

Week 12 - Containerization

Up to this point in phase 3 you have taken three machine learning models and exposed their predictive abilities via a RESTful interface and built a web application that provides a single interface you can share to help others interact with your predictive capabilities. While you started by answering a simple analytical question, you have now expanded your project to include an application that supports telemarketers at your bank throughout their workflow. Beyond ranking potential leads for them using your models from Phase 2, you have also introduced a sentiment analysis capability to help them classify ambiguous language a lead uses. Up until this point your application is cumbersome to run and requires you to open multiple terminal windows and specify a different port to bind each application to.

Learning Objectives

- LO1. Describe the core components of and necessary commands in a Dockerfile
- LO2. Create a Dockerfile for Python based web applications
- LO3. Read the logs from a running container
- LO4. Create a container network and deploy a multi-container application

Tasks

Note: This week's tasks will require you to use [Docker](#) extensively. While many people on the internet have shared their opinions of or instructions in how to use Docker, you should use, as your first and principle reference, the [Docker Documentation](#). If you haven't written a Dockerfile before, review the [Dockerfile Reference](#) and [Dockerfile Best Practices](#) in particular. If there are new terms you are unfamiliar with in this week, reference the [Docker Glossary](#) for a description of what they mean.

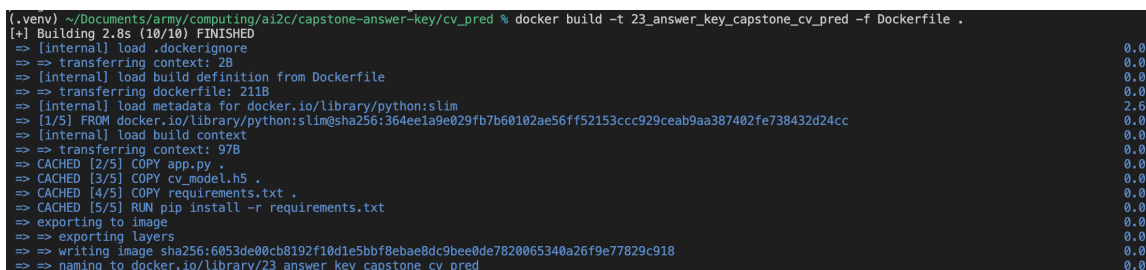
1. [Ind] Containerize your Flask applications. As you start this process, you should create and checkout a new branch. In each of your predictive services, navigate into the Flask app's directory and create a file named `Dockerfile`. (Note: VSCode has extensions specifically to assist in linting Dockerfile's and managing Docker resources. We highly recommend you install this extension in your editor.)
 - i. Define the base image. The Dockerfile you write describes changes to the filesystem and commands to be executed at runtime. The first thing you will need to specify is the base image from which Docker will begin building your container image. While for future projects you may use an [AI2C image](#), in this project we will use a base [python](#) image. A core benefit of containers are their lightweight nature and therefore you will often find "slim" or minimal images. These are an excellent choice for this and future projects for a number of reasons. Start by defining your base image as the latest version of the `python:slim` image.
 - ii. Label the file with a source and description. While not absolutely necessary to build a container image, labeling your images is an important practice to promote reusability and improve the process of sharing tools and images within our organization and the broader community. We will use the open container initiative ([OCI](#)) labeling standard. Your labels should appear immediately following the `FROM` line defining your container's base image and should look like this where `app_name` corresponds with the containerized application (`cv_pred`, `marketing_pred`, or `nlp_pred`):


```
LABEL org.opencontainers.image.source=https://github.com/afc-ai2c/<your-team-repo>
LABEL org.opencontainers.image.description="2023 AI2C Tech Capstone Team # : <app name>"
```
 - iii. Copy the necessary files into the container's file system. Once you have specified your base image, you will need to copy the files you need into your container's file system. Ensure you only copy the files you need for your Flask application to run. For each of the Flask applications, you should be copying three files into your container's file system, the `app.py` file, the model file (except in the case of the sentiment analysis app), and a `requirements.txt` file. You should create a `requirements.txt` file in your application directory where you include only the packages necessary for the `app.py` file in the Flask application you are creating the image for.
 - iv. Prepare the environment by installing necessary packages. Similar to how you would install the Python packages you need locally when you are developing your applicaiton, you will need to use `pip` to install the packages specified in your `requirements.txt` file. Install these packages by running the `pip install -r requirements.txt` command in your `Dockerfile`.

- v. Expose necessary ports. Since you are containerizing a web application, you will need to expose the port from your container properly. As a general rule of thumb, you should always use the default port for whatever software you are containerizing. In the case of Flask, this means you should expose port 5000. (Note: Depending on how you have been running your Flask applications in development, you may have a line in your `app.py` file that checks if you are running the file directly in the Python interpreter. This is generally done by checking `if __name__ == "__main__"` You should remove this code at this point in development)
 - vi. Define an entrypoint. As the final step in defining this simple container image, specify the entrypoint for your Flask application. You should use the following method to run the app, `flask run --host=0.0.0.0`. You may note the host flag as a difference between the way you were running this locally versus the way you will run it in your container. Briefly describe in your app's `README.md` file what it means to set the host to `0.0.0.0` and why you need to specify it as such for your application to be accessible from outside the container.
- [Ind] Build your container image. In the terminal, `cd` into the directory you have created for the predictive application (either `cv_pred`, `marketing_pred`, or `nlp_pred`). From this directory, run the following command to build your container image. Pay particular attention to the repository name you use. Your repository name must follow this format: `23_<team number>_capstone_<cv_pred|marketing_pred|nlp_pred>`

```
docker build -t <repository_name> -f Dockerfile .
```

This command builds a container image with the given repository name, referencing the specified `Dockerfile`, and furthermore specifies it should execute in the current context (`.`). If it succeeds, you will see output like this in your terminal. If you don't see this, revisit the `Dockerfile` creation instructions above.



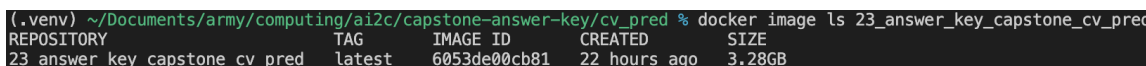
```
(.venv) ~/Documents/army/computing/ai2c/capstone-answer-key/cv_pred % docker build -t 23_answer_key_capstone_cv_pred -f Dockerfile .
[+] Building 2.8s (10/10) FINISHED
=> [internal] load .dockerignore
=> => transferring context: 2B 0.0s
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 211B 0.0s
=> [internal] load metadata for docker.io/library/python:slim 2.6s
=> [1/5] FROM docker.io/library/python:slim@sha256:364eela9e029fb7b60102ae56ff52153ccc929ceab9aa387402fe738432d24cc 0.0s
=> [internal] load build context
=> => transferring context: 97B 0.0s
=> CACHED [2/5] COPY app.py . 0.0s
=> CACHED [3/5] COPY cv_model.h5 . 0.0s
=> CACHED [4/5] COPY requirements.txt . 0.0s
=> CACHED [5/5] RUN pip install -r requirements.txt 0.0s
=> exporting to image 0.0s
=> exporting layers 0.0s
=> writing image sha256:6053de00cb8192f10d1e5bbf8ebae8dc9bee0de7820065340a26f9e77829c918 0.0s
=> naming to docker.io/library/23_answer_key_capstone_cv_pred 0.0s
```

Figure 3.18: Successful Docker build output

- [Ind] If these commands were successful, take a screenshot of your terminal output and include it in the Flask application's `README.md` file. Describe what each stage in the build process did. (Hint: Stages are depicted with the `[/Total #]` line prefix). Be sure to include the command you used and what that command did in plain english.
- [Ind] Verify your container was built using the following command, substituting the name you used above for the repository name.

```
docker image ls <repository_name>
```

You should see output like this if it was successful. Include a screenshot of this output in your `README.md` file.



```
(.venv) ~/Documents/army/computing/ai2c/capstone-answer-key/cv_pred % docker image ls 23_answer_key_capstone_cv_pred
REPOSITORY          TAG         IMAGE ID      CREATED       SIZE
23_answer_key_capstone_cv_pred  latest     6053de00cb81  22 hours ago  3.28GB
```

Figure 3.19: Built Image List

- [Ind] Run your container. In order to run a container, you will need to use the `docker run` command. You did this in the previous week to run your database container.

```
docker run --name <container_name> \
-p <local_port>:5000 \
<repository>:latest
```

Confirm your container is running using the following command. Include a screenshot of this output in your `README.md` file.

```
docker ps -f name=<container_name>
```

You should see the container you just created with a status of "Up". Verify the application within the container is running by navigating to `http://127.0.0.1:<local_port>/heartbeat` in your browser. If you completed the previous steps properly, you should see your Flask application's heartbeat response. Include a screenshot of that output in your `README.md` file.

You should also be able to run your Django application as you did before, and still query your Flask predictive services for predictions on-demand.

6. [Ind] Once you have done so, verify your container's log output. You can view container logs using the `docker logs <container id>` command. Include a screenshot of your application's logs in your directories `README.md` file. Now is a good time to stage, commit, and push your changes if you have not done so already.
7. [Ind] Share your container image with your teammates. While you could each follow the steps above to build the image for each application, this would negate the benefits of building containerized software and applications. Instead, you will need to use the AFC-AI2C GitHub Container Registry to share your container images.

- i. Create a personal access token (classic) by following the steps at [this link](#). Ensure you set the token to have both write and read packages permissions only. Once created, tokens can never be accessed again. Create a new file in the root of your capstone directory titled `pat` and add `pat` to your project's `.gitignore`. Store your newly created personal access token in this file.

Set your new token as an environment variable as follows.

```
export CR_PAT=$( cat ../pat )
```

Note: The command above assumes you are in the terminal within your Flask application's directory. You can substitute the path the file with the appropriate path to the file on your device.

- ii. Authenticate with the container registry.

```
echo $CR_PAT | docker login ghcr.io -u <your_github_username> --password-stdin
```

This command logs into the GitHub Container Registry (`ghcr.io`) using your GitHub username and the newly created container registry token `CR_PAT`.

If you do not see the output, `Login Successful`, revisit the instructions above.

- iii. Tag your container image. In order to push the image to your GitHub repository, you will need to tag the image you created appropriately. Do so using the following command. Ensure the repository name you define is identical to that defined in step 2 and viewed in step 4 previously.

```
docker tag <repository_name> ghcr.io/AFC-AI2C/<repository_name>
```

- iv. Push your image. Execute the following command to push your container image to the GitHub Container Registry. `docker push ghcr.io/AFC-AI2C/<repository_name>` If this command succeeds you will see a number of hashes and status bars until the image push succeeds. You can verify the package was pushed and linked to your repository by opening your team's GitHub repository and viewing the packages listed on the right hand side of the screen.

8. [Ind] Pull your teammates container images after they have built and pushed them successfully using the command below. In this case `repository_name` should refer to the container images your teammates have made.

```
docker pull ghcr.io/AFC-AI2C/<repository_name>
```

Once you have successfully pulled the image, test it by running the container as described above. Your team should come up with a convention for the port numbers your Flask app's run on. Include a section in your overall repository `README.md` that describes that specification.

9. (Optional) [Ind] Some of you may have noticed the warning your Flask application displays whenever you run the development server about not using it in production. As an optional step this week you can upgrade that part of your application now using a tool called [gunicorn](#). gunicorn is a production WSGI server that will enable you to host your Flask applications in a more reliable, secure manner going forward. Should you choose to make this upgrade to any of your team's Flask applications, include a section in that application's `README.md` file describing what configuration settings you selected for gunicorn and why you selected them. Additionally, be sure you build and share an updated container image to the GitHub Container Registry.

10. [Grp] Containerize your Django application. Using what you learned by containerizing your Flask applications, create a `Dockerfile` and `requirements.txt` file in your team's `tsa` directory. A few notes as you do so follow.

- i. Similar to how you should never push automatically generated files to remote source code repositories like GitHub and GitLab, you should not copy those into your container images either. To protect against this, create a `.dockerignore` file in your `tsa` directory. This should include filters to directories like `__pycache__` and files like `.DS_Store`. Most importantly, you must not copy your `.env` file into your container's file system. Instead you will need to pass these environment variables to your container at runtime.

- ii. You will need to pass an argument to your runserver command in order to tell Django to expose port 8000 on all addresses (0.0.0.0:8000).
- iii. When you are writing your `Dockerfile` you will at first experience a build error as the slim version of the python image does not contain necessary binaries for the `psycopg2` package. To alleviate this issue, add the following command to your `Dockerfile` before you run `pip install -r requirements.txt`.

```
RUN apt-get update && \
    apt-get --assume-yes install libpq-dev && \
    apt-get --assume-yes install gcc
```

In the `tsa` directory `README.md` file, explain what this command does in plain english and why it would be better to add only this layer to the image instead of using a larger base image with these packages pre-installed.

- iv. Once you have successfully built your container image run your Django application container with the following flag added to the `docker run` command. `docker run ... --env-file tsa/.env ...`
 - v. The above command should fail due to no database being available at the specified port and host. In order to enable docker containers to communicate with one another you will need to create a bridge network. The instructions to do so follow in the next step.
11. [Ind] Create a Docker bridge network. `docker network create capstone-net` If this command successfully completes you can inspect the results by running `docker network ls` Take a screenshot of the results you find here and add it to your `tsa` directory's `README.md` file. Connect your already running database to your newly created network using the following command. `docker network connect --alias capstone-db capstone-net capstone-db`
Update your `.env` file's value for `DB_HOST` to be the alias created above. Now, when you return to your `docker run` command, add the following flag telling docker to connect your newly started container to the `capstone-net` network.
`docker run ... --network capstone-net ...`
 12. [Ind] Once you have finished these steps and can load your Django application, link your already running Flask applications to the same network in the same way you linked the database container. Test that your Django application is able to communicate with the Flask applications and database container. If all the previous steps succeeded, you can push your container image to the GitHub container repository. Add the appropriate tag and do so now. Once you have done so, each team member should go through the steps above to setup their local machine to run your team's application. Now would be a good time to stage, commit, and push your changes to your team's remote repository if you have not done so already.
 13. [Grp] In your overall project `README.md` answer the following questions.
 - i. Are there dependencies between your containers?
 - ii. If so, which ones are dependent on others to run and which ones are not?
 - iii. Additionally, if there are dependencies, can you think of ways to automate and ensure they are met before the dependent container is run?