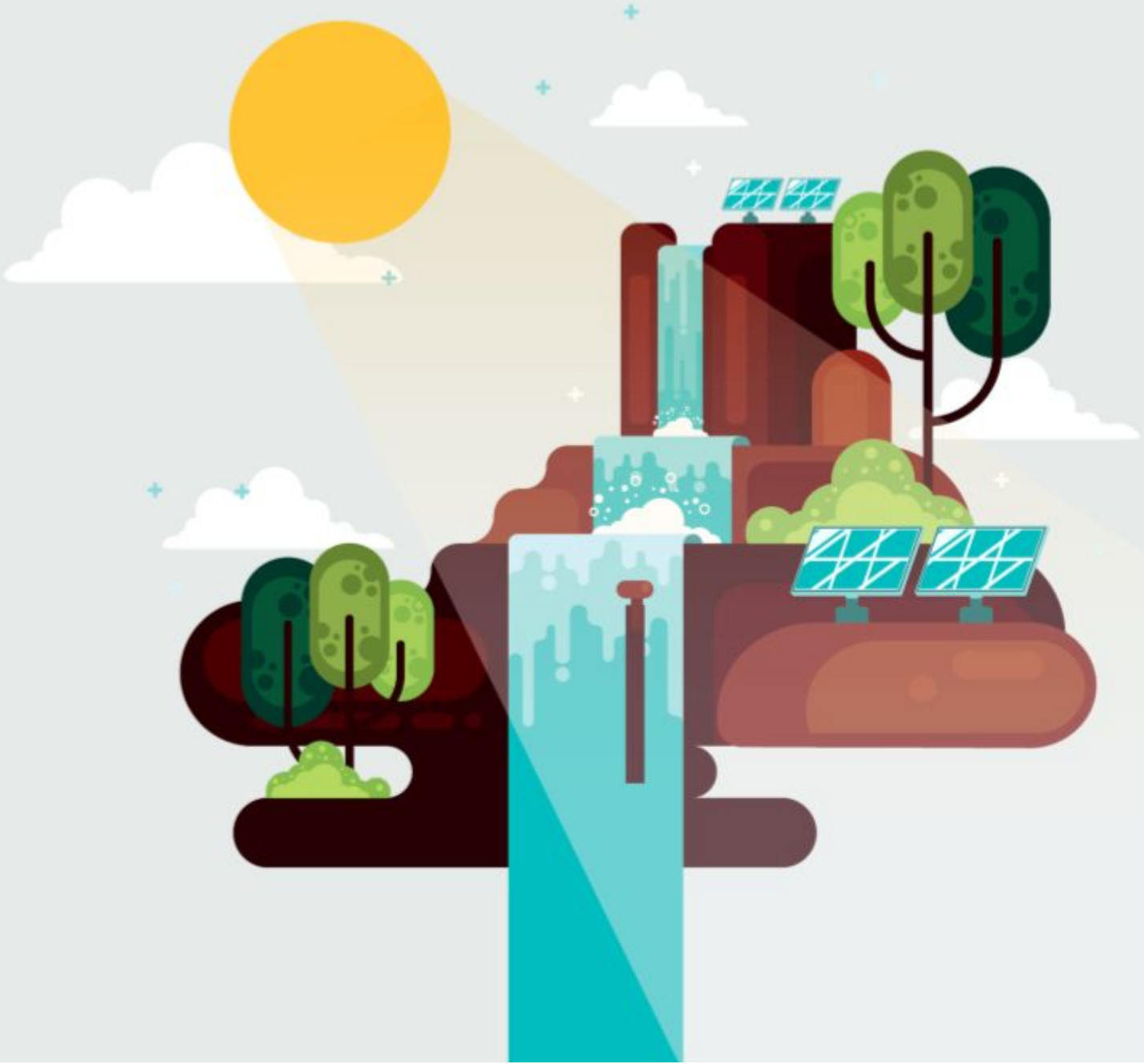
greendeal report

Application Setup

CSCI E-599 Harvard Extension School

April 20, 2020

Team Members:
Alan Martinson, Carly Gloge, Hemant Bajpai
Mark McDonald, Pritam Dey, Taylor Meyer



Index

Overview

The EU Green Deal application described in this document is designed to collect data from a variety of sources and present consolidated results in an easy-to-understand manner. Although the data collected is scientific in nature, the application is intended for non-scientific users to understand trends and patterns specifically in relation to environmental issues and more specifically in relation to the EU Green Deal Project.

The EU Green Deal provides a roadmap for environment improvement through a roadmap of actions is an EU initiative that sets a variety of targets that are aimed at becoming climate neutral by 2050 by pursuing 4 main goals:

- Zero pollution
- Affordable secure energy
- Smarter transport
- High-Quality Food

The application described in this document is a prototype application that focused on the “Zero Pollution” target by reporting air quality measurements and comparing them to targets set forth in the Green Deal agreement.

The technical structure of the application is based on a Django Web Server (which is based on Python). Data for the application is stored in a PostgreSQL database which can be hosted in a local Docker container or any cloud service of your choice.

Knowledge of Django (and Python) is strongly recommended before attempting to extend the application; however, the application can be downloaded and run locally without any knowledge of Django.

Installation

The following steps describe the detailed steps necessary to install the application locally and prepare it for extended development. If you run into problems during any of these steps and simply want to run the prototype application, you can run the entire application as a Docker container locally. See Run Docker Image (Optional) for details.

IMPORTANT NOTE: This application was developed on a Mac OS platform and has been tested to work on a Mac and should also work on Linux. Installation on Windows may require special handling of the GeoPandas installation. Please refer to the documentation for GeoPandas installation on WIndows if you run into a problem installing the dependencies using Windows. Using the Docker instance of the application should be possible to run on any

platform that supports Docker. To do this, follow the instructions under Run Docker Image (Optional).

Clone Repository

Access to the repository is restricted and not public. If you do not have an account which can access the application, please request access via email to mcdomx@gmail.com. Please include a desired user ID for the account that will be created for you. Once you have an account established, you can clone the repository using:

```
> git clone
https://bitbucket.org/capstoneeureporting/eugreendeal/src/master/
```

Install Dependencies

Python

The application requires Python 3.7. Please refer to the Python installation documentation if you do not have 3.7 installed locally. To check your Python version:

```
> python --version
```

Any 3.7 version will work.

Install pip

You're welcome to install required packages using other installers, but we recommend using pip. Instructions to install pip can be found here.

Pipenv

The application uses pipenv to manage dependencies. To use pipenv, you will first need to install the python module:

```
> pip install pipenv
```

Psycopg2

Check to make sure you have the PostgreSQL adapter installed:

```
> pip freeze | grep psycopg2
```

If it isn't installed already you can install it using one of these commands:

```
> pipenv install psycogp2-binary
```

ALTERNATIVE: If you have issues installing via pipenv, you can install with:

```
> brew install postgresql
```

SpatialIndex

SpatialIndex is a library used to calculate data that is rendered in some of the visualizations in the prototype application.

```
> brew install spatialindex
```

Docker

The initial setup will be done using a Docker database. This requires docker to be installed locally. Instructions for installing Docker can be found at https://docs.docker.com/get-docker/.

Setup Local Database

The application database can be installed using a local Docker Postgres container.

Navigate to the application '../eugreendeal/infrastructure' directory:

```
> docker-compose up -d db
```

The first time this is run, the proper container will be downloaded. This may take several minutes.

Activate Environment

Activate the pipenv environment for the project. This will install all of the required packaged from the specified within Pipfile.lock

From the '../eugreendeal/ directory':

```
> pipenv install
```

When prompted, enter the following to activate the virtual environment:

```
> pipenv shell
```

Delete Existing Migrations

Delete all the Django migrations from airpollution/migrations. Leave the __init__.py file in place. The migrations are located in the '..eugreendeal/airpollution/migrations' directory.

Setup Tables in Local Database

Once the Docker database container is up and running, the application's data tables can be created:

From the ../eugreendeal/ directory:

```
> python manage.py makemigrations
> python manage.py migrate
```

Create a Superuser

As a superuser, you have the ability to navigate to table views in order to see the data stored in the application.

From the ../eugreendeal/ directory:

```
> python manage.py createsuperuser
```

Follow the prompts to set up a username and password.

Create a Default Persona

The application needs a default persona for a non superuser to be assigned a persona.

To do this, we need to create a default persona if this is the first time the application is being initialized.

From the ../eugreendeal/ directory:

```
> python manage.py createdefaultpersona
```

Launch Application

At this point, you can launch the application to see that it is functioning properly.

From the ../eugreendeal/ directory:

```
> python manage.py runserver
```

Navigate to the site at:
http://localhost:8000/

To see the Django administrative console, navigate to:
http://localhost:8000/superadmin/

IMPORTANT NOTE: At this point, you have not populated the database with any data so you'll get a number of errors if you try to load in plots. Please follow the instructions to load in data below:

Load Data

The application relies on a large amount of data to provide meaningful analysis and reports. For testing, a limited set of data can be loaded.
First, shut down the application using (Ctrl + C). Then run the following command

```
> python manage.py populate_db
```

IMPORTANT NOTE: this process can still take a few hours to complete. Please be patient.

Run Docker Image (Optional)

If you face any compatibility issues while installing any dependency or due to some reason you are unable to bring up the Django server, you can opt to build a local container and run it as a docker image along with the docker postgres instance. To do that, navigate to the project home directory and execute:

```
> docker build -t eugdserver .
```

Once the build is complete, you can bring up the entire infrastructure by naviagting into ../eugreendeal/infrastructure directory and executing:

```
> docker-compose up -d
```

To bring stop the docker instance:

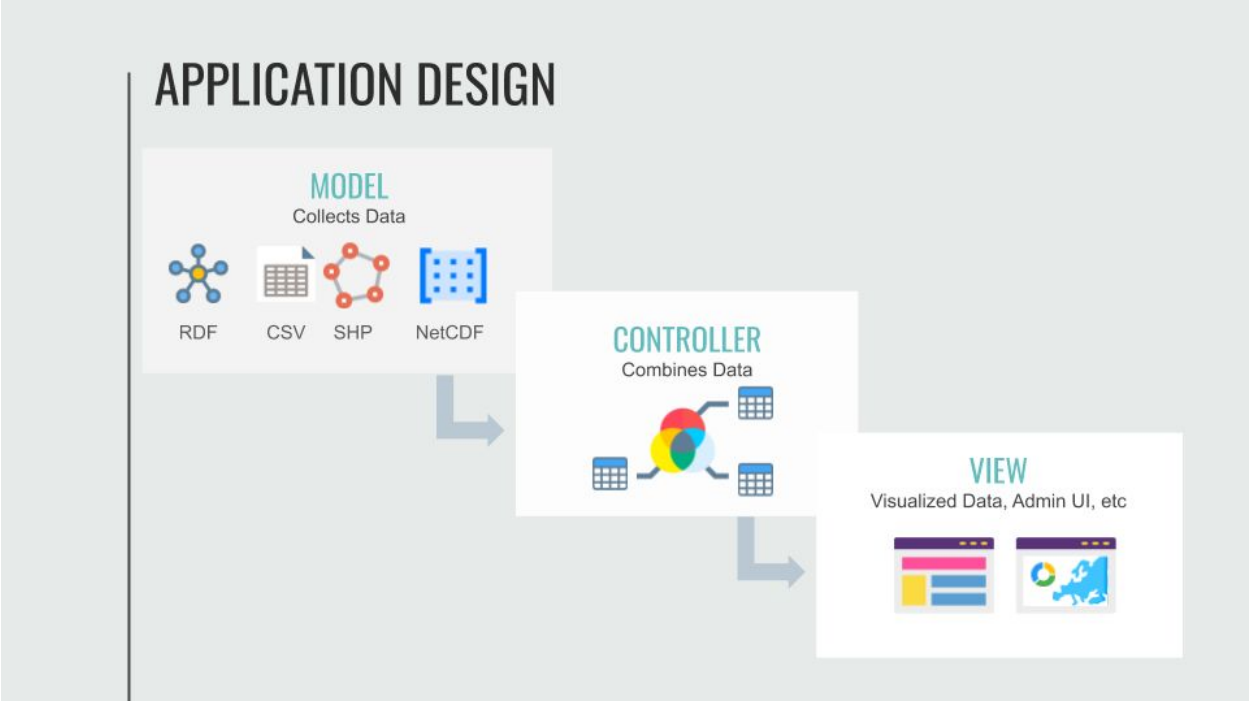
```
> docker-compose down
```

Architecture / Extending Functionality

The application is based on a Django Web Server which is python dependent. The application allows for the extension of new data sources and visualizations. Knowledge of Django is highly recommended and knowledge of Python is essential to extend the application.

Architecture Overview

The application follows a traditional MVC design approach.



Extending Application

Extending the application requires new components to be built which include:

- Creating DB tables for new data
- Collecting and loading external data
- Creating accessor functions to access the table data
- Creating API functions to consolidate data for front-end use
- Creating visualizations
- Placing the visualizations on web pages

If new data is not necessary, then only new visualizations must be created and you can start at the Data Access (API) or Views section of this document. Data for your new visualization may be readily accessible via existing API functions, so you may not need to create any functions to access data specific to your visualization. It is advisable to start with the Views section and go back and add API functions as you find that they are necessary to build your visualization.

A summary of the extendable classes are listed in the table below:

Note: Examples for each class are included in the prototype application.

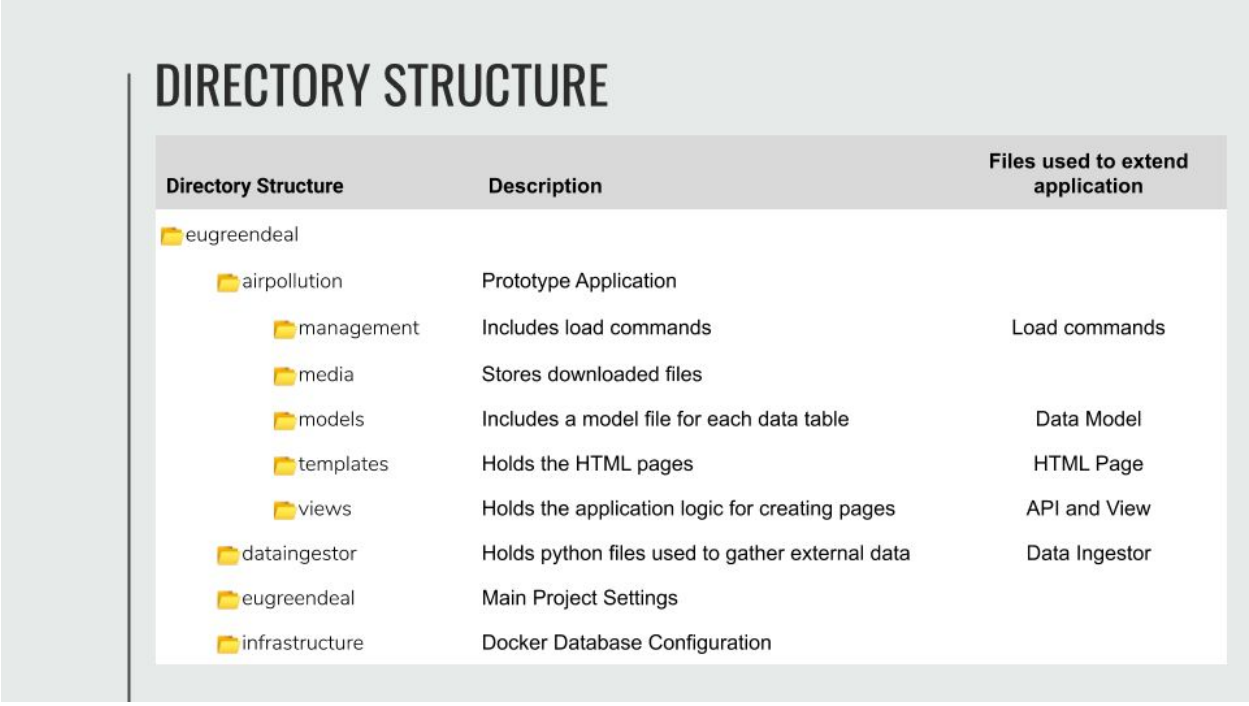
APPLICATION DESIGN

The following components require configuration to extend the application to collect new data and present it on a web page.

Model Collects Data		**Controller** Combines Data		**View** Presents Data
Data Ingestor	Data Model	API	View	HTML Page
Handles the specifics of communicating with the external data source and brining the data to the application. A new class is created for each data source which extends the DataSource.py class A Django loading process can be triggered by including a Command function in the 'management' directory.	Django uses an ORM wrapper for database tables. For each table, a new data model must be created. Each model extends the Django models.Model class. Each model class also includes basic data accessors for data in the model.	An API layer consolidates data from one or many Data Models into simplified functions that can be used by the front-end components. Functions in the API layer should call the data accessor functions included in the model layer. An initial API is included in the prototype application in the ' directory.	Each visualization corresponds to a vews.py file stored in the '../eugreendeal/airpolluti on/views/' directory. Functions called in these files should return JSON objects that can be handled by a Django webpage which will use Jinja to render the data.	Pages are built using Jinja templating. New visualization can be created in a templated html page which can be included in a respective dashboard.

Django uses a predefined directory structure to store files of types. Subdirectories under each directory can be created, but files of similar types must be kept under the 'model', 'views' and 'templates' directories. This is important as Django will look for files under certain directories to perform application management tasks.

A list of the directories used is presented below:



DIRECTORY STRUCTURE

Directory Structure	Description	Files used to extend application
eugreendeal		
airpollution	Prototype Application	
management	Includes load commands	Load commands
media	Stores downloaded files	
models	Includes a model file for each data table	Data Model
templates	Holds the HTML pages	HTML Page
views	Holds the application logic for creating pages	API and View
dataingestor	Holds python files used to gather external data	Data Ingestor
eugreendeal	Main Project Settings	
infrastructure	Docker Database Configuration	

Data Model

If a new presentation of data requires new data to be externally collected, a new data model must be created. This will likely require some knowledge of how Django's ORM works, but you can follow the templates created in the prototype application to cover most cases.

Step 1:

Create a new model class python file in the '../eugreendeal/airpollution/models' directory.

Step 2:

In that file, create your new class inheriting from the Django models.Model class. (use examples from the base application as a template.)

For each field that will be used in the application, create a variable from the models class.

Step 3:

In the new class, create data accessor functions which the API will use to get data from the new model. These functions should specialize in collecting summarized data efficiently from the model. No accessor functions should

access other models. Functions that combine data are in the API layer described later (see Data Access).

Data Ingestion

After a model is created, the logic for collecting external data can be started. The design of this task will vary depending on what datasource you will use and how that data is accessed. Of the steps necessary to extend the application, creating a process to load external data is likely to require the most time, effort and skill.

A new class is required to handle the data collection task. This new class must inherit from the DataSource class which can be found in the '../eugreendeal/dataingestor/' directory. The DataSource class is specific to this application and not a Django class. The DataSource class only requires a `load_data()` and a `load_dummy_data()` function to be implemented but you can, and should, create others to build the process.

These functions are expected to execute the process for collecting external data and saving it into the application database. The `load_data()` function can accept a 'kwargs' dictionary argument allowing flexibility for accepted arguments. These arguments will likely be necessary to set parameters for restricting the data load to a specific date or for only certain elements.

Since the variety of datasources is extremely large, no specific instructions are included but a variety of working examples are provided with the prototype application.

Data Access (API)

In this section, we review the API layer and its purpose.

Access to each data source's table data is created in the model (see Data Model). Those functions are basic data accessors and provide a benefit by making fast data queries without requiring the API to know the detailed model structure.

The API layer provides intelligence for the application. Here, data may be combined from several calls to one of many data model accessors and organized for concise presentation to the front end. Functions in the API layer will be called by views, each of which are designed to serve a specific visualization (see Views). The flow of data requests is simplified below:

Front-end HTML ⟶ **my_view** ⟶ **the_application_API** ⟶ **my_model(s)**

The prototype application includes an API script in the 'views' directory called 'aq_api_v1.py'. Existing functions can be re-used or additional functions can be added. You may also create a new api.py file to organize your work more effectively.

Creating an API function is dependent on the tasks necessary and follows no specific formatting requirements in the application. The main purpose of the API layer is to prevent any views from needing direct access to the data model objects as well as preventing any views from needing to combine data from various functions. The primary guideline to follow is that API functions should use model accessor functions to make calls to the database. Generally, the API should not include functions that directly call a database table.

Views

If the application already has the data you need, you may be able to start at this step for extending the application. If you find that you need a new API function during this step, new API functions should be created (see Data Access).

Views are a Django term and refer to Python scripts that collect data and render a web page or a component of a web page with that data. For the purposes of extending this application, each new view should represent a new visualization and should collect the data necessary for the visualization.

Step 1:

Create a new views_<name of view>.py file in the 'views' directory. The contents of this file should be a named function that will be called to render your visualization. The function should return accept a single argument of 'request' and return a 'render' object.

```
def my_visualization_view(request):
    ..logic and code
    return render(request, 'airpollution/my_vis.html', dict(data1='my_data', data2='more_data'))
```

This will render the page passing the dictionary of data into the page which can be accessed using jinja templating.

Alternatively, you can return a JSON object if your page uses javascript to render the page.

```
def my_visualization_view(request):
        ..logic and code
        return JsonResponse(dict(data1='somedata', data2='moredata'))
```

Step 2:

Create a new HTML file that will host your visualization. Copy the 'visualization_template.HTML' file as a starting point.
This new file should only include the javascript and HTML tags specific for the visualization you are creating. This new HTML file inherits from the site's layout and navigation design.

Step 3:

Create a route to your visualization.
During development, you may want to see your progress. To create a route to your visualization without rendering an entire dashboard page, you can create a route in the 'eugreendeal/airpollution/urls.py' file by adding a route to the 'urlpatterns' list. Follow the logic presented in the template. Your added item should follow the pattern below:

```
path("my_route_name", views.my_visualization_view, name="my_visualization_name"),
```

You will be able to reach your page at:
http://localhost:8080/my_route_name

Later, this route will be used to place your visualization into an existing dashboard.

Step 4:

Once the visualization is completed, it can be integrated into a dashboard page. Select the dashboard page to include the visualization into and find the location for your page based on the dashboard layout. Insert the call to your page's route where applicable.

New visualizations are quite simple to insert into HTML. First add a div with a unique id to reference. This is where your visualization will be placed. For example;

```
<div id="reg-rep-ch-1"> </div>,
```

The div is populated using a JavaScript function in the dashboard page. The following `update()` function is called in the dashboard page when the page is first loaded and when the page's selections are changed. The function performs the following:

- Selects the specified div in the page
- Initially places a a preloader gif image in that div

- If it receives a successful JSON response from the API, then it populates the div with the Bokeh plot embedded within the JSON response. Note that the API function in the example below is called from the route `pollution_over_time` . This route must be included in the `urls.py` urlpatterns list.
- If the JSON response throws an error, it places an error image within that div

```
function update(country) {
   var chart_div = document.querySelector('#reg-rep-ch-1');
   chart_div.innerHTML = "<img style=\"height:300px;width:400px\" src={% static
'airpollution/assets/img/gif/pre-loader-2.gif' %}>";

fetch('/pollution_over_time?pollutant=PM10&start_date=2020-02-29&end_date=2020-03-01&c
ountries=' + country)
   {#pollutant=PM25&start_date=2020-01-01&end_date=2020-03-22#}
       .then(function (response) {
           return response.json();
       })
       .then(function (item) {
           var chart_div = document.querySelector('#reg-rep-ch-1');
           chart_div.innerHTML = "";
           Bokeh.embed.embed_item(item, 'reg-rep-ch-1');
           chart_div.setAttribute('data-url',
'/pollution_over_time?pollutant=PM25&start_date=2020-01-01&end_date=2020-03-22');
       }).catch((error) => {
       var chart_div = document.querySelector('#reg-rep-ch-1');
       chart_div.innerHTML = "";
       chart_div.innerHTML = "<img style=\"height:200px;width:400px\" src={% static
'airpollution/assets/img/tech-snag.png' %}>";

   });
}
```

Once implemented, the above steps should yield your visualization presented in the browser window when navigating to the chosen route.