Pack Manager

CS130: Software Development
Computer Science Department
University of California, Los Angeles
Spring 2015
Git repository: `https://github.com/asanciangco/PackManager`

Alex Sanciangco
Nick Westman
Steven Holtzen
Mitchell Binning
Kacey Ryan
Heather Blockhus

Contents

1 Introduction and Motivation

Pack Manager is a tool for helping you pack. It gives you a wardrobe list based on the weather of where you're going. It takes your (1) location, (2) time of travel, (3) personal dressing preferences, and gives you a set of clothes for you to wear and necessities that you can't forget. The goal is to make packing for an outing simple and fast. The application will have a very clean user interface; since its function is simple, its use must be equally simple. The application must be extensible, which means that adding new features must be a well-defined process that is easy.

1.1 User Benefits

A user may benefit from using our application if they are frequent travelers. It is very difficult to track what to bring when traveling, especially if travel arrangements frequently change or users travel to many locations in sequence. Our application can update them on changing weather patterns for arbitrarily complex trips spanning many days and destinations.

Another example is someone traveling to an area with unfamiliar weather patterns. This makes it difficult to predict what would be necessary to bring. This user would simply enter their information into our application, which would automatically keep them up to date regardless of whether or not they know about monsoon season in Indonesia.

2 Feature Overview

When the user first starts the application, he or she will be presented with a login screen where they can choose to log in with Facebook. Once logged in, the user will enter their preferences, at which point the application is ready to make packing suggestions. Once this initial login is complete, the preferences will be assigned to their account, making future trip planning seamless. A packing list is suggested after the user enters their destination, which can include multiple stops, date of departure, duration, and takes into account preferences and trip parameters. From the "My Trips" tab, the user can view and modify this packing list. The application will notify the user in case of weather changes; this will require updates to the packing list.

2.1 Initial Requirements

The following are the requirements that we initially set out to fulfill.

2.1.1 General Requirements (G)

1. Able to pull relevant weather information from Open Weather Map for arbitrary dates. (Related: G2, G4, G6, API1, API3, API4, UX7, UX8)
   (a) If no projected weather data, pull historical.
2. Provide a framework for accepting and using user preferences. (Related: G5, G8)
   (a) User preferences must be consistent between sessions.
   (b) Must be able to export preferences.
   (c) Plan for eventual transition to decentralized storage (move to cloud).
3. Create packing lists for vacations of arbitrary durations. (Related: G7)

4. Provide a framework for trip preferences. (Related: G2, UX2)
    (a) Availability of laundry machines Level of cleanliness ("I don't care about wearing underwear 13 times")
    (b) Specific activities Purpose (business, leisure, etc.)
    (c) These options affect the packing lists but are optional with customizable defaults.
5. Sends user a push notification if weather changes in destination will affect packing list, or a critical weather alert is issued. (Related: G1, API4)
6. Ability to modify already created packing list. (Related: G4, G8)
    (a) Add items, remove items, reevaluate (pull weather again)
7. Ability to create an itemized list of clothing and other essentials for travel. (Related: G2, G7, UX1, UX4, UX6, API2)
    (a) List can be customized after suggestion by user.
8. Allow the user to login through Facebook. (Related: API3, API6)

2.1.2 User Experience (UX)

User Experience:

1. 3 user input (minimum) to get to a packing list: start date, duration, location. (Related: G5, G8)
    (a) Additional options available if necessary
    (b) Duration must be nonnegative
    (c) Destination: zip code or city, state/country
2. Prompt for user preferences on first launch. (Related: G3, G5)
3. Obtain user feedback after a trip to adjust user temperature preferences. (Related: API4)
4. User can store packing lists upon save (Related: G8, UX5, API2)
    (a) Possibly autosave
5. Export lists to other apps (reminders, calendar) (Related: UX4)
6. Packing lists formatted as a table. Each row composed of item and quantity (e.g. T-Shirt x3, Socks x4, Suntan Lotion x1, Umbrella x1) (Related: G8)
7. Error Reporting: If network unavailable, prompt user for solution (Related: API1, API4, G1)
8. Error Reporting: No relevant weather information for provided destination (Related: API1, API4)

2.1.3 API Interactions and Extensibility (API)

1. Application has a single point of failure for interacting with external APIs (Related: G1, API3, API6)
2. Packing and destination information can be exported in JSON format (Related: G8, UX4)
3. Build a wrapper around Open Weather API and Facebook for easy API usage (Related: G1, G9, API1)
4. Build a push notification framework for communicating with users in a device-agnostic way (Related: G6)
5. The system must have the capability to be localized (allow for arbitrary languages and character sets to be used)
6. Facebook API is used for login and identity management. (Related: G9, API1)

2.1.4 Non-Functional Requirements (NF)

1. Test coverage and automated building
2. Free to download
3. Secure
   (a) user cannot see your packing lists
   (b) Outside users cannot modify your packing lists through any API
4. iOS Application targeted for iOS 8 and above

2.2 Requirements Revisions

We added several new features that were not initially listed in our design requirements. We added the ability to autocomplete your location using the Google API. We also added a Google Maps visualization, so that users can see their destinations. These features were deemed necessary because it is necessary for a good user experience: without seeing on a map what you typed in, it might be the wrong location. It is also very hard to type in an exact address, so the autocomplete is necessary as well.

We decided to omit some of the requirements that were less important. We decided not to export user data to other applications such as calendar and reminders because as we were designing our application we realized that i) exporting to the calendars app wouldn't be particularly useful, and ii) we would rather have the to-do list features of the reminders app remain within our app. We also realized that we didn't want to prompt users for the preferences on the initial launch of the application since this made for an obtrusive and confusing UX; we would rather get the users right into the functionality immediately, then allow them to change settings if they so desired. We decided against gathering user feedback after their trips also for UX reasons: we realized that users likely wouldn't open our application immediately after a trip since our app is intended to be used before a trip. Our application does not currently export packing information to JSON because at this time it is not necessary. However, the data structure that stores the packing list can easily exported to JSON if that functionality ever become relevant. Lastly, we did not end up implementing push notifications because that would require usage of Apple Push Notification Service (APNS), which would require us to develop a server to send the notifications.

3 Design

The application is structured using the model-view-controller architecture dictated by the iOS development framework. In an iOS application, the overall flow is defined in storyboards. These files define transitions between panes, hold content. The main storyboards for our application is in the `Main_iPhone.storypad` file. The storyboards are the views in this framework.

The controllers can be found in the `Controllers` subdirectory. These files dictate the flow of control within the application; they are the glue between the storyboards and the models, which define how data is stored and manipulated. By carefully separating the concerns between storing and loading data from the user interface logic, we get a clean modular structure that allowed Heather and Alex to both work simultaneously on different aspects of the application.

3.1 External API Usage

3.1.1 Weather API

The Weather API is an important and very tricky component of our application. See `PackManager/WeatherAPI.m` for our implementation. We currently use Open Weather Map[1] in order to gather weather data, which gives us a 16 day weather forecast for a particular geographic coordinate. We also use the NOAA weather API[2] for date ranges that are outside of a 16-day weather forecast. This data source includes historical data and can allow us to make coarse-grained projections over a much greater timespan and in a larger area. However, we can not get historical weather API for out of the country, since the NOAA is a U.S. government organization. Both data sources have completely different output formats (which is why our parsing file is over 500 lines long!).

3.1.2 Google Places API

We used the Google Places API[3] for auto-completion as the user is entering their desired destinations. Handling this API was tricky, since it has a limited number of requests per day. See `PackManager/GooglePlacesAPI.m` for implementation.

3.2 Class Diagrams

See Fig.1, Fig.2, Fig.4 for an overview of our class design decisions.

3.3 Clothes Selection Algorithm

In order to generate a list of items one should bring on their trip we needed to come up with an algorithm. The packing algorithm would have to account for the trip's duration, each day's weather, and a number of user preferences (i.e. re-wearing jeans, access to laundry, their hot cold preferences, etc). Our algorithm first initializes a packing list, checks each day's weather and appropriately increments clothing, and finally produces estimates based on user preferences. The initialization creates a packing list that contains the bare essentials such as toiletries. Additionally, the initialization will check if the user has marked if they will be swimming or need formal attire. If they checked swimming then it will include swimwear as a function of days. Next, for each day in the trip the algorithm calls several incrementing functions, one for each type of clothing (tops, bottoms, etc). Each of these incrementing functions have a simple interface that hides its

[1] See `http://openweathermap.org/`

[2] See `http://www.weather.gov/`

[3] See `https://developers.google.com/places/`

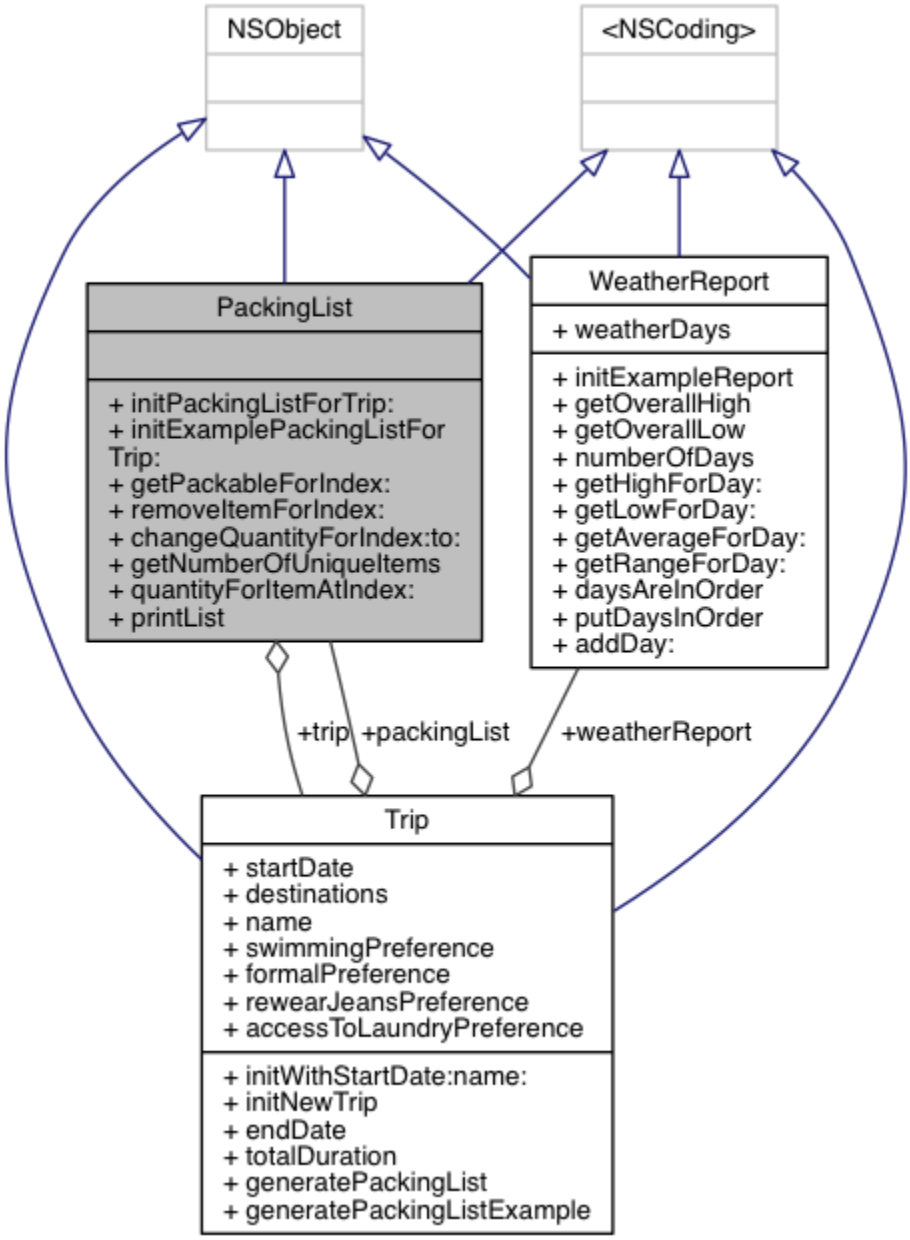


Figure 1: The `Trip` class is the core structure for holding details related to an individual trip. Currently trip attributes are stored as booleans within this structure; we have plans to make these easier to change in the future.

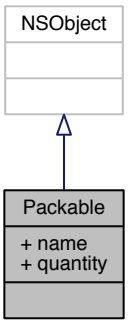


Figure 2: The packable interface defines the basic methods that are necessary to make an individual object packable. All articles of clothing will inherit this interface; it will likely need to be expanded as more required functionality is discovered.

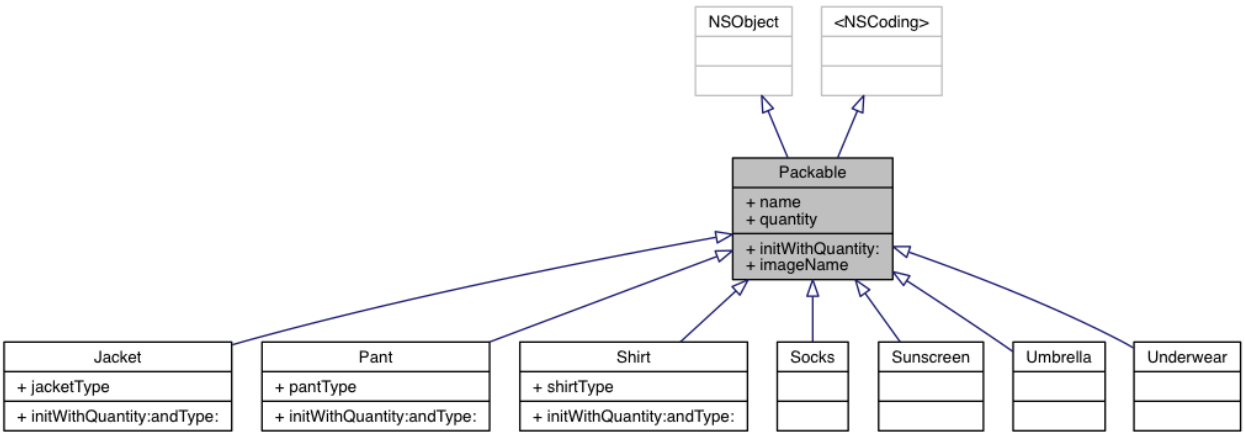


Figure 3: Packing lists are made up of packables and this diagram shows the packable class and the multiple types of packable items.

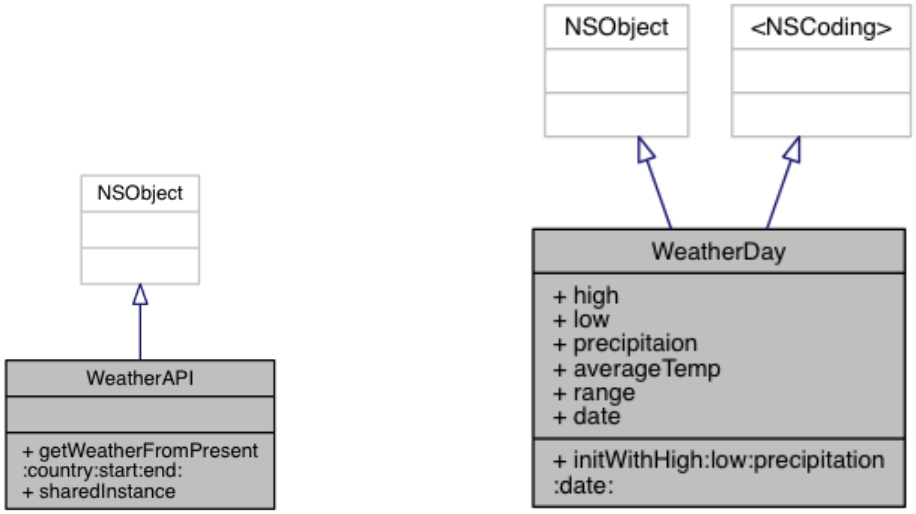


Figure 4: As shown in the first figure, the trip takes into account the weather report, which is generated by the Weather API shown above. Additionally the weather report consists of weather days, shown above.

implementation details from its caller, exercising the information hiding principle. Each function checks the day's average temperature against the user's temperature preference and perform different clothing increments based on what the user thinks of that day weather (hot, warm, normal, cool, cold). Finally, now that the algorithm has produced a maximal list, it checks the user's preferences for things like access to laundry and rewearable jeans and reduces the list's numbers. For example, if laundry is checked it will cap the max number for each type of clothing article. Similarly, re-wearing jeans would cap just the bottoms category. x

3.4 API Description

Our API description can be found inside of our documentation, in `doc/html/index.html`. We took care to document most of our classes and functions in the Doxygen style, allowing for automatically generated formatted descriptions. An example of our documentation can be found in Fig.5. Overall, all API accesses flow through the `WeatherAPI` and `GoogleAPI` classes; these classes act as mediators, filtering information and handling de-bouncing of frequent requests.

4 Quality Assurance

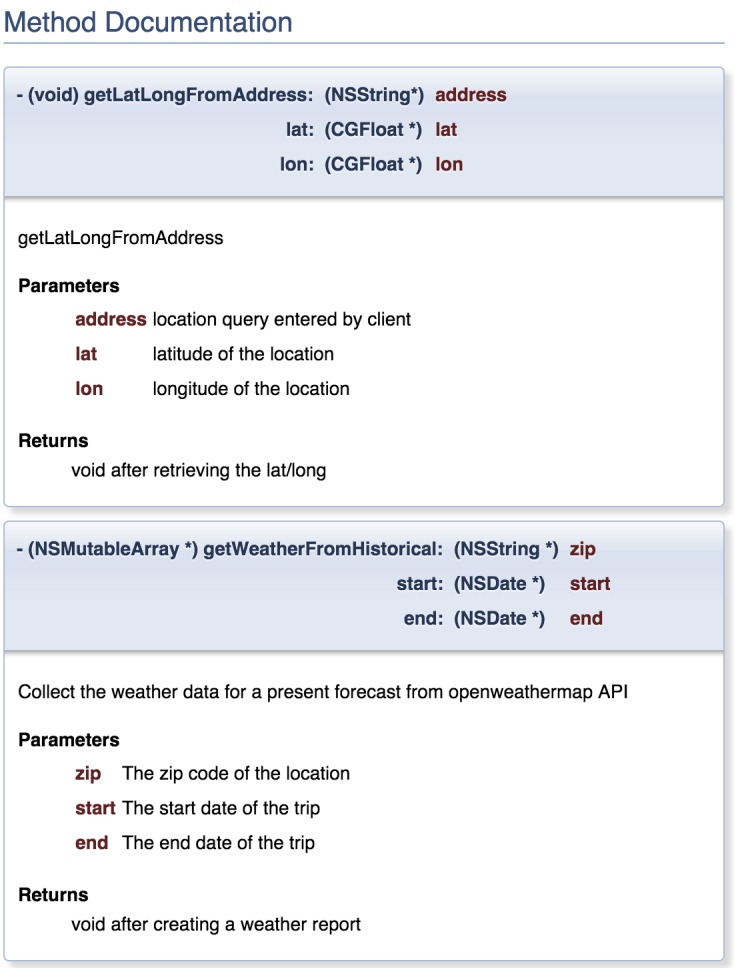
4.1 Testing

We set up a test harness using Travis CI[4]. Travis CI is a continuous integration suite that automatically runs our tests whenever a pull request is performed on Github. Before a change is merged all the tests are run in order to guarantee that all code that gets checked in passes all the tests. See Fig.6 for an example Travis CI output.

Travis CI configured in the `.travis.yml` file, which specifies the language framework as well as the test harness. Our tests can be found in the `PackManagerTests` directory. We used Nocilla[5] in

[4]See `https://travis-ci.org/`

[5]See `https://github.com/luisobo/Nocilla`



Method Documentation

- (void) getLatLongFromAddress: (NSString*) address
lat: (CGFloat *) lat
lon: (CGFloat *) lon

getLatLongFromAddress

Parameters

address location query entered by client
lat latitude of the location
lon longitude of the location

Returns

void after retrieving the lat/long

- (NSMutableArray *) getWeatherFromHistorical: (NSString *) zip
start: (NSDate *) start
end: (NSDate *) end

Collect the weather data for a present forecast from openweathermap API

Parameters

zip The zip code of the location
start The start date of the trip
end The end date of the trip

Returns

void after creating a weather report

Figure 5: Example of the documentation generated by Doxygen for the `WeatherAPI` class.

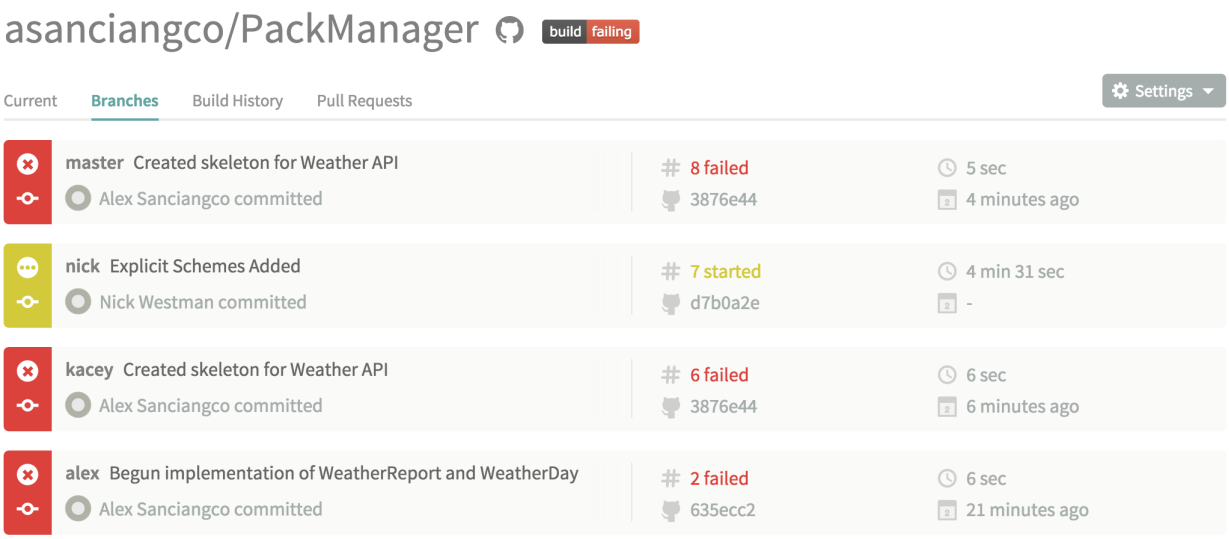


Figure 6: Output of Travis CI. It shows the status of all branches as well as whether or not they are currently passing the tests that are specified in `.travis.yml`. This will prevent ever merging a branch that does not pass tests, which will prevent regressions.

order to create mock HTTP requests to simulate the weather API behavior. This lets us test the Weather API without making HTTP calls. We then use these same mock API calls throughout the rest of our tests. We test much of our application in isolation of the Internet, which allows our tests to be reproducible with expected outcomes, letting us use assertions.

We used testing in order to validate our application; the goal is not to achieve ultra high reliability, since our application is likely to rapidly change. In line with this goal, we have isolated unit tests for the core functionality, including tests for the weather API, Google APIs, the packing algorithm, and the integration between these components. These tests can be found in `PackManagerTests`.

We make extensive use of swizzling, which is a built-in utility for iOS that lets us swap out implementations of individual methods for stub methods. For example, when the `getDate` method is called, we return a specific date for use in assertions (for example, Jan. 1 1970). This lets us make assertions regardless of what day of the week it is.

5 User Interface

See Fig.7 for some screen shots of the user interface.

6 Contributions

1. Heather worked on the user interface. She worked on developing flow of the application, working closely with Alex. Heather is experienced with designing user interfaces for iOS.

2. Alex is the lead programmer. He worked on integrating Heather's user interface with Kacey's weather API, as well as being a source of knowledge on general iOS development and Objective-C, which many group members were not familiar with.

3. Steven worked on writing the report and the documentation. He added Doxygen and used LaTeXto write the paper.

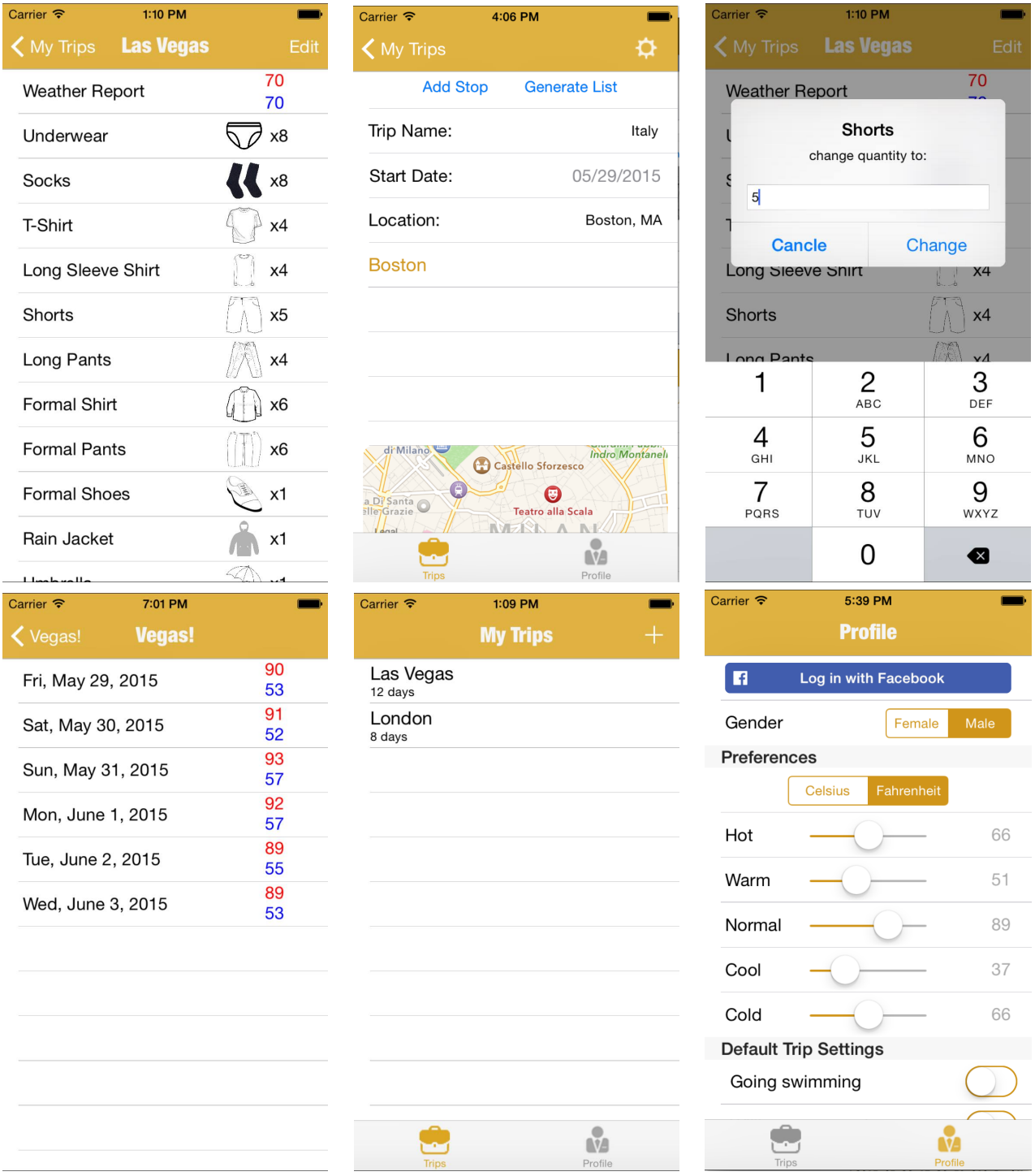


Figure 7: User interface snapshots. Top left, a generated packing list; top middle, the trip creation screen with location auto-complete and Google maps visualization; top right, an option to enter a specific number of shorts; bottom left, example weather forecast for a trip; bottom middle, selecting between multiple pre-made trips; bottom right, the user profile, with Facebook login.

4. Kacey worked on the weather API, doing research and wrapping it to make it accessible and easy to use.

5. Mitchell worked on the presentation and report as well as general programming tasks.

6. Nick was in charge of build management and testing, integrating Travis-CI and managing the test infrastructure.

7 Conclusion

We completed almost all of the requirements that we listed for our application on time, in addition to some extra requirements that we felt were necessary as we were developing. In total the project is over 3000 lines of Objective-C and over 1300 lines of comments, with major commits and contributions from every group member, even the ones that were not initially familiar with Objective-C.