

Guide: Use, Maintain & Diagnose the Platform[Debugging & Deployment]:

Project Repository: https://github.com/gingebrigtsen/NMT_Schedule_Builder

Needed `FrontEnd/.next/`` directory: [here](#).

The front and backend production branch servers run at `localhost` (127.0.0.1) :3000`` and `127.0.0.1:5000`` respectively – these can be connected to the host domain URL in order to access the site at a larger scale and thru a cleaner address. The hosts/sysadmins will typically want to start the back-end first, as it has multiple threads and will take a little bit longer to start. This requires a pipenv environment, with the proper dependencies installed, as well as the Unicorn WSGI server service. Starting the back-end properly can be done by:

- Start the Flask-based back-end server using the pipenv environment, and install the proper requirements from the `pipfile.lock` using `pipenv install``.
 - Ensure that pipenv is locally or globally installed by installing it with pip.
- Next, start it with `pipenv shell``. Then, execute the properly configured WSGI Python server with `gunicorn -w 4 -b localhost:5000 app:app``.
 - Ensure that Gunicorn is installed and up to date in the pipenv shell, it can be installed with `pip install gunicorn`` and version-checked with `gunicorn --version``.
- Ensure the redis session management service is installed and running in the pipenv shell, with `sudo redis-cli ping`` and expect `PONG`` if working properly. This service is what maintains users' cart session data and must be running for Flask's configuration to work properly.

Next, when starting the front-end, it's a good idea to re-compile the build (in case of any issues or artifacts in the process of cloning from the project repository), then start the server using the native package manager, yarn. Here's how:

- If you need to install the project dependencies, use `yarn install`` this will use the existing `package.json` and `yarn.lock` to fetch all the dependencies and requirements specified.
 - Sometimes, yarn will erroneously install nested `node_modules`` folders inside packages within `node_modules``. These will stop the build from running or compiling.
 - They must be deleted before the front-end will work so if possible using the modules directly off the cloned repository would be the best way to avoid this issue.

- If you do need to find and delete them to resolve build errors, they're in `node_modules/@types/:`
 - `hoist-non-react-statics`
 - `react-beautiful-dnd`
 - `react-redux`
 - `react-virtualized-auto-sizer`
 - `react-window`
- `yarn build`
- `yarn start`

The back-end has a debugging route, a `Hello World` page that will display at the base URL if the backend server is working properly, and the front-end's base URL will render the index/home page of the website when fully up and running. A clean startup for the backend will look like:

```
(BackEnd) st-n9177@ST-N9177:~/work/NMT_Schedule_Builder/BackEnd$ gunicorn -w 4 -b localhost:5000 app:app
[2023-05-04 19:55:49 -0600] [722] [INFO] Starting gunicorn 20.1.0
[2023-05-04 19:55:49 -0600] [722] [INFO] Listening at: http://127.0.0.1:5000 (722)
[2023-05-04 19:55:49 -0600] [722] [INFO] Using worker: sync
[2023-05-04 19:55:49 -0600] [724] [INFO] Booting worker with pid: 724
[2023-05-04 19:55:51 -0600] [732] [INFO] Booting worker with pid: 732
[2023-05-04 19:55:55 -0600] [740] [INFO] Booting worker with pid: 740
[2023-05-04 19:55:58 -0600] [748] [INFO] Booting worker with pid: 748
[2023-05-04 19:57:26 -0600] [722] [CRITICAL] WORKER TIMEOUT (pid:732)
```

And a clean startup for the frontend will look like this (the next config warning is about exporting and can be ignored):

```
st-n9177@ST-N9177:~/work/NMT_Schedule_Builder/FrontEnd$ yarn start
yarn run v1.22.18
$ next start
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
info - Loaded env from /home/st-n9177/work/NMT_Schedule_Builder/FrontEnd/.env.local
warn - Invalid next.config.js options detected:
  - The value at .output must be one of: "standalone".
```

See more info here: <https://nextjs.org/docs/messages/invalid-next-config>

This process can be automated by creating a bash file e.g. `start.sh` that runs these commands properly according to the host environment. That generally concludes the process of starting the web platform. To stop or shut down either branch, the goto method is the keyboard interrupt, `Ctrl + C`. This will abruptly but cleanly stop the front-end server, and issue an `atexit` thread cleanup in the back-end before a clean shutdown. This shutdown protocol should also be automatable with a bash script written according to the host environment. Or the automation of

the web platform can be done using an external management library or software that SysAdmins are likely to already have access to in the host environment.

The back-end branch ``/BackEnd`` is relatively simple and stable compared to the front. The overhead ``app.py`` has all of the Flask, Flask-CORS, and Redis application and session configurations, as well as all of the routes/endpoints utilized by the front-end. There are also several subdirectories: Controllers, Models, and Services. For this project, the Controller branch is unused and unneeded – it's only included because it's an initial part of the boilerplate Flask project. The Models branch contains my Python scripts for scraping and parsing data from **Banweb**, which is then stored in a local subdirectory ``csv``. These scripts are controlled by an autonomous thread started and managed by the overhead application, and will run every 24 hours. Finally, the Services branch contains a Python script for some side/helper functions that are key to both the back-end endpoints and the front-end course lookup. They regularly check, and update configuration data based on information provided on Banweb. After extensive testing, these routes, services, and scripts seem to be working properly and stably and thus shouldn't need much in the way of diagnosis, interference, or maintenance. But in the case they do, my Python code is well vertically separated for readability, and also very well commented. This branch should be easy to work with.

The front-end branch is much more complex: it came from a starter kit and as such it has numerous modules and dependencies. That being said, I only actually worked on a handful of files and anything that I've touched is extremely well commented and additionally formatted by prettier to be easier to read. Those can be found in ``FrontEnd/src/pages/``; they're executable typescript files (.tsx) that utilize javascript, ElasticUI styling elements, and rely on the Next.js service and Node engine to run and render React pages. The Node packages, types, and modules can be found in ``FrontEnd/node_modules``, these are the custom web programming JavaScript files the front-end relies on. As further explained and itemized in Section 3 of the Appendix, there *may* be occasional issues with this branch, in that certain Node @Types can end up with a nested modules folder in them, and it'll need to be removed in order to build/start the front-end server. There are several other directories and loose files in the FrontEnd branch, but they're of less significance. Things like the public directory (for local media and styles), package.json, and yarn.lock requirement configurations, and also some generic scripts and documents that came

with the starter kit. Finally, there's the ``FrontEnd/.next/`` directory. This is the actual service that runs the production server, and it contains the static build (compiled) website pages, which will run and load much more efficiently and cleanly than on the development server. This directory is very straightforward but also problematic in that it's not available as a part of the GitHub repository. It contains numerous large files and will have to be transferred over an external medium such as [Google Drive](#), etc from myself to the hosts/SysAdmins in order to include it in the FrontEnd branch. That being said, the build in the repository theoretically *can* generate this directory and its resources using ``yarn build``, but to be safe I'd like to supply the existing build first which the host environment can then build on top of to correct any new issues.

There's also a significant issue in the build that the hosts/SysAdmins will need to resolve in a matter of weeks or months – once a particular package update is available. I contacted Alec Benson, former NMT CCoE project deployment manager, and we evaluated and discussed my primary issue, the `@FullCalendar-react` component, which works fine in development mode, but not in the build, in terms of styling/formatting. We tried several different versions of the package and other debugging options including reinstalling all node modules and changing the `.next.config.js` configurations. We scoured forums and documentation, tried tips, and I submitted a support ticket to the (relatively active) package management repository. We also noted a handful of existing other tickets and posts related to this exact issue, and the consensus is to wait a couple of weeks-months for the proper update to `@FullCalendar-react`. Until then, I can propose a handful of temporary solutions for the hosts:

- POSSIBLY: run the existing build with broken calendar formatting until the proper update becomes available and the front-end can be rebuilt
- OR: run your frontend server in the development branch until the update is available, at the cost of speed and cleanliness in page loading until the proper update becomes available and the front-end can be rebuilt
- OR: do not host the project until the packages can be updated properly and the front-end can be rebuilt
- THEN: when the fix is available re-install the modules with ``yarn add @fullcalendar/react@latest`` (being wary of the possibility of nested modules issues) and re-compile the front-end and host it properly.

I'm so sorry to pass on such an issue but it's essentially out of the scope of my control of this project so outlining this solution for the hosts is the best I can do to mitigate the risks to the quality and objectives of the website and project overall. I believe this will just be a temporary setback and not affect the platform in the long term.

Finally, it's strongly recommended but was unfortunately out of scope for me to handle, that both branches of the project, FrontEnd, and BackEnd, are containerized using a service like Docker. This would serve to support my projects' use of ``localhost 3000/5000``, make it easier to simultaneously boot or shut down, and connect to the host/public domain URL. Containers have never been a particularly strong skill for me but in my development experience at CCoE, they were extraordinarily useful for the ending phases of development (and though I've never done it myself, the SysAdmins require them for ease of deployment and security), making sure that the branches can talk to each other coherently, and are presented at the proper public URL. The previously mentioned bash scripts for automating running the platform could then be controlled by the docker container instead, as configurations support startup command sequences. With access to a better environment than my WSL2 development IDE, and more resources than my laptop, configuring this container, and compressing/formatting the project shouldn't be too complicated and there may even be good templates available. In conclusion, this would likely be the easiest way to adapt the project to and stably deploy it within the destination host environment, but I couldn't handle this task because I couldn't access the proper packages/templates to do so as they were behind a paywall.

For any other information, refer to the included Appendix and Attachments section below, which provides more direct steps with less narrative explanation, and some tips/documentation.

Appendix and Attachments [Code and Documentation]:

Project Repository: https://github.com/gingebrigtsen/NMT_Schedule_Builder

Needed ``FrontEnd/.next/`` directory: [here](#).

- 1. Front-End Code (excludes starter kit & components)[FrontEnd/src/pages:]**
 - 1.1. *header.tsx (The top bar, primary navigation menu for the website)*
 - 1.2. *footer.tsx (The bottom bar, secondary navigation menu for the website)*
 - 1.3. *index.tsx (The main/index/home page, the landing page for all users)*
 - 1.4. *lookup.tsx (The course lookup and results page where users select courses)*

- 1.5. *calendar.tsx* (The weekly calendar generated from users' cart of courses)
- 1.6. *about.tsx* (Largely static page about the purpose and uses of the website)
- 1.7. *help.tsx* (Completely static page, giving simple step-by-step directions, and site FAQ)
- 1.8. *report.tsx* (Simple report form on a plain page, for users to report glitches, technical issues, or other problems)
- 1.9. *404.tsx* (Completely static Error 404 graphic page, which offers a back button to return to the last page, though it still has the header and footer so users will still have navigation abilities)
- 1.10. *_app.tsx* (React Component template for the setup of each page, controls style, and `<head>` configuration for each page on the site)

2. BackEnd Code (Main files, excludes data CSVs and minor files)

- 2.1. *app.py()*:
- 2.2. *BackEnd/Models*
 - 2.2.1. *conf.json*: JSON formatted configuration for collecting and scraping from Banner, as well as populating search options from valid **Banweb** options.
 - 2.2.2. *CollectData.py()*: actually scraping **Banweb** utilizing the requests library, BeautifulSoup4, and html5lib. Generates an un-parsed CSV file, which is then passed into the parser
 - 2.2.3. *ParseData.py()*: parses broken CSV entries by fixing lines with missing data, secondary instructors, no CRN, etc.
 - 2.2.4. *DEBUG*: Also included in the project repository are isolated debug functions for dumping banweb HTML, collecting, and parsing smaller sections of course data. These can be run as standalone methods using ``python3 {script}.py``.
- 2.3. *BackEnd/Services*
 - 2.3.1. *requestService.py()*: background helper functions for the scraping and parsing processes, managed by discrete threads in the overhead application.
 - 2.3.2. Services are used to hold the actual Python scripts and methods the overall app and routes draw from and require – they're where actual computation tends to take place after being properly routed by the app or the controller.
- 2.4. *BackEnd/Controllers*
 - 2.4.1. *requestController.py()*: unused.
 - 2.4.2. Controllers are used to build more precise route blueprints, for properly finding and executing methods within the Flask back-end as a whole – typically for more intensive or less commonly used functions.
 - 2.4.3. No controller was necessary for this project; though originally I had all of my routes templated here, later on in development I concluded doing this was not necessary for the stack to run effectively.

3. General Guidelines for Startup & Use

- 3.1. Start the Flask-based back-end server using the pipenv environment, and install the proper requirements from the *pipfile.lock* using ``pipenv install``.
- 3.2. Next, start it with ``pipenv shell``. Then, execute the properly configured WSGI Python server with ``unicorn -w 4 -b localhost:5000 app:app``.
- 3.2.1. Ensure that pipenv is locally or globally installed by installing it with pip.

- 3.2.2. *Ensure that Gunicorn is installed and up to date in the pipenv shell, it can be installed with `pip install gunicorn` and version-checked with `gunicorn --version`.*
- 3.2.3. *Ensure redis is installed and running in the pipenv shell, with `sudo redis-cli ping` and expect `PONG` if working properly. This service is what maintains users' session data and must be running for Flask's configuration to work properly.*
- 3.2.4. *Note: Banweb Data is automatically collected and parsed every 24 hours.*
- 3.2.5. *Note: Updates to Banweb data are automatically checked every 24 hours. Term codes and subject codes will automatically be updated when new course data becomes available, corresponding to announcements on the Banweb page.*
- 3.3. *Ensure the yarn package manager is installed, `sudo apt update` then `sudo apt install yarn` (If the repository is already configured).*
- 3.4. *Even though the deployment-ready project stack will already have been built, it's a good idea to do it again, using `yarn build` from within the FrontEnd folder.*
- 3.5. *The project can also be exported to an extremely minimal production build that can be run with a Python HTTP server, for debugging purposes or if it's too resource intensive for the host environment. This can be done with `yarn export`.*
- 3.6. *Start the Node-based front-end using a terminal from using yarn to boot the server with `yarn start`.*
 - 3.6.1. *Note: front-end pages are accessible via `url/page`, whereas back-end functions are accessed via `url/api/{route}`.*
 - 3.6.2. *Note: watch out for peer dependency issues, the front-end has **a lot** and may need to be restarted if it overflows and stops working.*
- 3.7. *Navigate to the site in a web browser to ensure it's functioning properly.*
 - 3.7.1. *Note: All main pages, including the users' cart, are accessible from the top header. The footer, help, and about pages all offer supplementary pages and links for users in the case of an error or if they need more information.*
 - 3.7.2. *Note: Typically, the largest thing pages must load is the NMT logo, but it's been optimized to load fairly quickly.*
- 3.8. *This startup process for the front-end and the back-end can be automated by creating a bash file to run these commands – not included because I have no reference for the platform on which the project will be deployed. The sysadmins will need to either use their existing software management options or quickly put these commands together according to their environment.*