UVA Automated Course Advising Assistant

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Rahul Batra

Spring, 2020

Technical Project Team Members

Bugi Abdulkarim

Christine Cheng

Sean Gatewood

Alex Hicks

Scott Lutz

Rahat Maini

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Luther Tychonievich, Department of Computer Science

Rahul Batra
Sean Gatwood
Scott Lutz

UVA Automated Course Advising Assistant

I. Introduction

This document describes the functionality of the UVA Automated Course Advising Assistant (referred to herein as, "the Assistant") . This website, developed as an undergraduate research project, provides students with an interface to create long-term course plans (i.e. which courses to take in each of their remaining semesters). Students can specify constraints by “pinning” courses to specific semesters and by setting the maximum course loads for their semesters, and the system will attempt to redistribute the remaining courses to satisfy all of the constraints. This document will explain how to deploy the Assistant locally or on a cloud virtual machine, describe the overall system design, and explain the most complex components in the source code. A screenshot of the interface is shown in the Appendix.

II. Deployment

Dependencies. The Assistant is distributed using Github. It is containerized using Docker, and it is built with Make. Consequently, if Git, Docker, and Make are installed, these tools will take care of fetching all other dependencies.

Local deployment. After pulling the repository,[1] navigate to the top-level `advising-assistant` folder. Run `make build_and_launch`. This command builds and starts the docker containers. Then, open `http://localhost:8000` in a browser to view the app. CMD+C will stop the containers. After you have built the containers, you can run `make launch` to restart the same containers without rebuilding them from scratch.

```
git clone
git@github.com:advising-assistant/advising-assistant.git
cd advising-assistant
make build_and_launch
# open http://localhost:8000 to view the app!
```

Figure 1: Programmatic summary of deployment instructions

The app will not fully function at this point, however, as you need to populate your Course database with course data from the SIS catalog. See “Helpful Scripts” for instructions on how to do this.

[1] https://github.com/advising-assistant/advising-assistant (a private repository)

Deployment on a cloud virtual machine. During the development of this project, we regularly deployed the system on a Linode[2] server. The details of this task may depend on your specific cloud virtual machine, but high-level instructions are given here. First, you must undergo the process of securing a domain name and pointing it at your machine's nameservers. (Without a domain name, the google authentication will not work.) We used iwantmyname.com for this, but any domain name service should work. Next, you should set up HTTPS on your cloud virtual machine. You can do this for free with Let's Encrypt,[3] and Linode has a guide specifically on how to set this up.[4] Make sure to edit `docker-prod.yml` to match your domain information:

```
version: '3.7'

services:
  letsencrypt:
    build: ./nginx-prod
    container_name: letsencrypt
    environment:
      - PUID=1000
      - GUID=1000
      - TZ=America/New_York
      - URL=www.exampledomain.com
      - VALIDATION=http
      - EMAIL=john.walker@gmail.com
    volumes:
      - staticvolume:/collected-static
      - letsencryptvolume:/config
    ports:
      - 443:443
      - 80:80
    networks:
      - nginx_network
    depends_on:
      - web

...
```

Figure 2: Relevant section of docker-prod.yml

Finally, follow the steps above for local deployment, except run `make deploy`.

[2] https://www.linode.com/
[3] https://letsencrypt.org/
[4] https://www.linode.com/docs/security/ssl/install-lets-encrypt-to-create-ssl-certificates/

Helpful Scripts

The Assistant's Django backend is controlled via Django's usual `manage.py` script (see https://docs.djangoproject.com/en/3.0/ref/django-admin/). However, because this Django application runs in a Docker container, running manage.py outside of the container will not work. As a fix, a bash script named `manage.sh` has been provided to relay command(s) to manage.py in the Docker container using `docker exec`. If additional commands need to be run within the Docker container, running a bash script named `ssh.sh` will start a bash shell in the Docker container.

The `manage.py` script also contains functionality to aid in database maintenance. Running `./manage.sh` will output a list of all functionality that manage.py can provide, and any of these commands can be run through the manage.sh script. One specific command `make_db` is necessary in populating the Course database with data from the SIS catalog. This command can be run by `./manage.sh make_db`. *The magic wand routine will not work until you have run this command*. Note, `./manage.sh make_db` utilizes a JSON-cached version of the SIS catalog included in the repository. At the time of writing, it is expected once a year (depending on the frequency SIS catalog updates), one can run the following command to update this cache of SIS data: `./manage.sh pullall`.

`./manage.sh doc` generates the HTML documentation of the backend Django app in the advising-assistant/pdoc folder.

Testing and Debugging

Print statements from the codebase will show up in the window running the Docker container. If the file executing the print statement is run in the background and not directly executed via Docker then the print statement will not show up in the Docker window. Automated tests of the magic wand routine are found in `advising-assistant/backend/tests.py`. These tests can be executed via the command `./manage.sh test`**.** The command `./coverage.sh` will also run these tests and generate HTML reports of the test coverage in the `advising-assistant/htmlcov` folder.

III. System Design

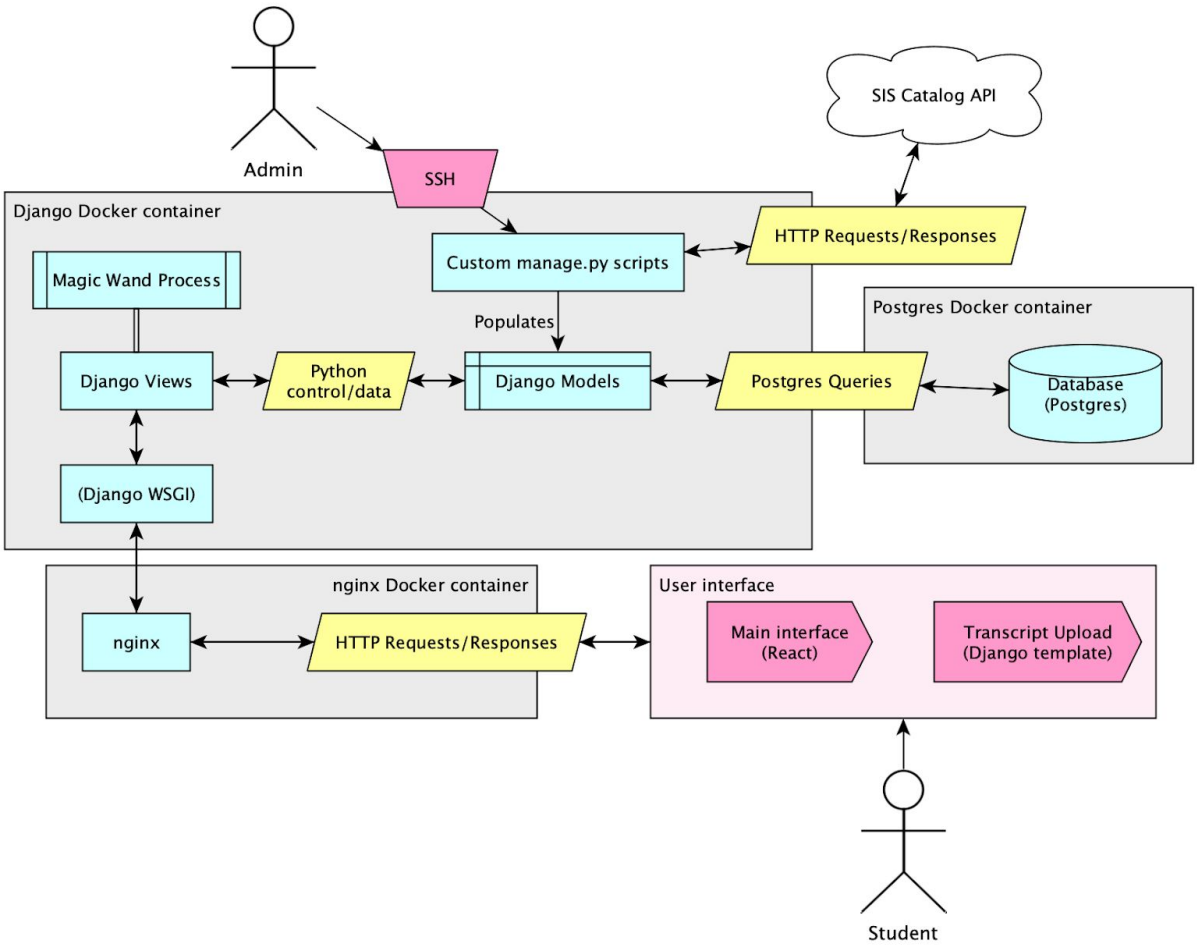

Figure 3: High-level system design

User interface

The main front end interface is written mostly in React, with the exception of the transcript uploading component, which is made from pure Django templates. Components within the interface make HTTP requests to the server hosting the Assistant, and the Assistant responds with the data necessary for rendering the interface. For instance, when the user loads the main interface, the `<Plan />` react component makes an HTTP request to `<hostname>/backend/get_semesters`, and the server responds with the JSON that React uses to render the user's semesters.

nginx

Instead of handing the HTTP requests directly to Django, nginx acts as middleware proxying the requests to the Django container. This encapsulation increases security and scalability, as well as customizability. For instance, if the admin wants to change what port the Assistant is exposed on, they can do so within the nginx configuration (in the `nginx` and `nginx-prod` folders) without changing the Django configuration.

Django Views

Django's views are the control logic in the Django MVC framework. These are functions that receive an HTTP request, complete some internal subroutine (usually involving the access and mutation of data models), and respond with an HTTP response. For instance, the `get_semesters` view accesses the data models for the user's Semesters, and returns the relevant data as JSON.

Custom manage.py functionality

As mentioned in the Helpful Scripts section, manage.py's functionality was extended to include database management commands. If you are working within the Docker container they can be run directly with the python script (e.g. `./manage.py make_db`). Otherwise, `./manage.sh` can be used (i.e. `./manage.sh make_db`).

Models

Django's models define a python API for accessing the database. Models abstract much of the actual query transactions, simplifying database access and allowing the developer to switch out the underlying database implementation. (We chose PostgreSQL for scalability.) Models also improve cohesion and decrease code repetition because common operations have been defined as methods in the model's class.

Database Schema

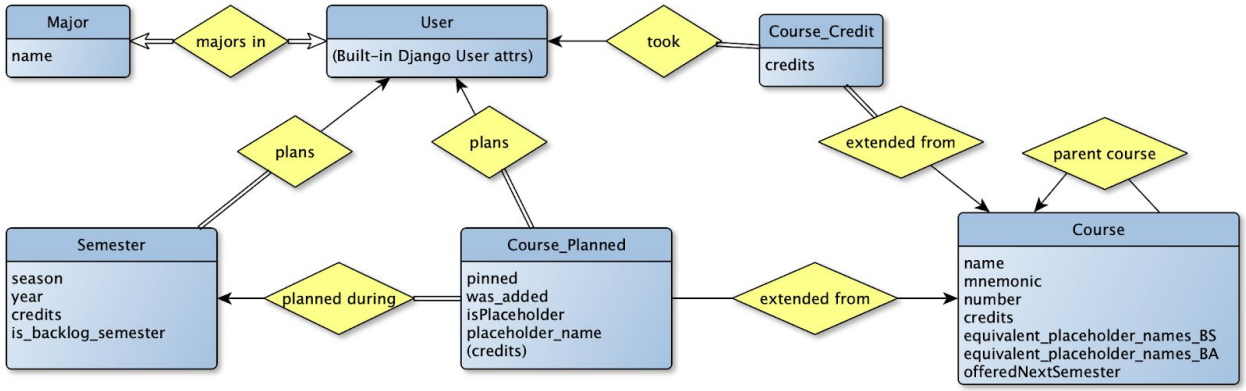

Figure 4: Entity-relationship diagram of database schema

Course. The database contains information on every course offered in the SIS catalog. Each course is stored as a Course model. The Course database is populated from the SIS catalog with the custom command `./manage.sh make_db`**.** Some courses can have "Parent Courses", relating them to the course they depend on (for instance, CS 2150 lab depends on CS 2150 lecture). This attribute is not currently used. Course_Planneds and Course_Credits both have foreign keys to the Courses they relate to. This design was chosen to store information about courses in only one place.

The equivalent_placeholder_name attributes are the current implementation of elective mapping. Each Course has a JSON list of potential elective types it could fill, for each major implemented, with an order of the priority for which it should try to fill. For instance, for B.S. Computer Science, CS 4810 would have the list `["CS Elective", "Unrestricted Elective"]`. However, this design should be refactored. It does not match the design of SIS (where this information is stored with majors, not courses), and additional attributes would need to be added to the database schema for every single major. Both Course_Credits and Course_Planneds use these attributes to map to elective placeholders.

Course_Credit. This model represents a course that the user has credit for. It is a one-to-many relationship between User and Course, with a "credits" attribute representing how many credits it counted for. Note that credit information cannot be derived from the related Course object for two reasons. First, if the course is variable-credit (such as CS 4980), its credit value depends on how many credits the user took it for. Second, if a repeatable course (like CS 4501) is seen twice on the transcript, the Assistant currently combines the two courses into one with the credit totals added together. The user's Course_Credits are added to the database by the transcript uploading feature, which scrapes the user's course history from their transcript. The magic wand routine disregards any courses or constraints that relate to courses the user has taken.

Course_Planned. This model represents a course that the user has planned to take in a future semester. It has foreign keys to the user it belongs to, the user's semester it was assigned to, and the Course it references. It also stores information on its state in the interface, such as whether it was added by the user to fill electives, whether it is pinned, and whether it is a placeholder. If the Course_Planned is a placeholder, it does not have a foreign key to a Course model, and instead its placeholder_name is displayed on the interface. Although the Course_Planned's user could technically be accessed through the Course_Planned's semester, it became common in our source code to query all of the Course_Planneds for a specific user. (For instance, the magic wand routine must query all of the user's Course_Planneds to know which courses the user has pinned.) Thus, adding this redundant attribute reduced the database query time for our existing code.

Semester. Semesters can be modeled as lists of Course_Planneds, with some additional attributes about the semester's term, year, and credit maximum. There is also a field to define a backlog semester which can store a list of Course_Planneds without tying them to a term, year, or credit maximum. The magic wand routine places any courses that cannot satisfy all of the constraints in the user's backlog semester.

User. The User model is the built-in Django User model in `django.contrib.auth.models`.

Major. Each User has one major, which is a model storing a major name, like "B.S. Computer Science." This one-to-one relationship was added as a model because Django does not recommend adding attributes to the User model. That is, there is one Major object for every User object in the database. This should be refactored to a many-to-many relationship to cut down on the duplication of Major information, and to allow users to have multiple majors. The user's major is currently used to select which course_data file to load during the magic wand routine, as well as which equivalent_placeholder_name attribute to use when mapping courses to electives. (These decisions would also need to be rearchitected if multiple degree programs per student are allowed.)

IV. Complex Routines

Magic Wand

This is the routine that redistributes the user's remaining courses to satisfy all of the constraints. The routine accomplishes this by reducing the task to a Constraint Satisfaction Problem (CSP),[5] which is solved with the Arc Consistency Algorithm #3 (AC-3)[6] with backtracking.

Our implementation of the AC-3 algorithm works with three inputs:

- **Arcs**: A list of pairs of courses. This represents the edges in the directed graph AC-3 works on.
- **Domains:** The set of possible semesters for each course (implemented as a dictionary of Course → list of possible semesters). This represents the initial domains that the AC-3 algorithm narrows down.
- **Constraints:** A dictionary of arc → {boolean function/lambda with 2 inputs}. This represents the binary constraints that must be satisfied between courses. Our system does not have unary constraints.

The result of the AC-3 algorithm is another dictionary of domains. Backtracking is performed to convert the resulting domains to a schedule (dictionary of semester_number → list of courses). The magic wand routine works as follows:

I. The user clicks the magic wand icon.
    A. The icon is a React component, programmed to send an empty GET request to `backend/magic_wand`.
        1. The request goes through nginx and the Django WSGI, and the `magic_wand` view is called.
        2. This view creates a `Magic_Wand_Request`. This is a class that wraps most of the functions needed for the magic wand routine in views.py. It stores some information as attributes, which cuts down on parameter passing without storing anything in the global scope.
        3. The magic_wand view runs the Magic_Wand_Request with its `.run()` method.
            a) A `User_Data` object is created. This class wraps information about the user's course history, pinned courses, semesters, etc. Wrapping this information in a class cuts down on parameter passing and allows for easier development of features in the future, even deep within the call stack.
            Among the User_Data's attributes is a list of `course_credits`. This includes not only courses that the user has credit for according to the Course_Credit model, but also any Course_Planned objects that have been backlogged. Although

[5] https://en.wikipedia.org/wiki/Constraint_satisfaction_problem
[6] https://en.wikipedia.org/wiki/AC-3_algorithm

these objects have different types, they both have `get_credit_string()` methods to allow for some polymorphism.

b) The `run()` method passes the user_data to `get_schedule()`
   (1) This function (in backtracking.py) first creates a `Course_Data` object for the user.
      (a) The purpose of the Course_Data class is to set up the arcs, domains, and constraints for the AC-3 algorithm.
      (b) When initialized, it pulls the hard-coded constraint information from either `course_data_BS.py` for `course_data_BA.py`, depending on the user's major.
      (c) It then converts the non-placeholder course strings in these files to Course objects, using the `Coursefier` class for some caching.
      (d) At this point, the arcs, domains, and constraints are valid for a new first-year student with no course history.
      (e) Finally, all arcs, domains, and constraints that relate to any courses in the user's course history (or backlogged courses), including elective placeholders, are filtered out.
   (2) Then, `get_schedule()` passes the course_data to `solve_csp()`. This function makes and runs a `CSPSolver` for the course_data's arcs, domains, and constraints. The CSPSolver runs the AC-3 algorithm and returns the results.
   (3) Then, `get_schedule()` initializes a `Schedule_Generator` object, and calls its `run` method.
      (a) A Schedule_Generator converts the resulting domains from the CSPSolver (a dictionary of Course → list of possible semesters) into a schedule (dictionary of semester_number → list of courses). This component was originally a function, but it was wrapped in a class to encapsulate all of its helper methods

(b) At a high level, this routine repeatedly selects the course with the smallest remaining domain, and assigns it to the semester in its domain that has the fewest courses already assigned to it.

(4) Then, get_schedule() returns the results from the Schedule_Generator

c) If all of the constraints were unable to be resolved, then the offending courses are moved to the backlog semester, and the `run()` method restarts.

d) Otherwise, the `run()` method assigns each course to its planned semester (creating/modifying Course_Planned models), and returns an HTTP response about any courses that were backlogged.

4. The magic_wand view returns the HTTP Response generated by the `run()` method. This response is returned to nginx, and is sent back to the client.

B. Back on the front-end, the React component receives the HTTP response, and executes a callback function. If any errors occurred, a vanilla javascript `alert()` displays the error to the user. If the error was not fatal (i.e. courses were backlogged), a page reload is triggered.

II. The page reloads if there were no fatal errors. The user sees that the system has redistributed the remaining courses to satisfy constraints.

Transcript Upload

This is the routine that populates the Course_Credit database for a user. It works as follows:

I. The user clicks the upload document icon, which is labeled as upload transcript.

II. This calls the view `transcript` in `backend/views.py`

III. The user then uploads a pdf of their transcript.

A. The form ensures that a pdf is uploaded by checking the file extension.

B. If an unexpected pdf (Not a transcript) is uploaded python will raise an exception.

IV. `pdf2text.py` is imported and the function `covert` is called to get a text output of the file in the following format.

A. [Mnemonic] [Course Number] [Credit Hours] [0 or 1]
e.g. `CS 1110 3 1`

B. A zero means that a student earned below a C- in the course and it will not be selected to be included as a satisfied course by default.

C. Each course is separated by a newline

D. The output is read into the session and the text output is immediately deleted. The user's transcript is never saved.

V. The user is then redirected to the `edit_upload` view in the same file.

A. This generates a dynamic form of booleans and users can check or uncheck classes they want to add.
B. After submitting the form it will display a list of the courses that were added.

Appendix A: Screenshot of the Assistant's main interface