Group 01 Phase 2 Technical Report

Kevin Xu, Josh Yu, Ben Yu, Bryan Lee, Tyler Kubecka

Overview

In this report, we will be detailing our BrighterBeginnings project to provide a means of navigation through the repository and understanding of the architectures. This involves explaining the structure, testing, and hosting for the front-end and back-end.

Purpose

We deem this website necessary as we believe that financial constraints should never hinder the pursuit of knowledge. BrighterBeginnings aims to empower low-income K-12 students to reach for the stars by connecting them with knowledge about scholarship opportunities and organizations providing aid, whether financial or otherwise.

Toolchains

The tools we used to create our web application are as follows:

- Development
    - Gitlab
    - Docker
    - Visual Studio Code
- Front-end Tools
    - React
    - Bootstrap
    - Node
- Back-end Tools
    - Flask
    - SQLAlchemy
    - Gunicorn
    - MySQL Workbench
- Hosting
    - AWS Amplify - Front-end
    - AWS EC2 - Back-end

    - AWS RDS - MySQL Database
    - Namecheap - Domain Name Registration
- Testing
    - Jest - Front-end
    - Selenium - Front-end
    - Unittest - Back-end
    - Postman - Back-end

Front-end Architecture

The front end is a React app, which you can build and run the app like this:

```
(in cs373-group-01/front-end)
      npm install
      npm run start
```

If running locally, it will be at `localhost:3000` by default.

Structure

The structure of the website currently is as follows:

- A home page
    - Relevant files:
        - src/pages/Home.jsx
        - src/components/cards/ExploreCard.jsx
- An about page (/about)
    - Pages for each developer on click
    - Relevant files:
        - src/pages/About.jsx
        - src/pages/AboutSubPage.jsx
        - src/data/about.js
        - src/components/cards/StatsCard.jsx
        - src/components/cards/ToolsCard.jsx
- Cities page (/cities)
    - Pages for each city on click

        - Contains images and info about each city
        - Links to organizations with help resources in those cities
    - Relevant files:
        - src/pages/Cities.jsx
        - src/pages/CitySubPage.jsx
        - src/data/cities.js
        - src/components/cards/CitiesCard.jsx
- Organizations page (/organizations)
    - Pages for each organization on click
        - Contains images and info for each organization
        - Links to scholarships and cities that these organizations are based in
    - Relevant files:
        - src/pages/Organizations.jsx
        - src/pages/OrganizationSubPage.jsx
        - src/data/organizations.js
        - src/components/cards/OrganizationsCard.jsx
- Scholarships page (/scholarships)
    - Pages for each scholarship on click
        - Contains images and info for each scholarship
        - Links to organizations that provide those scholarships if applicable
    - Relevant files:
        - src/pages/Scholarships.jsx
        - src/pages/ScholarshipSubPage.jsx
        - src/data/scholarships.js
        - src/components/cards/ScholarshipsCard.jsx
- Routing is done in src/App.js

Testing

Jest is used to test the rendering of React components as well as the proper text and links are rendered on those components. It allows for us to render the components in isolated environments outside of the context of the whole website.

Selenium is used for the acceptance testing by going through the actual domain. It uses a headless browser (in our case Firefox) to click through all the elements on the page to make sure they route to the correct places. It also verifies the text content of each of the pages upon loading.

Hosting

For our front-end, we are hosting it on AWS Amplify, in which it automatically pulls from the Gitlab repository to update. The domain is https://www.brighterbeginnings.me. To configure the domain, all that is necessary is to navigate to the domain registrar and add the ALIAS and CNAME records provided on Amplify to the advanced DNS settings of the domain.

Back-end Architecture

Structure

Within our back-end folder, we have our Flask application saved as app.py. In app.py, we import the models.py and schema.py modules, which are used to define the models of our data instances and the SQLAlchemyAutoSchema classes for them, respectively. Within models.py, this is where we establish our database connection, define the table names and model attributes, as well as create the tables. Within schema.py, we define the schema classes for our models, which serve to transform complex data types into a format that can easily be serialized into JSON form. The dependencies needed to run the Flask app are found in the requirements.txt file, which our Dockerfille uses to install all the necessary packages. Also within our backend folder, we have two folders labeled data and data collection. The former contains csv files for all of our models with data on every instance in our database, whereas the latter contains the sources and web scrapers used to collect said data from third party APIs. We used pandas to transform data from API calls into our desired format.

API Documentation

Our API is able to get any single or all data instance(s) for all of our models, including all the fields displayed on our website. To learn more about our API, you can read about it on Postman.

Testing

Within our backend folder, we have a tests.py and postman_collection.json file that are used to test our API endpoints we defined in our app.py. To ensure that every request worked as intended by making sure that our get all methods returned the right number of instances and that our get specific instance methods returned the one and only correct instance.

Hosting

For our back-end, we are hosting it on AWS EC2, where we run our Flask app in a tmux session on the EC2 instance terminal. The domain is https://api.brighterbeginnings.me. To actually run the server, we use Gunicorn, a WSGI server, and enter the command ‘gunicorn -w 2 -b 0.0.0.0:8080 ‘app:app’. This specifies two worker processes to handle incoming requests, binds all available network interfaces (‘0.0.0.0’) on port 8080, and runs our app module.

Database Architecture

Structure

For our data connections, we made use of SQLAlchemy ORM (object-relational mapper). Within our MySQL database, we have a Scholarship, Organization, City, and an association table called Scholarship_City_Association. The latter exists to create a many-to-many connection between scholarships and cities, while the organization schema has a scholarship_id and scholarships field to create a connection between organizations and scholarships. To look more into the model schema, attributes, and connections, refer to the model.py and schema.py files.

Models

We have three models for scholarships, organizations, and cities with the following attributes:

- Scholarship - id, name, awarded_by, award_amount, merit_based, need_based, essay_based, nationwide, img_src, link, cities, organizations
- Organization - id, name, email, phone, organization_type, img_src, scholarship_id, scholarship
- City - id, name, population, state, median_income, unemployment_rate, college_educated_rate, poverty_rate, img_src, scholarships

Hosting

We currently host our database on AWS RDS, which is connected to an AWS EC2 instance. In terms of specs, our database runs on a MySQL Community engine, size db.t3.micro, and on the us-east-2a availability zone with an endpoint on MySQL’s default port, 3306. We also have a network load balancer set up for our EC2 instance (target group) in order to route traffic.

User Stories

From our collection of user stories, we have listened to feedback to implement a cleaner website design (decluttered unnecessary or repeated data fields), included mapped images to make locations clearer, and included links and pictures to supplement scholarship information.

We were asked to include more data fields on the models pages, so we added many more information points, especially on the organizations page. We were also asked to include a number of total instances per model page, so, using our API, we retrieved the number of data entries and displayed it at the bottom of our website, integrating it with our pagination components. We added a video on our splash page to make it more engaging, as requested. We are also working to make the video responsive in size so that the text written always positions itself in the same relative place on the video. Users were also interested in what each of our tools were used for, so we added a tooltip when the tools were hovered over using the react-bootstrap card component. We reduced the erroneous padding as well, which users were complaining about creating too much whitespace on the site. The site now has less whitespace without being overwhelming.

Challenges Faced

We ran into quite a few challenges in the planning phase because we had to ensure that everything was possible to be implemented by the deadline and that the data was findable. For the actual project, many challenges faced were React-specific and were resolved with the help of online forums where others faced similar challenges. We also ran into an issue with the CI/CD pipeline because we were not resolving warnings as he went, but we are now doing so.

During our front-end development, we ran into many problems regarding our routing. We eventually restructured our pages and subpage link generation and navigation bars to remedy this. Before, most of our links would sometimes lead to blank pages, or generate totally invalid links that would lead to blank pages. We also had trouble with setting up our instance pages in a nice way. Oftentimes, the data would be formatted weirdly, especially with some fields having blank data (for example, some organizations did not possess a directly contactable email). Another problem with our front-end was during testing, as the selenium library was initially difficult to figure out from reading just the documentation. We were having most of our issues with setting up a headless browser to work on to interact with the elements. We also found that certain commands would only work on certain browsers.

One of the main challenges of phase 2 was setting up hosting for our back-end. Taking into account that AWS is a very complex service, there are a multitude of steps to follow precisely and contexts to keep track of. To illustrate how convoluted the process is, some of the steps are creating an EC2 instance, configuring a Docker image, production deployment, creating load balancers, configuring security and target groups, network load balancer listeners, acquiring an ACM certification, etc. just to name a few. Debugging is also an extremely arduous endeavor; for example, we had an issue with an unhealthy AWS instance, which meant our load balancer could not access the test endpoint. Even though there is an error log, it did not give any information on the real bug, but it turned out that we needed to change the endpoint tested.

Another challenge was finding sufficient data from APIs and integrating all the disparate data. Throughout our scraping period, there were countless instances when the data we would retrieve from an API would have holes. Therefore, we had to look for other sources to scrape, leading us to have somewhat scattered data. Much of the data we collected did not follow mainstream standards, and so we had trouble formatting the instances to be correct. In our next phase, we will put emphasis on data refinement and find better ways to aggregate the results.

The last significant challenge, back-end wise, was getting through the learning curve of all the different technologies required to create, test, and deploy our Flask application. Looking through documentation for Flask querying, SQLAlchemy, marshmallow, unittest, Gunicorn, etc., as well as trying to understand how these tools all work together took a substantial amount of time. While using Docker to create environments that fulfilled our dependencies helped, even figuring out what libraries to use, which versions to use in order to avoid conflict, etc. was difficult.