

Developer Manual for the Southwestern Study Abroad Program Finder Webpage

Written by the 2020 Computer Science Capstone
at Southwestern University:
Alyssa Case, Daniel Merritt, Luke Yates,
Mason Daniel, and Ryan Wheeler

Instructor: Dr. Barbara Anthony, Math and CS Department

Last Modified: 29 April 2020

Copyright © 2020 by Southwestern University Computer Science 2020 Capstone

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Table of Contents

Table of Contents	2
Overview	4
Coronavirus Related Issues	4
Database	5
Configuration	5
Object Relational Model (ORM)	5
Database Classes	6
Structure	6
Default Class Structure	6
Program Class	7
Admin	8
Provider	9
Location	9
Database ER Diagram	10
Relationships	10
Association Tables	11
Testing	11
API	11
Configuration	11
Design	12
API Front end Methods	12
Home()	13
Results()	13
API Database Methods	14
Create New Program	14
Change Program	14
Methods to Change or Remove Relationships	15
Areas	15
Terms	15
Languages	16
Locations	16
Remove Program	16
Front End	16

Configuration	16
Narrowing Down the Working Directory	17
Design	18
Why React?	18
User Interface	20
Authentication	21
Testing	22
Contact Us	23

Overview

This project is a part of the 2020 Computer Science Capstone Projects. The desire of the project was to develop a Program Finder for all study abroad programs offered through Southwestern's Study Abroad Department headed by Monya Lemery. The program was meant to allow students and parents to search from a fully completed database of all study abroad programs with several criteria to easily find a fit for them. To accomplish the task, a fully completed front end, API, and database needed to be designed and deployed. However, the Coronavirus caused school to be moved to remote teaching which delayed most development. This document will include all design decisions made during development as well as an overall structure of code. This manual will be updated with the code itself and will be reflected in the ReadMes in the project repository.

Repository Link: <https://github.com/MerrittD/StudyAbroadCapstone>

UI Sample Demo: <https://github.com/MasonTDaniel/studyabroad-demo>

Coronavirus Related Issues

Due to the coronavirus, the team members were not able to spend the same amount of time working together as we normally would have. Despite this, we almost fully completed the project—or at least a pared down version. Development was almost completed, but the API was never fully completed and tested, so the project as a whole has never been tested. Therefore, the database should be fully functional and completed. The front end is also completed and tested with a very basic user interface and sample search results. However, the API was very close to completion and only has an issue with filtering and returning search results. We believe that if the API were to be finished and the routes used by the front end to the API were inserted onto the front end code, then the project would be ready for testing and ultimately completed.

Database

The database was designed using an object relational model and satisfies the requirements for third normal form. It is written in python using Flask-SQLAlchemy. All code was written using Visual Studio while running Python 3.7.6. It is an SQL database and was tested using an SQLite database. All functionalities of the database have been tested and passed.

However, due to the difficulties of the semester, this database was never used in conjunction with the front-end, so compatibility was never fully tested.

Configuration

The configuration of the database is done in the file “databaseConfiguration” which starts the Flask app. with the desired url of the database. It then initializes an instance of the flask application, “db”, which contains all functions from both sqlalchemy and sqlalchemy.orm. The “db” object is used to declare models, initialize tables, and change database entries with sessions.

The flask app is created using the url of the desired memory location for the instance of the database. Currently, the url is a link to a local test database within the repository because full integration with Google cloud was never completed. For future development, this url should be the location of the final database with all stored entities.

Object Relational Model (ORM)

The object relational model for the database is in the file named “databaseORM”. This file contains the code for designing all tables and methods which will be used to control and interact with the database. This file imports the “db” object that is created in “databaseConfiguration” to define classes for tables and association tables. The ORM also contains methods for serialization of the database to send information to the API. The classes defined include Program, Admin, Provider, Area, Term, Location, and Language. The association tables include the many to many connections from Program to all classes except Admin. The Admin class should be used to define usernames and passwords for admin personnel to edit program information.

Database Classes

Every class defined in the ORM represents a table in the database when the command “db.create_all()” is used. The tables then follow the rules set in the ORM to store, update, and query the information in the database.

Structure

Each ORM class is defined with a table name, individual attributes, individual methods, class methods, properties, and sometimes relationships. All table names were defined to be the

exact string as the class name, however, these names can be different. For example, the name of the Location class is 'Location'. Each table must contain columns which are attributes that every entry may have. Most classes only list an id and name as attributes, but every class may contain as many attributes as desired. For example, the Location class is defined with 3 attributes: id, the city name, and the country name. In the code, these are referred to by calling the attributes name an object of the chosen class. For example, a Location's city name can be retrieved by calling: "location_var_name.city"

Next, the individual methods are defined to act on one object at a time. All classes contain a save, constructor, and repr method. A repr method returns a string representation of the object and is very useful during testing. Class methods act on the table as a whole like a query would. Each class contains a find by id and find by name and must be called on the class itself. For example, if a programmer has a string representation of a Location and needs the location object, then they can call "Location.find_by_name(City, Country)" to retrieve the object or row of the Location class.

Each class also contains a property method that is used to serialize the information before it is sent to the API.

Lastly, a class may contain relationships. Each program may offer multiple languages to be learned at multiple locations, so in order to not duplicate data, relationships were used to cut down on storage space. These relationships are all declared using separate association tables which are described below.

Default Class Structure

Each class or table has several default attributes which are duplicated in each class and listed below. Future Development could refactor these to cut down on code reuse. Any additional class specific information is included under each class name below. The Area, Term, and Language classes all follow the structure below. Furthermore, most classes were given a serialize method. These methods are used by the api to convert the database objects into json objects that can be used in the front end.

- Table Name: Same as Class Name
- Individual Attributes:
 - id (Primary Key, Integer)
 - name (String, Unique, Non-Nullable)
- Individual methods:
 - save_to_bd(): saves the current state of the object to the database
 - Constructor [or init]: creates a new entry for the class table given all necessary attributes.

- Repr: returns a string representation of the object
- Class Methods:
 - Class.find_by_name(name): returns the object in Class table with the given name
 - Class.find_by_id(id): returns the object in Class table with the given id
- Properties:
 - serialize(): prepares the object to be sent to the API
- Relationships: None

Program Class

The Program class is the table for all programs that are available at SU which requires many attributes in order to accurately reflect a program. Several methods used below were used for testing and are never actually used by the API.

- Additional Individual Attributes:
 - comm_eng - Community Engagement Offered (Boolean, Default = Null)
 - reasearch_opp - Research Opportunity Offered (Boolean, Default = Null)
 - internship_opp - Internship Opportunity Offered (Boolean, Default = Null)
 - date_created - date and time of creation (Time, Auto Set)
 - date_modified - date and time of last change (Time, Auto Set)
 - cost - the total cost of the program (String, Nullable)
 - cost_stipulations - any quantifiers that should be included with cost (String, Nullable)
 - description - the overall description of a program (String, Nullable)
 - url - a link to the program's website (String, Nullable)
- Additional Individual Methods:
 - program_var_name.add_relationship(object): adds a many to many relationship from a program to either a language, location, term, or area. ["relationship" is replaced with either language, location, term, or area] ["object" is replaced by the variable name for the corresponding relationship]
 - For example: program_name.add_language(tempLanguage)
 - program_var_name.remove_relationship(object): removes a many to many relationship from a program to either a language, location, term, or area. ["relationship" is replaced with either language, location, term, or area] ["object" is replaced by the variable name for the corresponding relationship]
 - For example: program_name.remove_language(tempLanguage)
- Additional Class Methods:
 - Program.sort_all_programs() - returns all programs sorted alphabetically

- `Program.sort_by_relationship(relationship_name)` - returns all programs sorted by the desired relationship. [“relationship” is replaced with either language, location, term, or area] [“relationship_name” is replaced by the variable name for the corresponding relationship]
 - For example: `Program.sort_by_langauge(“Spanish”)`
- Additional Relationships:
 - Areas - This is a many to many relationship to the Area class which is stored in the association table `Programs_Areas`.
 - Languages - This is a many to many relationship to the Language class which is stored in the association table `Programs_Languages`.
 - Locations - This is a many to many relationship to the Locations class which is stored in the association table `Programs_Locations`.
 - Terms - This is a many to many relationship to the Term class which is stored in the association table `Programs_Terms`.
- Additional Properties:
 - `serialize_manyToManyRelationship()`: This calls the serialize method for every object in a relationship for Json formatting. [“Relationship” can be replaced by “Area”, “Lang”, “Loc”, “Term”]

Admin

This class handles all admin logins including username and passwords so that admins may edit the information on the site. A normal user will only be able to view the information. This class is never used because the admin testing was done using another method described below. This class would be used for future development. This class was written by Daniel Whitney: GitHub Repo below:

https://github.com/Daniel-Wh/radiosonde/blob/master/models/station_model.py

- Additional Individual Attributes:
 - username (instead of name) - username of SU Study Abroad Staff
 - password - username of SU Study Abroad Staff

Provider

Most programs are controlled and offered through a Provider. One provider can have many programs. The client desired for this class to be included, but it was also never used by the front end.

- Additional Individual Methods:
 - `Provider.add_program(programName)`
 - `Provider.remove_program(programName)`
- Additional Class Methods:

- Provider.return_all_providers()
- Additional Relationships:
 - Programs: This is a one to many relationship to the Programs class which is stored in the association table Programs_Providers.
- Additional Properties:
 - programSerial(): This serializes all programs in a providers relationship so that it can be made into a Json.

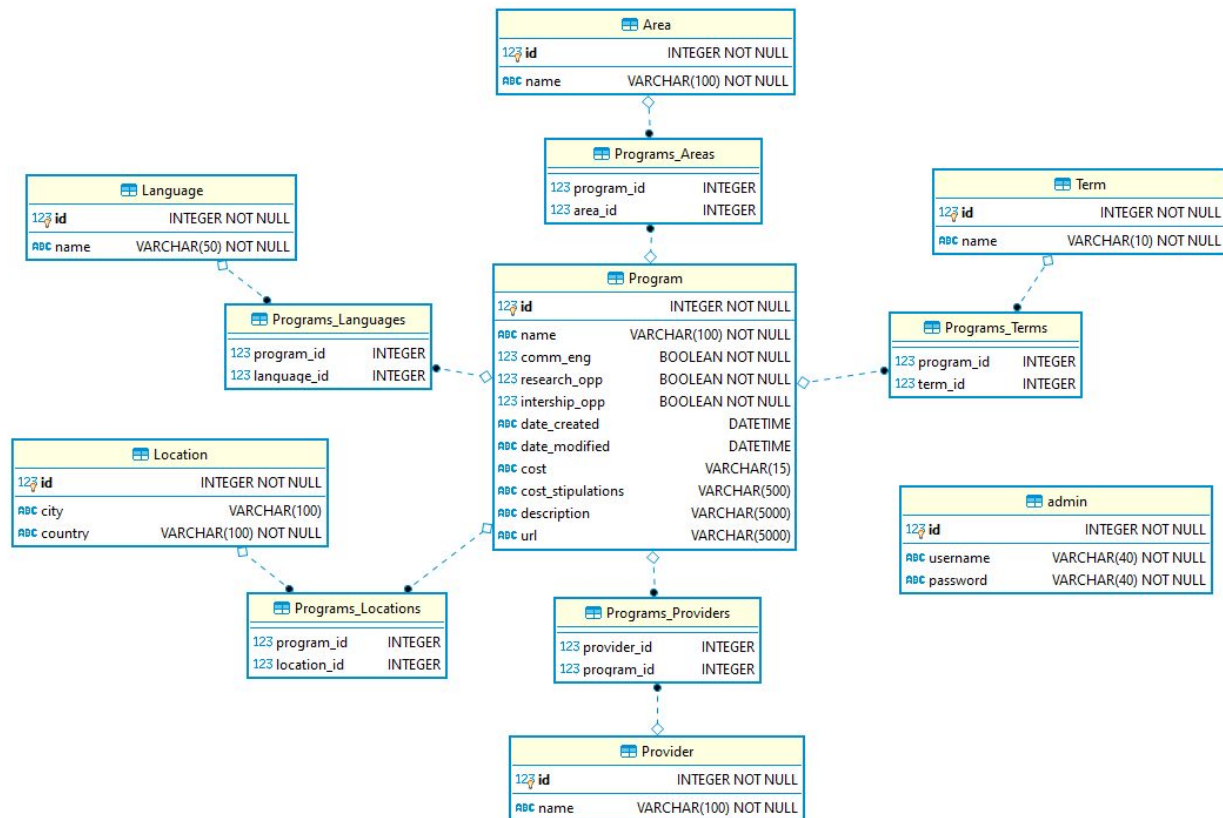
Location

This class defines the Location table which holds all locations where a program can be offered. This class contains both the city and country of a location due to the complexity of having both a city and country with an association table in between. The excess storage of the association table seemed to outweigh the duplicated city in different countries like Paris, Texas and Paris, France.

- Additional Individual Attributes:
 - City (replaces name from default) - city name
 - Country - country name

Database ER Diagram

Every box below represents a table in the database along with a list of the table's attributes.



Relationships

The relationships listed above under the classes Program and Provider are established using the SQLAlchemy relationship method which allows for the relationship to be treated as a list. For example, the “`program.add_language(language)`” command will append a new language to the specific programs language relationship list. In reality, the list doesn't exist and is only created by a query on the association table. SQLAlchemy handles all updates of the association table through this list structure. However, if an object is deleted, then only the association is deleted from the table, but the entity stays even if it was the only entity's relationship. For example, if the only program which offers “Swahili” as a language is deleted, then Swahili would still exist in the Language table without any reference to it. This is solved by the remove and change relationship methods in the API.

Association Tables

Every Relationship is stored in an association table as mentioned above. Association tables hold only two columns of information, and in this case, both columns are ids of other classes. Program ids are included in all association tables because this is the class for which the project was designed. There exist a many to many connection from Program to language, area, term, and location, and a one to many connection from Providers to Programs. In SQLAlchemy, association tables are declared by using the Table() command with the name and desired id types to be stored. Once the table is created, then the relationship must be mapped in the parent class. For example, the relationship from Program to Language will always be changed per program, so the relationships only exist in the Program class and not evident to the Language class. This could be changed easily and would allow for more complex queries than the queries implemented in the APU currently to occur easily.

Testing

All classes and tables in the Database were tested using scripts in the folder “TestScripts+Databases”. These scripts created multiple programs to be created, modified, and deleted to ensure all needs of the database could be performed without error. All databases that were used during testing are stored in the “PreviousDatabaseTests” folder for project documentation.

The tests were run using Visual Studio and DBeaver Community Edition 7.0.2 to check the changes made to the database actually occurred. All tests passed when solely testing the database using a local empty database, but the database was never tested with the API and front-end.

API

Configuration

The Flask application created and connected to the sqlalchemy database is used by this API. The API itself launches this app in debug mode and sets routes that can be used by the front end for data requests and uploads. Currently, the API must be run on a local machine using the command “Python API-WIP.py”. This will run the app on the host computer and create a local

instance for connection. For future development, this API should be migrated onto either a permanent local machine or a Google cloud instance.

Design

The API is fully defined within the API-WIP.py file, and is divided into two overall sections. The first section defines the routes that the front end calls for data and methods used by these routes. The second section consists of helper methods that could be moved to a separate file.

Four main routes are defined in the first half and are designed to be used by separate pages of a website. The first section, defined as “/”, simply returns all programs and providers to the front end. This is enough to create a single page application for displaying data, and contains all information needed for all programs and providers.

API Front end Methods

The second route, “/results”, allows for the front end to have results filtered based on predetermined attributes before returning the results. Currently, each search category is limited to only one value, though this restriction will be worked around in the future. Furthermore, it makes a lot of requests to the database itself, querying all tables and filtering with every call. This is inefficient and should be changed in the future to minimize tables queried.

The “/login” route currently is not usable. In the future, this will be defined and used with a local security login function. It should be used by administrators so that modifications to the database can be made.

Finally the “/admin” route gives access to methods that can search the database with filters, modify entries, add entries and remove entries. The filter search has the same issues as the “/request” route, as it calls the method connected to the route. Modification and addition of entries requires a formatted json file to be sent to the API containing all needed data. Finally, deletion requires only the name of the program be passed. In the future, this should be changed to accept a primary key.

All methods in the second half are called by the API only and should be moved to a separate file in the future. These methods include: create_new_program, change_program, methods to change or remove relationships, and remove_program. These will be elaborated upon in Database Methods.

Home()

This method is connected directly to the route “/” and can only handle GET requests. It takes no parameters. First, all programs and their providers are queried via the database method,

using `Provider.return_all_providers()`. These objects are then serialized and returned via Json file.

Results()

This method can be called by the `‘/results’` and the `‘/admin’` route via GET requests. It accepts five main parameters that match with the options for searching displayed on the front end. Once these are received and broken apart, they are then used to create a query request. Each category is checked for values, then those values are used to append new filters to the query request. Finally, all returned values are serialized and returned as a Json. Should no results be found, and empty Json will be sent.

Parameters

- Strings sent and separated via commas: `lan, loccity, loccountry, area, term`.
- Default values are set to null. Requests and parts of the request are sent via standard API request format. An example of such a request would be `?lan=Spanish,English&loccity=Mexico City&loccountry=Mexico`. Currently, city and country must be sent for a request

Check()

This method is called via the `‘/admin’` route and can handle GET, POST, PUT and DELETE requests. GET requests simply call the results method and require the same parameters. Post is used to modify entries and takes in the original name, the values to change, and a formatted json file with additional information. Put requests simply take a Json of all needed values. Delete simply takes the name of the program.

Parameters

- For get requests, all parameters are the same as request.
- Post requires all parameters, Put requires the Json, Delete requires the `originalName`.
 - Any parameter left blank will be a none value in the database
- Strings: `originalName, langPar, termPar, areaPar, locPar`.
 - The Par parameters must be strings either “add” or “remove”
- Json: an json file with the following entries
 - Strings: `upname, upsti, updesc, upurl, uplang, uploccit, uploccou, upter, uparea, upprov`.
 - These represent the new name, cost stipulations, description, url, languages, cities, countries, terms and areas. Cities and countries must be in order.
 - Booleans: `upcom, upre, upin`
 - These represent communication, research, and intern opportunities
 - Integers: `upcos`
 - This is simply the cost of the program

Login()

This method currently does not work. It would use the Get request to login and the Put to add new users.

API Database Methods

Due to the complications caused by coronavirus, most development of the API had to be completed without physical team interaction and pair coding. Due to this, the API was written in 2 parts, the filtering and communication as the first with database interaction methods second. Therefore, the second half of the API code is methods which are called in the filtering and communication section to simplify development. This approach seems to accomplish the desired outcome during testing and all methods work as intended, but the API was never fully finished, so these methods were never tested by the front end. These methods could be changed very easily to increase efficiency if proper coordination between the database had been established.

Create New Program

Create_new_program takes in the name of the program and every individual attribute of a program in their native type, including the areas, terms, languages and locations as lists. It does not automatically connect it to a provider. In the future, this feature would be added. Reference the database code for a list of all parameters that are used for the program creation.

Parameters:

- Atomic Values: providerName, programName, com, res, intern, cost, cost_stipulations, description, url
- Lists: areas, terms, languages, locations
- Notes
 - If a parameter is nullable, then pass "None" for the value: cost, cost_stipulations, description, url.
 - If it is meant to be a list, then leave the list empty "[]": areas, terms, languages, locations

Change Program

change_program(programName, com, res, intern, cost, cost_stipulations, description, url)

Change_program takes in the name of the program and all individual aspects of the program, excluding the relationships. The provider cannot be changed using this method. In the future, this would be added and this method would call the individual relationship changing methods. For example, if a program only needs to change the cost and cost_stipulation attributes, then the call would be made below:

“change_program(None, None, None, None, “\$6,578”, "Not all included", None, None)”

Methods to Change or Remove Relationships

Each of these methods takes in a string, telling the code to either remove or add items to a relationship, as well as the program name and the list of things to add. These methods can be used to modify the areas, terms, languages and locations of a program. In order to achieve a multi use method, the use a string variable is used to decide between adding or removing of the relationship. If the method is removing, then the front end will pass "remove" to the method and all entities in the parameter list will be removed from the relationship. However, if it is adding, the front end will pass "add" instead. For example, if the SU Buenos Aires program is now offering spanish as a teachable language, the call below should be made:

change_or_remove_languages_for_program("add", "SU Buenos Aires", ["Spanish"])

Areas

Call: “change_or_remove_areas_for_program(change, programName, areas)”

This method will define changing or removing an area from a program and will return true if no value error was raised and the change was made.

Parameters:

- change- (string: "add" or "remove") specify desired change to db
- program- (string) name of program that the areas are being changed for
- areas- (list) names of areas being added or removed from program

Terms

Call: “change_or_remove_terms_for_program(change, programName, terms)”

This method will define changing or removing a term from a program and will return true if no value error was raised and the change was made.

Parameters:

- change- (string: "add" or "remove") specify desired change to db
- program- (string) name of program that the areas are being changed for
- terms- (list) names of terms being added or removed from program

Languages

Call: “change_or_remove_languages_for_program(change, programName, languages)”

This method will define changing or removing a language from a program and will return true if no value error was raised and the change was made.

Parameters:

- change- (string: "add" or "remove") specify desired change to db
- program- (string) name of program that the languages are being changed for
- languages- (list) names of languages being added or removed from program

Locations

Call: “change_or_remove_locations_for_program(change, programName, locations)”

This method will define changing or removing a location from a program and will return true if no value error was raised and the change was made.

Parameters:

- change- (string: "add" or "remove") specify desired change to db
- program- (string) name of program that the locations are being changed for
- locations- (list) names of locations being added or removed from program

Remove Program

Remove_program and Remove_programs_from_provider are used to remove programs from the database. Remove_programs_from_provider takes the provider and the list of program names to remove. Remove_program only takes the name of the program. Future development would change these to take in the primary keys and support adding programs to providers.

Front End

Configuration

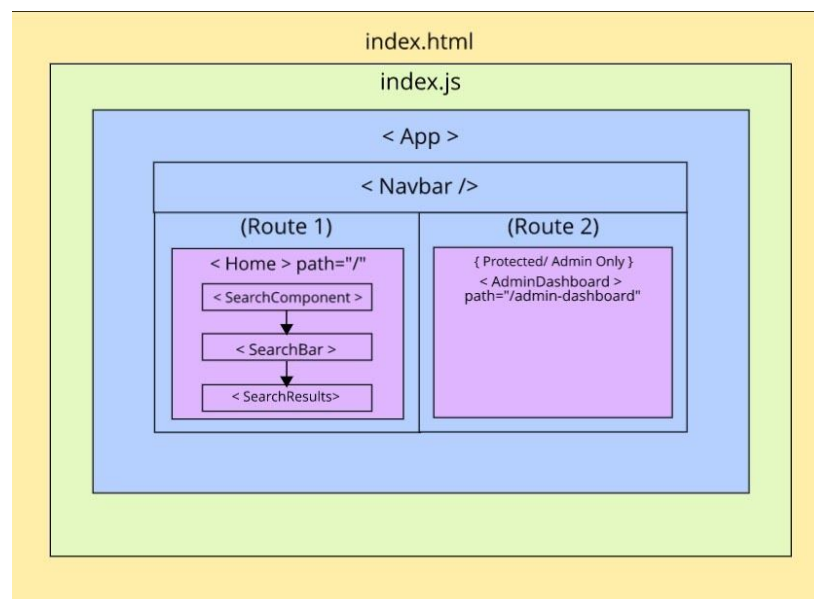
The front-end code for this project is held entirely within the **front-end** folder of the repository. It was developed using the JavaScript library React. Since React can be intimidating to configure and build, we utilized the very popular [create-react-app](#) for the initial setup of this user interface (create-react-app requires node.js to be installed). A list of dependencies for the front-end can be found in the package.json file, and the node_modules folder holds the webpack/babel configurations that were generated by create-react-app during initial setup (both of these are in the root directory). In order to start working/testing in this directory, once the

front-end folder is opened in an editor (VSCode works well), run 'npm install' in the terminal to get necessary dependencies, and then everything will be configured. Try to run 'npm start' in the terminal and it should pop up a local dev server (if not, you most likely need to npm install more dependencies, and the error messages will tell you which ones) Note: make sure you are in the front-end directory. You may need to type 'cd front-end' into the terminal before 'npm start'.

Narrowing Down the Working Directory

There are many config files that are generated with create-react-app that make the front-end directory a bit overcrowded. For developing purposes, it is important to note that we actually only deal with a few files when it comes to the front-end code, which are the following (all except the first are found in the 'src' folder—the first is found in the 'public' folder):

- **index.html**: this is essentially the parent--the most important file that holds the entirety of the front-end files (found in 'public' folder, the rest of the following are found in the 'src' folder). The 'root' div is the entire project
- **index.js**: the second-most important file, which essentially places App.js into index.html by binding the App in App.js to the root div mentioned in index.html
- **App.js**: the file that holds the actual App itself, with all routing specified
- 'pages' folder:
 - **Home.js**: homepage
 - **AdminDashboard.js**: admin dashboard page
- 'components' folder:
 - **NavBar.js**: the navigation bar which is present on the homepage and admin dashboard page
 - **SearchComponent.js**: the parent component of SearchBar.js
 - **SearchBar.js**: the parent component of SearchResults.js
 - **SearchResults.js**: the child of SearchBar.js, which is displayed upon search



These are the only files that were altered during development. The 'css' and 'img' folders in src are simply resources folders for css bootstrap files and image files respectively. Any of the other files present in the front-end working directory are abstracted by create-react-app, and all we really need to know is that they are necessary--not their purpose.

Design

Why React?

React is a component-based library, which means the webpages are made up of separate but reusable components (e.g. the Search Bar and Nav Bar are both components). Although our web application is not at a large scale, we felt that its use of components would be beneficial to us since we knew exactly what our pages would look like before development and could change them easily if needed (i.e. highly scalable and in-place changes). It is also used by a decent amount of software developers and thus would provide hand-on experience with a practical technology.

Components and Pages

This project had very specific requirements in regards to its functionality, and therefore the design and layout of the project is concise. Our project has only two pages made up of four components. Components and their state may be inherited from one another (e.g. SearchResults is the child of SearchBar), and components may also be shared between pages (e.g. NavBar being used in both the Home page and the AdminDashboard pages).

Here are the basic functions of each component:

- NavBar
 - Provide links to navigate through the web application
- SearchComponent
 - Makes an http get request to the database for all programs.
 - Stores the data in an array
 - Passes down that array to SearchBar
- SearchBar
 - Takes in the array of all programs
 - Extract all possible terms, countries, areas of study, and languages currently present in the data and maps them to four respective arrays
 - Sorts and removes duplicates in all these arrays, then populates the dropdown filters with them
 - When dropdowns are changed, the component state is updated to reflect those changes

- When 'Search' is clicked, make another get request for all programs
 - Filter the programs into an array if they match the input filters
 - Render SearchResults and pass down array of filtered programs
- SearchResults
 - Display the filtered program array in a table
- Home
 - Holds the SearchComponent component
- AdminDashboard
 - Perform an http get request for all programs and display them in a table
 - 'Add a Program' feature allows admin to add a program to the database
 - 'Edit' feature allows admin to edit a program already existing in the database
 - 'Delete' feature allows admin to delete a program already existing in the database

Notes About Components and Pages

These are the current state of the front-end directory in the final project. Due to time constraints from the displacement resulting from covid-19, not all of these components and pages are in their most efficient form. A few structural changes could be applied to make the code more lightweight:

- SearchComponent and SearchBar both perform get requests. The get request in SearchBar could probably be removed and instead just use the array filled with all program data assigned after the get request in SearchComponent (a.k.a. props).
- AdminDashboard could be separated into components, since components are meant to represent 'pure' functions
- Our project was initially meant to take care of filtering the programs in the backend/api. Due to time constraints, this was not able to be implemented fully. Therefore, the filtering of the programs done in SearchBar was a last minute change to have a working demo. While deployment is possible with this approach, a backend filtering approach would mean less work on the client side—although due to the relatively small amount of programs in the database, this may not be an issue anyway.
- There are likely many refactorings that can be done to this code to give it better performance, efficiency, etc.

User Interface

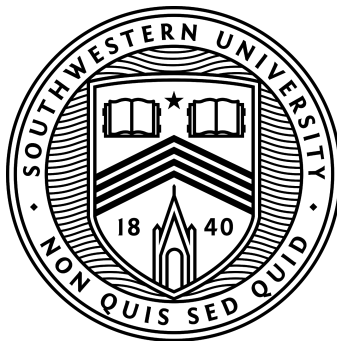
The user interface design is highly based on Southwestern University's [style guide](#):

Logo



SOUTHWESTERN
UNIVERSITY

Icon



Typography

Type

Playfair Display Regular and Italic

This type should be used for headlines and introductory copy. Italic should be used for emphasis.


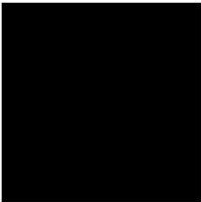
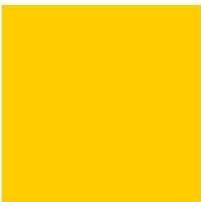
Aa Aa

HK Grotesk Light, Regular, and Bold

This type should be used for all body copy, including quotes and sub- or category-headlines.

Aa Aa **Aa**

Color Palette

	Hex: #FFFFFF	R: 255 G: 255 B: 255	C: 0 M: 0 Y: 0 K: 0	PMS C: N/A PMS U: N/A
	Hex: #000000	R: 0 G: 0 B: 0	C: 75 M: 68 Y: 67 K: 90	PMS C: Black C PMS U: Black U
	Hex: #FFCD00	R: 255 G: 205 B: 0	C: 0 M: 19 Y: 100 K: 0	PMS C: 116C PMS U: 114U

We have done our best to comply with these principles, and will continue to make changes to the user interface in order to maintain them if necessary.

Authentication

In its current state, this project does use authentication. This is done using a tool called [Okta](#). Okta is a third party application that is relatively easily integrated into a React application. Most of the overhead is handled by Okta—all one must do is implement some of their code into their project to have a functioning login. The implemented Okta code is documented in the code, which includes the following files: App.js and NavBar.js. /admin-dashboard is the only protected route, and you can see that it is wrapped in a <SecureRoute> tag in App.js accordingly. What this means is that if a user navigates to /admin-dashboard, they will be redirected to a login screen. This route is the only way a user would be able to alter the database, so protecting that route in turn protects the data. As far as we know, this implementation serves as working authentication. It seems as though this would work in deployment, too, and would keep things

secure—with that said, it could not be, and a manual authentication implementation may be necessary.

With that said, there was a big limitation to using Okta. The demo version (masontdaniel.github.io/studyabroad-demo/) does not use authentication because there were issues with it during deployment. Navigating to /admin-dashboard (whether by clicking Admin or typing it in the address bar) does redirect to the sign in screen. The problem is that when the user clicks ‘sign in’ and is authenticated, it redirects to a page that is not held within the available urls and produces a 404 error. This may be due to the fact that this project was test-deployed on github pages, and another host may not have this issue. Therefore, our project repo does include authentication because it works locally, but the demo version does not. This issue will hopefully be looked into further depending on if this project gets carried out.

Testing

Testing for the front-end is pretty simple. Because we used create-react-app to generate the project, we can run a local dev server to test the front-end by typing ‘npm start’ in the terminal (as aforementioned, making sure you are in the front-end directory first). This local server displays the current render of the project, and automatically updates upon saving.

The next step was to try and test out the filtering, adding, updating, and deleting of programs from a data storage (i.e. http requests get, put, post, and delete). This initially was done using a [JSON server](#). The issue with this approach, though, was that data did not persist between requests. In other words, you could only perform one request, and then it would undo that request before making the next. So, the JSON server was deployed using Heroku. Since the JSONified data was now hosted, we were able to perform tests on the front-end using a fake REST api. From this testing, we got all of the requests to work.

The only limitation to this was performing the filtering. This would theoretically be done by performing a get request on the front-end, which would pass a payload containing the filters to the api, which would then handle the filtering and respond back with the appropriate data. The fake REST api could not replicate this, so filtering was done on the front-end during testing. In other words, if this project were to deploy, we could keep it in its current state (with front-end filtering) and all would be well—hopefully. Otherwise, the api filtering would need to be completed and tested, and then the front-end code would have to be altered to account for those changes (by removing the filtering and adding a payload to the get request in SearchBar).

Contact Us

We all wish to see this project deployed some day, so for any questions regarding anything, feel free to reach out to us at any of the emails provided below.

<u>Name</u>	<u>Email</u>	<u>Part Contributed to</u>
Alyssa Case	ajcase1997@gmail.com	
Daniel Merritt	DanielBMerrittIII@gmail.com	API, Database
Luke Yates	lukeyates1@gmail.com	Database, API Database Methods
Mason Daniel	masondanielt@gmail.com	Front-end
Ryan Wheeler	ryanwheeler67@gmail.com	Front-end