Project Tasks

Assuming you have a terminal window open, you're in the project directory, and your `.devops` environment is activated, you can continue viewing and editing project files!

> *This section is quite dense and so, it is suggested that you approach one **task** at a time, carefully reading through the instructions and completing the task, then taking a break if you desire, and coming back to the next task.*

Task 1: Complete the Dockerfile

Docker can build images automatically by reading the instructions from a `Dockerfile`. The Dockerfile contains all the commands a user could call on the command line to assemble an image.

To view the contents of the `Dockerfile` type: `cat Dockerfile`. You can edit any file by opening it in a text editor and saving it.

`Dockerfile`

You can see that you have been given a couple of lines of code in the `Dockerfile` and some instructions.

`FROM` is provided for you; the FROM instruction initializes a new build stage and sets the **base image** for subsequent instructions. In this case, it specifies Python3 as the base image for this application. The rest of the `Dockerfile` instructions are left for you to complete. You should have instructions that:

- Specify a working directory.
- Copy the `app.py` source code to that directory
- Install any dependencies in `requirements.txt` (do not delete the commented `# hadolint ignore` statement).
- Expose a port when the container is created; port 80 is standard.
- Specify that the app runs at container launch.

After you complete this file and save it, it is recommended that you go back to your terminal and run `make lint` again to see if `hadolint` catches any errors in your Dockerfile. **You are required to pass these lint checks to pass the project.**

Task 2: Run a Container & Make a Prediction

In order to run a containerized application, you'll need to build and run the docker image that you defined in the `Dockerfile`, and then you should be able to test your application, locally, by having the containerized application accept some input data and produce a prediction about housing prices.
`run_docker.sh`

Next, open and complete the file, `run_docker.sh` to be able to get Docker running, locally.

Within `run_docker.sh`, complete the following steps:

- Build the docker image from the Dockerfile; it is recommended that you use an optional `--tag` parameter as described in the build documentation.
- List the created docker images (for logging purposes).
- Run the containerized Flask app; publish the container's port to a host port. **The appropriate container and host ports are in the Dockerfile and `make_prediction.sh` files, respectively.**

You can find a list of all the docker commands you might need to use in the documentation.

Running the complete script

This file is a **shell script** which you can see from the extension `.sh`. To *run* a shell script from your terminal, you type `./<scriptname>`.

To run and build a docker image, you'll type `./run_docker.sh`. After typing this command, you should see something like the following in your terminal, followed by a number of build steps:

```
(.devops) Cezanne-2:project-ml-microservice-kubernetes cezannec$ ./run_docker.sh
Sending build context to Docker daemon  751.6kB
```

After a brief waiting period, you should see messages indicating a successful build, along with some indications that your app is being served on port 80 (also, a warning about the development server is to be expected, here).

```
Successfully built <build id>
Successfully tagged <your tag>
```

This indicates a successful build and **if you keep this application running** you can make predictions!

Making predictions

Then, to make a prediction, you have to open a **separate tab or terminal window**. In this new window, navigate to the main project directory (some computers will do this automatically) and call `./make_prediction.sh`.

This shell script is responsible for sending some input data to your containerized application via the appropriate port. Each numerical value in here represents some feature that is important for determining the price of a house in Boston. The source code is responsible for passing that data through a trained, machine learning model, and giving back a predicted value for the house price.

In the prediction window, you should see the value of the prediction, and in your main window, where it indicates that your application is running, you should see some log statements print out. You'll see that it prints out the input payload at multiple steps; when it is JSON and when it's been converted to a DataFrame and about to be scaled.

After making a prediction, you can type `CTRL+C` (+enter) to quit running your application. You can always re-run it with a call to `./run_docker.sh`.

Your next task will be to add a log statement in `app.py` that prints out the pre-trained model **prediction** as Log.info.

The complete text output from these logs will be submitted as part of the complete project.

Task 3: Improve Logging & Save Output

Logging is an important part of debugging and understandability. Many times, logs will be how engineers figure out what an app is doing as it processes some input. In this case, `app.py` is responsible for

1. Accepting an input JSON payload, and converting that into a DataFrame.
2. Scaling the DataFrame payload.
3. Passing the scaled data to a pre-trained model and getting back a prediction.

So far, the logs print out the JSON and DataFrame payloads, but do not have any statements for the scaled input or the resultant prediction. The prediction is an especially important piece of information and so you definitely want that value in the logs.

Add a prediction log statement

Your task is to **add at least one log statement to `app.py` that prints out the output `prediction` values**. You can add more log statements than that, but that is what is required.

Once you have updated your `app.py` code, save it and `./run_docker.sh` again and make the same prediction in a separate terminal window.

Create `docker_out.txt`

Copy and paste this terminal output, which has log info, in a text file `docker_out.txt`. A sample output is shown below.

```
[2019-05-13 02:57:25,911] INFO in app: JSON payload:
{'CHAS': {'0': 0}, 'RM': {'0': 6.575}, 'TAX': {'0': 296.0}, 'PTRATIO': {'0': 15.3}, 'B': {'0': 396.9}, 'LSTAT': {'0': 4.98}}
[2019-05-13 02:57:25,933] INFO in app: inference payload DataFrame:
   CHAS     RM    TAX  PTRATIO      B  LSTAT
0     0  6.575  296.0     15.3  396.9   4.98
[2019-05-13 02:57:25,945] INFO in app: Scaling Payload:
   CHAS     RM    TAX  PTRATIO      B  LSTAT
0     0  6.575  296.0     15.3  396.9   4.98
[2019-05-13 02:57:25,950] INFO in app: output prediction: [20.35373177134412]
172.17.0.1 - - [13/May/2019 02:57:25] "POST /predict HTTP/1.1" 200 -
```

Sample log output

The `docker_out.txt` file should include all your log statements plus a line that reads something like `"POST /predict HTTP/1.1" 200 -` The 200 is a standard value indicating the good "health" of an interaction. **The `docker_out.txt` file will be one of two, log output files that will be part of a passing, project submission.**

Again, after making a prediction, you can type `CTRL+C` (+enter) to quit running your application. You'll need to quit running before you can move on to the next steps and upload the built, docker image.

Task 4: Upload the Docker Image

Now that you've tested your containerized image locally, you'll want to upload your built image to docker. This will make it accessible to a Kubernets cluster.

Upload your Docker image

To upload an image to docker, you'll need to complete the `upload_docker.sh` file:

- Define a `dockerpath` which will be "/path"; the path may be the same as the build tag you created in `run_docker.sh` or just some descriptive path name. Recall that your docker username is your unique docker ID.
- Authenticate and tag image; this step is responsible for creating a login step and ensuring that the uploaded docker image is tagged descriptively.
- Similar to how you might push a change to a Github repository, push your docker image to the `dockerpath` defined in step 1. This push may take a moment to complete.

Assuming you've already built the docker image with `./run_docker.sh`, you can now upload the image by calling the complete shell script `./upload_docker.sh`.

If you've successfully implemented authentication and tagging, you should see a successful login statement and a repository name that you specified, printed in your terminal. You should also be able to see your image as a repository in your docker hub account.

Task 5: Configure Kubernetes to Run Locally

You should have a virtual machine like VirtualBox and `minikube` installed, as per the project environmet instructions. To start a local cluster, type the terminal command: `minikube start`.

After minikube starts, a cluster should be running locally. You can check that you have one cluster running by typing `kubectl config view` where you should see at least one cluster with a `certificate-authority` and `server`.

This is a short task, but it may take some time to configure Kubernetes, and so this deserves its own task number.

Task 6: Deploy with Kubernetes and Save Output Logs

Now that you've uploaded a docker image and configured Kubernetes so that a cluster is running, you'll be able to deploy your application on the Kubernetes cluster. This involves running your containerized application using `kubectl`, which is a command line interface for interacting with Kubernetes clusters.

`run_kubernetes.sh`

To deploy this application using `kubectl`, open and complete the file, `run_kubernetes.sh`:

The steps will be somewhat similar to what you did in both `run_docker.sh` and `upload_docker.sh` but specific to kubernetes clusters. Within `run_kubernetes.sh`, complete the following steps:

- Define a `dockerpath` which will be "/path", this should be the same name as your uploaded repository (the same as in `upload_docker.sh`)
- Run the docker container with `kubectl`; you'll have to specify the container and the port
- List the kubernetes pods
- Forward the container port to a host port, using the same ports as before

After completing the code, call the script `./run_kubernetes.sh`. This assumes you have a local cluster configured and running. This script should create a pod with a name you specify and you may get an initial output that looks as follows, with a cluster and status:

```
(.devops) Cezanne-2:project-ml-microservice-kubernetes cezannec$ ./run_kubernetes.sh
pod   < your pod name >   created
NAME                READY     STATUS              RESTARTS   AGE
< your pod name >   0/1       ContainerCreating   0          0s
error: unable to forward port because pod is not running. Current status=Pending
```

Pending pod

Initially, your pod may be in the process of being created, as indicated by `STATUS: ContainerCreating`, but you just have to wait a few minutes until the pod is ready, then you can run the script again.

Waiting: You can check on your pod's status with a call to `kubectl get pod` and you should see the status change to `Running`. Then you can run the full `./run_kuberenets.sh` script again.

Make a prediction

After you've called `run_kubernetes.sh`, and a pod is up and running, make a prediction using a *separate* terminal tab, and a call to `./make_prediction.sh`, as you did before.

`kubernetes.out.txt`

After running a prediction via Kubernetes deployment, what do you see in your main terminal window?

Copy the text output after calling `run_kubernetes.sh` and paste it into a file `kubernetes_out.txt`. This will be the second (out of two) text files that are required for submission. This output might look quite different from `docker_out.txt`; this new file should include your pod's name and status, as well as the port forwarding and handling text.

Task 7: [Important] Delete Cluster

After you're done deploying your containerized application and making test predictions via Kubernetes cluster, you should clean up your resources and **delete the kubernetes cluster** with a call to `minikube delete`.

You can also pause your work and save the cluster state with a call to `minikube stop`.

Almost Ready for Project Submission

Now, you are almost ready to submit your project!

- Check that you have all complete files
- Push your work to a Github repository
- One last step: **CircleCI Integration**

Task 8: CircleCI Integration

CircleCI is a tool that defines an automated **testing environment**; getting a CircleCI *badge* that reads "Passed" on a repository indicates that the project code has passed all lint tests. CircleCI uses a YAML file to identify how you want your testing environment set up and what tests you want to run. On CircleCI 2.0, this file must be called `config.yml` and must be in a hidden folder called `.circleci`. On Mac, Linux, and Windows systems, files and folders whose names start with a period are treated as system files that are hidden from users by default.

- To create the file and folder on GitHub, click the `Create new file` button on the repo page and type `.circleci/config.yml`. You should now have in front of you a blank `config.yml` file in a `.circleci` folder.
- Then you can paste the text from this yaml file into your file, and commit the change to your repository.
- It may help to reference this CircleCI blog post on Github integration.

Setting up and Building a Project

To test your repository with CircleCI, you will need a CircleCI account, which you can get via their signup page + clicking "Start with GitHub." Once you have an account, you'll be able to build project using the CircleCI dashboard.

On the dashboard, you will be given the option to set up a new project. To add your new repo, ensure that your GitHub account is selected in the dropdown menu in the upper-left, find the project repository that you've created, and click the `Setup project` button next to it. You can leave all set up configurations as their default value then click `Start building`.

You should see your build start to run, and if your project passes the lint tests, you'll see that the project passes!

- You can then add a status badge indicating that your project has "Passed" CircleCI tests, by looking at the markdown in the Notifications section of your project's settings > Status Badges.
- Best practice is to add the badge via markdown into the Github project's README.md file.

PASSED

Task 9: README.md

A complete README file should include:

1. A summary of the project
2. Instructions on how to run the Python scripts and web app (simply listing command line calls will suffice), and
3. A short explanation of the files in the repository.

The README should also include the "passed" status badge (shown above) at the **top** of the README.

Project Submission

Congratuations, you have successfully containerized and deployed a machine learning application using Kubernetes. And you are ready to submit your complete Github repo!

Check that you've passed all rubric items, then **go to the next page** and submit via a link to your Github repo. After submission, your project will be sent to one of our reviewers, who will give you feedback on your project.

Supporting Materials
config.yml