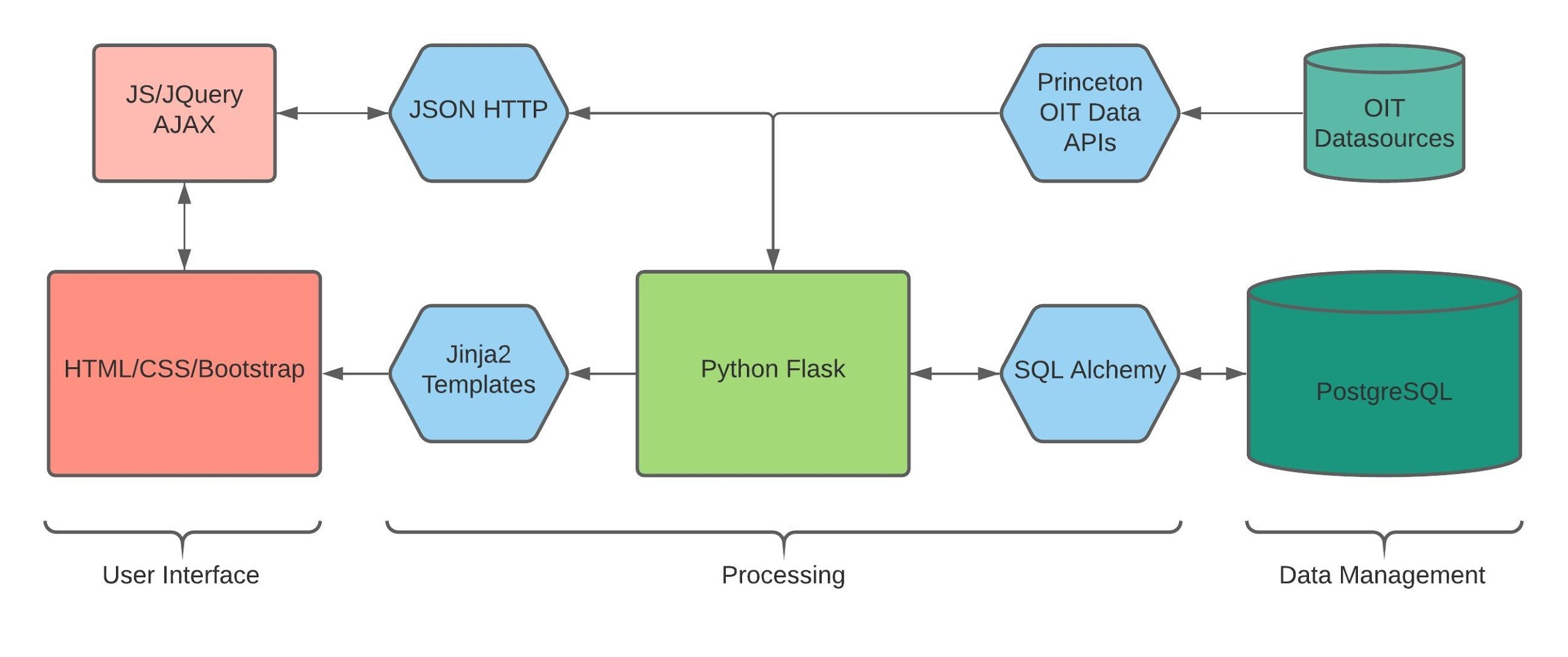
Class 5 – Programmer’s Guide

High-level Overview

**Technologies**

The Class 5 system is built using the Python Flask framework with the Jinja2 template model and a PostgreSQL Relational Database.



**Front-end**

1. **HTML/CSS/Bootstrap** for Layout and User Interface Design
2. **jQuery** and **JavaScript** for responsive web pages and AJAX calls

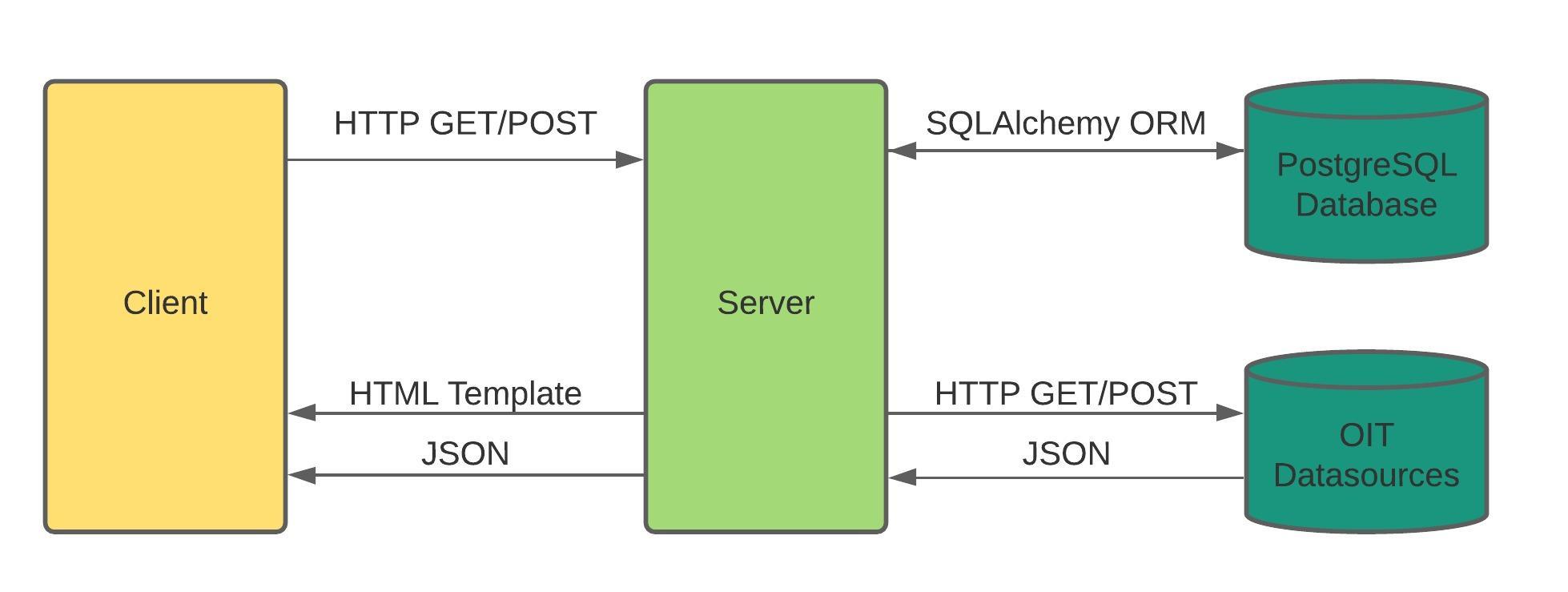
**Processing Tier**

1. **Python Flask** is the backbone of our system. It processes all the incoming HTTP requests and handles all communication with the databases.
2. For handling page requests, the **Jinja2** template engine generates responses
3. For handling API requests, Python generates **JSON** responses
4. The **SQLAlchemy** ORM is used to ease our communication between Python and the PostgreSQL Database

**Data Management**

1. A **PostgreSQL** relational database handles the majority of persistent data on our server including user profile information and Princeton courses data
2. Other course information, such as new course info for the upcoming semester, is pulled from the OIT Datasources
3. Some course information, such as certificate names and concentration information, is not stored in the database at all and is instead hard-coded into Python dicts. This decision was made because there are less than 100 records – so space was not an issue – and because the data was obtained by us programmers instead of from a data source or user.

Communication



This figure highlights the communication protocols and response types between the various tiers of the application

**Client-Server Communication**

At a high-level, a client can make either HTTP GET or HTTP POST requests to the server. The GET requests are for accessing pages or API data on the server\*. The POST request is for updating the user’s information, such as Liked and Disliked courses.

The server can respond to requests in 2 ways:

1. With a Jinja2 template containing HTML which we call a page
2. With JSON

These responses extend to error handling for the client. In the event that an error occurs on the server, such as while accessing the database, if the intended response type was supposed to be a template, the status.html template is instead returned with an appropriate error message for the client. If the intended response was supposed to be JSON, then a JSON response with an appropriate error message is returned.

**Server-Database Communication**

The Flask server communicates with the PostgreSQL database exclusively through the SQLAlchemy ORM.

The procedure for communicating with the OIT Datasources, specifically for MobileApp for generating course info and ActiveDirectory for user info, was originally adapted from the code from this github repo: [https://github.com/vr2amesh/COS333-API-Code-Examples](https://github.com/vr2amesh/COS333-API-Code-Examples//). For communication with the Princeton OIT Datasources, the server makes HTTP GET and POST requests. First, a POST request is sent to <https://api.princeton.edu:443/token> which generates a new access token for the server. With this token, the server can now make GET requests to access the course info data, which the Datasource responds with in a JSON format. This course information is then stored in the PostgreSQL database whenever it is initialized.

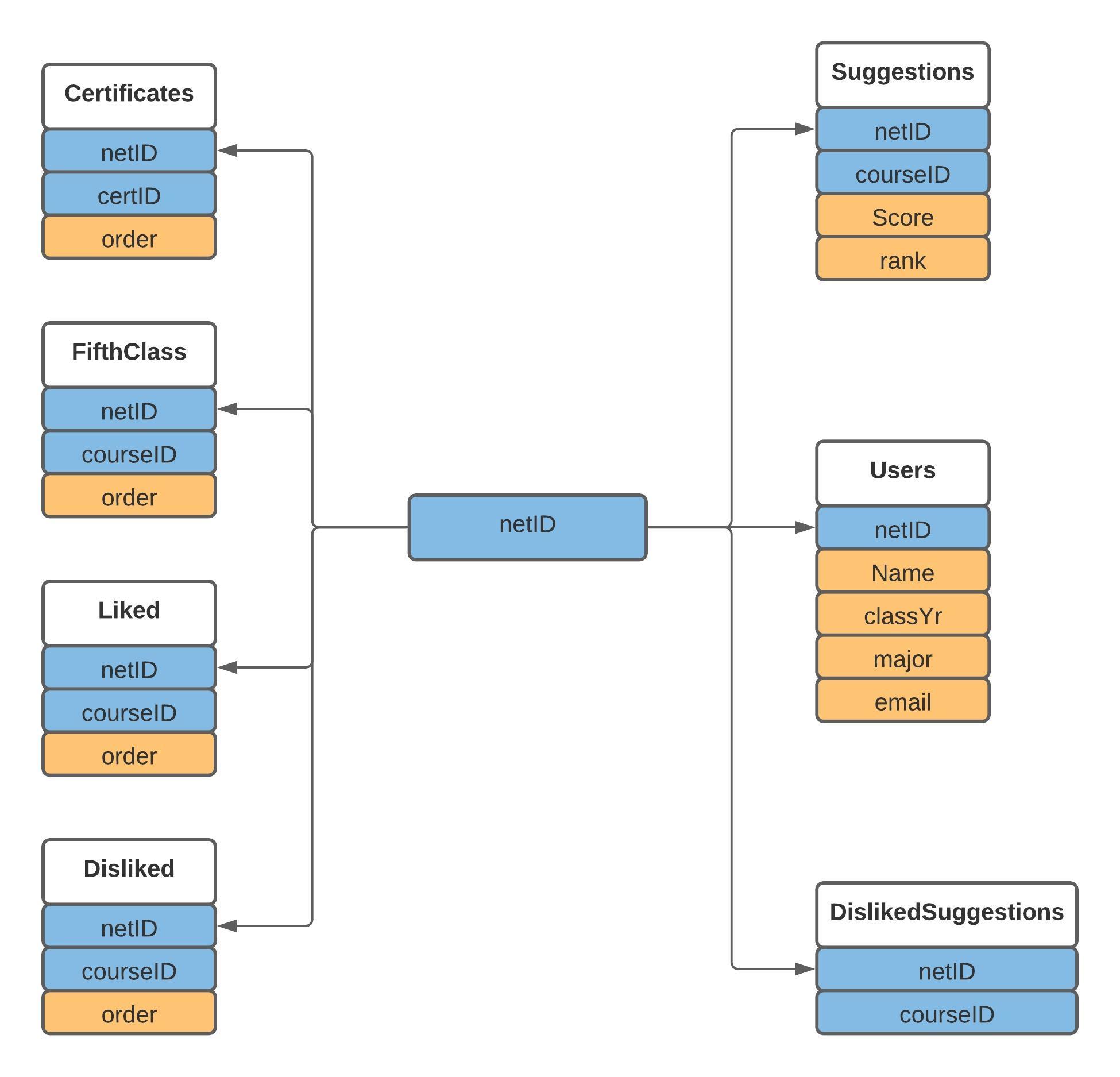
For accessing the certificate name and major data, since this is stored locally in Python, it is simply called from the dict it resides in.

Database Schema

**Overview**

For the majority of data storage, the application uses a PostgreSQL database with 11 tables: 7 for User Information, 4 for Course Information.

**User Info Tables**

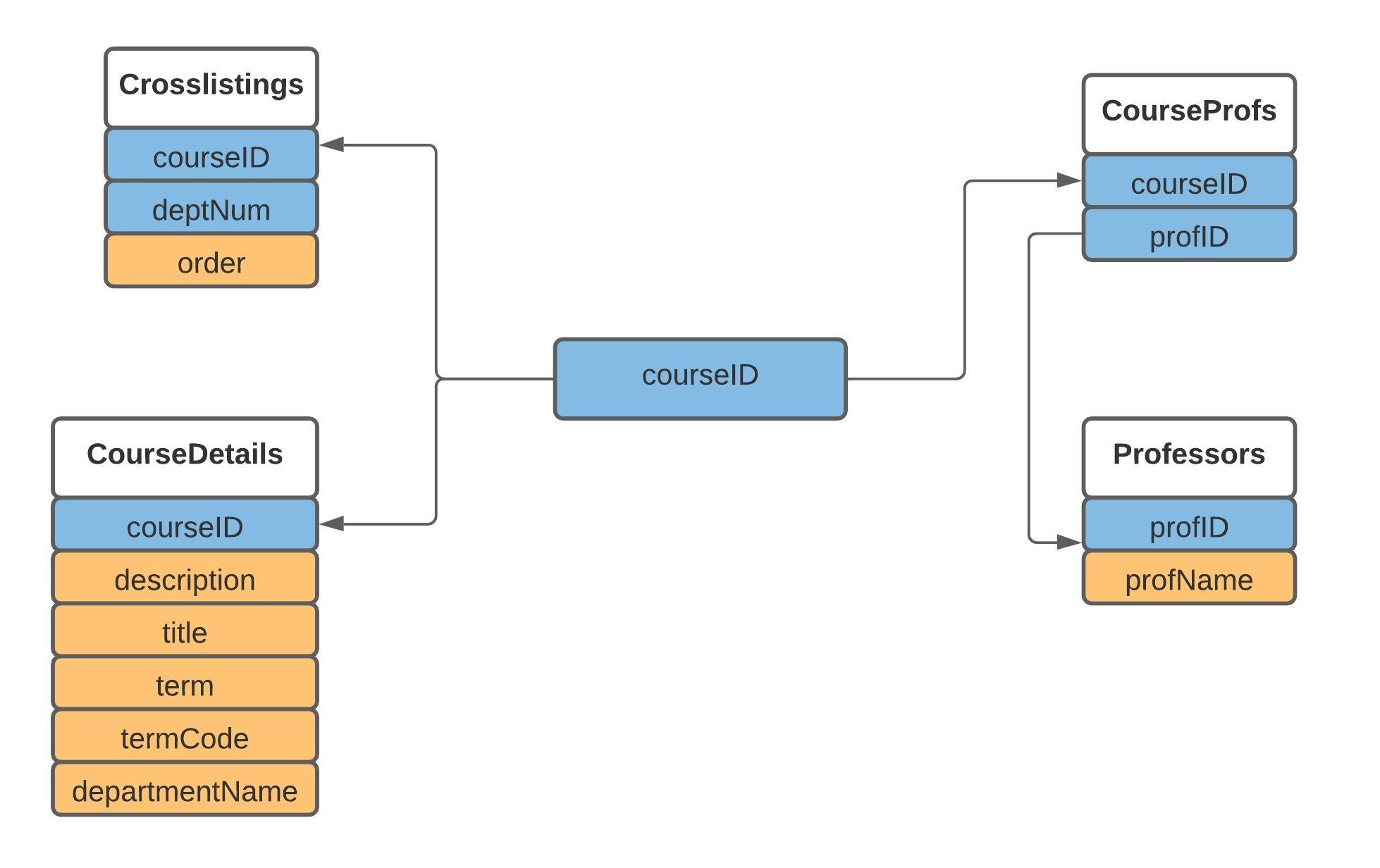


The above figure shows the layout of the 7 User Info tables. These tables hold all of a user’s information, such as what courses they like and dislike as well as their name and credentials. Until a user has at least 2 Liked and Disliked courses, they cannot access their suggestions. The blue boxes of each table are the columns of the table that are used for its primary key. The common column that all these tables share is the netID, which allows the app to determine information about a specific user. However, in order to follow the database rules, many of the tables have a primary key of 2 columns, (netID, courseID), which ensures that each row has a unique identifier and that the database is in 3rd Normal Form.

It is also important to note the order column in many of the tables. This column, similar to a create, is the order in which the elements were input. It helps to maintain this order when the data is retrieved for the user.

All the tables are updated by a HTTP POST request to /profile. DislikedSuggestions can also be updated during a HTTP GET request to /refresh.

**Course Info Tables**



The above figure shows the layout of the 4 Course Info tables. These tables hold all the course information needed for the application and follow the exact same format as was used in the assignments. As a result, the database is in 3rd Normal Form. These tables are mainly accessed using the courseID column.

Unlike the User Info tables, these tables are not updated by users, but instead by the programmer. Much of the information is pulled from the OIT datasources and populated using the initialize\_database.py module (see “Installation” below). Ideally, the programmer will update this information each new term of the school year to add in the new semester’s courses.

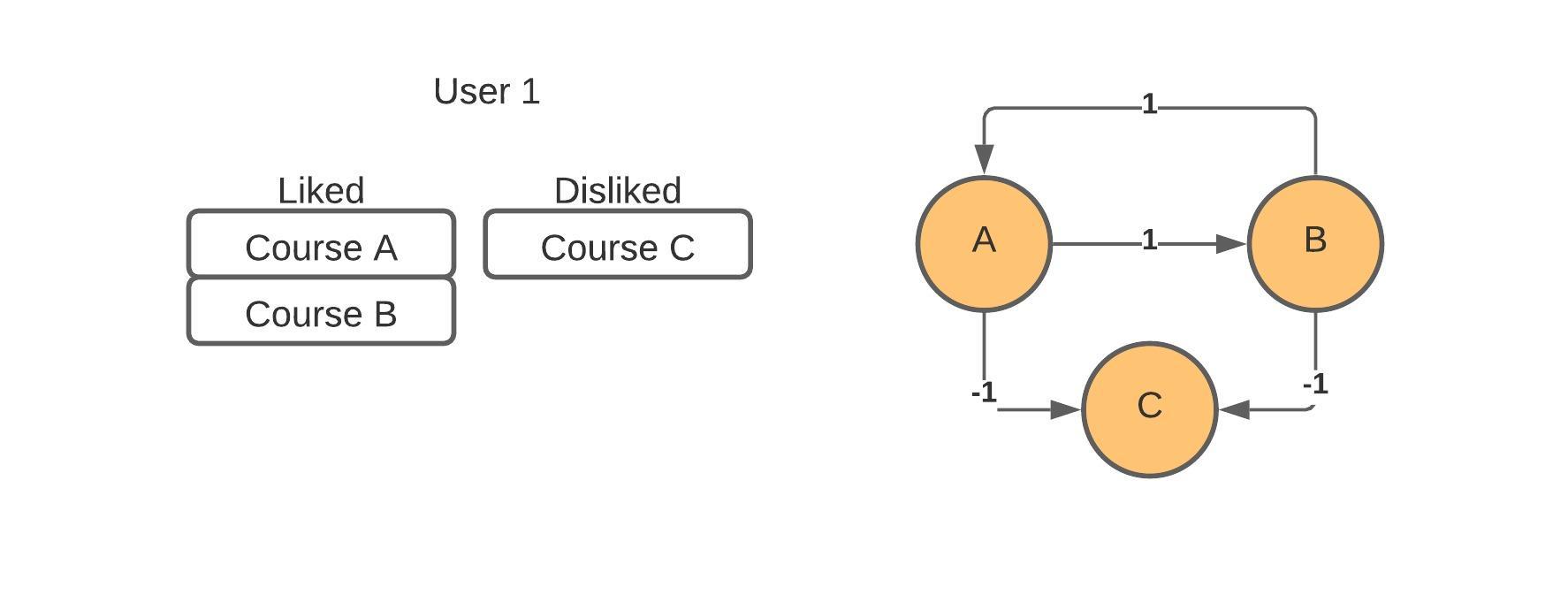
Course Graph Abstraction

**Overview**

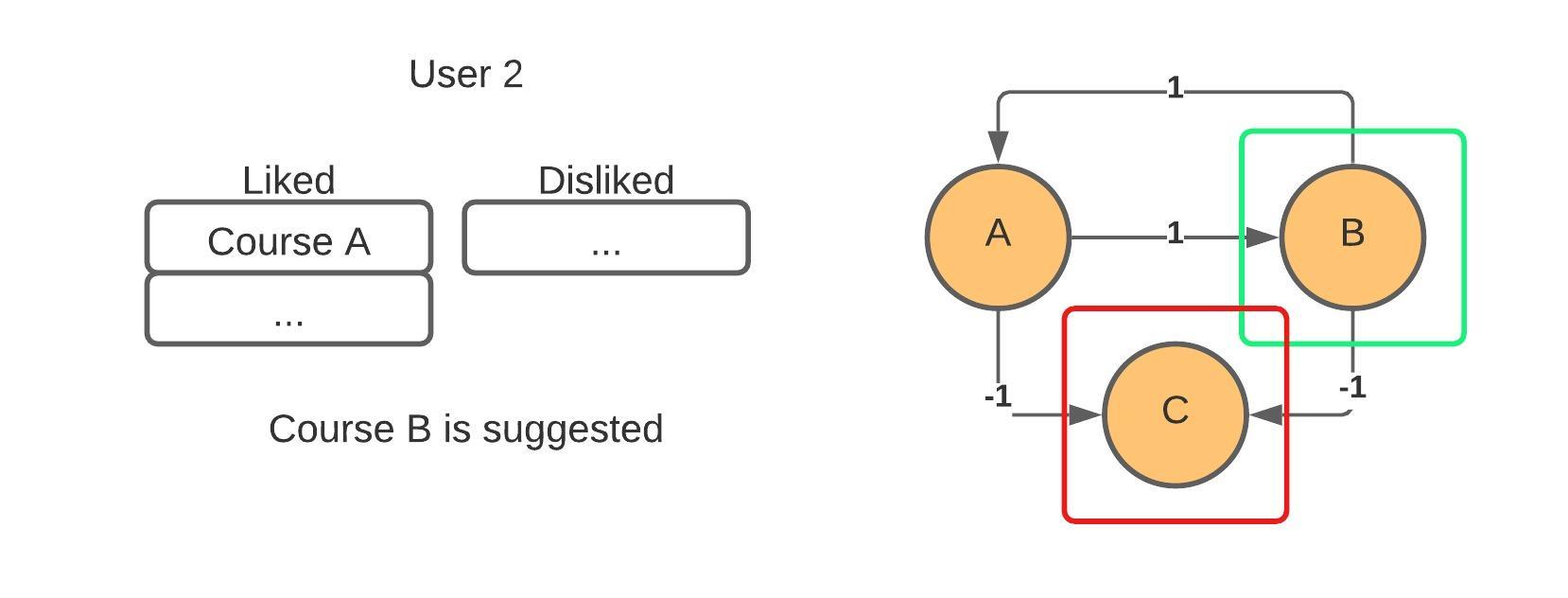
At the heart of our application is the suggestions and the intuition behind these suggestions can be thought of as a graph with courses as nodes and the relationships between courses as weighted-edges.

The graph is initialized as a directed graph with all edge weights of 0 (using the Adjacency list representation, edges with weight 0 are equivalent to edges that do not exist, and so are not stored):

1. Say User 1 logs in and creates a profile where they like Course A and Course B and dislike Course C. When User 1 saves their profile, the graph weights are updated to what is shown in the first figure below in the following way:
   1. Add 1 to the weight of any Liked --> Liked edge
   2. Subtract 1 from the weight of any Liked --> Disliked edge
   3. Do not change the weight of Disliked --> Liked or Disliked --> Disliked edges



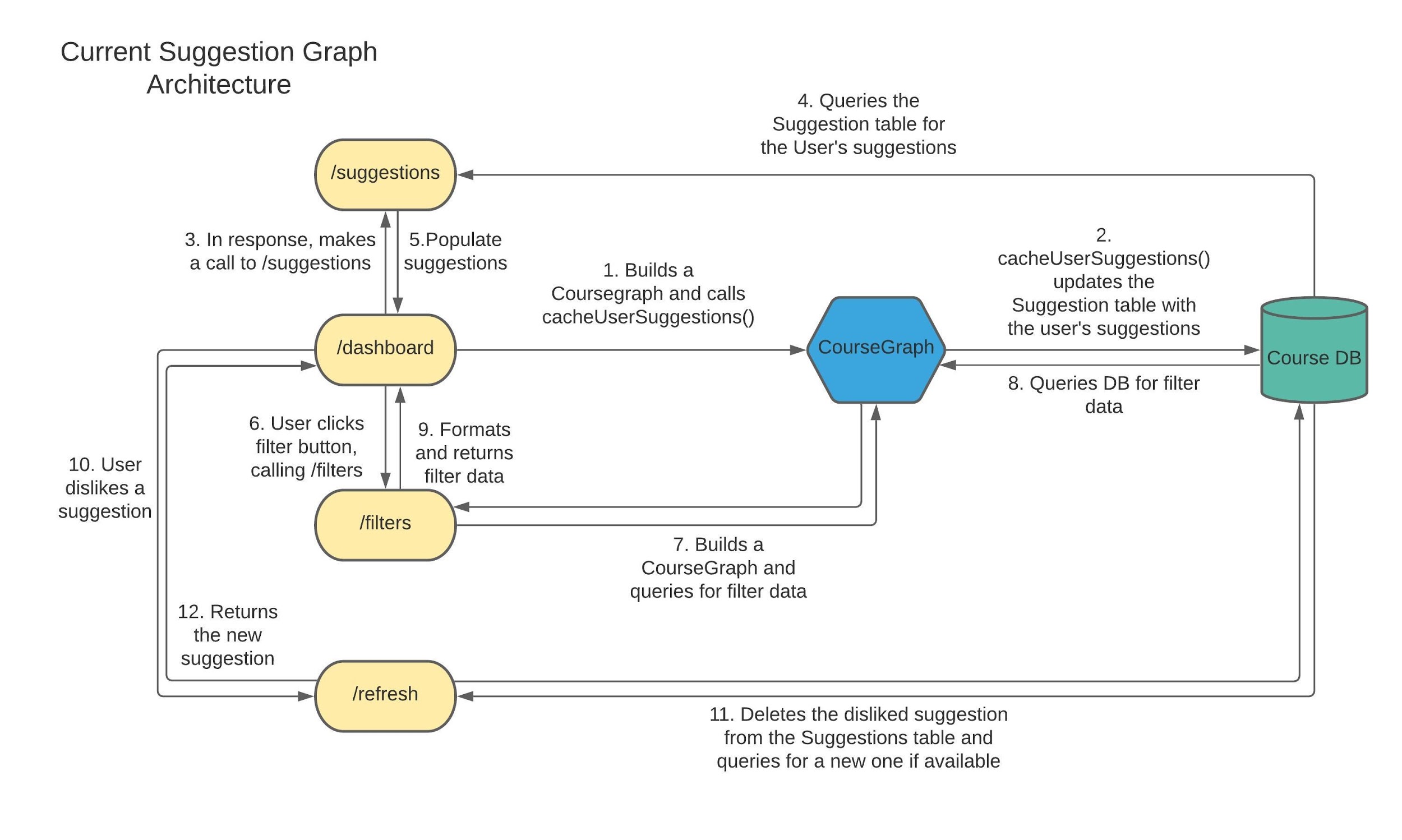
1. Now say User 2 logs in and creates a profile where one of their liked courses is Course A. In order to generate suggestions for User 2, the graph looks at the courses User 2 likes, such as Course A, and looks at the highest edge weights. In this case, (A --> B) = 1 is the largest, so Course B is then suggested to User 2 while Course C is not.



1. Additionally, every course node has a “Unitary Weight” (also called “Popularity Score”). This value corresponds to (# of appearances as a Liked course) - (# of appearances as a Disliked course). For generating suggestions, the algorithm uses a weighted combination of the edge weights and popularity score (ie. For a user, the suggestion algorithm returns a list of 10 courses with the top 10 highest scores. Currently, score = 2\*(Edge weight) + 1\*(Popularity Score)). This popularity score helps fill the list of suggestions for courses that do not have lots of edges. This is important for users who like courses that no one else has liked, since these courses do not have pre-existing edges. Since the edges do not exist, these users still get good suggestions from the most popular courses

**Implementation of Course Graph**

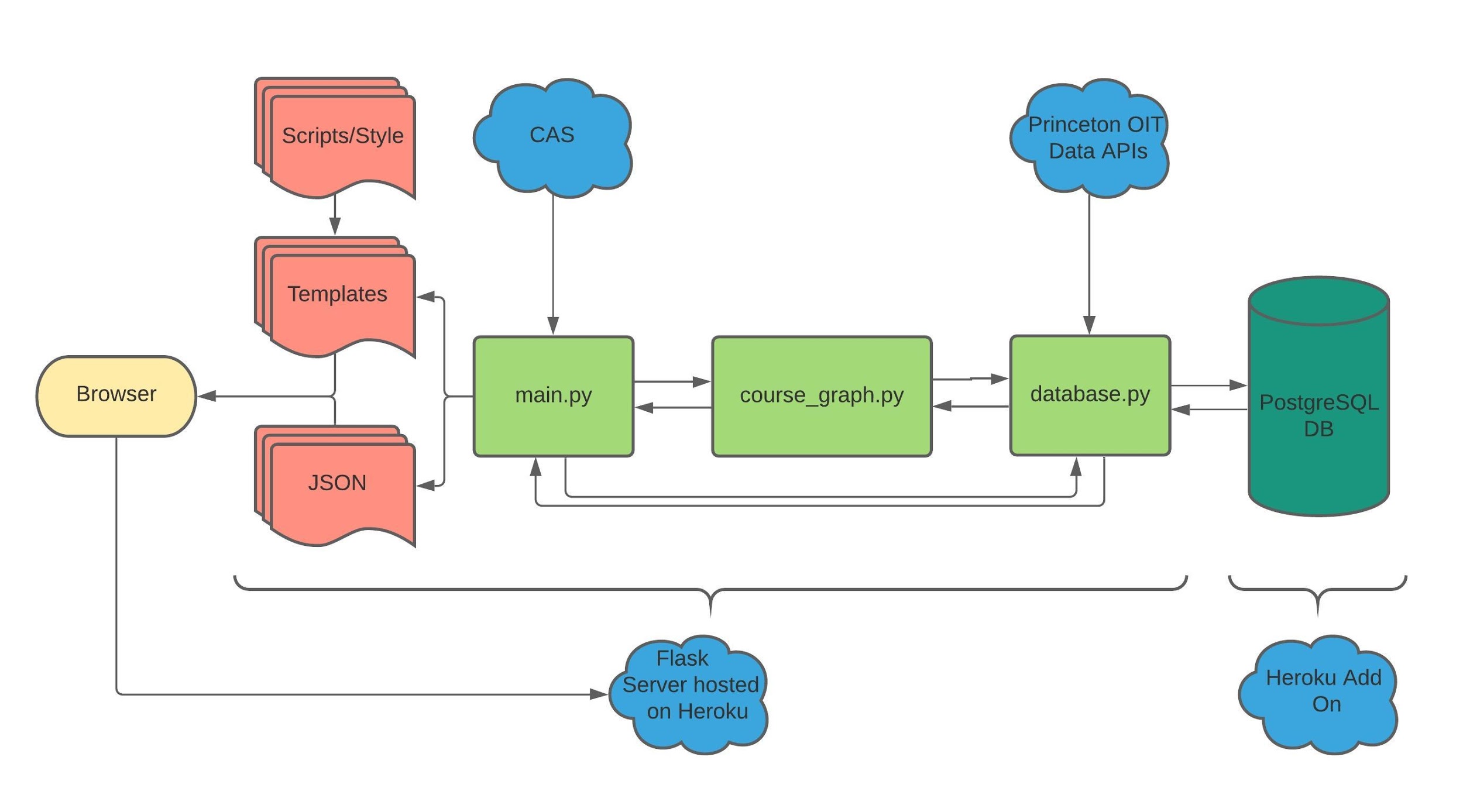
Due to the short time frame of this project, we decided to use the PostgreSQL Relational database for our data storage because it worked well with Heroku; however, relational databases are not the best format for representing a graph since certain common operations, such as edge-lookup, are inefficient. While for the small size of this project, PostgreSQL works fine, if we were to continue scaling Class 5, we might consider switching to a graph-based database like Neo4j or an in-memory store like Redis. For the time being, we instead created a Course Graph abstract data type in Python that is meant to help maintain the intuition of thinking about our system as a graph.



The above figure displays how different endpoints interact with the Course Graph ADT and Databases. Steps 1-5 handles the typical flow of what happens when a user goes to /dashboard. Steps 6-9 handle when a user clicks a filter, and steps 10-12 handle when a user dislikes a suggestion.

**Note**: As the figure indicates, this current implementation of the abstraction is still fairly messy. Ideally, the Course Graph ADT would be the only entry point between the user and the Course Info in the Database. Next steps for this application would be cleaning up this ADT or switching over to the graph-based database.

**Important Modules**



**Main.py**

As the name suggests, main.py is the central hub of the Class 5 application. The module contains all the endpoint handling functions of the application. Here is a comprehensive list:

Page Endpoints (Return a Jinja2 HTML template)

|  |  |  |
| --- | --- | --- |
| **Path** | **Description** | **Method** |
| /, /index, /home | View home page | GET |
| /faq | View FAQ Page (in progress) | GET |
| /login | Initiate CAS login → Redirect to /dashboard | GET |
| /logout | Initiate CAS logout → Redirect to /home | GET |
| /dashboard | View suggestions page | GET |
| /profile | View profile page | GET |
| /profile | Update profile page | POST |

API Endpoints (Return JSON)

|  |  |  |
| --- | --- | --- |
| **Path** | **Description** | **Method** |
| /suggestions | Retrieves a list of suggestions for a user | GET |
| /refresh | Removes a disliked suggestion and returns a new suggestion | GET |
| /courses | Retrieves a list of all courses that start with the supplied substring | GET |
| /majors | Retrieves a list of all majors that start with the supplied substring | GET |
| /certificates | Retrieves a list of all certificates that include the supplied substring | GET |
| /filters | Retrieves a list of courses with the supplied filter type | GET |
| /course-info | Retrieves a supplied course’s course information | GET |

**Course\_graph.py**

This module represents the Course Graph ADT and its corresponding methods. These methods are mainly called in main.py to generate and cache user suggestions and filters. As explained above, there is still a fair bit of cleaning that needs to be done to fully take advantage of this abstraction.

**Database.py**

This module is the data abstraction layer for the PostgreSQL database. It contains the SQLAlchemy classes that represent the tables in the database, both for the course info and for the user info. The methods in database.py are primarily called in main.py and course\_graph.py to access and update the database.

**Create\_data\_dicts.py and initialize\_database.py**

These files interact with each other to populate the database with information from Princeton’s OIT Data APIs. For full details on how to use these files, please refer to “Installation” near the bottom of this file. We basically get JSON responses and reformat them to fit our needs into dictionaries. We then use these dictionaries and iterate through their information to populate our database.

While we haven’t specifically written code for updating the database, initialize\_database.py can be used for updating course information and certificate information. Simply load in the dictionary containing course information from the terms of your choice, and only run the pertinent course information population functions.

Frontend Maintenance

**Overview**

While the user interface is quite extensive as a responsive mobile web app, the frontend code is relatively simple and straightforward to understand. For styling, the HTML templates use a mixture of Bootstrap libraries and barebones CSS located in the /static/style/ directory. All images are located in the /static/img/ directory.

For responsiveness, the app uses a mixture of JavaScript and JQuery located in the /style/scripts/ directory. All AJAX calls are made using the JQuery $.ajax() method and maintaining global request variables.

Both Bootstrap and JQuery are loaded using external CDNs, so we always get the most updated versions.

**Maintenance of Major Features**

Autocomplete

The autocomplete feature for forms was adapted from a W3Schools tutorial located here: <https://www.w3schools.com/howto/howto_js_autocomplete.asp>

For maintenance pertaining to autocomplete, the AJAX getter methods are located in **autocomplete.js** and the styles pertaining to displaying the dropdown is in **autocomplete.css**

Populating Suggestions and Filters

Loading and clearing lists of suggestions and courses based on filters is a key part of the Class 5 application. For maintenance pertaining to creating and clearing these lists, **dashboard.js** contains the JQuery AJAX getter methods.

Course Information

For maintenance pertaining to displaying course information and updating the course info when a new list of suggestions is loaded, look in **dashboard.js**

Persistent User Information

For maintenance pertaining to retrieving user information on the Profile Page, the data is generated using the Jinja2 templates in **main.py** but is then stored as global Javascript variables in **profile.html**. These global variables are then accessed in **profile.js**, which uses them to populate the data. As the application continues to scale, we intend to move away from passing in variables through the Jinja2 templates and instead load all frontend data through AJAX calls, which will make maintenance easier.

“Are You Sure” Alerts

For maintenance pertaining to checking if the user has saved their profile information, the main implementations reside in **profile.js**, **dummy-input.js**, and **jquery-are-you-sure.js**, called in **profile.html** (this code was heavily adapted and modified for our needs from a GitHub repository named: <https://github.com/codedance/jquery.AreYouSure/>). This repository checks if an input in a form has been modified, and if the user tries to unload the page, the browser will display a prompt asking if the user wishes to leave. The problem with this is that A) browser messages are generic and irreplaceable and B) if a user adds or removes an input, this package will not detect that.

To make a specific browser message, we catch an input change event and install a handler on specific links in the navigation bar. This handler will create a javascript confirm box and display it to the user if they click on a link within the website after having changed a form input. It will also suppress the unload handler that jquery-are-you-sure has predefined so that the user does not get recurrent messages (this code is in jquery-are-you-sure.js).

In order to make responsive elements fire input events, we create a ‘dummy’ input within every form ‘bubble’ (this code is in dummy-input.js). This dummy input is hidden from the users, but we need this dummy input so that the package jquery-are-you-sure can work with our code. When a user edits an input field, deletes an input field with text in it, or adds an input field and then writes text in it, the corresponding dummy input’s inner text will change, and we will send off a changed input event. This code is called within profile.js for modularity purposes. Essentially, the responsive elements (‘+’ and ‘x’ buttons) are communicating with a dummy input so that the correct events can fire if a user makes a change to the form.

CAS Client

For maintenance pertaining to CAS, the validation and authentication code itself is located in **cas\_client.py** which follows from CASClient.py written by Alex Halderman, Scott Karlin, Brian Kernighan, and Bob Dondero. Most of the endpoint flask methods in main.py call an authenticate() method located in **main\_util.py** which cuts down on code repetition.

Responsively Disliking Suggestions

For maintenance pertaining to removing suggestions from the list and adding them to Disliked Suggestions, the code for event-handling and frontend logic is located within **dashboard.js**, and the code interacting with the database resides in **main.py**. Whenever a suggestion is removed, *removeSuggestion()* is called in **dashboard.js**, which removes the <div> and disables the “click” event for “x” buttons on all other suggestions. This is done by setting the attribute of ‘onclick’ to the empty string (‘’). *RemoveSuggestion()* then calls *refreshSuggestions()*, passing in the removed deptNum as an argument, and sends this Disliked Suggestion to the server within a request. In **main.py**, the callback function *refresh()* is called, which interacts with the **database.py** module to update the disliked suggestion and fetch a new suggestion. Back in **dashboard.js**, once the response is received in *handleSuggestionResponse()*, a new suggestion is appended to the list, and the click events for remove buttons are restored.

**Note**: The ‘onclick’ attribute is not set back to *‘removeSuggestion’* until *after* a refreshed suggestion has been retrieved and appended to the end of the Suggestion list, which guarantees that only one request will be sent to the database at any given time.

Installation

**Dependencies**

1. Clone the Class 5 repository to your computer by running “git clone https://github.com/EthanSei/Cos-333-Project-Class-5”
2. Create a new Python environment using Python 3.7 (3.7.6 is preferred)
   1. Use this environment whenever running code within this project.
   2. In this Python environment, install the packages within the file titled “requirements.txt” by running “pip install -r requirements.txt”.

**Database**

1. Create a PostgreSQL database.
   1. For example, on the Heroku App dashboard, go to the Resources tab and type in Heroku Postgres in the Add Ons search bar (See “Deploying on Heroku” below for instructions on how to set up Heroku)
2. Within the “Database” folder, find the file named “configs.py”.
   1. Change the environment variables as prescribed below:
      1. DB\_HOST: host on which the database is running
      2. DB\_NAME: name of the database to connect to
      3. DB\_PORT: (default is 5432) port where the database server is listening
      4. DB\_USER: username to connect to the database with
      5. DB\_PASSWORD: password associated with your username
3. Populate the database
   1. Go to /DataSources/course\_info.py
      1. At the very top should be a variable named “ORDERED\_TERM\_CODE\_LIST”.
         1. This list contains the term codes to access the Princeton course information from. For example, code ‘1214’ would get the course information (i.e. cross listings, professors, course titles, etc.) for the Spring 2021 semester.
         2. These course terms are formatted as follows:
            1. 1 + last\_two\_digits\_of\_the\_semester\_year + term code (1 for summer, 2 for fall, 3 for winter, 4 for spring)
            2. Example: ‘1212’ would be the Fall 2021 semester. ‘1204’ would be the Spring 2020 semester
         3. **Note**: the term codes must be ordered in descending chronological order (i.e. the most recent term code should be placed first)
         4. You can enter multiple term codes to get information for courses that are not offered in multiple years. More details will follow in the “Data Sources” section of this paper.
   2. In the command prompt, execute: **python create\_data\_dicts.py** within the main directory.
   3. Then execute: **python initialize\_database.py** within the main directory.
      1. This file will require the user to enter the password to the database as prompted input (i.e. DB\_PASSWORD that was set earlier in these steps)
4. Run the application
   1. Execute: **python main.py** within the main directory to run the app locally
   2. Go to your browser and enter the following into the search bar: “localhost:55555”

**Deploying on Heroku**

1. Go to <https://dashboard.heroku.com/apps> and login or create a new user account
2. Go to New > Create New App
3. Input a name (“class-5-evaluation” is available as of 12/5/2020)
4. On the Deploy tab, click connect to Github → Connect your Github and add the above repo
5. Make sure your github has a requirements.txt, a runtime.txt, and a Procfile with “web: gunicorn main:app” in the root level of the directory.
6. Once your Github is connected, you can click the: Deploy Branch button to deploy the build
7. Once deployed, click View to see the project

Error Handling

If the application encounters an error, such as a problem with the database, the majority of the handling is done in try/except statements in each endpoint method of main.py. These try/except statements call the handleError() method in main\_util.py, which outputs an appropriate, descriptive message for the programmer in the server-side logs, and a separate, more vague message for the client, alerting them that a server-side error has occurred.

If the error occurred where a page response was expected, handleError() returns the status.html page with an appropriate client-side error message. If a JSON response was expected, handleError() returns a JSON object with an appropriate message.

For viewing server-side error messages, you can view the logs Dashboard > More > View Logs on Heroku. The message will write to stderr a general description of what was happening in the code when the error occurred and the exact error message itself. In no reasonable instance, should the application crash.

We also use database rollbacks and input validations to maintain security and prevent user corruption.

Design Problems

**Pulling outside course information**

* Problem: We needed to find a maintainable way to pull information from outside sources that was also easily expandable.
  + We also needed a way to store course information from multiple years
  + For example, COS 226 is offered every semester, so the course information should always be for the current semester
* Solution: We decided to allow the maintenance programmer to input a list of course terms they want to get information from.
  + This list must be ordered in descending chronological order (i.e. most recent course term first).
  + The dictionaries will not overwrite any information already stored, so let’s say the term is Spring 2020 and COS 226 has been stored in the dictionary, it’s information will not be overwritten by the course information from Fall 2019, for example.

**Updatable Database**

* Problem: Expanding on the above problem, we needed our database to be updatable, i.e. I should be able to add new users, new courses, etc.
  + We originally wrote our database to have explicit add, delete, and update abstractions. However, we found this to be cumbersome and error prone (due to the many moving parts).
* Solution: To combat this, nearly every database function abstracts adding and deleting and instead updates the database.
  + This is really nice as nearly everything can be added or updated with one function call per task, which is a simple abstraction.
  + For example, I can initialize the database with course information from the Spring 2020 semester, and I can update it with the same function calls with information from the Fall 2020 semester.

**Updating Edges in the CourseGraph**

* Problem: One major feature of our system is that we allow users to go back and change their preferences in the profile page. The idea is that they can get a new set of suggestions
  + Of course, because our backend is a graph, we need to edit this graph according to the user’s new preferences.
* Approaches:
  + At first, we decided to optimize for time. We only decided to update the edges that a user had changed (and we kept the existing edges the same)
    - This is problematic because it introduces a bug where not all relevant edges were changed
  + We then tried to fix the bug, so we decided to update the edges that a user had changed, and also the current edges that were existing in the database from those edges…. You can see why this approach was also problematic.
    - There were too many edge cases to consider.
  + Then we decided to not optimize for time and to just get a working system
    - Of course this ended up working much better, but it was much slower.
    - Now *whenever* the user hits the save changes button, the system will update every edge corresponding to that user, whereas in the past, let’s say a user didn’t update anything and tried to save changes, the system wouldn't have changed anything.

**Form Validation**

* Problem: At first, if a user made a mistake in the Profile page, i.e. entered a wrong course department and number, we used to not commit any of those changes. Users told us this was a problem as, on their initial welcome to the website, if they made a mistake, all of their progress would be lost. This is clearly undesirable.
* Solution: The solution was to commit fields that had no connection to the errors that were made. For example, if I put in a wrong liked course department and number, my disliked course changes will still save.
* New Problem: This new robust saving system introduced a problem to our course graph edges system.
  + Previously, we would store the user’s inputs and their relationships to the database. If a user made a mistake, we wouldn’t have to store the mistakes (as we would defer storing anything).
  + However, now, if a user makes a mistake, we still very well might have things to change in the course graph.
    - But if a user made a mistake, how are we supposed to update the edges with the correct information?
* Solution:
  + If a user made a mistake, let’s say in the liked courses section, we would have to take the liked courses from the database and use those to update the edges. Of course, this seems simple in practice, but we needed the implementation to be absolutely correct and robust to errors. It also needed to be maintainable. So we heavily modularized and documented the code.

**Fully Populating User Suggestions**

* Problem: As explained above, our course graph works by suggesting courses to users by examining the edges of the courses they liked. However, we also need to be able to suggest 10+ courses to users who have liked courses that have no or very few edges.
* Solution: Keep track of a “popularity score” for each course, which is (# of times the course has been liked) - (# of times the course has been disliked), and fill the user suggestions with the most popular courses. This means if a user’s liked courses have combined less than 10 edges, they will still get 10 good suggestions by also populating with the most popular courses.

**Getting Suggestions Out of the Box**

* Problem: The most important aspect of our system is the quality of the suggestions. The suggestions rely heavily on user relationships. However, if we have no users in the system, the initial suggestions would be… terrible.
* Solution: We worked really hard to get data from as many people as possible from as diverse a group as possible. We asked them to fill a google sheets of all the information we would ask for in the actual website. This of course took lots of preplanning.
  + We also had to then load this CSV of data into python and populate our database with this data.
* Caveats: While we tried to get a diverse population of students, most of them ended up liking STEM heavy courses. So in the initial days of our application’s deployment, users will mainly get STEM recommendations. So, we just need more humanities to use our system, and hopefully they get helpful suggestions!