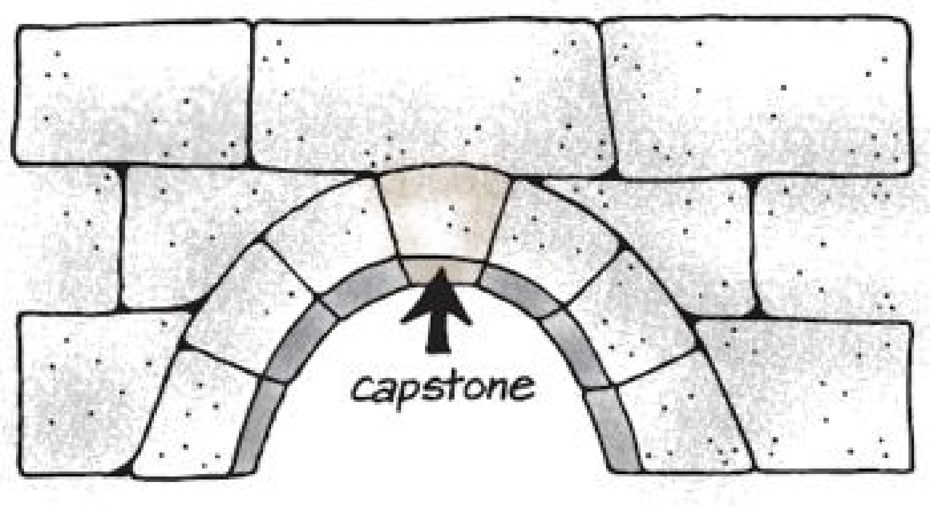
PettyCash Documentation

Group Name: Group 3



Team Member Names: Emeka Ezeji-Okoye, Ben Friedman,

Josh O'Steen, and Alex Foster

I. **Chosen Project:** We would like to pursue development of a Budgeting/Savings application which would provide users an incentivized savings model and track goal progress. Our project will have both a web platform and mobile applications. The hardware we might need is to do this is a Web Server, Database, iOS devices, Android devices, MacBooks running Xcode.

   A. **Mentor:** We would like our mentor to be Dr. Shyu. He is an experienced lead researcher in relational/non-relational databases and is the chair of Computer Engineering and Electrical Engineering.

II. **Goals:** The goal for our application is to promote better fiscal discipline and help users achieve their budgeting goals. Our mobile and web platforms will help users set goals, track their progress, and most of all, save money. This will be done by creating emotional incentives to the users by attaching savings plans to a virtual pet, which will have various reactions to different scenarios i.e. Contributing towards your savings, withdrawing funds, paying bills on time etc.

III. **Capstoners Team:**

Emeka Ezeji-Okoye is a senior Computer Science major with a minor in business as well. He has working experience using HTML/CSS, JavaScript, AngularJS, PHP, and Java. Also, he is interested in both web and mobile application development.

Ben Friedman is a senior majoring in Computer Science. He is familiar with the following languages and technologies: Java, C, PHP, HTML/CSS, JavaScript, SQL, AngularJS, and CodeIgniter. He is interested in app development and web development (back-end).

Josh O'Steen is a senior in the Computer Science program. He prefers to work on mobile platforms such as iOS and hopes to contribute primarily to this component of the project. He also has extensive experience in web frameworks/development, database administration, as well as Agile and Test Driven development.

Alex Foster: Senior student in the Information Technology program, with a minor in Computer Science. He has experience in Object Oriented Design and development especially in team environments. He has worked with technologies such as, C, Java, PHP, C# .NET, HTML/CSS, Source control, and Test Driven Development.

IV. Introduction:

Introduction: The problem the Capstoners will attempt to solve is that of fiscal irresponsibility, particularly with saving money, and budgeting appropriate goals.

Problem: Many people consider the task of money-managing and saving to be tough or problematic, to the point where it may endanger them economically in the future. Whether it is spending too much out at the bars, online shopping, or take out, it will affect you financially in many ways.

Solution: Our application is an easy to use and fun process of creating budget goals, encouraging deposits to your savings, monitoring your spending, and reminding you of upcoming expenses in order to give the user more reliability in a financial monitoring tool.

V. Requirements:

User Requirements: The users would expect this application to help them save their money.

Hardware Requirements:

- Database: Handled by Apple's CloudKit Service.
- Microsoft Azure hosted Apache Web Server.
- Mobile devices: all iOS platforms.
- Web-Facing devices: personal computers with the latest releases of either Google Chrome Browser, Safari, Firefox, or IE.

Functional Requirements

- **Login**: For a user to login will require they use their Apple iCloud username and password. For the iOS application, it will be required that the user be logged into their iCloud account at the system level within the operating system's settings. Upon opening the iOS application, it will automatically check for an existing iCloud account, proceed successfully if one is present, or notify the user that they must be logged in to their appropriate iCloud account otherwise. For the web application, users will be redirected to a secure Apple iCloud login page when they choose to login to their account. Once they have entered their credentials and are verified by Apple, they will once again be redirected back to the web application where they will be presented with their personal landing page.
- **Forgot Email or Password**: Since we will not be storing any user login credentials, but instead use Apple's iCloud verification, if a user were to forget their iCloud email or password, they would be directed to the Apple website for resetting such credentials. A

link will be supplied on the Apple provided login gateway for the web application. A similar link is built into the iOS Settings application where a user would login to their iCloud account.

- **Create a Goal**: From the user's landing page, they will be able to create a savings goal. A savings goal will consist of a monetary amount and an optional date in the future. The user can choose from which of their accounts they would like the funds to be withdrawn from as well as to which account they would like the funds saved to. Lastly, the user will choose their pet or plant that will be associated with the created goal.

- **View a Goal:** From a user's landing page, they will be able to select a previously created goal. Upon selecting the goal, they will be presented with their pet or plant that is associated with that goal.

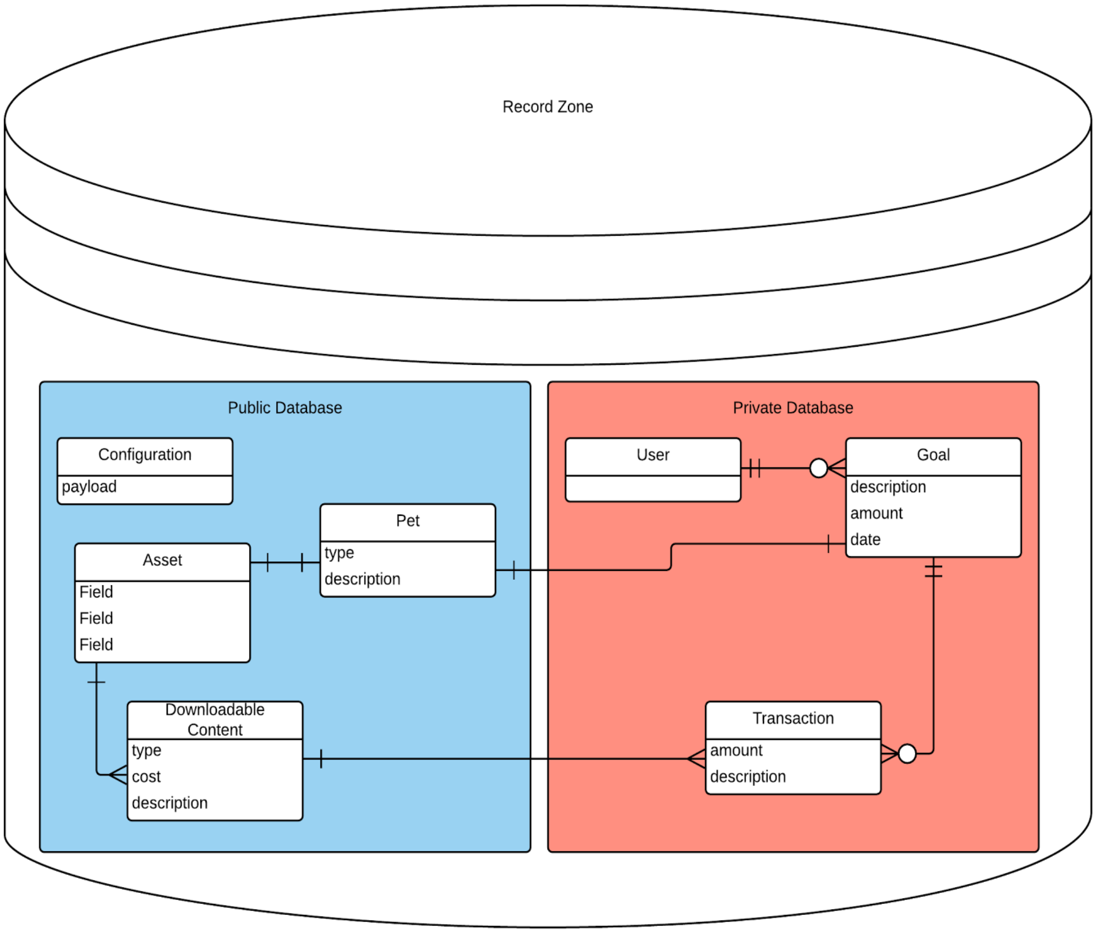
- **Maintain a Goal:** Once a goal has been created, the user will receive periodic notifications on their iOS device. The notifications will be used to remind the user to tend to their pet or plant, which is representative of depositing money into their savings account to save towards their goal.

- **View Transactions/Expenses:** Upon completion of the bank account linking upon initial login the user will be able to view a list of their transactions They will also be able to filter the transactions by spending category type and see a chart breakdown of where their spending has gone.

Designs

The design section for our project documentation is meant to allow the reader to visualize the system architecture, database, user/system processes, and site/app mockups before the application and website are completed. In this design section, we have included an ERD, System Architecture diagram showing the flow of information from the back end to the front end, several flowcharts detailing the user decision process through the key components of the application, as well as a few site mockups to help visualize the eventual application.

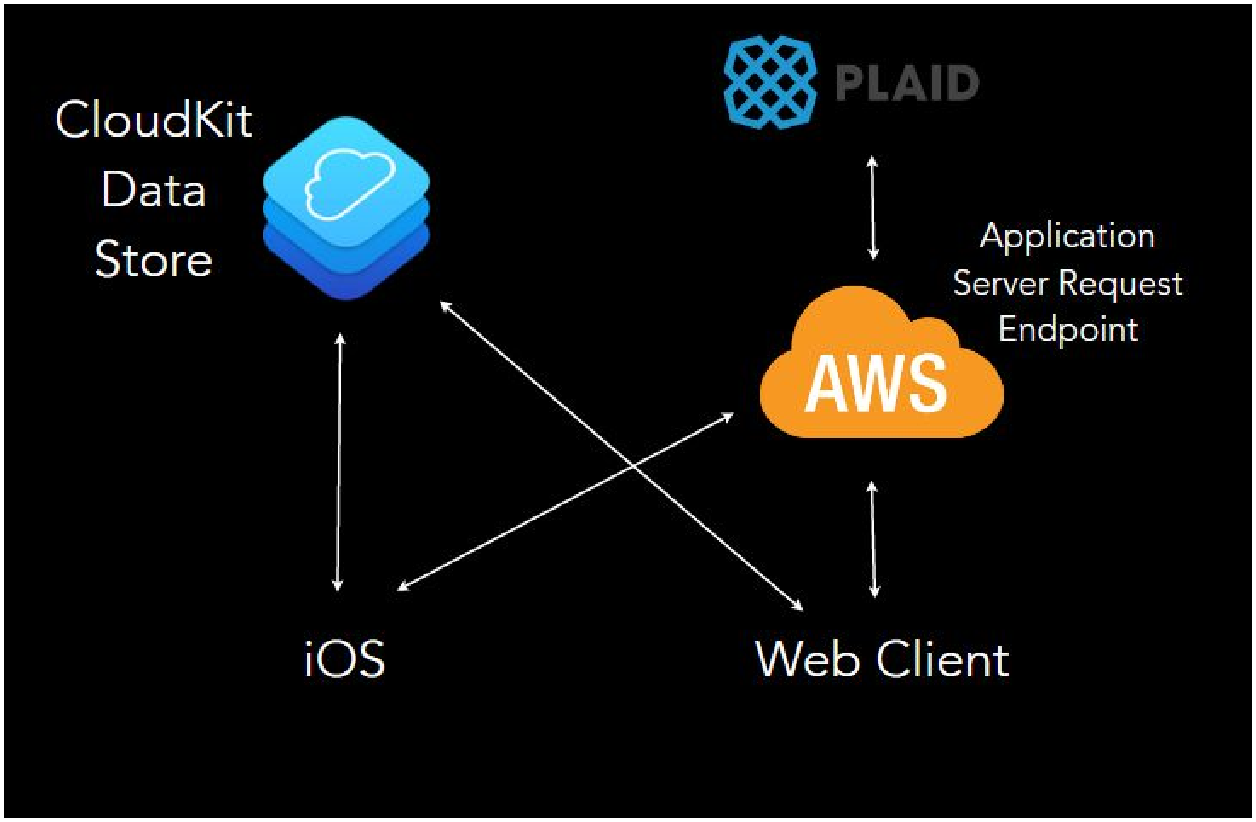
ERD:



Because CloudKit is not a traditional DBMS and rather an ORM, these ERD tables should be though more of like objects rather than just rows. While traditional properties and relationships are present, the structure of CloudKit databases is distinct as is its built in metadata for each record. We will not need to worry about saving primary keys for relationships but rather directly relate records using back references. The built in metadata such as creation date, record author, and built in unique identifiers allow for easy and quick quering and sorting.

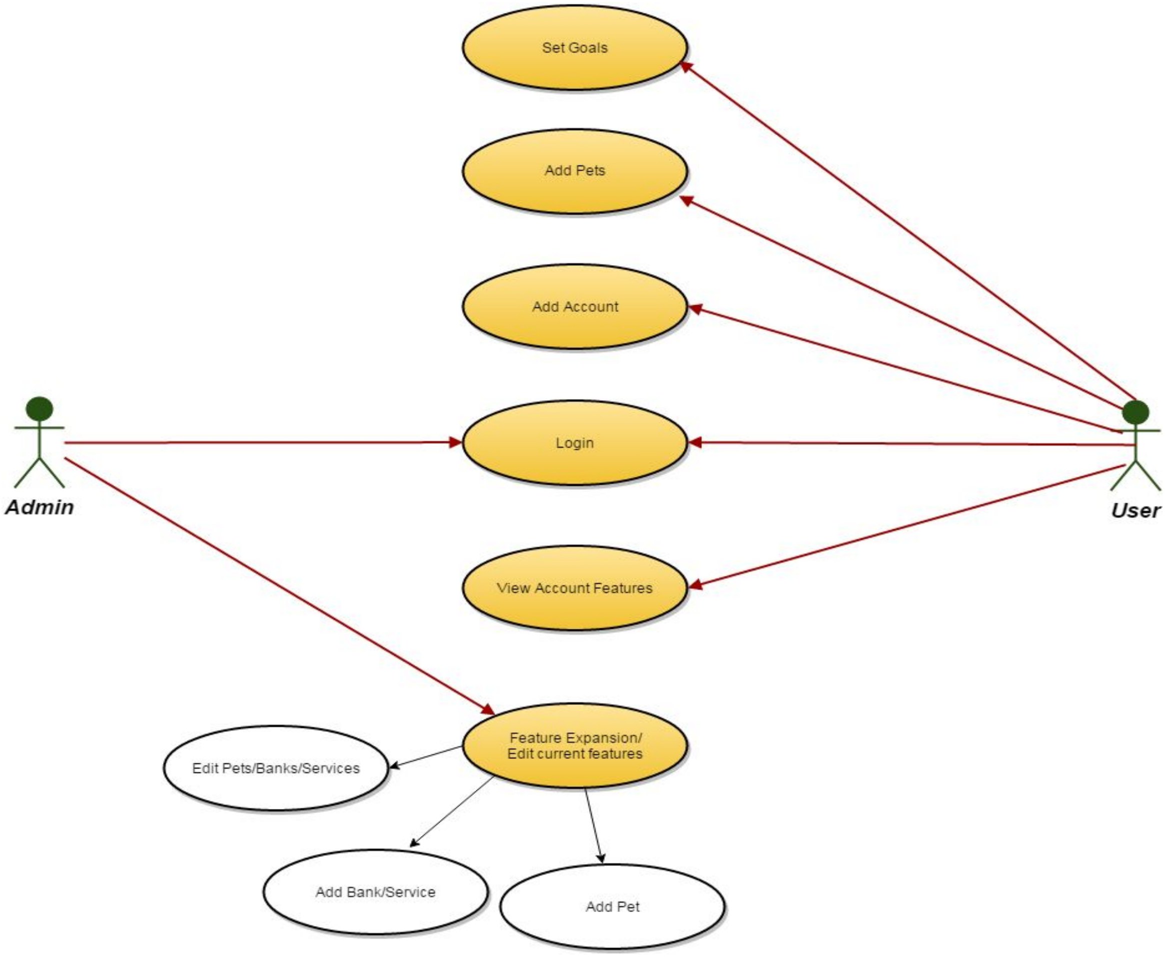
A CloudKit record zone can be considered a single application's workable storage area. This area is partitioned into two distinct regions: public and private. The public database will be used for records that will be accessible to all users of the application. These files will consist of configuration records that will allow for easier and smoother app configuration updates. It will also contain records used for downloadable content that will allow the user to customize their personal experience. Asset records for will be used to save larger data files such as images or videos and will always be back referenced by another record. Lastly, the public database will contain records representing the various pets or animals that are available to nurture for the duration of the savings goal. The private database is unique to the user and is read/write only by the user to which it belongs. This creates the perfect region to store a user's goals and their associated transactions. User records are supplied by CloudKit by default for login and transaction verification and tracking. However, we will not be utilizing the user records directly. Again because CloudKit acts more as an ORM, many of these records are abstract in nature and can be reused to represent multiple subtypes of the same record type; assets have the ability to be an image, video, or file any other file based subtype without any special system configuration.

System Architecture:



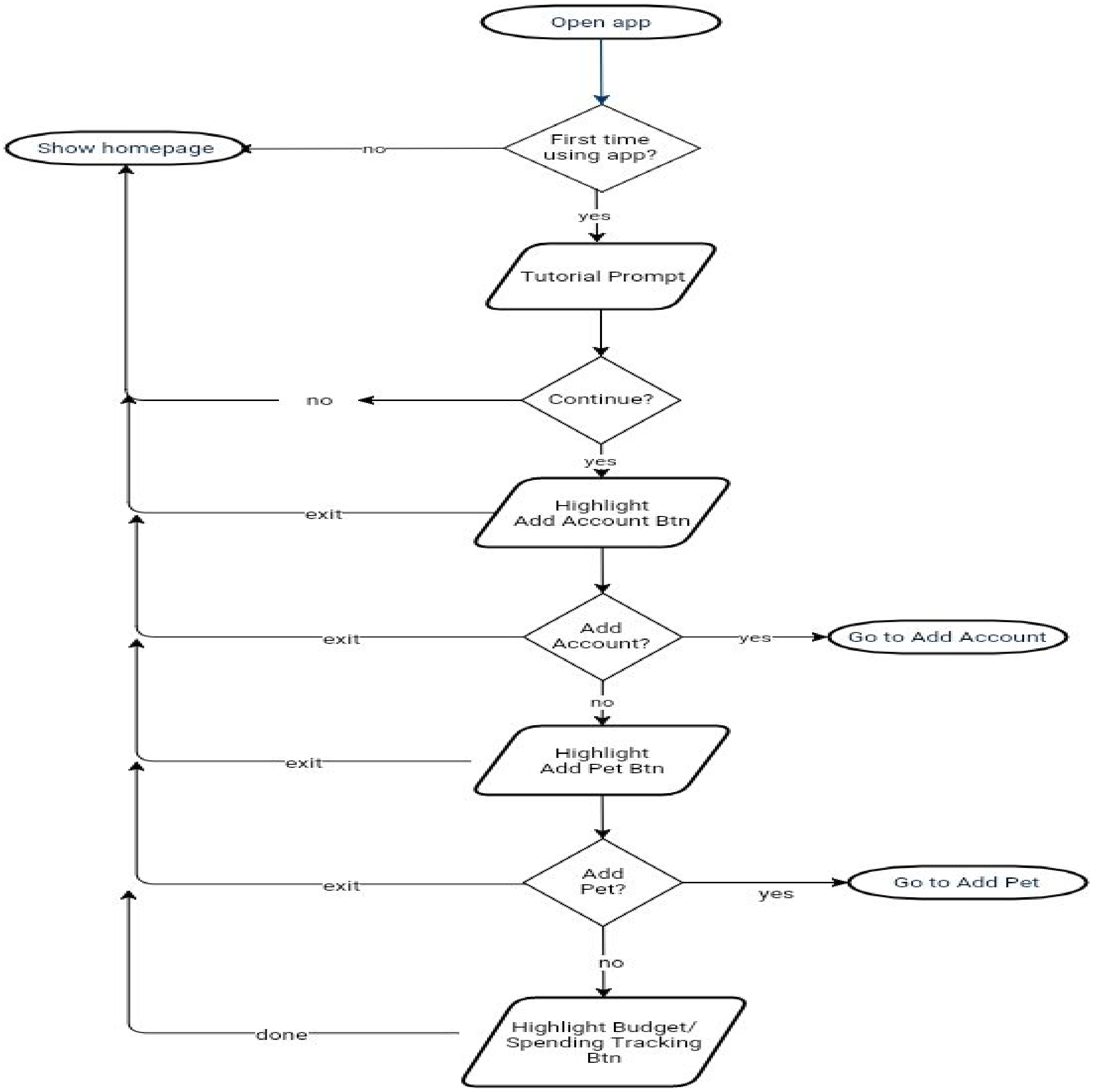
This is the system architecture model for our application(s). The graph illustrates how the iOS application and the Web application interact with the CloudKit database, as well as how the applications will leverage the application server and Plaid API to retrieve transactions and account information from their bank account.

Use-Case:



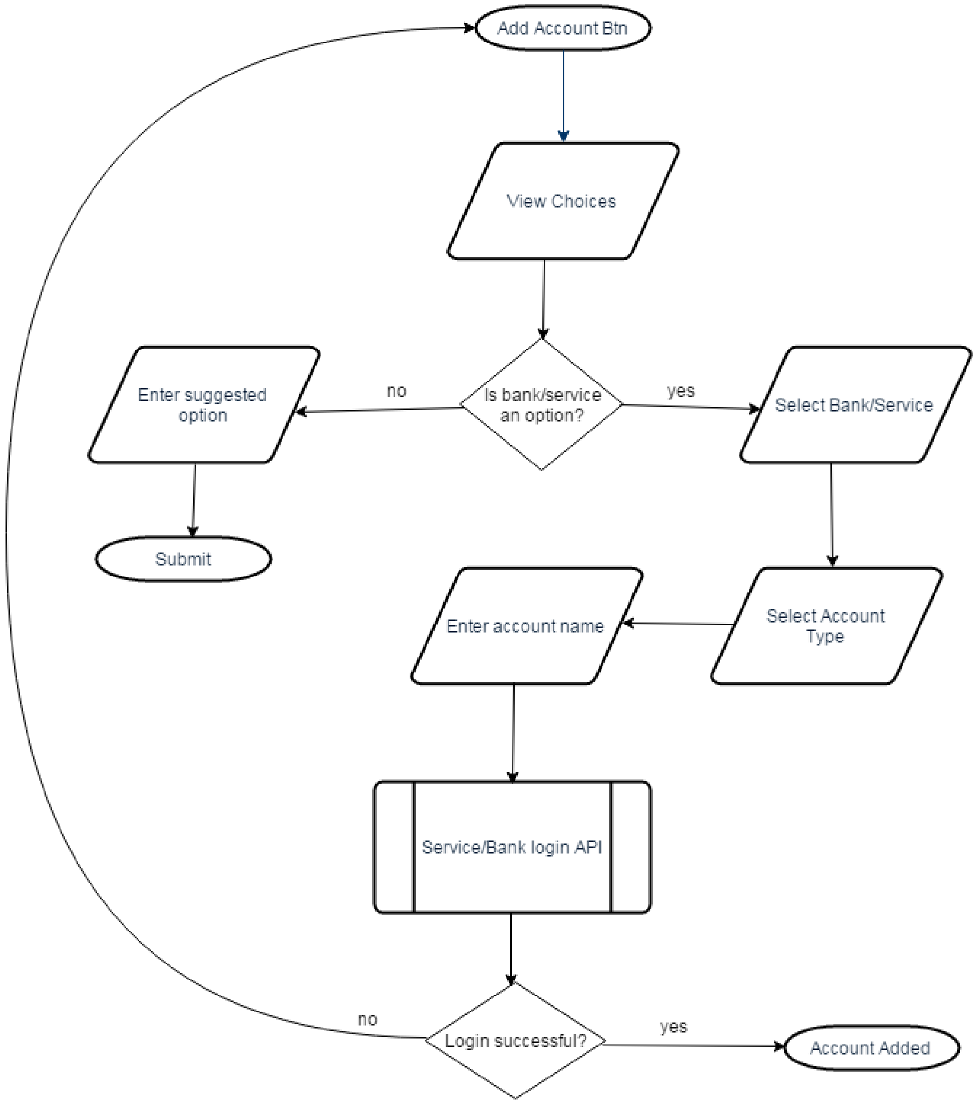
This is the Use-Case diagram for our project. For the mobile app there is only one kind of user, simply called the ‘User’. However, for the web version there is a second user called ‘Admin’ which has one ability other than login. That is the ability to add more features to the website or to edit the currently available ones.

Tutorial Flowchart:



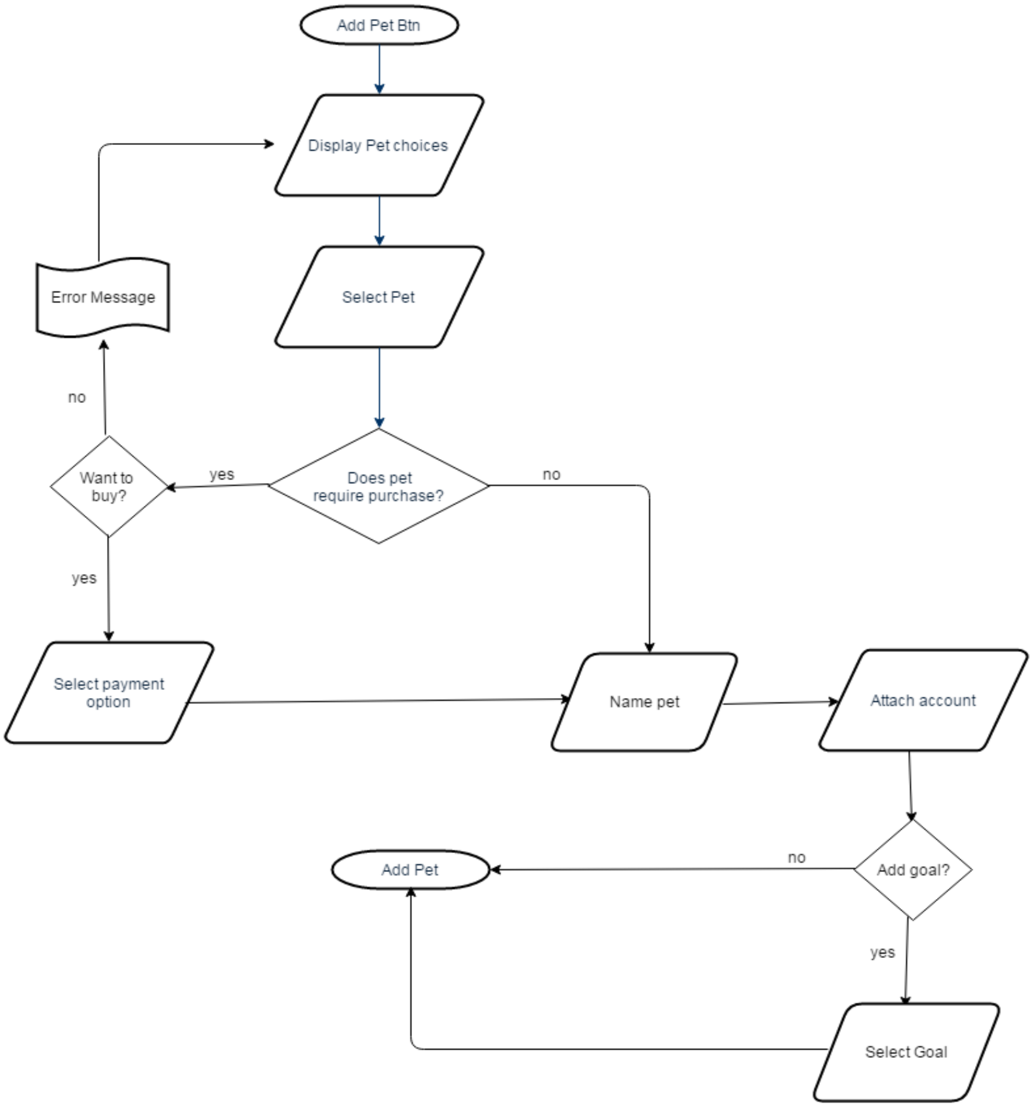
This flowchart shows the application and user processes if the user is opening the application for the first time. Once the user opens the app, a tutorial will begin. During the tutorial, the available buttons on the home screen will be highlighted and explained. The user has the option to exit the tutorial at any time. For the more important features (Accounts and Pets), the user will also be given a prompt to either add an account or pet.

Add Account Flowchart:



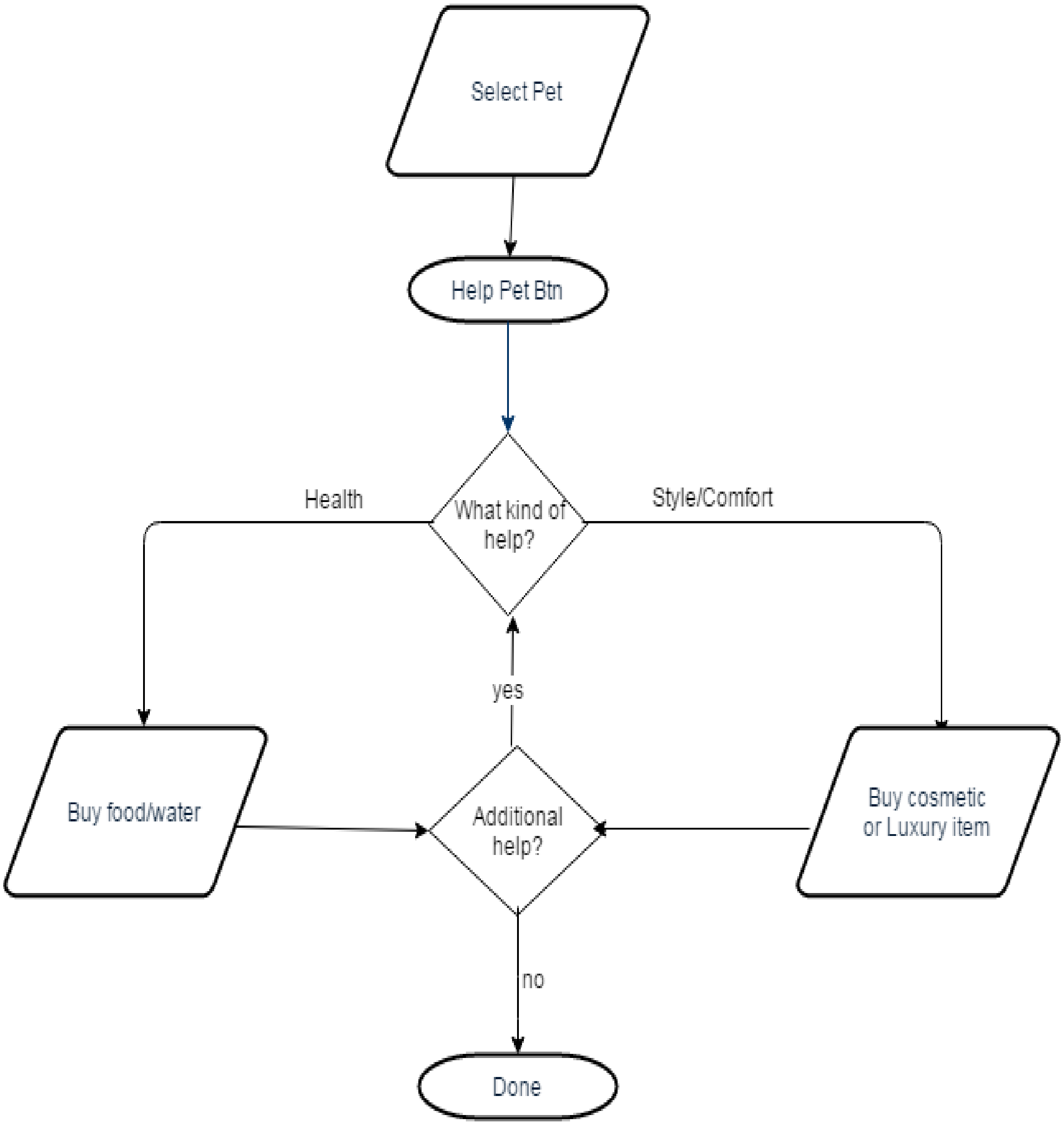
This flowchart details the user process for linking a financial service account to the application. The user will have the ability to submit a suggestion if their desired service is not currently available on the app. Once the user has selected their service, account type, and account name, they will be directed to the Login system of the chosen service/bank.

Add Pet Flowchart:



This flowchart shows the user process of adding a pet. When selecting a pet, the app will determine whether the selected pet requires a purchase or not, since not all pets will be available for free. After giving the pet a name, the user will then attach a payment account to the pet and will also have the option of adding a goal.

Upgrade Pet Flowchart:

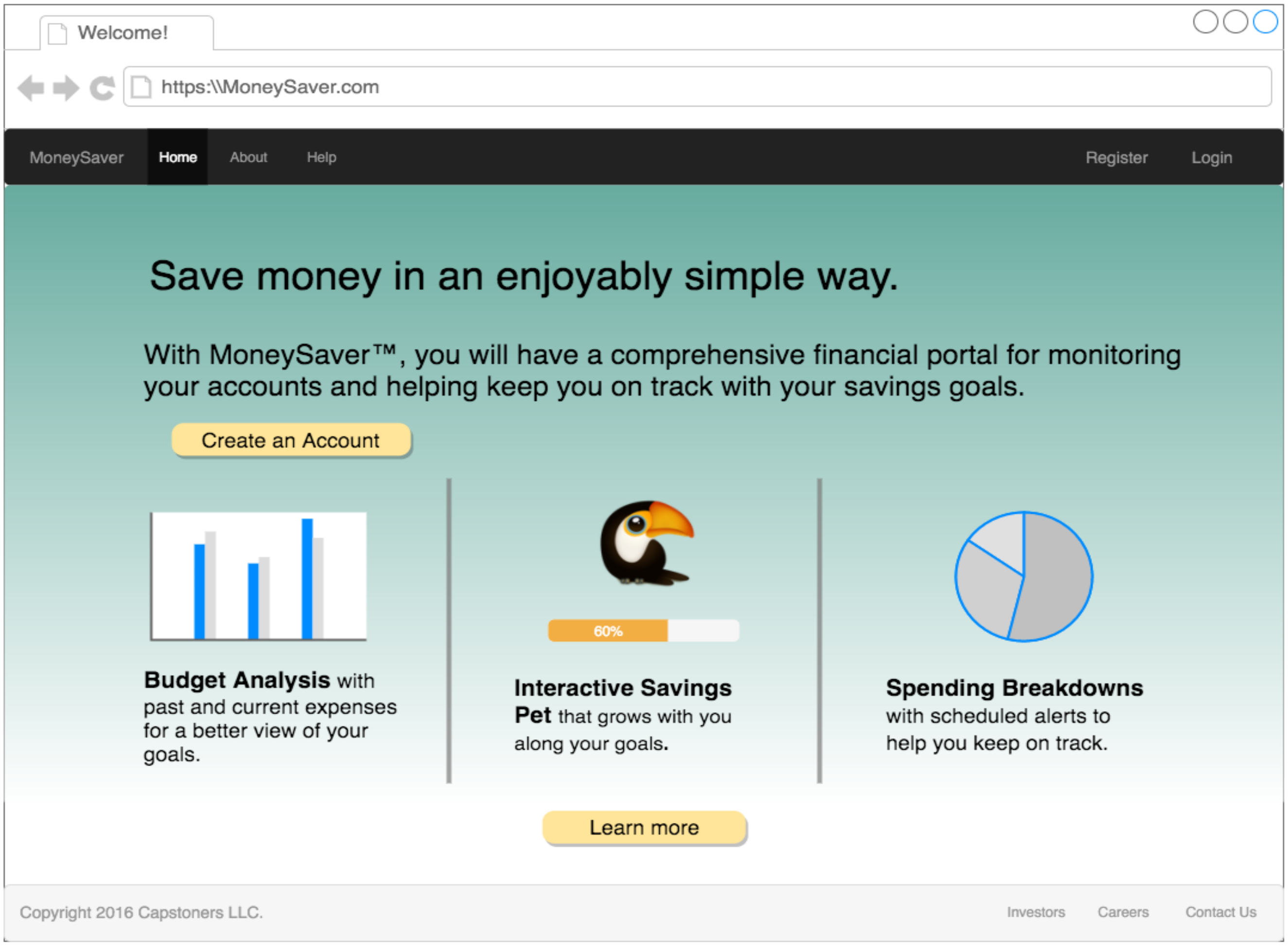


This is the flowchart for options a user will have for upgrading/improving/helping their pets. There are two kinds of improvements that can be selected: Health or Style/Comfort. The user can only buy food/water for the pet under health, but can choose between a number of different cosmetic or luxury items under Style/Comfort.

While upgrading the cosmetic attributes of the pet is dependant on user input and in-app purchases, the actual changes to the UI (state of the pet) are determined by the underlying code. Throughout the “lifespan” of each savings goal, pet growth will be determined by the user’s ability to reach benchmarks in said goal. Depending on the amount of money to be saved, goals will be appropriately and realistically partitioned into “sub-goals” that when met apply default changes to the user’s pet. The algorithm used to perform this process will analyze the speed at which the user achieved a sub goal and when added to the level of goal completion, evaluate the state of the pet. As the user continues to reach benchmarks in a savings goal, they will approach the “final” form of their pet which can then be further upgraded through additional purchases. This end-game scenario would be used if a user decided that they wanted to extend an existing savings goal for any reason. With respect to animation, the user’s pet will be simply animated using SpriteKit and PhaserJS while the application is in an idle state.

<u>Mockups</u>

Web-facing application:



Save money in an enjoyably simple way.

With MoneySaver™, you will have a comprehensive financial portal for monitoring your accounts and helping keep you on track with your savings goals.

Budget Analysis with past and current expenses for a better view of your goals.

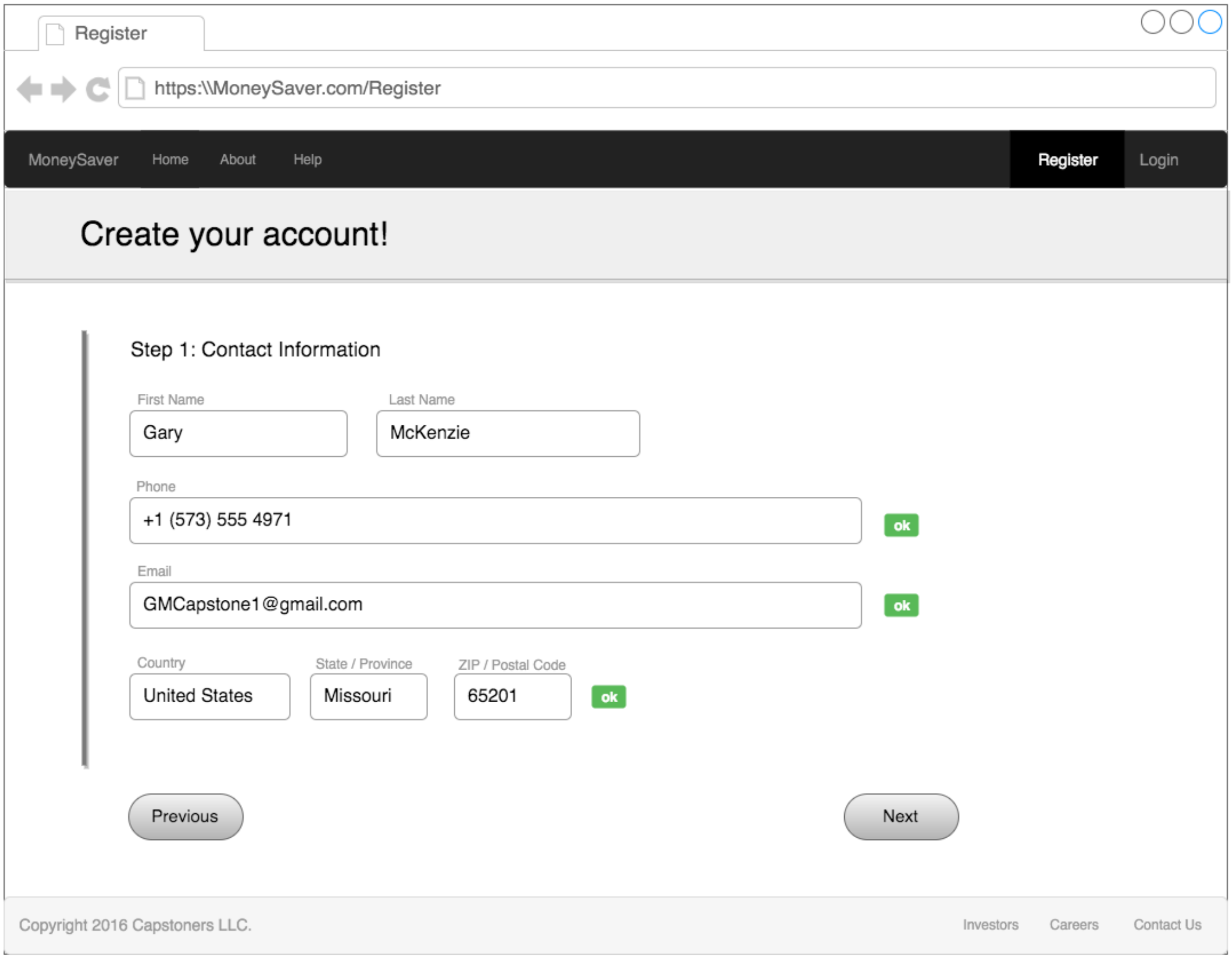
Interactive Savings Pet that grows with you along your goals.

Spending Breakdowns with scheduled alerts to help you keep on track.

Copyright 2016 Capstoners LLC.

The homepage of our application will give a quick glimpse into our product and feature a dynamically sizing interface for multiple device sizes.

Web-facing account creation:



Create your account!

Step 1: Contact Information

First Name: Gary

Last Name: McKenzie

Phone: +1 (573) 555 4971 ok

Email: GMCapstone1@gmail.com ok

Country: United States

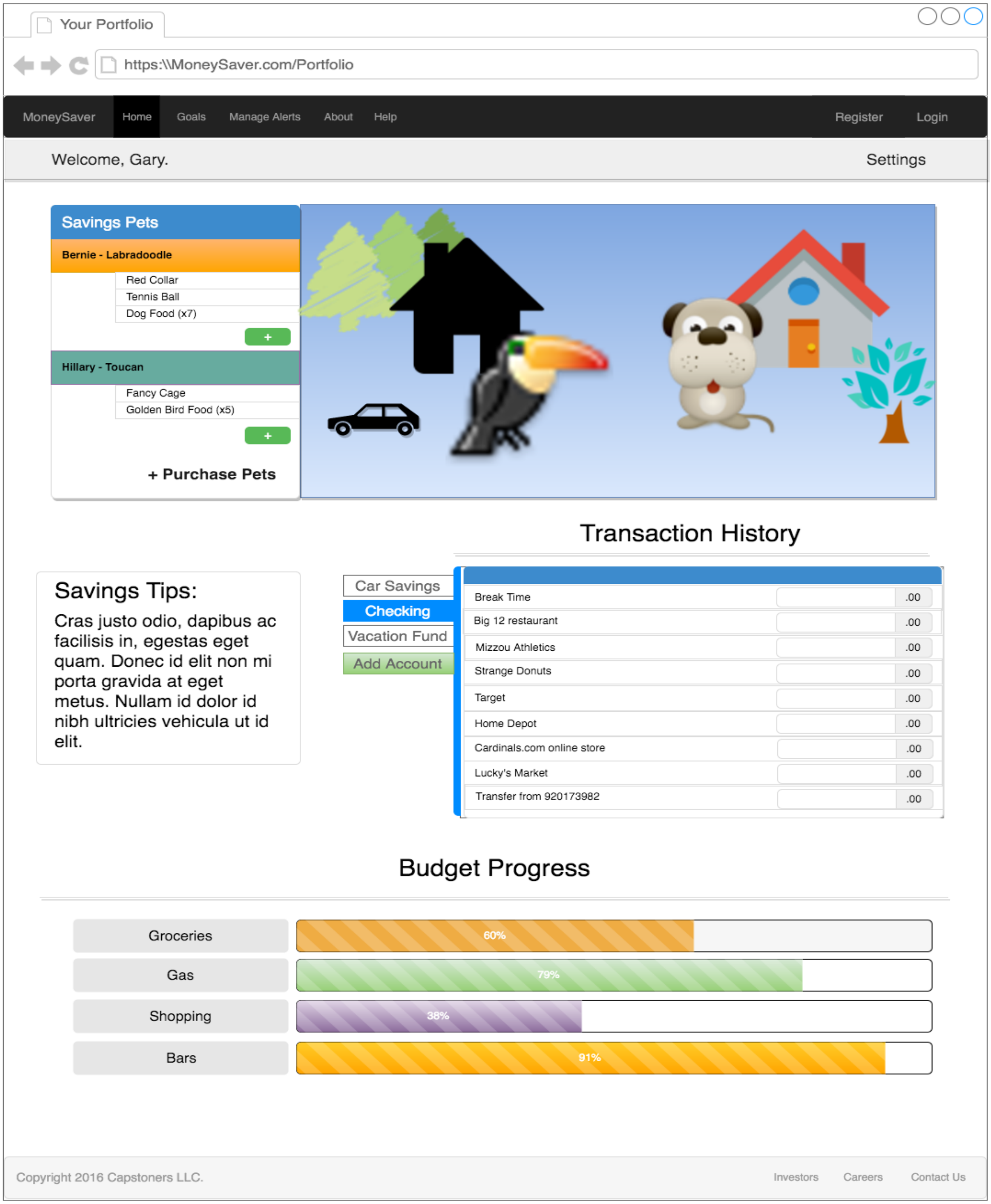
State / Province: Missouri

ZIP / Postal Code: 65201 ok

Copyright 2016 Capstoners LLC.

The account creation process will have a few steps. The first being an account creation and personal info storing. Then details for your Bank/Paypal accounts, and finally budget creation and pet selection.

Web-facing logged in portfolio homepage:



Savings Tips:

Cras justo odio, dapibus ac facilisis in, egestas eget quam. Donec id elit non mi porta gravida at eget metus. Nullam id dolor id nibh ultricies vehicula ut id elit.

Break Time	.00
Big 12 restaurant	.00
Mizzou Athletics	.00
Strange Donuts	.00
Target	.00
Home Depot	.00
Cardinals.com online store	.00
Lucky's Market	.00
Transfer from 920173982	.00

Groceries	60%
Gas	79%
Shopping	38%
Bars	91%

Development

Possible Troubles:

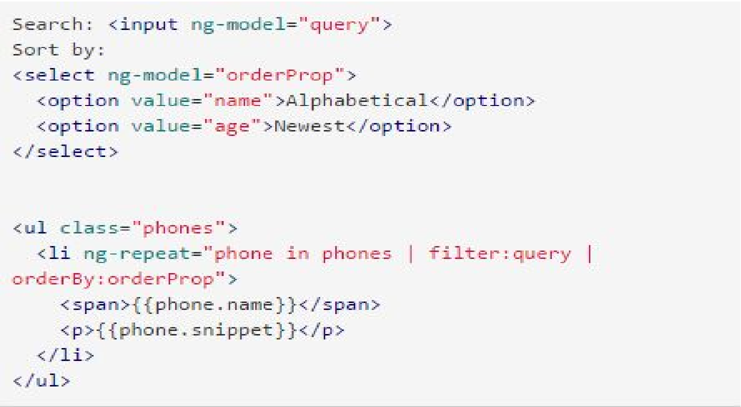
- Recreating the same Graphics and Animations cross-platform
- Synchronization due to the properties of each Pet being computed on the fly based on values stored within CloudKit for each user.
- UI/UX differences between platforms
    - Possible solution: Established UX design and team communication.
- Ease of access for using the payment processing platforms: OpenACH, PayPal etc.
    - Possible solution: use a MySQL database as a Mock bank account with an accessible list of transactions.
- CloudKit event notifications on the server side
    - Possible Solution: Create our own push notification server

Project Related Research:

JavaScript Frameworks

For this project, our group has decided to go the route of developing both an iOS application and a web application. While the selection of what language to use for iOS is made for us with Swift, how we want to develop the web application portion of this project was to be decided. Ultimately we decided to develop the application using JavaScript. The next decision to be made, the problem that needed a solution, was what JavaScript framework we were going to use to help us over the course of this project. We realized that none of us had much experience using JavaScript frameworks to fully develop applications. The desired JavaScript framework for developing our project needs to allow for a complex web application that is fast and user friendly. Also, the framework needs to follow the Model-View-Controller design pattern or as close to it as possible. The MVC design pattern divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. From my initial research into the different JavaScript frameworks available, generally three of the most popular frameworks include: AngularJS, Backbone.JS, and Ember. AngularJS is the only framework on the list that at least one person in our group has toyed with in the past, but nothing substantial. The popularity of these frameworks warrant further research to determine which one suits the needs of our group the best, and that allow for the easy development of the web application portion of our project.

AngularJS is an open-source framework that is mainly maintained by Google. The key attracting point of AngularJS seems to be the use of Two-way-binding that allows for automatic synchronization of data between model and view components [1]. The following picture illustrates the use of curly brackets to dynamically populate the view with data from the model. As the user changes how the list of phone numbers of sorted, the list will automatically update using the bindings. “This essentially takes away all the pain of writing manual code for DOM manipulation” [3]. Another benefit of AngularJS is that the framework appears to actually be the most popular JavaScript framework for the general public [2]. This indicates there are a large number of web developers either amateur or professional that feels that AngularJS provides them with an easy to use framework that can handle any application of differing complexity. Also, if we had questions for how to properly use the framework, there will be a vast amount of resources to solve our problems.

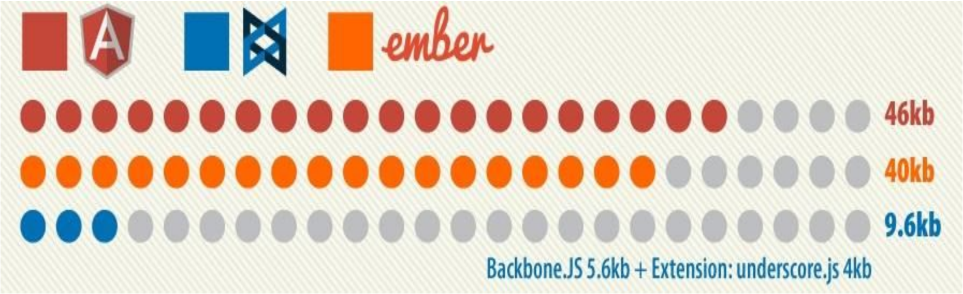


```
Search: <input ng-model="query">
Sort by:
<select ng-model="orderProp">
  <option value="name">Alphabetical</option>
  <option value="age">Newest</option>
</select>


<ul class="phones">
  <li ng-repeat="phone in phones | filter:query |
orderBy:orderProp">
    <span>{{phone.name}}</span>
    <p>{{phone.snippet}}</p>
  </li>
</ul>
```

However, there are some downsides to using AngularJS. One such downside is Angular's complex directives API. In the picture above shows the use of directives, the green colored 'ng-repeat' or 'ng-model'. In conclusion, AngularJS would make a good choice for our project due to its two-way-binding and large community to help us along the way, which would help us overcome the possible downside of the complex directives API.

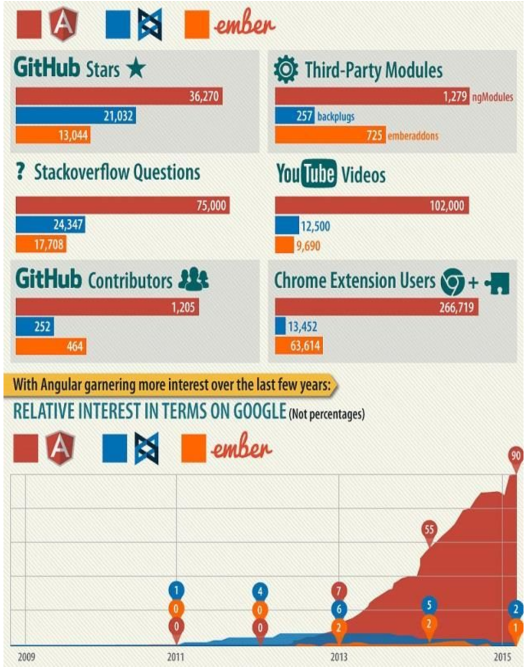
Backbone.js is a framework built with a RESTful JSON interface and is a lightweight MVC framework. Backbone.js gives structure to web applications by providing models with key-value binding and custom events [1]. One of the main differences between Backbone.js and the other two frameworks, which also happens to be Backbone's primary attraction, is that Backbone is much smaller in comparison to the other two. The following chart illustrates the distinct difference in size between all three frameworks.



With a standalone size of only 5.6kb before the extensions, Backbone.js allows for very fast response times for the users. In a world where speed is one of the most important things in web application, Backbone delivers. One reviewer says: "Because it's so lightweight, Backbone shines brightest when used in simpler projects where speed is a priority - think single page apps (Twitter, Pinterest) or a widget that is part of a traditional web application" [2]. Backbone's size allowing for quick response times could also be a downside for new users. The lack plugins and helpful APIs might present issues for our group project since we will have limited time to set aside to learn about the selected framework. Another downside to using Backbone is that it lacks server-side rendering [2]. In conclusion, Backbone would possibly make a good choice for our web application because the small file size allows for incredibly fast response times that our users will prefer.

Ember is another open-source MVC framework similar to AngularJS. Ember attempts to compete with AngularJS through its own version of the two-way-binding system illustrated earlier [3]. One benefit according to one reviewer is that Ember is “continuously strengthening itself with new superpowers. It is coming up with Fastboot.js module that allows server side rendering of DOM” [3]. Server side rending allows for better performance in complex UI rendering. Another benefit to using Ember, especially for new users, is the ability for the Ember framework to self-configure many aspects if users don’t define them themselves [1]. The major downside to using Ember is readability. Through the use of “tons” of script tags, a lot of users are unable to properly or with complete accuracy read the written code. Also, these script tags can break CSS styling [1]. In conclusion, Ember could be a good choice for our group project because of the use of two-way-binding similar to AngularJS and the ability for the framework to self-configure if not previously defined.

The three frameworks researched for this paper: AngularJS, Backbone.js, and Ember are three of the most popular JavaScript frameworks available. They all provide prospective web developers with the ability to create reliable and user friendly applications, with fast response times. However with all of this said, the solution to this problem comes from the need to have a lot of helpful resources and a little bit of prior knowledge of the desired framework. In the end, our group deemed it necessary that we use the AngularJS framework. The vast amount of resources avaible with AngularJS compared to the other frameworks, due its large community as evidence from the following charts.



Angular's two-way-binding, large community with tons of resources, and ability to flourish for complex web applications, are the most attractive reasons for choosing this framework over the other excellent frameworks Ember and Backbone. Our group feels that AngularJS will give us the best possible experience for developing the web application portion of our project, and we cannot wait to begin using it.

CloudKit

Relational database management systems, commonly referred to as "RDBMS", have been a primary data storage system for over four decades. In the early 1970's, two developers from IBM named Donald Chamberlin and Raymond Boyce often utilized these databases and sought out to find more efficient ways to communicate with them. Eventually, the two created a new query based language called the Structured English Query Language to manipulate data stored in relational database management systems. Over the years, as the number of large scale dynamic applications has grown so has the need for more efficient and structured data management. Several companies and software engineers have attempted to tackle this task, and after many years of different implementations and the growth of cloud computing Apple finally released a database migration framework called CloudKit. Working in cohesion with iCloud, CloudKit offers data storage with both public and private databases for developers and users alike. With the default CloudKit plan, users are given 100 megabytes of database storage and 10 gigabytes of asset storage for free [7]. Although for a large-scale application this does not seem like much storage, Apple provides users with expanded storage packages based on the number of users interacting with their application. While planning the development of our application, The Capstoners have considered NoSQL databases, SQL databases (MySQL specifically), and now CloudKit sits on the forefront of the backend design plan. Unlike a database that would be created and managed by the developers, CloudKit provides a plethora of features that would otherwise be very code intensive to implement. While using this backend-as-a-service, also known as a BaaS, developers can count on this framework to handle user authorization and database security so that they can put all of their efforts into the client side of the application. The Capstoners hope that this system will help protect sensitive user data such as financial transactions, and allow users to easily access their accounts from multiple platforms- two of our biggest issues at hand.

The first part of any user dependent application with protected data is the method of authorization, or how a user can log into their account. While using a traditional RDBMS or NoSQL database, this process must be completely and correctly implemented by the developer to maintain security of user information. With the use of CloudKit, user authorization is handled completely by the framework and users are able to just sign in using their Apple ID and password. Although choosing to use CloudKit does restrict the methods of authorization to just an account associated with Apple, we chose to use this framework because you are able to control the level of security for nearly all aspects of the databases. Depending on the design of your app, you can pick and choose different assets to remain either completely anonymous, or allow them to be viewed by a specific set of users; any user that logs in using their Apple ID remains anonymous unless they explicit grant an application permission to share their user information. This amount of added security is always useful when dealing with daily user input that will most likely not be shared with the general public. Also, a huge advantage that CloudKit has over other database management systems is that all of the data will be stored on a shared cloud so that user accounts can be accessed by any other compatible device if need be. Working on a shared cloud allows users of CloudKit applications to utilize the same information that they had access to on their primary device. This is done by allowing the third-party application to save data to the private database, which is automatically synchronized throughout all devices using iCloud [7]. A great use to this feature is the ability for users to continue part of an application from a saved state on any iOS client that they wish to use. As part of our application will be focused on the progression of some type of virtual pet, it is important for the user to be able to track their progress on any device. Another useful feature of a shared space between iCloud and CloudKit is the ability to store structured data sets (such as tables) and entire files such as images and music. Since so much of the data manipulation is handled under the hood of CloudKit, iOS and web developers are able to focus on the front-end of their application as opposed to the whole development stack. With a clean and effective user interface, CloudKit enables us to manipulate assets used and maintained by users.

As shown in the image above, when developing an application using CloudKit you can alter the schema, public, and private databases with ease. The architecture of an application using CloudKit is much different from a traditional design as well. While testing, adding, and changing database schema can all be done in the development environment, the state of the production environment is dependent upon changes made to said assets through the application's services. The flowchart shown below depicts this new type of application architecture that has been created and utilized heavily by CloudKit based applications.

Even though CloudKit seemingly fits the needs of the Capstoners' application, there have also been several other options presented to persist a growing and dynamic dataset. Many, if not all members of the Capstoners has experience building and maintaining SQL databases so naturally we felt at first that this would be our best option. As our vision plan for our application expanded, so did the importance of security and maintainability. Although these qualities could be achieved by our team, they are most like better exemplified by a trusted and widely supported framework. With the help of CloudKit, we are able to put more of our time into developing sophisticated budgeting algorithms and continuing feature expansion instead of worrying about user authorization. Another reasonable- and arguably better choice for storage would be Parse, an open source cloud database with even more impressive features than CloudKit. Unlike CloudKit, Parse allows developers to backup data in JSON on your own local server, and also offers data analytics of user behavior [5]. These analytics could be used to monitor the traffic of users, compile general information about application usage, and track crashes- all of which are not offered by CloudKit. Unfortunately, we chose not to use this service because it is being discontinued as soon as January of 2017.

In conclusion, CloudKit is the best option for the Capstoners to use in order to build and maintain a scalable dataset. By using the framework, we are able to focus more on the client side of our application, and let it handle the privacy and authorization of user information. Also, it will allow our users to track the progression of their savings goals from any device that allows them to sign into their iCloud account (web and mobile). As we continue to design and eventually develop our application, the features of CloudKit will allow us to control and maintain our dataset with ease. Looking forward, we hope to see further improvements made to the service so that it is of even more use to us as developers.

Graphics, Animations, and Gaming Engines

A primary component of most any user facing computer application is its graphical user interface. Without a GUI, users are left with nothing more than a primitive text based input and output mechanism. We plan to leverage the power of high quality graphics and animations to develop an application that creates emotional value with a user’s savings goals. Doing so will require at least a basic knowledge of 2D graphics animation and gaming engines to create an environment for these “pets” that will represent a user’s savings goal.

For the iOS platform specifically, there are several options for 2D gaming graphics. The first option is the Cocos2D SDK. Cocos2D is an open source gaming framework written in C++. 1

It has many features including graphics engine, animations, particle effects, and shaders. However, there is a considerably high entry point to using Cocos2D despite being open source. First, it is not natively integrated into Apple’s IDE, Xcode. It also does not include any physics simulation, which is crucial for developing interactive and life-like animations. Extending the functionality of Cocos2D to include physics simulations would require the installation

Furthermore, use of the Cocos2D would require intermediate to advanced knowledge of C++. For these reasons, we shall move on.

The next option is Apple’s own SpriteKit framework. Similar to Cocos2D’s graphics engine, animations, and particle effects, SpriteKit also includes physics simulation. This addition however comes at the expense of losing built in shader support. This is a trade off that we are willing to make since we are more concerned with life-life physics and can live with primitive lighting and shading, especially in the context of 2D graphics rather than 3D. Additionally, SpriteKit is natively integrated into Xcode, allowing for easy of entry into this framework which means less time installing and configuring a framework. This of course affords us more time to develop the application. Furthermore, because it is already included in the IDE and supported by

Apple, SpriteKit natively supports both the Objective-C and Swift programming languages. This alone is a major advantage as several of our team members already have experience with one or

both languages. Below is an overview of the feature sets of both SpriteKit and Cocos2D:

Out last consideration for graphics and animations is the ubiquitous Unity game engine. Unity is by far the most capable of all the considered frameworks and platforms. While its bread and butter are primarily in the realm of 3D game development, Unity can also be utilized for 2D games. Probably its most advantageous feature is the ability to develop once and build for multiple platforms such as iOS and web. This could cut down development time significantly and guarantee that both of our platforms provide similar experiences. However, Unity, like Cocos2D comes at a cost. Unity does not offer integrated or native support for iOS. This would require development time be further split between web development, iOS, and Unity environments with the additional overhead of later integrating Unity builds into the other two platforms. Additionally, Unity would also require the use of a third programming language - C#. This additional overhead, both the additions of programming languages and IDE's, leads us to believe that Unity would push the scope of such a project too far given the time constraints. Because it is native to the platform that we are primarily developing for, SpriteKit has absolutely no overhead to purchasing, installing, or developing for such framework as is the case

with Unity and Cocos2D. Another added benefit of such nativeness is the minimal number of languages that we will be required to learn. While the team does have some knowledge of C# and JavaScript (the latter of which can also be used to develop Unity games, albeit an internally defined variant) 2 , Swift remains the primary language of choice and represents the most well known of the available development languages for the iOS platform. SpriteKit also has all the features needed without the bloat of all the features we do not - something that cannot be said of

Unity. Graphics engine, animations, and simulated physics will be our primary uses for SpriteKit in order to create pets and their environments along with ways to interact with both. All things considered, I believe that SpriteKit is the correct choice for this project.

Performing Transactions through a Web Applicaiton

A core element of our project, PettyCash, is based off of financial transactions and queries being able to take place. These transactions would then allow our user to track their spending habits, develop budgeting goals, contribute to their Savings Pet and more to aid them in fiscal responsibility. A common issue with online financial transactions though, is with the privileged security that is usually required to have any access at all to them. Many banks and financial institutions do not have APIs or front facing portals for data about their accounts to be found, even if you know the access information. This could be one of our biggest issues and also a catapult into putting our idea into motion if we can find a way to be able to handle the monetary transfers over our application for little to no startup costs or fees. I will now present a few of the methods we plan to solve this issue and implement our application.

The first option that I came across to possibly solve this problem is using an open source service called OpenACH. OpenACH, from their website, states that they are “the world's first free, open-source, secure web-based ACH origination and payment processing platform... Payment processing made simple.” [11] For a little background context, every transaction that occurs, whether through credit and debit cards, or account transfers/bank movements goes through an extensive network called the ACH. ACH stands for Automated Clearing House and is a way for banks and financial institutions to easily coordinate and handle the constant flood of money being credited and debited from accounts as well as a way for payment processors to split up each transaction from their Point-Of-Sale systems in order to send them on their way to each institution for which they're addressed. [12] The group that maintains this network is called the NACHA and luckily, membership is not required in order to process these documents, and is mostly limited to the larger banks.

For the Implementation of OpenACH it gets tricky. It appears to be a product in its early stages of production and there isn't a whole lot of useful information on how to get the use of OpenACH running quickly and easily for a 3rd party service. The service relies on what they call a BYOB –"Bring your own Bank" payment processing platform. This platform means that we would need to link the backend implementation of our web application to be able to send ACH requests to this partner bank's ACH gateway of some kind who would then place the file on the ACH network for us. Interesting to note here that there is no mention of costs being incurred, but I have found out that the costs would vary from bank to bank depending on how their plan to rate ACH entries is made. [13] Most likely this would involve a per-ACH file fee being deducted from a specified account. For this reason I do not believe that using OpenACH as our payments service would be the best option to move forward on.

After deciding that implementing OpenACH may be a larger feat than originally hoped for, I looked again at what we needed out of our transaction service. I came to the conclusion that our Savings Pet would need to be either an intermediary account or simply a placeholder for keeping track of how much money is to be moved at the end of the day/week/month, whichever is decided to be the best duration.

This led me to investigate both the PayPal API, and also a mobile payments platform owned by PayPal Holdings inc. called Venmo. First off, the PayPal API is a way for web applications or e-commerce stores to implement payment solutions through their own front-facing markets. This would be useful for us in order to allow the user to sort-of be able to move money around, yet would be a large hindrance because the money would one, have a processing fee of 2.9% applied and also the money would have to go to another account held by us. This implementation would also seem like a stretch for us since we would not like to have the payments deposited into our own account if they are supposed to be going back to the user's savings account. Venmo, on the other hand, might have exactly what our project requires in terms of a middle-man service. The idea behind Venmo is that mobile payments or even e-payments can happen directly over their network through their secure bank-level encryption and processing. [14] What appeals to me about the use of the Venmo API is to allow our user a place to store their money in which will be separate from their checking account in which it is being pulled and also be able to be moved around in the case of emergencies or cash-outs. Venmo would solve these issues, yet we could only use this solution for our development as it is unclear how we could levy Venmo with more than one user securely.

Another alternative that we have for development would be to mock up our bank account using a remote database with various tables that would describe transaction histories, account balances, and using serialization we could even pull data from that database across to ours in which we would use as our intermediary for the Savings Pet tool and also use the separate 'Mock Bank Account' server as a RESTful API which we could query for our dynamic data in the various UI elements including the budget tracking features, and the interactive savings pet.

Overall, this topic is going to need to be further explored in how we could move it to a commercialized and finished product with the correct amount of security so that the users would trust it. I strongly believe mocking up our bank data and use of Venmo is going to be easiest option in order to focus more on our implementation of the application logic, rather than try to learn all about the payment processing platforms without industry guidance.

Alternate Strategies:

Many of the alternate strategies of design were referenced above in our research on performing transactions over web applications, and we will go into more detail here.

Our main design strategy invokes the use of a free standing web server running our javascript framework and referencing CloudKit along with our localized database for persistence of other data. Our first alternate strategy for the implementation would be to include scripts on our web server to be our middle-man process of querying a payment processing API and then using our application logic to handle the returns from it.

On the other hand since our iOS implementation is very important to our design, another strategy would be to try and integrate the Venmo mobile payments solution into the application if it will allow us to query the account and receive data from it.

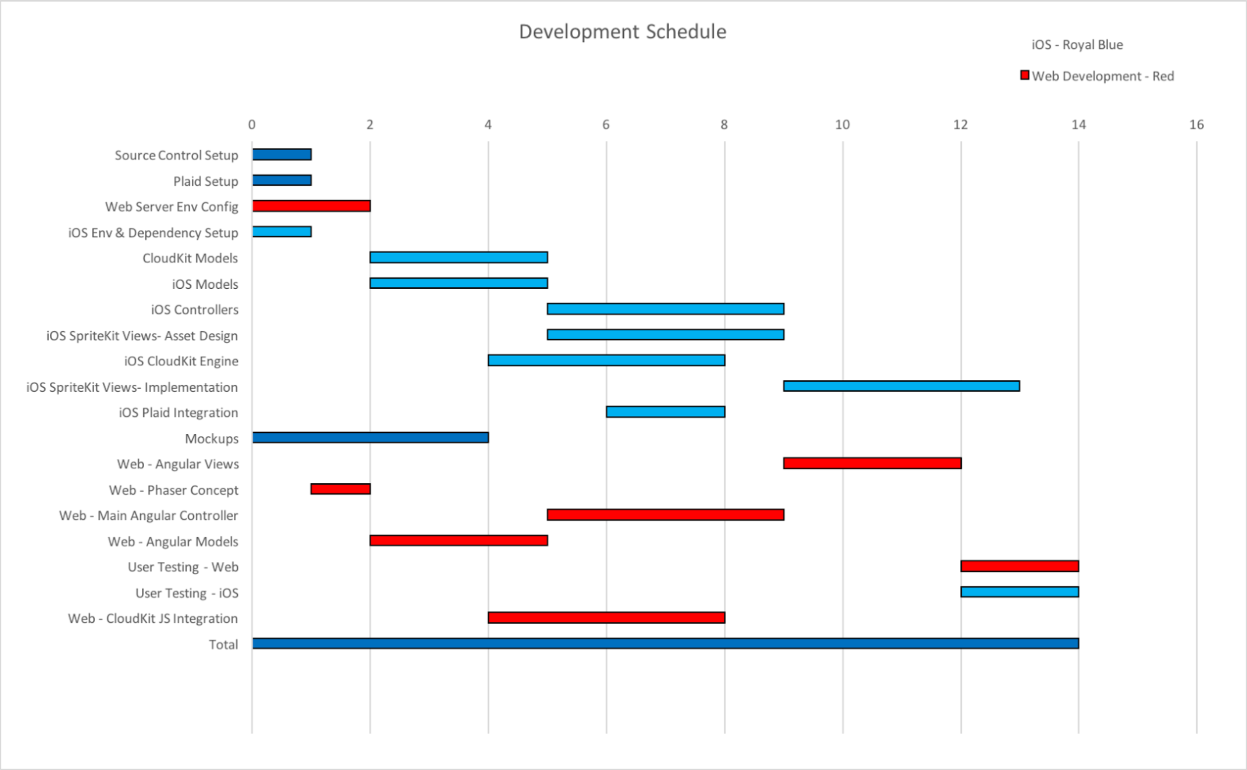
Our backup plan strategy is to mock out our data using a separate web-server + database and have that be accessible via its own API endpoint so that we may query data and transaction info from it and focus on implementing our product.

Development Strategy:

We will be utilizing the Agile methodology on our project. The main reason for choosing Agile as opposed to the other strategies out there is due to our team members' familiarity with it throughout our jobs/internships. We also believe that it will be the most flexible with our busy class schedules, which will allow us to more effectively meet our sprint goals.

In order to test our product during development, we will rely on a few different testing methods. For the iOS application we will heavily UI test using the built in XCTest Framework that Swift contains to ensure a well-written application. Additionally, the iOS side will explore options to utilize tests when accessing our payment processing portals and backend. For the web side, we will rely on the Jasmine framework for Angular JS unit testing. Angular also contains built in dependency injection protection when using its core libraries which will provide even further protection.

Development Timeline:



Work Delegation:

- **Josh Osteen:** CloudKit setup and administration, iOS Development, Mobile UX, iOS Unit Testing, Local Data Persistence
- **Alex Foster**: Web Styling (CSS), Angular MVC development, Plaid Standup, JS Testing, Mock Data Acct standup
- **Emeka**: SpriteKit Development, iOS Development, Mobile UX, iOS Unit Testing
- **Ben**: Phaser JS animation and teaching, Angular Models, SpriteKit assets into Phaser, JS Testing, Server Setup and Admin.

Testing Plan:

I. OVERVIEW

This document provides a description of the tests we intend to run on both our web and mobile applications. The tests are a combination of quantitative and qualitative measures intended to ensure consistent performance, usability, functionality, and security across both iOS and Web platforms. While development is in progress, the testing guidelines rubric will contain comments and any changes made in order to pass the tests. Upon completion, the testing guidelines rubric will include the results for all of the testing scenarios.

II. SOFTWARE

- ❏ Performance
  - ❏ We intend to test the components of our application to measure its scalability and response time when used in different Internet browsers as well as on our mobile devices. Tools such as Webpagetest.org as well as Google's web inspector will be used to record the response time of our web application and the loading of required sources..

- ❏ Compatibility
  - ❏ These tests will be performed on the platforms and Internet browsers mentioned below:
    - ❏ • Windows 10 - Chrome 53
    - ❏ • Mac OS X 10.12 - Chrome 53
    - ❏ • Mac OS X 10.12 - Safari 10
    - ❏ • iPhone 7, 6/6s, 5/5s/5c (iOS 9+)

- ❏ **Reliability**
  - ❏ The web application along with its middleware backend service is hosted on Amazon Web Services EC2 Load Balancing server. The application is utilizing Apple's CloudKit authentication service as well as the Plaid financial data API, which are hosted privately within those companies. The application should not go offline unless the server goes down and AWS is unable to transfer services to another healthy instance, or if there is an interruption with Apple's authentication or Plaid's API service.

III. IMPLEMENTATION

- ❏ **Performance**

  To measure the performance implementation we will utilize the following categories:
  - ❏ **Web-App**

        - ❏ Script load performance when multiple clients send requests
        - ❏ Testing of specific javascript snippets to iterate through data to decrease completion time will be done utilizing JSPerf http://www.jsperf.com/
        - ❏ Overall load times of resources utilizing Google's developer tools to narrow which specific API requests are limiting load times. Guide:
            - ❏ Since AngularJS is an entirely client-side framework, we will use timing tools to evaluate which processing-heavy scripts we can perform on the backend AWS instance instead.
        - ❏ CSS is minified when exceeding 100 lines in length for performance testing.
    - ❏ **iOS**
        - ❏ The ability to request and view client transactions and goals without any visible delay from the server.
        - ❏ Reasonable comparative latency to our web application; Clients should have a uniform experience between the two platforms.
        - ❏ Utilize changes in Swift to make authentication faster and requests to the server more efficient.
- ❏ **Usability**

  To measure the usability of our applications we will utilize the following categories:

    - ❏ **Web-App**
        - ❏ Landing page is intuitively designed with information about our product and how it will help individuals. (UX)
        - ❏ Post-login, the dashboard, animation displays, goals and other elements will display within the main usability area (no elements riding the edge of pages, or unnecessary popups) and follow guidelines and standards reccommended by the W3C https://www.w3.org/standards/webdesign/ (UI/UX)
        - ❏ A consistent navigation bar at the top of the page for ease of use. (UX)
        - ❏ The process of adding bank accounts, creating goals, and monitoring progress will be easy enough so that the learning curve can quickly be overcome. (UX)
        - ❏ Web Appearance correctly scales between 10" - 23.5" displays without issues. (UI)
    - ❏ **iOS**
        - ❏ Minimalistic yet modern mobile design that makes the application intuitive for new and existing clients.
        - ❏ When applicable, we will take advantage of the larger iPhone sizes (ex. 6 PLUS, 7 PLUS) to make the user interface more enjoyable for the client.

    - Clean sliding menu will make it easier for clients to access the information they desire and make changes to their account.
    - Similar to the web application, goals and newest transactions will be shown on the client’s landing page.

- **Security**

  To secure of our applications with our limited resources we will follow these guidelines:
  - Authenticating through our web application will follow Apple’s Best Practices for use with the CloudKit Authentication and data persistence service. No privileged data is visible or kept in public datastores.
  - Retrieval of actual banking information through the Plaid service will only be done on specific test real accounts or using the Plaid Tartan test environment.
  - Denial of access to non-navigation related html/JS files through the web server.
  - Sanitize data and prevent cross-site scripting access from unwanted scripts

- **Functionality**

  To test the proper function from our applications we will perform the following:
  - Test all forms
    - Client-side first field Validation
    - Authentication handling from CloudKit
    - Intuitive and explanatory errors for invalid input
    - Optional and mandatory fields
  - Testing CloudKit datastores
    - Data pulled correctly from individual accounts
    - Save, Delete, and Update records
    - Stress testing with large amount of test data (>100 records per -store)
  - Testing Plaid API
    - Ensuring data for all endpoints returns in proper format
    - Persistence of ‘access_token’ client-side to maintain speed when requesting data through the AWS backend middleware.
  - Testing regular use
    - Test multiple end-to-end scenarios for quality assurance
    - Adding to goals within the application persists data and that data is returned for view upon session close then reopen.
    - Data is consistent across Web/iOS platforms

Security:

Data integrity and system security will be vital to this project. As with any financials based application, users will expect that their information is managed in an appropriate manner. This is just another reason why we have chosen to use CloudKit for backend data persistence and transport.

CloudKit is simply an extension of Apple’s iCloud storage and cloud computing infrastructure. Thus, to use our application, a user must be signed in to their iCloud account using the appropriate iCloud email and password. As previously mentioned, this is a system login within the Settings application on iOS devices. For the web component, users will be redirected to a secure Apple login portal where the same credentials will be used for login on the website. Once the credentials are validated by Apple, users will be redirected to their personal landing page.

The use of iCloud accounts places the responsibility for verification and credential storage solely with Apple. This scenario is ideal for several reasons. First, Apple has the necessary resources to ensure a high level of system security that should meet and/or exceed the expectations of users of the application. Second, this will aid in ease of use for users since they will be able to use account credentials that they are already using for iCloud. This is especially true for the iOS application as users will not be required to login inside the application if they are logged into their iCloud account through the device. Third, and as mentioned in the CloudKit ERD section, a further level of system and personal account security is added in the use of the CloudKit private database. The application will store a user's relevant personal financial information that is critical to application logic and functionality in this private database.

The database is aptly named due to its exclusive access by validated iCloud accounts, namely the currently logged in account. This will not only allow users to sync their data across multiple devices using the same iCloud account, but it also distributes sensitive financial information across each individual user's account rather than relying on a centralized aggregate repository of all user's information that would be a literal goldmine of information.

When considering the Plaid API to be implemented on the web server, we will be securing all data transactions with SSL at minimum. Because this application is merely a proof of concept, we will not be implementing any further security measures in regards to federal or state financial regulations and security standards.

We believe that the above outlined security measures will be sufficient for the duration of development and initial release. If the possibility of further development and maintenance arise after initial release, the team will reevaluate these security measures and make changes accordingly.

Results:

The results of this project could have been better. On the mobile side of the project, essentially every feature that was planned was included in the application. Goals can be created, the pet can be interacted with and these interactions contribute towards the goal's completion. The user can also view bank account transactional info from Plaid's API. This data is then visualized to allow the user to better see what areas of their lives they are spending the most money and where to potentially cut back.

On the web side of the things, goals can be created, the pet can be interacted with and these interactions contribute towards the goal completions just like the iOS application. The user can also view bank account transactional info from Plaid's API.

Future Work:

In the future for this application, we will probably implement full bank account linking rather than just demo data from Plaid. Another possible implementation is the ability to predict future user spending habits and report these to the user so that they can be better informed on the balance of their account going forward.

Works Cited

1. http://www.improgrammer.net/most-popular-javascript-frameworks-2015/
2. https://www.lullabot.com/articles/choosing-the-right-javascript-framework-for-the-job
3. http://noeticforce.com/best-Javascript-frameworks-for-single-page-modern-web-applications
4. https://docs.angularjs.org/tutorial/step_04
5. https://yalantis.com/blog/10-reasons-why-cloudkit-is-worse-than-parse
6. http://stackoverflow.com/questions/24069221/cloudkit-no-server-side-logic

7. https://developer.apple.com/library/ios/documentation/DataManagement/Conceptual/CloudKitQuickStart/EditingSchemesUsingCloudKitDashboard/EditingSchemesUsingCloudKitDashboard.html#//apple_ref/doc/uid/TP40014987-CH5-SW1

8. http://www.macworld.com/article/2455127/why-you-should-care-about-cloudkit.html

9. http://www.cocos2d-x.org

10. https://www.raywenderlich.com/61532/unity-2d- tutorial-getting- started

11. http://openach.com/

12. https://en.wikipedia.org/wiki/Automated_Clearing_House

13. http://openach.com/blogs/admin/mon-01162012-1614/us-bank-becomes-first-openach-bank

14. http://www.investopedia.com/articles/personal-finance/032415/how-safe-venmo-and-why-it-free.asp