

UNIVERSITATEA BABEȘ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATIĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ENGLEZĂ

LUCRARE DE LICENȚĂ

**Dezvoltarea aplicațiilor cu interfață reactivă
folosind ReactJS și Redux**

Conducător științific
Lect. Dr. Sterca Adrian

Absolvent
Barbu Ioan Cristian

2016

BABEȘ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER
SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

**Developing reactive interface applications by
using ReactJS and Redux**

Supervisor

Lect. Dr. Sterca Adrian

Author

Barbu Ioan Cristian

2016

Contents

1.Introduction.....	4
1.1.Context.....	4
1.2.Problem statement.....	5
1.3.Thesis Structure.....	5
2.Reactive web applications.....	7
2.1. AngularJS.....	9
2.2. Ember.js.....	11
3.ReactJS.....	12
3.1.JSX.....	12
3.2.React Components.....	13
3.3.VirtualDOM.....	15
3.4.React libraries.....	15
4.Redux.....	16
4.1.Actions.....	16
4.2.Reducers.....	17
4.3.Store.....	17
4.4.Data Flow.....	18
4.5.Redux libraries.....	18
4.6.Usage with React.....	19
5.The VideoBucket Application.....	20
5.1.Use cases and specification.....	20
Use Case 1: Logging in.....	20
Use Case 2: Creating an empty video list.....	20
Use Case 3: Adding a video to a video list.....	21
Use Case 4: Deleting a video list.....	21
Use Case 5: Searching public video lists.....	22
5.2.Packaging and deployment.....	22
5.2.1.Packaging.....	22
5.2.2. Deployment.....	24
5.3.Architecture and design.....	25
5.3.1.Presentation layer.....	26
5.3.2.Logic layer.....	27
5.3.3.Data layer.....	30
5.4.Usage.....	32
5.4.1.Logging in.....	32
5.4.2.Managing video collections.....	32
5.4.3.Searching and viewing public collections.....	33
6.Conclusion.....	34
References.....	35

1. Introduction

1.1. Context

Websites that focus on usability, interoperability of end users and user-generated content, fall under the category of Web 2.0 websites. Web 2.0 does not represent the next technical iteration of the original web, instead it is a change in the way Web pages are designed and used.

In the era of the traditional Web 1.0 websites, users were limited to just viewing the content offered by the site's creators. An example would be the Britannica Online website, where all the content would be written and curated by professionals and experts. In contrast, the content of a Web 2.0 site is user-generated and curated by a virtual community, which can collaborate and interact with each other in a social media dialogue, as content creators. An example similar to Britannica Online but which follows the Web 2.0 standard is Wikipedia. Wikipedia allows its users to modify existing entries or even create their own if they provide a source to the information they want to add.^[10]

There are the three main aspects of Web 2.0:

- **Rich Internet application (RIA):** Also known as “Installable Internet Application”, a RIA is a Web application designed to deliver similar features and functions that would be normally associated with desktop applications. In RIAs processing is generally split between the client and the server, with the client side handling user interface and related activities, and the data manipulation and operation being left to the server side. ^[13]
- **Web-oriented architecture(WOA):** The official Gartner definition of WOA: “WOA is an architectural substyle of service-oriented architecture that integrates systems and users via a web of globally linked hypermedia based on the architecture of the Web. This architecture emphasizes generality of interfaces (User interfaces and APIs) to achieve global network effects through five fundamental generic interface constraints: 1. Identification of resources; 2. Manipulation of resources; 3. Self-descriptive messages; 4. Hypermedia as the engine of application state; 5. Application neutrality;”^[14]
- **Social web:** The Social web encompasses how websites and software are designed and developed in order to support and foster social interaction. As a person's activities and communication on the Web increase, so does the information about their social relationships steadily become more available. These online social interactions form the basis of most

online activities like shopping, education, gaming and social networking websites. The social aspect of Web 2.0 communication has been to facilitate interaction between people with similar tastes.^[15]

The increasing number of internet users has also played a big part in the popularization of Web 2.0 applications. Trying to appeal to this new large potential user base, companies shifted attention to single-page applications, in order to offer a more solid user experience.

1.2. Problem statement

The idea of an easily accessible “online container” where I could find curated lists or collections of videos, podcasts, articles, books or any other source of information, appealed to me during my countless hours of trying to sort and organize my personal bookmark collection. Whether I was attempting to gather well made educational videos, trying to manage a list of unappreciated short films, or collecting resources for a project, none of the existing solutions satisfied my needs completely.

What I needed was a web application that would allow me to manage my own collections of data; data that I could label with personal tags and then search by said tags, instead of trying to remember where to find it in a poorly organized bookmarks bar. Thus, the “VideoBucket” application came to be.

The purpose of the application presented in this thesis is to let users search and manage online video collections. In the current state, the application only allows YouTube videos to be added, but ultimately users will be able to add any embedded videos to their own custom collections. Many essential features still need to be implemented (rating system, tagging of videos and/or video lists and filtering videos by tags), but the current form of the application serves as a foundation for what it could become.

Following the current trend of software development, I decided to develop the “VideoBucket” application as a single-page web application, in order to become more familiar with modern technologies.

1.3. Thesis Structure

The following thesis is composed of six chapters, several sub-chapters and a reference section at the end. This first chapter is aimed to be an introduction for the subjects to be discussed.

In the second chapter, I talk about reactive web applications: what they are; how they are used and how they came to be. Also, I include some examples of two JavaScript frameworks that are currently used to implement such applications: AngularJS and EmberJS.

Chapter three is meant to be a short description of ReactJS and some of its features. Some features will be explained in-depth, such as the purpose of JSX, react components and the virtual DOM. This chapter also covers some of the packages needed to use React.

The fourth chapter covers Redux. Altho, React can be considered the main focus of this thesis, Redux deserves its own chapter because of the big impact it has on the logical layer of the application. Its purpose and features will be explained, together with the packages required for it to be used with React.

In chapter five, I describe the development process and the usage of the “Video Bucket” application. This chapter has four sub-chapters, each with a clear focus:

- 5.1 Use cases: In this section I describe the use cases of the application.
- 5.2 Packaging and deployment: Here I talk about what packages are required by the application and how the application is then deployed to Heroku.
- 5.3 Architecture and design: This section describes how the internals of the application work. Each layer of the application is then explained.
- 5.4 Usage: This section explains how the application is supposed to be used.

The final chapter sums up the thesis into a short description of the application and the development process.

2. Reactive web applications

Reactive web applications are more easily scalable and responsive than standard applications because of the way that components communicate asynchronously between each other as they react to user input and events.

The Reactive Manifesto aims to condense the knowledge accumulated from software development on how to design reliable and highly scalable applications into a set of necessary traits and to define some common terms to enable a more approachable method of communication between all participants of the development process. Thus, a Reactive System is defined as a system with the following properties^[2]:

- **Responsive:** If the system can respond to a request, it will do so in a timely manner. Responsiveness means that problems may be detected and dealt with, quickly and effectively, making this property the cornerstone of usability and utility. The consistent behavior of responsive systems is defined by their focus on providing rapid response times and delivering a consistent quality of service. In turn this behavior simplifies error handling, builds end user confidence, and encourages further interaction.
- **Resilient:** In the case of a failure, the system will remain responsive. Resilience is achieved by replication, containment, isolation and delegation. Because components are isolated from each other, failures are contained within each component, making them easier to handle.
- **Elastic:** Regardless of the workload, the system must stay responsive. In a reactive system changes in the input rate is handled by increasing or decreasing the allocated resources aimed to service these inputs. *“This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures.”*^[2]
- **Message Driven:** In order to establish a boundary between components Reactive Systems rely on asynchronous message-passing. This ensures loose coupling, isolation and location transparency and provides the means to delegate failures as messages. *“Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or*

within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.“^[2]

The development of reactive web applications started with *Single-Page Applications*, or SPAs. They are Web applications that load the entire content on a single HTML page which is then dynamically updated as the user interacts with the site.^[1]

While in traditional web applications each page has to be re-rendered when a change is made to the content, in SPAs the main page is loaded once, after which any modification happens through AJAX calls to the server. These server calls do not return markup, instead they return data, usually in JSON format. This data is then used to update only the necessary parts of the page, without needing to refresh it, making the application feel more fluid and responsive. The difference in server page lifecycles can be observed in Figure 1.

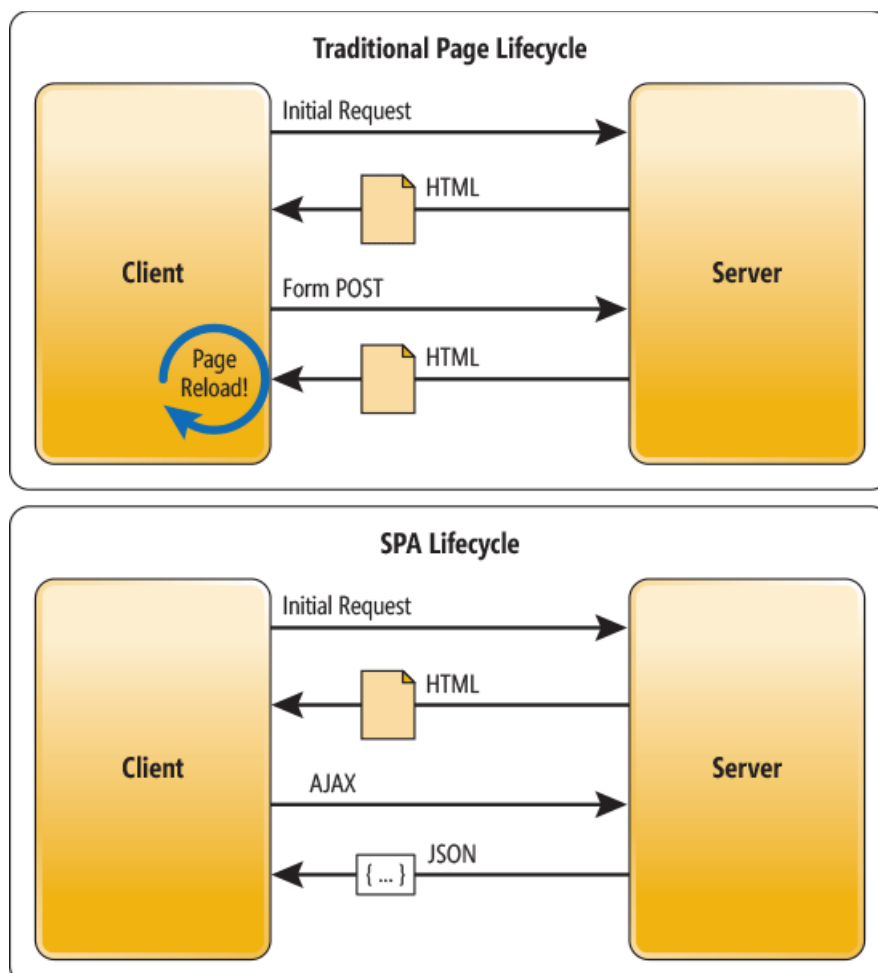


Figure 1: The Traditional Page Lifecycle vs the SPA Lifecycle[1]

This difference in server calls leads completely different application architectures. Currently there are a lot of solutions for building SPAs, each aiming to solve a specific set of problems.

2.1. AngularJS

AngularJS is a complete JavaScript-based open-source front-end web application framework maintained by Google and by a community of individuals and corporations. Angular was originally conceived as a radically more productive way for front-end developers to enhance web pages.

“AngularJS is a client-side web application framework that reimagines HTML.”^[12]

Some of the benefits of using Angular are^[11]:

- **Flexibility:** Current trend of UI-agnostic JavaScript frameworks are fully embraced by Angular. UI frameworks such as Bootstrap, can be easily integrated by using the add-ons offered by AngularUI or other similar third-party projects.
- **Familiarity:** Angular templates look very similar to traditional HTML pages, the difference being the use of inline custom tag attributes called directives.
- **Immediate productivity:** By just adding special attributes across an ordinary HTML page and a minor amount of JavaScript code, you can deliver reasonably complex client-side user interface with CRUD operations on data that would take an order of magnitude more skill and effort to build using lower-level libraries.

At first glance, it is tempting to relate Angular to Model View Controller(MVC) or Model View ViewModel(MVVM), but it is not really the case. Angular focuses on what is called two-way data binding. This means that the View and the Model are directly linked to each other; when changes happen to one, the other is automatically updated. The data is handled in what Angular calls a “controller”, but this does not enforce any specific design pattern.

There are three fundamental constructs that are available in Angular: expressions, directives and scope.

1. Expressions

Expressions have a JavaScript-like syntax and can be mixed with HTML. They are more limited than regular JavaScript and do not allow flow control structures like loops or throwing errors. All expressions are encapsulated in double curly brackets “{{}}”. Here are some examples^[12]:

```
//Good examples
<p>The number {{3 + 4}}.</p> //Will show: "The number 7"
<p>Are they equal? {{'2' == 2}}</p> //Will show: "Are they equal? True"
<p>{{"Angular" + ".js".toUpperCase()}}</p> //Will show: "ANGULARJS"
<p>{{a = 1}} remains {{a}}</p> //Will show: "1 remains 1"
<p>There {{num = 1; (num == 1 ? "is " : "are ") + num}} of them</p>//: There is 1 of them
//Bad examples
<p>{{alert("Hello world")}}</p> //Will not work.
<p>{{a = 1; ++a}}</p> //will show: "1"; the "++" is not allowed
<p>{{for (i = 0; i < 10; i = i + 1) {i}}}</p>//: // Syntax error
```

Most of JavaScript's global object cannot be used. Usually when Angular encounters an expression it cannot interpret it returns its value as an empty string.

2. Directives

“Directives are the heart and soul of Angular”^[12]. Directives come in the form of custom HTML attributes, that can be used on any tag. All built-in directives, start with the “ng-” prefix (which is a reference to the “ng” in Angular). These are some of the built-in directives:

- “ng-init”: This directive can be used to declare simple variables or objects. It can accept multiple declarations by separating them with a semicolon. The init directive cannot be used to declare functions, as they are meant to be declared in the application controllers.
- “ng-bind”: Can be used as a replacement for the double curly brackets used to evaluate an expression. By setting its value to an expression, it will replace the innerHTML of the tag with the result of the expression.
- “ng-show” and “ng-hide”: These directives can be used to choose whether to display an element or not. Their value can be set to either “true” or “false”.

These is only a very small selection of all the existing directives, considering the fact that there are also third-party directives. A good mindset to consider when developing an Angular application is “There must be a directive for that”.^[2]

3. Scopes

Scopes are used as a “single source of truth”.The main idea to using them is, that no matter in how many places some data is displayed in the view layer, it should only have to be changed in one place (a scope property), and the change should automatically propagate throughout the view.

In the directives section above, “ng-init” was described as being able to “create” variables or objects. In fact, these values are set as scope properties. The “ng-app” is usually used on the “body” tag in order for the entire site to be granted access to the global scope.

These are only some of the basic features of Angular, but even so, it makes the library a good candidate when choosing a framework for responsive web applications.

2.2. Ember.js

Ember.js is an open-source JavaScript web framework, based on the Model–view–controller (MVC) pattern. It allows developers to create scalable single-page web applications by incorporating common idioms and best practices into the framework.^[16]

From the beginning Ember was designed around several key ideas^[16]:

- **Focus on ambitious web applications:** Ember sets out to provide a wholesale solution to the client-side application problem. This is in contrast to many JavaScript frameworks, like React, that start by providing a solution to the view, and attempt to grow from there.
- **More productive out of the box:** Ember is one component of a set of tools that work together to provide a complete development stack. The aim of these tools, that are integrated in the Ember CLI, is to make the developer productive immediately.
- **Stability without stagnation:** This principle enforces the idea that backward compatibility is important and can be maintained while still innovating and evolving the framework.^[11]

Using Ember in conjunction with the Ember CLI, reduces the amount of “glue code” required and enables the developer to focus more on building the application. “Glue code” refers to unit testing, compiling assets, interacting with some back-end API, and so on. This is achieved with the built in tools that come with the Ember CLI: Broccoli (a build tool), QUnit (a JavaScript unit testing framework) and Testem (a framework agnostic, automated test runner).^[16]

The “View” part of an Ember application is formed out of components. These components are represented in two different files: a JavaScript component file that is used to model the data, and a Handlebars template, which contains the HTML. These files are automatically generated by using the command “ember generate component <component_name>”. A component can be called by encapsulating its name in double curly brackets ({{component-name}}).

In Ember data is handled in a Routes. They are in charge of everything related to setting up state, bootstrapping objects, specifying which template to render, etc. Modifications to the DOM are made by the rendering engine called Glimmer. Glimmer analyzes templates at compile time, so it can clearly differentiate between static and dynamic areas. Doing so allows for a much faster update process, making the applications feel more responsive.

There is a focus on uniformity and following the practices of web standards, which make Ember a strong contender to be chosen as a JavaScript framework for a web application.

3. ReactJS

React is an open-source JavaScript library used for creating user interfaces and is often referred to as the *V* in MVC. React offers a simple way for developers to express how the application should look at any given point in time and at the same time automatically manages all necessary UI updates whenever a data change occurs.^[3]

The library was created by a Facebook engineer named Jordan Walke, and was open-sourced at JSConf US in May 2013. It is currently maintained by Facebook, Instagram and a community of developers.

React supports most popular browsers, including Internet Explorer 9 and above.^[18]

3.1. JSX

Firstly, one of the new things that React introduces is **JSX**, a JavaScript syntax extension that looks similar to XML. Altho it is not mandatory to use React with JSX, is is recommended because it has a concise and familiar syntax.

```
//Jsx:
var Greeter = React.createClass({
  render: function() { return ( <h1 className="hello">Hello JSX!</h1> );}
})
//JavaScript:
var Greeter = React.createClass({
  displayName: "Greeter",
  render: function render() {
    return React.createElement("h1",{className: "hello"},"Hello JSX!");}
})
```

Since JSX is JavaScript, identifiers such as “class” and “for” cannot be used as XML attribute names. Instead, React DOM components expect property names like “className” and “htmlFor”, respectively.^[8]

Attribute expressions are another feature of JSX. When adding an attribute to a component or HTML element, the value assigned to it can be a JavaScript expression wrapped in curly braces. It is also worth mentioning that if an attribute is added without having a value assigned to it, JSX will treat it as “true”. *Spread attributes* is a new feature in JSX that is very similar to the ES6 spread operator (the “...”). When a component is called, it can be given an object as a spread attribute, this resulting in all o the properties from the object being set as props for the component.

```
//example object
var props = { foo: "something", bar: "something else" };
//component called without spread operator
var component = <Component foo={props.foo} bar={props.bar} />
//component called with spread operator
var component = <Component {...props} />
```

In the above example, Component will receive the same props in both cases. If multiple spread operators are used, the last set attributes override the previous ones. In the following example, the component will take both attributes from the first object, but the “bar” attribute will be overwritten with the value from the second object.

```
//example objects
var props1 = { foo: "something", bar: "something else" };
var props2 = { bar: "something better" };
//component called with multiple spread operator
var component = <Component {...props1} {...props2} />
```

This features makes JSX both easier to write and to read.

3.2. React Components

Instead of using templates like other web frameworks, React uses **components**. Components are a mixture of JavaScript and HTML that make it easier for the developer to understand the links the visual aspects and the data flow.

Creating a component is done by simply calling the “*React.createClass()*” function. The most simple component is required to have a method called “*render*”. Everything that is returned by the render method will appear on screen when the component is called. It must be noted that if a component needs to return more than one tag, the root element must always be a “div”.

One important aspect about components is the way data is handled. In React, there is no actual global data, instead data flows from parent to child component through “*props*”. When a child component is called, the props are set as attributes inside its tag. There is also the notion of “local” data for each component, and is referred to as “*state*”. The key difference between state and props is the way they can be modified. State is controlled by the component itself using methods like “*getInitialState()*”, “*setState()*”, “*replaceState()*”, *etc.* Props are controlled by whatever renders the component, allowing the component itself to only use method like “*getDefaultProps()*” (provide backup data in case some props are omitted) or “*propTypes()*” (to validate the props)^[9]. It is recommended that the state should mainly contain data relevant to the UI, so that a component's event handlers may change it in order to trigger an UI update.

Components can render multiple other components dynamically, usually based on some data stored in the props. In the case of **dynamic children**, the react documentation recommends giving each child a way to be uniquely identified by using the “key” prop. *“The key should always be supplied directly to the components in the array, not to the container HTML child of each component in the array.”*^[17]

```
// WRONG!
var ListItemWrapper = React.createClass({ render: function() {
  return <li key={this.props.data.id}>{this.props.data.text}</li>;
}});
var MyComponent = React.createClass({ render: function() {
  return (
    <ul>
      {this.props.results.map(function(result) {
        return <ListItemWrapper data={result}/>;
      })}
    </ul>
  );
}});
```

```
// Correct :)
var ListItemWrapper = React.createClass({ render: function() {
  return <li>{this.props.data.text}</li>;
}});
var MyComponent = React.createClass({ render: function() {
  return (
    <ul>
      {this.props.results.map(function(result) {
        return <ListItemWrapper key={result.id} data={result}/>;
      })}
    </ul>
  );
}});
```

Example: Creating dynamic children in React^[17]

In the above example, “results” is expected to be a list of objects all of which have at the very least a property “id”. The value of the id should be unique for each item in the list.

There are three main parts to a component’s lifecycle:^[18]

- **Mounting:** A component is being inserted into the DOM.
- **Updating:** A component is being re-rendered to determine if the DOM should be updated.
- **Unmounting:** A component is being removed from the DOM.

React provides two types of lifecycle methods that can be specified to hook into this process: **will** methods, which are called before something happens, and **did** methods which are called right after something happens.

3.3. VirtualDOM

React never directly talks to the DOM, instead it maintains a fast in-memory representation of the DOM, known as the *Virtual DOM*. The “render()” methods actually return a description of the DOM, which is compared with the in-memory representation to compute the fastest way to update the browser. React also provides escape hatches enabling developers to use the underlying DOM API directly.

The official documentation also describes the event system: “*Additionally, React implements a full synthetic event system such that all event objects are guaranteed to conform to the W3C spec despite browser quirks, and everything bubbles consistently and efficiently across browsers. You can even use some HTML5 events in older browsers that don't ordinarily support them!*”^[18]

A reference to a DOM node is needed to interact with the browser. A “ref” attribute can be attached to any element, which allows referencing the **backing instance** of the component. This is useful for accessing the underlying DOM nodes.

3.4. React libraries

React can be added to a web application by including the scripts for the **React** library itself and **ReactDOM** library, or by installing them from the NodeJS Package Manager (npm). These two libraries are separate because the ReactDOM functionality is not needed when developing native applications for IOS or Android^[7]. They also come in a compressed and uncompressed version for use in production and development.

Aside from the two core libraries, React provides a collection of useful utility modules called React add-ons. These modules are in a constant changing state and are considered experimental. They can only be used by installing them individually via npm, and some of them, like the “TestUtils” add-on, only come in uncompressed versions because they are meant to be only used during development.

If the application is written in React’s JSX syntax, the **Babel** library must also be included. Babel is a transpiler typically used to interpret the latest version of JavaScript (currently ECMA Script6/ES6) and transform it into ES5 code in order for it to be run on most current browsers.^[7] As with ES6, Babel is used to translate the JSX syntax to regular browser friendly JavaScript.

4. Redux

Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

Redux follows 3 fundamental principles^[6]:

- **Single source of truth:** A unique store is used to manage an object tree representing the state of the application
- **State is read-only:** Emitting an action is the only way to modify the state.
- **Changes are made with pure functions:** The way actions transform the state tree is described in the reducers using pure functions

State (also called the *state tree*) is a broad term, but in Redux the state of the application refers to all the underlying data that dictates how the application is displayed. It's often represented a deeply nested object that, on the top-most level, either has an object or some other key-value collection like a Map. It is good practice to keep JSON in mind when building the state, because it ensures the *serialization* of the application.

4.1. Actions

An **action** is a basic JavaScript object that describes a certain event that can occur in the application. The only requirement of actions is that they must includes a “*type*” attribute, which allows Redux to know how to handle each action. The other attributes depend on the use case, but as a general rule, actions should not contain unnecessary data. All actions are created by **action creators**, which are functions that take as argument all the necessary data for their specific actions and return the action object.

```
export var addVideoList = (videoList) => { //ES6 syntax;equivalent to function(videoList)
  return {
    type: 'ADD_VIDEO_LIST',
    videoList // ES6 syntax; equivalent to "videoList: videoList"
  }
}
```

In this example, the “*addVideoList*” action creator expects as an argument an object “*videoList*”. The action object returned by “*addVideoList*” will have the value of the “*type*” attribute equal to “*ADD_VIDEO_LIST*” and a “*videoList*” attribute equal to whatever the value of the “*videoList*” argument was.

4.2. Reducers

While actions describe the fact that something happened in the application, it's the **reducer** that specifies how the state of the application will change based on a specific action. A reducer is a *pure function* that takes two arguments, the previous state of the application and an action, and returns the new state of the application. Reducers can handle any number of action types, but it's up to the developer to decide if the application should use one big reducer for the entire state or multiple smaller reducers, each only affecting a certain part of the state.

```
export var searchTextReducer = (state='', action) => {
  switch(action.type){
    case: 'SET_SEARCH_TEXT': return { searchText: action.searchText };
    default: return state;
  }
}
```

In this example, the “*searchTextReducer*” is created, with the value of the initial state being an empty string. Each time an actions with the type “SET_SEARCH_TEXT” is called, a new state is created using action’s “searchText” attribute. For any other action type, a copy of the old state is returned.

Data handling logic can be split into multiple reducers using *reducer composition*. For example, if the state of the application could be logically decomposed into separate smaller states that do not depend on each other, a reducer can be created for each of these sub-states that would only handle actions that deal with that particular part of the state. Redux provides a function called “*combineReducers*” that is used to assign different parts of the state to other reducers. The function creates and returns one big reducer, which in turn can be combined with other reducers or be used as the reducer for the entire state.

4.3. Store

The **store** is the main component of Redux and acts as a binder for the entire application. Its functionality includes: holding and granting access to the application state, allowing the state to be updated by *dispatching* actions, *registering* and *un-registering* listeners. Each application will have a single store, as opposed to Flux. In Flux instead of using multiple reducers that call the same store, multiple stores are used to handle different parts of the state.

The way the store updates the state of the application is by **dispatching** actions. This is done with the dispatch method provided by the store. This method takes as an argument an action, usually returned by the action creator, and passes it to its assigned reducer, which in turn, passes the action to all the smaller reducers.

```

/* in the render method of a component*/
return (<div>
  <input type="text" ref="itemName" placeholder="Name of new item" />
  <button onClick = {
    this.props.dispatch(addNewItem(refs.itemName.value));
  }> Add Item </button>
</div>);

```

In the example above, each time the button’s “onClick()” event is triggered, a new action is created and dispatched, using the value of the “itemName” input as an argument. It is assumed that the “dispatch” method is passed down as a prop to the component (this also assumes that an initialized store exists), and the “addNewItem()” action creator is imported, or declared.

4.4. Data Flow

Redux architecture is built around a **strict unidirectional data flow**, meaning that all data in an application follows the same life-cycle pattern. This makes the logic of the application easier to understand and more predictable. It also encourages data normalization, so that multiple copies of the same data can be avoided.

There are 4 steps in all Redux data life-cycles ^[5]:

- 1) **Calling store.dispatch(action):** The store’s dispatch method can be called from anywhere in the application, and with the appropriate action as an argument, it is able to modify any part of the application’s state.
- 2) **The action is forwarded to the root reducer:** After the dispatch is called, the store calls the root reducer that was assigned to it, with the current state of the app and the dispatched action as arguments.
- 3) **The root reducer combines the output of the other reducers:** A new state tree is created by copying the structure of the old state; For each state attribute the value in the new state tree is determined by calling the appropriate reducer with the action. The state tree is then returned by the root reducer.
- 4) **The Redux store replaces the state with the new state tree returned by the reducer**

4.5. Redux libraries

Redux can be added to a web application by installing the library using npm, or by adding the precompiled UMD build as a “<script>” tag. There are also some complementary packages that can be installed alongside Redux such as the React bindings and the developer tools.

4.6. Usage with React

Redux has no strict relation to React and can be use together with Angular, Ember, jQuery or even plain JavaScript, but it works especially well with libraries like React because they allow UI to be described as a function of state.

In order to use React with Redux the **React bindings** library must be used. This library is not included by default with Redux and can be added by installing the “react-redux” package.^[19]

The mostly used features from the bindings are: the “Provider” component, which expects the store as a prop and passes it down to all its children, and the “connect()” function, which is used to create a wrapper around a component, granting it access to the store.^[7]

The React bindings focus on the idea of separating *presentational* and *container* components. The difference between these two can be seen in Figure2.

	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Figure 2: Presentational vs Container components^[19]

Presentational components are only used for displaying the data without having to know where it comes from, making then highly reusable. On the other hand container components are connected to the store and use the data provided to them to create other components.

Structuring an application by this principle implies designing a component hierarchy that ties in with the shape of the root state object.

5. The VideoBucket Application

As it stands, the VideoBucket applications delivers a simple solution for managing personal collections of YouTube videos.

5.1. Use cases and specification

Use Case 1: Logging in

Actor: User

Entry conditions:

- User has Google / GitHub account

Exit conditions:

- User is logged in

Event flow:

1. User clicks the “Log in with GitHub” button.
2. A pop-up window will appear, requiring the user to log in.
 - a) After successfully logging in, the user must grant the application access to their account information, by pressing “Allow”.
3. The log in window will close.

Alternate flow: If the user wishes to log in with Google

1. User clicks the “Log in with Google” button.
2. Resume the normal log in process from step 2.

Use Case 2: Creating an empty video list

Actor: User

Entry conditions:

- User is logged in
- User is on the “My Collection” tab

Exit conditions:

- A new private empty video list is created

Event flow:

1. User scrolls down the page to reach the form (It will always be the last item on the page).
2. The user types the name of the new List in the text box.
3. The user clicks the “Create new List” button.

Alternate flow: If the user wishes to create a public list.

3. The user clicks the “Public” check box.
4. The user clicks the “Create new List” button.

Use Case 3: Adding a video to a video list

Actor: User

Entry conditions:

- User is logged in
- User is on the “My Collection” tab
- User has at least one video list created

Exit conditions:

- A new video is added to the corresponding list

Event flow:

1. User decides which list to add his video to and goes to the corresponding form.
2. The user types the name of the new video in the first text box.
3. The user types the YouTube id of the new video in the second text box.
4. The user clicks the “Add new Video” button.

Alternate flow: If any of the text boxes is left empty

2. The user clicks the “Public” check box.
3. The user clicks the “Create new List” button.

Use Case 4: Deleting a video list

Actor: User

Entry conditions:

- User is logged in
- User is on the “My Collection” tab
- User has at least one video list created

Exit conditions:

- The video list is no longer visible on the page

Event flow:

1. User decides which list to add his video to and scrolls down to the list.
2. The user clicks the “X” button next to the list title.

Use Case 5: Searching public video lists

Actor: User

Entry conditions:

- User is logged in
- User is on the “Public collections” tab

Exit conditions:

- Only the video lists corresponding to the search term remain visible

Event flow:

1. User clicks the input field with the “Search video lists” placeholder
2. User starts typing the search keyword
 - a) As the user types the list is updated to correspond to the input text

5.2. Packaging and deployment

5.2.1. Packaging

All of the libraries used in this application were installed using the **Node Package Manager**. NPM is a very convenient solution for adding new modules to an application and managing dependencies. These dependencies are saved in a configuration file called ‘package.json’ which must be stored in the root folder of the application and is created with the command “npm init”.

To add a new module, the developer has to run the “npm install package_name@version --save” command. This adds the new package together with its version to the list of dependencies, and installs it in the “/node_modules” folder. When backing up the files of the application, or pushing them to GitHub, the “/node_modules” folder is usually left out because of its size. So by running “npm install” with just the configuration file, the modules folder is re-created entirely. ^[7]

The core packages used in the application are the ‘react’, ‘react-dom’, ‘redux’ and ‘react-redux’ packages. These represent the *React* and *Redux* libraries together with the *React bindings* library.

For styling, the ‘foundation-sites’ package is used. *Foundation* is a CSS framework similar to Bootstrap. It has many useful features such as a Grid system, support for right-to-left languages, media queries, support for “rem” units and others. It is used to style the application, along with some custom SCSS files. Foundation also makes its global SASS variables accessible, making the framework highly customizable.

Babel is another essential library used with the application. It is included by installing the “babel-core” and “babel-loader” packages. As mentioned in the React chapter, Babel interprets JSX and ES6 syntax and transforms it into plain JavaScript. In order to do this, Babel requires some “presets” to be installed: “babel-preset-es2015” and “babel-preset-stage-0” are used for ES6 with experimental features support and “babel-preset-react” is used for JSX support.

The final important package used is “webpack”. **Webpack** is a module bundler; it takes modules with dependencies and generates static assets representing those modules. It can only process JavaScript natively, so in order to transform other resources into JavaScript, it uses *loaders*. By doing so, every resource used forms a module. Aside from the “babel-loader” mentioned above, the loaders used and installed for this application are: “css-loader”, “sass-loader”, “script-loader” and “style-loader”.

Webpack needs to be configured in order to use loaders. This is done by creating the “webpack.config.js” file.

```
// A simplified version of the webpack.config file used for the application
module.exports = {
  entry: ['./app/app.jsx'], //where to start processing the code
  output: { path: __dirname, filename: './public/bundle.js' }, //where to output
  resolve: {
    root: __dirname,
    modulesDirectories: [ 'node_modules', './app/components' ],
    alias: { applicationStyles: 'app/styles/app.scss' },
    extensions: ['', '.js', '.jsx'] //files with JavaScript, JSX and no extensions
  }
  module: {
    loaders: [{
      loader: 'babel-loader',
      query: { presets: ['react', 'es2015', 'stage-0'] },
      test: /\.jsx?$/,
      exclude: /(node_modules)/
    }]
  }
}
```

Adding the “./app/components” folder to the “modulesDirectories” list simplifies how component files need to be required. Without this setting, when importing a component, its path from the root directory had to be specified before its name. An alternative would have been to create an alias for each component, by specifying the alias name and the appropriate file path.

After setting up the configuration file, running the “webpack” command in the terminal will bundle the entire application into the “bundle.js” file. When the server is run, this file is loaded into “public/index.html” which is used by the server to run the app.

5.2.2. Deployment

The web server is created using the “Express.js” library and is saved in the root directory in the “server.js” file.

```
//The web server
var express = require('express'); //Import the library
var app = express();

const PORT = process.env.PORT || 1234; // Set the PORT constant to 1234 if localhost

app.use(express.static('public')); //Tell the server which folder to serve.

app.listen(PORT, function() { // Start the server
  console.log('Express server is up on port ' + PORT);
});
```

It can be run using the “node server.js” command.

The version control system used for the application is Git. Git makes it very easy to manage different versions of an application and to commit changes. Setting up it is again very simple: after installing it, by running “git init” command from the root directory of the application Git is added to the project. Using “git add .“ will add all the files in the project to the list of tracked files. Folders and files that don’t need to be tracked can be ignored by adding their paths in the “.gitignore” file. Whenever the changes made to the application need to be saved, they must be first committed using “git commit -am ‘short description of changes’” and then pushed.

I chose to deploy the application on Heroku. One of the most compelling features of Heroku is the ease of setup. To add the application to Heroku, the Heroku CLI must be installed and the user must run “heroku login”. After that it is a very simple process. First the “heroku create” command creates a new application on Heroku, and adds a new remote to git. Any changes can be pushed by using the “git push heroku master” command. After a push, Heroku will install all necessary dependencies, and then run the application by executing “npm start”. This script can be set in the “package.json” in the “scripts” property. In the case of the VideoBucket application the start script is set to “node server.js”.

When deploying to production, Webpack is run with the “-p” flag, which optimizes the “bundle.js” file and considerably reduces its size (from over 10Mb to about 1.2MB). I chose to run Webpack locally and only then deploy it to Heroku, but the start script can be setup in such a way to allow Webpack to be run on Heroku before the server is started.

5.3. Architecture and design

Firstly I want to talk about folder structure of the application:

- The root folder: The root folder is the home of all the different configuration files needed by some modules (git, Webpack, npm), the server.js file used to run the server and a readme.md file.
- “./public”: This server stores the bundle.js file created by Webpack and the index.html file used to run the site
- “./config”: This folder contains configuration files used by Webpack to set the Node environment variables responsible for Firebase authentication;
- “./app”: This folder contains the entire application itself structured into multiple sub-folders and the root “app.jsx” component.

The app can be logically structured into a simplified component hierarchy as seen in Figure 3.

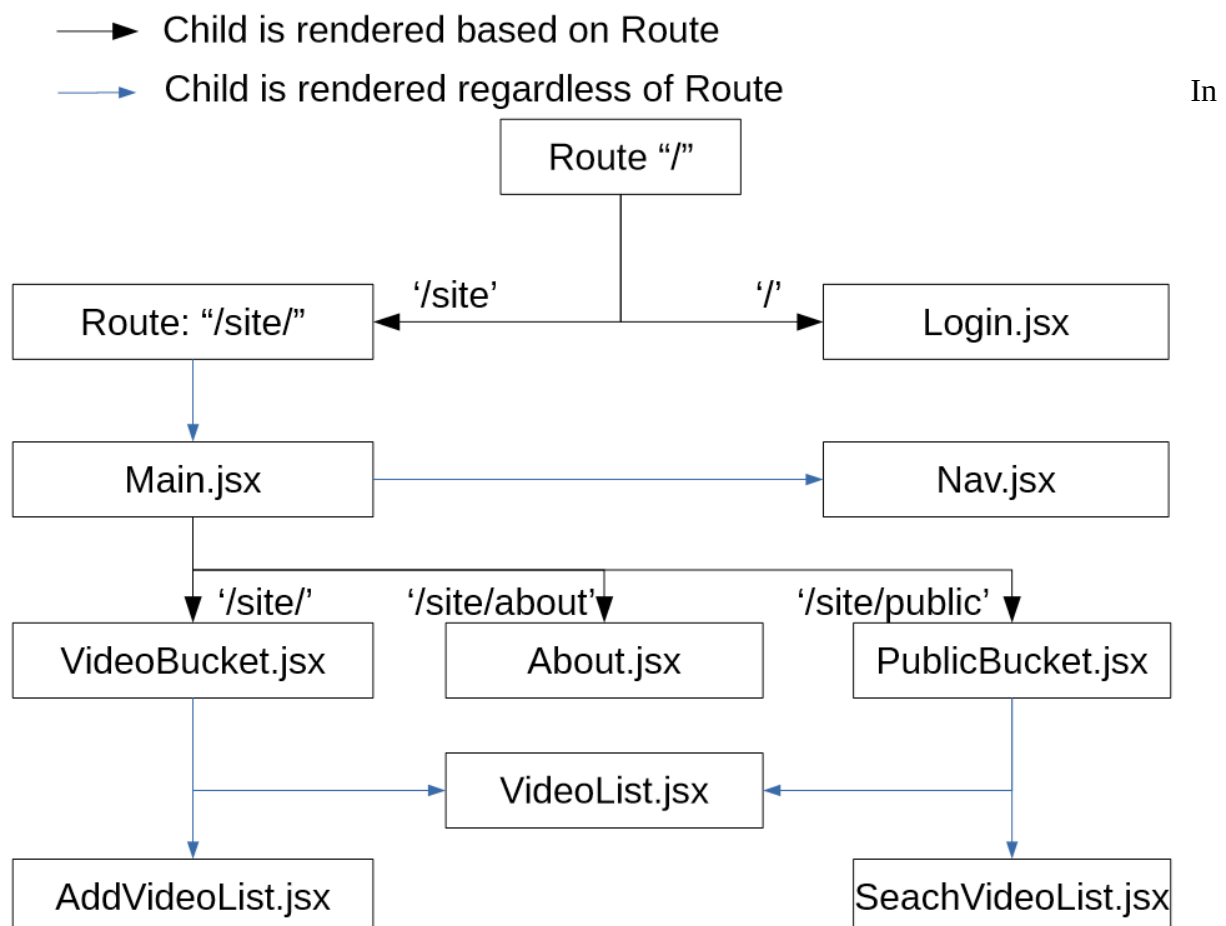


Figure 3: VideoBucket simplified component hierarchy

Figure 3, I purposely left out the root “app.jsx” component, it’s only child, the “Provider” component required for Redux, and the children of “VideoList” to make the illustration more

readable. The “Route” components are provided by the “React-router” library. The purpose of the React-router library is to offer the option of using a page router without having to implement one on the server.

The application presented in this thesis is based on a layered architectural approach, including three stacked layers:

5.3.1. Presentation layer

The user interface is handled by React. Following is a set of short descriptions of the main components used in the application and what they should display to the screen:

1. The “*Login*” component: This is a very simple component that should render a title with the text “Video Bucket” and a container with two log in buttons, one for Google and one for GitHub.
2. The “*Main*” component: This is a typical container component; it practically displays the entire site by first displaying the “Nav” component followed by whatever component gets passed down by the Router (“VideoBucket”, “PublicBucket” or “About”). This leads to the “Nav” component being visible at all times, while the user is logged in.
3. The “*Nav*” component: This components is made out of two parts: the left side, which is a list of links used to navigate the application and the right side which is a Log out button. It is displayed on the top of the page.
4. The “*VideoBucket*” component: This component is used to render the personal video collections of the user. In order to display the collections, it expects an array of objects describing them as the value of the “videoLists” prop. It takes that array and dynamically creates a “VideoList” component for each element in the list. After the lists are displayed, the “AddVideoList” component is called. This component returns a simple form consisting of two input tags, one for text and one check box, and a button with the text “Add New Video List”.
5. The “*PublicBucket*” component: This component is used to render the public video collections available to the user. The first element of the page is the SearchVideoList component, which displays a text input tag with the placeholder “Search Video Lists”. Similar to the VideoBucket component, a VideoList component dynamically creates VideoList components using the data received from the “publicLists” prop.

6. The “VideoList” component: Arguably, this is the most versatile component in the application. It expects as props the attributes of an object describing a list of videos (title, createdAt, isPublic, videoArray) and one more prop called “PublicList” which what parts of the component should be rendered. If PublicList is ‘false’(meaning that it was called from VideoBucket), the list displays some extra control elements, namely the “delete” button, the “Public List” check box and the AddVideo component (which in turn displays a form used to add videos to the list). Regardless of the value of “PublicList”, a list of dynamically created Video components is displayed, using the data from the “videoArray” prop.

7. The “Video” component: The final main component expects the information of a video as a prop alongside the “PublicVideo” prop which inherits the value of its parent’s “PublicList” prop. This component displays the name, creation date and the YouTube itself, inside a gray container, adding a “delete” button in the top right corner if the PublicVideo prop is ‘false’.

5.3.2. Logic layer

In this layer the application mostly make use of Redux. In the root of the application, “app.jsx”, the store is created when the application starts. It is then passed as an argument to the Provider component, which encapsulates all the other components passing the store down as a property when needed.

```
//Expected form of the application state
{
  searchText: 'string',
  auth:{
    uid: 'string'
  },
  videoLists: [{
    videoListId: 'string',
    title: 'string',
    createdAt: 'number',
    deletedAt: 'number',
    isPublic: 'boolean',
    videoArray: [{
      videoId: 'string',
      title: 'string',
      youtubeId: 'string',
      createdAt: 'number',
      deletedAt: 'number'
    }]
  }],
  publicLists: [/* exast same structure as videoLists */ ]
}
```

As can be seen in the example above, the state should have four root properties, namely: searchText, auth, videoLists and publicLists. The purpose of each property should be highlighted by the reducer that is assigned to it (explained below).

The reducer used to initialize the store is formed out of four other smaller reducers, each handling one of the different root properties of the state.

```
var reducer = redux.combineReducers({
  searchText: searchTextReducer,
  videoLists: videoListsReducer,
  auth: authReducer,
  publicVideoLists: publicVideoListsReducer
});
```

1. The *searchTextReducer*: handles all the actions that affect the *searchText* attribute. Because the *searchText* is a string, the initial state of the reducer will be an empty string. These are the action types handled by the reducer:

- *SET_SEARCH_TEXT*: Actions with this type are created by the “*setSearchText(searchText)*” action creator, which is called each time the text from the input field is changed in the “*SearchVideoList*” component. The “*searchText*” attribute of the action, will be passed down as the value of the new state.
- *LOGOUT*: This type of action is created by the “*logout()*” action creator, which is called when the “*Logout*” button from the “*Nav*” component is pressed. The reducer will set the new state to an empty string.

2. The *authReducer*: handles all the actions that affect the “*auth*” attribute. Because *auth* is an object, the initial state of the reducer will be an empty object. These are the action types handled by the reducer:

- *LOGIN*: This type of action is created by the “*login(uid)*” action creator, which is called when either of the log in buttons are pressed. The reducer sets the new state to an object with a *uid* property equal to that of the action.
- *LOGOUT*: The reducer sets the new state equal to an empty object.

3. The *publicVideoListReducer*: handles all the actions that affect the “*publicVideoLists*” attribute. Because *publicVideoLists* is a list of objects, the initial state of the reducer will be an empty list. These are the action types handled by the reducer:

- *GET_PUBLIC_VIDEO_LISTS*: This type of action is created by the “*getPublicVideoLists(publicVideoLists)*” action creator, which is called after the user is successfully logged in. The reducer will set the new state equal to the value of “*publicVideoLists*” argument.

4. The *videoListReducer*: which is the final reducer, handles all the actions that affect the “*videoLists*” attribute. As in the case of the *publicVideoListReducer*, the initial state of this reducer will also be an empty list. These are the action types handled by the reducer:

- **ADD_VIDEO_LIST:** This type of action is created by the “addVideoList(videoList)” action creator, which is called when the “Add New Video List” button from the VideoBucket component is pressed. The reducer creates the new state by copying all the elements from the old state and appending the new video list object provided by the action.
- **ADD_VIDEO_LISTS:** This type of action is created by the “addVideoLists(videoLists)” action creator, which is called after the user successfully logs in. The reducer creates a new state by concatenating the elements of old state with the elements of the videoLists attribute.
- **UPDATE_VIDEO_LIST:** This type of action is created by the “updateVideoList(videoListId, updates)” action creator, which is called after the “delete button” of a certain list is pressed in the VideoList component. The reducer creates a new state by copying all the video lists from the old state, while comparing each list’s id with the videoListId attribute of the action and modifying the data of the list that matches, with the attributes of the updates object.
- **ADD_VIDEO_TO_LIST:** This type of action is created by the addVideoToList(videoListId, video) action creator, which is called when the “Add new Video” button is pressed in the VideoList component. The reducer creates a new state by copying all the video lists from the old state, and when it matches the videoListId, it creates a new copy of the videoArray attribute and appends the video from the action at the end of it.
- **UPDATE_VIDEO_FROM_LIST:** This type of action is created by the updateVideoFromList (videoListId, videoId, updates) action creator, which is called when the “delete” button is pressed in the Video component. The reducer creates a new state by copying all the video lists from the old state where the videoListId does not match. If it does match, it then creates a new copy of the videoArray adding all the videos that do not match the videoId unmodified and altering the video that does with the attributes of the updates object.
- **LOGOUT:** The reducer sets the new state equal to an empty list.

All dates are handled using the **Moment.js** package. In the application, dates are used for the createdAt and deletedAt properties, which are expected to be numbers and interpreted as “time stamps”. These time stamps represent a moment in time by the number of seconds that have elapsed since January 1, 1970 (The Unix epoch) . To get the current time the “moment.unix()” function is called. Moment.js makes it very easy to format dates stored this way, by using the ‘moment.format()’ function. The dates displayed in the “Video” component are formatted as follows: “moment.unix(createdAt).format('MMM Do YYYY)’”.

When a “delete” button is pressed, the data of the video or video list is not removed from the state. Instead, a “deletedAt” attribute is added to the object containing the time at which the item was deleted. I chose to not remove the data in order to have the option of implementing a future “Undo” functionality.

Whenever multiple collections or videos need to be rendered, the data is first filtered using the “filterVideoLists(videoLists, searchText=)” and “filterVideos(videoArray)” methods, provided in the “VideoAPI.jsx” file. By default, these methods filter all objects containing the ‘deletedAt’ property. In the “PublicBucket” component when the search text is modified the list of collections is constantly updated. Here the “filterVideoLists” function receives the search term as an argument, and only returns lists containing the search term in their title.

5.3.3. Data layer

The database solution used in this application is Firebase; a cloud-hosted NoSQL database where data is stored as JSON. The only thing required to use Firebase is a Google account. In order to manage a new database you are required create a new ‘Project’. Each project has its own database and authentication section.

The authentication section on the Firebase site allows the developer to manage different Sign-in providers. For the VideoBucket application the Google and GitHub providers were set up. These providers can then be used in the application to authenticate the user.

```
// Calling a provider
var myProvider = firebase.auth.GoogleAuthProvider();
firebase.auth().signInWithPopup(myProvider).then((result) => {
  console.log('Auth worked! For user: ', result.user.uid);
}, (error) => {
  console.log('Unable to auth', error);
});
```

The signInWithPopup function returns a promise and to handle its result the ‘then()’ method is used. In case of a successful log in, the first argument function is called which receives the information of the logged in user as an argument. If the log in fails the second method is called to handle the error.

In order to communicate with the database in a timely manner, a middle-ware library for Redux called ‘redux-thunk’ is used. This library enables the use of asynchronous actions.^[7] More specifically, it allows action generators to return functions which receive the dispatch function and the state as arguments. These new types of action generators will be used to handle all interactions with the database and dispatch all appropriate actions to update the local store:

- `startLogin()`: Called from the Login component, it uses the `firebase.auth().signInWithPopup` function to log the user in and dispatches a “login” action on success using the uid returned by the server.
- `startLogout()`: Called from the Nav component, it uses the `firebase.auth().signOut()` to let firebase know the user is no longer logged in and dispatches a “logout” action.
- `startGetPublicVideoLists()`: Called from the root component, it goes through the entire database, and returns a list of object that describe the public video lists. It then dispatches the “getPublicVideoList” action with that list.
- `startAddVideoList(title, isPublic)`: Called from the “AddVideoList” component, it uses the arguments to create a new video list object. The uid property from `state.auth` is used to reference the users data in firebase. I then adds the new video lists in the database, extracts its automatically generated id, adds it to the object and dispatches the “addVideoList” action.
- `startDeleteVideoList(videoListId)`: Called from the “VideoList” component, it gets as an argument the id of the list to be marked as deleted. It references that list, adds the `deletedAt` attribute with the appropriate time and then dispatches the “updateVideoList” action.
- `startAddVideoToList()` and `startDeleteVideoFromList()`: work in similar fashion with the previous two, but they are called from the “VideoList” and “AddVideo” components respectively, and the handle video data instead of video lists.

Access to the database is managed on the Firebase application page. In the “Database” section, there is a “Rules” tab. The rules are defined as follows:

```
{
  "rules": {
    ".read": "auth !== null",
    ".write": "auth !== null",
    "users": {
      ".read": true,
      "$user_id": {
        ".read": true,
        ".write": "$user_id === auth.uid"
      }
    }
  }
}
```

With these rules the following restrictions are applied:

- If a user is not authenticated, he does not have any read or write access to the data.
- Otherwise any authenticated user can read all data stored.
- Users are allowed to write data on their assigned space in the database

5.4. Usage

Next I will describe how the VideoBucket application is meant to be used.

5.4.1. Logging in

When first entering the site, the user will see the “Log in” page. Here, two methods of authentication can be chosen by selecting the appropriate Log in button, as seen in the Figure. After one of the buttons have been pressed a pop-up window will appear, requiring the user to input their credentials. If this is the first time the credentials are used, the user will be prompted to grant the application to access the account information. After the user is logged in, the user is redirected to the ‘My Collection’ he has access to the entire site.

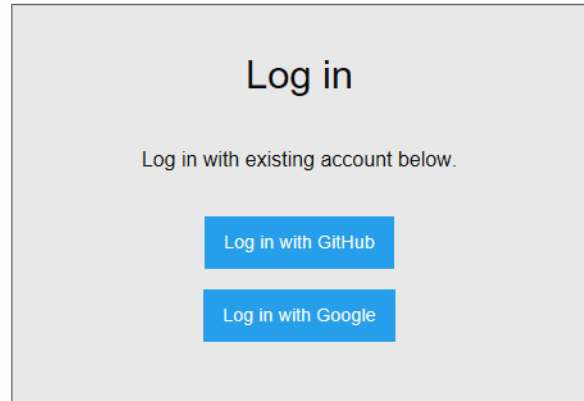
The image shows a 'Log in' form on a light gray background. At the top, the text 'Log in' is centered in a large, bold, black font. Below it, the text 'Log in with existing account below.' is centered in a smaller, regular black font. There are two blue rectangular buttons stacked vertically in the center. The top button contains the text 'Log in with GitHub' in white, and the bottom button contains the text 'Log in with Google' in white.

Figure 4: “Log in” form

5.4.2. Managing video collections

If the user has no created video collections, all he will see will be the navigation bar and a form used to add video lists as seen in the following Figure.

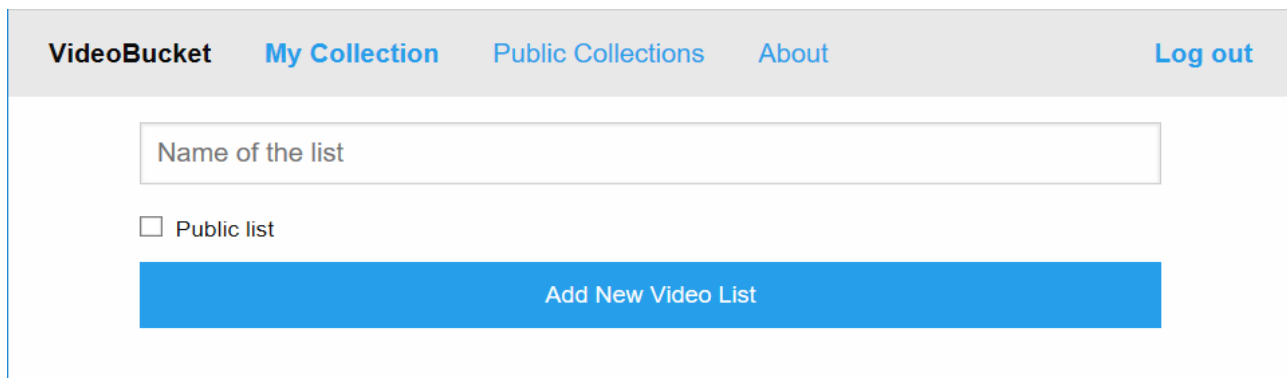
The image shows a web page titled 'My Collection'. At the top is a navigation bar with a light gray background. It contains the text 'VideoBucket' in bold black, followed by 'My Collection' in blue, 'Public Collections' in blue, 'About' in blue, and 'Log out' in blue. Below the navigation bar is a white form area. It contains a text input box with the placeholder text 'Name of the list'. Below the input box is a checkbox labeled 'Public list'. At the bottom of the form area is a large blue button with the text 'Add New Video List' in white.

Figure 5: My Collection page (after first log in)

At this point the user can create any number of video lists, by typing the name in the input box and clicking the ‘Add New Video List’ button. The lists will appear one after the other, from oldest to newest. For each list the user can do the following things:

- Change the availability of the list by clicking the “Public List” check box.
- Delete the list by clicking the red “X” button on the right side of the title.
- Add a new video to the list by filling out the form on the bottom right of the list.

Computer Science



☒ Public List

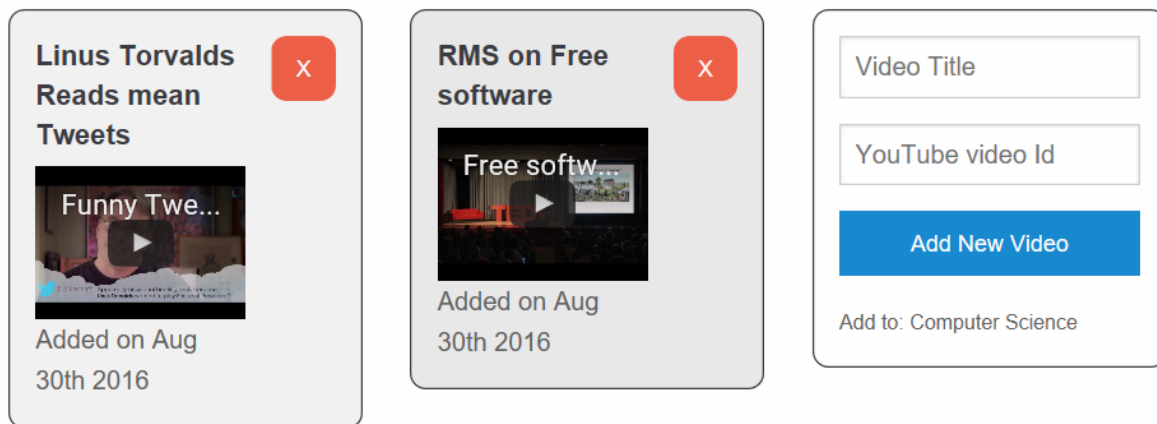


Figure 6: Example of a video list

In order to reference a video from YouTube the user must type the id of the video. This id can be taken from the URL of the video:

`s://www.youtube.com/watch?v=dQw4w9WgXcQ`

Figure 7: YouTube video ID

5.4.3. Searching and viewing public collections

The user can view public collections created by other users by clicking on the “Public Collections” link in the navigation bar. To search for a certain list, the user can type the search term in the input box. As the user types, the search results are updated. All of the displayed videos can be played.



Software Development Talks

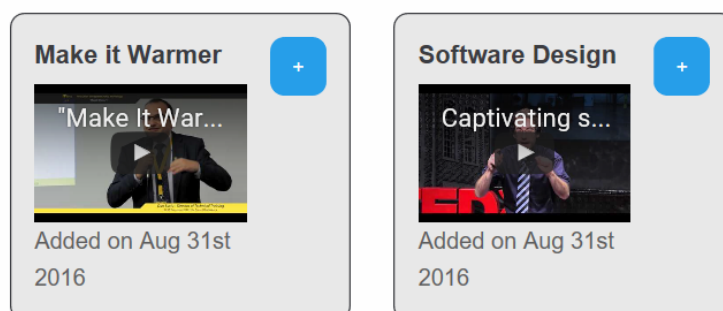


Figure 8: Searching public collections

6. Conclusion

Developing the presented application has been a unique experience. This was the first time I had the pleasure of working with modern JavaScript libraries, my previous experience being limited to HTML, CSS and simple JavaScript for the front end and some AJAX and PHP for the back end. This was also my first time working with NoSQL databases and I have to say that it was refreshing to see a different approach to storing data. I hope to improve these newly found skills and that they help me build up my future career path.

Building the application made me realize the importance of design-patterns and how not respecting them can cause many troubles. I also learned that version control systems like Git, not only help with managing the code, but constantly committing changes also offers a satisfying feeling of progression.

Considering the current state of the application, I want acknowledge the few essential features that were left out because of time constraints:

- A rating system, whether it would be like/dislike system like YouTube ratings or a ‘reaction’ based system similar that of Facebook.
- The ability to assign tags to videos and to search by them; this could either mean a set of predefined tags or community driven tags.
- The ability to add public video to a personal list;
- Adding the possibility to add a short description to a list.

There are surely other features that can be implemented, but these were on the initial design draft when I started developing the application.

Moving forward, the logical step, for me, seems to be learning React-Native. The library is used for developing Android and IOS applications, and learning to use it could further extend my grasp of the React library while at the same time acquiring some knowledge in mobile development. Or if I were to go down the Redux path, I could try to use Redux alongside an other library and observe the differences.

References

- [1] Mike Wasson: *ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET*, <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>
- [2] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson: *Reactive Manifesto*, <http://www.reactivemanifesto.org>
- [3] React Community: *Why React?*, <https://facebook.github.io/react/docs/why-react.html>
- [4] Pete Hunt: *Why did we build React?*, <https://facebook.github.io/react/blog/2013/06/05/why-react.html>
- [5] Redux Community: *Basics*, <http://redux.js.org/docs/basics/index.html>
- [6] Redux Community: *Three Principles*, <http://redux.js.org/docs/introduction/ThreePrinciples.html>
- [7] Andrew Mead and Rob Percival: *The Complete React Web App Developer Course*, <https://www.udemy.com/the-complete-react-web-app-developer-course>
- [8] React Community: *JSX in Depth*, <https://facebook.github.io/react/docs/jsx-in-depth.html>
- [9] Paul Shen: *Tutorial: State*, <http://buildwithreact.com/tutorial/state>
- [10] Tim O'Reilly: *What Is Web 2.0* <http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html?page=1>
- [11] Chris Smith: *Introduction*, <http://www.angularjsbook.com/angular-basics/chapters/introduction/>
- [12] Chris Smith: *Basics*, <http://www.angularjsbook.com/angular-basics/chapters/basics/>
- [13] Margaret Rouse: *Rich Internet Application (RIA)*, <http://searchsoa.techtarget.com/definition/Rich-Internet-Application-RIA>
- [14] Nick Gall: *WOA: Putting the Web Back in Web Services* http://blogs.gartner.com/nick_gall/2008/11/19/woa-putting-the-web-back-in-web-services/
- [15] Social Web: https://en.wikipedia.org/wiki/Social_web
- [16] Adolfo Builes: *ember 101* <https://leanpub.com/ember-cli-101/read>
- [17] React Community: *Multiple Components*, <https://facebook.github.io/react/docs/multiple-components.html>
- [18] React Community: *Working With the Browser*, <https://facebook.github.io/react/docs/working-with-the-browser.html>
- [19] Redux Community: *Usage with React*: <http://redux.js.org/docs/basics/UsageWithReact.html>