**Plans – Progress**

**Plans**

For the software/frontend/backend, our initial plan was to use a “full-stack” framework, where the backend could also serve as a frontend to present the data from the sensors sent from the Arduino board.

The Django platform with Python was a no-brainer is does have full-stack capability, and its Model-View-Template design pattern and REST Framework is agile and effective for our prototype/proof of concept which need to meet a deadline for the assignment.

However, as we are going to be bringing across / rebuilding our team website, and being able to set our own REST APIs with our server, we thought that seperating the backend from the frontend would be a better approach, and would let us re-use the frontend to showcase our team, product and have the live data from the prototype.

We will be using Vue.JS framework in order to display a simple yet dynamic static website that can do Get Request to display the data from Django and then be published to Github Pages.

For the mobile app, as we needed to get a simple frontend app that could also display the data from the backend server, we’ve decided to go with Google’s Flutter toolkit.

As I do not own an Apple developer license or Apple machine, the app will be compiled for Android as an .apk file.

**Progress**

*Backend*

Starting with the backend environment, I started by making sure my Python3 install was up to date (3.9.6 at the time of writing).

I am using a Linux machine to do the development, which should help as installing modules and the framework is fairly simple through the terminal.

Best practice is to set a virtual environment to develop the server, so I’ve installed venv and started this through a terminal session.

My IDE of choice is Visual Studio Code as it is lightweight and has Plug-Ins for Django, to help with coding.

Once my virtual environment and Django was installed, I could proceed to creating a new Django Server project, which generates the files for the server itself.

At the top of level, manage.py is the Python script which executes Django to kick start the server. This is generally not edited as it contains the automation to publish the server on the machine.

Within this high level folder contains an automatically generated SQLite server by default.

For this proof of concept, this was sufficient as we’re only receiving data from 3 sensors on the Arduino Board. On bigger deployments/projects, the most popular relational database paired with Django with either PostgreSQL or mySQL.

In the Server folder, the crucial part is settings.py as it dictates all of the apps/modules used by the server, contains the unique secret key used for the server as well as parameters regarding the database.

Urls.py is also an important file as it contains the url routing for the pages, and the location of additional apps added to the project.

By default, an administration “backend” page is generated by Django and we can run a command to generate a new SuperUser.   
This SuperUser has access to create other users, access any data models set on the Database and add/remove data easily without having to do manually SQL queries.

Now that we have the main server running, we need to add an App, which will serve as our entry point for the web server, so the board can post data to Django and in return will store it in the database.

Once we generate the app from a command with Django’s module, this generates a new folder within the project with the python files needed for the app to run.

At this stage we need to make sure that settings.py is amended to have the new app listed in the Installed Apps array, so the server knows of the existence of this app.

Within the app folder itself, there are three important parts which makes Django’s Model-View-Template software design pattern:

- Models.py is where we store the different data models and how they’re returned.

For our prototype, I’ve set up a Temperature, Humidity and Motion class.

Temperature and Humidity originally were set with integer values, but we’ve found with Jason that the Arduino board would return decimals for both values, so this has been amended to be a standard field that can take both values. Motion is a boolean value, which takes true or false.

- Views.py takes the data models that we have defined and we re-create objects from them and choose how we want to manipulate the data. (Hence importing the classes from the models module)

In here, we have an index function for the main page of the server, where we can manipulate the objects of the different classes that we have set. For instance, I’ve got a temperature objects variable which will be an array of all the temperature records, sorted in order by their date of creation. And I also have a temperature\_latest variable which only retrieve the last entry in the database for Temperature.

- Templates folder which contains the html pages. Django uses this to display the frontend part of the server, with its Templates framework, and using a mix of HTML and its own markup code to render the data from Views and also have a working webpage that can have more than just programatically set content.

It is also worth mentioning that the Urls.py file here is set to define the diferrent URL path for the website. As this is the API app, I’ve set this to be showing up as ‘/api’ so it is away from the main page.

Out of the box, when creating a Django server or app, this doesn’t come with the REST Framework pre-installed by default, although it is a fairly straight forward process to integrate this in the app.

At a high level, we leverage the universal REST API frameworks in order for the data in the tables sitting on the Django database to be used/read by any frontend by serialization and convert the data in a JSON format. In short, Django handles all the transaction/queries to the Database, and when a Get request arrives, it will have the information ready to go on a silver plate for the requester to ingest.

For the REST Framework to be in place, we need to install this module through the Terminal as a Python PIP package. Once we have the virtual environment loaded with this package, we can proceed to updating settings.py so the server knows that we’re going to be using it.

Django also has a migration command that will take care of making sure the project is updated to reflect the new package being installed and used.

Once the project is updated, we can start serializing our models within the API App so they can be used in POST/GET requests. We add a new Python file called serializers.py and import the model classes and get them converted.

The next step is to update our Views so we can determine what can be queried from the API and how we want to present this.

For example, the Temperature class becomes an endpoint and when a Get request hits ../api/temperature, the JSON returned will contain the id of the record, the temperature value itself and the date of creation of this record.

The last step is to update URLs so we can set the location where we want our API to be accessed.

Now that the Django backend server is configured, we’ve made a basic landing page for testing purposes so we can check that new records are being added and we can display the data for all the sensors from the board.

After doing the test and confirming that connectivity and API calls were working, I’ve used PageKite service in order to host my server on my personal hardware have a tunnel forwarder so it is published on the PageKite domain online.

It’s a simple Python script that links up to my PageKite account and forward port 8000 of the web server on my computer and can now be accessed via http://ricjouas.pagekite.me

*Frontend*

To set up Vue.JS, the process is similar to Django where we both install the Vue CLI and Node.JS prerequisites.

Vue runs in conjunction with Node.JS to generate its app.

Once we have generated the project, which sets up the folder structure and adds the Node dependencies, we can start working on building our skeleton base for the website.

The Vue framework is template driven, and borrows most of the HTML markup language, but adds its own twist to enhance the elements and get them to be more dynamic.

App.vue, which contains the entry point for the app, contains the high level structure of the website.

We separate each part of the landing page into sections, which are called components.

This approach lets us have one page and a navigation system that scrolls through to get to the specific section upon choosing a menu option.

We then route all different components in this App.vue file, so it knows how we want to define the page. A few JavaScript elements have been added to handle the page interaction.

As it is a large framework, it supports adding additional packages/libraries, and for the NodeSpider website, I have included both Axios and Vuetify.

The Axios package is imported so we can generate HTTP Get Requests within the Frontend, in order for data to be retrieved from our Django Backend server.

We call this package with a script element and once we build the HTTP request with the proper headers and authorization tokens, we bind it to a variable, which can then be passed on our frontend element as a string.

The challenging part here was to make sure we could get the right information from the endpoints set in the backend’s API.

We had to re-edit the Django deployment so it was easier for the Vue App to get, for example, the temperature instead of getting the entire JSON response, which includes the Record ID, Temperature data and the date of creation.

The Vuetify package works as an extension of Vue, and adds extra theme-ing such as Material Design to make our Website feel more modern and fast to scroll through.

This has helped make card elements such as the sensor data really become alive and interact well with the page with smooth transitions and animations.

Once the coding has been completed with the app working seamlessly with its components, we proceed to publish the App to Github Pages.

The process starts by hooking up the Vue App Project through to our Github Team Repository, but running Git commands.

(This setup also helps doing commits directly without having to use Github Desktop)

Once this is complete, publishing the App requires a Node.JS script, which will automate the build/publish process.

The script sets up the GitHub Pages branch and compiles the development app into a production app ready to be used as a static site.

*Mobile Application*

As both the Web Backend and Frontend setup and testing took the majority of the development cycle to make, in order to have a working prototype / proof of concept for the assignment, we couldn’t achieve the mobile application on time for the deadline.