Retirement Planner

Peace of mind from knowing saving and investing will grow over a lifetime.

Contents

[Group Identity 2](#_Toc70446959)

[Proposal 3](#_Toc70446960)

[Project Requirements 3](#_Toc70446961)

[Data Structures 4](#_Toc70446962)

[Node Map 5](#_Toc70446963)

[Application Workflow 6](#_Toc70446964)

[The Model Controls 6](#_Toc70446965)

[The Filter Bar 7](#_Toc70446966)

[The Results Window 7](#_Toc70446967)

[The Add/Edit Person Window 7](#_Toc70446968)

[Features 8](#_Toc70446969)

[Project Planning 8](#_Toc70446970)

[Test Plan 9](#_Toc70446971)

[Development changes throughout the project 11](#_Toc70446972)

[Conclusion 11](#_Toc70446973)

# Group Identity

Software Systems Capstone, CSPC 49200-001, Group #8

Group members:

* Brock Herion
* Carson Christ
* Robert Bilbee

# Proposal

The concept for this application is a locally installed application that stores financial data related to a person’s retirement investments. The application will use the investment data to model the growth of the investments into retirement. The included tax and inflation calculations will let the user gauge the amount of income available to them in retirement and how the withdraws effect the balances of their investment accounts.

# Project Requirements

Development IDE: [VSCode](https://code.visualstudio.com/)

Platform: [NodeJS](https://nodejs.org/), [Electron](https://www.electronjs.org/), [JavaScript](https://en.wikipedia.org/wiki/JavaScript)

Web development libraries: [Redux](https://redux.js.org/), [Bootstrap](https://getbootstrap.com/), [Bootswatch](https://bootswatch.com/)

The project is being built using VS Code as our IDE, with the basis of the project being built on Node.JS. NodeJS handles all the package management for libraries we have included in the project and the interaction between our application and the operating system.

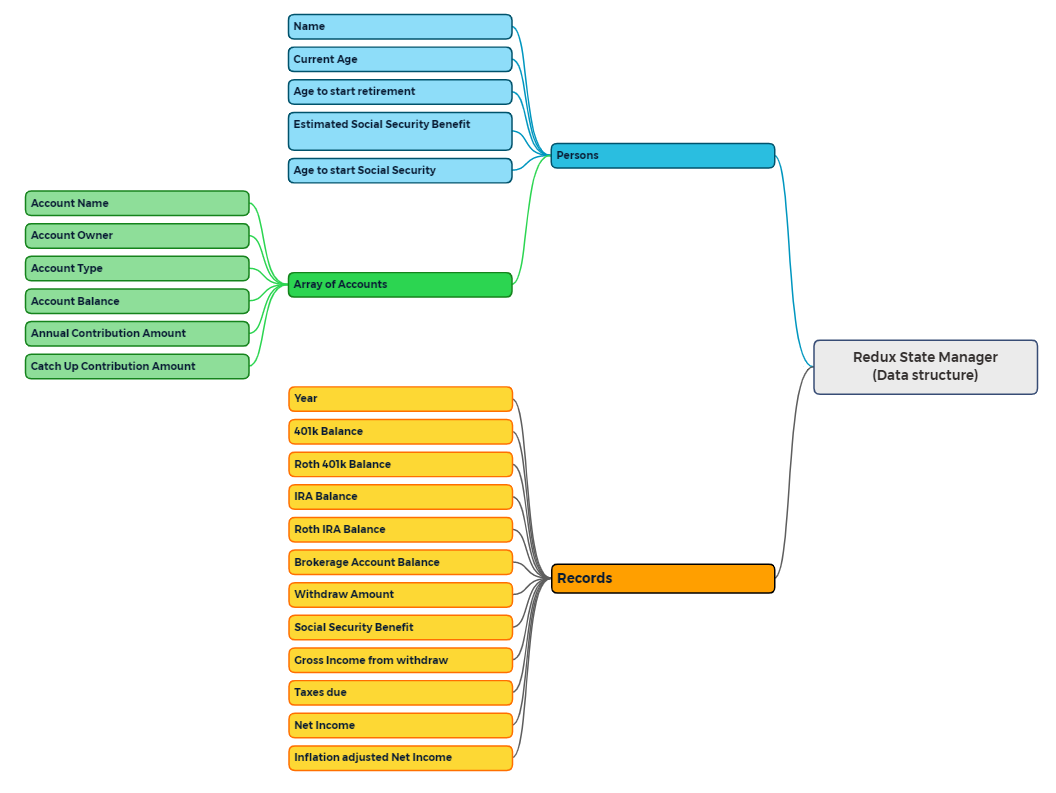
JavaScript is being utilized within NodeJS and Electron to enhance all the HTML in our interface, as well as preform all the backend processes. All of the computational workload, data structure state management and interaction between the user interface and the backend processes is coded using JavaScript.

The user interface is driven Electron, which is a library that sits on top of NodeJS and JavaScript. Electron enables us to create a modern web-based user interface for a locally install application. Electron achieves this by creating a chromium-based browser object to display the interfaces HTML pages in.

Our data structure is managed using the Redux library for state management. By using Redux, any changes to the Person, Accounts, or the records in the dataset are available to all elements of the program. This global state manager allows us to simplify the interface removing the need for the user to save or load data manually, because those events are tied to state changes within the state manager.

The modern web interface of our application utilizes industry standard BootStrap and BootSwatch CSS. The incorporation of BootStrap and BootSwatch enforces a design grid and uniform look and feel of the application controls. The sleek user interface’s design is heavily influence by mobile applications, which enforces a simple, intuitive, user friendly interface.

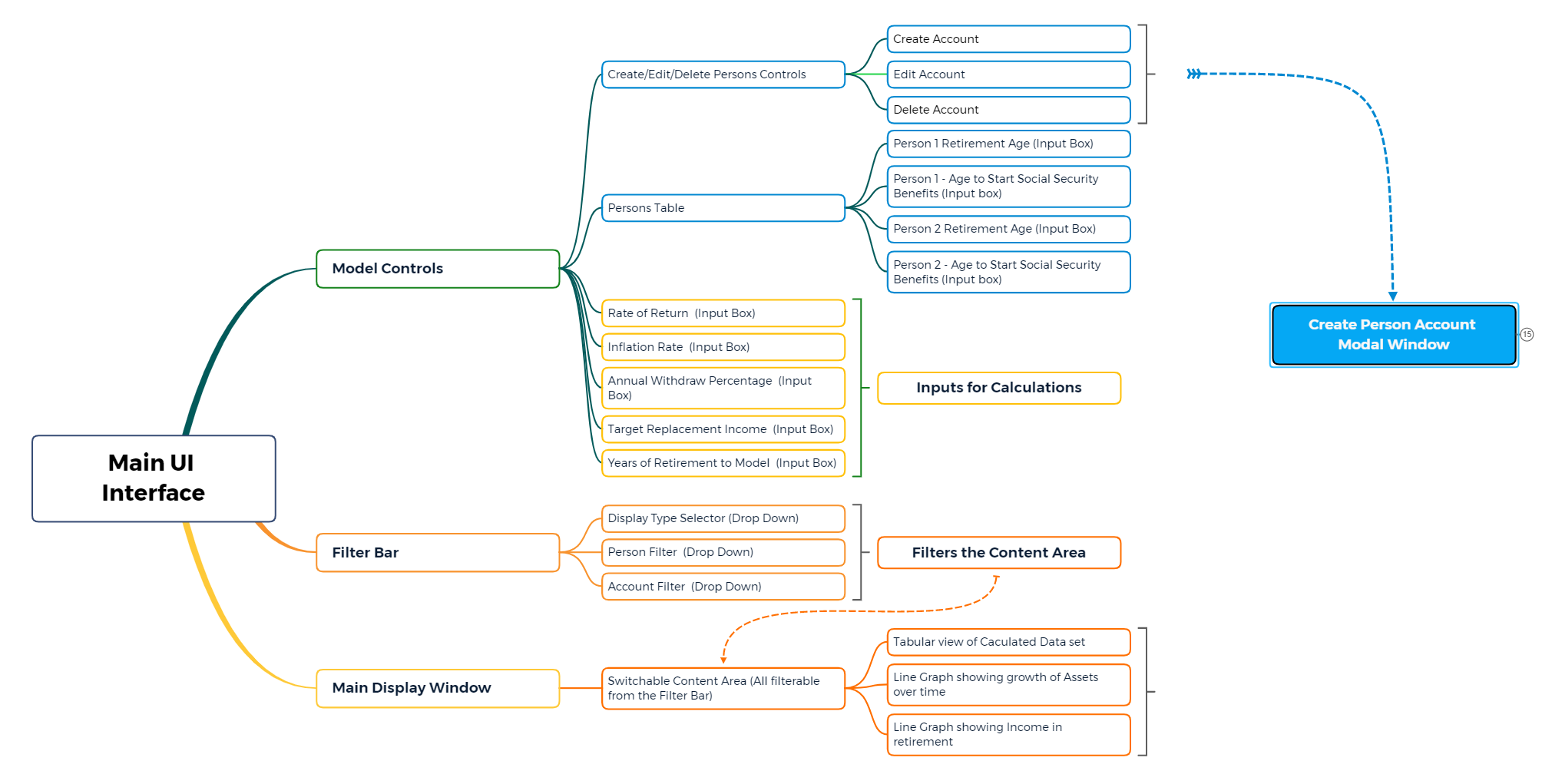
# Data Structures



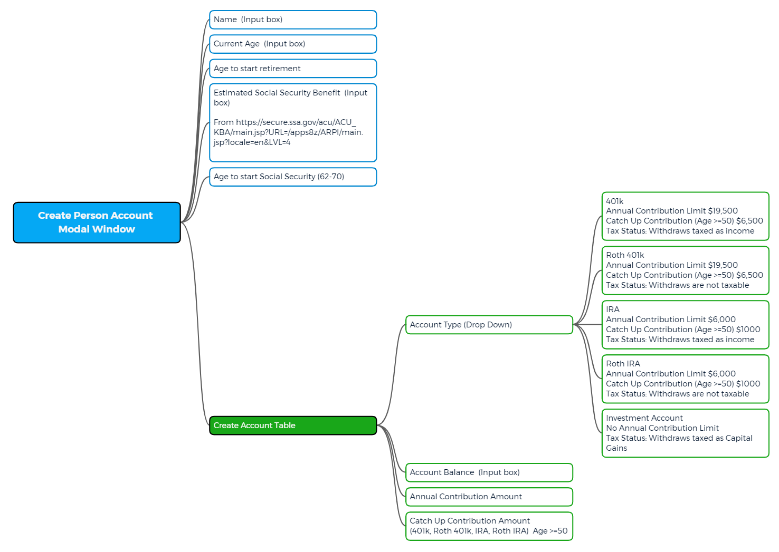
Our data structure is contained within the Redux state manager which manages the state or our user data and the calculated record set across the entire application. When a Person or Account object is created or edited it is instantly available to all parts of the program. When the model is run, all records for each year of the model are held in the state manager, making them available to any output control the client selects.

Central to our decision to use Redux for state management of our dataset is the ability to tie events to the changes in the state of objects. This has allowed us to remove the traditional user interface elements such as menus for saving and loading data. Those actions are tied to events such as the application being loaded will load the previously saved dataset from a JSON file, into the state manager. A change to a Person or Account will trigger an event which saves that data to the designated JSON file.

# Node Map



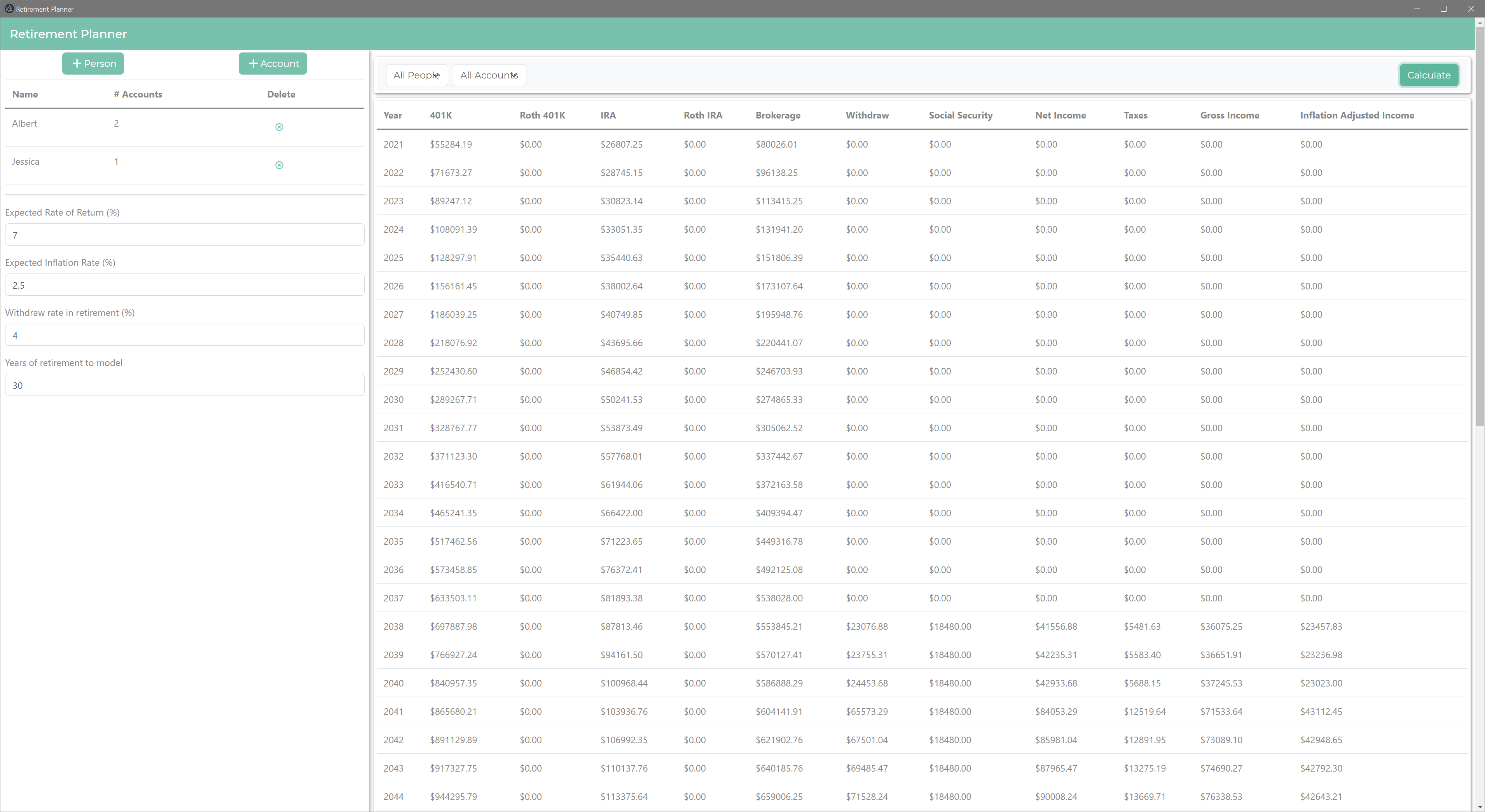
One of our primary design goals was to have a simple user interface akin to a mobile application. We have accomplished that goal through by having a user interface which is made up of two HTML pages. The primary user interface is composed of three areas, the **model controls**, the **filter bar**, and the **main display window** where the model data is displayed.

When the user clicks on the **Add / Edit Person button**, a separate modal window will open. Within that modal window the client is presented with all the controls to create or edit new persons or accounts related to those persons.

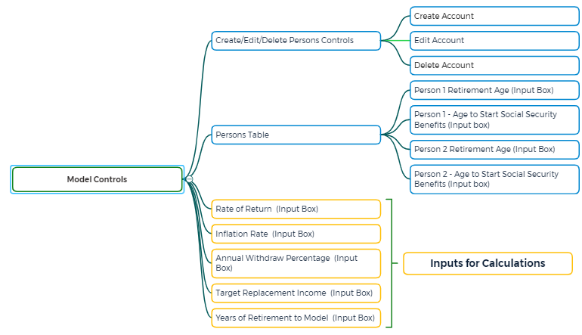
This simplified design presents the user with an interface that is intuitive and easy to use. As stated previously, one of our primary design goals was an interface that borrowed heavily from mobile application design with sleek user-friendly interfaces. The absence of a lot of controls does not diminish the functionality of our program through the inclusion of integrated actions from the Redux state manager.

# Application Workflow

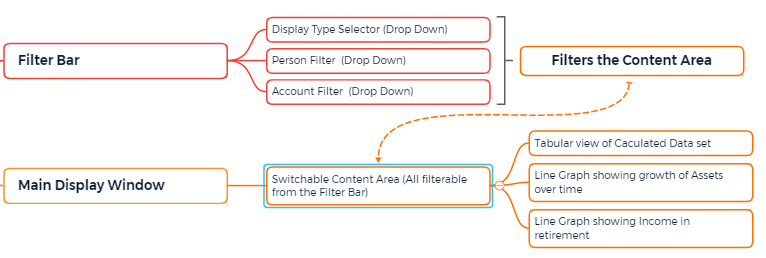
The User Interface (UI) is defined by thee major sections; the Model Controls, a Filter Bar and Main Display windows where the results are displayed. An additional HTML page is used as a Person and Account Popup window when the user creates or edits a Person and their accounts. Each of these sections are descripbed in detail below.



## The Model Controls

The **Model Controls** contains the controls to create and edit Persons and account for use in the model. The Person/Account data is displayed in a simple tabular format for easy viewing. This section will also contain controls that modify the model results, such as the years of retirement, withdraw rate, and the rate of return. These controls provide a method for the user to change the model parameters used in the calculations that make up the data model displayed in the Results Windows.

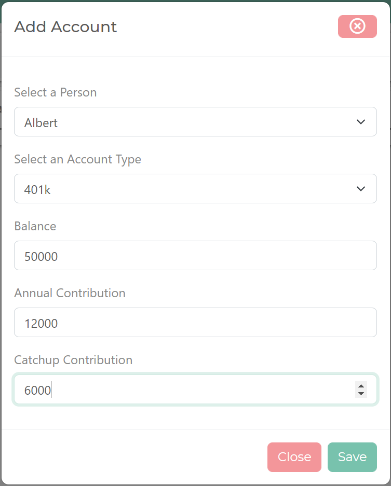
## The Filter Bar

The **Filter Bar** appears across the top of the right pane of the main application window. It has two functions related to filtering and displaying the results of the model. Dropdown boxes will provide controls related to filtering the data set to display specific people and accounts. A separate set of controls will provide options for toggling the type of display in the Results Window. A final control on the filter bar is the Calculate button which executes the computational routine that generates the model to display in the Results window.

## The Results Window

The most prominent feature of the user interface is the **Results Window**. This area of the UI is a dynamic canvas that will change based on the options selected in the Filter Bar. The tables displayed in this section utilized the data model based on our calculations projecting the account balances into the user’s retirement years. The dataset is held in the state manager, allowing us to change the display based on the filters selected.

## The Add/Edit Person Window

The Person and Account popups are opened from the Model Controls. Each page opens as a modal popup window, overlaying the main user interface. In this popup window, the user has all the controls available to create or edit Persons and their associated accounts. When a change is made in this form the data is added or updated in the datastore, making it available to the rest of the application. The data is also saved to the static JSON file upon a change in the data store.

# Features

|  |  |
| --- | --- |
| **Feature** | **Description** |
| Main User Interface | The main UI window is implemented in NodeJS using the Electron library. The interface is built using HTML with JavaScript for the controls |
| Intuitive user-friendly interface | The concept for our user interface borrows heavily from mobile application design, with sleek interface with few controls. |
| State management | State management from Redux to globally control the data structure throughout the entire application. |
| Static Data file | The user’s data is saved into a JSON file for storage or retrieval. |
| Multiple Persons | Many households have multiple people who can contribute to retirement planning. Our application allows for the creation of multiple people as account owners. |
| Multiple Account types | Each person can have multiple types of accounts associated with them. Additional account types can be added in the future. |
| Filters | The Filter Bar will allow the user to show all or part of the data in their calculations. This isolation allows for individual people or accounts to be analyzed. |
| Tabular data view | Display the calculated future values of accounts, taxes, withdraw, and inflation adjusted income in retirement in an easy-to-read tabular format. |

# Project Planning

This project has been the culmination of ideas and brainstorming by all three members of the team. We have divided up tasks among ourselves to develop them into the components of the project. Where needed we have collaborated and assisted each other to utilize our individual strength to bolster each other throughout the development process. A brief summary of the tasks that combined into our project are.

* Project conceptualization – Whole team participation, led by Robert Bilbee
* Platform choice – Whole team participation, led by Brock Herion
* Node map – Robert Bilbee
* Project Manual – Robert Bilbee
* Presentation Slides – Robert Bilbee
* Data structure modeling – Whole team, led by Brock Herion
* State Management development –Brock Heroin
* Static data preservation in external JSON file – Carson Christ
* UI Development – Main Application Window - Brock Herion
* UI Development – Add Person / Account Window - Brock Herion
* UI Development – Model Controls section – Whole team
* UI Development – Filter Bar Controls – Robert Bilbee
* UI Development – Results Area, Table view – Whole team
* Calculation Functions – Compound Interest and Future Value Series – Robert Bilbee
* Calculation Functions – Constructing the Record set – Robert Bilbee

# Test Plan

|  |  |
| --- | --- |
| **Model Controls Section** | Test that each of the controls in the input section are readable from the JavaScript and that the input validation is working. |
| **Filter Bar** | Test that each of the controls in the filter bar are readable from the JavaScript and that the input validation is working. and filter the results based on the values in the controls. |
| **Create Person/Accounts** | Create several person objects and validate the save and restore functionality work properly from the JSON file and state manager. |
| **State Management** | Validate that the person and account information is properly held in the data store. |
| **JSON output storage** | Read/Write from the static JSON file when the program launches and when there is a state change in the data store. |
| **Calculations** | Validate the calculation functions are producing correct values within the JavaScript functions. The growth in the table can be manually calculated to verify the results. The official testing involves creating person/accounts and verifying the values. See below; |
| **Results Window** | Test the display of the table format is displayed properly within the section. |
| **Tabular Data view** | Check for proper table formatting across a wide range of values to show the data displays properly. |

The calculations testing is to be completed in two passes.

### Pass #1

1. Create a test person with these parameters.

{age = 45, retirement age = 65, Est social security = 1500, Social Security age = 67}

1. Add account balance and contributions to the test person.

{401k, balance = 10000, contribution = 1000, catchup contribution = 1000}

1. Set the model parameters to;

{Rate of Return = 7%, Inflation = 2.5%, Withdraw Rate = 4%, Years to model = 30}

1. The model should total to these values at year 2041 (1st year of retirement)

|  |  |
| --- | --- |
| **Withdraw Amount** | $4727.15 |
| **Net Income** | $22727.15 |
| **Taxes Due** | $2529.76 |
| **Gross Income** | $20197.39 |
| **Inflation adjusted gross income** | $12172.72 |

at year 2060

|  |  |
| --- | --- |
| **Withdraw Amount** | $8197.59 |
| **Net Income** | $26197.59 |
| **Taxes Due** | $2946.21 |
| **Gross Income** | $23251.38 |
| **Inflation adjusted gross income** | $8662.21 |

### Pass #2

1. Create two test persons with these parameters.

{age = 35, retirement age = 65, Est social security = 2000, Social Security age = 67}

{age = 30, retirement age = 65, Est social security = 1500, Social Security age = 67}

1. Add account balance and contributions to person #1.

{401k, balance = 10000, contribution = 1000, catchup contribution = 1000}

{IRA, balance = 20000, contribution = 2000, catchup contribution = 2000}

{Brokerage, balance = 0, contribution = 100}

1. Add account balance and contributions to person #1.

{roth401k, balance = 10000, contribution = 1000, catchup contribution = 1000}

{rothIRA, balance = 20000, contribution = 2000, catchup contribution = 2000}

1. Set the model parameters to;

{Rate of Return = 7%, Inflation = 2.5%, Withdraw Rate = 4%, Years to model = 30}

1. The model should total to these values at year 2051 (1st year of retirement)

|  |  |
| --- | --- |
| **Withdraw Amount** | $27360.33 |
| **Net Income** | $27360.33 |
| **Taxes Due** | $3098.82 |
| **Gross Income** | $24261.51 |
| **Inflation adjusted gross income** | $11351.58 |

at year 2070

|  |  |
| --- | --- |
| **Withdraw Amount** | $103732.27 |
| **Net Income** | $121732.27 |
| **Taxes Due** | $10135.39 |
| **Gross Income** | $111596.87 |
| **Inflation adjusted gross income** | $32275.89 |

# Development changes throughout the project

Throughout the course of the project development, we have made several changes to our design. This is part of the story of the creation of this application and why it is detailed here.

Originally, we included a TypeScript library to make our JavaScript strongly typed. Our reasons for including this package were to preemptively validate the datatypes store in variables and objects to reduce the workload in debugging the application during the development phase. During development, we determined that the inclusion of the TypeScript library added unnecessary complexity and offered little benefit to us due to the size and scope of our project.

The inclusion of the Redux state management library brought several changes to the project. Our original data model using Person, Account, and Record classes changed to the slices within the state manager for each “type” of data we are managing in the data store. The effect of having a global state manager for our data also led to changes in our User Interface. Once the Redux library was added to the project, we gained the ability to add events that were tied to the state of our dataset. This realization has allowed us to further simplify the user interface by removing the traditional menus. Using those event driven actions, the application loads the contents of the JSON file into the data store when the application loads. Any changes to the state of the person and accounts data triggers a separate event and action to save that date to the JSON file automatically with no further user interaction required.

The changes away from using TypeScript and Object-Oriented classes for our People and Accounts, as well as introducing Redux did add a considerable amount of complexity to the project. All the benefits discussed previously were paid for in time and rework on the project to accept the state manager into our existing code. Due to this we had to par back some features that we originally planned to include a graphical display of the tabular data that would be displayed in the results section, as well as a printable report. Both features could be added with relative ease, but we simply ran out of time due to the evolution of the project.

# Conclusion

Throughout this project we have learned as a team that the cycle of planning, developing, implementing, and debugging is an iterative cycle. Each new cycle adds to the previous cycles building the project from conceptions into a viable program. Every cycle brings new challenges ranging from the development of new features, implementing them with existing code, or finding bugs that were unexpected. Having a clear end goal has been tantamount to our success because we have had the end goal in mind throughout the project. That does not mean that it was always smooth sailing. As detailed in the previous section, we made several design decisions at during development. Making these changes was a reaction to the issues at hand in the development cycle and those changes caused some rework to be performed. By analyzing the cost/benefit of these changes we were able to take a step back, redesign, and restart the development on those features, integrating the changes into the entire scope of the project.