Arman Berek, Mary Fan, Jeffrey Lin
CSCI 499
Foodies Final Demo Script

Introduction

Arman: I'm Arman.
Jeffrey: I'm Jeffrey.
Mary: I'm Mary.
Arman: And we'll be presenting our final project, Foodies.

Product Definition

Arman: So what is Foodies? Our product allows users to save time, money, and compromise between a group of friends. We want to solve the problem of going out with friends and having a hard time deciding where to go to grab a meal. We solved this problem by creating an app where users can create events and invite their participating friends. The group of friends will then see a list of restaurants based on the group's preferences and each person will vote for the restaurant they would like to visit. The winning restaurant will be the one we visit. We know our product works when we see satisfied users who get correctly recommended restaurants based on preferences.

Demo

Mary: [Opens 3 simulators]
So this is our project so far. We can login with existing accounts or create an account if the user doesn't have one.
[Register page]
To register for an account, the user only needs their name, a unique email, and a password.
[Login Page]
When the user logs in, they will be able to see the different lists of events that are attending and organizing. For each of these events, there is an overview. For example, this event has a budget of 2, happens at this time, and has 4 attendees.
[Profile]
For each user, they will be able to set their preferences on their profile page. As you can see, the list of food categories include American, Pizza, and Chinese. For each of these categories, the user is able to set a bias from -10 to 10.

[Create Event]
The user can also create a new event. Let's name it "Henry's birthday party", give a price range from 1 to 4, and give a time. The location by default is the user's current location but they can change it here. For now only the coordinates are shown but in the future we would want to use some geocoding API to show what street it's near. We can also add guests to the event using their email.
[Add the other logged in simulator accounts.]
If we try to add the email of a user that doesn't exist, we will get this error. We can also remove people that we've added by accident. After creating the event, we will see it in their organizing list.
[Event]
For each event, we see who it's hosted by, the list of people attending, and the list of recommended restaurants based on the preferences set by the attendees, the budget, the opening and closing times, and the location.
[Voting]
Each guest would then be able to vote for a restaurant of their choice. When they have voted, they have to wait for the rest of the members to vote. After everyone has voted, they will then be able to see the winning restaurant and the number of votes for each restaurant. From here, the user will then go to that restaurant and enjoy their meal.

Jeffrey: [Opens Foodies on phone]
Foodies is meant to be a mobile app and here's how it looks on a real device.

Project Overview

Architecture

Mary: Our project is backend heavy. The user facing APIs include creating an account, logging in, setting preferences, creating events, getting event details, and also voting for each event. The server communicates to our database to get the user account information such as the user preferences and events. It also returns recommended restaurants based on the price range, time, location, and user preferences. In the initial proposal, we wanted to recommend restaurants by choosing one category that had the highest bias within the guest list and choosing restaurants from that category. Since we thought that was pretty limiting, we changed it so that every restaurant will have a bias given by the group preference based on its own category and then introduce a random jitter so that each restaurant

will have a chance at being selected. As you can see, the recommended restaurants come from different categories.

Work Distribution

Mary: Jeffrey worked on the backend, Arman worked on styling and research, and I worked on the frontend.

Technologies Used

Jeffrey: Our project uses Dart with the Flutter SDK for the cross-platform user interface. The user interface makes HTTP requests to our backend, hosted on Heroku and written in express.js. Restaurant and user data is stored in an SQLite database.

The initial restaurant data comes from the Department of Health and Mental Hygiene's restaurant inspection dataset, available publicly on NYC OpenData. We obtained open hours and price data from the Google Places and Yelp Fusion APIs by querying them using the Julia language.

Ideas for future development

Mary: With our current project, when a new event is created, the invited guests would have to refresh the page to check for new events they are invited to. In the future, we would want to implement push notifications so that they are informed when they are invited. We also mentioned adding a feature to clone events to make adding the same friends to an event easier. Right now, we add guests to events by their email. In the future, it may be easier to use a QR code.

Challenges faced/Lessons learned

Mary: Reading documentation is one of the biggest lessons we learned from this project because many of the tools we used like Flutter and Google Places API were new to us. A challenge we faced was communication. We worked on different branches and sometimes we would work on the same files. We would often get merge conflicts when trying to pull so that's something we could've improved with better communication.

[Take Questions]

Technical Presentation (Google Places API)

Arman: For the first technical presentation, I will be talking about the Google Place API. The places API is a service that returns information about places. A place can be defined as an establishment, geographic locations, or prominent points of interests. The places API is able to service requests such as returning a list of places based on a user location or a search term, returning more detailed information about a place such as phone numbers, user ratings, and reviews. The API can also provide access to photos related to a place stored in the Google Places database. The API also provides a web-service called Place Autocomplete that returns place predictions in response to an HTTP request. So as a user is typing a name of a place, the request returns related places in real-time. To make a request, you must use a unique API key, an input to identify a target, and an input type such as text query or a phone number. Then a field parameter is added to the request which is a list of places data types to return like geometry, photos, permanently closed, opening hours, price level, and ratings. Additionally a request can be outputted as either JSON or XML. In the example seen on the right, there is a response for a Find Place request for "Museum of Contemporary Art Australia" with its output as a JSON. The response also included the photos, formatted_address, name, rating, opening_hours, and geometry fields. (Next Slide). For our app, we initially were planning to just use the Department of Healths database to get restaurant information such as food categories, but we realized it was missing essential information for our app. As a result we decided to use the Google Places API to get opening hours and price level data for restaurants as shown in the request below.

Technical Presentation (Flutter)

Jeffrey: Flutter is an open-source mobile UI framework backed by Google. It allows developers to create iOS and Android apps using a single codebase. Flutter apps are developed using the Dart programming language, which compiles to native code as opposed to using a JavaScript bridge. The basic building blocks of Flutter apps are widgets, which are similar to components in React. A widget can define structural elements such as buttons, styles, and layout.

We can create a basic app that follows the Material Design guidelines by declaring a class that extends StatelessWidget and providing it with a build method, which tells Flutter how to render the widget.

Let's add a bit of complexity, and make a simple incremental game. We'll need to keep track of the number of times we clicked the button, so we have to store state in a State class. Notice that MyApp is now a Stateful widget, and most of the work happens in the state. We have a few additions to our Scaffold, including the

AppBar, FloatingActionButton, and Center'ed body. As you might expect, clicking the FloatingActionButton will update the counter. Then the setState() function tells Flutter that we need to rebuild the body of our Scaffold, including the text component that shows the actual count to the user.

But what if clicking the button performs an action that might take a while? In this example, we'll force the computation to be artificially delayed for three seconds, but this is not unrealistic: the network might be too slow, or the server might be overloaded. What happens? First, the app will hang, since it is waiting on the main thread. Second, this doesn't even compile! If we go back a slide, look at how count is used. The Text widget expects a String, but we have a Future<String>. So the solution that comes to mind is to make the entire function async, and await count before handing it off to Text. The issue now is that we return a Future<Widget>, which is obviously not the same as a Widget, so it just won't work.

How do we handle this? Flutter has a Widget called the FutureBuilder that takes Future objects and returns Widgets from them. Here's how it works: the widget takes a future and makes Snapshot objects of them as it changes. Notice that now we're working with AsyncSnapshot<String> instead of Future<String>. So we break it down into three parts: first, if we have data, do the same thing we had before. Next, if we have an error, do something with the error (in this example, it's very unlikely we'll ever see that happen). Finally, if neither of those cases are true, show a loading indicator to tell the user that something is happening.

[Take Questions]