



Department of Computer Science

This project has been satisfactorily demonstrated and is of suitable form.

This project report is acceptable in partial completion of the requirements for the Master of Science degree in Software Engineering.

Rekappa – a React/Redux app for Twitch

Jason Ebueng

Student Name (type)

Chang-Hyun Jo

Advisor's Name (type)

Advisor's signature

Date

Ning Chen

Reviewer's Name (type)

Reviewer's signature

Date

Abstract

Rekappa is a media platform for Twitch that provides better features for curating content. This paper will demonstrate software engineering processes for building the application with Front-End Technologies, such as JavaScript, React, and Redux.

Twitch has been of the most popular entertainment websites at the turn of the decade – it is a livestreaming media platform that’s focused around video games. The software system servers two primary types of Users, Viewers and Streamers. The driver of media are the Streamers, users who broadcast themselves playing video games. In 2019, there are nearly 15 million unique viewers *daily* for Twitch.tv. In contrast, there are 3.8 million Streamers for Twitch as well, producing content.

<https://twitchtracker.com/statistics>

<https://www.businessofapps.com/data/twitch-statistics/>

Keywords

Keywords: Agile, requirements elicitation, Front-End Development, Software Engineering, JavaScript, HTML, React.js, Redux, API Interactions

Rekappa

Jason Ebueng

Department of Computer Science
California State University, Fullerton

Table of Contents

Contents

| | |
|--|----------|
| Department of Computer Science | 1 |
| Rekappa – a React/Redux app for Twitch..... | 1 |
| Abstract | 2 |
| Keywords..... | 2 |
| Table of Contents | 4 |
| List of Figures and Tables | 7 |
| 1.0 Introduction | 8 |
| 1.1 Problem Description..... | 9 |
| 1.2 Project Objectives | 11 |
| Mobile App Objectives | 11 |
| Coursework Objectives | 11 |
| Development Objectives..... | 12 |
| 1.3 Development Environment..... | 12 |
| Hardware | 12 |
| Software | 12 |
| 1.4 Operational Environment..... | 13 |
| 2.0 Research..... | 13 |
| 2.1 Area of Focus: State Management..... | 13 |

| | |
|--|----|
| 2.2 Area of Focus: Developing with React/Redux..... | 15 |
| 2.3 Area of Focus: Initial Attempts and Research Results | 16 |
| 3.0 Pre-Game: Planning and Requirements..... | 17 |
| 3.1 Vision..... | 17 |
| Application Description..... | 17 |
| Vision Statement..... | 17 |
| Goals for the Application | 18 |
| 3.2 Users Profile | 18 |
| Users..... | 18 |
| Budget and Time Constraints..... | 20 |
| Scrum Roles..... | 20 |
| 3.3 Requirements..... | 21 |
| Functional Requirements..... | 21 |
| Nonfunctional Requirements..... | 22 |
| 3.4 User Stories | 23 |
| 3.5 Product Backlog..... | 25 |
| 3.6 Release Planning | 28 |
| 3.7 Wireframes..... | 29 |
| 3.8 Prototype | 32 |
| 4.0 Design Description | 33 |
| 4.1 Architecture | 33 |
| 4.2 Internal Functions | 35 |
| 4.3 Interfaces..... | 36 |
| 5.0 Development..... | 38 |
| 5.1 Iteration I..... | 38 |
| Iteration Planning..... | 39 |
| Daily Stand-Up Meetings..... | 39 |
| Development Work..... | 41 |
| Testing..... | 41 |
| Sprint Burndown Chart | 42 |
| Sprint Review | 43 |
| Sprint Retrospective..... | 44 |

| | |
|---|----|
| 5.2 Iteration II..... | 45 |
| User Stories | 45 |
| Iteration Planning..... | 45 |
| Daily Stand-Up Meetings..... | 45 |
| Development Work | 46 |
| Iteration Planning..... | 48 |
| Testing..... | 48 |
| Sprint Burndown Chart | 51 |
| Sprint Review | 52 |
| Sprint Retrospective..... | 52 |
| 5.3 Iteration III..... | 53 |
| User Stories | 53 |
| Daily Stand-Up Meetings..... | 53 |
| Development Work | 54 |
| Testing..... | 57 |
| Sprint Burndown Chart | 58 |
| Sprint Review | 59 |
| Sprint Retrospective..... | 59 |
| 6.0 Implementation..... | 61 |
| 6.1 Organization of Source File Structure | 61 |
| Main View Component | 65 |
| List View Component | 66 |
| Clip Player Component..... | 67 |
| 6.2 Reference List of Files | 69 |
| 7.0 Test and Integration | 70 |
| 7.1 Acceptance Testing | 70 |
| 7.2 Results | 72 |
| 8.0 Release | 73 |
| 8.1 Installation Instructions..... | 73 |
| 8.2 Operating Instructions | 74 |
| End User Operation..... | 74 |
| 9.0 Recommendations for Enhancement | 76 |

| | |
|-------------------------------|----|
| 10.0 GitHub Repository | 77 |
| 11.0 Concluding Remarks | 78 |
| 12.0 Acknowledgements | 79 |
| Bibliography | 80 |

List of Figures and Tables

| | |
|--|----|
| Figure 1 Displaying Clips from a Twitch Streamer's profile page | 9 |
| Figure 2 Twitch Clips Search Filters | 10 |
| Figure 3 Twitch Clips Search | 10 |
| Figure 4 Twitch Clips Search Filters | 11 |
| Figure 5 Redux Cycle Data Flow | 14 |
| Figure 6 Initial Twitch Clip Component | 16 |
| Figure 7 Product Backlog in Trello | 23 |
| Figure 8 Main View | 30 |
| Figure 9 Player View | 31 |
| Figure 10 List View | 31 |
| Figure 11 Prototype Core Code | 32 |
| Figure 12 Data-Flow Diagram of the Redux Architecture | 33 |
| Figure 13 Client-Server Architectural Pattern | 34 |
| Figure 14 Root Component Code | 35 |
| Figure 15 Redux Store Schema | 36 |
| Figure 16 React Components Directory Structure | 36 |
| Figure 17 React Components Directory Structure | 37 |
| Figure 18 Twitch Search Container Component | 37 |
| Figure 19 Twitch Search Container Component | 38 |
| Figure 20 Scrum board for Iteration I | 40 |
| Figure 21 Webpack Compilation Success | 41 |
| Figure 22 Webpack Compilation Failure | 42 |
| Figure 23 Sprint Burndown Chart for Iteration | 43 |
| Figure 24 Sprint Review Demo for Iteration I | 44 |
| Figure 25 Alternative Twitch Search Queries for Iteration 1 Demo | 44 |
| Figure 26 Scrum board for Iteration II | 46 |
| Figure 27 Updated Clip Display | 48 |
| Figure 28 Adding function to the Component and Container | 48 |
| Figure 29 Searching by Channel Name | 49 |
| Figure 30 Period Drop Down Menu | 49 |
| Figure 31 Querying with Period options | 50 |
| Figure 32 Adding function to the Component and Container | 50 |

| | |
|---|----|
| Figure 33 Adding function to the Component and Container | 51 |
| Figure 34 Sprint Burndown Chart for Iteration II | 51 |
| Figure 35 Scrum board for Iteration III..... | 54 |
| Figure 36 Implementation of Navigation Menu..... | 55 |
| Figure 37 Initial View Layout | 56 |
| Figure 38 Final View Layout | 56 |
| Figure 39 Successful Sort by Game | 57 |
| Figure 40 List View | 58 |
| Figure 41 Rekappa deployed to gh-pages..... | 58 |
| Figure 42 Sprint Burndown Chart for Iteration III | 59 |
| Figure 43 Application Directory Structure | 61 |
| Figure 44 Application Directory Structure | 62 |
| Figure 45 Twitch API Functions | 62 |
| Figure 46 Action Dispatcher | 63 |
| Figure 47 Rekappa Store Schema..... | 63 |
| Figure 48 Rekappa Root Component | 64 |
| Figure 49 Container component for Twitch Clips Search..... | 65 |
| Figure 50 Render Template for Twitch Search..... | 66 |
| Figure 51 List Container Component | 66 |
| Figure 52 List Template | 67 |
| Figure 53 Clip Player Container | 67 |
| Figure 54 Clip Player Methods | 68 |
| Figure 55 Clip Player Render Template..... | 68 |
| Figure 56 Default view for Rekappa | 74 |
| Figure 57 Item View per Search Result | 74 |
| Figure 58 List View | 75 |
| Figure 59 Clip Player View | 75 |
| Figure 60 Github Repository | 77 |
| | |
| Table 1 Functional Requirements | 21 |
| Table 2 Nonfunctional Requirements | 22 |
| Table 3 User Stories | 24 |
| Table 4 Developer Tasks..... | 25 |
| Table 5 Product Backlog..... | 26 |
| Table 6 Release plan with outline of user stories assigned to each iteration..... | 28 |
| Table 7 Release plan as organized on Trello | 29 |
| Table 8 User stories for Iteration I | 39 |
| Table 9 User stories for Iteration II | 45 |
| Table 10 User stories for Iteration III | 53 |
| Table 11 Acceptance criteria and acceptance test results..... | 70 |

1.0 Introduction

1.1 Problem Description

The primary motivation for Rekappa is provide Users a better tool for searching and browsing content from Twitch. The primary content in question are *Clips*. Within Twitch.tv, Broadcasters's content can be captured into short durations. So for example, a user will activate the Clip feature and capture 30 seconds from the broadcast, much like an *instant replay* used in sporting events.

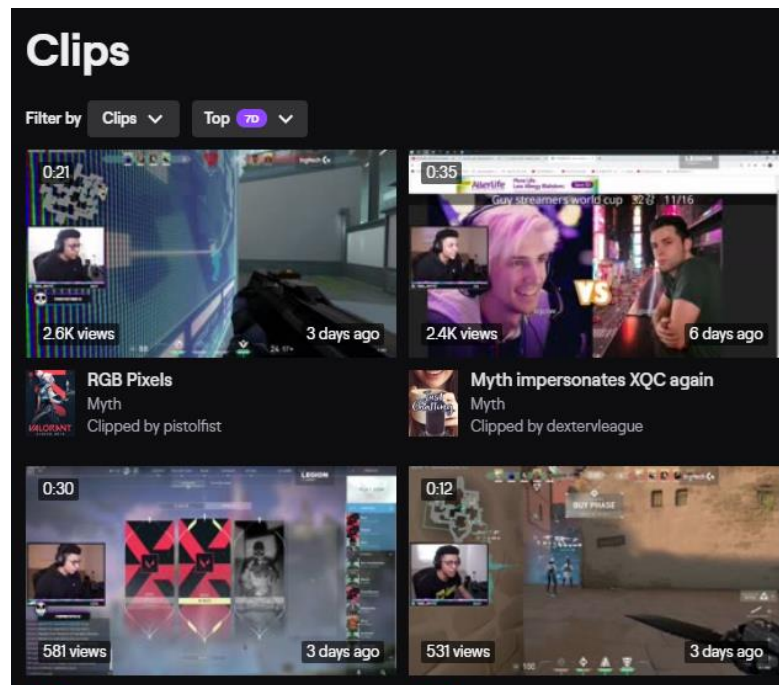


Figure 1 Displaying Clips from a Twitch Streamer's profile page

Compared to other video platforms such as YouTube, filtering and sorting features of Twitch's Clips are lacking.

Figure 2 Twitch Clips Search Filters

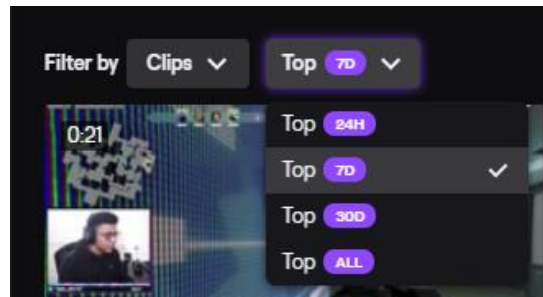


Figure 3 Twitch Clips Search

| UPLOAD DATE | TYPE | DURATION | FEATURES | SORT BY |
|-------------|----------|---------------------|------------------|-------------|
| Last hour | Video | Short (< 4 minutes) | Live | Relevance |
| Today | Channel | Long (> 20 minutes) | 4K | Upload date |
| This week | Playlist | | HD | View count |
| This month | Movie | | Subtitles/CC | Rating |
| This year | Show | | Creative Commons | |

With the figures above, one can see the stark difference in filtering options. Twitch Clips only allows filtering based on the creation time, and it only provides four durations. In contrast with Figure 1.1.3, YouTube provides substantially more filtering options on top of the time periods that Twitch Clips provide.

Currently there aren't any active or popular tools available for these kind of filtering options. However, Twitch's current API provides extra querying options that aren't available in the current interface. Thus developers can leverage this API.

Figure 4 Twitch Clips Search Filters

| Name | Type | Description |
|-----------------------|---------|--|
| <code>channel</code> | string | Channel name. If this is specified, top clips for only this channel are returned; otherwise, top clips for all channels are returned. If both <code>channel</code> and <code>game</code> are specified, <code>game</code> is ignored. |
| <code>cursor</code> | string | Tells the server where to start fetching the next set of results, in a multi-page response. |
| <code>game</code> | string | Game name. (Game names can be retrieved with the Search Games endpoint.) If this is specified, top clips for only this game are returned; otherwise, top clips for all games are returned. If both <code>channel</code> and <code>game</code> are specified, <code>game</code> is ignored. |
| <code>language</code> | string | Comma-separated list of languages, which constrains the languages of videos returned. Examples: <code>es</code> , <code>en</code> , <code>es,th</code> . If no language is specified, all languages are returned. Default: <code>""</code> . Maximum: 28 languages. |
| <code>limit</code> | long | Maximum number of most-recent objects to return. Default: 10. Maximum: 100. |
| <code>period</code> | string | The window of time to search for clips. Valid values: <code>day</code> , <code>week</code> , <code>month</code> , <code>all</code> . Default: <code>week</code> . |
| <code>trending</code> | boolean | If <code>true</code> , the clips returned are ordered by popularity; otherwise, by viewcount. Default: <code>false</code> . |

1.2 Project Objectives

The objectives of this project are divided into three categories. The “Website Objectives” describe the objectives of the Rekappa application. The “Coursework Objectives” describe the objectives of applying coursework and software engineering processes towards the project. Finally, the “Development Objectives” describe the objectives that are relevant to the development of the project.

Mobile App Objectives

The objectives below describe what the iOS app itself aimed to accomplish. These are relevant only to the complete, functional application.

1. *Create a React-Redux application with the Twitch API.*
2. *Create a user interface for the Rekappa Clip Curation.* The application will be clearly branded to represent The Skin Center. This will help patients identify The Skin Center better and help them associate the technology they are using (the iOS app) with The Skin Center.

Coursework Objectives

The coursework objectives directly reflect the project experience the Masters of Software Engineering program teaches to its students [3]. The objectives selected for this category were the ones that were most applicable and relevant to the project.

1. *Implementing a software process.* The author in question has had prior Agile experience within the tech industry, but never had to make his own Agile backlog. Software process implementation and documentation will be the focus.
2. *Research.* The project will be exploring relatively new technologies in Redux and React.js, and implementing their interactions with a Web API. The author seeks to leave behind accessible documentation that other developers can rely on.

3. *Critical Thinking and Problem Solving.* Successfully implementing a relatively new technology in Redux will be challenging as the JavaScript landscape continues to evolve. Putting all the pieces together will require thoroughness.

Development Objectives

The development objectives were determined in relation to pragmaticism and relevancy in the current technology paradigms.

1. *Create accurate user stories and a product backlog through requirements elicitation.* This identifies appropriate stakeholders and using requirement elicitation techniques. Completing this will demonstrate experience of requirements elicitation how they interact within a system.
2. *Create a working HTML5/JavaScript app using Agile methodologies.* Both JavaScript and Agile Methodologies are active paradigms in web development. Many websites are built with JavaScript and Agile practices are popular with tech companies as well.
3. *Deploy application with continuous integration.* The final development objective is to successfully deploy a front-end project alongside completion of the code. Completing this objective will help establish credibility with current engineering practices.

1.3 Development Environment

The client-side web application was developed with JavaScript, and Microsoft's Visual Code Editor was used as the development environment for this project. Visual Code Editor is cross-platform, so development was able to be tested across Windows, OSX, and Linux platforms. The application was written using the latest version JavaScript.

Hardware

- PC with Intel i5 9600K CPU
 - Windows Subsystem
Linux (WSL)
- Mac OSX 17" 2017

Software

- JavaScript
- NPM
- Trello Productivity Board
- Linux-Based Environments

1.4 Operational Environment

The client-side application is intended to be a cross-platform built with JavaScript and HTML5. Given that it is JavaScript, this means most major platforms are supported between Windows, OSX, Linux, Android, and iOS platforms. Specifically, the Google Chrome and iOS Safari Browser were tested against.

Hardware

- PC with Intel i5 9600K CPU
- Mac OSX 17" 2017

Software

- Windows 10
 - Visual Studio Code
 - Google Chrome Web Browser
- Mac OSX
 - Visual Studio Code
 - Google Chrome Web Browser

2.0 Research

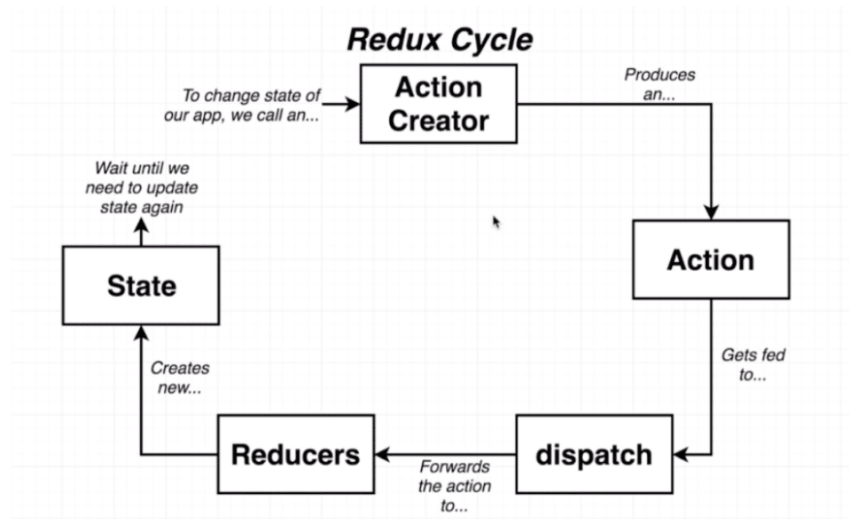
I have 2 years significant experience of building with JavaScript, but not as much deployment experience. Within these two years, I became familiar with React, but it has changed and evolved ever since. This grouping helped structure and organize the activities that occurred during this phase. The two areas of research focus are: 1) Programming Language; and 2) Development Framework.

2.1 Area of Focus: State Management

From my previous experience in the back-end, I leveraged Ruby and it's web development framework, Ruby on Rails. So I'm familiar with the concept of retaining data in the back-end, but I have not practiced it as much in the Front-End. Before Backbone.js would've been the framework of choice for handling data in a Model-View-Controller approach, but React and Redux buck that trend.

React's primary role is to present; it is a library and API for building User Interactions with JavaScript and HTML5. Redux on the other hand, is the state management framework for organizing and managing internal system data. Redux doesn't follow the pattern of Model-View-Controller for data flow but rather a "unidirectional" data flow.

Figure 5 Redux Cycle Data Flow



This has been a different approach towards organizing data, and it has been designed around the issues that recent technologies have encountered. I think the consensus is that the Model-View-Controller pattern helped started the philosophy but encouraged code to unnecessarily interact. For example, the Controller as the middleman has more responsibility because it interacts with the Model and the View. With more responsibility means more complications and relationships to test and address. Each component or part of the flow can only send data one way. The Redux Unidirectional data flow helps reduce side-effects and complexity by restricting the data flow.

Thinking in Redux is dramatically different from the Model-View-Controller pattern. With the unidirectional data flow, the developer creates hooks and reactions in response to receiving data. Redux is a JavaScript framework for managing the front-end state of the web application. It allows storing information in an organized manner in a web app and to quickly retrieve that information from anywhere in the app. The coding-pattern model it follows is called "Flux".

Redux was created by Dan Abramov in 2015. Abramov wanted to remove "unnecessary boilerplate code" that followed when created the Flux coding pattern and eliminate frustrating debugging aspects. When trying to debug a web app, one must often go through the series of steps that cause the bug to occur each time the code is changed. Debugging can be a tedious and frustrating activity, but Abramov envisioned developer tools that would allow one to undo or replay a series of actions at the click of a button.

Three principles were recognized as central to the philosophy of Redux. They are:

- *A Single Source of Truth*. The state for an entire Redux app is stored in a single, plain JavaScript object.
- *State is Read Only*. The state object can not be directly modified. Instead it is modified by dispatching actions.
- *Changes Are Made with Pure Functions*. The reducers that receive the actions and return updated state are pure functions of the old state and the action.

The advantages of Redux are:

- It is very lightweight since the library is 2 kilobytes.
- It is very fast because the time to insert or retrieve data is low.
- It is predictable since interacting with the data store in the same way repeatedly will produce the same result.

Redux's role in the client-app to help manage the data retrieved from the Twitch API and coordinate with the React components to display the application. For example, the "Clips" API will be utilized to retrieve a list of recent Clips uploaded. This will be fed into Redux's unidirectional data flow and update the client-application's State, which then updates the view for the User.

2.2 Area of Focus: Developing with React/Redux

<https://github.com/reduxjs/react-redux/blob/master/docs/introduction/quick-start.md>

I've had several experiences of building back-end applications with Ruby and Rails. The Ruby on Rails framework is very "opinionated", meaning that it forces the developer to build applications using their architecture and patterns. I anticipated more of a Model-View-Controller relationship for the front-end, since I'd expected front-end development to mirror back-end architecture. However, that was quite a naïve approach by me and fails to consider the unique circumstances and environmental challenges .

Redux and React aren't officially integrated with each other, so a developer must go out of their way to setup React and Redux together. This involves installing an additional library "React Redux" in order to officially bind React components to React's state. React reads Redux's state, and is paying attention to whenever the state changes whenever it receives "actions". The store is the central element of Redux's architecture because it holds the global state of an application. The store is responsible for updating an app's state via its "reducer", broadcasting the state to an application's view layer through subscription, and listening for actions introduced into the data flow.

The store holds data in a key-object relationship. The data in the store isn't manipulated, but rather "simplified" or reduced by the "Reducers". The Reducer is another JavaScript structure that accepts unfiltered data and extracts out specific parts. For example, an "Orchard" Store might contain a variety of fruits such as Apples, Grapes, Bananas, Pears, and Cherries. However, a "Pears" Reducer will read the Orchard store's entire listing of fruits and extract only the Pears data. This filtered data is then read by

the components. The only way for the components to trigger changes to the store, is to dispatch “actions”. These actions contain objects that signal instruction to create, read, update, or destroy data in the store.

In spirit of minimalism and tact that React embodies, React-Redux promotes separating components for presentation and logic. For example, displaying a component is contained within a “Container”, which is a wrapper for both presentation and business logic components. One component handles the rendering, the other component handles data management.

2.3 Area of Focus: Initial Attempts and Research Results

What makes React.js accessible is that each React Component has basic state coded within each component. I practiced with this in order to gauge whether I truly need Redux’s robust state management framework or if I can simply get away with the stock state management API from React.

Figure 6 Initial Twitch Clip Component

```
class TwitchData extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      game: "PLAYERUNKNOWN'S BATTLEGROUNDS",
      period: "month",
      vods: [],
      fetchedVods: [],
      offset: 0
    }
    this.launchChannel = this.launchChannel.bind(this)
    this.searchTwitch = this.searchTwitch.bind(this)
    this.fetchMore = this.fetchMore.bind(this)
    this.removeDuplicateChannels = this.removeDuplicateChannels.bind(this)
    this.resetResults = this.resetResults.bind(this)
  }
}
```

The above is the model for the Twitch Data with a pure React component. Note how the “state” is a key-value object, and that the fetchedVod is an array for storing the objects. The other class methods are helper methods to help facilitate behavior.

The TwitchData React component is nested under a parent component, App. This doesn’t follow the Flow unidirectional code pattern because the state can interact with the state and read the state, and the parent’s state is also a factor in interactions. In researching Redux, there should be only one store to deal with. Ideally the parent component will be the single source of truth and trickle it’s data to the TwitchData component within the context of it’s data flow.

The ideal architecture for a TwitchData component would be to make a Container component, which will have the Clip Views and Clip Reducers for filtering data. There should also be helper methods that are wrapped in another library. The aforementioned TwitchData above shows these types of methods as

direct class instance methods. With the initial design, debugging and code readability suffered due to the logic and presentation being held under one object. By breaking up this component, I can conquer smaller cases and build with a healthier approach in terms of testing.

3.0 Pre-Game: Planning and Requirements

The preparation for the project will cover Vision, Stakeholders, Logistics, and Agile methodologies. These aspects are made in regard to a sole-developer, but I will attempt to cover and wear many different “hats” in order to understand the various perspectives in working within an engineering team. Scrum roles handled within these perspectives along with the User profiles.

3.1 Vision

Application Description

The client-side web application is developed as a media application for consuming Twitch media for viewers and streamers. It is intended to help curate content by rapidly retrieving the Clips and streaming their playback, enabling the user to favorite and filter clips. Currently the Twitch Clip search experience doesn’t feature a robust filter set nor does it have a workflow to search clips by Games or Channel. Currently, searching through clips is a cumbersome process because viewing a single clip requires a page refresh. To compound the process, searching clips is limited to date and time durations.

Vision Statement

By providing a service that curates content quicker, content creators will be able to create more compilations and create highlights featuring a Streamer or a Game. A Streamer won’t have to rely on curating their own 7 hour broadcasts, but be able to quickly iterate through their more popular clips with the Twitch API. For Viewers, they’ll be able to consume popular clip compilations more quickly as opposed to waiting for a compilation to be uploaded to another video platform, such as YouTube or Vimeo. Analysts will also enjoy the feature set by collecting these popular clips and compare them across Games or Channels.

Goals for the Application

There are three major goals for this application. The goals are not weighted, so the first goal listed is not necessarily more important than the second or third. All goals were created with the same weight in mind. They are as follows:

- To create a relevant JavaScript project with React and Redux
- To demonstrate previous web development skills alongside current tools
- To create a useful application for the Twitch audience of Viewers and Streamers

Creating a relevant React and Redux application is a goal because they are popular front-end development technologies for many tech companies and startups. According to Stack Share, a website that reports analytics for technology stacks and setups, over 8000 companies use React. Not only is that successful on a large scale, it is also very popular with smaller startups as well due to its accessible yet powerful nature. Accomplishing this goal with the rigor and documentation of software engineering practices will help demonstrate higher level engineering skills alongside web development skills.

The second goal is demonstrate my growth as a software developer. This project is front-end focused, but I began as a back-end developer using Ruby and Rails. This was more into the developer operations, deployment, and server sides of technology. Although I enjoy the kinds of challenges back-end server has, such as query optimization and deployment, I have more interest in pursuing front-end challenges. Accomplishing this goal will help identify me as a full-stack web developer.

The final goal is to help identify myself towards an industry I'd like to center my career around. Twitch is a video platform for video games, and has interesting challenges for its userbase of video gamers. Regardless of actually being a part of Twitch or the video game industry, there are still many others who enjoy video games. Twitch's API is well documented and active, so leveraging a popular technology platform with JavaScript will help me find front-end opportunities based around media querying.

3.2 Users Profile

Users

Due to the smaller scope and nature of Rekappa, there isn't a team nor organization needed to provide a tool that can be powerful for many. The types of users were identified in terms of their role within the Twitch platform, the size of their associated viewership, and overall intended use for Rekappa. Building User profiles with a quantified approach should help clarify the types of users within the Twitch audience and their motivations for using Rekappa. To provide some context towards the level of streamers, there are nearly 700,000 active Streamers on Twitch, and Joseph ranks in the top 10,000 of streamers. The top 50 streamers range from 50,000 viewership to 10,000 viewership.

This exercise is akin to recognizing Stakeholders as people who have a vested interest in Rekappa's abilities and success.

GamesDoneQuick, Top Streamer with 50,000 viewers average

GamesDoneQuick is a popular channel with an average viewership of 50000 users. It is not one broadcaster, but more of a variety show that showcases a group of broadcasters throughout a whole week. It is important to note that they are also a charity stream and they don't actively broadcast year-round, but bi-annually instead.

GamesDoneQuick broadcasts 24/7 for a week, which is about 150 hours of content.

GamesDoneQuick don't curate their own broadcasts by themselves and relies on numerous content producers and editors to capture highlights for them. Rekappa can be a powerful tool for Joseph and his community in quickly iterating through his highlights with specific games, time durations, or creation date. Particularly, watching Clips filtered by Games would be a powerful tool for them.

Joseph, Popular Streamer of 1500 viewers average

Joseph is a popular streamer with a daily viewership of 1500 users. This is considered to be a strong community and viewership among the Twitch channels. There are more factors that distinguish Joseph from other streamers – his community has loyal fans. While 500 viewers may be from referrals or being featured on another platform, Joseph has a consistent 1000 viewers that watch his broadcasts only.

Joseph broadcasts range from 8 to 10 hours. As a popular streamer, Joseph doesn't curate his own broadcasts and relies on media editors and his community capture highlights for his brand. Rekappa can be a powerful tool for Joseph and his community in quickly iterating through his highlights with specific games, time durations, or creation date. Watching Clips filtered by View Counts would be a powerful tool for his type of broadcast.

Elyse, Small Streamer of 20 viewers average

Elyse popular streamer with a daily viewership of 20 users. Unlike larger broadcasters and channels, Brandon doesn't have a team of content producers nor a large media presence on YouTube or Twitter. Elyse is a well-received streamer, but lacks consistent broadcasts. Nonetheless she still produces hours of content.

Elyse broadcasts range from 3 to 5 hours. As a popular streamer, Elyse doesn't curate her own broadcasts and relies on viewers to capture highlights for her channel. Rekappa can be a powerful tool for Elyse by letting view the counts of her Clips, and quickly select highlights to produce content with.

Matthew, Content Producer for Broadcasters

Matthew isn't a popular streamer, but is a YouTube Content Producer that specializes in Video Editing. Matthew is contacted and contracted by Streamers who need expertise in producing media content for their Brand. For example, a channel like Joseph mentioned above will seek out his services in order to curate content and produce a compilation.

Matthew's needs as Content Producer extends beyond collecting clips. He's also responsible for stylizing and making cohesive content utilizing highlights and Clips edited together. Rekappa can be a powerful tool for him because it'll enable him to iterate and view Clips with a more streamlined user flow.

Budget and Time Constraints

No budget constraints were identified for the project.

The project was bound by time constraints and suffered unusual external circumstances. At the time of the project, a deadly pandemic has afflicted the world and disrupted daily scheduling and activities. Due to the COVID-19, a virus that has shut down cities and countries, production quality has suffered. The development in particular had to relocate and re-allocate resources in response to the deadly Corona Virus.

This event has restricted scope and re-evaluated User Stories performed in the Product Backlog.

Scrum Roles

Stakeholders: Users recognized above

Product Owner: Jason Ebueng

Scrum Master: Jason Ebueng

Development Team: Jason Ebueng

Product Owner is responsible for the success of a Product and for maximizing the value for Stakeholders.

As the Product Owner, I recognized various types of users who would have a vested interest in a tool such as Rekappa. Although a Twitch Clip tool isn't unheard of, there are currently no popular or recommended clip curating tool presently available. Informal discussion and interviews were conducted during the Research and Pre-Planning phases. Throughout the project, being a Product Owner demonstrated the mindset of handling the project. A key aspect was prioritizing Stakeholders and their relationship with Rekappa.

Scrum Master facilitates the Scrum process.

As the Scrum Master, I followed Scrum and Agile practices as elicited from the MSE program and current industry practices. I have prior experience working within Agile, so I was familiar with the process and workflow. However I never was a Scrum Master or Product Backlog planner, and it was a responsibility and task I'm eager to practice.

Development Team engineers and builds the application.

The Rekappa application is handled by a sole developer. Important to note that I identify as a developer most. However practicing better software engineering practices and management tools will improve development skills. I handled multiple “hats” and roles while being mindful of different priorities and responsibilities.

3.3 Requirements

The Product Owner elicited functional and nonfunctional requirements after research and informal interviews. The requirements were determined after analyzing priorities between stakeholders, and revised several times in order to scale down the scope to work within the development resources. Discussion was held through a “retrospective” type format, where positives, neutral, and negative opinions were shared and discussed in a timed meeting. The notes were then collected and evaluated again.

Functional Requirements

Functional requirements specify how the application will behave functionally.

Table 1 Functional Requirements

- FR-1** The application shall open to a homepage with a search bar.
- FR-2** The user shall search Twitch Clips with thumbnails displayed
- FR-3** The application should have game filtering options
- FR-4** The application should have channel filtering options
- FR-5** The application should have creation time filtering options
- FR-6** The user should be able to watch an autoplayed list
- FR-7** If a user clicks on a channel, then the Twitch Channel page should open up in a background tab
- FR-8** Users should be able to see a table sorted by popularity
- FR-9** There should be a About page

Nonfunctional Requirements

Nonfunctional requirements describe the quality attributes, such as reliability, portability, and usability. Throughout development, these nonfunctional requirements would become user stories or tasks.

Table 2 Nonfunctional Requirements

- NFR-1** The application shall be deployed for constant service online
- NFR-2** The application shall be built with JavaScript tools React and Redux
- NFR-3** The application shall be usable on a web browser for PC or OSX
- NFR-4** The application shall be usable on a web browser for smartphones
- NFR-5** The application shall display an easy to understand filtering options display
- NFR-6** The application shall be a single-page application.

- NFR-7** The application should be loaded within 2 seconds.

- NFR-8** The application shall be able to run in either portrait or landscape mode.

3.4 User Stories

Productivity website *Trello* was utilized to build the and organize the Product Backlog. Trello is a collaborative and productivity tool that organizes projects into boards. It is inspired by the *Kanban* method which is very popular among Agile methodologies. Other tools were considered such as Pivotal Tracker or ScrumDesk, but Trello was the most familiar and accessible to me.

Figure 7 Product Backlog in Trello



The User Stories were determined and adapted from the requirements. In addition to the User Stories, Tasks were also introduced to also quantify initial development engineering setups. This was to help account for Story Points when prioritizing work, and to also acknowledge the beginning phases of a product.

Table 3 User Stories

| User Story ID | User Story |
|---------------|--|
| US-01 | As a User I want to search Clips by Game so I can watch those clips |
| US-02 | As a User I want to search Clips by Broadcaster so I can watch their clips |
| US-03 | As a User I want to filter clips by Time so I can view clips in particular date ranges |
| US-04 | As a User I want to sort search results by Game |
| US-05 | As a User I want to sort search results by Broadcaster ID |
| US-06 | As a User I want to sort search results by Region |
| US-07 | As a User I want to sort search results by Followers Count |
| US-08 | As a User I want to sort search results by View Counts |
| US-09 | As a User I want to be able to save a list of search results |
| US-10 | As a User I want Twitch Clip search results to automatically play |

Table 4 Developer Tasks

| Tasks ID | Description |
|----------|--|
| DV-01 | As a developer, I want a React/Redux Environment to build the application with |
| DV-02 | As a developer, I want a Github repository to version control and save code |
| DV-03 | As a developer, I want a Twitch API key in order to use the Twitch API |
| DV-04 | As a developer, I want Github Pages to deploy to |
| DV-05 | As a developer, I want React/Redux architecture to organize the project |

3.5 Product Backlog

Product Backlog was compiled under a meeting between Product Owner, Project Manager, and Development. Project Manager conducted requirement sessions and compiled them into the figure below. Scoring was hosted by the Project Manager between Product Owner, Project Manager, and Development. Scoring followed a “fibonnaci” scale of 1, 2, 3, 5, 8, and 13. These are to roughly estimate the size and scope of each user story.

Table 5 Product Backlog

| Story ID | Story Description | Tasks | Priority | Story Points |
|----------|--|--|----------|--------------|
| US-1 | As a User I want to search Clips by Game so I can watch those clips | Create FetchGameTwitchClips functions | H | 8 |
| | | Create CLIPS actions | | |
| | | Create Search component | | |
| | | Create Results Table | | |
| | | Add Clips to Schema | | |
| US-2 | As a User I want to search Clips by Broadcaster so I can watch their clips | Create FetchChannelTwitchClips functions | M | 3 |
| | | Add to Search Component | | |
| | | Add Results Table | | |
| US-3 | As a User I want to filter clips by Time so I can view clips in particular date ranges | Update Search Components | H | 5 |
| | | Update Fetch API | | |
| US-4 | As a User I want to sort search results by Game | Create GameFilterReducer | M | 2 |
| US-5 | As a User I want to sort search results by Broadcaster ID | Create BroadcasterFilterReducer | M | 2 |
| US-6 | As a User I want to sort search results by Region | Create RegionFilterReducer | M | 2 |
| US-7 | As a User I want to sort search results by Followers Count | Create FollowersCountReducer | M | 2 |
| US-8 | As a User I want to sort search results by View Counts | Create ViewCountReducer | M | 2 |
| US-9 | As a User I want to be able to save a list of search results | Add Lists to Schema | H | 3 |
| | | Create function to parse state Lists into text | | |
| | | Display Lists Results | | |
| US-10 | As a User I want Twitch Clip search results to automatically play | Create Player Component | M | 5 |
| | | Create Player Component State | | |
| | | Create a Player Management system | | |
| | | Create Embedded Twitch Clip Component | | |
| DV-01 | As a developer, I want a React/Redux Environment to build the application with | initialize NPM | H | 5 |
| | | Setup React components | | |
| | | Setup Redux Boilerplate | | |
| DV-02 | | initialize git | M | 3 |

| | | | | |
|--------------|---|---------------------------------------|---|---|
| | As a developer, I want a Github repository to version control and save code | create development branches | | |
| | | push repository | | |
| DV-03 | As a developer, I want a Twitch API key in order to use the Twitch API | setup at Twitch.tv developer sections | M | 1 |
| DV-04 | As a developer, I want Github Pages to deploy to | create gh-pages branch | L | 1 |
| DV-05 | As a developer, I want React/Redux architecture to organize the project | create schema | H | 5 |
| | | create API Utility Methods | | |
| | | Setup React-Router | | |
| | | create components structure | | |

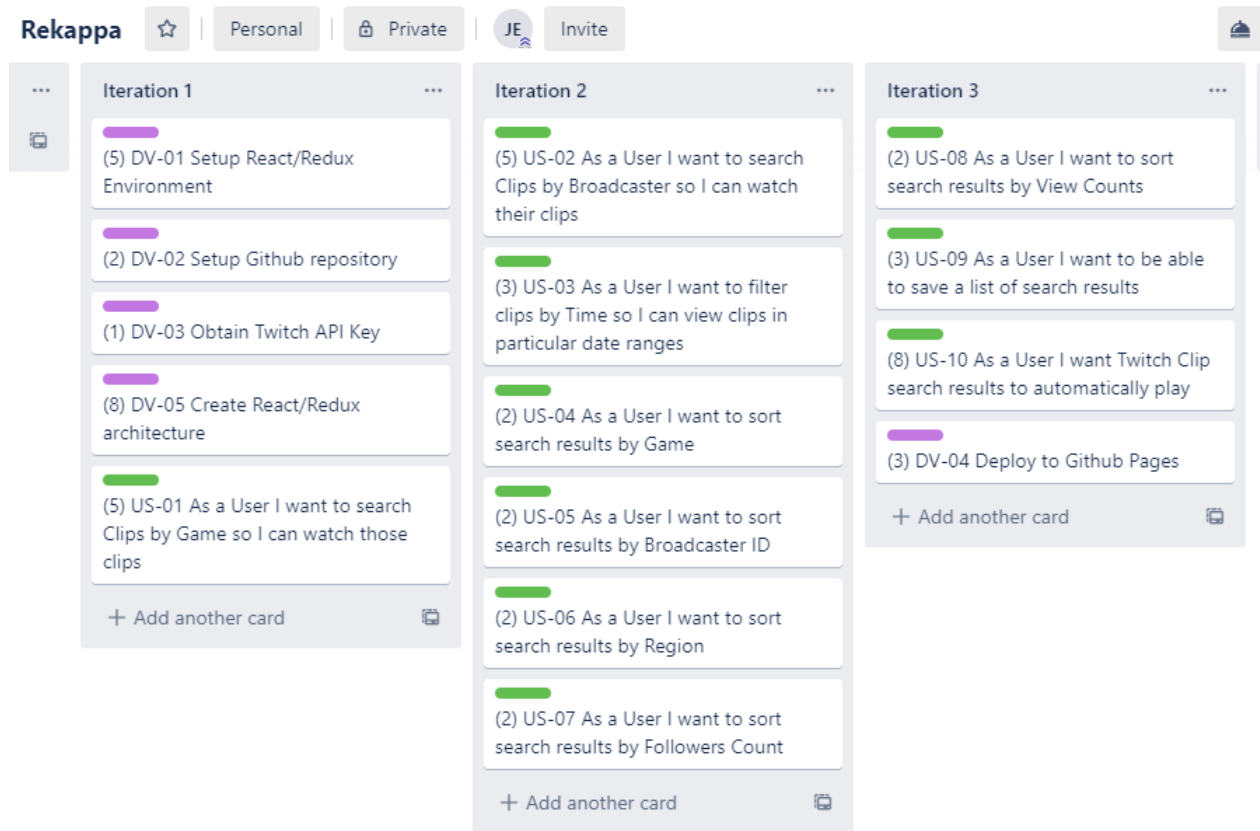
3.6 Release Planning

A release plan was determined after scoring tasks and stories. The minimum viable project will be the goal after 3 iterations, with one sprint per iteration. After each sprint, a retrospective will be performed to determine and gauge performance. The release plan is demonstrated in the following table; it's important to note that it is tentative. Within Agile, I should respond actively to any challenge and adapt to the situation.

Table 6 Release plan with outline of user stories assigned to each iteration

| Iteration | Type | Estimated Start | Estimated End | Stories | Requirements | Story Points | Estimated Hours |
|----------------------|------------------------------|-----------------|---------------|--|--|--------------|-----------------|
| Iteration I | Development Initialization | 3/20 | 4/10 | DV-01 DV-02 DV-03 DV-05 US-01 | Engineering Foundation is established for software development | 21 | 22 |
| Iteration II | Core mechanics and functions | 4/11 | 4/18 | US-02 US-03 US-04 US-05 US-06 US-07 | Sorting features are implemented within React/Redux | 16 | 20 |
| Iteration III | Minimally Viable Product | 3/21/18 | 4/4/18 | US-08 US-09 US-10 DV-04 | Application is deployed and demonstrated | 16 | 20 |

Table 7 Release plan as organized on Trello



3.7 Wireframes

During the research and pre-planning phases, sketches were developed to help visualize and prototype the user interface for Rekappa. These were drawn crudely on draw.io, a website that is popular among the tech community.

Figure 8 Main View

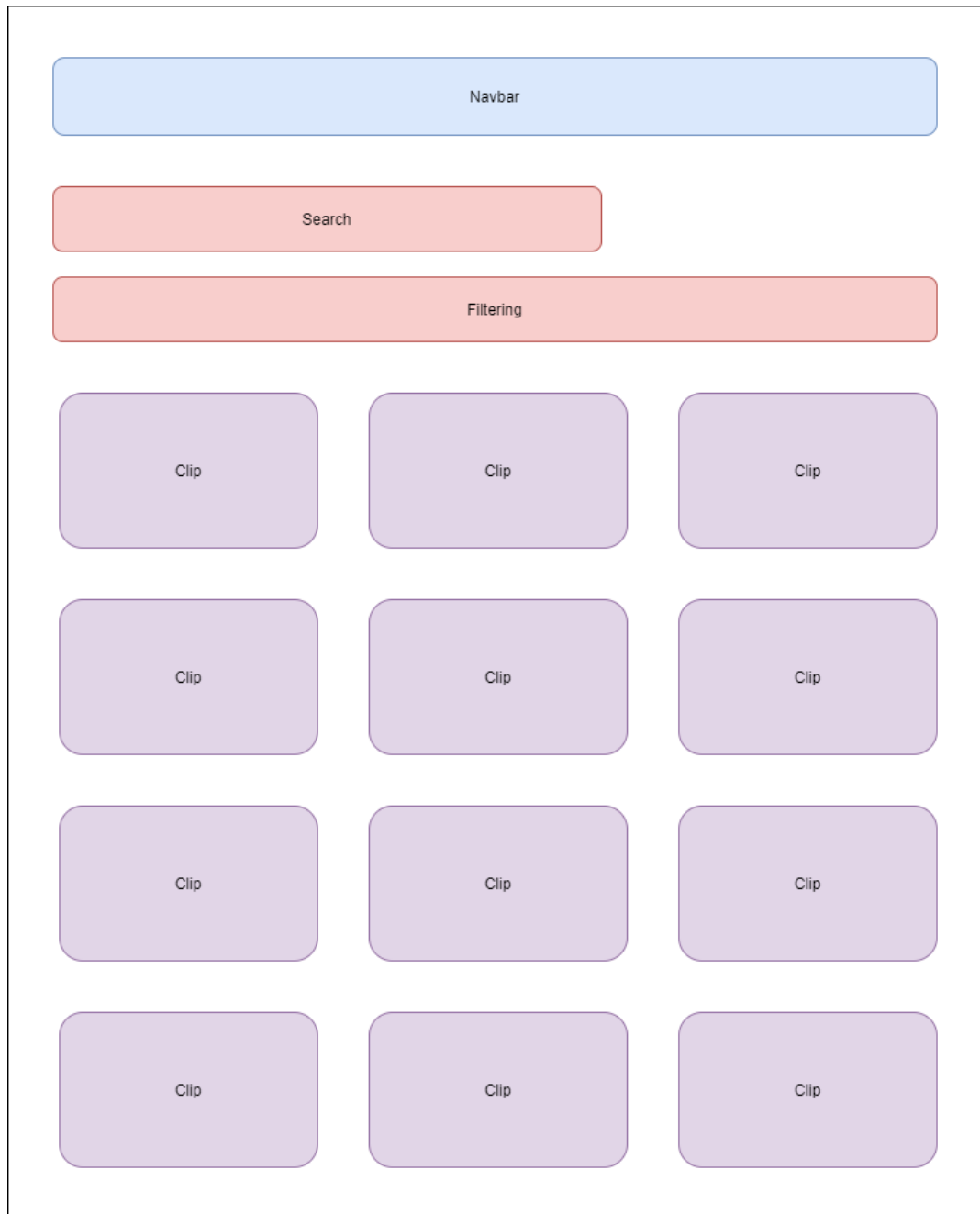


Figure 9 Player View

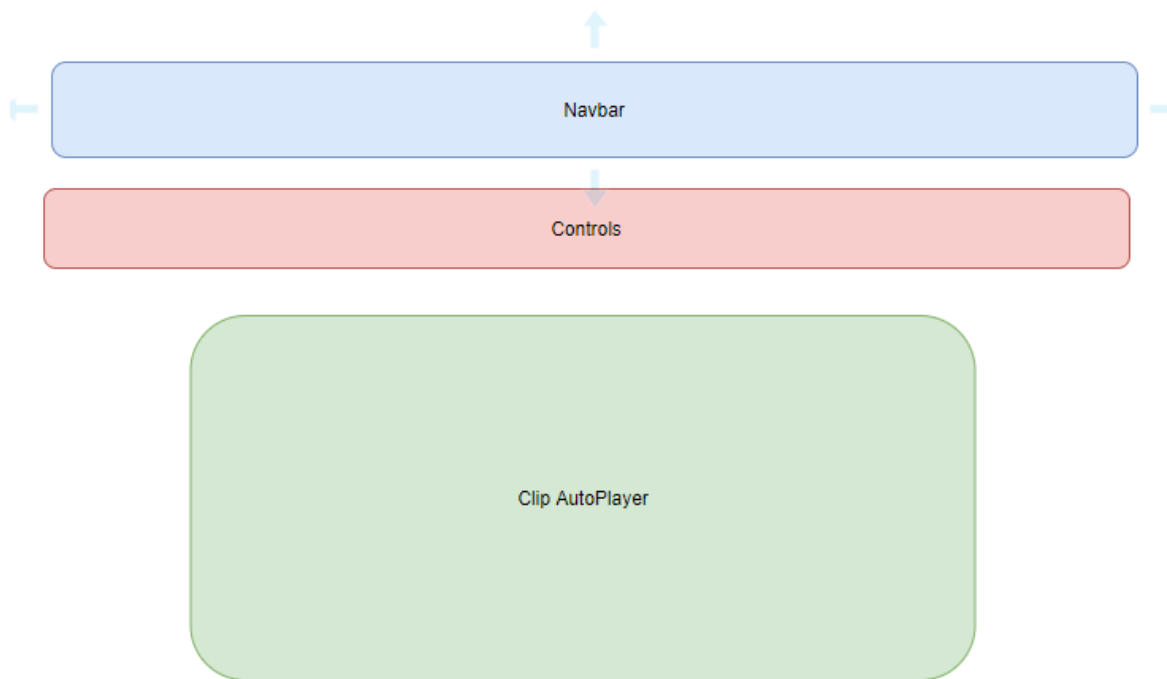
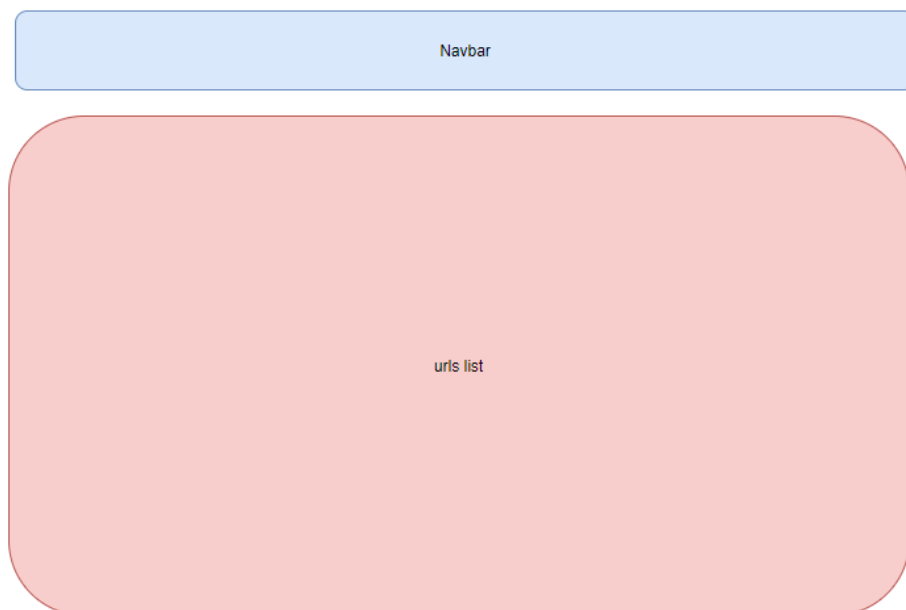


Figure 10 List View



3.8 Prototype

During the research and prior pre-planning, a prototype was built with only React. Redux wasn't used for state management, but rather React's own state API within its components. The main purpose here was to interact with the Twitch API and become familiar with its queries. It contains most sorting features, which is implemented through a public tool and reusable component "React-Table". This is a component created to automatically have sorting based on its data in its rows, but it is quite heavy and a large consumption of resources in testing.

This prototype reiterated the need for a more powerful state management and development framework. However, it was reassuring to observe the API's behavior and work out some challenges. One such challenge was using the authentication token with Twitch. A Redux solution would allow a central source to store state, which would help in securing and privately passing sensitive data within the application.

Figure 11 Prototype Core Code

```
12 class ClipBox extends React.Component {
13   constructor(props) {
14     super(props)
15 >   this.state = { ...
27   }
28
29   this.handleKeyPress = this.handleKeyPress.bind(this)
30   this.searchTwitch = this.searchTwitch.bind(this)
31   this.shuffleClips = this.shuffleClips.bind(this)
32   this.filterClips = this.filterClips.bind(this)
33   this.fetchChannelClips = this.fetchChannelClips.bind(this)
34   this.fetchGameClips = this.fetchGameClips.bind(this)
35   this.resetClips = this.resetClips.bind(this)
36   this.setClipBoxState = this.setClipBoxState.bind(this)
37   this.reverseClips = this.reverseClips.bind(this)
38 }
39
40 > componentDidMount() { ...
51 }
52
53 componentDidUpdate(prevProps, prevState) {
54 >   if (this.state.currentStr !== prevState.currentStr) { ...
56   }
57 }
58
59 > fetchChannelClips(channel, period, limit, cursor = "") { ...
72 }
73
74 > fetchGameClips(game, period, languages, limit, cursor = "") { ...
87 }
88
89 > filterClips() { ...
95 }
96
```


The prototype's core code was a monolithic React component, not utilizing Redux but it's standard API for controlled state management. Initially this was heavily considered for the minimum-viable-project, but the application's logic became more complicated than anticipated.

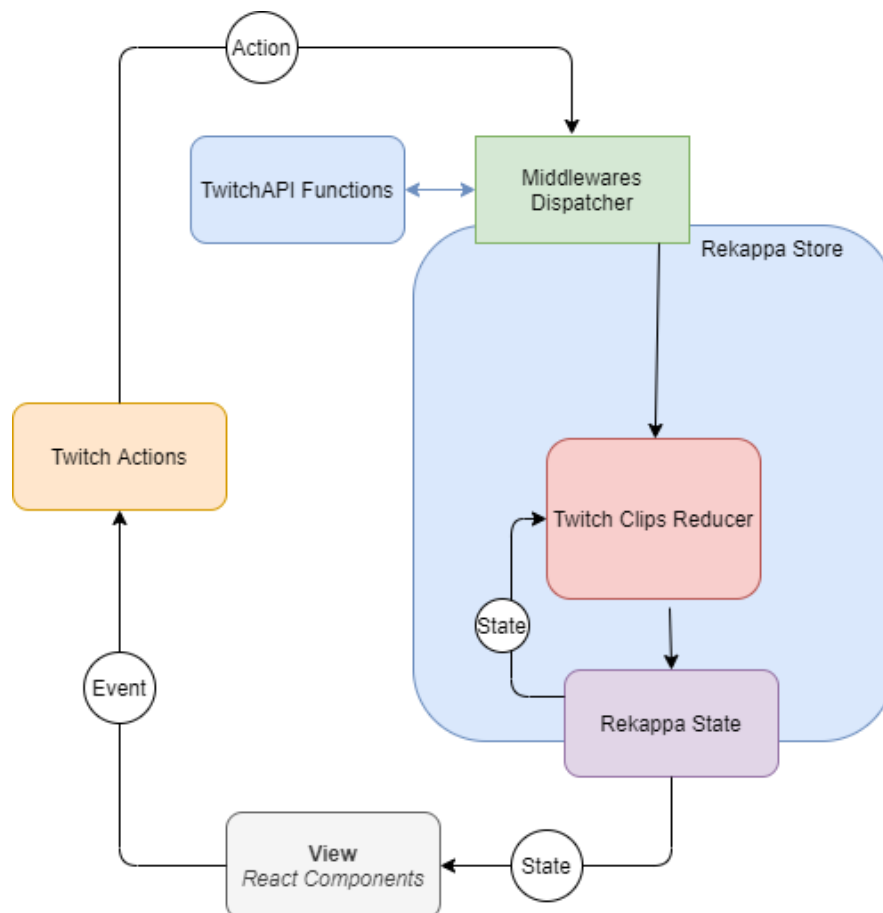
With an idea of how the code will behave, Wireframing to sketch out the application and planning began.

4.0 Design Description

4.1 Architecture

The Twitch API functions essentially start the chain of data that will flow and trigger user interfaces throughout Rekappa. Redux's data-flow is inspired by Flux, which is a unidirectional data-flow pattern.

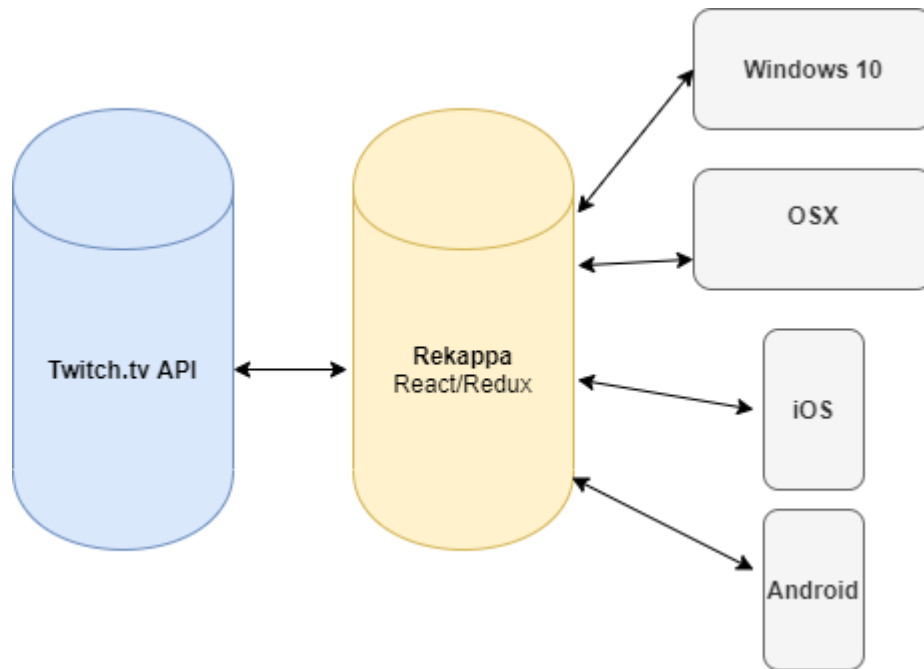
Figure 12 Data-Flow Diagram of the Redux Architecture



These should mirror the directory structure referenced for the Redux architecture. Once the Twitch API functions receive data from the Twitch API servers, it *dispatches* an action carrying a message and the data. Compared to a Model-View-Controller pattern, this would be most similar to a “Controller”

responsibility. However this is a one-way data-flow, whereas a Controller can send and receive data between the Model and View. The stack architecture is recognized by the back-end services in Twitch.API, Github, and Heroku, and the front-end architecture being any modern web browser supporting JavaScript.

Figure 13 Client-Server Architectural Pattern



The stack architecture's core is the JavaScript stack of React and Redux. It will interact with Twitch.tv's API servers to request *Clips* using their queries. After receiving the data payload, it will then create HTML5/CSS/JavaScript files that any platform running a web browser. These web browsers include Windows 10's Microsoft Edge, Google's multi-platform Chrome Browser, and Apple's Safari Browser for its iOS devices. Unfortunately, this means that network connectivity is a bare minimum in accessing the intended deployment, but the code repository will also be available for manually downloading and compiling.

4.2 Internal Functions

There are two main internal functions that are important to the app's functionality. The first function is the framework that handles the side menu. The second function is the WebKit WebView.

1. Root Component

Figure 14 Root Component Code

```
components > root.jsx > [e] Root
1  import React from 'react';
2  import { Provider } from 'react-redux';
3
4  import TwitchClipsSearchContainer from './twitch_clips_search_container'
5
6  const Root = ({ store }) => {
7    return (
8      <Provider store={store}>
9        <TwitchClipsSearchContainer />
10      </Provider>
11    );
12  };
13
14  export default Root;
```

The root component is the knot that ties Redux and React together. The *Provider* component wraps the developer made components into its Store. This helps curbs the issues of *prop threading* – where data and state is transferred down many child components. That creates a large dependency on a data to flow through many components.

2. Redux Store

Figure 15 Redux Store Schema



```
store > JS store.js > ...
1 import { createStore, applyMiddleware } from 'redux';
2 import thunk from 'redux-thunk';
3 import rootReducer from '../reducers/root_reducer';
4
5 const configureStore = () => {
6   return createStore(rootReducer, applyMiddleware(thunk));
7 };
8
9 export default configureStore;
```

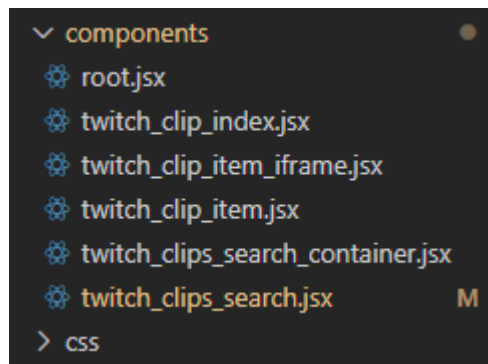
```
reducers > JS root_reducer.js > ...
1 import { combineReducers } from 'redux';
2
3 import twitchReducer from './twitch_reducer';
4
5 export default combineReducers({
6   clips: twitchReducer
7 });
```

This is the central code unit for managing state. The Store organizes data and stores it, while simultaneously updating the data-flow cycle to alert React components to respond to.

These code are critical towards the infrastructure of the client-side code. Trouble-shooting and debugging should begin and consider these modules with more priority in examination.

4.3 Interfaces

Figure 16 React Components Directory Structure



```
▼ components
  ⚙ root.jsx
  ⚙ twitch_clip_index.jsx
  ⚙ twitch_clip_item_iframe.jsx
  ⚙ twitch_clip_item.jsx
  ⚙ twitch_clips_search_container.jsx
  ⚙ twitch_clips_search.jsx
  > css
```

The primary user interfaces are from the JavaScript library, *React*. The *Root* component summons the *twitch_clips_search* as a parent component. The component is comprised of a search form, and an unordered list. In theory, the HTML is light, and the JavaScript framework around it is light as well. These are supplemented by the *css* styling, which utilizes Flexbox to quickly setup a wrapped list evenly.

Figure 17 React Components Directory Structure

```
.twitch_clip_search {  
  display: flex;  
  flex-direction: column;  
  align-items: center;  
}  
  
.clips_index {  
  width: 75vw;  
  height: 75vh;  
  display: flex;  
  flex-wrap: wrap;  
  justify-content: space-evenly;  
}
```

The container component connects the actual React Components to the Redux data-flow. It receives the API action creators and make a “higher-order” function out of the inserted React Component. The *twitch_clip_search_container* utilizes the *mapStateToProps* and *mapDispatchToProps* to integrate the Redux Store API.

Figure 18 Twitch Search Container Component

```
components > twitch_clips_search_container.jsx > mapDispatchToProps  
1  import { connect } from 'react-redux';  
2  import TwitchClipsSearch from './twitch_clips_search';  
3  import { fetchSearchTwitchClipsByGame } from '../actions/  
   twitch_actions';  
4  
5  const mapStateToProps = state => {  
6    return { clips: state.clips };  
7  };  
8  
9  const mapDispatchToProps = dispatch => {  
10    return {  
11      fetchSearchTwitchClipsByGame: searchTerm =>  
        dispatch(fetchSearchTwitchClipsByGame(searchTerm))  
12    };  
13  };  
14  
15  export default connect(  
16    mapStateToProps,  
17    mapDispatchToProps  
18  })(TwitchClipsSearch);
```

Figure 19 Twitch Search Container Component

```
components > twitch_clips_search.jsx > TwitchClipsSearch > constructor
1  import React from 'react';
2
3  import TwitchClipsIndex from './twitch_clip_index';
4
5  class TwitchClipsSearch extends React.Component {
6    constructor() {
7      super();
8      this.state = { searchTerm: 'Super Mario Bros.' };
9      this.handleChange = this.handleChange.bind(this);
10     this.handleSubmit = this.handleSubmit.bind(this);
11   }
12
13   componentDidMount() {
14     this.props.fetchSearchTwitchClipsByGame(this.state.searchTerm);
15   }
16
17   handleChange(e) {
18     this.setState({ searchTerm: e.currentTarget.value });
19   }
20
21   handleSubmit(e) { ...
25   }
26
27   render() { ...
39   }
40 }
41
42 export default TwitchClipsSearch;
```

These together form the skeleton and processing unit of the parent component. Note how similar it is towards the prototype, but integrated with Redux's state management.

5.0 Development

The development section is divided into three subsections, one for each iteration of the project. Each section describes the events that occurred during the iteration. These include the iteration planning, daily stand-up meetings, development work, testing efforts, sprint burndown chart, sprint review, and sprint retrospective.

This section will follow 3 Iteration phases of the software development cycle, which is comprised of the stories on the Product Backlog. Each phase will cover iteration planning, daily-stand up meetings, software development, and sprint analysis.

5.1 Iteration I

User Stories

Table 8 User stories for Iteration 1

| Story ID | Story Description | Tasks | Priority | Story Points |
|----------|--|---------------------------------------|----------|--------------|
| US-1 | As a User I want to search Clips by Game so I can watch those clips | Create Twitch APIs | H | 8 |
| | | Create CLIPS actions | | |
| | | Create Search component | | |
| | | Create Results Table | | |
| | | Add Clips to Schema | | |
| DV-01 | As a developer, I want a React/Redux Environment to build the application with | initialize NPM | H | 5 |
| | | Setup React components | | |
| | | Setup Project | | |
| DV-02 | As a developer, I want a Github repository to version control and save code | initialize git | M | 3 |
| | | create development branches | | |
| | | push repository | | |
| DV-03 | As a developer, I want a Twitch API key in order to use the Twitch API | setup at Twitch.tv developer sections | M | 1 |
| DV-05 | As a developer, I want React/Redux architecture to organize the project | create schema | H | 5 |
| | | create API Utility Methods | | |
| | | Setup React-Router | | |
| | | create components structure | | |

Iteration Planning

After the Product Backlog was determined for the Rekappa project implementation that will be deployed and live. Note that there are Developer Stories, which I think are relevant tasks to be done during development that should be recognized for their requirements and how it takes away from the project's resources. Thus stories DV-01, DV-02, DV-03, DV-05, and US-01 were prioritized.

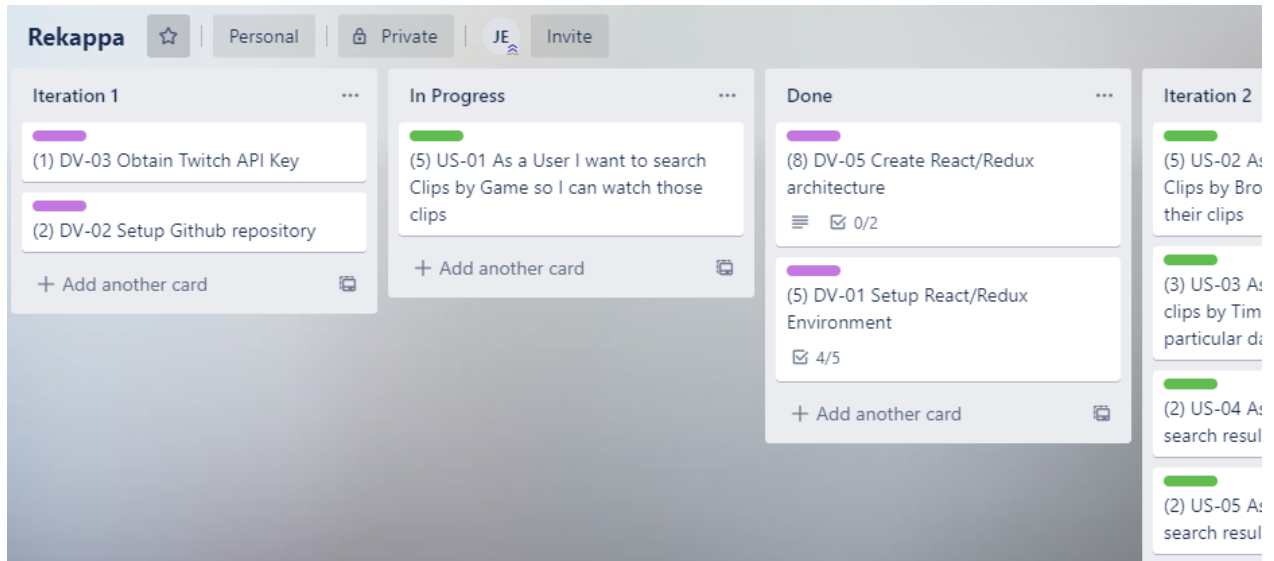
This iteration phase should result in a healthy foundation from which the rest of the Rekappa development can safely be built from. This establishes the environment for operations and development, with Git version control and the structure of the JavaScript project files. Although React development was more familiar to estimate, the Redux development represented new territory to explore and thus more difficult to estimate. Nonetheless the iteration took 12 hours in Iteration 1.

Daily Stand-Up Meetings

A standard Scrum daily stand-up meetings is about 15 minutes. Their primary function is to sync the team on immediate goals and progress towards the project. The stand-up meetings for Iteration 1

focused on the goals of establishing the application groundwork – the development and project management sides perspectives were considered towards these goals.

Figure 20 Scrum board for Iteration 1



Development Work

DV-01, DV-02, DV-05

These user tasks were largely establishing the infrastructure for the whole project. A Redux/React boilerplate template was utilized in order to establish the project directories. React's terminal command *create react app* was not utilized due to the extra software packed in – this infrastructure was built more primitively for a more lean stack.

The React/Redux environment is being developed primarily on a Linux based environment with Window's 10 *Windows Subsystem Linux*, which essentially installs a Linux environment thus enabling web development with the Linux ecosystem. Most of the effort here was double-checking operational environments and the JavaScript compatibilities between packages.

DV-03

This task was to setup the Twitch integration that Rekappa will rely heavily on. Although this task was primarily allocating a developer key, much work was done working with the Twitch API urls and examining their documentation. At the beginning of the project, the API was at Version 5, Kraken. However during development, Twitch has started to push the new api *Helix*. Twitch has sent notice and warning that Version 5 will be deprecated, so more research was done to adapt to the new API.

US-01

This task is more similar to the follower user stories, however this user story will be a culmination of the developer stories. US-01 is the core feature of Rekappa; implementation of the Redux/React architecture of this vertical slice of the project will enable future features to be centered around these fetched results. The Redux store was updated with a *clips reducer*, and the Wireframes for the Main View was used to implement the components with React.

Testing

DV-01, DV-02, DV-05, DV-03

Testing was performed primarily with the npm tool *Webpack*. Webpack compiles JavaScript code and packages together while resolving dependencies or incompatibility issues. Notably, it automatically compiles the Rekappa JavaScript code every time there's an update to the project, like code or a new file.

Figure 21 Webpack Compilation Success

```
jason@DESKTOP-B4P5PSG:/mnt/c/Users/Jason/WebDev/Rekappa$ npm run start
> Rekappa@1.0.0 start /mnt/c/Users/Jason/WebDev/Rekappa
> webpack --watch --mode=development

webpack is watching the files...

Hash: 55a6e7313f66a0b23c58
Version: webpack 4.42.1
Time: 9174ms
Built at: 04/22/2020 2:05:36 PM
    Asset      Size  Chunks             Chunk Names
  bundle.js  1.08 MiB       0  [emitted]          main
bundle.js.map 1.21 MiB       0  [emitted] [dev]    main
Entrypoint main = bundle.js bundle.js.map
```

I begin development by running *webpack watch*, and a successful compilation would be highlighted green. This follows the familiar “Red-Green-Refactor” development pattern. Whenever there’s an error, such as a new component or syntax error, Webpack will alert with the most recent update.

Figure 22 Webpack Compilation Failure

```
bundle.js.map 1.21 MiB   main [emitted] [dev] main
Entrypoint main = bundle.js bundle.js.map
  69 modules

ERROR in ./components/twitch_clips_search.jsx
Module build failed (from ./node_modules/babel-loader/lib/index.js):
SyntaxError: /mnt/c/Users/Jason/WebDev/Rekappa/components/twitch_clips_search.jsx: 'import' and 'export' may only appear at the top level (42:0)

   40 |   // }
   41 |
>  42 |   export default TwitchClipsSearch;
      |         ^
   43 |
      at Object._raise (/mnt/c/Users/Jason/WebDev/Rekappa/node_modules/@babel
```

The main focus was producing the most central feature, *twitch search by Game*. Once those components were successfully compiled, then primitive user interfacing testing was used to confirm the application’s behavior.

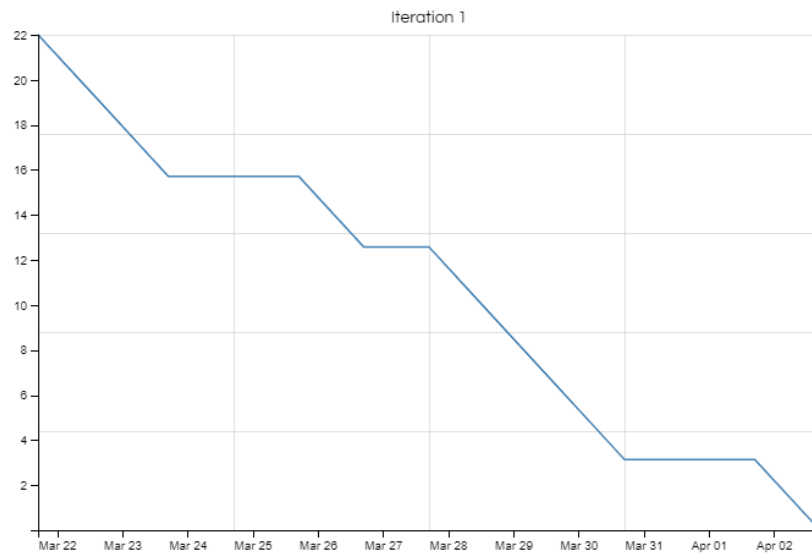
US-01

As a User, I should be able to input a Game title and view the results. The first user story of the project, and it an important one since it should be the infrastructure for the rest of the application. Testing was done by first testing the twitch API helper methods, and confirming that data was being sent back. Afterwards, the Redux store methods and actions were setup and confirmed that it performs a data-flow cycle of fetching Clips. Afterwards, the story was completed when the user interface was created, allowing a user to type in a Game title and receive clips.

Sprint Burndown Chart

The Iteration 1 Sprint Burndown Chart reflects my available time to work on the project in conjunction with the program curriculum work and my part-time job as a Computer Science Tutor. Although I allocated myself 2 weeks, the chart shows more activity towards the end of the weeks because of availability.

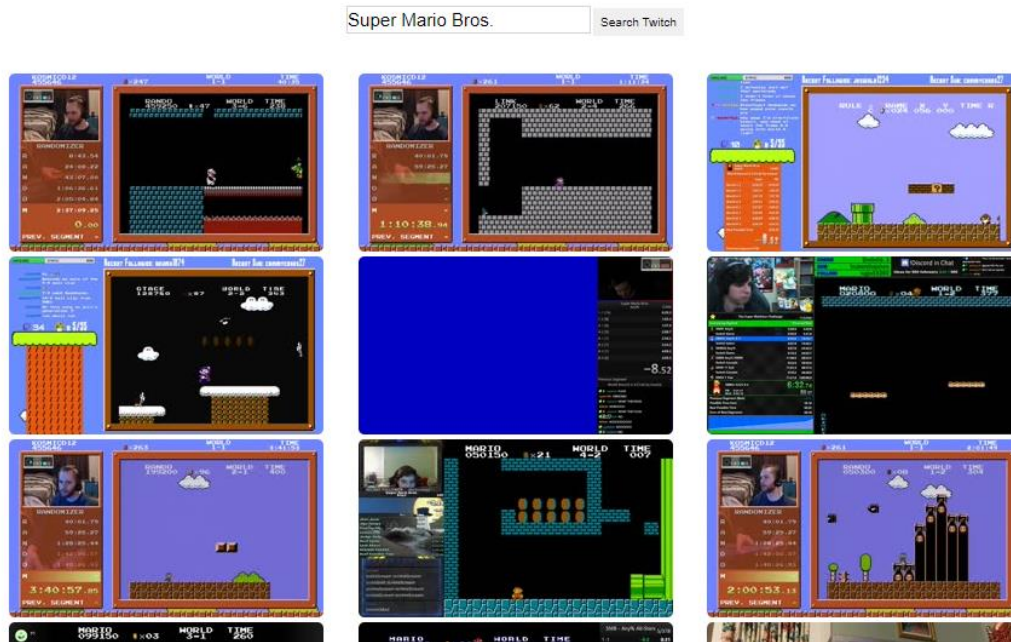
Figure 23 Sprint Burndown Chart for Iteration



Sprint Review

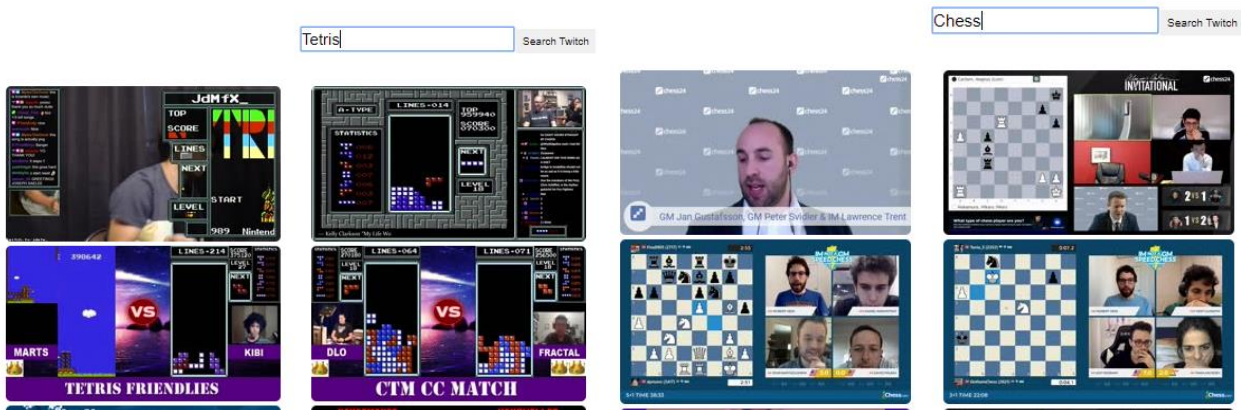
This Sprint helped satisfy the expectations of the Product Owner and gave a good sense of estimation for the Developer. There was a vertical slice of functionality that gave the core feature of fetching and displaying clips, as featured below. The Review began by going over the User Stories and Requirements between Product Owners and Developer. After testing results were accepted, a demo was presented showing the current state of Rekappa.

Figure 24 Sprint Review Demo for Iteration 1



This is the user interface after sprint I, which accomplishes the main feature of Rekappa of fetching media with a given Game title.

Figure 25 Alternative Twitch Search Queries for Iteration 1 Demo



These designs mimic the Wireframes. Redux handles the data, updates the React, and the CSS Flexbox properties were used to visually organize the fetched clips.

Sprint Retrospective

Iteration 1 accomplished the primary goal of establishing the ground work. There were risks taken in researching the new Twitch API, learning fundamental CSS to avoid using packaged libraries, and implementing the Redux Store in contrast with the prototype.

The story points felt appropriate, but didn't describe the uncertainty surrounding these new aspects. For the next Sprint, the goal will be to build upon this infrastructure and improve the search querying with filters. This would require more mature Reducers, which will be the code to filter the fetched data. The Twitch API doesn't support these kind of queries outside of finding the most popular clips in a certain window range. Thus these functions will lead to the client-side application to contain more business logic.

5.2 Iteration II

User Stories

Table 9 User stories for Iteration II

| Story ID | Story Description | Tasks | Priority | Story Points | Estimated Hours |
|----------|--|--|----------|--------------|-----------------|
| US-2 | As a User I want to search Clips by Broadcaster so I can watch their clips | Create FetchChannelTwitchClips functions | M | 3 | 4 |
| | | Add to Search Component | | | |
| | | Add Results Table | | | |
| US-3 | As a User I want to filter clips by Time so I can view clips in particular date ranges | Update Search Components | H | 5 | 3 |
| | | Update Fetch API | | | |
| US-4 | As a User I want to sort search results by Game | Create GameFilterReducer | M | 2 | 2 |
| US-5 | As a User I want to sort search results by Broadcaster ID | Create BroadcasterFilterReducer | M | 2 | 2 |
| US-6 | As a User I want to sort search results by Region | Create RegionFilterReducer | M | 2 | 2 |
| US-7 | As a User I want to sort search results by Followers Count | Create FollowersCountReducer | M | 2 | 2 |

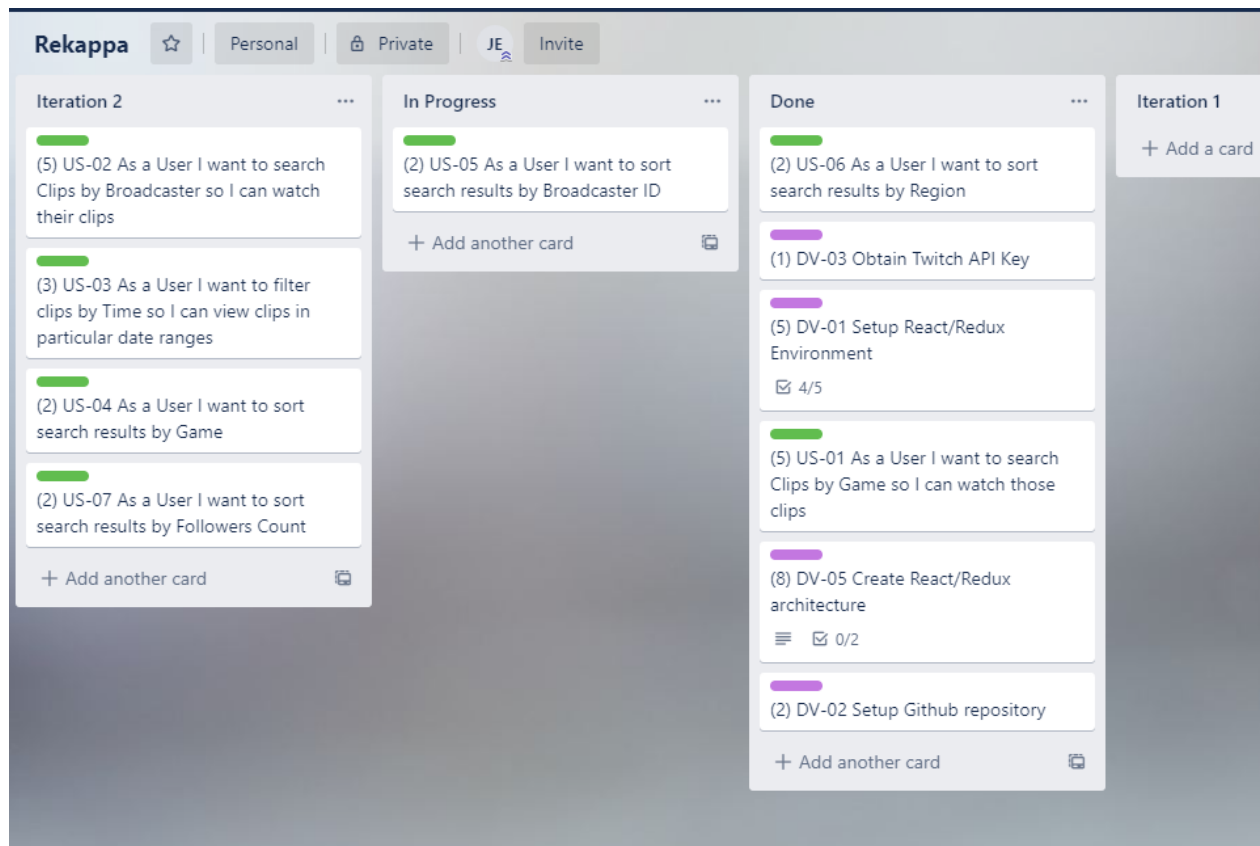
Iteration Planning

This iteration is primarily developing the entire infrastructure of Rekappa, by implementing the filtering abilities. Most of the features should be similar, particularly the stories about searching by Game, Regions, and Followers since those attributes are available in the Twitch API. However, filtering results by date windows involves some extra challenges regarding parsing Dates and windows. Additionally, there is concern of working with HTML input elements and how their behavior will affect Redux's state.

Daily Stand-Up Meetings

Iteration 2's daily stand-up meetings revealed extra challenges and technical issues with the stories.

Figure 26 Scrum board for Iteration II



Development Work

US-02

This feature took advantage yet again of Twitch's API for supporting language support. This fetches clips with particular languages – for example, English and Spanish clips can be specifically targeted with the query "en, es". Initially a drop down-menu was pursued but was deemed not essential towards the scope of the project. However, it is noted and the interface should massively improve after these set of Sprints.

US-03

Searching by a date range was also implemented similarly to US-05. The initial attempt was thought to employ a sorting algorithm manually implemented with a reducer, but Twitch's API end point already supported an option for date durations. These date durations are weekly, daily, monthly, and all clips. Implementation required making a drop down menu with the option input html tags and leveraging the API util method that was re-written to support parsed date durations.

US-04

This story was pushed into Iteration 3 due to it sharing similar functionality as simplifying querying the Twitch servers for a game. However, it'll be useful for sorting a Channel's clips between games.

US-05

This story was targeted first due to an initial impression how it would work. Originally, it was envisioned to be compiled automatically of all the available languages. Additionally, it was thought that it would be a filtering implementation, but Twitch.TV's API already supports an extra query for searching purely for a language. For example, using the query option *languages=ko* will return media that are in the Korean Language. The story enabled this flexibility of implementation, and it had to be revisited with the Project Owner in order to explore the user expectations of using the app. The component ended being a normal input form. Further enhancements would be a drop down menu that would automatically be compiled by search results, but better user stories are needed.

US-06

Implementation began by reviewing US-03 and US-02. A Reducer was examined, but the most difficult aspect was creating an input form. The language options aren't all available unfortunately, so manually entering the text was proposed and accepted by Development. After the input is collected, it is then sent as an extra query parameter in the API call to Twitch's clip endpoint.

US-07

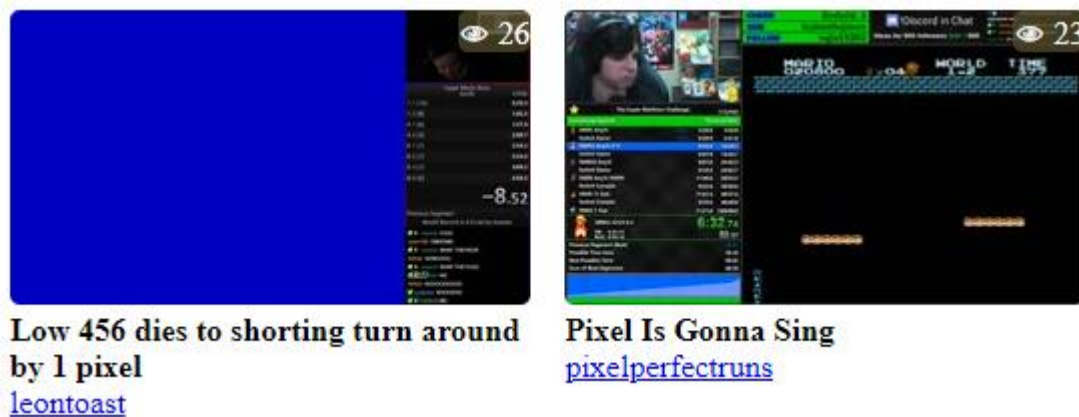
This story unfortunately was deemed unfeasible at this part of the project. The story was to establish the sorting feature, but the api request data model doesn't have a Follower value for the clips. The initial design relied on the attribute being present, therefore easy to sort. As it stands, implementation would require making another API call with the User api end point on Twitch to find follower count.

The story US-08 was pursued instead, since upon inspection the Views attribute is present and is an integer value.

US-08

The goal of this story is establish the second major feature of Rekappa, sorting by Views. However, the first major issue was creating a better display for each clip in the index. Previously, the Clip only showed the thumbnail. Since these sorting features are now present, the user now needs to be able to read that information they're sorting by.

Figure 27 Updated Clip Display



Note the Clip Title, Clip Channel, and views are now present in each clip. The Views total is in the upper right corner. With the View information present, implementing the sorting can begin within the Redux data-flow. The process of this development required reviewing sorting data within a React/Redux application. There seems to be valid implementation with Selector, and one with making it within a Actions method. Ultimately I've decided to go with the Actions method since Selector seems to operate with a different set of State-Derived data. This begins with adding from the component, then updating the Redux-data flow in order of dispatching actions, sorting the data with reducer helper methods, and updating the Rekapp Redux Store.

Figure 28 Adding function to the Component and Container

```
handleSortChange(e) {  
  20  
  e.preventDefault();  
  21  
  this.props.sortClipsBy(this.props.clips,  
  22  
  e.currentTarget.value);  
  23  
  24  
  25  
  26  
  27  
  28  
  29  
  30  
  31  
  32  
  33  
  34  
  35  
  36  
  37  
  38  
  39  
  40  
  41  
  42  
  43  
  44  
  45  
  46  
  47  
  48  
  49  
  50  
  51  
  52  
  53  
  54  
  55  
  56  
  57  
  58  
  59  
  60  
  61  
  62  
  63  
  64  
  65  
  66  
  67  
  68  
  69  
  70  
  71  
  72  
  73  
  74  
  75  
  76  
  77  
  78  
  79  
  80  
  81  
  82  
  83  
  84  
  85  
  86  
  87  
  88  
  89  
  90  
  91  
  92  
  93  
  94  
  95  
  96  
  97  
  98  
  99  
  100  
}
```

This sets up the general data-flow structure for introducing new states and data derivations. This process should be used for the next story as well, sorting by Channel.

Iteration Planning

Iteration III contains one of the larger features, automatically playing clips. US-07 was demoted to a lower priority after doing reviewing the value with Project Owners and Stakeholders. Although it would be beneficial to have Follower Count data, the work required to implement could allocated to higher value stories. This story will be investigated after the higher priority items are completed first.

Another considerable feature that departs largely from the main features is producing a list of search results. This was reviewed with Stakeholders, and this will enable content creators and curators to save a text file that can be used with other tools. From an implementation point of view, this would mean introducing a routing system or displaying another view.

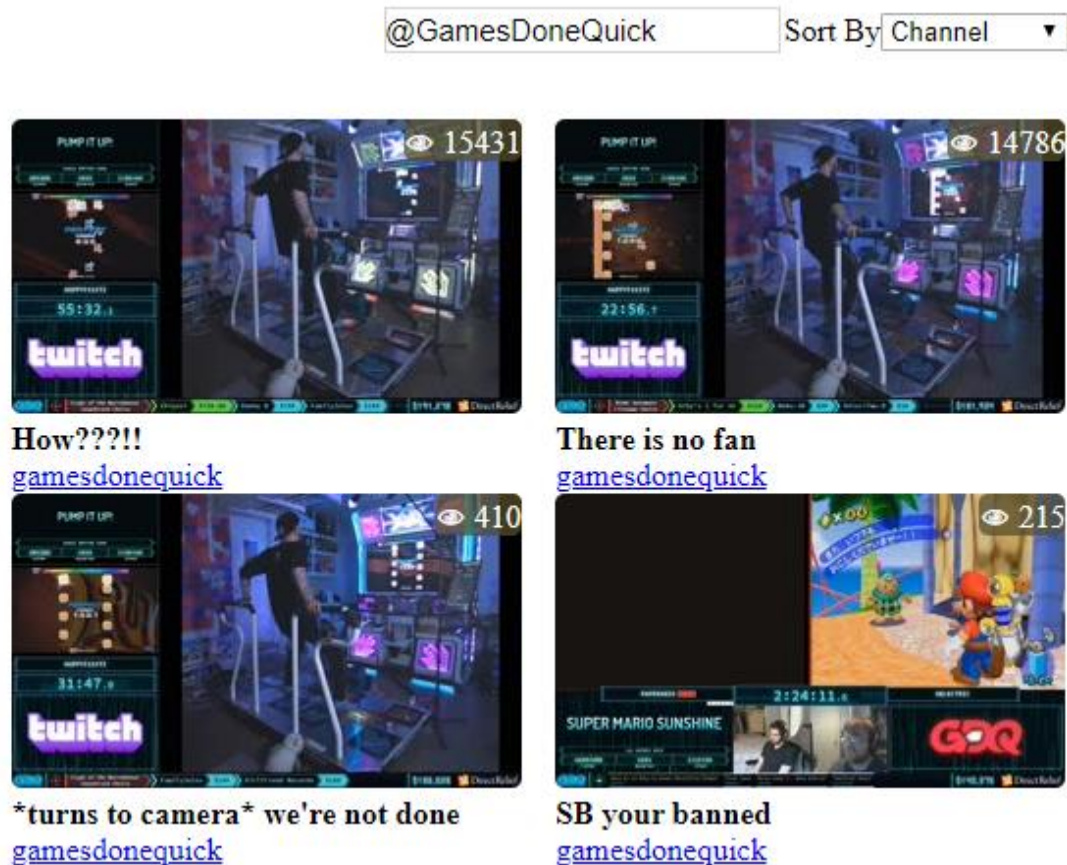
Testing

US-02

As a User, I want to search by Broadcaster or Channel name. This was tested by having implementing a special character to target a Channel or Broadcaster name in the Rekappa Search Input form. As it

stands, entering an input like “Tetris” will assume the search string will be a Game. However, by entering a @ in front of the text, it will prompt Rekappa to look for clips from a specific channel.

Figure 29 Searching by Channel Name

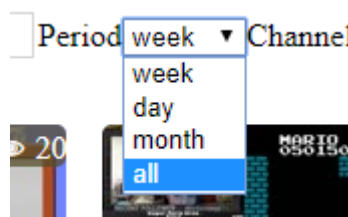


Success was confirmed when the results were indeed only clips from the specific channel. Note in the above figure how every channel is “GamesDoneQuick”.

US-03

As a User, I want to filter by Time Periods so I can view clips in particular date ranges. This was tested by having a functioning drop-down menu, and confirming a successful API response from Twitch.

Figure 30 Period Drop Down Menu



Afterwards, the API request and responses were examined to see if it behaved correctly. Below is successful calls querying for different time periods between week, month, day, and all.

Figure 31 Querying with Period options

| Name | Status | Type |
|--|--------|------|
| <input type="checkbox"/> top?game=Super%20Mario%20Bros.&limit=100&period=week | 200 | xhr |
| <input type="checkbox"/> top?game=Super%20Mario%20Bros.&limit=100&period=month... | 200 | xhr |
| <input type="checkbox"/> top?game=Super%20Mario%20Bros.&limit=100&period=day&l... | 200 | xhr |
| <input type="checkbox"/> top?game=Super%20Mario%20Bros.&limit=100&period=all&la... | 200 | xhr |

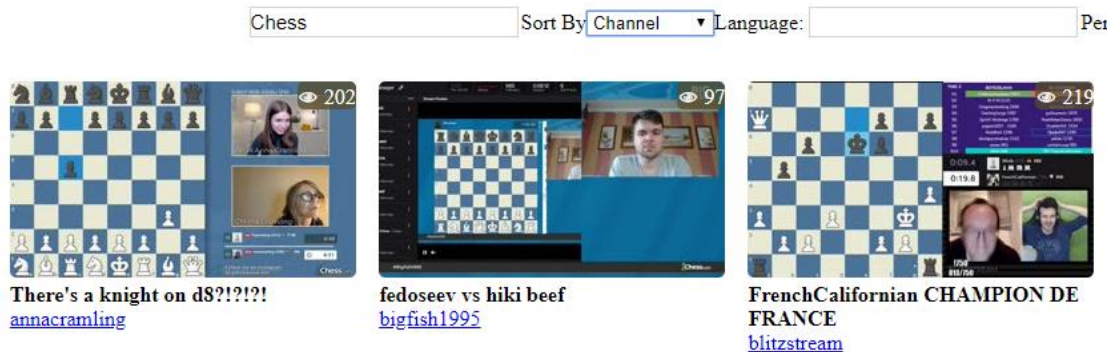
US-04

This story was pushed into Iteration 3 due to it sharing similar functionality as querying the Twitch servers for a game. However, it'll be useful for sorting a Channel's clips between games.

US-05

As a user I want to be sort search results by Broadcaster or Channel name. Sorting by Broadcaster name was tested by confirming results are re-displayed in a alphabetical order. First the option was created, then Redux updates the list. A successful sort is shown below.

Figure 32 Adding function to the Component and Container



US-06

As a I user, I want to search by Region. Very similar implementation to US-03. This was tested by having a functioning input form, and confirming a successful API response from Twitch. This time with querying options regarding language. After the input form was created, the data was captured and sent to API Util functions to query correctly.

US-07

This user story was pushed into the next iteration. Regardless it is considered failing for this iteration.

US-08

As a user I want to be able to sort search results by View Counts. Sorting by View Counts was performed similarly to US-05, sorting by Channel. The default sort is Most Views, as given from the Twitch API. Selecting *Least Views* should display Clips in order of smallest View counts first.

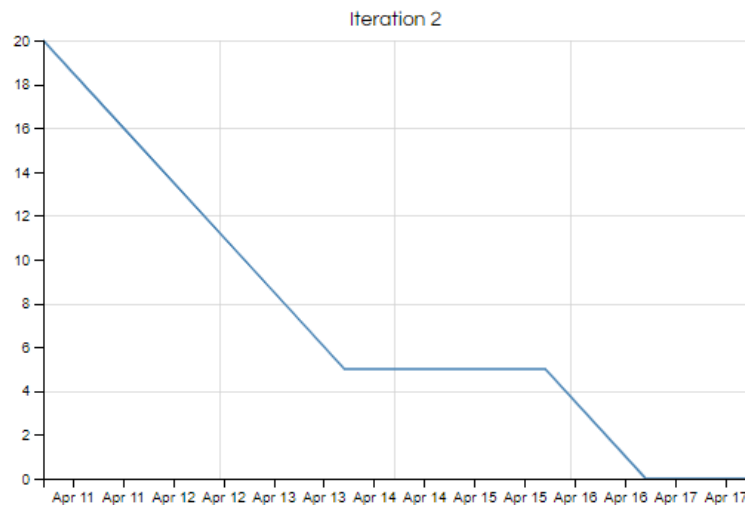
Figure 33 Adding function to the Component and Container



Sprint Burndown Chart

The Iteration II Sprint Burndown chart captures the difficulty I had implementing the sorting features into Rekappa. Progress was going fine, but tackling the Sorting features meant re-implementing user interfaces, adding more to the Redux data cycle, and investigating CSS issues.

Figure 34 Sprint Burndown Chart for Iteration II



Sprint Review

Iteration 2 Sprint Review was conducted very similarly to Iteration 1 Sprint Review. The User Stories was examined against the test results and requirements with the Product Owner and also other Stakeholders. This brought up different perspectives on how Rekappa will be used. By default it is intended to retrieve contents per game, which is useful a normal Streamer. However a larger Channel like GamesDoneQuick would need functionality that enables them to sort their content by Games. This was pushed into the next iteration.

The Product Owner is also concerned about the clarity of User Stories. In particular, the user stories should have been more clear about sorting and filtering. The differences led to conflicting architecture designs. The User Stories were revisited and re-created to consider these more.

Sprint Retrospective

Iteration 2 introduced several obstacles that I didn't predict when implementing the application. The first issue was the lack of Follower count per broadcaster. This isn't an essential component of the application, but it would've been a good feature to have for further analytics. Additionally, the index and item displays had to be revisited frequently to accommodate these features. Earlier implementations only showed the thumbnail, so sorting by Game or View counts weren't going to be easy to identify. This led to extra information and elements being introduced, along with CSS Styling.

Although the core feature was working within Redux, the user interface looked very primitive and difficult to read. This was solved by leveraging CSS styling, but this part of the sprint required much research and time investment. After the CSS styling was solved, then another obstacle in sorting Redux data arrived. I was confused about the role of Selectors in contrast with Actions, but I found a few articles and sample code online to confidently choose an implementation. With that data-flow implemented, adding extra filters or sorting shouldn't be as difficult.

5.3 Iteration III

User Stories

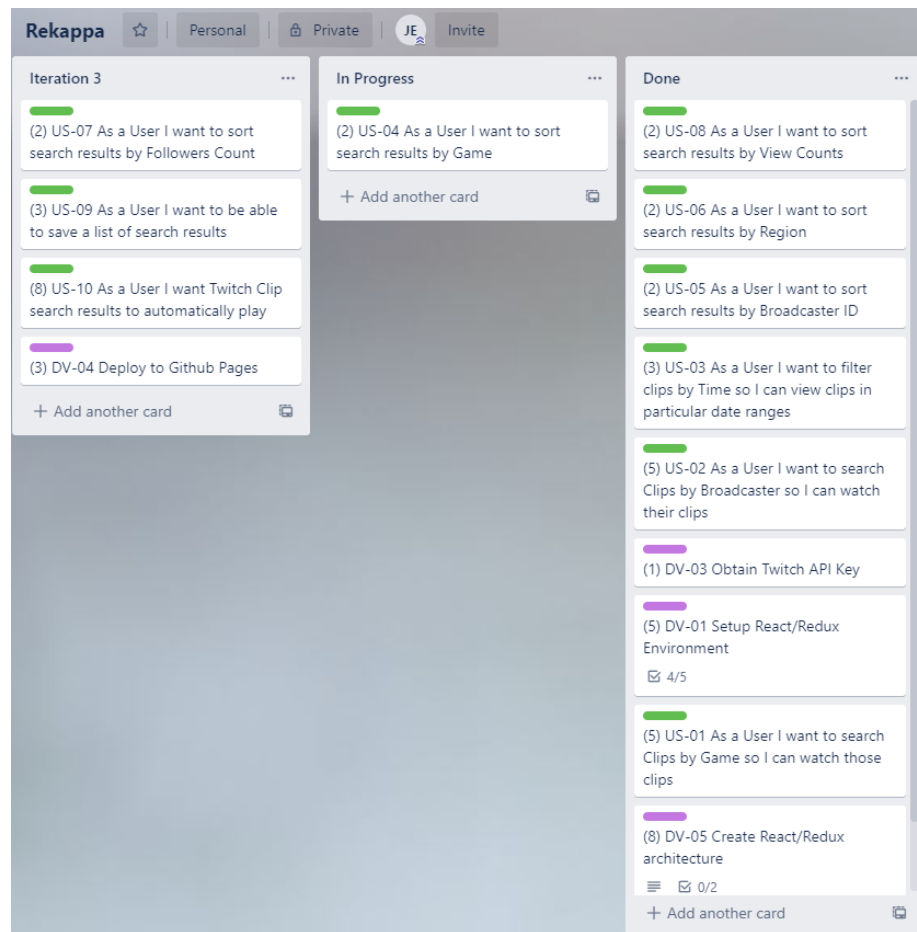
Table 10 User stories for Iteration III

| Story ID | Story Description | Tasks | Priority | Story Points | Estimated Hours |
|----------|---|--|----------|--------------|-----------------|
| US-04 | As a User I want to sort search results by Game | Create Game Option | M | 2 | 2 |
| | | Create Container method | | | |
| | | Create Sort Action for Game | | | |
| | | Create Action in Twitch Reducer | | | |
| US-07 | As a User I want to sort search results by Followers Count | Research API Documentation | L | 2 | ? |
| US-09 | As a User I want to be able to save a list of search results | Add Lists to Schema | M | 5 | 5 |
| | | Create function to parse state Lists into text | | | |
| | | Display Lists Results | | | |
| | | Create List View | | | |
| US-10 | As a User I want Twitch Clip search results to automatically play | Create Player Component | H | 5 | 8 |
| | | Create Player Component State | | | |
| | | Create a Player Management system | | | |
| | | Create Embedded Twitch CLip Component | | | |
| DV-04 | Deploy to Github Pages | create gh-pages branch | H | 3 | 2 |
| | | double-check operational settings | | | |

Daily Stand-Up Meetings

Iteration III's Daily Stand-Up Meetings revealed unforeseen issues in development. US-04 should be a straightforward process, but the remaining stories represent different technologies. The autoplaying component should be relying mainly on JavaScript's own timeout API, instead of relying on a React component. Developers noted that a routing system should become available, and it should be a developer or user story. There was heavy consideration in prioritizing the other features, then implementing a routing system after this Iteration.

Figure 35 Scrum board for Iteration III



Development Work

US-04

Development of US-04 was straightforward now that similar functionality has been implemented. First development began in the View, adding option input for *game* title. Then that was connected in the Twitch Search container, connecting the props to the Redux data-flow. Then actions were defined and the sorting algorithms were implemented, which involved the use of a comparator function. The sorting functions were the most difficult part, but fortunately using object attributes alleviated most of the work.

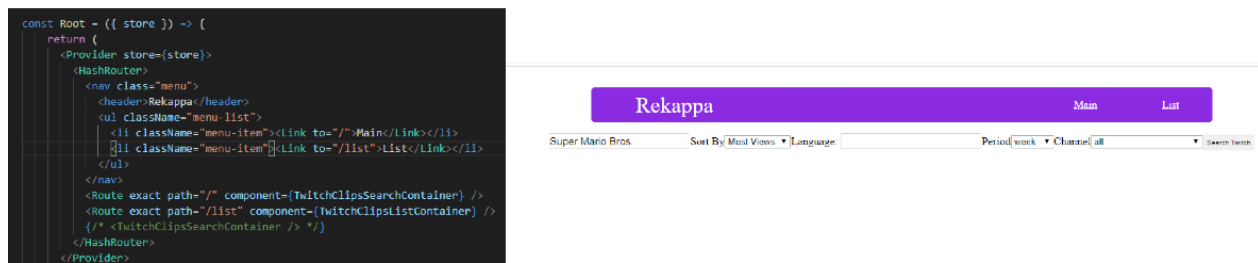
US-07

This story was carried on from the previous Iteration, due to unsupported attributes from the Twitch API. For this to work, Rekappa will have to make another API call to Twitch to receive Follower Count information per User. This will be a taxing and inefficient process, due to the multiple API calls required. Implementing it would have to rework the story and possibly a different User Flow. For example, perhaps it can be retrieved if the User requests, but that'd be a different feature altogether.

US-09

This story required significant additional development en route to complete the task. With the Wireframes in mind, there has to be a navigation menu or bar to swap between various views. One of these views is Lists, where a User can quickly list the item urls so they can copy them into another tool or save them as a text document. Prior to this story, there were no navigation bar. Subsequently, *react-router* was installed into the architecture and styling was applied.

Figure 36 Implementation of Navigation Menu



Now the application can support multiple views while being connected to the same data. The Main view will fetch the Twitch content, which can then be shared to the List view which outputs the clip urls.

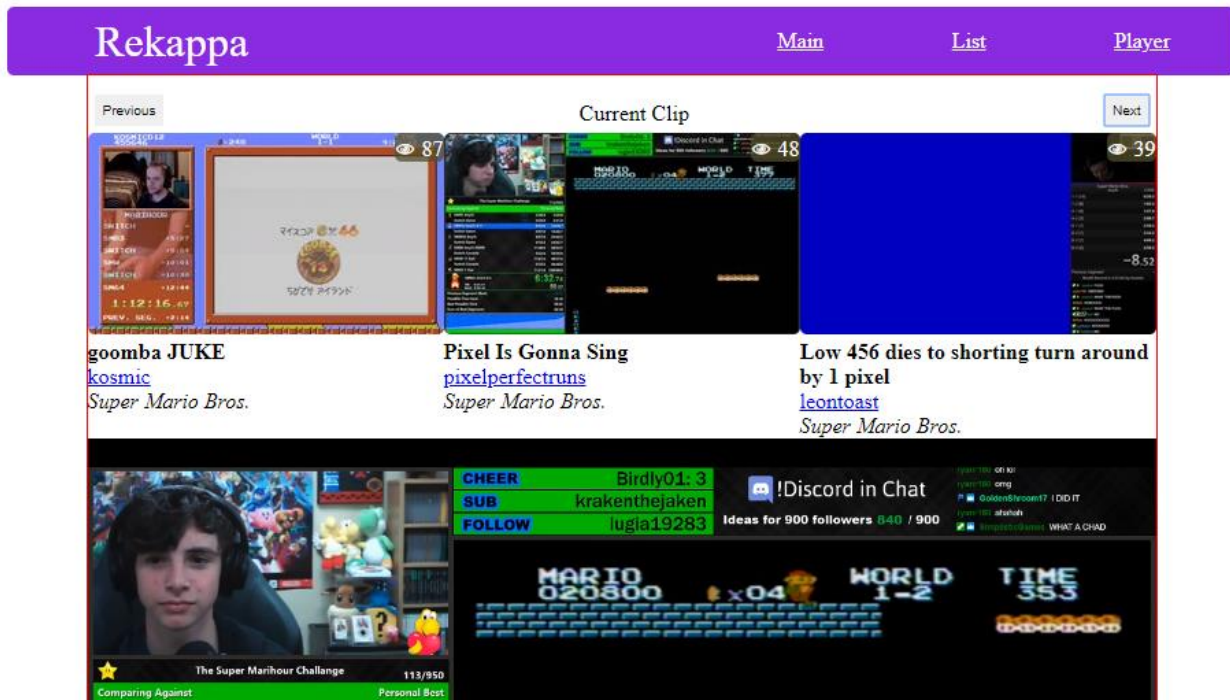
US-10

This story is took an unfortunately long time to develop. The most difficult issue was on how to implement the Clip Player's internal state – should it be with it's own controlled Component, outside of Redux? Or should it be connected to Redux's state and data-cycle? I read several opinions and code samples online, and opted to keep the Redux state simple and put the Clip Player state management within the component. This avoids adding unessential data updates to the Redux data-cycle.

Keeping track of the Clip Player state was still troublesome due to the changing nature of the Clips list. The user flow is that the Clip Player should play clips from the clips fetched in the Main view. Therefore whenever the lists is changed, the Clip Player should reset. This proved hard to test with *setTimeout*, as managing and tracking the current clip required more thorough edge cases.

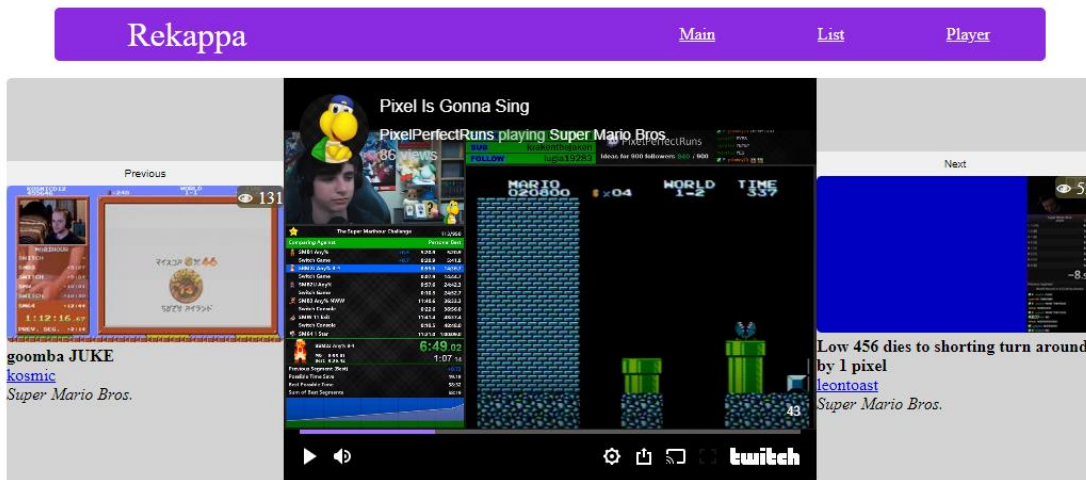
A new view was developed against the WireFrames, but styling with CSS was initially tricky given the Twitch's embedded html. The initial implementation for the Clip Player resulted with the buttons and preview clips above the player.

Figure 37 Initial View Layout



However, looking at the Twitch.tv's own implementation of current livestreams, I took inspiration with their layout and applied to the Clips Player. I think the final design is more intuitive than the initial one. The current clip is recognizable as the bigger and central clips, and the previous and next buttons are intuitive positions as well.

Figure 38 Final View Layout



DV-04

Implementation was much simpler than anticipated. Development anticipated setting up a host with Heroku, an online deployment platform. However, since this is client-side code, there are more options

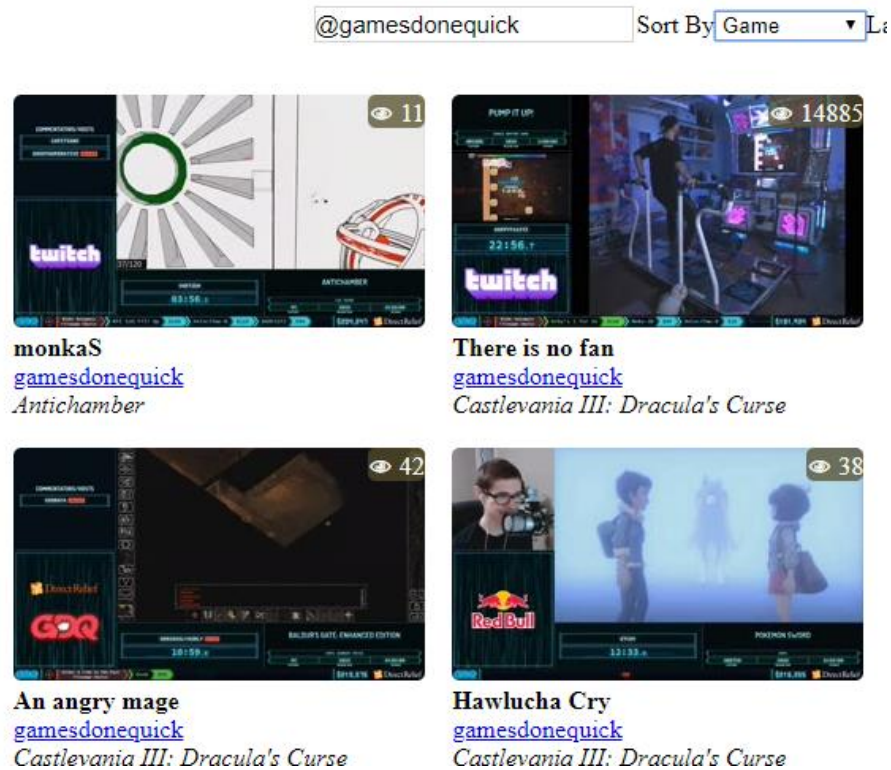
for hosting JavaScript. This turned out to be ideal for Github's Github Pages – a new branch titled “gh-pages” was created and served as the “production” branch. By pushing code there, Github will host the code and provide an automatic public URL.

Testing

US-04

The clips should be displayed by Game, and there should be Game name info in the item.

Figure 39 Successful Sort by Game



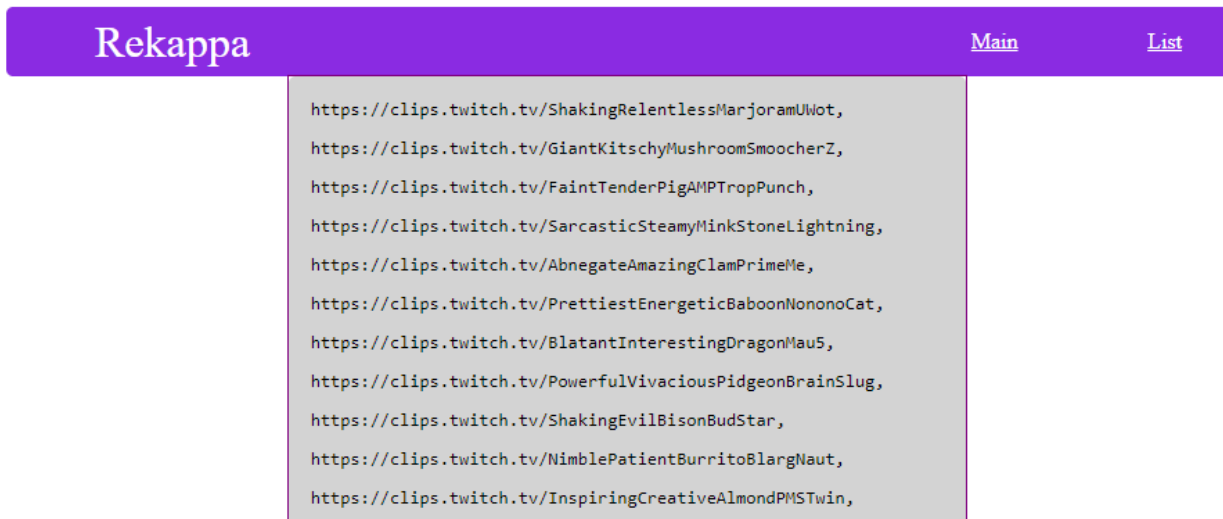
US-07

No testing can be performed since the functionality is deemed unfeasible within the remaining time. The feature will re-evaluated in terms of user flow, requirements, and priority. Discussions with the Project Owner and Development suggests that the feature is feasible if there are more user stories requesting it.

US-09

The User should be able to display the search results as text with their Clip Urls. This was deemed passing when Rekappa changed views accordingly.

Figure 40 List View



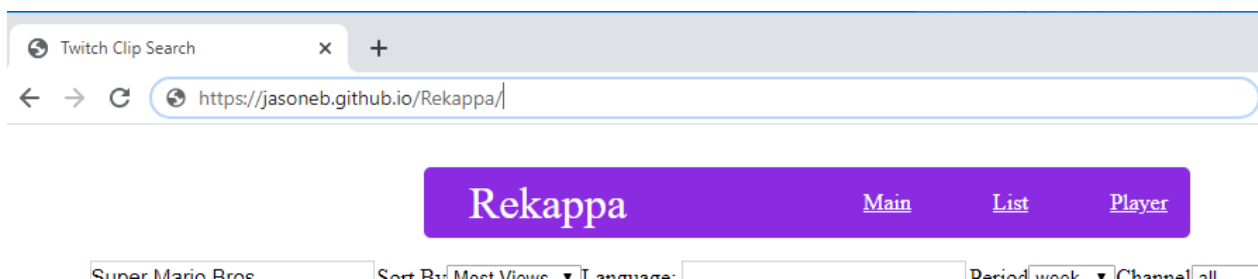
US-10

As a User Rekappa should automatically play my Clips I searched for. This was a successful tests in regards with functionality – it was deemed passing when the User was able to successfully search results, then automatically playing them with the *Player* link in the navigation bar. The clips should automatically switch into the next clip, which the Clip Player performs. Resetting and fetching new clips works as well.

DV-04

As a User, I should be able to visit Rekappa in my web browser. This was tested successful by navigating to the URL <http://jasoneb.github.io/Rekappa> and seeing the Rekappa application ready for usage.

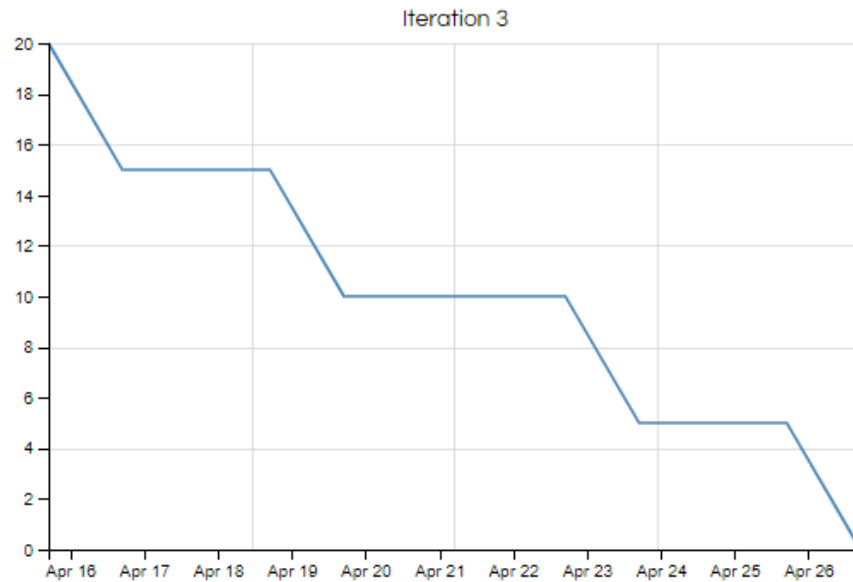
Figure 41 Rekappa deployed to gh-pages



Sprint Burndown Chart

Iteration III's burndown chart shows a more sporadic progression due to the new technologies implemented. The velocity is comparable, but implementing Clip Player statemanagement, new CSS styling, and adding to the Redux store with React Router led to more issues and requires a re-evaluation on user stories and requirements.

Figure 42 Sprint Burndown Chart for Iteration III



Sprint Review

The first part of the Sprint Review was conducted with the Developers under the leadership of the Product Owner. The Product Owner reviewed the user stories in Iteration III, in addition to the previous work to the items in Iteration II. Development raised insight and concern with the Clip Player user stories, due to how much additional time it took in regards to styling and implementation. The Stakeholders Streamers and Content Producers, Users of the targeted demographic, recognizes the functionality but believe the user interface is adequate. Development believes this has satisfied the stories, but ultimately the Product Owner feels that the presentation could be enhanced considerably.

Development demonstrated the user flow by navigating to the URL, entering a new search string, and examining the contents of *List* and *Player*. *List* showcased a plaintext collection of urls, but Stakeholders expressed they wish they can also save those URLs as a text file. The demo revealed unusual bugs as well. For example, if the User were to use the *previous* and *next* buttons to go forward in clips, then the Clip Player becomes inconsistent when it switches into new Clips. Rekappa crashed, and Stakeholders also requested that there should be an Error page for the end-user.

Nonetheless a full demonstration was provided and the functionality is well-received, with presentation and better user-flow being the more requested requirements.

Sprint Retrospective

This Sprint ended up being much more difficult than expected, and applied extra stress and pressure on Development. Story points weren't being completed, and progress wasn't linear. There were several layouts and implementations that were created and disregarded, however the iterations proved valuable in refining the project. Another issue is the User Stories score pointing could've been more generous with the estimation, since the Clip Player required significantly more investigation and research for the implementation.

Product Owner felt the Sprint Velocity rightfully felt slower, especially after last Sprint's success. Development however feels more confident and capable of implementing future enhancements. Development cites the lean nature of the architecture and minimal reliance on a styling library, like Bootstrap from Twitter or the Material UI from Google. Stakeholders feel like the Rekappa application could benefit from the presentational updates, which the Product Owners strongly agree. Development feels that styling is not its strength, so finding more personnel could help immensely.

Future Sprints and iterations will focus on refining the Rekappa application core. There remains more quality-of-life changes, such as a Responsive Design, working and testing mobile environments, refining the Clip Player, and expanding on the List view and functionality. The next major features would be to save List as text file, and enabling a Favoriting system so Users can pick Clips to automatically play and save.

6.0 Implementation

6.1 Organization of Source File Structure

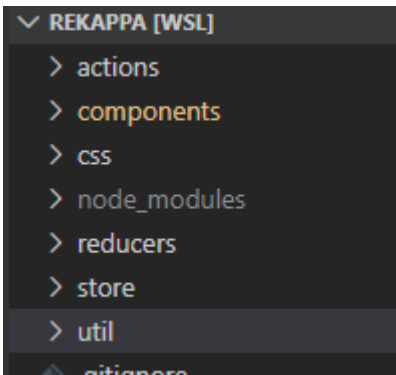
This section will highlight the Project Directory Structure and review the architecture and infrastructure for Rekappa. The operational environment was Linux, but it was developed with the Windows Subsystem Linux, which means it was ultimately built under Windows 10.

Figure 43 Application Directory Structure

| ; PC > mainDrive (C:) > Users > Jason > WebDev > Rekappa | | | |
|--|-------------------|----------|--|
| Name | Type | Size | |
| .git | File folder | | |
| actions | File folder | | |
| components | File folder | | |
| css | File folder | | |
| node_modules | File folder | | |
| reducers | File folder | | |
| store | File folder | | |
| util | File folder | | |
| .gitignore | GITIGNORE File | 1 KB | |
| { | File | 0 KB | |
| bundle | JavaScript File | 1,242 KB | |
| bundle.js.map | MAP File | 1,367 KB | |
| index | Chrome HTML Do... | 1 KB | |
| package.json | JSON File | 1 KB | |
| package-lock.json | JSON File | 215 KB | |
| twitch_clips_search.jsx | JSX File | 1 KB | |
| webpack.config | JavaScript File | 1 KB | |

The primary front-end client architecture is with Redux, with React handling the user interfaces. For Rekappa, there is a data being fetched from Twitch's API servers. This application's directory structure is

Figure 44 Application Directory Structure



The key folders are *actions*, *components*, *reducers*, and *util*. The *util* folder contains a library of helper methods for interacting with the Twitch API.

Figure 45 Twitch API Functions

```
util > JS api_util.js > [?] fetchGameClips
1  export const searchTwitchClipsByLoginName = loginName => {
2
3      let headers = { 'Client-ID': 'l622tc0oqzk3cso09retwcmw8rixl7',
4                      'Accept': 'application/vnd.twitchtv.v5+json' }
5
6      return $.ajax({
7          method: 'GET',
8          url: `https://api.twitch.tv/helix/users?login=${loginName}`,
9          headers: headers
10     })
11 };
12
13 > export const fetchGameClips = ([game, period='week', languages="", limit=100, cursor=""]) => { ...
25 }
```

The API request is a *GET* Request towards Twitch.TV's *Clips* endpoint. These requires authentication in order to receive more privileged access, namely in calling for more data. The Rekappa will respond to the fetched data on the *View* level, but actions are a Redux structure.

Figure 46 Action Dispatcher

```
actions > JS twitch_actions.js > ...
1  import * as APIUtil from '../util/api_util';
2
3  export const RECEIVE_SEARCH_GAME_CLIPS = 'RECEIVE_SEARCH_GAME_CLIPS';
4  export const REQUEST_SEARCH_GAME_CLIPS = 'REQUEST_SEARCH_GAME_CLIPS';
5
6  export const fetchSearchTwitchClipsByGame = searchTerm => dispatch => {
7    return APIUtil.fetchGameClips(searchTerm)
8      .then(resp => dispatch(receiveSearchGameClips(resp.clips)))
9  };
10
11 export const receiveSearchGameClips = clips => {
12   return {
13     type: RECEIVE_SEARCH_GAME_CLIPS,
14     clips
15   }
16 };
17
```

Above is sample code showing the *Actions*. The API fetches Game Clips then dispatches it to Redux. These Action objects are sent to the *Store*, which is equivalent to the Model of the Model-View-Controller pattern.

Figure 47 Rekappa Store Schema

| | |
|---|--|
| <pre>store > JS store.js > ... 1 import { createStore, applyMiddleware } from 'redux'; 2 import thunk from 'redux-thunk'; 3 import rootReducer from '../reducers/root_reducer'; 4 5 const configureStore = () => { 6 return createStore(rootReducer, applyMiddleware(thunk)); 7 }; 8 9 export default configureStore;</pre> | <pre>reducers > JS root_reducer.js > ... 1 import { combineReducers } from 'redux'; 2 3 import twitchReducer from './twitch_reducer'; 4 5 export default combineReducers({ 6 clips: twitchReducer 7 });</pre> |
|---|--|

The *Store* performs very similarly to a key-value data structure, or a JavaScript Object. The code on the left wraps the code on the right – note the *clips* key in the *twitchReducer* section. This specifically defines an attribute on the Store Schema, which is the core structure of Redux. The *Reducers* filters and reduces the data, thus its name. These Redux components are all wired together and subscribed to the data changes. Each time data or state changes, it alerts all the components in sequence.

The data-flow culminates in triggering the *View*, which are the React Components that make the user interface for Rekappa.

Figure 48 Rekappa Root Component

```
components > root.jsx > [Root] Root
1 import React from 'react';
2 import { Provider } from 'react-redux';
3
4 import TwitchClipsSearchContainer from './
5
6 const Root = ({ store }) => {
7   return (
8     <Provider store={store}>
9       <TwitchClipsSearchContainer />
10    </Provider>
11  );
12 };
13
14 export default Root;
```

```
components > twitch_clips_search.jsx > ...
1 import React from 'react';
2
3 import TwitchClipsIndex from './twitch_clip_index';
4
5 class TwitchClipsSearch extends React.Component {
6   constructor() {
7     super();
8     this.state = { searchTerm: 'Super Mario Bros.' };
9     this.handleChange = this.handleChange.bind(this);
10    this.handleSubmit = this.handleSubmit.bind(this);
11  }
12
13   componentDidMount() {
14     this.props.fetchSearchTwitchClipsByGame(this.state.searchTerm);
15   }
16 }
```


Main View Component

The component has a container component that serves as the parent component to separate concerns. This connects the Redux data-flow to the presentational component Twitch Clip Search.

Figure 49 Container component for Twitch Clips Search

```
components > twitch_clips_search_container.jsx > mapStateToProps > clips
1  import { connect } from 'react-redux';
2  import TwitchClipsSearch from './twitch_clips_search';
3  import * as twitchActions from '../actions/twitch_actions';
4  import * as sortActions from '../actions/sort_actions'
5
6  const initialState = {
7    clips: [],
8    channel: ""
9  }
10
11  const mapStateToProps = (state = initialState) => {
12    return { clips: state.clips }
13  };
14
15  const mapDispatchToProps = dispatch => {
16    return {
17      fetchSearchTwitchClipsByGame: (searchTerm, languages, period) =>
18        dispatch(twitchActions.fetchSearchTwitchClipsByGame(searchTerm, languages, period)),
19      filterClipsByChannel: (clips, channel) => {
20        return dispatch(twitchActions.filterClipsByChannel(clips, channel))
21      },
22      sortClipsBy: (clips, sort_type) => {
23        return dispatch(sortActions.sortClipsBy(clips, sort_type))
24      },
25      resetClips: (clips) => dispatch(twitchActions.resetFetchedClips(clips))
26    };
27  };
28
29  export default connect(
30    mapStateToProps,
31    mapDispatchToProps
32  )(TwitchClipsSearch);
33
```

The presentational component Twitch Clips Search is compromised of a Search Bar and an unordered list to make the Clips Index.

Figure 50 Render Template for Twitch Search

```
components > twitch_clips_search.jsx > TwitchClipsSearch >
69
70     return channelsList;
71   }
72
73   render() {
74     let { clips } = this.props;
75
76     return (
77       <div className="twitch_clip_search">
78 >       <form className="search-bar">...
108       <TwitchClipsIndex clips={clips} />
109     </div>
110   );
111 }
112 }
113
114 export default TwitchClipsSearch;
115
```

List View Component

The List View Component is comprised of the Container parent component, in order to connect Redux's store.

Figure 51 List Container Component

```
1  import { connect } from 'react-redux';
2  import * as twitchActions from '../actions/twitch_actions';
3  import * as sortActions from '../actions/sort_actions';
4  import TwitchClipPlayer from './clips_player';
5
6  const initialState = {
7    clips: []
8  }
9
10 const mapStateToProps = (state = initialState) => {
11   return { clips: state.clips }
12 };
13
14 > const mapDispatchToProps = dispatch => { ...
26 };
27
28 export default connect(
29   mapStateToProps,
30   mapDispatchToProps
31 )(TwitchClipPlayer);
32
```

The List View Component is an unordered list with the *pre* tag, to output the urls.

Figure 52 List Template

```
render() {
  let { clips } = this.props;
  return (
    <div className="twitch_clip_list">
      <ul className="clip_list">
        {clips.map( (clip) => <li>
          <pre>
            {clip.url.replace("?tt_medium=clips_api&tt_content=url", "")},
          </pre>
        </li>)}
      </ul>
    </div>
  );
}
```

Clip Player Component

The Clip Player Component template also has a container component in order to retrieve clips from the Main View.

Figure 53 Clip Player Container

```
components > clips_player > clips_player_container.jsx > initialState > clips
1  import { connect } from 'react-redux';
2  import * as twitchActions from '../actions/twitch_actions';
3  import * as sortActions from '../actions/sort_actions';
4  import TwitchClipPlayer from './clips_player';
5
6  const initialState = {
7    clips: []
8  }
9
10 const mapStateToProps = (state = initialState) => {
11   return { clips: state.clips }
12 };
13
14 > const mapDispatchToProps = dispatch => { ...
26 };
27
28 export default connect(
29   mapStateToProps,
30   mapDispatchToProps
31 )(TwitchClipPlayer);
32
```

The internal state management of the Clip Player is separate from the Redux store. The Clip Player template draws upon the *iframe* html tag, which is the embedded url for Twitch's Clips player.

Figure 54 Clip Player Methods

```
class ClipPlayer extends React.Component {
  constructor() {
    super();
    this.state = {
      idx: 0
    };

    this.handleControls = this.handleControls.bind(this);
    this.incrementIdx = this.incrementIdx.bind(this);
    this.decrementIdx = this.decrementIdx.bind(this);
    this.setClip = this.setClip.bind(this);
    this.timer = null;
  }
}
```

These methods help operate the autoplay of the fetched Clips. It leverages the Twitch Clip's query points for its iframes within the embedded html. The template wraps the iframe in its own div tag.

Figure 55 Clip Player Render Template

```
79 render() {
80   let { clips } = this.props;
81   let { idx } = this.state;
82   let currentClip = clips[idx];
83   let prevIdx = (idx - 1 <= 0) ? clips.length - 1 : idx - 1
84   let prevClip = clips[prevIdx]
85   let nextIdx = (idx + 1 > clips.length - 1) ? 0 : idx + 1
86   let nextClip = clips[nextIdx]
87
88   return (
89     <div className="twitch_clip_player">
90       <div className="prev-clip">
91         <button className="prev-button" name="prev" onClick={this.handleControls}>Previous</button>
92         <TwitchClipItem clip={prevClip} />
93       </div>
94       <div className="player">
95         <TwitchClipItemIframe clip={currentClip} options={{autoplay: true}} />
96       </div>
97       <div className="next-clip">
98         <button className="next-button" name="next" onClick={this.handleControls}>Next</button>
99         <TwitchClipItem clip={nextClip} />
100       </div>
101     </div>
102   );
103 }
104 }
```

6.2 Reference List of Files

This shows the files per directory in the project folder.

Items in bold denote folders, with hyphenated items being located in those folders. Underscored denotes directories and their respective items.

Root

Actions

- twitch_clips_search.jsx
- twitch_clips_search_container.jsx

Components

reducers

clips_player

- clips_player.jsx
- clips_player_container.jsx

util

twitch_clips_list

- twitch_clips_list.jsx
- twitch_clips_list_container.jsx

css

- bundle.
- package-lock.json
- webpack.config.js
- bundle.js.map
- index.html
- package.json
- twitch_clips_search.jsx

Reducers

- clips_selector.js
- root_reducer.js
- twitch_reducer.js

Actions

- sort_actions.js
- twitch_actions.js

Util

- api_util.js

Components

- root.jsx
- twitch_clip_index.jsx
- twitch_clip_item.jsx
- twitch_clip_item_iframe.jsx

CSS

- clips_list.css
- clips_player.css
- navbar.css
- rekappa.css
- twitch_clips.css

When the application is loaded, the Document Object Model (DOM) is loaded into the browser and trigger the entry file, *twitch_clips_search.jsx* in the parent directory. From there, the *root* node is taken over by React and begins initializing the rest of the environment. The Redux Store and Router is setup next as the parent container for all components, allowing access to the entire application's state and keep track of browser navigation. Finally, the main route is loaded into the Main view, with the parent container component *twitch_clips_search_container*.

7.0 Test and Integration

7.1 Acceptance Testing

Primitive quality assurance and testing was performed by development, but acceptance testing was conducted by the Stakeholders under the Project Manager. They were scored per iteration, and collected throughout the sprints. Acceptance testing differs in regards to more consideration towards the requirements.

Table 11 Acceptance criteria and acceptance test results

| Story ID | User Story | Acceptance Criteria | Results | Notes |
|----------|--|---|---------|---|
| DV-01 | As a developer, I want a React/Redux Environment to build the application with | <ul style="list-style-type: none">- Project Directory is created- Webpack is setup- Redux Boilerplate is setup | PASS | |
| DV-02 | As a developer, I want a Github repository to version control and save code | <ul style="list-style-type: none">- git version control is setup- github online repository is setup | PASS | |
| DV-03 | As a developer, I want a Twitch API key in order to use the Twitch API | <ul style="list-style-type: none">- twitch API key- twitch client id key | PASS | |
| DV-04 | As a developer, I want Github Pages to deploy to | <ul style="list-style-type: none">- heroku server setup- config file- gh-pages branch setup | PASS | |
| DV-05 | As a developer, I want React/Redux architecture to organize the project | <ul style="list-style-type: none">- Redux Store schema drawn out- Wireframes | PASS | |
| US-01 | As a User I want to search Clips by Game so I can watch those clips | <ul style="list-style-type: none">- App displays a search bar- App displays input options- App displays results in a grid | PASS | |
| US-02 | As a User I want to search Clips by Broadcaster so I can watch their clips | <ul style="list-style-type: none">- App displays result items with broadcaster name- App has an input menu for different Broadcaster names- App correctly filters results according to selected names | PASS | - no input menu was shown, but rather a drop-down menu. |
| US-03 | As a User I want to filter clips by Time so I can view clips in particular date ranges | <ul style="list-style-type: none">- App displays a drop-down menu for different time periods- App correctly retrieves clips based on time periods | PASS | |

| | | | | |
|--------------|---|---|------|--|
| US-04 | As a User I want to sort search results by Game | <ul style="list-style-type: none"> - App displays a drop-down meny for sorting by Game name - App re-organizes clips by Game name alphabetically | PASS | |
| US-05 | As a User I want to sort search results by Broadcaster ID | <ul style="list-style-type: none"> - App displays a drop-down meny for sorting by Broadcaster name - App re-organizes clips by by Broadercaster name alphabetically | PASS | |
| US-06 | As a User I want to sort search results by Region | <ul style="list-style-type: none"> - App displays a drop-down meny for sorting by Language name - App correctly displays clips based on Language | PASS | Story originally listed Region as an attribute, changed to Language during development |
| US-07 | As a User I want to sort search results by Followers Count | <ul style="list-style-type: none"> - App displays follower count per broad caster - App displays a drop-down meny for sorting by Follower Count per broadcaster - App re-organizes clips by broadcaster follower count | FAIL | Feature implementation was not feasible for this phase |
| US-08 | As a User I want to sort search results by View Counts | <ul style="list-style-type: none"> - App displ,ays view counts per clip - App displays a drop-down meny for sorting by Game name - App re-organizes clips by Game name alphabetically | PASS | |
| US-09 | As a User I want to be able to save a list of search results | <ul style="list-style-type: none"> - App Should have a link for List view - App should display search results in a plain text format for copying | PASS | |
| US-10 | As a User I want Twitch Clip search results to automatically play | <ul style="list-style-type: none"> - App should have a link for Player view - Player should automatically play clips - Clips played should be from search results | PASS | Design was changed from the initial wireframe |

7.2 Results

The Product Backlog planned for Iterations 1, 2, and 3 were largely completed with accepted testing. User Stories were adapted as requirements and features were re-evaluated after the Sprint Review. For example, US-10 had an initial design of having the player above the clip results, but Product Owner felt it would be more appropriate as a front-page design. Ultimately, the User Stories were initially short-sighted, but Agile methodologies encouraged flexible and adaptive approaches. This led to US-07 being de-prioritized, re-evaluated, and it led to US-10 to be changed from its initial design.

There were concerns about using “Developer” stories to have as tasks for the Product Backlog, but ultimately Project Manager felt this was appropriate. Developer User Stories helped build a healthy developing environment around the Rekappa project. Although those tasks could’ve been done in the pre-planning, this project wanted to document the first few steps of setting up an environment. This has been a practice observed in previous technology companies, which allowed future development to have a good reference for how the technology stack was initiated.

The main story to have failed is US-07. This is primarily due to a misconception on what data was available from the Twitch API endpoint. This is still considered a valuable story to address, since it’ll enable more robust querying with the Follower Count attribute. Follower Count implies a very large and popular Broadcaster or Channel, where a tool like Rekappa can excel in querying.

8.0 Release

End-user documentation was created after the three iterations were completed and reviewed. The documentation delivered to the stakeholders include installation instructions, section 8.1, and operation instructions, section 8.2.




8.1 Installation Instructions

Rekappa is available to and deployed at the url <https://jasoneb.github.io/Rekappa>. To install on a local machine, the following instructions should guide a user with a Linux environment. OSX Environments already support Linux, but Windows requires additional packages installation which aren't covered in this report. It is assumed that the user knows how to setup their own operation environment.

- 1.) Go to the Github Repository address, <https://github.com/JasonEb/Rekappa>
- 2.) Click "Clone or download"



- 3.) Extract folder
- 4.) Navigate to folder directory with Bash/Shell
- 5.) Run "npm install"
- 6.) Run "npm run start"
- 7.) Within the folder directory, open "index.html"

| name | type | size |
|---|-------------------|----------|
|  bundle.js.map | MAP File | 1,367 KB |
|  index | Chrome HTML Do... | 1 KB |
|  package.json | JSON File | 1 KB |

- 8.) Rekappa should open up in web browser

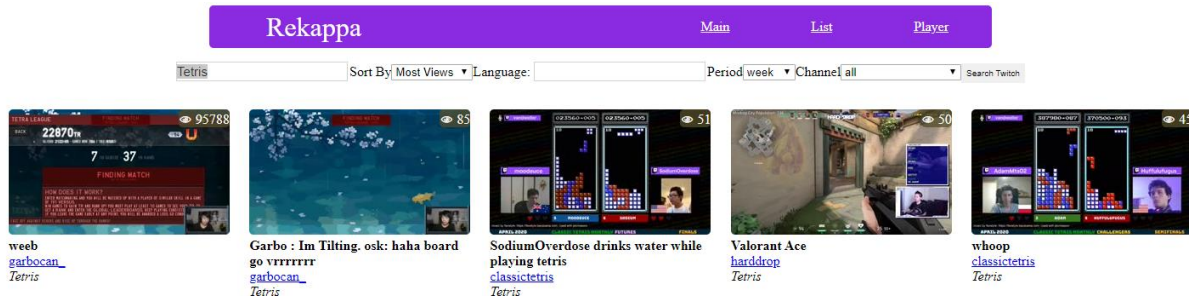
8.2 Operating Instructions

End User Operation

Rekappa will automatically search for “Super Mario Bros.” as a default Game, retrieving the most popular clips.

The User shall enter a new Search parameters for new content or set sorting parameters.

Figure 56 Default view for Rekappa



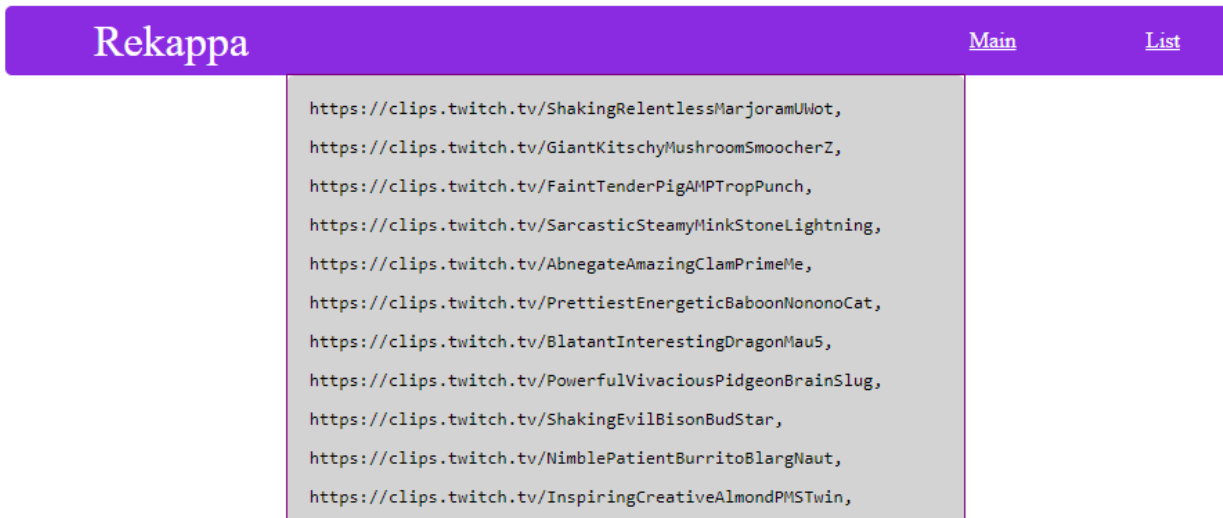
Examining the search result, each item contains hyperlinks to the clip’s Twitch channel or page.

Figure 57 Item View per Search Result



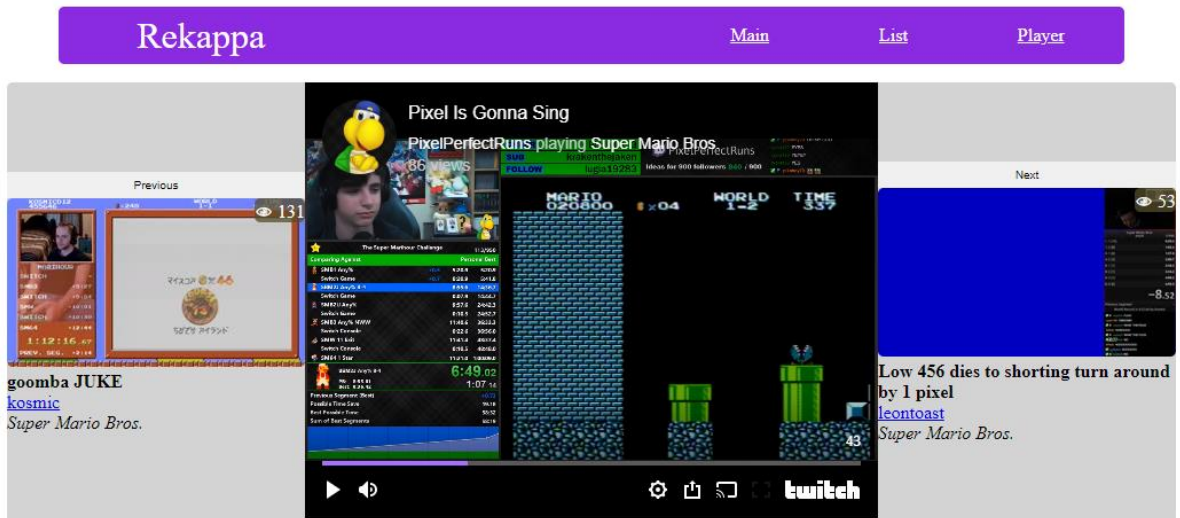
The User can click on the thumbnail and be taken to the Twitch Clip page for more information. The User can click on the channel name highlighted and underlined to be taken to the Twitch Channel page. The User can view a list of urls by clicking on link in the main navigation bar at the top.

Figure 58 List View



The list view provides the User an easier means of collecting urls in a plain text format. Finally, the User can view a stream of clips by automatically playing them in the “Player” link.

Figure 59 Clip Player View



The user can now enjoy their search results playing in the middle. The left “Previous” button will go back 1 clip, and the right “Next” button will go forward one clip. The clips will play sequentially according to the search results. The preview pictures will also show the previous and next clips respectively.

9.0 Recommendations for Enhancement

1. *Combine Clip Player and Twitch Search Container.* This was the original design, but the Clip Player took much of this own effort. It should be easily implemented due to the component nature of React/Redux. However, this does rely on improving the next enhancement.
2. *Move Clip Player state into the Redux Store Schema.* Currently Rekappa's Clip Player exists outside of the Redux state management. By combining them, fetching and caching clips will also update the current index of the current clip.
3. *Add favoriting clips.* As a User, I want to be able to pick clips to play in the Clip Player. As it stands, it plays search results, but there are hundreds of search results. It would be good to pick from the search results like a form of queue.
4. *Add sorting options by Creation Date.* The current time periods and sorting features are due to View Count. This is the main sorting option to pursue in order to have better insight with the search results. For example, newer clips may not have high View Counts yet but still can be trending.
5. *Add Twitch Integration.* As a User, I want to be able to view clips from my favorited channels. This feature was realized after further inspection with the Twitch API. Conceivably, the User will login with Twitch's authentication flow, and their profile and user information should be able to be read by Rekappa. Rekappa can then compile clips and make a list based on the User's favorited channels.
6. *Add Mobile Layout.* As a User, I want to be able to view Rekappa on a smartphone or tablet. Currently it is viewable, but not optimized.
7. *Add Playlist Making.* As a User, I want to be able to make my own playlists of clips. This is a feature not realized in many other platforms. This will allows more content to be collected into a montage for Content Producers.
8. *Improve Presentation and UI.* This will require consultation from external experts, or a skillset improvement from Development.

10.0 GitHub Repository

All client-side JavaScript code can be downloaded from Github. The project can be cloned via Git or downloaded as a ZIP archive from the url <https://github.com/JasonEb/Rekappa>.

Figure 60 Github Repository

The screenshot shows the GitHub repository page for JasonEb / Rekappa. The repository is described as a "Twitch Clips search application, with React+Redux". It has 16 commits, 6 branches, 0 packages, 0 releases, 1 environment, and 0 contributors. The latest commit is d6744ea, made yesterday. The repository contains three files: actions, components, and css. The actions file adds sorting by game, components adds navbar, and css adds styling.

JasonEb / Rekappa

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security 0 Insights Settings

Twitch Clips search application, with React+Redux

Manage topics

16 commits 6 branches 0 packages 0 releases 1 environment 0 contributors

Branch: master New pull request

Create new file Upload files Find file Clone or download

JasonEb adds styling Latest commit d6744ea yesterday

| | | |
|------------|----------------------|-----------|
| actions | adds sorting by game | yesterday |
| components | adds navbar | yesterday |
| css | adds styling | yesterday |

11.0 Concluding Remarks

Documentation has always been important to me because of my history as a back-end developer who specialized in API documentation. I was tasked with analyzing legacy code with very minimal documentation in previous jobs, so I understand and appreciate the value of technical documentation. This capstone project ended up being very valuable to me because it embodies both modern industry practices and academic reporting.

I was familiar with Agile methodologies in my time at Silicon Valley, but exploring the Southern Californian technology industry has been significantly different. It is different in terms of technology stacks and expectations from developers. I feel that Agile methodologies haven't been as widely adopted in Southern California, where it seems to be very popular in San Francisco. Meeting with other developers who worked in technology companies outside of the start-up scene, it was interesting to note the difference in culture and conduct.

Developing this project side by side with the report was a great process because it allowed me to exercise my writing skills with my coding abilities. I feel I was able to represent my results and thought process well with this report. Before I'd just write my tests results and satisfy test suites. But this report enabled me to take screen-shots, add figures, and convey my work process much better. I feel like that would be very valuable in presenting myself as software developers to future employers.

12.0 Acknowledgements

I would like to thank Cal State Fullerton and its Masters of Software Engineering program for their personnel and curriculum. Dr. Chang-Hyun and Dr. Ning Chen guided my career development with projects that exercised higher-level software engineering and management. All of the program culminated in this capstone project that combines my previous start-up careers with my new software engineering abilities.

I would also like to show appreciation for Twitch.tv's online platform as a entertainment hub and development platform. Their web API's have been more than generous and easy to use.

I would like to finally thank my previous software bootcamp, App Academy, for instilling and training me to be a full-stack developer. Their curriculum nurtured the front-end skills necessary to build Rekappa.

Bibliography

- [1] grokonez, Author. "Redux Reducer Example - Filter & Sort Data " Grokonez." Grokonez, 10 Apr. 2018, grokonez.com/frontend/redux/redux-filter-sort-example-redux-reducer-example.
- [2] Pak, Brian. "React-Redux Flow, Terminologies, and Example." DEV Community, DEV Community, 17 May 2019, dev.to/bouhm/react-redux-flow-terminologies-and-example-104b.
- [3] "Redux." Redux, redux.js.org/advanced/usage-with-react-router.
- [4] "Trello." Trello, trello.com/.
- [5] "Twitch Revenue and Usage Statistics (2020)." Business of Apps, 04 Apr. 2020, www.businessofapps.com/data/twitch-statistics/.
- [6] "Twitch Statistics & Charts." TwitchTracker, twitchtracker.com/statistics.
- [7] California State University, Fullerton, Department of Computer Science. 2009. *Master of Science in Software Engineering Handbook*. Fullerton, CA.