# Informal Technical Commentary

Produced by Edvinas Vrublevskis on 20th of May, 2021.

For project: Demand-Side flexibility visualization dashboard.

Located in: <https://github.com/e-exy/uom-ml-power>

## Initial setup for 1st time demo launch

Install Python 3.8 from <https://www.python.org/downloads/windows/>

From Command Prompt (Windows) run ***.\venv\Scripts\activate*** , to activate the virtual environment

Run ***pip install -r requirements.txt*** , in order to install all libraries to the virtual environment.

Afterwards, while still in virtual environment, run ***python index.py*** this will launch the demo dashboard.

In the web browser, access **localhost:8050/**

In order to terminate the server press **CTRL + C** in the Command Prompt

## Launching demo, after initial setup was done

While in project directory:

From Command Prompt (Windows) run ***.\venv\Scripts\activate*** , to activate the virtual environment

Afterwards, while still in virtual environment, run ***python index.py*** this will launch the demo dashboard.

In the web browser, access **localhost:8050/**

In order to terminate the server press **CTRL + C** in the Command Prompt

## Intro to project structure and key files

Folders:

./.git/ -> git commit history. (Disposable)

./\_\_pycache\_\_/ -> local cache for python environment. (Disposable)

./apps/ -> for future extensions of the application, not used.

./assets/ -> media files + dashboard styling file.

./data/ -> folder for local prediction data files in .csv format.

./docs/ -> relevant documents for the dashboard.

./venv/ -> virtual environment folder. Contains all the libraries for the project.

Files:

./.gitignore -> git related. For file exclusion from commit history. (Disposable)

./app.py -> server configuration.

./callbacks.py -> all interactivity and data processing logic.

./components.py -> HTML “components”. Everything related to each visual “part” of the dashboard.

./index.py -> index page structure, interactivity for the multi-page structure. This is the file to run.

./layouts.py -> HTML “structure” for each page. Uses components.py and combines into a page.

./Procfile -> For Heroku hosting setup.

./requirements.txt -> Library dependencies.

./runtime -> For Heroku hosting setup. Specifies Python version.

./Readme.md -> Short intro into the setup and etc.

## Information sources

As application is based on Dash framework. Well-written documentation can be found in <https://dash.plotly.com/>

Code is self-explanatory, as long as, core understanding of Dash framework is achieved.

## Current situation

Dashboard runs only as a web-based product.

Dashboard does not automatically communicate with Matlab. Therefore, each .csv data files must be placed manually in the ./data/ folder and code changed to reflect to existing data files.

## Technical explanation of the dashboard structure

Dashboard is run by executing index.py. This setups the navigation bar, menus, welcome page. Has a few styling properties applied for the interactive side bar and menu navigation. Create the multi-page interface. Key aspect – bottom lines contain app.run\_server(debug=False). In order to see the error’s in the dashboard or any debugging related info, debug=False, must be set to debug=True. In the render\_page\_content(pathname), each web address path returns the exact “layout” (layouts.py) for that page. Index.py also call the app.py server setup. In app.py a visual theme is used and is referenced in /assets/classic.css, .css file is generally a specific styling for any element in the dashboard.

Afterwards, for the exact “layout”, layouts.py is referenced and layout is returned. The layouts.py file specifies the text content as well as the interactive elements. Page structure is controlled here. Width of components is specified here as well. After an exact “layout” is called, it checks for any custom “components” to be used. These “components” are usually any visual and interactive elements, can be found in components.py.

Each “component” is similar to each “layout”. It specifies the “component” structure, provides element classes & ids for unique interactive elements and configures some small functionality for library provided “components”. Often, each component has a refresh interval – when to change the chart visualization for new data series or new users selection of dataset.

When a component is called (rendered), it’s functionality usually is created in callbacks.py by addressing the exact class or id for that component. In callbacks.py, the main functionality unit is a “callback”. Essentially, code for each callback is simply a logic/setup to do, before each refresh interval for the graphical elements. callbacks.py holds the hardcoded data file names. “Callbacks” operate by explaining why to execute the logic and where to put the results, known as Input(), Output() functions in each @app.callback() decorator. Input() can be refresh rate for the graphical element or user clicked choice & etc. Output() is where to put the results after the logic in the function is executed. First argument for each function is either class or id of a “component”, so that software knows what component to update. Each callback usually specifies what type of chart to show, different styling options are available as well. All in all, callbacks.py holds all of the functionality and is the key file for interactivity and functionality.

IMPORTANT, all styling is done is ./assets/classic.css and ./assets/styles.css.

Concluding this flow, the dashboard is shown and is updating the graphs at refresh rates or user interactions.

## Further comments on certain components

At the moment, all Matlab produced data in .csv files must be manually imported to ./data/ folder. Key point, that .csv headers are all hardcoded in the callbacks.py. In future, it would be good to have a unified naming convention for the exported Matlab results, so that any header would be identical, otherwise, a typo in the .csv header naming will break the code and show no data. Now, 2 datasets are hardcoded by hand, in future, python should scrape all possible .csv files in ./data/ folder and provide a dropdown selection option for the user to specify the dataset of the interest. This would be the scalable option.

The bus diagram picture lacks responsiveness and does not scale well, but provides the demo interactivity. Previously, a real map was used and bus points were mapped in that map. If this is to be brought back, use MapBox API, <https://www.mapbox.com/> .

Certain improvement must be done on the actual graphs. This is regarding the scaling of how many hours the user is interested to see. At the moment, it is a random time frame into the future. It can be seen by searching the x\_stamps variable name. For not demo purposes, these time frames must be improved by providing exact sizes of how much data we want to see at any given time.

## High customization, desktop software deployment, performance and web deployments

All “components” and etc are created in React.js, therefore if certain elements are needed that are not available in Dash. They can be added creating custom React.js components and adding them to the dashboard. For more: <https://dash.plotly.com/react-for-python-developers>

If dashboard to be produced as a stand-alone desktop software, it must be ported. As Dash is simply a combo of Flask server, React.js front-end, Plotly.js graphing libraries. It can be ported to desktop software using Electron framework (<https://www.electronjs.org/>), but there a few workaround on how to run Dash on Desktop (<https://community.plotly.com/t/convert-dash-to-executable-file-exe/14222/18>). Generally, as Dash is quickly developing, there may be even easier and better tailored solutions to port it as a desktop software.

As for the performance, Python is slow in general, but all other parts, React.js and Plotly.js are world leading elements for front-end. Therefore, performance can be improved by optimizing callbacks and improving the times of dataset accesses. For example, instead of reading .csv files, fetch all available .csv files into a local database (light-weight SQLite3 or similar) and fetch data from there. But due to the fact, that dashboard can be updated periodically (15min intervals or etc), anything above few seconds update times, make Python a very good speed choice for such data analytics dashboard. Therefore, easiest way to increase performance, is simply to host the dashboard on a powerful server.

This dashboard can be deployed to Heroku (<https://www.heroku.com/>) free of charge. This would let any user with internet access to access the dashboard. At the moment, due to an issue with local .csv files, some extra configuration in the Heroku related code had to be done, therefore it crashed. Although, Heroku setup files are present in the project and only the .csv file accesses must be resolved. It is a common issue, but due to deadlines, it was not resolved for this project.

## Concluding comments

The dashboard in the current condition on 26/05/2021 is of a demo quality. Basic and advanced Dash capabilities are used. Codebase structure contains a foundation for future work. Quite a lot of improvement should be done further in terms of data handling & visual aspects of the dashboard. For the successor of the project – “Read the Dash documentation to understand basic concepts. Then read it side-by-side with the code. Everything is self-explanatory and easily coded.”