CSCI 4448
Homework #9

Ben Limmer &
Zachary Clark

Summary

Throughout the course of the three iterations, we produced a final system true to our original mockups. The system allows you a registered user to add three widgets: calendar, bus and weather. The object-oriented nature of the design allows for this prototype to be significantly expanded upon with little to no maintenance of existing code.

The following sections relate to how the system has changed since our submission to Homework 6:

UI Mockups: our final system is very close to the original UI mockups we submitted in the sixth homework assignment. Instead of gesture-based control of the settings page, we opted for an interface that did not respond to gestures (having an edit button instead of requiring a gesture). Other than this small deviation, we implemented the system to match our UI mockups as closely as possible.

Data Storage: as discussed in Homework 6, Ruby on Rails handles interactions with the database, so the specific database used on the back-end was not of great importance. Our current production system utilizes an PostgreSQL database, but it can easily be swapped out for another flavor of database because of the interaction between ActiveRecord and the database in Ruby on Rails.

Sequence Diagrams: the interactions with the system match the sequence diagrams from Homework 6.

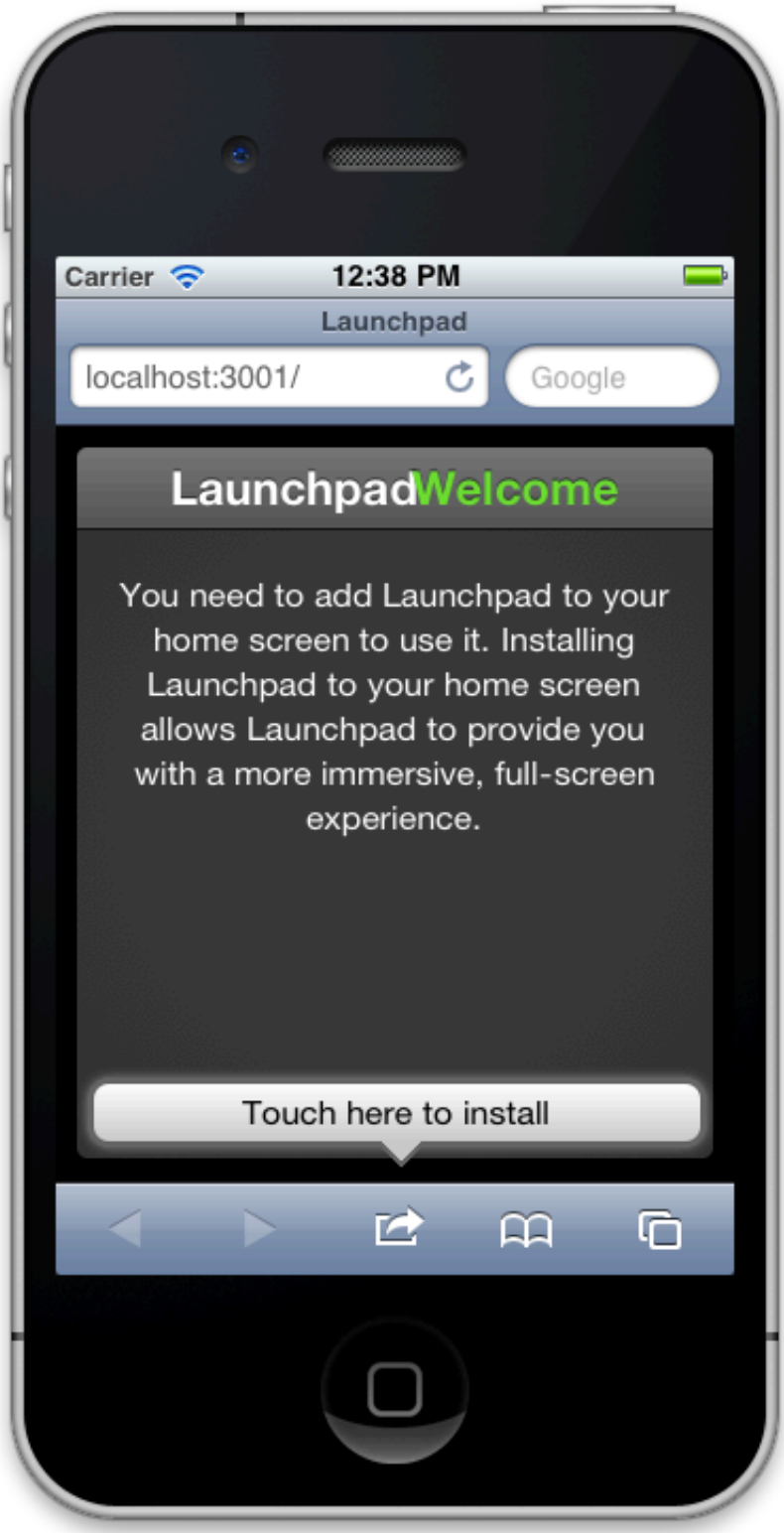
Architecture Diagram: the final architecture of the system matches that discussed in Homework 6.

Class Diagram: please see the class diagram later in this submission to see how its changed. The main differences are in naming (for methods and attributes), as well as the specific implementations of each widget's data and data runner's. These changed when we got far enough to begin seeing what sort of data the APIs we used were returning. We also added a new controller (HomeController) to handle a few actions that didn't make sense within the WidgetsController, such as the api endpoint for remembering which widget the user was last viewing.

All our work is available at https://github.com/spyyddir/launchpad. You can find our source code in addition to a timeline of commits and issues resolved by Zac (spyyddir) and Ben (l1m5).

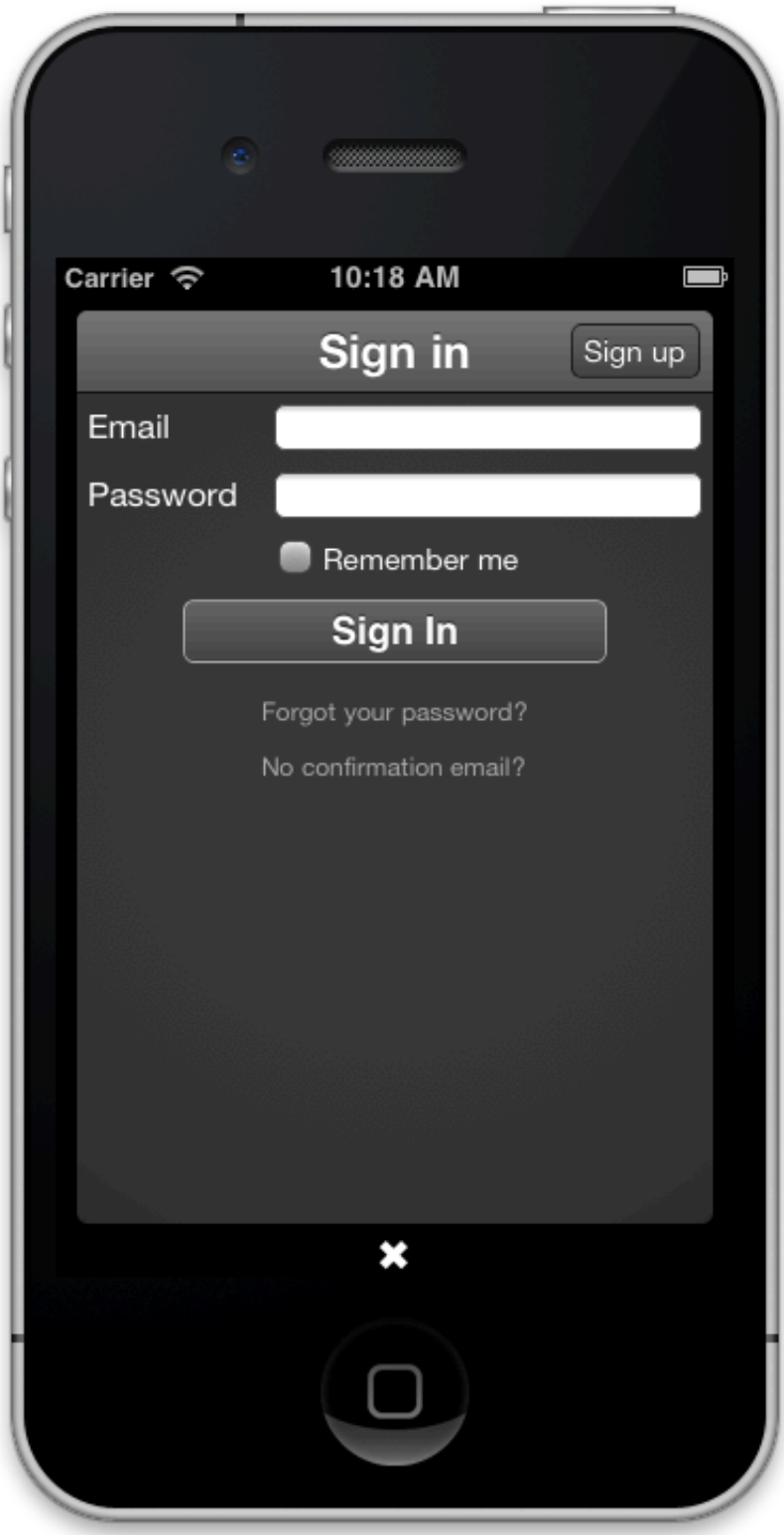
Additionally, you can navigate to our live application hosted at http://oolauchpad.heroku.com.

All the following images are screenshots of the iOS simulator running our installed app. None of the following are mockups.

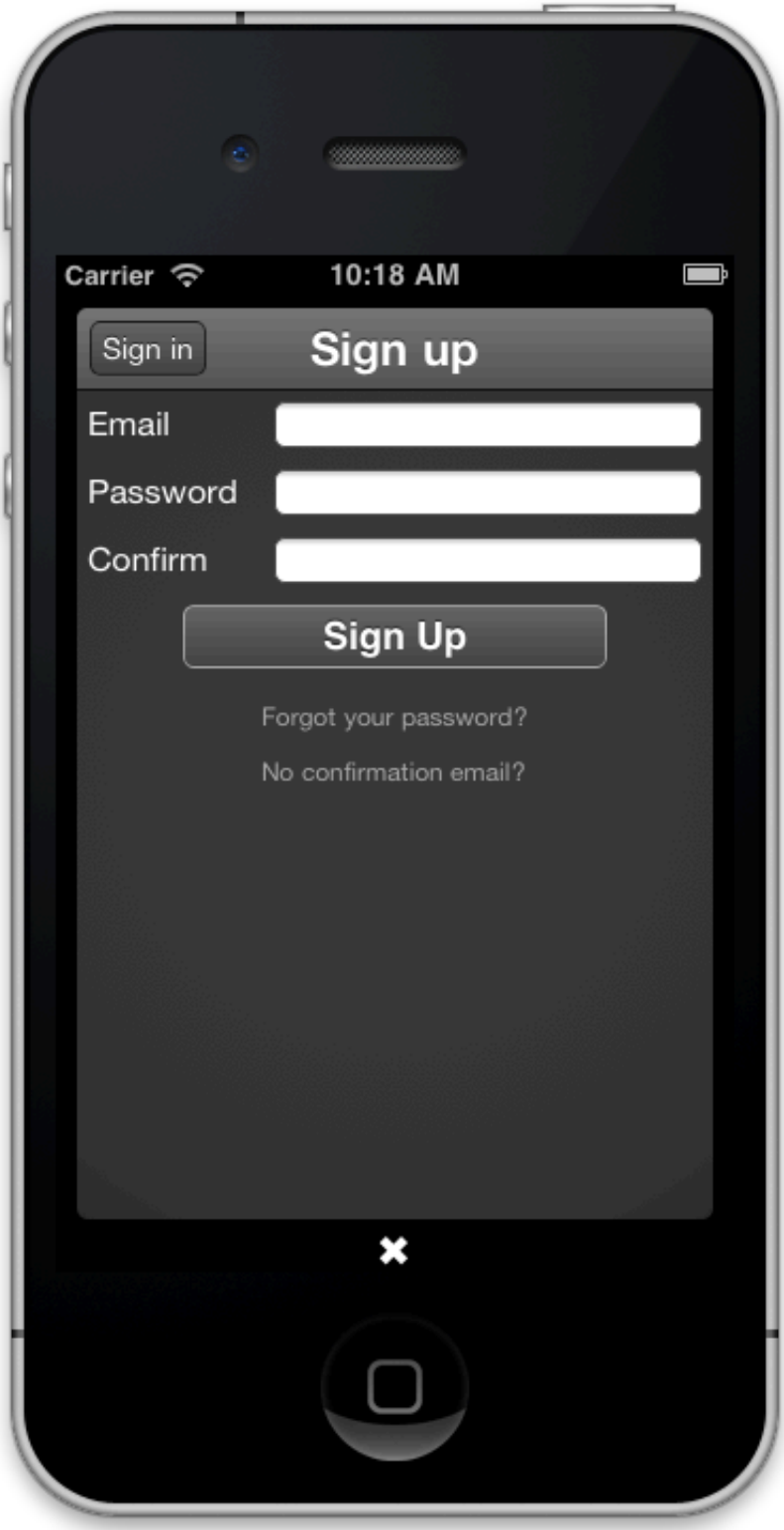


If Launchpad is browsed to using Mobile Safari, this page is displayed informing the user that they should install the application to their home screen for a better browsing experience. The tooltip shows the user how to go about installing the application.

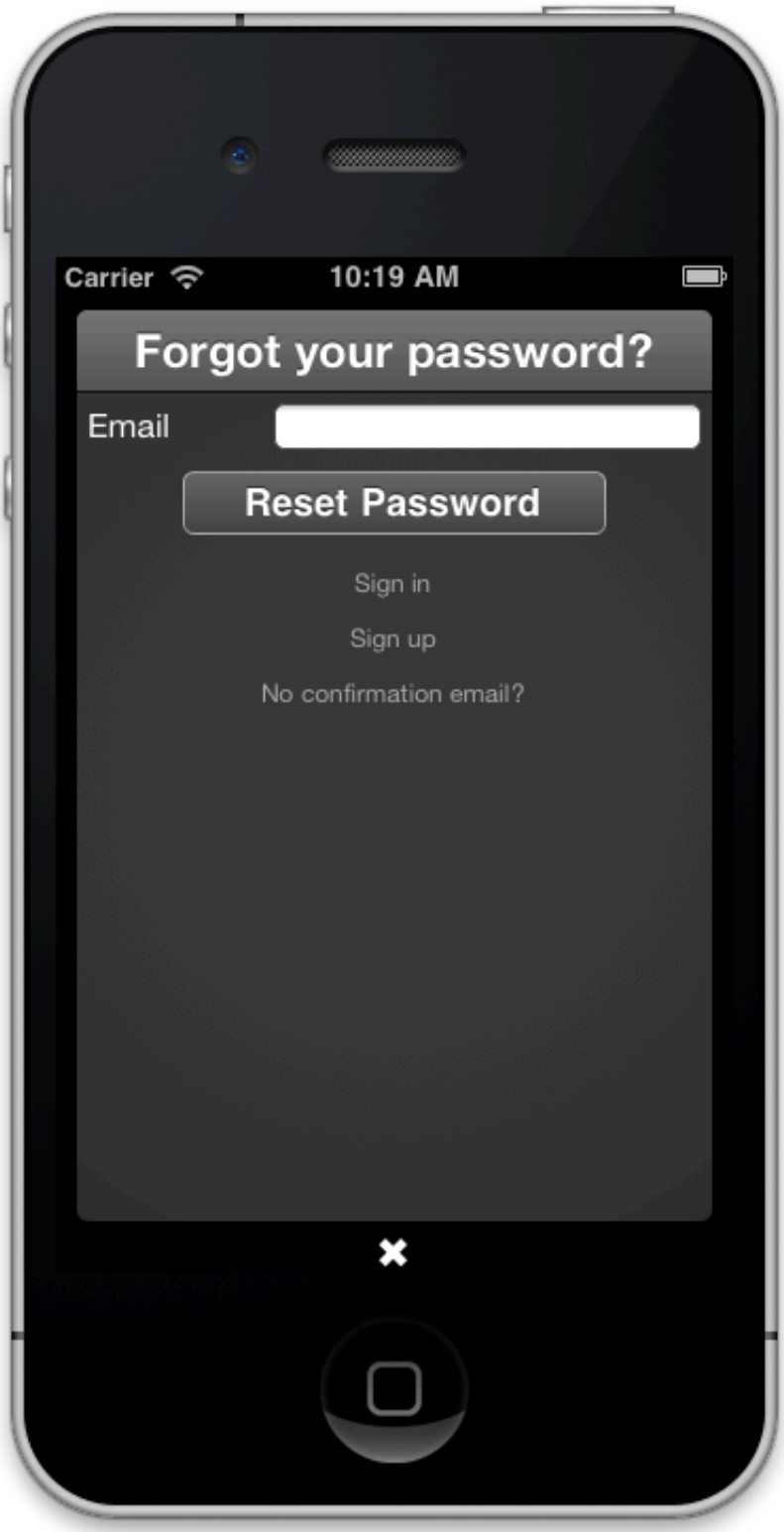
This is our neat-o splash screen that comes up as the application is loading. Its awesome-ness can't be described in words.

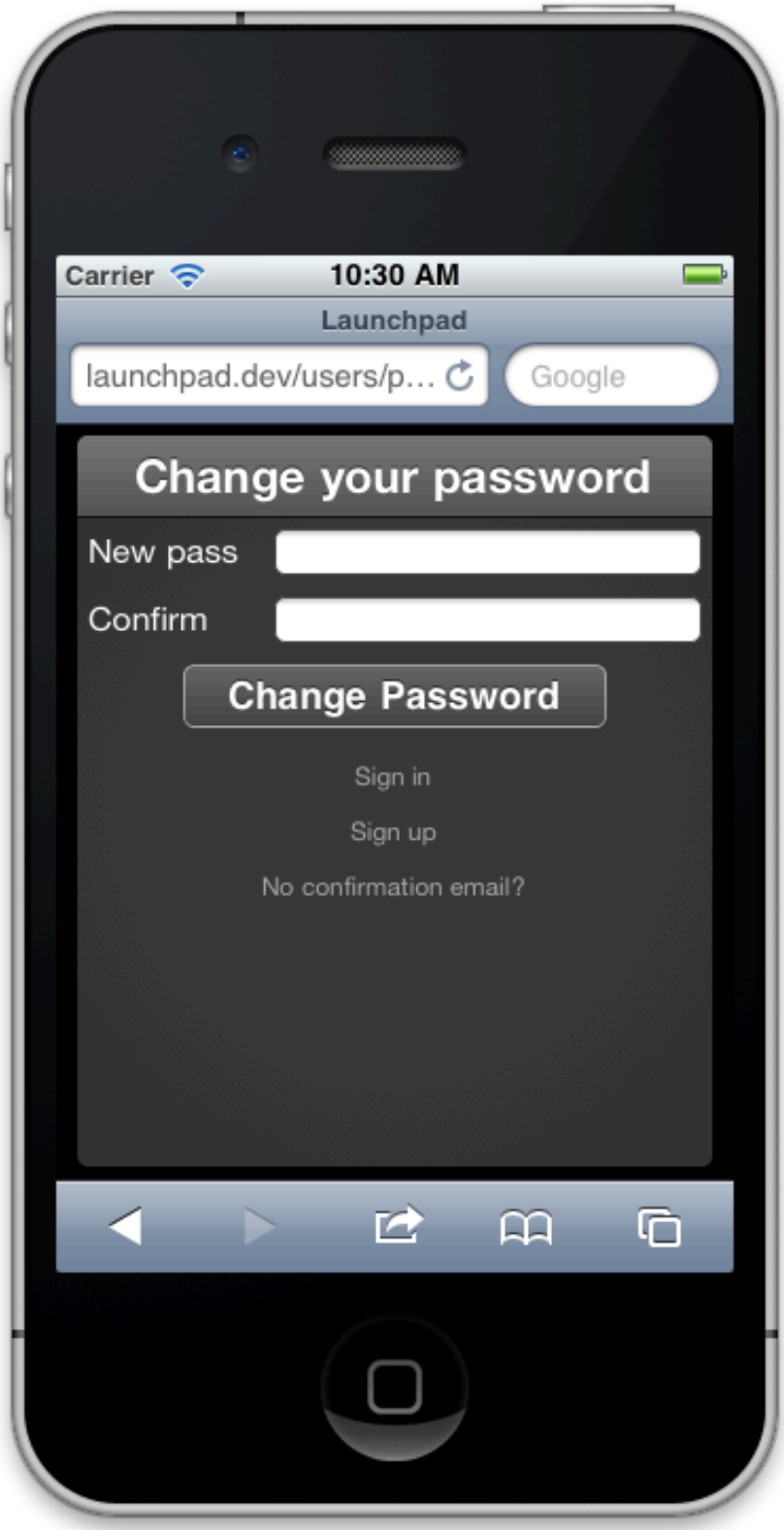


Once installed, the application presents this screen for the user to sign in or sign up. We tried to mimic Apple's design and stick within the guidelines presented in the HiG. All the tap targets are reasonably sized, and the positioning of UI elements falls within the Apple norm.

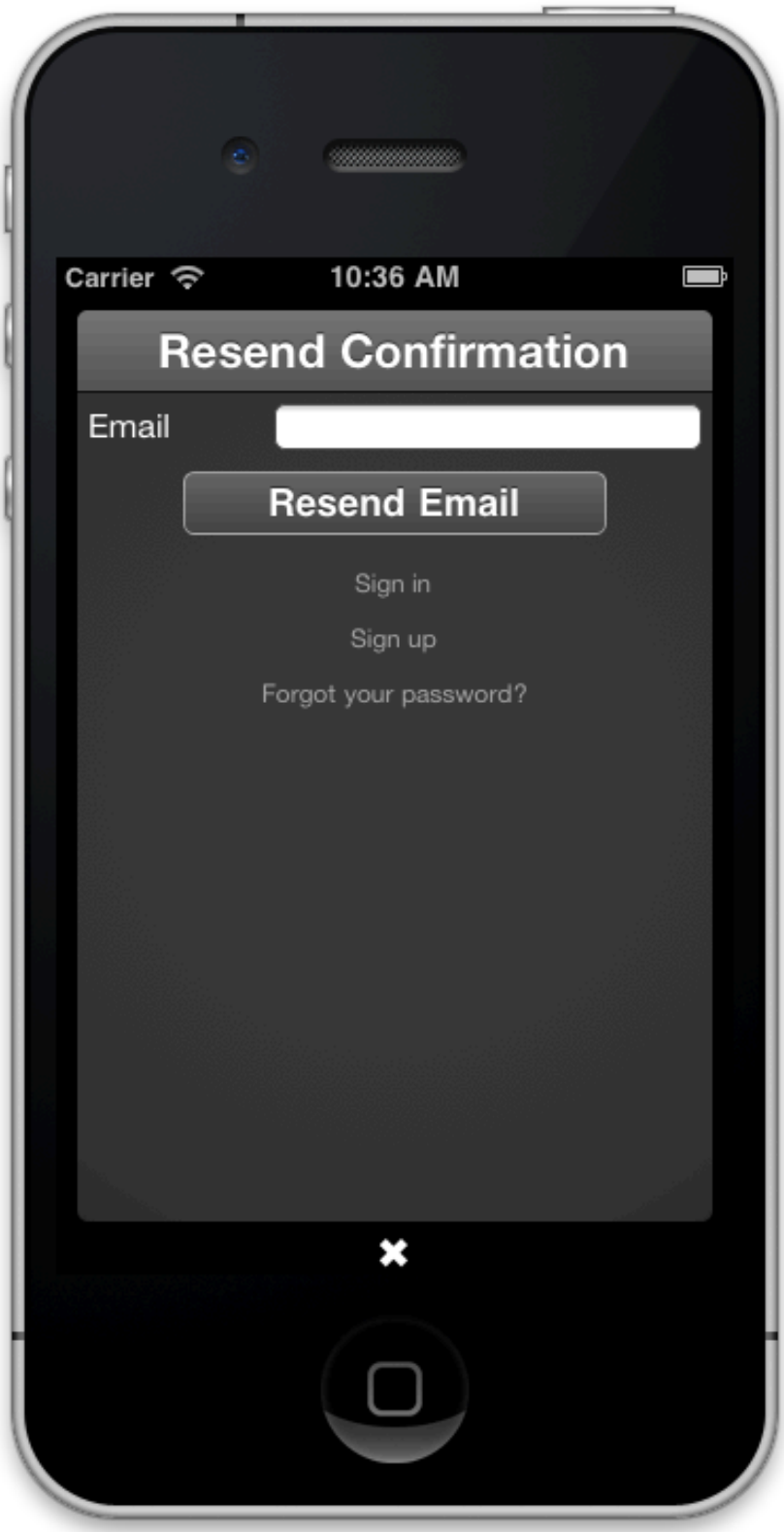


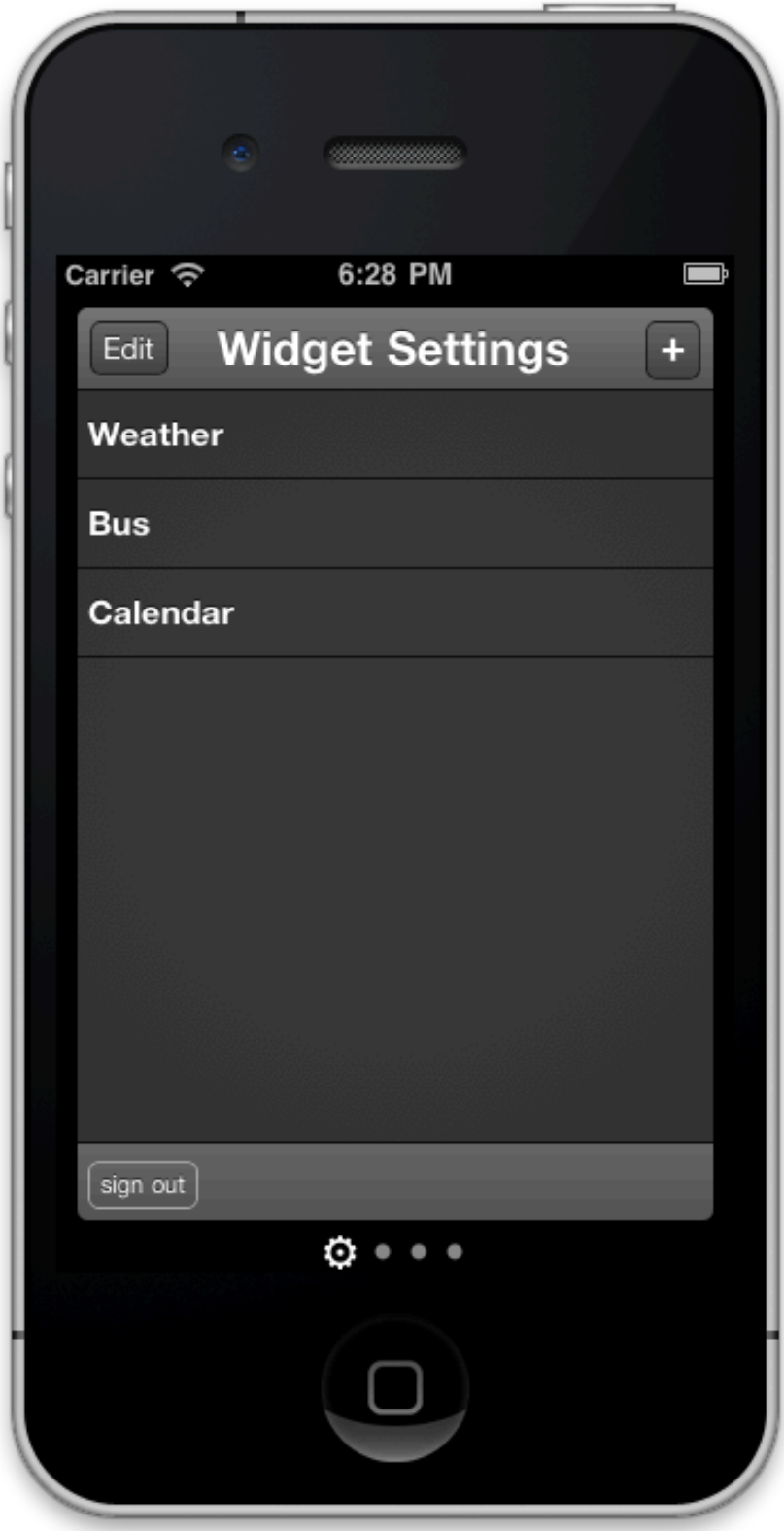
If the user taps "Sign Up" on the previous screen they are taken here. We decided to keep the number of required fields to a minimum to facilitate easy sign up. The next few screen are all based on Devise for the authentication system, and are self-explanatory.


Carrier
10:19 AM
Forgot your password?
Email
Reset Password
Sign in
Sign up
No confirmation email?



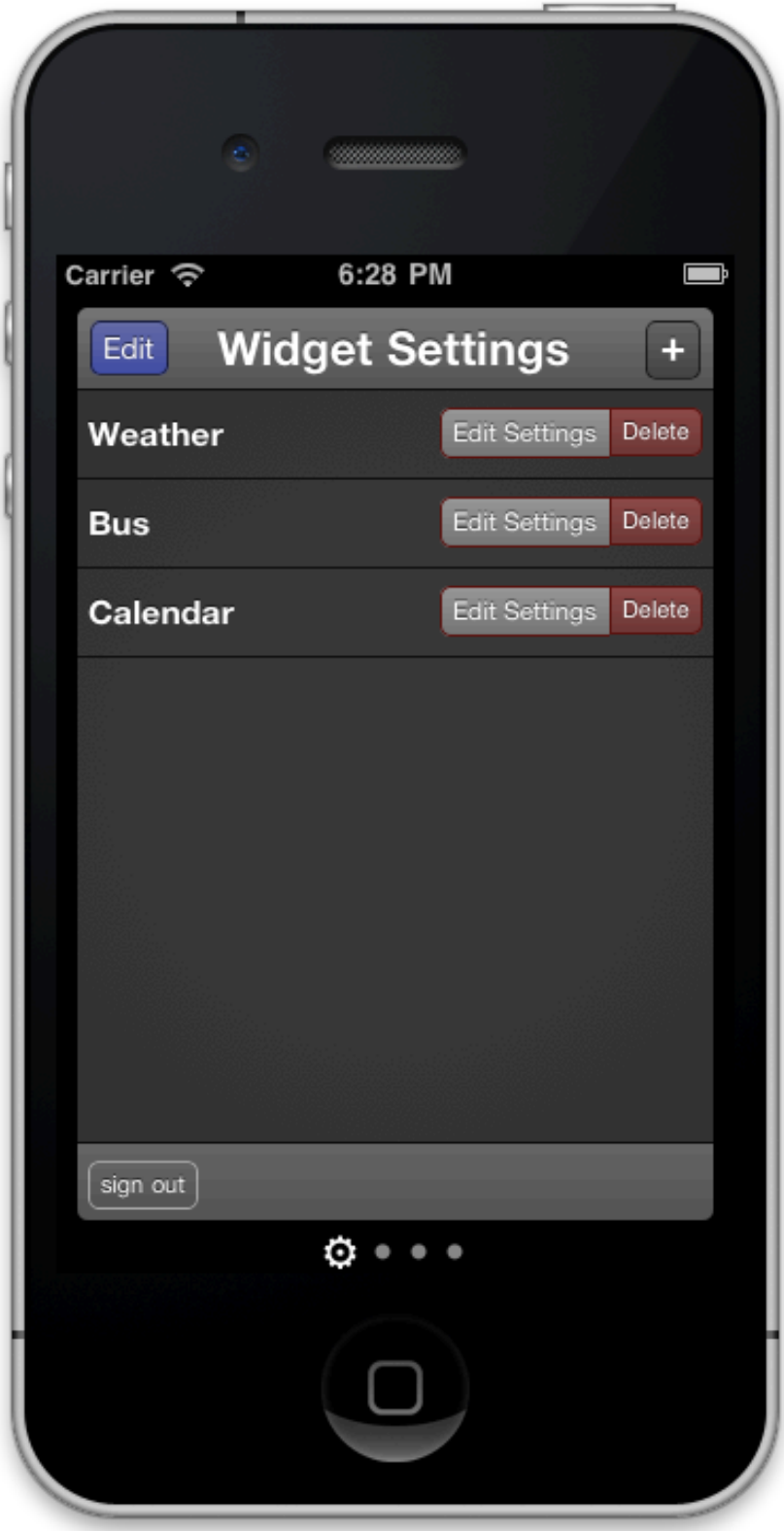
This screen is in Mobile Safari, rather than the installed app, because this screen would only be reached by a user clicking a "reset my password" link from within an email. iOS does not provide a way for links to open in installed web apps, so this screen was built to fit within the UI of Mobile Safari.


Carrier
10:36 AM
Resend Confirmation
Email
Resend Email
Sign in
Sign up
Forgot your password?



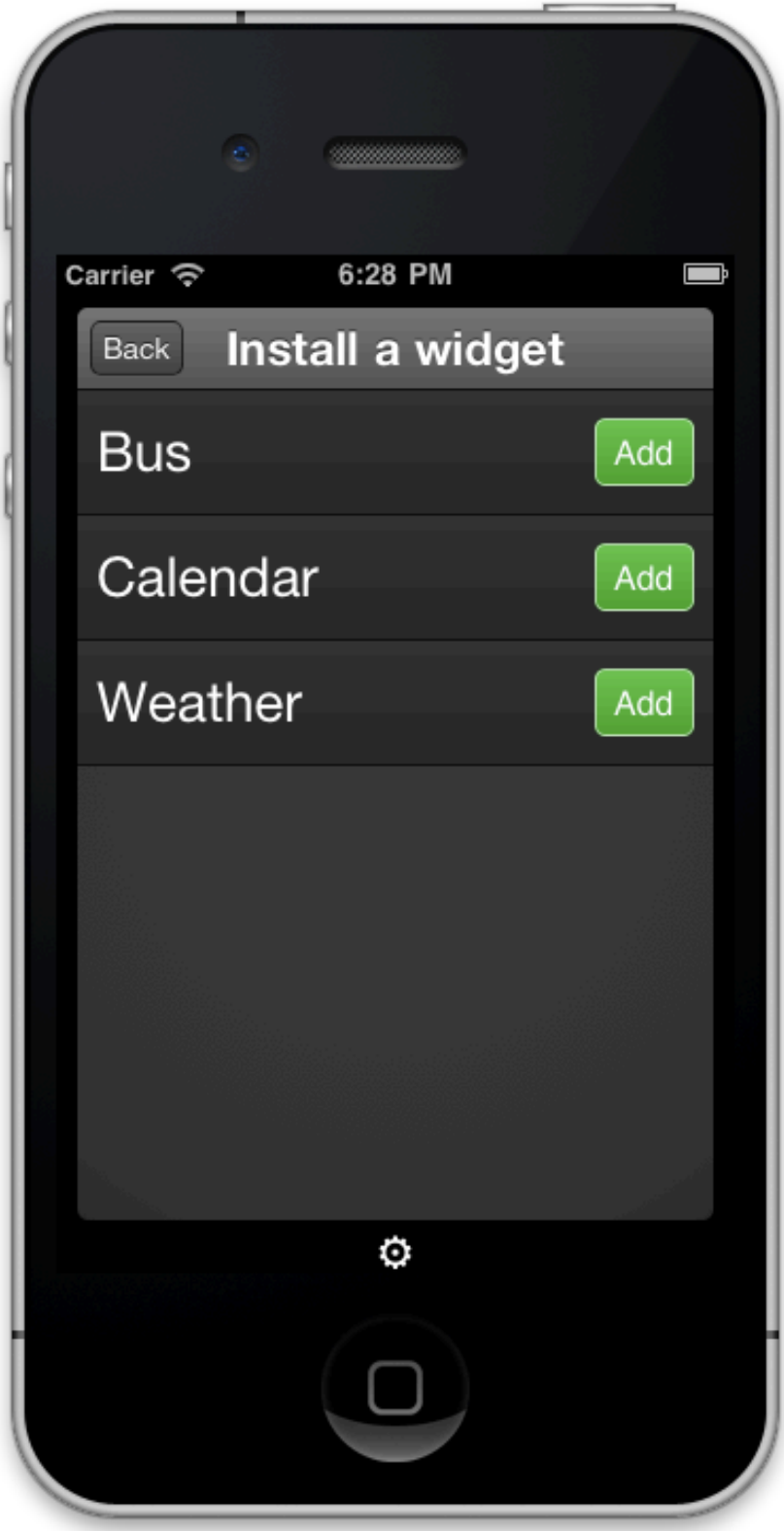
This is the "Settings Widget", which lives as the farthest left panel of the set. We put Widget in quotes there, because its a rather special case. It is designed like a Widget, but its functionality is to edit the settings and add and remove the user's choice in Widgets. You can see in this screen that the user has 3 Widgets.

Also, if the user wishes to sign out, we have let them do that at this screen.

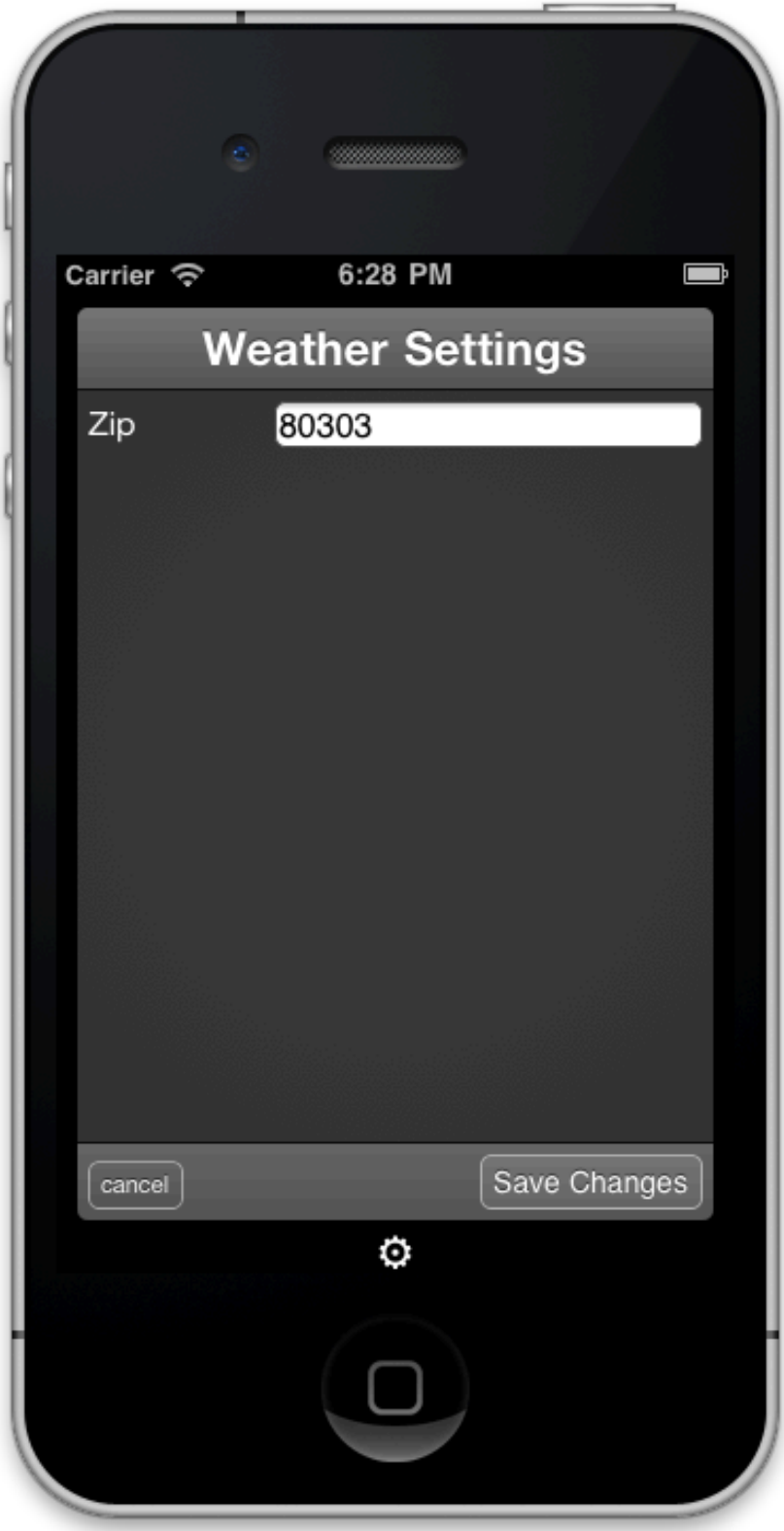


This is the settings Widget with the "Edit" toggle turned on. Inspired by Apple's default installed World Clock app, the settings pane shows the options for the Widgets once the "Edit" link has been tapped. Tapping it again hides them. You can't see it in a static screenshot, but the "Edit Settings" and "Delete" buttons fade in and out, similar to how the controls do in the World Clock app.

The last 3 screen shots are all states within the main view of the application. The screens sit next to each other, and are all delivered by one action within the WidgetsController ("index"). The user simply needs to tap the dots at the bottom to switch screens.



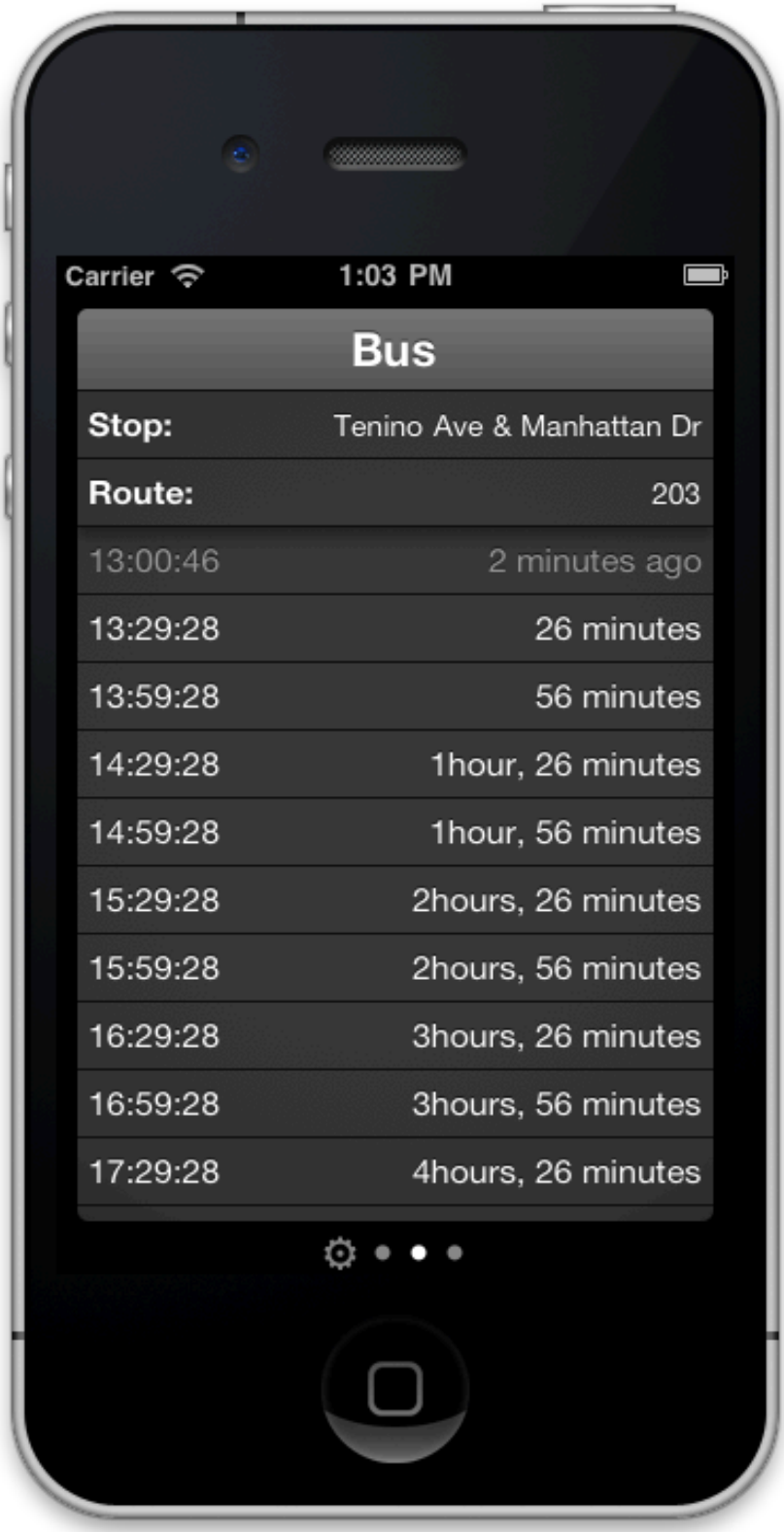
This screen is powered by WidgetsController.new and is reached when the users taps the "+" on the settings widget. Tapping any of the "Add" buttons submits a form to WidgetsController.create, which instantiates a new Widget object and adds it to the current user's collection of Widgets, then redirects the user to WidgetsController.edit for the new Widget so they can put in their settings.



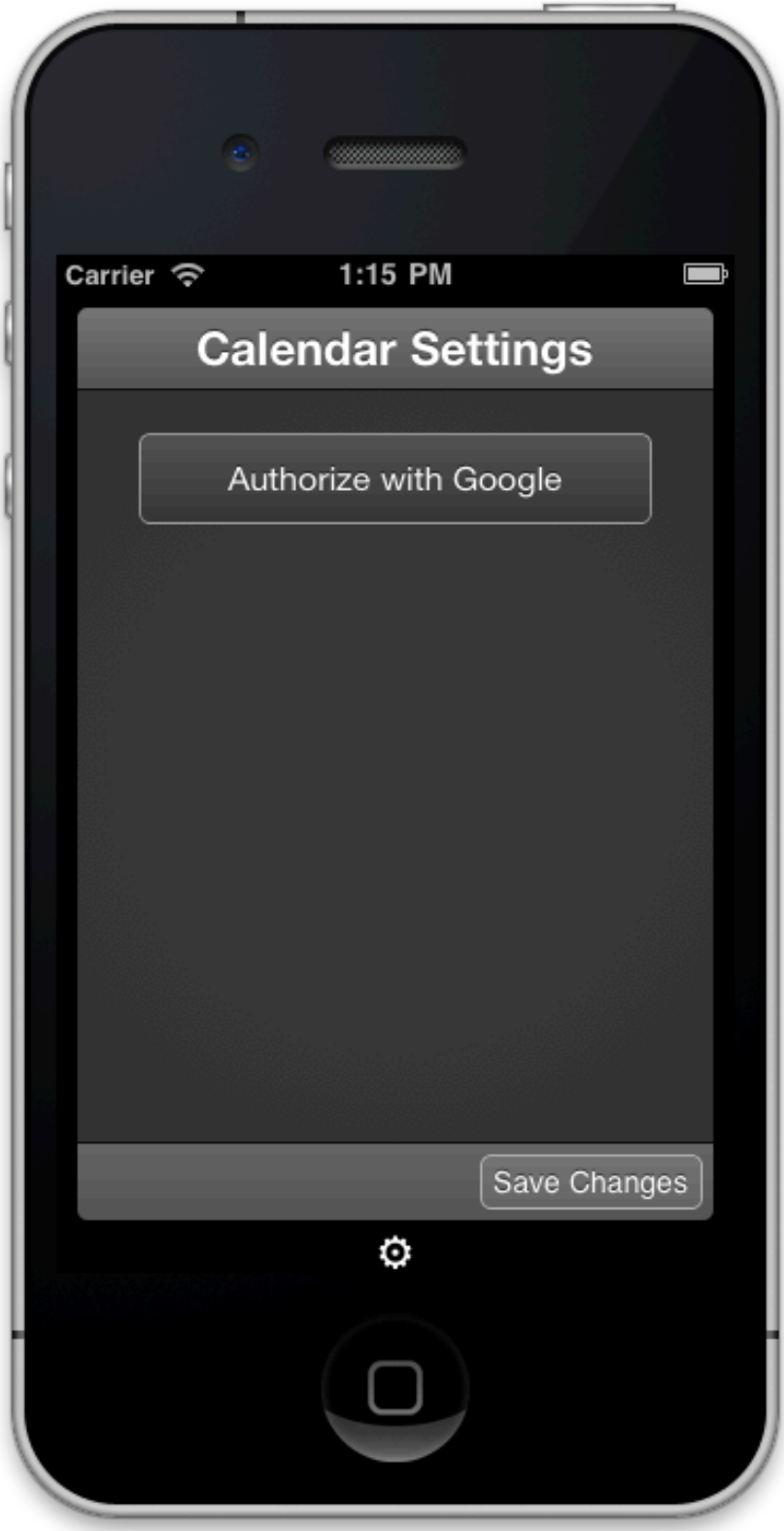
This is the settings page for the Weather widget. You enter the zip code that you're interested in finding out weather information for. This view is rendered through the WidgetsController "edit" action. It polymorphic-ly determines which settings options to show based on what subclass of Widget is being shown.

This is our Weather Widget running live. Based on the zip code the user has set (seen in the settings screen), the Widget calls its DataRunner (in this case hooked up to Wunderground) to get a forecast for today and the next 3 days. Because we could theoretically switch out data runners to provide access to a different API (like Yahoo! for example), the data is stored without reference to its source: simply as a set of days, each with "high", "low", "conditions", "day of the week", and "conditions image". The code is also loosely coupled enough that it would be trivial to design another widget that works off the same data, presenting the weather differently.

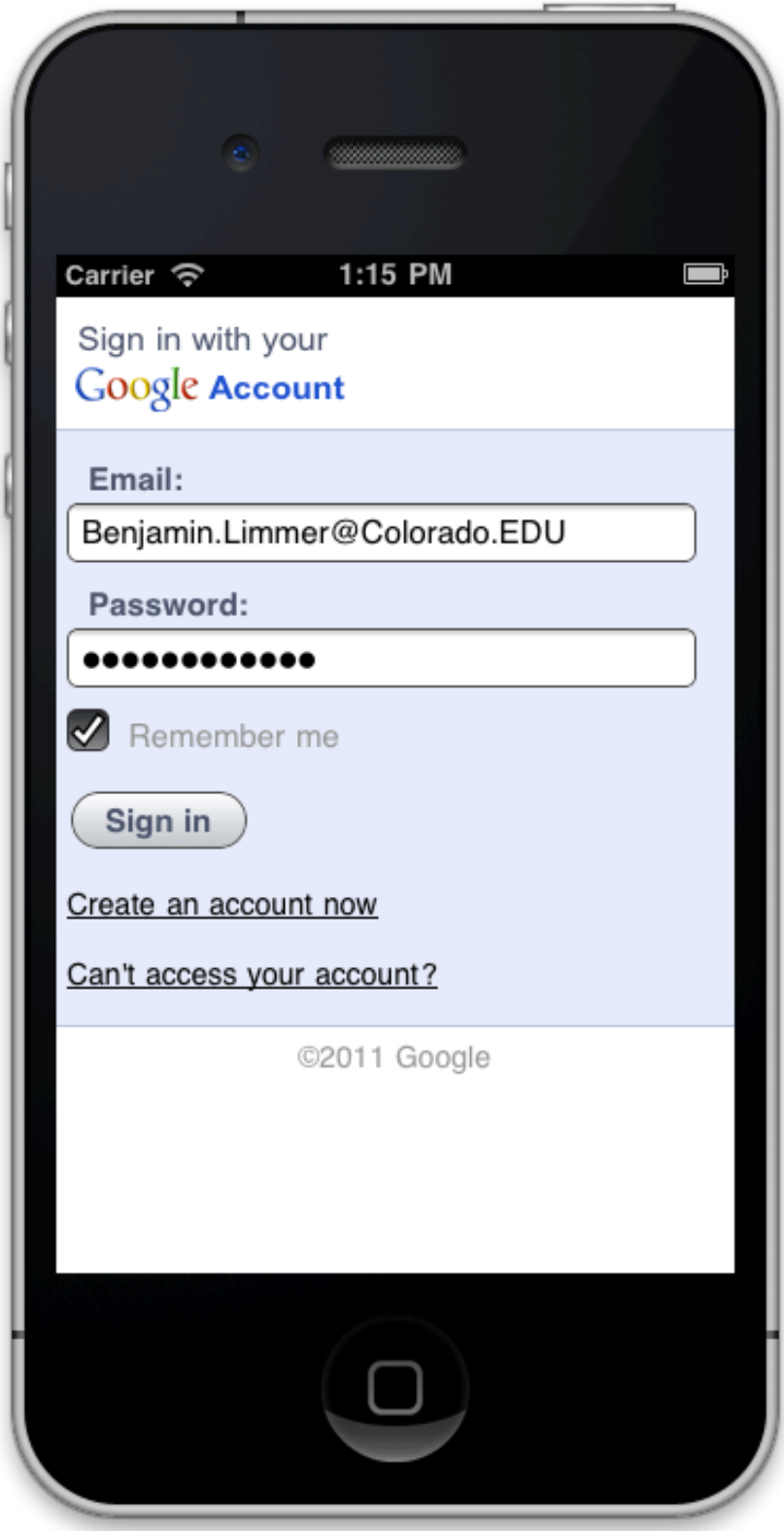
This is the view to edit settings for the bus widget. Currently, this is not the most user-friendly way to enter data since you have to enter a stop ID from RTD's database. In the future, this could be expanded to use GPS or another more user-friendly data input method. This view is rendered through the WidgetsController "edit" action. It polymorphic-ly determines which settings options to show based on what subclass of Widget is being shown.



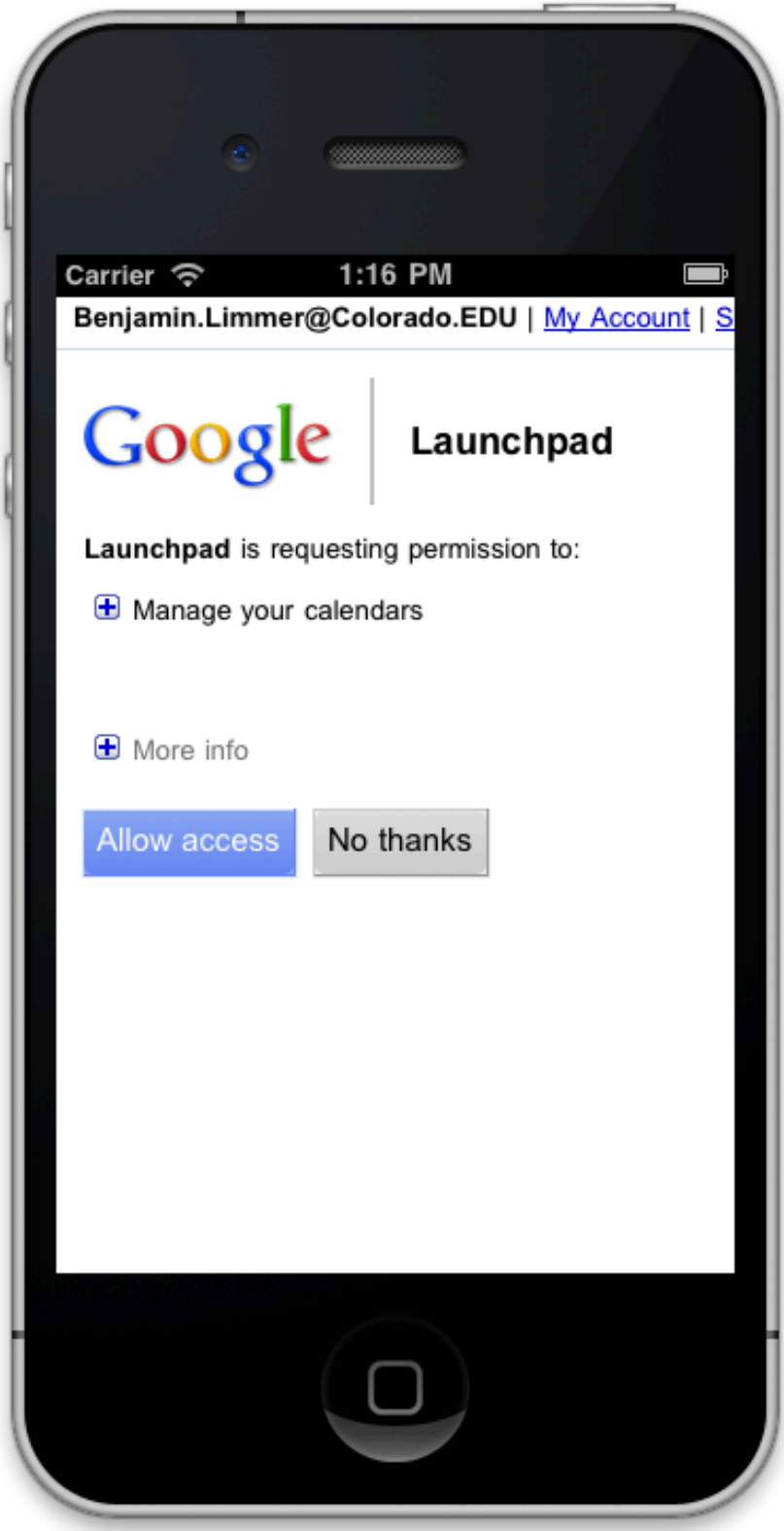
This is our Bus Widget running live. Based on the stop ID and route the user has set (seen in the settings screen), the Widget calls its DataRunner (in this case hooked up to the RTD data files) to display the upcoming times and past buses for the last thirty minutes. Data is read from data files in Google's Open Transit format, which the majority of public transportation systems nationwide use to provide data to Google Maps.



This is the (very) simple interface for the Calendar settings. It simply utilizes the Google Calendar API and OAuth2 to grab information from the user's Google Calendar. We opted to use OAuth2 because our application no longer needs to handle the user's sensitive Google credentials. The next screenshot shows the Google pages that the user is directed to in order to authorize Launchpad to access their calendar information.

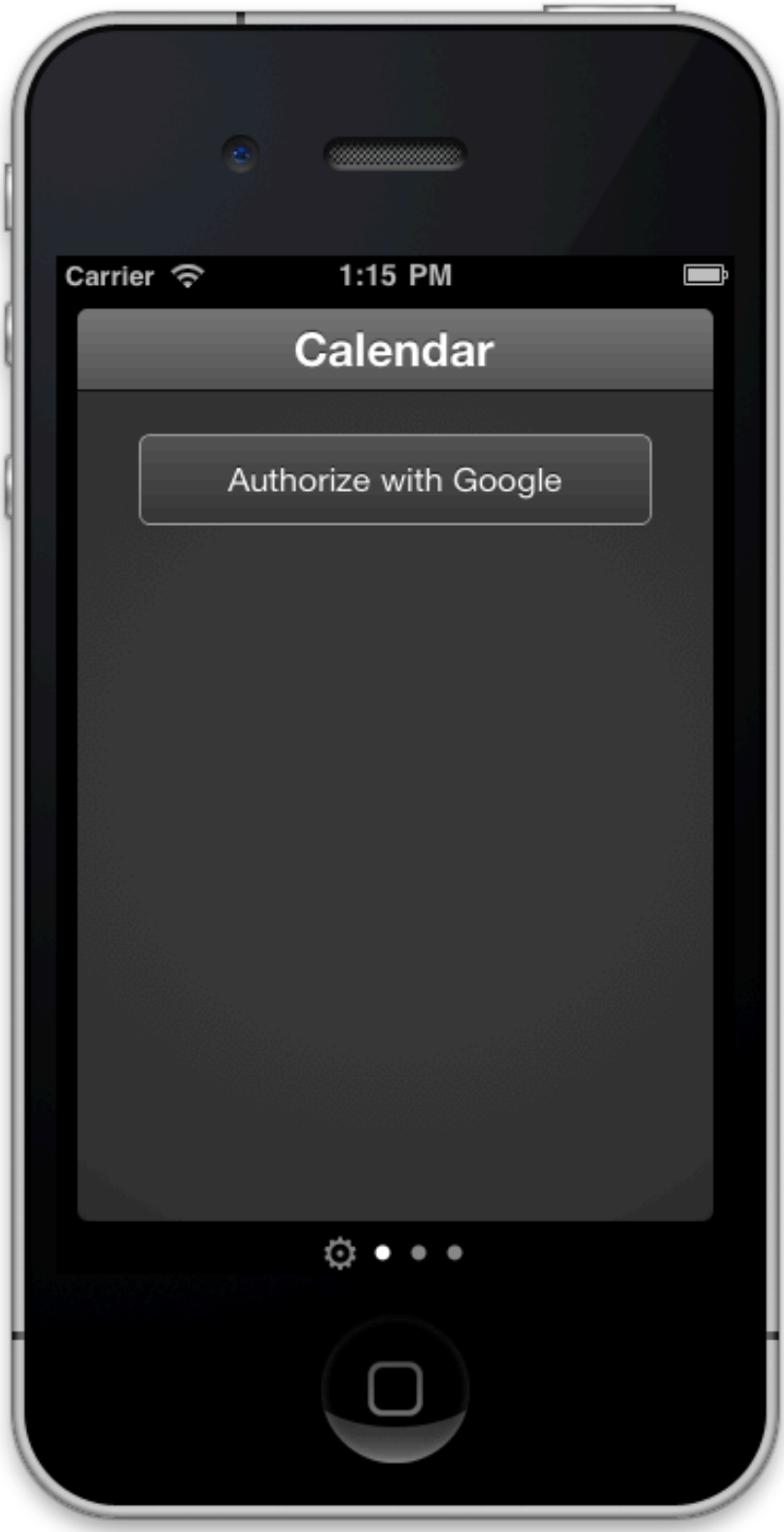


The user enters in their Google username and password into the Google page displayed. This interaction happens only with Google, and Launchpad never receives any information regarding the user's Google username or password.



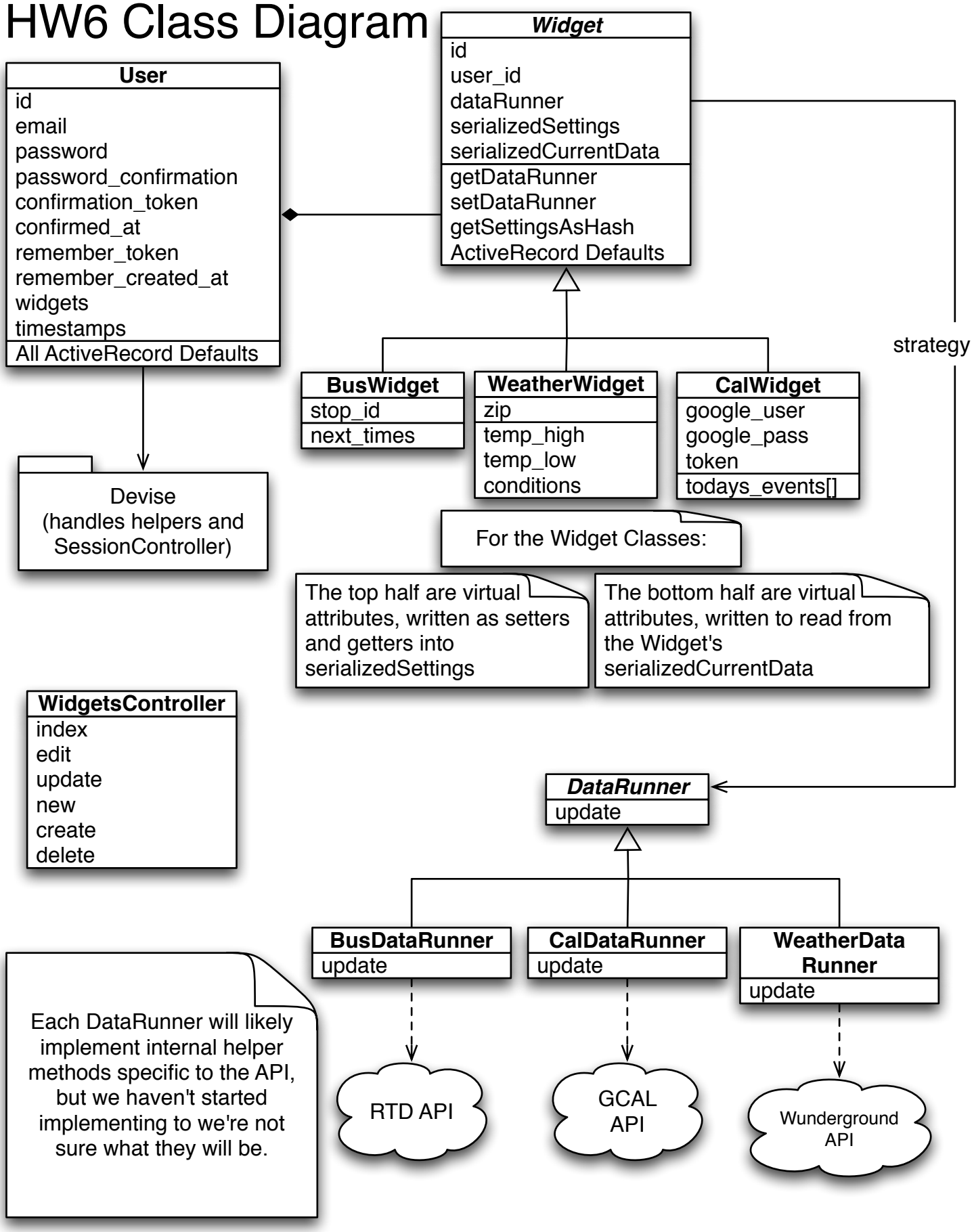
The user now sees the "scope" that Launchpad is requesting and is prompted to allow or deny access to the application. When the user presses "allow access", Google send us an authorization code with which an access token is provided. This token is then used to grab the user's Google calendar information.

This is our Calendar widget running live. Data is pulled from the Google Calendar. Currently, the view only updates manually, but a cron job could easily be set up to run every hour, every five minutes, etc. The reason the production code does not do this is that our host, Heroku, charges for this functionality. Additionally, the reason these do not update constantly is that Google charges if we go over their free API usage.

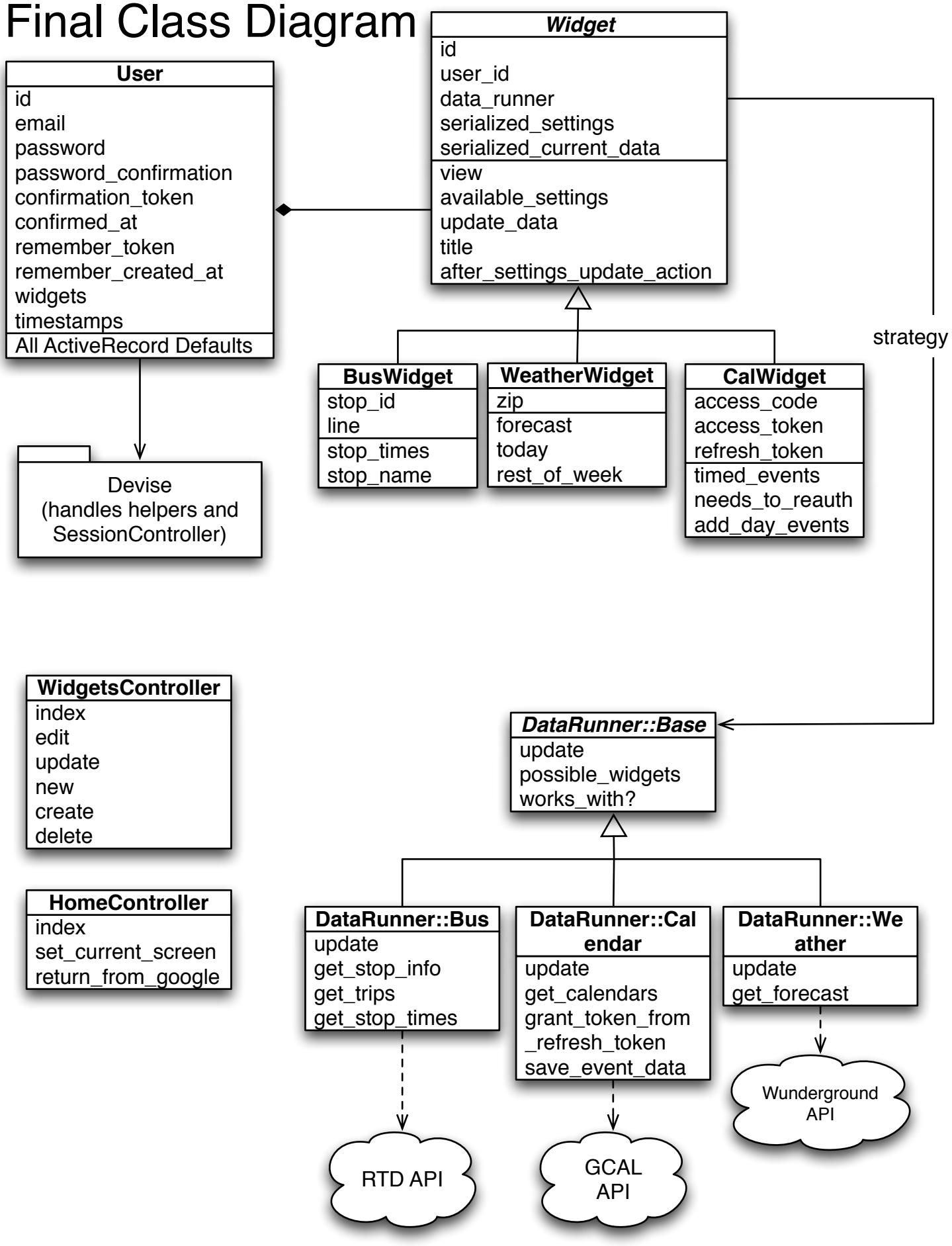


Due to the flow of OAuth2, Launchpad will occasionally be deauthorized from Google's API. In the event that this happens, the user will be prompted on the Calendar widget page to reauthorize with Google. The access flow is the same as shown in the settings above.

HW6 Class Diagram



Final Class Diagram



What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system in five weeks?

The most apparent thing to us was how much it helped to spend a lot of time thinking about the problem before writing any code. For both of us, previous programming projects often involved a bit of pre-planning, but mostly design decisions were made while already programming. Doing this often lead to numerous structural changes over the course of the project. In contrast, with this project, our design did not need to change after its initial creation. The time we spent working on our design, and thinking through how the system should work, really paid off while iterating.

We also learned how valuable Design Patterns are. Initially, we were very unsure of how to implement our data runners in a clean way. While trying to think through it, we realized it was a perfect example of delegating behavior, and from there easily found the Strategy pattern to be a good fit. While we're sure our system could be even more loosely coupled, working within some patterns we learned in class along with others we found elsewhere helped keep our system agile. In the final iteration, we found ourselves conveniently working on different areas of code that interfaced with each other without issue of stepping on the other person's work.