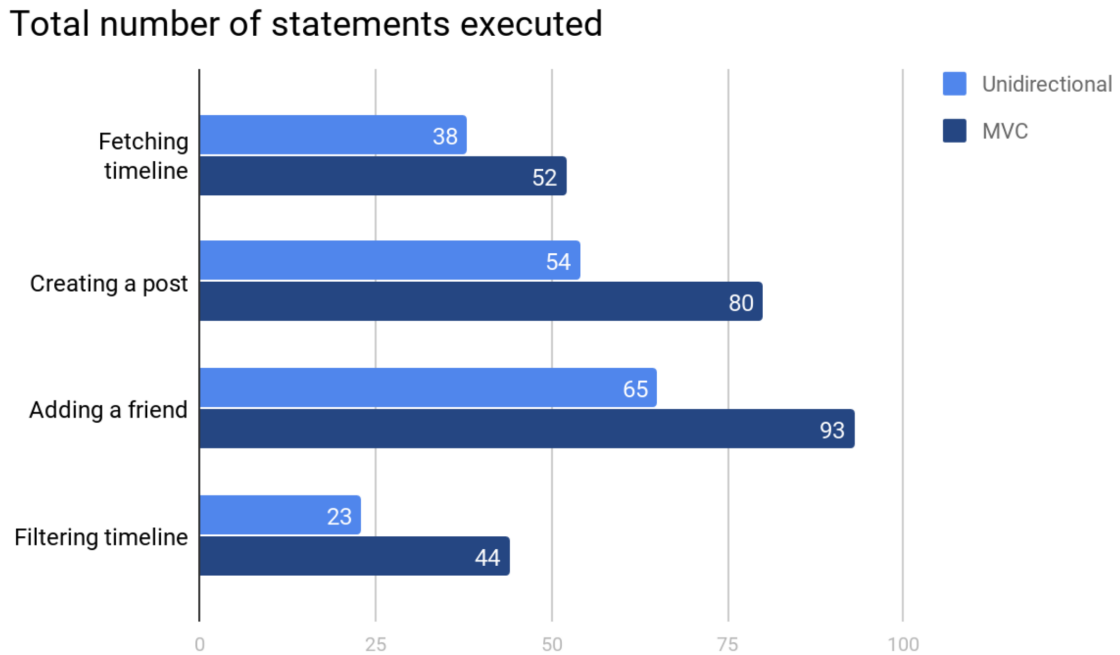


Figure 4.14: Total Number of Statements Executed Comparison.

Again, the smaller amount of statements executed draws attention to the benefits of using unidirectional flow; this is highlighted on real-time operations when the application is updated with the result of a user action.

### 4.1.3 Real-Time Engine

As part of the research, the author developed an open-source library called Real-Time Engine. [\[Ferreira, 2018\]](#)

The aim is to address the initial difficulties in understanding the nuances around the unidirectional architecture as well as to decrease the time spent in adapting and adjusting to the unidirectional flow mindset (learning curve).

The Real-Time Engine is a state manager focused on handling real-time features for React applications.

With a minimum amount of setup, it controls the unidirectional flow and provides the updated values from any source as props.

It stores the state in a tree-like structure where each "domain" is a branch.

It provides two functions: *registerPayloadHandler* and *withRealTime*.

## registerPayloadHandler

It is the function used to register a payload handler. It accepts one parameter: domain, which is the branch in the state where the data will be stored. It returns a function to be set as the callback on the event stream. Every time this function is called, it stores the payload on the domain provided.

Example with Firebase:

```
1 // Imports the registerPayloadHandler from the real-time-engine-  
  package.  
2 import { registerPayloadHandler } from "real-time-engine";  
3  
4 // Creating a handler for the "posts" domain.  
5 const payloadHandler = registerPayloadHandler('posts');  
6  
7 // Every event that comes through is passed to the handler to be  
  stored.  
8 database.ref('/').on('value', (snapshot) => {  
9   payloadHandler(snapshot.val());  
10 });
```

## withRealTime

It expects one parameter: domain, which is the branch in the state where the data will be stored. It returns a high-order component that wraps a component and map as props the updated values on the domain branch of the state.

Example:

```
1 // Imports the withRealTime from the real-time-engine-package.  
2 import { withRealTime } from "real-time-engine";  
3  
4 // Creating a HOC connected with the domain "posts."  
5 const RealTimePostsHOC = withRealTime('posts');  
6  
7 // Wrapping the PostsList Component with the RealTimePostsHOC.  
8 const RealTimePostsList = RealTimePostsHOC(PostsList);
```

or shorter version:

```
1 import { withRealTime } from "real-time-engine";  
2  
3 const RealTimePostsList = withRealTime('posts')(PostsList);
```

The most important aspect of the Real-Time Engine is to encapsulate the control of the unidirectional flow. Consequently, the rest of the application is decoupled from the specifics of the implementation. Additionally, it also increases composability, since different components can be wrapped in the same domain.

```

1 import { withRealTime } from "real-time-engine";
2
3 const RealTimePostsHOC = withRealTime('posts');
4
5 const RealTimePostsList = RealTimePostsHOC(PostsList);
6 const RealTimePostTitlesList = RealTimePostsHOC(PostTitlesList);

```

Alternatively, however, one component can be listening to more than one domain:

```

1 import { withRealTime } from "real-time-engine";
2
3 const RealTimePostsHOC = withRealTime('posts');
4 const RealTimePostsAndUsersHOC = RealTimePostsHOC('users');
5
6 const RealTimePostsAndUsersList = RealTimePostsAndUsersHOC(
  PostsAndUsersList);

```

Moreover, since the function returned from *registerPayloadHandler* has the new information injected on it, it is easier to operate with any type of Real-Time Database or a WebSocket object, again increasing reusability.

Example with WebSocket:

```

1 import { registerPayloadHandler } from "real-time-engine";
2
3 const payloadHandler = registerPayloadHandler('posts');
4 const exampleSocket = new WebSocket("ws://www.example.com/server", "
  protocol");
5
6 exampleSocket.onmessage((event) => {
7   payloadHandler(event.data);
8 });

```

Generally, it can be said that the implementation of the unidirectional flow is difficult to comprehend and achieve in certain cases. This library's main aim is to abstract the complexity away by offering a simple wrapper.



# Chapter 5

## Conclusion

The current use of client-side processing requires better structure as the applications grow more complex. This increase in web-application complexity has contributed to the rise of frameworks, as they are meant to keep the codebase reasonable and scalable. Among the most used frameworks are Backbone and Angular. Similar to many other frameworks, they use the MVC pattern (and its variations) to administer the data flow.

The MVC pattern and its variations work well and deliver what is expected: good separation of concerns, which makes the codebase more maintainable, reasonable, and scalable.

However, inserting a new prerequisite changes this behaviour: real-time functionalities.

Real-time functionality became common after the WebSocket protocol made it relatively easier to implement. MVC's answer to update the view immediately (real-time) after it is updated somewhere is known as "model binding." Model binding tends to develop quickly into a web of connected views and models as the application grows.

This is where the unidirectional flow brings considerable improvement. By using a state manager, the single source of truth is the state store, and all the data in the application is derived from it. Any part of the application can subscribe to the piece of the store that is relevant to it and get notified when any change is made on that piece.

The unidirectional flow also brings benefits to making the application predictable. Since every event is processed and mutated before being saved on the store, the application has control over how it will behave.

In two-way data binding (bi-directional flow), the view is updated when the state changes and vice versa. For instance, if there is a change in the model, the view will automatically reflect the changes; moreover, a given input field in the view layer can change the model. It leads to cascading updates as changing one model may trigger further updates. Since both controller and view can mutate the state, the data flow can become unpredictable.

On the other hand, since the flow of data is unidirectional, there is only one way through which the view can be updated. Consequently, this provides a high degree of confidence that the view is a derivation of the data in the store and, hence, is

deterministic: The state in unidirectional applications is represented using immutable data structures. Immutability stops views from polluting state, allows easy observation of state changes, and provides for performance optimisations by caching views.

Since the state is immutable, views become pure functions. Given a specific state as input, the output is guaranteed to be the same for every function call. One of the more significant benefits of this is easy time travel and the option to undo. For each change, a state can be saved, and the entire application can be rendered at any given point by rewinding the state.

Since applications utilising unidirectional flow are less complex and have less code produced for each relation between the controller, model, and view, they tend to implement the same features of an application using MVC with less code and with a greater Maintainability Index.

However, the implementation of this architecture can become more involved compared to bi-directional data flow architecture due to its further control over how data flows in the application.

## 5.1 Opinions

### 5.1.1 State Managers

Since Facebook created Flux [Facebook, 2018a] following their experiences with MVC to manage their primary application [Avram, 2014], an increasing number of state managers have been and continue to be introduced. Many are a direct fork implementation of Flux: Redux [Abramov, 2018], Reflux [Reflux, 2018], Alt [Alt, 2018], and others. Additionally, others follow a different implementation but have the same base goal: MobX [MobX, 2018], Relay [Facebook, 2018b], and others.

They will continue to be created to perform better and become easier to maintain.

### 5.1.2 Helpers

Furthermore, packages and libraries will be introduced to help administer the unidirectional flow; they can be classified as tools that abstract the unidirectional flow and implement it as a third party library, like the Real-Time Engine [4.1.3]. Also, they could be classified as helpers that not directly implement the execution of the unidirectional flow, but, per example, enforcing that the flow will not move in the wrong direction or alarming when the state is being mutated directly somewhat without dispatching an action.

The ecosystem around the unidirectional flow will continue to grow as implementation details are automated and abstracted by tools.

### **5.1.3 Building the View**

Of course, React is a library that has, as its core concept, the unidirectional flow, and it is not the only one. Vue.js also enforces one-way data flow between the components.

Going forward, an increasing number of libraries that will build views will have unidirectional flow as their core concept.

### **5.1.4 Support**

The learning curve of developing unidirectional flow applications can be steep. However, as more developers deal with it, the trend is to have an increasing number of applications using it built using different types of libraries and frameworks.

## **5.2 Recommendations**

### **5.2.1 More Comparisons**

To obtain more detailed results, it would be worth performing more comparisons using other alternatives to unidirectional flow, while also building groups of comparable applications for examination.

### **5.2.2 Server-side Unidirectional Flow**

Event-driven architecture is a server-side architecture pattern that promotes the production, detection, consumption of, and reaction to events. An event can be described as “a change in the state.”

It sounds very similar to the client-side state manager and the real-time architecture.

One possible next step for this research could be the implementation and abstraction of unidirectional flow on the server side for services using event-driven architecture.

### **5.2.3 Convert Bi-directional to Unidirectional Flow**

Follow and document step by step a given application being re-factored from using the bi-directional flow to using the unidirectional flow.

Moreover, is it possible to automate this process?

## **5.3 Unidirectional Flow is the Natural Step to Handle Real-Time Events.**

The unidirectional flow emerges naturally from the need to support real-time events as they are parsed and flow into the actions. Then, the actions, in a predictable manner, mutate the state. The new mutated state is derived by the view through the use of mapping functions, so views are decoupled from the rest of the application, which increases maintainability.

# Bibliography

- [Abramov, 2018] Abramov, D. (2018). Redux documentation. Accessed 23th October 2017.
- [Alt, 2018] Alt (2018). *Alt Docs*. Accessed 13th June 2019.
- [Audsley, 1993] Audsley, N. C. (1993). *Flexible Scheduling of Hard Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, UK.
- [Avram, 2014] Avram, A. (2014). Facebook: Mvc does not scale, use flux instead.
- [Buttazzo, 2011] Buttazzo, G. C. (2011). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition.
- [Caspers, 2017] Caspers, M. K. (2017). React and redux. In *[2017] Rich Internet Applications w/HTML and Javascript*, pages 11–14.
- [Chaniotis et al., 2015] Chaniotis, I. K., Kyriakou, K.-I. D., and Tselikas, N. D. (2015). Is node.js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97(10):1023–1044.
- [Christensson, 2008] Christensson, P. (2008). Application definition.
- [Coleman et al., 1994] Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49.
- [Davis et al., 2016] Davis, R. I., Cucu-Grosjean, L., Bertogna, M., and Burns, A. (2016). A review of priority assignment in real-time systems. *J. Syst. Archit.*, 65(C):64–82.
- [Deitel and Deitel, 2011] Deitel, H. and Deitel, A. (2011). *Internet and World Wide Web How To Program*. Prentice Hall Press, Upper Saddle River, NJ, USA, 5th edition.
- [Dhote and Sarate, 2013] Dhote, M. R. and Sarate, G. G. (2013). Performance testing complexity analysis on ajax-based web applications. *IEEE Software*, 30(6):70–74.
- [Douglas C. Schmidt and Natarajan, 2004] Douglas C. Schmidt, A. G. and Natarajan, B. (2004). Frameworks: Why they are important and how to apply them effectively. *ACM Queue magazine*.
- [Dulimarta, 2017] Dulimarta, H. (2017). Ezpoll: a progressive implementation of cloud-based polling systems. In *[2017] International Conference on Grid, Cloud, and Cluster Computing*, pages 35–41.

- [Eden and Kazman, 2003] Eden, A. H. and Kazman, R. (2003). Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 149–159, Washington, DC, USA. IEEE Computer Society.
- [Enache, 2017] Enache, M. C. (2017). Web application frameworks. In *[2015] Annals of the University Dunarea de Jos of Galati: Fascicle: XVII*, pages 82–86.
- [Facebook, 2018a] Facebook (2018a). *Flux Docs*. Accessed 20th January 2018.
- [Facebook, 2018b] Facebook (2018b). *Relay Docs*. Accessed 13th June 2019.
- [Farines et al., 2000] Farines, J.-M., da Silva Fraga, J., and de Oliveira, R. S. (2000). *Sistemas de Tempo Real*, pages 1–20. Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina.
- [Fedosejev, 2015] Fedosejev, A. (2015). *React.js Essentials*. Packt Publishing, 1st edition.
- [Ferreira, 2018] Ferreira, E. P. A. (2018). Real-time engine.
- [Fette and Melnikov, 2011] Fette, I. and Melnikov, A. (2011). *The WebSocket Protocol*.
- [Firebase, 2018] Firebase (2018). Firebase realtime database. Accessed 10th August 2017.
- [G. Shin and Ramanathan, 1994] G. Shin, K. and Ramanathan, P. (1994). Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82:6 – 24.
- [Gruber, 2017] Gruber, J. (2017). *JavaScript code coverage*.
- [IEEE, 1990] IEEE (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.
- [ISO/IEC, 2001] ISO/IEC (2001). *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC.
- [Joseph, 1991] Joseph, M. (1991). Problems, promises and performance: Some questions for real-time system specification, on proceedings of rex workshop on real-time: Theory in practice, on proceedings of rex workshop on real-time: Theory in practice.
- [Kopetz, 1997] Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition.
- [Kyriazis et al., 2018] Kyriazis, D., Menychtas, A., Kousiouris, G., Boniface, M., Cucinotta, T., Oberle, K., Voith, T., Oliveros, E., and Berger, S. (2018). A real-time service oriented infrastructure. *GSTF Journal on Computing (JoC)*, 1(2).
- [Lann, 1990] Lann, G. L. (1990). Critical issues for the development of distributed real-time computing systems. In *[1990] Proceedings. Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 96–105.

- [Lengstorf and Leggetter, 2013] Lengstorf, J. and Leggetter, P. (2013). *Realtime Web Apps: With HTML5 WebSocket, PHP, and jQuery*. Apress, Berkely, CA, USA, 1st edition.
- [Loreto et al., 2011] Loreto, S., Saint-Andre, P., Salsano, S., and Wilkins, G. (2011). *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*.
- [MacCaw, 2011a] MacCaw, A. (2011a). *JavaScript Web Applications*, chapter 1. O'Reilly Media.
- [MacCaw, 2011b] MacCaw, A. (2011b). *JavaScript Web Applications*, chapter 8. O'Reilly Media.
- [Meteor, 2018] Meteor (2018). Meteor api docs. Accessed 10th August 2017.
- [MobX, 2018] MobX (2018). *MobX Docs*. Accessed 20th January 2018.
- [Naboulsi, 2011] Naboulsi, Z. (2011). Code metrics – maintainability index.
- [Newswire, 1995] Newswire, P. (1995). Netscape and sun announce javascript.
- [Overview, 2001] Overview, W. (2001). *Apple Inc.*
- [Rauschmayer, 2014] Rauschmayer, A. (2014). *Speaking JavaScript: An In-Depth Guide for Programmers*, chapter 4. O'Reilly Media, 1st edition.
- [Reenskaug, 1979] Reenskaug, T. M. H. (1979). The original mvc reports.
- [Reflux, 2018] Reflux (2018). *Reflux Repository*. Accessed 13th June 2019.
- [RethinkDB, 2018] RethinkDB (2018). Rethinkdb documentation. Accessed 10th August 2017.
- [Spewak and Hill, 1993] Spewak, S. H. and Hill, S. C. (1993). *Enterprise Architecture Planning: Developing a Blueprint for Data, Applications and Technology*. QED Information Sciences, Inc., Wellesley, MA, USA.
- [Stankovic, 1988] Stankovic, J. A. (1988). Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19.
- [Stankovic and Ramamritham, 1989] Stankovic, J. A. and Ramamritham, K., editors (1989). *Tutorial: Hard Real-time Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA.
- [Stankovic and Ramamritham, 1990] Stankovic, J. A. and Ramamritham, K. (1990). What is predictability for real-time systems? *Real-Time Syst.*, 2(4):247–254.
- [Street, 2015] Street, D. (2015). The case for unidirectional data flow. Accessed 15th November 2017.
- [Thomson et al., 2016] Thomson, M., Damaggio, E., and Raymor, B. (2016). *Generic Event Delivery Using HTTP Push*.
- [Ullman and Dykes, 2007] Ullman, C. and Dykes, L. (2007). *Beginning Ajax*. Wrox Press Ltd., Birmingham, UK, UK.

- [Voss, 2018] Voss, L. (2018). The state of javascript frameworks, 2017. Accessed 10th February 2018.
- [W3C, 2001] W3C (2001). *Transport Message Exchange Pattern: Single-Request-Response*.
- [Wang et al., 2013] Wang, V., Salim, F., and Moskovits, P. (2013). *The Definitive Guide to HTML5 WebSocket*. Apress, 1st edition.
- [Wood et al., 2014] Wood, D., King, M., Landis, D., Courtney, W., Wang, R., Kelly, R., Turner, J. A., and Calhoun, V. D. (2014). Harnessing modern web application technology to create intuitive and efficient data visualization and sharing tools. *Front Neuroinform*.
- [Young, 1982] Young, S. J. (1982). *Real Time Languages: Design and Development*. Halsted Press, New York, NY, USA.
- [Zhao et al., 2018] Zhao, Y., Gala, V., and Zeng, H. (2018). A unified framework for period and priority optimization in distributed hard real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2188–2199.