Project Proposal

Team: Ted Lim, Scott Buttinger, Kevin Bishop, Ryan Hails, Chase Brandon

Description of the Product:

The proposed product is an airline flight management and scheduling mobile application. By forwarding a received airline flight confirmation email from an airline company like United to the proposed application's user-specific specialized address, the application will then create and compile a cohesive list of a person's flight details. The proposed application will further be able to send the user's flight details, based on departure and arrival time, to a scheduling tool of a user's choice and integrate itself into the user's schedule. The application can not only be used for personal use, but it could also be used for scheduling pickup times for friends and acquaintances. They would be able to send their airline flight confirmation email to a user with the application, and automatically integrate the information into a user's schedule. The application will further include a notification for the user for when either the time of departure or arrival of an airline's flight has changed.

Need for the Product:

Being able to organize the constant chaos in daily life has become a necessity in this technology driven era. One of the areas that we want to bring more stability to is around travel. There is always concern around the unpredictability of traffic going to an airport and the chaos around going through airport security. The application hopes to help people prepare and schedule their time around airport related concerns by bringing impending tasks to the user's attention.

Another need for the application is to facilitate the act of locating buried flight information and details in a cluttered email inbox. It is certainly possible to find a relevant email given that people know what they are looking for. However, this application hopes to organize and schedule emails, specifically around airline travel such that users only have to be concerned about flight details that are presently relevant or will be relevant rather than past dates that always show up in the recesses of the inbox. By having the upcoming relevant flight details compiled in a more organized fashion, in either a list, calendar, or other scheduling form, people can be updated from their mobile device from wherever they may be.

Potential Audience:

The potential audience for the product is a variety of travelers and online shoppers. Whether you're someone who shops online everyday, or someone who travels every once in awhile we feel our product can be useful to you. The size of our audience will be dependent on the number of users that we are able to attract, but with over 200 million online shoppers in the U.S.(source) and 900 million U.S. flight scheduled passengers per day (source), the sky's the limit! We see our product being most useful to millennials who frequently order packages from various sites and the various people who have hectic travel schedules. Given the timeline of the class, we see our product serving just American ios users.

In order to have the largest number of users we possibly can we want to make our product as simple as possible. Users will need to be able to download and login to our application, and be able to forward the correct emails to the application. Other than that there are no other technical barriers that might prevent a user from using our product.

Discussion of Competing Products:

The main competitor is Apple's Passbook application. This app allows users to aggregate e-tickets for events and flights. Despite attempting to cater to the wide variety of e-tickets that are available today, Passbook suffers from the fact that many airlines use ticketing formats that are incompatible with the app. Our project will overcome this flaw by using a user's flight confirmation email to generate flight "event items" that the user can easily sort through within the application. By simply parsing out important information from the user's email, we can create "event items" that are compatible with all airline ticketing schemes.

Another app in a similar space is Slice, a mobile application that tracks and aggregates online shopping orders. The catch however, is that Slice requires full access to a user's email account. It needs total access so that it can scan the user's emails for specific keywords and pull out information related to shopping purchases. Our application is designed such that we do not require full access to a user's email account. By simply assigning each user a unique email address to which they can forward shopping and flight specific emails, we give users peace of mind by keeping the remainder of their inbox private.

High-level Technical Design:

Account Creation and Login:

- Name, personal email address, unique email address generation,

To create accounts and login into accounts we will use Swift's GenericKeychain library. Because we are using swift, it seemed like a module that could be integrated smoothly. When users create an account we will generate a unique email address affiliated with their account and they will be asked to fill in various information about themselves.

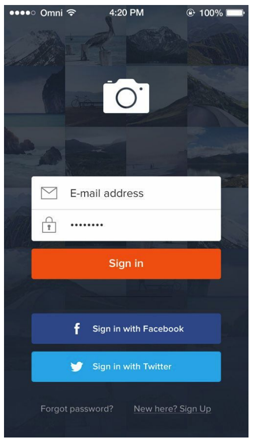
Store User Information:

- User ID, name, email address, unique email address, list of forwarded emails, list of order/flight "event items"

We will store this information using the CloudKit Framework. This will work nicely because we are only developing an application for iOS and therefore can simply tie a user's personal information to their iCloud account. By storing user data using iCloud, we can leverage Apple's existing data protection and privacy measures.

User Interface:

- Login Page

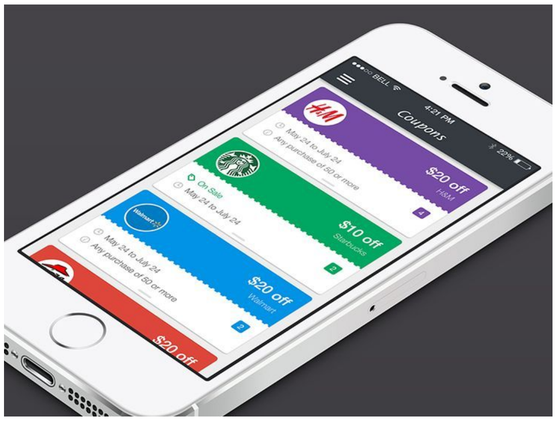


The login page will be very simple, consisting of a few buttons, labels and text fields. The photo to the right of this section displays a sample login page that is similar to the one we will use for our application.

- Account creation

The account creation page will also use buttons, labels and text fields.
Both the account and creation pages will use action listeners that will validate each form. Example of this would include ensuring that all fields are filled in and we would ensure that none of the necessary text fields were empty and using the NSRegularExpression class to validate email addresses. Those pages will also take advantage of the Facebook API to allow users to link their facebook accounts to use our application.

- Menu for viewing past/upcoming "event items" (flights or orders)



The menu for viewing past/upcoming events would utilize the iOS UICollectionView or the UITableView class. We envision a simple but elegant UI that would allow the user to scroll chronologically through their flight and order event items. The photo to the right of this section shows an application that utilizes a similar UI style.

- Menu for manually entering flight or event information

This screen would simply contain a few buttons, labels and text fields that allow the user to enter all of the information that is required for flight or shopping order event item creation.

- Settings menu

For our settings menu we will use the UITableView class in swift, where each UITableViewCell will correspond to a specific setting. We will likely use other classes to handle specific settings including UISlider and UISwitch.

Emails:

Our application will be responsible for both receiving and parsing emails. To do so we will use the following schema.

<u>Flight</u>

- Airline (string)
- Flight number (int)
- Date (DateTime)
- Time (DateTime)
- Passenger name (String)
- Destination (City)
- Departure (City)
- Confirmation number (String)

<u>City</u>

- Name (string)
- Country (String)
- Airport Code (string)

<u>Orders</u>

- Seller name (String)
- Tracking number (String)
- Date of purchase (DateTime)
- Product name (String)
- Shipping address (String)
- Expected delivery date (DateTime)
- Delivery status (int or struct)

We will use NodeJS and Express to handle incoming connections to receive emails. With the emails we receive will use a parser (either one we create or one like npm's simple-text-parser) to create JSON objects with the format described above. Once we've generated a response we will forwarding it to CloudKit.

<u>Resource Requirements:</u>

- iOS application for user, web server for parsing emails, cloud storage for user information

<u>Potential Approaches:</u>

Our proposed application mainly transfers data through the user required action of forwarding their own confirmation airline emails to the application's user-specific email. This could have been done through a more intrusive method such as aggressively collecting all of an application user's emails and filtering by the flight confirmation email. By sacrificing a little work on the user end, we want to ensure that we are not intruding on a user's personal data. Additionally, having users forward emails is a simple problem to tackle. As we learned in lecture, when tackling a problem, good engineering practice is to code with the simplest logic, and we believe that having users forward the emails is much simpler than collecting their email and filtering.

We also decided on iOS as our mobile operating system of choice due to its larger user base and accessibility. The other option we could have used would be Android; however, a larger number of the team knows iOS.

NodeJS+Express offers quick server hosting that abstracts a lot of C/python based server hosting details. Group members that were more comfortable with backend have worked with Javascript and Node before. It easily communicates with several databases and can easily serve the static information to requesting users. It's web-app communication is ideal for forwarding emails.

Assessment of Risks:

In terms of non-technical risks to the completion of the product, these issues mostly concern intra-group conflict or confusion. The most important pitfall would be lack of communication. It is necessary to be explicit on expected deadlines for features of the product to be successful. These deadlines also must be realistic. The determination for what is realistic is gauged by team communication. If individuals remain quiet about an issue they are facing, that will prevent them from meeting required deadlines and will delay the team as a whole. Currently, in order to address this risk, we have not only established multiple communications networks through group chat and slack channels, but we also have three scheduled meeting times throughout the week where we are able to discuss progress and future plans.

On the technical side, a large part of the risk is on compatibility and version control. For compatibility it is important that everyone is working with the same software versions. Many times different software versions can be the cause of infuriating bugs. These bugs can then create further complexity if pushed to the master branch accidentally, causing confusion among the team. The current method we will be implementing is agreeing upon a standardized procedure for how each member will pull and push code. Additionally, we will have one person, our tech lead or another individual who wishes for the responsibility, in charge of deciding when a current version of the project will be pushed to master.

Additionally, other technological issues that could occur could be discovering that a feature requires more knowledge than the team knows. In that case, we will likely address that issue by contacting the TAs for the class and asking them for advice.

Next Steps:

The two major components of the application are the flight and the package managers. We are planning to first implement the flight management system with a few major airlines (Southwest, JetBlue, American) and an iOS application that allows users to create and login to an account and look at their flights. Ideally we will have a version of this done by the mid-quarter demo. Once we've completed that we will integrate the package manager, incorporate more airlines, and further develop our user interface.