

## Week 10 - Django Web App

Last week you took three machine learning models and created simple Flask applications to expose their functionality via an HTTP interface. This week you will create a Django web application through which a user can interact with each of your team's three models. Beyond the web application setup, your team will implement a simple UI design to enable user interaction with your models, define application data models, run a containerized database, and define a data transformation/ingestion method. In the scenario of supporting a telemarketing campaign, your models have proven useful to management and now they want you to build a web application to support the marketing teams. This web application will make your models available to the marketing teams and should be easy for them to use.

### Learning Objectives

- LO1. Complete the steps to create a Django project
- LO2. Explain the difference between a Django project and application
- LO3. Apply the Django templating language to create a modularly defined web page
- LO4. Explain the benefits and drawbacks of a micro-services approach to web application development
- LO5. Explore the different HTML tags available to define a web page
- LO6. Explore the interaction between HTML and CSS
- LO7. Apply and then enhance a simple web-page mock-up

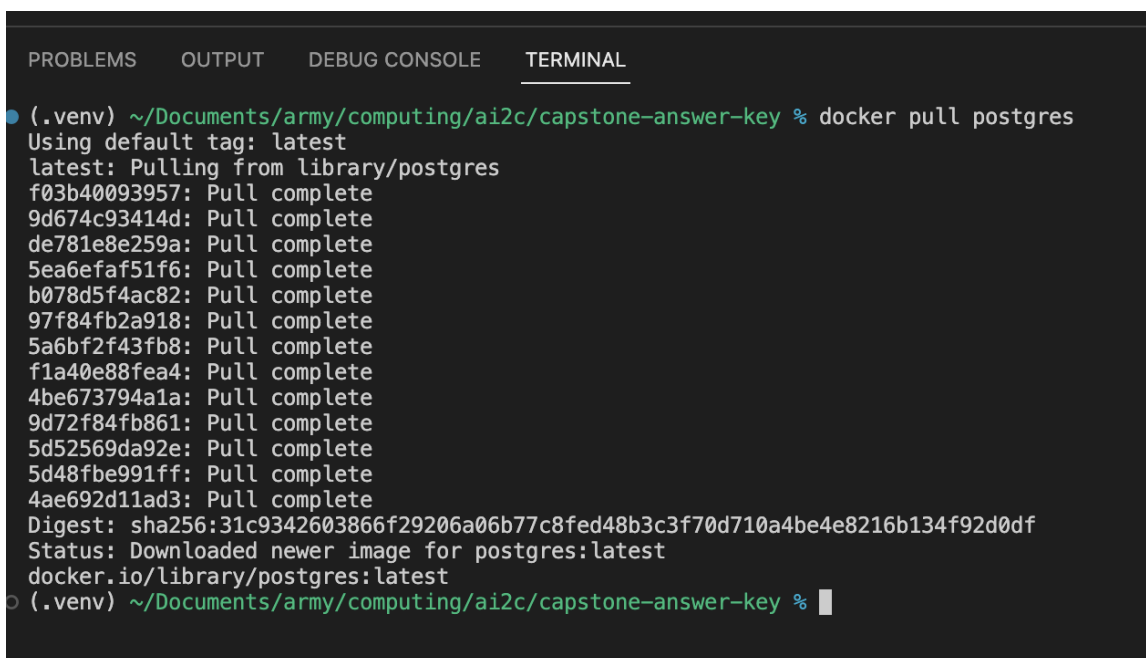
### Tasks

Instructions: Every member of your team needs to complete steps 1-6 on your own individual development machines. A single team member should then complete steps 7-13. Once you have completed this initial setup your team must work together to determine how you will complete the remaining steps for this week.

1. [Grp] Before you begin building your Django application every member of your team will need to install [Docker](#). You will use Docker to run a PostgreSQL database container that will serve as your application's persistent storage.  
Note: While not absolutely necessary for this week, you can also download the Docker extension for Visual Studio Code to assist with creating and editing Dockerfiles and exploring your local installation and images available.
2. [Grp] Open your capstone directory in VSCode. Ensure you activate your virtual environment and start Docker. Note: if you do not have docker running the error you see will ask if you have the "docker daemon" running. This simply means you do not have docker running and should start it.
3. [Grp] Pull [the official PostgreSQL container image](#) by opening the VSCode integrated terminal and running the following command.

```
docker pull postgres
```

If this succeeds, you will see something similar to the following output in your terminal.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
(.venv) ~/Documents/army/computing/ai2c/capstone-answer-key % docker pull postgres
Using default tag: latest
latest: Pulling from library/postgres
f03b40093957: Pull complete
9d674c93414d: Pull complete
de781e8e259a: Pull complete
5ea6efaf51f6: Pull complete
b078d5f4ac82: Pull complete
97f84fb2a918: Pull complete
5a6bf2f43fb8: Pull complete
f1a40e88fea4: Pull complete
4be673794a1a: Pull complete
9d72f84fb861: Pull complete
5d52569da92e: Pull complete
5d48fbe991ff: Pull complete
4ae692d11ad3: Pull complete
Digest: sha256:31c9342603866f29206a06b77c8fed48b3c3f70d710a4be4e8216b134f92d0df
Status: Downloaded newer image for postgres:latest
docker.io/library/postgres:latest
(.venv) ~/Documents/army/computing/ai2c/capstone-answer-key %

```

Figure 3.15: Successful Image Pull Terminal Output

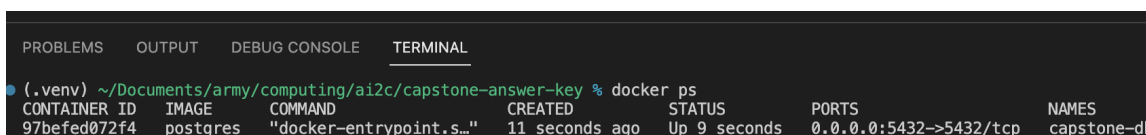
4. [Grp] Start a PostgreSQL container by running the following command. This command starts a postgres container named capstone-db in debug mode with the default PostgreSQL settings and specifies an admin username and password for the database the container will create. It also binds your computer's localhost network interface at port 5432 to the port exposed by the PostgreSQL container, 5432, making the database accessible to you on your local machine. You should note the username and password you use here for future reference, you will need it again soon.

```

docker run -d --name capstone-db \
-e POSTGRES_USER=<your_username> \
-e POSTGRES_PASSWORD=<your_password> \
-p 5432:5432 \
postgres

```

5. [Grp] Verify your PostgreSQL container is running with `docker ps`. If you were successful you will see the following output.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
(.venv) ~/Documents/army/computing/ai2c/capstone-answer-key % docker ps
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
97befed072f4  postgres  "docker-entrypoint.s..."  11 seconds ago  Up 9 seconds  0.0.0.0:5432->5432/tcp  capstone-db

```

Figure 3.16: Terminal output showing the container is running

6. [Grp] Once you see a running container, open the integrated terminal in VSCode and verify the PostgreSQL command line utility `psql` is able to view the database. Run the following command to list the databases available on localhost (the address you set your container to expose port 5432 on previously).

```
psql -h localhost -U <your_username> -l
```

Note: You must input the username you set previously in the previous command.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
(.venv) ~/Documents/army/computing/ai2c/capstone-answer-key % docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS                    NAMES
97befed072f4   postgres  "docker-entrypoint.s..."  11 seconds ago Up 9 seconds  0.0.0.0:5432->5432/tcp    capstone-db
(.venv) ~/Documents/army/computing/ai2c/capstone-answer-key % psql -h localhost -U postgres -l
Password for user postgres:
List of databases
-+-----+
Name      | Owner   | Encoding | Collate | Ctype   | Access privileges
-+-----+
postgres  | postgres | UTF8      | en_US.utf8 | en_US.utf8 |
template0 | postgres | UTF8      | en_US.utf8 | en_US.utf8 | =c/postgres +
           |          |          |          |          | postgres=Ctc/postgres
template1 | postgres | UTF8      | en_US.utf8 | en_US.utf8 | =c/postgres +
           |          |          |          |          | postgres=Ctc/postgres
(3 rows)

```

Figure 3.17: Databases available on localhost

The first database you see listed, `postgres` is the default name for a PostgreSQL database. Now that you have a database running, let's initialize a Django project that can use that database.

7. [Ind] Create and checkout a new branch.
8. [Ind] In VSCode, with your virtual environment activated, open the integrated terminal and install [Django](#) using pip.
 

```
pip install django
```

Don't forget to add `django` to your `requirements.txt` file, specifying the version you installed above.

9. [Ind] Start your Django project using the `django-admin` CLI tool. We will name the django project for phase 3 `tsa` (telemarketing support app).

```
django-admin startproject tsa
```

This command does a number of things for you. First, it creates a new directory with the name, `tsa`, in your `capstone` directory. Second, it creates a file, (`manage.py`), and a subdirectory, (`tsa`), within that directory. The `manage.py` file contains hooks to enable many useful web app management tools. You should not edit that file. The `tsa` directory contains the files necessary to define a django project. You'll work with the `settings.py` and `urls.py` files this week and will become more familiar with those soon. The `wsgi.py` and `asgi.py` files define django specific hooks for the [WSGI](#) and [ASGI](#) web server interfaces. These two tools are the primary ways a Python web server is defined. The Flask applications you created last week were WSGI-based applications.

10. [Ind] Before you do anything else, add a `README.md` file to your `tsa` repository. For now, simply include a title for this application and a short description of what role it will fill in this phase of the capstone.
11. [Ind] Before you run the development server, you must configure some critical [settings](#) so your django application can use the PostgreSQL database you have running. Open your `settings.py` file and find the setting named `DATABASES`. Configure this setting to properly reference your PostgreSQL database.

Hint: The django [settings documentation](#) tells you which fields are necessary for this configuration.

12. [Ind] To test your settings configuration, attempt to "make migrations". Migrations are a tool database administrators use to manage updates made to a database. Django automates this process for you using the `manage.py` file the CLI tool created when you started the project.

```
python manage.py makemigrations
```

If the command returns anything other than "No changes detected", you have no properly configured your database settings. Once you have run that command and view the above output, execute the migrations.

```
python manage.py migrate
```

This command performs the initial migrations needed for your application to function properly. In the `README.md` file within the `tsa` directory, include which applications applied changes to the database during this migration and a short description of what role they play in a Django application. (Hint: There are four applications you should see applying migrations initially)

Once all migrations have succeeded and you have updated the `README.md` file, verify your application works by running the development server.

```
python manage.py runserver
```

Open the URL in a browser and you will see a congratulations message display. With that complete, you have one final step before you can commit and push your changes.

13. [Ind] Protecting your application’s “secrets”. When you configured your database settings you likely put your database’s administrator username and password in plaintext in the settings file. These values should never be included in a your source code’s history and instead should be managed in environment variables or a `.env` file. For this project, you will use the library [python-dotenv](#) to manage your environment variables. Install `python-dotenv` using `pip` and add the version you install to your `requirements.txt` file.

```
pip install python-dotenv
```

Now, in the `tsa/tsa` directory (where your `settings.py` file is), create a file called `.env`. Put each of your database connection settings in your newly created `.env` file and update your `settings.py` file’s database settings configuration. When you have finished doing so, verify your development server is still working. If so, update your `.gitignore` file to ignore your `.env` file in your git history. Once you have done so, stage, commit, and push your newly created `tsa` directory to your team’s repository. Have a fellow teammate review your changes in GitHub and then merge them into your team’s `dev` branch. Your file system should look something like this at this point.

```
capstone/
├── .venv/
├── .gitignore
├── requirements.txt
├── README.md
├── marketing_pred/
├── cv_pred/
├── nlp_pred/
├── tsa/
│   ├── tsa/
│   │   ├── __init__.py
│   │   ├── .env
│   │   ├── asgi.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   ├── manage.py
│   └── README.md
```

14. [Grp] The remaining members of your team can now pull your team’s updated `dev` branch and you can all work together on your Django application moving forward. Once you have cloned the `dev` branch, create and checkout a new branch. If you did not complete the above steps (because one of your teammates completed the initial setup), create a `.env` file as described previously, verify your connection to the PostgreSQL container running on your machine and complete the initial migration step above. Once you have verified your Django application is working by running the development server, you are ready to continue developing.
15. [Grp] Begin by creating a new application in your django project. This terminology can become confusing quickly. So, before you complete the step below, briefly review the [Django documentation](#) and describe in your own words the difference between a django project and a django application. Once you have done so, create a new application using the `django-admin` CLI. We will name this application “campaign” as its purpose is to help the marketing team in their telemarketing campaign.

Note: It is best for a single team member to perform this initial application setup and then for other team members to contribute once the structure has been created.

```
django-admin startapp campaign
```

This tool does a number of things for you when creating a new application in your project. First, it creates a new directory for your `campaign` application. Second, it creates a number of new files in that directory. You will learn about each of these files by editing them in the following steps of this project. Before you do, however, now is a good time to stage, commit, and push your changes to GitHub.

16. [Grp] Another bank employee created an [Entity-Relationship Diagram](#) for you to base your new campaign application off of. This diagram describes the relationship between **data** models in your application. These are different from machine learning models and contain the necessary fields and attributes to define the structure of data objects in your project. Using the ERD in the starter code for this week and the [django models documentation](#), update the `models.py` file in your new campaign application directory to define models for each of the entities in the provided diagram.
17. [Grp] Update your project settings to include your newly created campaign application in the `INSTALLED_APPS` setting.

18. [Grp] Create and apply the database migration associated with the new models you have “defined”.

```
python manage.py makemigrations campaign
```

By specifying the campaign application you can better control which applications Django will migrate in the migration it is creating. Once the command successfully finishes, you will see a new file added in the `migrations` sub-directory within your campaign application directory. Investigate this file to see what changes django will make in your database as a result of the models you created. In your `README.md` file, include a brief analysis of what django is doing here to help you manage your database state. When it creates a model, what changes do you see between the declaration of your models in the migration as opposed to what you defined in the `models.py` file. If you are confident in the migrations you intend to make, execute them below.

```
python manage.py migrate campaign 0001
```

This command executes the migration for the **campaign** application that matches the filter “0001”. You should be this specific for all migrations you apply to your database by specifying the application you would like to migrate and which migration for that application you would like to run.

Once completed, this command will create database tables for each model you defined in your `models.py` file. Now is a good time to stage, commit, and push your changes to GitHub.

19. [Grp] While you could interact with these tables using SQL, it is easier to explore them in the GUI django provides via the admin application. Update the `admin.py` file to register these models with your project’s admin application. The admin application is one that comes included with Django and provides basic application management and administration features.

Hint: Your application should define a class that extends the [ModelAdmin](#) class for each class you define.

20. [Grp] Now that you have registered your models with the admin application, [create a superuser](#) account so you can access your project’s admin page.

```
python manage.py createsuperuser
```

Run the development server and navigate to the `http://127.0.0.1:8000/admin` path in your application’s URL. Login using the credentials you just created and explore the models you created.

Once you verify your models work, stage, commit, and push your changes to GitHub.

21. [Grp] Now that you have defined data models, it’s time to write functions that will handle user interaction with your application. Django calls these [views](#) and you define them in a `views.py` file. Write a view called `index` that returns a simple success message.

Hint: You may find it helpful to wrap your returned message in an [HttpResponse](#).

22. [Grp] To access this function, your application needs to define routes. These are defined in a `urls.py` file. One of these was already created in the `tsa/tsa` directory and you will need to create another one in the `campaign` directory. In your project’s base urls, create a route with an empty path that includes all urls in the campaign application’s `urls.py` file. In the campaign application’s `urls.py` file, define a route with an empty path that references the view you defined in the previous step.

23. [Grp] Now that you have defined a route, run the development server and verify it works. Take a screenshot and include it in the `README.md` file in your `tsa` directory. Once you have done so, stage, commit, and push your changes to your team’s GitHub repository.

24. [Grp] To improve usability of your web application, you will need to create a more useful user interface. Your bank’s UI team created a simple mockup in the starter code for this week. To get you started, we included a base html template in the starter code for this week. Using [Django templates](#), you must extend this base template with custom values for the blocked out title and form content. To start out, update the view you created at first to now render the base template we provide. To help your application find this file, create a subdirectory in your base project directory called `templates`. Within this directory, create another directory called `campaign`. Put the base html file we provided for this week in this directory.

Create an adjacent directory named `static` in which you will place your application’s static files (css, images, fonts, and javascript) and two directories named `css` and `fonts` within that. Place the provided starter `factory-style.css` and `style.css` files in the `css` folder and the provided `OpenSans-VarFont.ttf` font file in the `fonts`. Refer to the directory tree that follows for the location in the directory hierarchy to place these directories.

```

capstone/
├── ...
├── tsa/
│   ├── campaign/
│   │   ├── static/
│   │   │   ├── css/
│   │   │   │   ├── factory-style.css/
│   │   │   │   └── style.css
│   │   │   ├── fonts/
│   │   │   │   └── OpenSans-VarFont.ttf
│   │   └── templates/
│   │       ├── campaign/
│   │       │   └── base.html
│   └── tsa/
└── ...

```

25. [Grp] Test your view is working by running the development server and navigating to your webpage in the browser. Once you confirm it is working properly, stage, commit, and push your work to GitHub.
26. [Grp] Create two pages for each model your team made in Week 9. One page should display a form giving the user the ability to query the model you created while the second should display the predicted results. Each must extend the base template provided in this week's starter code using the Django templating language. As you create each form you will need to start the associated Flask applications from last week to test their connection. Include screenshots showing your django application querying the **predict** endpoint of its associated Flask application.  
 Note: For this week, do not use Javascript to query the Flask applications, instead, write views in your Django application that use the python **requests** library to send the query you receive from the browser. These functions should be defined in your **views.py** file as additional functions to the one you already made.
27. [Grp] Start all three Flask applications and run your Django development server simultaneously. Test the ability of your forms to send prediction requests to each prediction end point and display results. As you create and edit these additional views, routes, and templates, stage, commit, and push your changes to GitHub regularly.
28. [Grp] Each team member must come up with a change you would like to make to the UI and implement it. Annotate in the README.md file which element of the UI you elected to change and include a before and after of the change you made.