Meet N' Eat
API Final Project Description

Overview

Meet N' Eat is a social application for meeting people based on their food interests. For this project, you will create the API backend for this application using Flask. This project is intended to be very open-ended, allowing you to be as creative as possible with your implementation while still having endpoints that other developers can use.

Let's clarify the meaning of the following words that will be frequently used in this document:

Request - refers to one user specifying availability to meet for at a specific location, meal type and time of day (e.g. if a user wants to have doughnuts for dinner in Denver, Colorado "doughnuts" is the meal type, "Denver, Colorado" is the location_string, and "Dinner" in the meal_time)

Proposal - Users can view open Requests and if they are interested, can make a proposal, notifying the maker of the request that a user is interested in meeting.

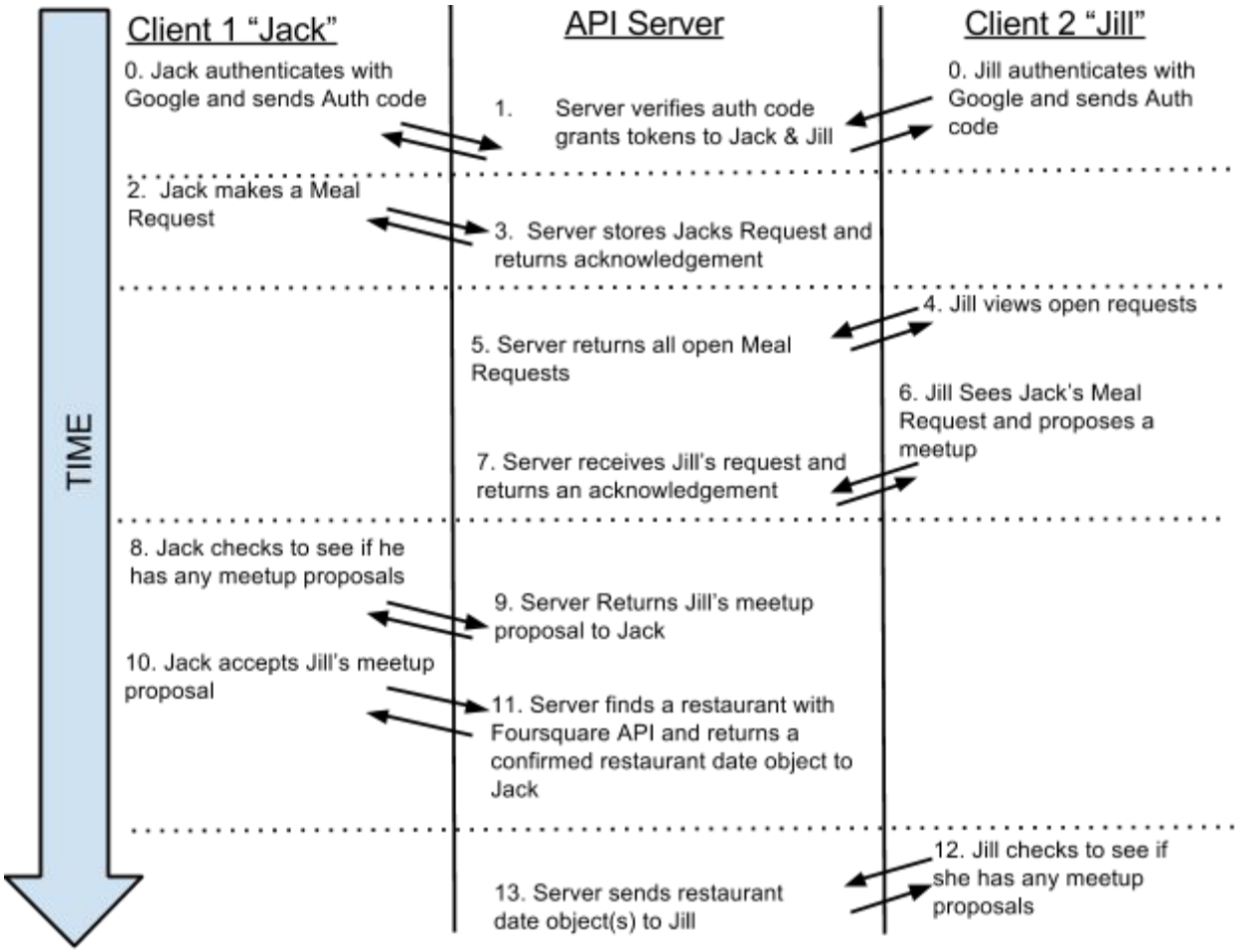
Date - If a maker of a request accepts a proposal, then a date is created on the backend. The date tells the users where to meet and at what time of day.

Features

- Meet N' Eat should leverage the Google Maps API and Foursquare API in order to find a restaurant for users and geocode locations into latitude and longitude coordinates.
- Users should be able to log in with either a username/password combination or using an OAuth provider like Google or Facebook. The application should generate it's own tokens for users.
- All endpoints should use rate limiting to prevent abuse of your API.
- Documentation for the API should be created in a way that is developer-friendly and aesthetically pleasing.

Communication Flow

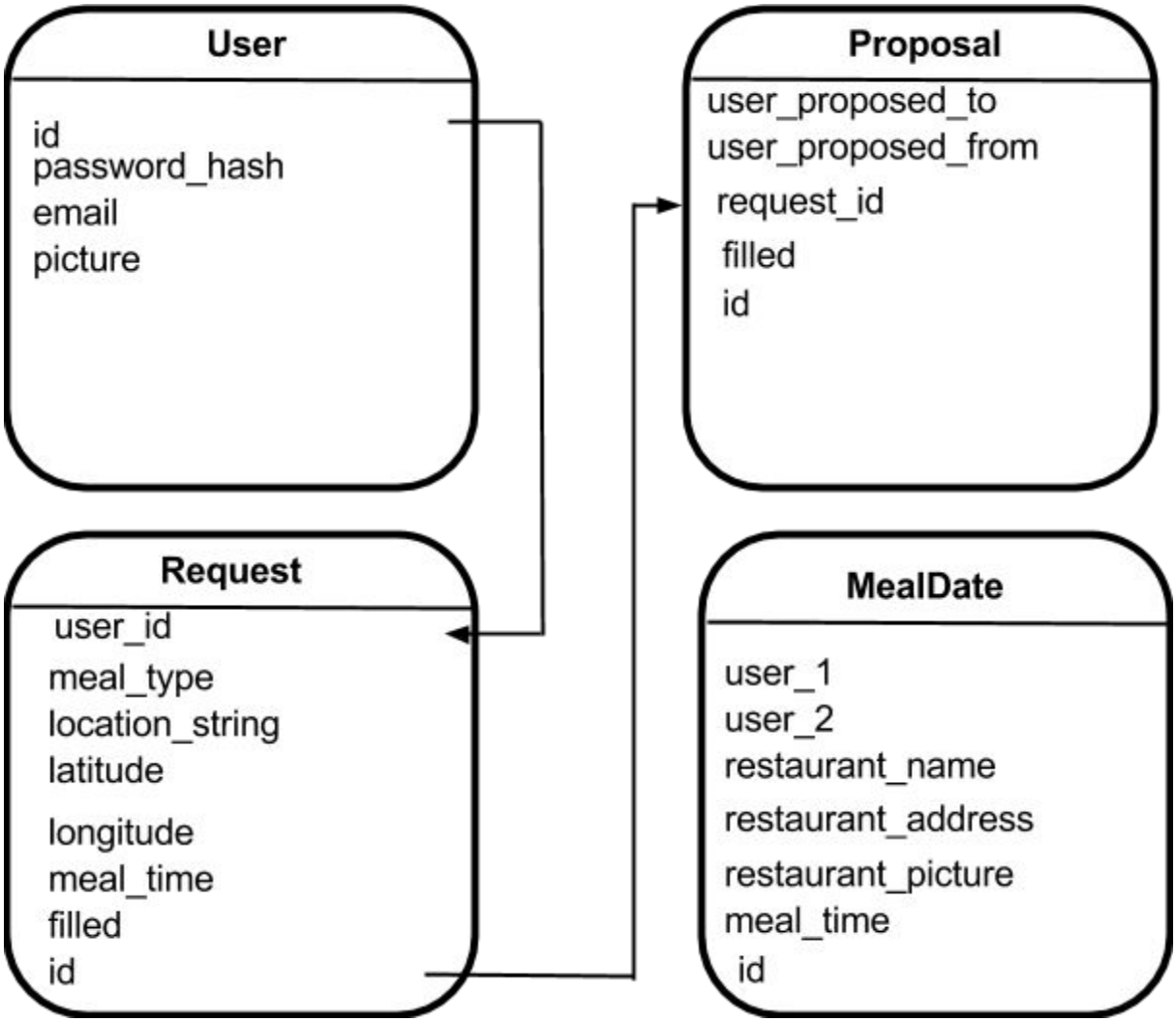
To better understand the functionality of this project. Walk through this timeline scenario that shows the basic client/server communications taking place.



These steps should provide some intuition in regards to the API endpoints your server should have. This is merely a suggestion, however, as how to go about implementing this project.

Database Layout

Below is an example database layout that you can use to implement your database models along with an explanation of each model. If this database layout does not meet the needs for the features of your application, you are not required to use it.



A **user** object represents each user of the Meet N' Eat application. The **request** represents an open request by a user who is open to eating a particular meal type(for example "Pizza", "Sushi", "Hamburgers") at a specific meal_time (for example "Breakfast", "Lunch" or "Dinner). Other users can browse open requests and if they are interested in meeting, a **proposal** object is created. The Proposal object contains the user id of the user who made the proposal to meet along with the user id of the recipient of the proposal and the id of the request object it relates to. If the recipient of the proposal agrees to meet then a **MealDate** object is created, finding a local restaurant that matches the requested cuisine type near the original proposer's location.

Endpoint Design

Below is an example of some of the URI endpoints that could be implemented to allow all of the desired functionality for this application. You should also decide whether parameters for your

api should be included within the URI parameters or embedded inside the body of the HTTP request.

HTTP Verb	Example URI	Description	Parameters	Logic/Security
POST	*api/v1/<provider>/login*	Allows logging into a specific OAuth provider and returns an access token to the client	-One-time Auth code	Server looks up user based upon the email address associated with the access token to identify user. If no user exists, then a new one is created.
POST	*api/v1/<provider>/logout*	Allows logging out of an application and rejects an access token.	-Token	Server looks up user based upon the email address associated with the access token to identify user. If user doesn't exist return an error.
GET	*api/v1/users*	Returns profile information off all users in the database.	Token	Only logged in users can view user profiles.
POST	*api/v1/users*	Creates a new user without using OAuth	-username -password	highly recommended to implement secure HTTP if this endpoint is implemented. As long as an existing username isn't in the database, create a new user, otherwise, return an appropriate error.
PUT	*api/v1/users/*	Updates a specific user's information	-Token -new user profile	Server checks token to make sure only the logged in user

			information	can update her profile.
GET	*api/v1/users/<int:id>*	Returns information for a specific user.	-Token	Only logged in users can view profile information
DELETE	*api/v1/users/*	Removes a user account.	-Token	Only the user with the correct token can erase her account.
GET	*api/v1/requests*	Shows all open meetup requests.	-Token	Only logged in users can view all open requests. **Advanced feature**: a user should not see their own requests, only everyone else's
POST	*api/v1/requests*	Makes a new meetup request	-Token -meetup request information	Only logged in users can make meetup requests. The id of the maker of the request is stored in each request object.
GET	*api/v1/requests/<int:id>*	Returns information for a specific meetup request	-Token	Only logged in users can view the details of a meetup request.
PUT	*api/v1/requests/<int:id>*	Updates information about a meetup request.	-Token -new meetup request information	Only the original maker of the request should be able to edit it.
DELETE	*api/v1/requests/<int:id>*	Deletes a specific meetup request	-Token	Only the original maker of the request should be able to delete it.
GET	*api/v1/proposals*	Retrieves all meetup proposals for a given user	-Token	User is verified by the provided token and only corresponding proposals are

				returned if they are the maker or the recipient of a proposal.
POST	*api/v1/proposals*	Creates a new proposal to meetup on behalf of a user	-Token -request_id	User is verified by the provided token and identified as the maker of the proposal. **Extra:** The user id of the person proposing and the recipient of the proposal should not be the same.
GET	*api/v1/proposals/<int:id>*	Retrieves information about a specific proposal.	-Token	the id of the user should match either the proposal maker or recipient in order to access this view.
PUT	*api/v1/proposals/<int:id>*	Updates information about a specific proposal	-Token -new proposal information	the id of the user should match the proposal maker in order to access this view.
DELETE	*api/v1/proposals/<int:id>*	Deletes a specific proposal	-Token	the id of the user should match the proposal maker in order to delete a proposal.
GET	*api/v1/dates*	Gets all dates for a corresponding user	-Token	Only the dates that contain the user id as one of the participants should be viewable by that user.
POST	*api/v1/dates*	Creates a new date request on behalf of a user	-Token -Boolean	If True, the recipient of a proposal has accepted this offer and is requesting that the server create

				a date. If false, the recipient of a proposal rejected a date and the proposal is deleted.
GET	*api/v1/dates/<int:id>*	Gets information about a specific date	-Token	Only dates where a user is a participant should appear in this view.
PUT	*api/v1/dates/<int:id>*	Edits information about a specific date	-Token -new date information	Only participants in the date can update the date details.
DELETE	*api/v1/dates/<int:id>*	Removes a specific date	-Token	Only participants in the date can delete a date object.

Rate Limiting

Since we are using HTTP our application functions on pulls from the client. This means that the client must periodically send requests to the server in order to see updates to the state of the application.

Make sure you application implements rate-limiting while still allowing a reasonable amount of requests such that the client can have a usable application that doesn't flood your server. For an application of this sort, 3 requests per second (or 180 requests per minute) should be more than sufficient for any endpoint, but you can adjust how you see fit.

The examples from the course reveal rate-limiting based on IP addresses. If you're feeling extra Udacious, try implementing rate limiting keys based on the user id.

Documentation

Create documentation for developers wanting to implement your API. Be creative and think of the most developer-friendly ways you can convey information to the developer allowing him or her to quickly be able to use your endpoints.

Front End

Try creating a front-end for your application that uses the provided functionalities. You can make a web application using HTML/CSS/JavaScript, a Python application using a Graphical User Interface like Kivy, or if you know mobile development even an Android or iOS app that uses your API endpoints.

Share and swap your front-end applications with other students in the forums.

Going Further

Have some ideas for version 2.0? Maybe adding specific dates and times for meetups, or allowing users to send messages? Take some notes on some enhancements to this application you would like to add. Implement this more advanced version and add it to your portfolio. Solution code for this basic exercise is provided in the instructor notes but try to finish your own first prototype before referring to it.