Overview

"So what's Django?"
Django is a web framework; designed to simplify and facilitate the rapid development of web applications.

"But what's a framework?"
Frameworks provide a foundation for which developers can use to build upon to create their own applications.

"Is that not what a library does? I import a library and then I'm able to use modules within that library that facilitate the development of my code?"
Frameworks differ from libraries. With libraries you control the flow of the program; you call a module, it executes and upon completion the next line of your program executes. With a framework, the control is inverted. The framework has controls the flow; it calls your code rather than your code calling it. The framework provides the structure whilst you provide the detail of what your application is going to do.

Make sure you've successfully followed the steps in README.md and managed to start an instance of the server before continuing.

Structure

Django is split into apps - self contained packages that should only do one thing - an example of modularisation. At present, there is only one app that we've created for this project - `api`, however, we may add additional ones such as for the frontend and ML, depending on how we incorporate these two elements (not important at present).

Any work you do will take place in the `api` app (`api/`). There are a few files that you should familiarise yourself with:

- `api/models.py` - Each class represents a relation in the database.
  There are two aspects to each class.
  - The attributes for each model (attributes correspond to attributes in the relation)
    - The format for defining a models attribute is `VAR = DATATYPE`
    - `VARNAME` will be the attribute's name in the relation
    - `DATATYPE` will be it's data type - a complete list of data types available can be found here: `https://docs.djangoproject.com/en/3.0/ref/models/fields/#field-types`.
  - We've also included a nested class, `Meta`, inside each model. This allows us to:
    - Specify the relations name in the database
    - Enforce unique constraints for attributes which aren't primary keys using `unique_together`.

    Things to note:
    - You can't have composite primary keys. A workaround is to create an autogenerated id and introduce a `unique_together` constraint.
    - If you make changes to a model, you'll stop the server (*control-c*) and run:
      `python3 manage.py makemigrations` followed by:

`python3 manage.py migrate` and then restart the server via:
`python3 manage.py runserver`
Your changes will now be reflected in the database.

- `api/serializers.py` - Each class represents a serializer. Serializers allow for query sets from the database to be converted to native Python data types so they can be rendered into JSON.
  - Serializers help us get our data our of the database. For each model you're going to need at least one corresponding serializer.
  - Each serializer contains a nested class `Meta`. There are only two attributes you need to specify in order to create a serializer:
    * The model on which the serializer is based
    * What fields your serializer is going to contain - you can specify `'__all__'` for all attributes of the model or specific attributes using a tuple e.g `('fieldx', 'fieldy', 'fieldz')`.
- `api/views.py` - Views take a web request e.g (GET, POST) and return a response.
  - Due to an additional framework we're using - Django REST framework, creating views is simple.
  - At the moment we only have views which return data using GET requests, but we're also going to need views to insert data using POST requests.
  - Each class represents a view. Methods within the class should correspond to HTTP methods e.g `get`.
  - For GET requests, the structure is always as follows:

```
def get(self, response, argx, argy, ...):
      data = <MODEL>.objects.filter(<MODEL_ATTRIBUTE>=argx, ...)
      s = <MODEL_SERIALIZER>(data, many=True)
      return Response(s.data)
```

  - Essentially, you filter a model's objects on specific attributes (or use the method `all`), use the serializer to convert the query set and then return this data.
  - There are no examples of other HTTP methods in the code apart from GET, but the examples here are transferable - just change the variable names: `https://www.django-rest-framework.org/tutorial/3-class-based-views/`.
- `api/urls.py` - Links a url path to a view (registering an endpoint).
  - In order to access a view, you need to add a path.
  - There are numerous examples already and when adding new paths try to stick to the existing format.
  - For parameters in the URL the syntax is `<<TYPE>:<ARG>>`. For each parameter, your view methods should have a corresponding parameter.
- `api/test.py` - Automated testing of our API.
  - At present we don't have any automated tests. However, having had a quick glance, this seems to be a good resource on how to create them: `https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing`.
  - If you're accessing the API from the client side, you should find the script in `access/query` super useful. The API documentation shows exactly how to use it.

- api/admin.py - Allows for data manipulation via the admin panel
  - Once you've created a new model, you should register it in here.

Key Commands

- `python3 manage.py runserver` - Run an instance of the server
- `python3 manage.py makemigrations` - Create the migrations (SQL commands to be applied)
- `python3 manage.py migrate` - Run the migrations (execute the SQL)
- `git add .` - Add all files in the current directory recursively (make sure you're in the root of the repository for all project files to be added)
- `git commit -m "<MY COMMIT MESSAGE>"` - Saves the changes with a message
- `git push` - Push the changes to the remote repository (your fork on GitHub)
- `git pull parent master` - Updates your repository with any changes that have been make by other group members

In orders for others to view your changes after you have pushed them to GitHub, navigate to the repository's page on GitHub, click 'New pull request' and then 'Create pull request'.

Further Comments

We need to maintain the documentation for additional API endpoints. The source documentation file is located `backend/docs/api/source/index.html.md`. Simply copy and paste an existing entry and edit it to suit your additional endpoint. Changes you make to this document won't be reflected on `http://localhost:8000/docs/` as the webpages will have to be rebuilt - don't worry about that.

Most of the structure is already in place, certainly in regard to creating endpoints, therefore, in the majority of cases it's just copying existing code and manipulating it.

Good Luck - Let me know if you're having any issues (I'm more than happy to help)!