Team Information

Project Name: US Colleges & Universities Internet Database
Canvas Group: E3

Team Members:

- Harrison Berrier
    - EID: hlb962
    - Email: harrison.berrier@utexas.edu
    - Github username: harrisonberrier
    - Estimated completion time for each member: 10
    - Actual completion time for each member: 9
- Mohit Gupta
    - EID: mg58629
    - Email: mohit.gupta@utexas.edu
    - Github username: mohitg17
    - Estimated completion time for each member: 8
    - Actual completion time for each member: 10
- Nikhil Jalla
    - EID: nj5473
    - Email: nikhiljalla17@utexas.edu
    - Github username: nikhiljalla17
    - Estimated completion time for each member: 11
    - Actual completion time for each member: 12
- Silas Strawn
    - EID: scs3434
    - Email: strawnsc@gmail.com
    - Github username: StrawnSC
    - Estimated completion time: 6
    - Actual completion time: 7

Github Repo Link: https://github.com/UT-SWLab/TeamE3
Deployed Site: https://university-idb.uc.r.appspot.com/

Motivation and Users

We envision our site being used by prospective students to assess which college or university is right for them, as well as which major or field of study they should pursue. Our database lets students view universities in the US filtered by their city, or by the majors they offer. For example, if a student knows they are only interested in schools that offer a particular program, they can go to the major page for that field of study, and browse the colleges that offer that

major. On the other hand, if the student is interested in colleges only in a particular location, they can pull up all the colleges for a particular city. Once they're on a university or college's page, the prospective student will be able to see crucial information for making their choice. For instance, they will see facts on the cost of attendance, the acceptance rate, average SAT scores, the size of the school, etc. If they want to learn more, they can also follow a link to the institution's web page.

Requirements

User stories

Phase I:

- As a high school student, I want to be able to look through a list of universities in the United States so that I can decide where I want to go to college.

  We estimated that this task would take 2.5 hours. We were able to achieve this by creating our model.html template and populating our database with university information from the CollegeScorecard API. The task took 2.5 hours to complete.

- As a high school student that wants to move away from home, I want to be able to look at university education statistics for different cities so that I can decide where I want to move.

  We estimated that this task would take 2 hours. We were able to achieve this by creating our model.html template and populating our database with city information from Wolfram API and through web scraping. The task took 3 hours to complete.

- As a high school student that hasn't decided what I want to major in, I want to be able to look through a list of different majors and their associated statistics so that I can decide what I want to major in.

  We estimated that this task would take 2.5 hours. We were able to achieve this by creating our model.html template and populating our database with information about college majors from the Department of Education API. The task took 3 hours to complete.

- As a user, I want to read about the website that I am using so that I know where my information is coming from and what the creator's motivation is.

  We estimated that this task would take 2 hours. We were able to achieve this by creating an About page on our website and populating it with information about the website, the tools used, and the creators. The task took 3 hours to complete.

- As a user that isn’t very good with technology, I want the website’s interface to be simple and navigation to be easy so that I can access the information that I want quickly and easily.

  We estimated that this task would take 2.5 hours. We were able to achieve this by creating a simple splash page and nav bar that allowed easy navigation across the site and immediate access to any of the three models. The task took 3 hours to complete.

Phase II:

- As a user with limited time and a general idea of what I'm looking for in a university, I want to be able to view some attributes of a university without having to navigate to that university's instance page.

  We estimated that this task would take 4 hours. We were able to achieve this by adding the state, cost of attendance, and student population for each university to the universities base page. The task took 4 hours to complete.

- As a high-school student, I want to be able to view a list of all colleges offering a 4-year degree in the US so that I don't have to go elsewhere to find statistics on certain schools.

  We estimated that this task would take 4 hours. We were able to achieve this by only collecting data for colleges offering a 4-year degree in the US from the CollegeScorecard API. The task took 3.5 hours to complete.

- As a user with limited time, I want to be able to navigate by more than one page at a time so that I don't have to waste time clicking through pages of instances.

  We estimated that this task would take 2 hours. We were able to achieve this by adding multiple page options to our pagination links beyond just next and previous buttons. The task took 2.5 hours to complete.

- As a user looking through a long list of cities, universities, and majors, I want the list to load into pages rather than one large screen so that it makes it easier to navigate.

  We estimated that this task would take 2.5 hours. We were able to achieve this using flask-pagination and by rendering only a subset of instances on each page. The task took 3 hours to complete.

- As a user that is scrolling quickly through a long list of universities, I want each university to be listed with a picture to help me sort quickly through them.

We estimated that this task would take 3 hours. We were able to achieve this by scraping images of universities from Google using the Google Image Search API and adding the images to our database to be displayed with each university instance. The task took 5 hours to complete.

- As a user that is scrolling quickly through a lot of cities, I want each city to be listed with an image to help me sort quickly through them.

  We estimated that this task would take 3 hours. We were able to achieve this by scraping satellite images of cities using the Google Maps API and storing the images in our database to be displayed with each city instance. The task took 5 hours to complete.

- As a user that is scrolling quickly through a long list of majors, I want each major to be listed with an image to help me sort through them quickly.

  We estimated that this task would take 3 hours. We were able to achieve this by scraping images of majors from Google using the Google Image Search API and adding the images to our database to be displayed with each major instance. The task took 2.5 hours to complete.

Phase III:

- As a user looking at a city's page, I want to be able to see a list of the universities within that city so that I don't have to navigate back to the universities page and filter by state.

  We estimated that this task would take 2 hours. We were able to achieve this by adding links to universities in a given city on each city instance page. The task took 2.5 hours to complete.

- As a user that is interested in exploring the universities of a city, I want to see a map on the city's instance page showing me where the universities are located.

  We estimated that this task would take 3.5 hours. We were able to achieve this by adding interactive maps to our city instance page using the embedded Google Maps tool. The task took 3.5 hours to complete.

- As a user, I want to be able to sort universities by acceptance rate so that I can see which universities are considered to be the best.

  We estimated that this task would take 3 hours. We were able to achieve this by adding sorting by acceptance rate to our universities base page. The task took 2.5 hours to complete.

- As a prospective student seeking to compare different cities, I want to see how a particular city's median age and rent compares to those of other cities through some visual representation.

  We estimated that this task would take 2.5 hours. We were able to achieve this by adding progress bars for median age and rent to our city instance pages. The task took 2.5 hours to complete.

- As a high school student who knows what school I'm interested in, I want to be able to search for a specific university so that I can view their statistics.

  We estimated that this task would take 2 hours. We were able to achieve this by adding a search bar to our universities base page. The task took 3 hours to complete.

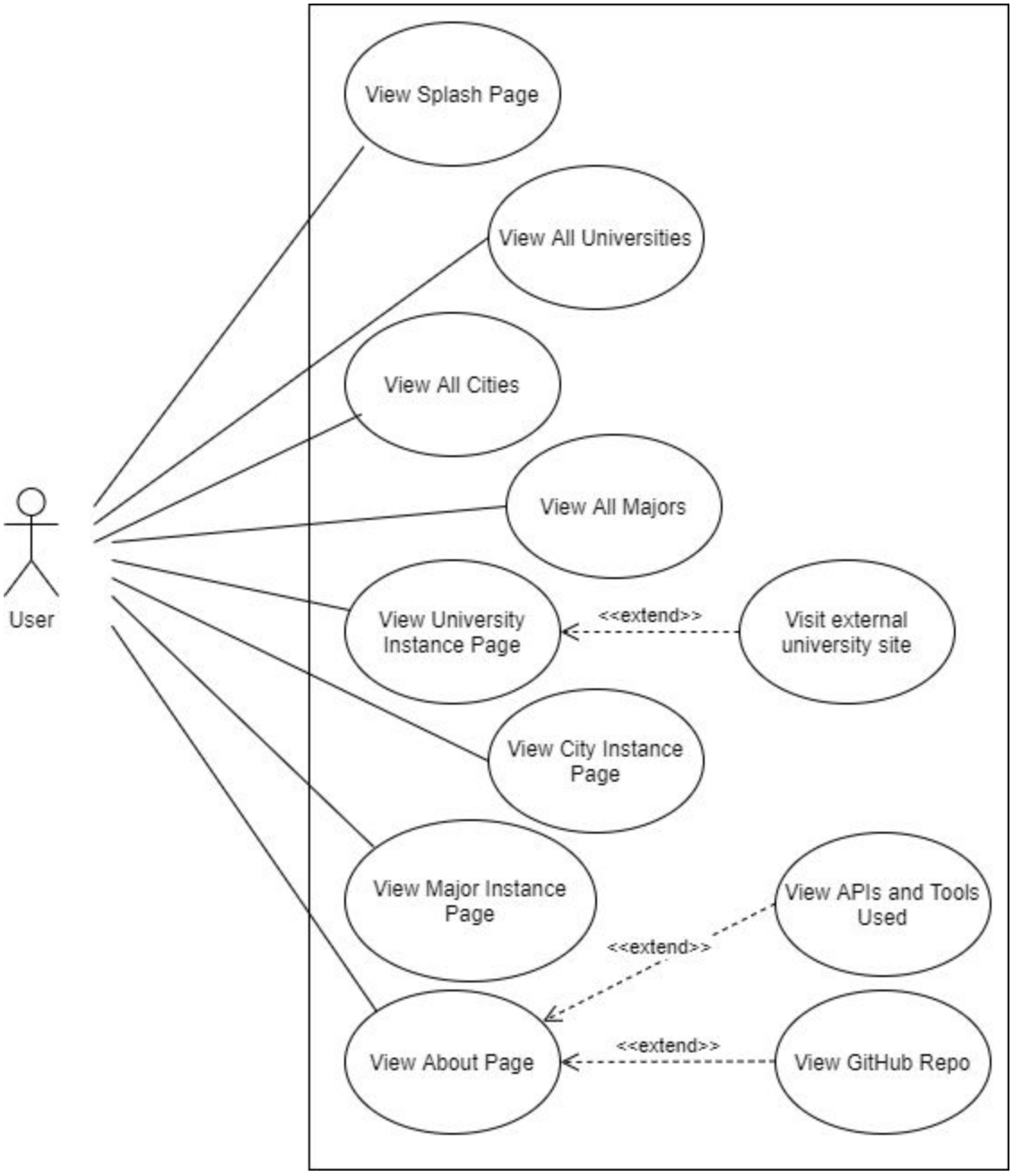
- As a prospective student, I want to be able to search for cities that I am interested in so that I can easily access more details about those cities.

  We estimated that this task would take 3 hours. We were able to achieve this by adding a search bar to our cities base page. The task took 3 hours to complete.

- As a concerned parent of a high school student, I want to see which universities are in my home state so that I can keep my child close.

  We estimated that this task would take 2.5 hours. We were able to achieve this by adding filtering by state to our universities base page. The task took 3.5 hours to complete.

Use Case Diagram



Design

Stack

We are using MongoDB (through MongoEngine), Flask, Jinja, and Bootstrap.

Jinja Templates

Every page inherits the base.html file, which includes the navbar. Each of the model base pages uses the model.html. Each instance uses the [model]_instance.html file. The landing page (index.html) is a central page where users can navigate to any of the model's base page. It contains a carousel and pictures that link to the models. The about page (about.html) contains information about the site and the developers.

Flask Routing

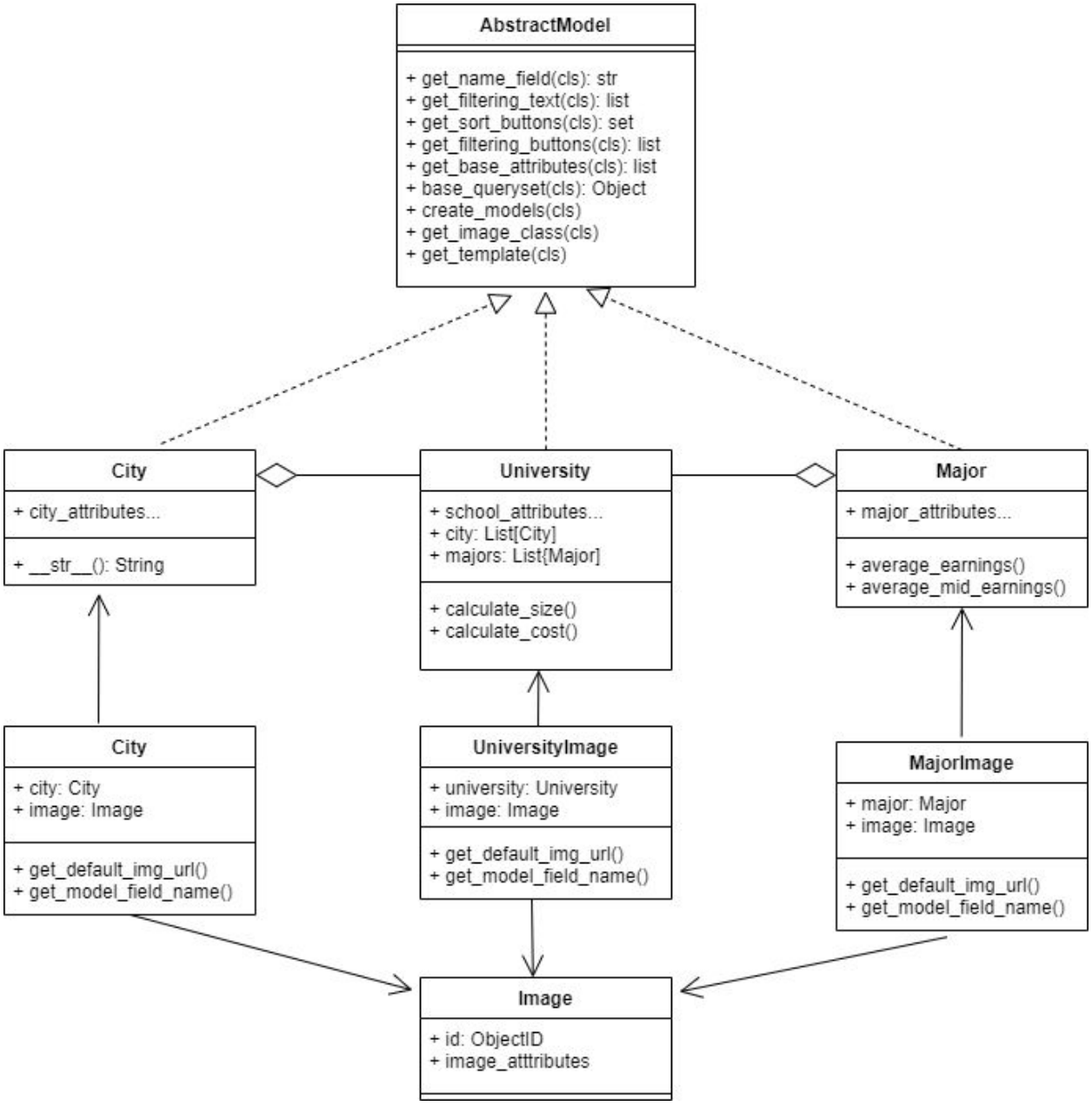
The flask application queries the database to retrieve the requested data. It retrieves data based on the type of requested data and the pagination settings. We used Jinja templates to design our pages and we render the templates with the requested data from flask.
The home page is at the root path ('/'). Each model is at the path '/base/[model]'. Each instance is at the path '/instance/[model]/[instance_id]'.

Database Design

A key requirement for our design, as shown in our Use Case Diagram, was to provide users with separate lists of universities, cities, and majors. We designed our database around three main collections - University, City, and Major - to achieve this goal. Keeping in mind that our website was centered around universities, we decided to also include references to relevant cities and majors for each university instance. Thus, we arrived at a design with three independent collections where each university instance contains references to related city and major instances. This structure is displayed in our database UML diagram.

We added collections for images of each model to facilitate ingestion of images. We were able to create one collection to store all of the image files. Each image instance contains a reference to the model instance with which it is associated, and a reference to the image file.

While refactoring, we created an abstract base class for the University, City, and Major models that contained function prototypes for the common functions between each model. Each model inherits this abstract base class and implements the prototyped functions.



Frontend Design

As shown in our Use Case Diagram, we wanted our website to give users access to three main things: 1) lists of universities, cities, and college majors in the US, 2) information about specific universities, cities, and college majors, and 3) information about the website and its creators.

1. Lists of universities, cities, and college majors

We accomplished our first goal through our model pages for universities, cities, and majors. We designed each model page to provide access to all of the instances in our database for each model. We incorporated pagination, searching, sorting, and filtering on our model page to enable users to easily find a specific instance or a particular type of instance. The pagination

allows users to easily navigate between all of the instances. The sorting feature allows users to view the instances based on criteria that is most important to them. Filtering enables users to refine their search based on certain attributes. Searching allows users to quickly find a specific instance. We provide a preview of three attributes for each instance to give users a high-level overview of the instance, and we enable users to access more information about an instance simply by clicking on its card. The model pages can be navigated to from the home page or from the navbar from any location on the site.

2. Information about specific universities, cities, and college majors
We accomplished our second goal by designing informative and simple instance pages that highlight key information and provide access to related pages. Each instance page follows a similar visual format, with a jumbotron at the top, a left panel for navigation or basic data, and a main panel for presentation of multimedia and additional data.

The university instance pages are rich with data that would be relevant to a prospective college student. Due to the abundance of data available on the page, the left panel is used for navigation between the sections. In the main panel, users have access to key statistics regarding cost and attendance, location, majors, demographics, and graduation statistics. The main panel also includes an image of the university. Key features on the university instance pages are links to the instance page for the city in which the university is located, and links to instance pages for majors offered at that university. These links are critical to providing users a cohesive experience on our website.

The city instance pages offer relevant information about cities to college students. The left panel contains median age and rent statistics, along with community type and a satellite image of the city to provide a high-level overview of the city. The main panel includes an interactive map of the city, more statistics, and links to majors offered by universities in that city, and links to universities located in that city.

The major instance pages offer key information about various college majors. The left panel contains links to related majors that could facilitate a user's search for the perfect major. The main panel contains a description of the major that gives users a high level overview of the major. These pages also contain an image related to the major, along with links to universities offering that major and cities in which universities are located.

3. Information about the website and its creators
Finally, we accomplished our third goal by creating an About page that details our motivation for developing the site, external links to the tools we used, a link to our Github repository, and information about each of the creators.

Testing

GUI Testing

Phase II

Our GUI testing uses the Selenium IDE to reduce the potential points of error. We test all of the links on the navbar by clicking them and asserting the title of the page. We tested the navbar to ensure that each link routes to the correct model page. The model page links are then tested in the same way. We tested the model page links to ensure that each instance routes to the correct instance page.

Each page's pagination is tested as well. The test makes sure that both navigation arrows work as well as the early and late page navigation. To test the page navigation, the script checks the text of the active tag, confirming that the page changed. It also checks to make sure that the first page and last page elements exist, regardless of which page is active. We tested our pagination links to ensure that the instances are correctly divided into pages and to check that all of the pagination links are functional.

Phase III

To test our searching functionality we test each model individually. For the universities we test that a school has been searched for correctly by selecting the search bar, entering the text for the school, clicking the search button, and asserting that the school's id is present on the page. We test this against a one-word search, a search from the resulting route (which is different from the base model route), and a multi-word search input. For the last search we navigate to the instance page and assert the header text to make sure that the page link is correct. Cities and majors are tested similarly, with a one-word search input, a search from the resulting page, and a multi-word input. We again navigate to the instance page of the last search result and assert the header text to make sure that the links are connecting to the correct instance pages. The search function is also tested for an incomplete search (for ex. searching "bos" and returning Boston, Massachusetts). We tested our searching functionality to ensure that users are presented with relevant search results when the search is used.

Unit Testing

Phase II

We used python unittest to test our database queries and data processing. We tested create, read, update, delete, and search operations for each of our main models to ensure that our database read/write operations were all working as expected. To test the create operation, we create an instance of each model using dummy parameters and save it to the database. We then queried for that instance and checked if the instance was found. We tested the create operation to confirm that our instances were being properly create in our database. To test update operations, we loaded an instance from the database, modified one of its properties, and

saved the object to the database. We then loaded that same object again and checked to confirm that the property was modified as expected. We tested update operations to confirm that our instances were being properly updated in our database. To test read operations, we queried the database for specific instances and compared the properties of the retrieved instance with the expected data.We tested read operations to confirm that our instances were being properly queried from our database. To test delete operations, we created a dummy object instance and saved it to the database, and then queried that object and deleted it. We then queried the database for the deleted object and confirmed that nothing was returned. We tested delete operations to confirm that our instances were being properly deleted in our database. Finally, to test searching, we queried the database with specific search parameters and compared the instances that were returned with the instances that were expected. We tested search operations to confirm that our instances were returning proper instances based on search criteria.

Phase III

We tested database queries more extensively because we rely on the data that we pull from the database to be exactly as expected. We tested queries for the university, major, and city objects, and compared the returned objects with the expected object. The objects are queried using varying combinations of their unique fields to ensure that all combinations work.

We also manipulate some of the data that we retrieve from the database before sending to our frontend. We tested our data manipulation to ensure that our modifications made the changes that we were expecting. The data processing test queries the database and mimics the data manipulation that we execute in our main file. The modified sample instance data is compared to the expected data for a particular instance.

These tests together validate that the database is returning the expected objects when queried and that the expected data is being sent to the model and instance pages.

Models

University Model

We collected data, including images, for over 2,000 universities, all of which is stored in our database and accessible via our website. Each university instance contains relevant statistics about the university along with its location and the majors that it offers. The page has an overview at the top that includes the school size, acceptance rate, and in-state tuition. Below that, there are sections for school info, majors, admissions, demographics, and tuition and aid. The left panel has links that navigate to each section. The right panel includes extra data about degrees awarded, completion rate, average earnings for graduates, and overall retention rate. All of the data is pulled from the CollegeScorecard API. The page showing all universities includes a picture of each university on its corresponding card, and each individual university page includes a picture of the city that it is located in.

We would like to present some of the data graphically to make the page more visually appealing. Demographics data can be easily represented in a pie chart. More data about admissions and completion rates can also be displayed in intuitive ways. We can also make the pages more interactive for the user by adding more responsive elements such as collapsibles and buttons. We also need to link the university instances to the city and major instances.

City Model

Given the number of university instances in our database, we ingested data and images for over one thousand cities. As with universities, all of these cities are viewable on our site. Each city instance is described by statistics that would be relevant to students who would like to attend a school in the respective city. Currently, the page lists the area of the city, population, population density, community type, and of course the schools that are in the city. Population data, city area, and median age were found using Wolfram Alpha's Full Response API. Rent data was a little harder to find, so it was scraped from a website called city-data.com. The schools in each city are listed based on the location data found in each university.

The city page also displays a satellite image of the city which was found by using Google Maps API. The page displays both a map view and satellite view.

Major Model

Our database holds data associated with 357 separate majors and images for every major. Like the other models, all of these majors can currently be viewed on our site. Each major instance page contains a list of universities that offer that major, as well as some aggregated statistics on the number of programs in the US for that field of study. We previously listed just the number of programs that offer a bachelor's degree, an associate's degree, and certificate (after less than one year of study) in the given major. However, since we decided to only list colleges and universities that offer a bachelor's degree as their "predominant" degree awarded, we now just display the number of schools that offer a bachelor's degree in a given major.

We also display the average starting salary for each major, which is based on the latest available earnings cohort of graduates who received a bachelor's degree in a given field. Since this data was collected at the university level, we had to aggregate it across all two thousand schools to get an average earnings figure by major. All of this data was acquired from the Department of Education's College Scorecard API. We also wanted to present a mid-career salary statistic. Since the Department of Education's API does not have this information, we approximate the mid-career salary as 1.693 times the starting salary, based on the Wall Street Journal's data on the average increase between early- and mid-career salaries. In future phases, we will seek a more accurate representation of mid-career salary, as we wouldn't expect the percent change to be exactly the same for every major.

Ideally, we would like to sort the universities in each major instance's list by the ranking for that particular major, but we have not found open datasets with such information. In future phases,

we will explore the possibility of scraping data from the US News & World Report site, although it seems most of their data is behind a paywall. For now, we simply display an unsorted, paginated list of all the universities that offer a particular major on that major's instance page. This list is populated by a MongoEngine query on the University collection that selects only schools where the current major is in that school's list of offered majors.

Finally, we also group majors by high-level categories. Given that we have several hundred majors, they tend to be narrow fields of study. It may be useful for the user to be able to look up schools that offer majors in"engineering" (a broad category) rather than "naval architecture and marine engineering" (one of our current majors).

Tools, Software, and Frameworks

Flask

Our backend framework is flask, a microservices framework for python. Flask is useful in that it is lightweight: it was trivially simple to set up the server and start running it locally. When we want to add new pages, we simply add a route for the new page. We also took advantage of Flask's template mechanics. Using flask templates, we dynamically generate instance pages for our major, university, and city models. This allows us to reuse the same HTML, instead of adding hundreds of nearly identical HTML files for each university.

Our use of flask only changed slightly through each phase. We incorporated flask_paginate in Phase 2 to handle pagination. Instead of using the model names in the routes, we changed the routes to use the MongoDB object IDs in Phase 3 to avoid routing issues associated with using instance names. We significantly reduced the amount of code used to control our Flask application by creating standard route formats for our base and instance pages in Phase 4. We also moved our model-specific data manipulation code from main.py to each of the model classes.

Bootstrap

Our frontend framework is bootstrap, which we use alongside HTML and CSS to design our web pages. Bootstrap has been crucial for allowing our pages to be reactive. The pages in our site dynamically resize based on the size of the viewport, allowing our format to remain robust and look clean, despite the window size limitations of the user.

Our use of Bootstrap was unchanged through each phase. We used bootstrap4 styling from the beginning and continued to use bootstrap as we added more elements to our pages in each phase.

Google Cloud Platform

Our site is deployed to Google Cloud Platform. We used GCP because it is simple to set up and use with flask applications. All we have to do to update the deployment of our application is run the shell command “gcloud app deploy” in the project root directory, using the gcloud CLI.

Our use of GCP did not change significantly between the phases. We enabled some Google APIs on our project that allowed us to utilize Google Maps and Google Images Search during Phase 2 and Phase 3. We also slightly altered the way our application is launched during Phase 2.

MongoDB and MongoEngine

All of our data is dynamically pulled from MongoDB, including the images for every model instance. Our models are defined with MongoEngine, an open-source Python Object-Document Mapper. MongoEngine will allow us to define “schemas” (MongoDB does not technically have schemas since it’s NoSQL) with python objects, so we can interact with our database purely in terms of University, Major, and City python objects that we design. MongoEngine ODM capability is useful for not just defining the schema for our model’s fields, but also enforcing constraints on certain fields, such as uniqueness constraints. For instance, MongoEngine forces our model instances to have university names to be unique with respect to the university’s city and state.

Our use of MongoDB and MongoEngine evolved over the course of our project. We started initially with collections for universities, cities, and majors. We added collections for images for each model in Phase 2. We changed the size of images that we stored in our database to meet storage restrictions. We added and manipulated data in our database in each phase.

We implemented MongoEngine early on and utilized it heavily in Phases 2 and 3. We created model classes for each of our collections and used the models to enforce requirements for each instance. We added new fields to our models as we collected more data. In Phase 2, we used MongoEngine queries to easily update instances as we collected more data. We used MongoEngine querying extensively in Phase 3 to implement searching and filtering.

Google-Images-Search

We used the Google-Images-Search python module (available on PyPi) in order to automatically collect large amounts of images. Since we have about two thousand universities, one thousand cities, and several hundred majors in our database (and accessible via our site), manually collecting images for each model instance would have simply been intractable. The Google-Images-Search module allows the user to specify a query (and some other attributes like the kind of image and the image’s licensing) and download one or more images using Google’s image search. We wrote scripts to search for universities by their name, download and resize the first image result, then upload the image to a new collection that maps the university instance in MongoDB to its corresponding image. We did the same to populate major images.

Our use of Google-Images-Search did not change between phases. We wrote data ingestion scripts in Phase 2 and Phase 3 to retrieve images from google using a standard method.

Python Google Maps Package

For city images, we decided to use satellite imagery, since there were a lot of open data sets available online (and we thought they would be visually interesting). We explored NASA and Google Earth APIs, but decided to use the googlemaps python package, (also available on PyPi). Although we did end up scraping latitude and longitude coordinates for each city, the google maps module allowed us to simply look up the location by city name and download a corresponding satellite image.

Our use of the Python Google Maps Package did not change between phases. We used it to ingest images of cities during Phase 2.

Sources

Flask documentation: https://flask.palletsprojects.com/en/1.1.x/quickstart/
We used the flask documentation to learn how to use routes and to help us format our URLs.

MongoEngine: http://docs.mongoengine.org/
This MongoEngine documentation helped us understand how to set up our model schemas, connect to the remote database, and query the database.

Bootstrap reference guide: https://www.w3schools.com/bootstrap4/default.asp
https://getbootstrap.com/docs/4.0/components/carousel/
We used the Bootstrap reference guide to help us understand how to use and format Bootstrap elements such as cards and the carousel.

Footer guide: https://www.w3schools.com/howto/howto_css_fixed_footer.asp
We used the footer guide to learn how to format a page footer.

Forms guide:
https://developer.mozilla.org/en-US/docs/Learn/Forms/Sending_and_retrieving_form_data
We used this as a guide for developing the search functionality

CSS variables: https://www.w3schools.com/css/css3_variables.asp
We used this CSS variables reference to help us format our elements and make them more visually appealing by changing font, color, size, etc.

Major salary data: https://www.visualcapitalist.com/visualizing-salaries-college-degrees/;
http://online.wsj.com/public/resources/documents/info-Degrees_that_Pay_you_Back-sort.html

Google Maps API documentation:
https://developers.google.com/maps/documentation/maps-static/usage-and-billing
We used the Google Maps API documentation to figure out how to grab satellite images and automate the process for all of the cities.

College Scorecard: https://collegescorecard.ed.gov/data/documentation/
We used the College Scorecard API and data dictionary to collect data on universities.

Flask Pagination: https://gist.github.com/mozillazg/69fb40067ae6d80386e10e105e6803c9
We used this pagination example to implement pagination on our website using the flask_paginate module.

Google-Images-Search: https://pypi.org/project/Google-Images-Search/
We used this python package and it's documentation to ingest images for each university instance and major instance.

Google Maps Package: https://pypi.org/project/googlemaps/
The documentation for this google maps package allowed us to collect satellite images for each city in our database.

Beautiful Soup Documentation: https://www.crummy.com/software/BeautifulSoup/
Beautiful Soup was used to pull relevant data from websites if it could not be easily obtained through APIs.

Wolfram Alpha Full Results API: https://products.wolframalpha.com/api/documentation/
We used the Wolfram Alpha Full Results API to collect data on cities.

Reflection - Phase I

What we learned:

We've learned how to work together and divide work evenly among us. We were able to distribute tasks early on and each person completed their parts when convenient for them. We also learned how to build a website from ground up using html, css, bootstrap, and flask. Through designing the site, we've learned more about bootstrap and how to style components. We've also learned how to use the APIs selected to acquire the data that we need.

Five things that went well:

1. We quickly moved to the use of templates so that multiple pages can be changed quickly.

2. We have been able to make decisions quickly and then adapt as problems arise, given our heavy usage of slack to coordinate and communicate.
3. Our API's are well documented, making them easier to interact with. In particular, the Department of Education's College Scorecard API has extensive higher education data, and provides a spreadsheet explaining all the possible fields one can request from the API.
4. Since flask is a lightweight backend, we were able to get started putting pages together and making routes for them easily.
5. Flask also allows us to pass in parameters to the html, and we used this to dynamically generate different instance pages off of the same base html. Although the data is currently hard-coded in the python flask app, we will be able to easily migrate this over to database queries without having to change the frontend code.

Five things that went poorly:

1. We haven't been able to add a stylesheet other than bootstrap so we are stuck writing styling into each file at the moment.
2. Formatting pages to fit requirements has been very difficult versus formatting a page without specifications because of the way that html elements and they're styling interact with each other.
3. At first, the college scorecard API was overwhelming because of the sheer amount of data and the complexity of crafting queries to get the needed information.
4. Often, the styling of elements with HTML, CSS, and Bootstrap does not work as we would expect. For example, when we were making the about page, we had to put in elements with our photos and bios together. For some reason, adding more text to the paragraph element next to an image would increase the image's size. We ended up using different bootstrap elements to get around this.
5. When we were first setting up the deployment on Google Cloud Platform, the deploy would seem to work in the CLI, but when we tried to visit the site, we would get a 500 error. Apparently, if you don't have the exact right directory structure and naming of certain modules, GCP doesn't know how to deploy your flask app.

Reflection - Phase II

What we learned:

We learned how to set up, access, update, and query a MongoDB database. To set up the database, we learned how to create collections and set up data models. We learned how to use a python object-document mapper, MongoEngine, to manipulate data in the database. To populate the database, we learned how to write ingestion scripts that call APIs to collect data and store them in the database. We also learned how to test both our frontend and backend code using Selenium and unittest respectively. To collect images, we learned how to use

existing APIs to automatically retrieve relevant images. We learned how to use pagination to limit the number of instances that are loaded at a time.

Five things that went well:

6. We started testing both our frontend and backend early on, which helped us resolve bugs much quicker than before. We used Selenium to confirm our frontend functionality and we used unittest to validate our database queries and data manipulation.
7. We were able to easily communicate with the database using python objects through MongoEngine. The object-based model allowed for logical data access and manipulation. It also allowed for greater flexibility in the structure of the data.
8. Since we used templates for phase I, we were able to transition from hard-coded data to dynamically accessed data very easily. Our data requires very little backend processing once it is received from the database, so large amounts of data can be aesthetically displayed in the model and instance pages. Data access is also very fast.
9. We wrote ingestion scripts to consume data from our APIs and populate our database very efficiently. After running our ingestion script initially, we were able to easily adjust the data and fields as needed.
10. We were able to divide responsibilities efficiently so that each person's work was not entirely dependent on another person's work. This allowed us to work on our own schedules and make good progress on all components of the project simultaneously.

Five things that went poorly:

6. Currently, all of the instances are loaded upfront when a model page is accessed, so there is a short loading time on the page. We will need to dynamically load instances for a given page in order to decrease loading time. We should also add indexes to our MongoDB collections to make queries faster.
7. We used a python package that uses the google images API to get images for our instances. While it mostly worked very well, we need to go through and replace images that are not accurate.
8. We needed to adjust our data models slightly after we had already run our data ingestion scripts, so we had to re-run some of those scripts with the updated data model. This was time consuming since we had to query an API for some of the new data.
9. After integrating the database code into the main app script, we weren't able to deploy our app to Google in its existing form. We had to make changes to the deployment process to make it possible to deploy to Google App Engine.
10. Some of the APIs we had planned to use were hard to work with so we had to find alternative APIs with more easily accessible data.

Reflection - Phase III

What we learned:

We learned the value of refactoring code. Up until this phase we had a significant amount of repeated functionality because of differences between our three models. For example, rendering each model page has a lot of similar code, but there are differences between the models that kept us from consolidating the functionality. During this phase, when implementing searching, we had to render a slightly different version of those model pages so we took the time to break the common functionality into separate methods which has improved the readability of our code and reduced the number of lines of code. We have also started to utilize comments more because, until now, the code had grown in an organic way while the whole team worked on it, so we all understood it. Our refactor moved significant portions of the functionality and comments helped to keep us all on the same page.

Five things that went well:

11. Using mongoengine made it much easier to query our database and helped with searching and sorting. Where we might've had to write our own searching and sorting functionality for a large data structure, we were able to rely on a well-tested framework.
12. The refactoring from this phase should improve our ability to make significant changes in the structure of our code. Changes that previously would've taken too much time to make will now be more manageable because of increased modularity.
13. We were able to ingest additional data for our city and major models, which allowed us to make the instance pages more rich with information and offer more insight to the user.
14. Since most of the data had already been ingested in prior phases, we could focus on the coding the site's searching, sorting, and filtering.
15. Even though we have thousands of university model instances, the searching, sorting, and filtering runs fast.

Five things that went poorly:

11. Because of the way that we render instances on the model base pages and because we're not using a react or angular framework, we have to redirect to load search results rather than being able to dynamically load them into the current page.
12. To split up the work, searching and sorting were assigned to different people so they have not been integrated with each other and will likely need to be refactored in the future.
13. Some of the APIs and data sources that we used did not have consistently formatted data, which made it difficult for us to write ingestion scripts to add additional properties to our model.
14. A true search engine in our site would be able to do fuzzy string matching. To be fair, our search is much better than requiring exact matches; it allows the user to match by a case-insensitive substring with whitespace trimmed. For instance searching for "harv "

matches “Harvard University” as well as “Harvey Mudd College.” Our search can’t do approximate string matches, like matching “Harvard University” from a slight misspelling, such as “Harverd.”

15. Over time, our main flask file has grown quite large, and is occasionally repetitive. We should have put more thought into how we could have broken up the file into separate modules.

Reflection - Phase IV

What we learned:

Through refactoring and applying design patterns to our project, we learned how to use abstractions and interfaces to reduce redundancy and improve readability. At the beginning of this phase, our main file had the vast majority of our python code, and many methods were very similar, with only minor variations. We learned the disadvantages of having such a large main file, which include poor readability and poor adaptability. We learned how to consolidate methods with slight variation into a single method to greatly reduce redundancy. We also learned how to identify opportunities to apply design patterns. While refactoring and implementing design patterns, we learned how to write code that depended on abstractions instead of concrete classes.

Five things that went well:

1. Our refactoring of our model page route allowed us to consolidate 3 routes into one (3 different times) and eliminated a considerable amount of redundant code.
2. We were able to switch from using instance names to using object IDs while refactoring our instance routes, so we no longer need to worry about URL encoding.
3. Through refactoring we reduced the length of our main file from 400+ lines to ~100 lines.
4. By using flask’s redirect function, we could have a seamless transition to the new route functions. We used redirects from the old routes to the new routes so that we could use the new routes before we went through the front end code updating to the new URLs.
5. All our model classes subclass mongoengine’s “Document” class, allowing us to integrate with MongoDB. Thankfully, python allows multiple inheritance, so we can have our model classes subclass an “AbstractModel” class which defines the model interface used by the main file. Using this interface, the main file can treat every model the same, reducing its complexity.

Five things that went poorly:

1. In order to implement the Factory Method pattern, we had to create 8 additional classes and introduce new dependencies, which increased the complexity of our code.

2. We had to use some method level imports to avoid circular dependencies caused by making our model classes and image classes both depend on each other.
3. While refactoring our main file considerably reduced its complexity, we ended up pushing a lot of logic that was in main into the model classes. Although it makes more sense for model-specific logic to be in the model classes, this code created high coupling between the model and the jinja model templates.
4. It was difficult to think of what design patterns made sense to introduce into our web app.
5. We originally intended to migrate the model image classes to a field on the models themselves. Even though it would have reduced the complexity of our app, we didn't take the time to do this, since the app still works well with the images in separate collections.