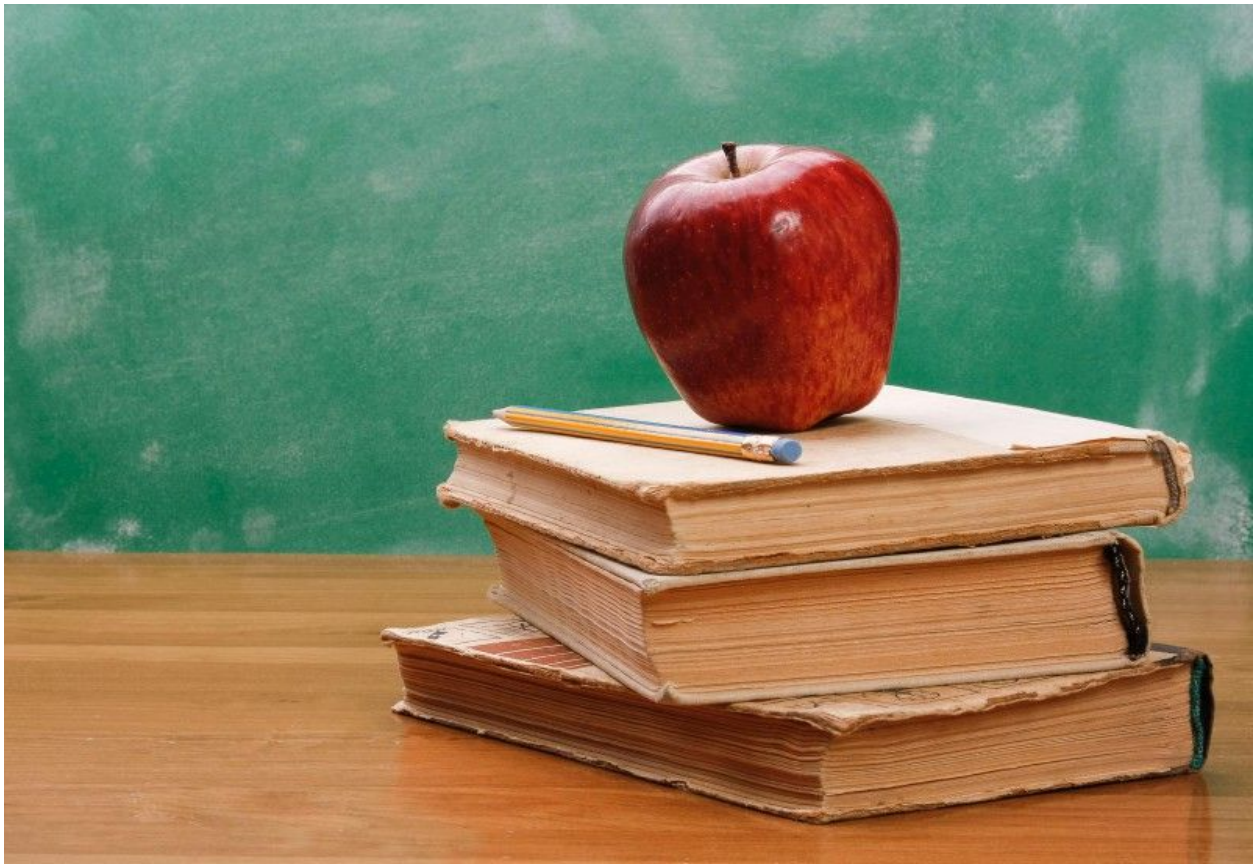
Equal Education Technical Report



CS373: Software Engineering Fall 2018

Authors:

- Kyle Sotengco
    - Front-end + Documentation
- Intae Ryoo
    - Front-end + Documentation
        - Phase 1 Leader
- Andrew Harmon
    - Back-end
        - Phase 3 Leader
- Jesus Palos
    - Front-end + API
        - Phase 2 Leader
- Prateek Kolhar
    - Back-end
        - Phase 4 Leader
- Ailyn Aguirre
    - Front-end

Motivation:

Our website is designed to help people find charitable opportunities to assist underprivileged children with their education. In today's society, a strong, reliant education is essential to living a successful life; yet there are many school districts in America that lack even basic school supplies. We wanted to showcase less fortunate areas in the US and allow users the opportunity to give their time or financial support to these communities. We hope that these contributions allow less privileged children have the same opportunities and benefits as more affluent areas.

User Stories:

Our core interest is to provide an extensive database of school districts in need of assistance. The website aggregates a large number of low-income areas and links them to charitable organizations and communities relevant to that area. Users can then either search for school districts within their area, search for specific communities in the United States to view their economic situation or donate to charity organizations that they trust. We believe that many people would be willing to donate their resources, but they might not know how or whom to trust. We want to showcase to our users just how many different, reliable ways there are to help out the community in regards to education.

Models:

Our three models are school districts, charitable organizations, and communities.

School Districts

The school district model consists of lower income school districts in America. Each school district will have its name, location, description, the percentage of population under poverty, total population, and communities that encompass the district. Users can sort by name of district, poverty level (using data from census.gov), and population size to determine which school is the best fit. There is also an option to filter districts by state, population size (small, medium, large), and poverty level so the user can easily find opportunities within their area. A graph showcasing the poverty levels of the district will be available on all model pages to help the user visualize the economic situation.

Charities

The charity model consists of several charitable organizations involved with education. Associated with each charity are its location (city, state), rating, their tagline and mission statement, the scope of the charity (region, national), their associated category, and a YouTube video related to the charity that showcases its purpose. The charities can be sorted the name of the company and their rating (as done by CharityNavigator.org), and filtered by scope, category, and state. We want to make sure users feel comfortable in giving their donations to a reputable cause. The user can also directly search for any specific organization they have in mind.

Communities

The communities model consists of a myriad of communities associated with a specific school district. Listed for each community is the location (state), counties within the community, a description of the community (brief summary, population details), and a map of the community via the Google Maps API. Communities can be sorted by name and population size and can be filtered by state and population size. We want to give our users a personal look into these communities so that they can assess how they can help out.

RESTful API:

Link: https://documenter.getpostman.com/view/5488016/RWgm41VA

The mockups for the models are built using Postman Mock server. We built 4 endpoints for each model based on i)page ii)id iii) links to the other entities:

We have three models:

1. Charities:

Sl no.	Name	Type	Description
1	id	int	Identifier
2	name	string	Name of the charity
3	tagline	sting	The short tagline of the Charity
4	state	string	State where the charity operates in (foreign key shared across all models)
5	scope	string	Regional or National
6	image	string	The path to the image file
7	Category	string	Type of the charity
8	Mission	string	Elaborate mission statement
9	website	string	Url of their website

API:

1. {{prod_url}}/charities/?id=182 : gets all the details of a charity for the given id.
2. {{prod_url}}/charities/?page=1 : gets the total number of pages (“num_pages”) and “grid” which contains charities with details upto 7th attribute in the above table.

2. Communities

Sl no.	Name	Type	Description
1	id	int	Identifier
2	community	string	Name of the community
3	image	string	The path to the image file
4	summary	string	The short summary of the community
5	state	string	State where the community is located (foreign key shared across all models)
6	counties	string	The counties located in the community
7	population	string	The population of the community
8	map_url	string	The path to the map

API:

1. {{prod_url}}/communities?page=5: get the grid page
2. {{prod_url}}/communities?id=39: get the entity with id

3. School Districts

Sl no.	Name	Type	Description
1	id	int	Identifier
2	name	string	Name of the school
3	poverty	int	Percentage of population under the poverty line
4	state	string	State where the school is located (foreign key shared across all models)
5	community	String	The community school is based in (foreign key from Communities model)
6	image	string	The path to the image file

7	district_code	int	code/id associated with each district (according to census.gov)

API:

- {{url}}/school_districts?page=2: gets school districts on a certain page
- {{url}}/school_district?id=451: gets school district with specific id

Search

We implement the search functionality by building indexes for our models and using the GCP search API. The API helps us to retrieve the ids of the model instances that have at least one field that matches the Query string. To build the indexes for each model we build a new server on Google app engine.
The search microservice's base URL is:
https://idb-search-dot-cs373-idb-218121.appspot.com/search_doc

This microservice is queried by the Django server to retrieve the corresponding ids of the instances of each model. Then the Django server queries the SQL DB to retrieve all the other fields corresponding to the instance. The Django server also highlights the corresponding text fields where the query term was found.

Microservice API:

- https://idb-search-dot-cs373-idb-218121.appspot.com/search_doc?query=Mississippi&entity=communities
- https://idb-search-dot-cs373-idb-218121.appspot.com/search_doc?query=Mississippi&entity=school_districts
- https://idb-search-dot-cs373-idb-218121.appspot.com/search_doc?query=Mississippi&entity=charities

The microservice takes the query and the entity label (charity | school_districts | communities) to return the ids.

Django server pulls the ids and returns the corresponding instances.

Django API:

- School districts Search: {{prod_url}}/school_districts?page=1&search=Texas
- Charities Search: {{prod_url}}/charities?page=1&search=Texas
- Communities Search: {{prod_url}}/communities?page=1&search=Texas
- All website Search: {{prod_url}}/all?page=1&search=Texas

Filtering and Sorting

We implement the filtering and sorting through a single endpoint for each model.
API:

- {{prod_url}}/school_districts?page=1&**population**=medium&**poverty**=extreme&**state**=alaska&**sort**=poverty&**desc**=true
- {{prod_url}}/charities?page=1&**category**=youth+education&**rating**=average&**scope**=national&**state**=alaska&**sort**=name
- {{prod_url}}/communities?page=1&**category**=youth+education&**rating**=average&**scope**=national&**state**=alaska&**sort**=name

The fields that are bolded in the URL represent the params that are accepted. Each model has 2-3 params to filter on. The sort param takes a field name to sort on and an optional desc param is used to sort in descending order.

Table Associations

Table relation	**Relation type**	**Foreign key**
Charity - Community	n-to-n	state
Charity - School District	1-to-n	community
School District - Community	n-to-n	state

Data Visualizations:

1. Poverty Data by State

a. The average poverty rate within a state and the number of districts below the poverty line (if any)
   i. Red states → greater than 16% poverty
   ii. Orange states → 14-16% poverty
   iii. Blue states → 12-14% poverty
   iv. Green states → less than 12% poverty

b. Modules used: D3, react-usa

2. Number of Poverty School Districts by State

a. The number of districts that are below poverty line for each state, if that state has more than 1 affected district
   i. Red bar → greater than 14 districts
   ii. Orange bar → 6-14 districts
   iii. Blue → 2-6 districts

b. Modules used: D3, react-easy-chart

3. Poverty Rate by State

a. The rate of poverty for each state, if the percentage is greater than 12 percent
   i. Red states → greater than 16% poverty
   ii. Orange states → 14-16% poverty
   iii. Blue states → 12-14% poverty

b. Modules used: D3, react-easy-chart

Tools:

- **Gitlab** used as our Git-repository manager
- We used **Docker** to deploy our backend
- Our project is hosted on **GCP**
- **Django** is our web framework
- Our Django application is running on **GKE**
- Our domain name is registered from **Namecheap**
- **React** JavaScript library used to build the User Interface
- **Postman** used to develop and test API
- Team communication was done primarily through **Slack**
- Front-end testing was done with **Mocha, Chai, and Enzyme**
- Acceptance testing for our frontend done with **Selenium**
- Created UML diagram with **PlantUML**
- Spelling and grammar-checking with **Grammarly**
- **Bootstrap** framework used for front-end design
- **Google Maps API** used to display locational information for communities

Hosting:

- **Google Cloud Platforming (GCP)** is used to host our website/API
- **Google Cloud Storage (GCS)** is used for storing and accessing data from database hosted by **GCP**
- Domain (EqualEducation.info) provided by **Namecheap**

Testing:

- **Unit Test - RESTful API using Postman**
- **Unit Test - JavaScript code using Mocha**
- **Acceptance Test: Java code using Selenium**

Mocha

Mocha is a JavaScript test framework running on Node.js and in the browser. We use Mocha for making asynchronous testing JavaScript.

Enzyme

Enzyme is a JavaScript testing utility for React. We use Enzyme for testing the React components by using `shallow` rendering.

Jest

We use the function `expect` from Jest. The function `expect` is used whenever you access to a number of "matchers" that let you validate different things.

Selenium

Selenium is a portable software testing framework for web applications. We use Selenium to write automated tests to test our website.

Testing Components

We test components in isolation from the child components they render by using `shallow` rendering API from Enzyme. To install it, run:

```
npm install --save-dev enzyme enzyme-adapter-react-16
```

Or, alternatively you may use `yarn` :

```
yarn add enzyme enzyme-adapter-react-16
```

Here is the example how we test navigation bar component:

test/unit_tests/Navbar.test.js

```
import React from 'react';
import Enzyme from 'enzyme';
import expect from 'expect';
import { shallow } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
Enzyme.configure({ adapter: new Adapter() });

import MyNavbar from
'../../src/components/MyNavbar/MyNavbar';

describe('MyNavbar', () => {
  it('renders without exploding', () => {
    expect(
      shallow(
        <MyNavbar />
      ).length
    ).toEqual(1);
  });
});
```

Import all the tools mentioned above including `enzyme-adapter-react-16`, and configure the `enzyme` with the `adpater`. Also, import the `.js` file containing the components that you want to test. I import **`MyNavbar.js`** as `MyNavbar` to test `MyNavbar` component. Then make an assertion to check if there is only one `<MyNavbar />` component with `.length` and `.toEqual(1)`.

To run the test, run the following command:

```
npm test
```

All the unit tests(`*.test.js`) under `test/unit_tests` directory will render without throwing, to shallow rendering and testing some of the output. With the navigation bar test, we have the following results.

```
> mocha --no-warnings --require ignore-styles --require
babel-polyfill --require babel-core --compilers
js:babel-register "test/unit_tests/*.test.js" -r
mock-local-storage

  MyNavbar
    ✓ renders without exploding

  1 passing (57ms)
```

- **Unit Test - Python code using unit tests**

Testing The Website

We use Selenium to write automated tests to test the front-end of the website. We utilize names, titles, and more to tests all of the pages as well as the functionality of each page. Our file can be found in **`test/acceptance_tests/guitests.py`**

To run selenium you must have the webdriver installed. This can be done on multiple web browsers, but our test code utilizes Selenium's Firefox webdriver. To install the Firefox Webdriver, first make sure that the path ~/.local/bin is in your execution PATH. Then:

```
$ wget
https://github.com/mozilla/geckodriver/releases/download/v0.19.1/geckodrive
r-v0.19.1-linux64.tar.gz
$ tar xvfz geckodriver-v0.19.1-linux64.tar.gz
$ mv geckodriver ~/.local/bin
```

Then install Selenium:

```
$ pip install selenium
```

We test the pages of the website using Python's unittest. We first set up the driver using the setUp method. After executing the tests, we close the driver using the tearDown method.

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class GUITests(unittest.TestCase):

      # Sets up the FireFox driver.
      def setUp(self):
            self.driver = webdriver.Firefox()
# TESTS...

      # Closes the driver.
      def tearDown(self) :
            self.driver.close()
```

```
# ----
# main
# ----

if __name__ == "__main__":
      unittest.main()
```

In the code that follows we demonstrate some of the tests for the home page of the website. Within the home page, we also test the elements in the carousel. We confirm that the text displayed is correct by finding the elements by the class name "carousel-option text-left".

```
def test_home_title(self) :
            driver = self.driver
            driver.get("http://www.equaleducation.info/")
            self.assertTrue("Equal Education" in driver.title)

      def test_home_mission(self) :
            driver = self.driver
            driver.get("http://www.equaleducation.info/")
            elem = driver.find_element_by_class_name("mission")
            assert "Here at Equal Education, our mission is to help people
find charitable organizations that support education in their desired
region." in elem

      def test_home_carousel(self) :
            # Test carousel elements.
            a = driver.find_elements_by_class_name("carousel-caption
text-left")
            # Should enter three times.
            count = 0
            for elem in a :
                  if count == 0 :
                        assert elem.text() == "School Districts"
                  else :
                        if count == 1 :
                              assert elem.text() == "Charities"
                        else :
                              if count == 2 :
                                    assert elem.text() == "Communities"
                  count += 1
```

If all of the tests are correct, then none of the assertions shall fail and we will get an OK from the Python unittest. In this way, we test the multiple pages of EqualEducation.info to confirm that it is functional.

Back-End Tests

The back-end tests can be found in *./idb/tests.py*. We have three classes of tests:

1. Unit tests for views and url parameters - This is similar to the Postman tests, except that the tests are not done over the network.
2. Unit tests for internal handler functions - These tests are checking for correctness of the individual handler functions. We setup a test database for this purpose.
3. Unit tests for database entries - We query the database to make sure the data format and the number of entries are as expected.

We use Django's internal unit test framework which is based out of python's unittest.

We have also setup the CI/CD on gitlab to run our tests automatically when a new push is made. All tests are run on a test db which is essentially a clone of our prod db.

Refactoring

Front-End fixes

- Several parts of the project have been refactored into separate components, such as the About Page and Model Page(s)
- Added redux functionality

- Removed search pagination container and moved the logic to SearchPage; only have to make three search calls now
- Used react pure component to make our rendering more efficient

Back-End fixes

- We wrote tests to make sure the back-end behaves consistently across the 3 models and 5 types of end points for each model.
- The views.py contained most of the logic that handled the API requests. This file was refactored to maintain the test results.
- We abstracted out different requirements like grid view, list view, searching, sorting, filtering, related entities, etc. into separate files and handlers.
- We followed the policy of single responsibility per handler. This helped us to enforce a clean structure in our code and we could easily split the work among our group members.
- There was a lot of code repeated across the different models. So the abstraction to different handlers helped us to reduce and clean up the code.
- The main API handlers now essentially only check for what the url is looking for and, in a water-fall logic manner, query the corresponding handlers.