DATA SCIENCE IN THE REAL WORLD

# Deploy a Machine Learning Model as an API on AWS, Step by Step

Brent Lemieux
Jun 10, 2019 · 8 min read ★

Sorry, I don't use Instagram, so I have to post pictures of my dog here.

There are dozens of great articles and tutorials written every day discussing how to develop all kinds of machine learning models. However, I rarely see articles explaining

how to put models into production. **If you want your model to have a real-world impact, it needs to be accessible to other users and applications.**

This step-by-step guide will show you **how to deploy a model as an API.** Building an API for your model is a great way to integrate your work into your companies systems — other developers need only learn how to interact with your API to use your model. **It's also an excellent way for aspiring data scientists to make their portfolio projects stand out.**

This tutorial is for data scientists and aspiring data scientists with little experience in deploying apps and APIs to the web. **By the end, you will know how to:**

1. Develop a model API using Flask, a web microframework for Python.
2. Containerize the API as a microservice using Docker.
3. Deploy the model API to the web using AWS Elastic Beanstalk.

## Why build an API?

Before diving into the code, let's talk about why we might prefer this approach over putting the model code inside the main application. Building a separate service that can be called by the main application has several advantages:

- Updates are more straightforward as the developers of each system don't need to worry about breaking the other.
- More resilient, as a failure in the main application does not impact the model API and vice versa.
- Easy to scale (when using microservice architecture for the API).
- Easy to integrate with multiple systems — i.e., web and mobile.

## The Model

If you're developing a project portfolio, it's best to use less conventional datasets to help you stand out to employers. However, for the sake of this tutorial, I'll be using the famous Boston house prices dataset. The dataset contains several features that can be used to predict the value of residential homes in Boston circa 1980.

I chose to use Random Forest to handle this regression problem. I arbitrarily selected a subset of features for inclusion in the model. If I were developing a "real-world" model, I would try out many different models and carefully select features.

Check out my GitHub repo for instructions to quickly build and save the model. Follow along, then build an API for your modeling projects!

## The API

Create the script `app.py` in the `app/` directory. This directory should also contain the saved model (`model.pkl`) if you followed the instructions in the repo.

These first few lines in `app/app.py` import useful functionality from Flask, NumPy, and pickle. We're also importing `FEATURES` from `app/features.py` which is included below. Next, we initialize the app and load the model.

```
# app/app.py

# Common python package imports.
from flask import Flask, jsonify, request, render_template
import pickle
import numpy as np

# Import from app/features.py.
from features import FEATURES

# Initialize the app and set a secret_key.
app = Flask(__name__)
app.secret_key = 'something_secret'

# Load the pickled model.
MODEL = pickle.load(open('model.pkl', 'rb'))
```

### Features

I've stored the feature list in a separate script for consistency between model training and predictions in the API.

```
# app/features.py

FEATURES = ['INDUS', 'RM', 'AGE', 'DIS', 'NOX', 'PTRATIO']
```

## Endpoints

Our Flask app object (defined above as `app = Flask(__name__)`) has a useful decorator method that makes it easy to define endpoints — `.route()`. In the code below, `@app.route('/api')` tells the server to execute the `api()` function, defined directly below it, whenever http://{your_ip_address}/api receives a request.

```
# app/app.py (continued)

@app.route('/api', methods=['GET'])
def api():
    """Handle request and output model score in json format."""
    # Handle empty requests.
    if not request.json:
        return jsonify({'error': 'no request received'})

    # Parse request args into feature array for prediction.
    x_list, missing_data = parse_args(request.json)
    x_array = np.array([x_list])

    # Predict on x_array and return JSON response.
    estimate = int(MODEL.predict(x_array)[0])
    response = dict(ESTIMATE=estimate, MISSING_DATA=missing_data)

    return jsonify(response)
```

## Parse Requests

We need to include the `parse_args()` function to parse our features out of the JSON requests.

```
# app/app.py (continued)

def parse_args(request_dict):
    """Parse model features from incoming requests formatted in
    JSON."""
    # Initialize missing_data as False.
    missing_data = False

# Parse out the features from the request_dict.
    x_list = []
    for feature in FEATURES:
        value = request_dict.get(feature, None)
        if value:
            x_list.append(value)
        else:
            # Handle missing features.
            x_list.append(0)
```

```
            missing_data = True
    return x_list, missing_data
```

## Start the Application

Finally, run the application on Flask's development server to make sure it's working.

```
# app/app.py (continued)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Start the server with `$ python app.py`. In another terminal window, send a request to the API using `curl`. Pass the features with the `--data` flag in JSON format.

```
$ curl -X GET "http://0.0.0.0:5000/api" -H "Content-Type:
application/json" --data '{"INDUS":"5.9", "RM":"4.7", "AGE":"80.5",
"DIS":"3.7", "NOX":"0.7", "PTRATIO":"13.6"}'

{
  "ESTIMATE": 18,
  "MISSING_DATA": false
}
```

## Production Web Stacks

Flask's development web server is great for testing, but for reasons I won't get into here, we'll need to look elsewhere for our production stack. You can read more about this here. We'll use Gunicorn for our application server and Nginx for our web server. Luckily, AWS Elastic Beanstalk handles the Nginx piece for us by default. To install Gunicorn, run `$ pip install gunicorn`. Create the script `app/wsgi.py` and add just two lines:

```
# app/wsgi.py

from app import app
app.run()
```

Now, run `$ gunicorn app:app --bind 0.0.0.0:5000`. You should be able to execute the same `curl` command as above to get a response from the API.

```
$ curl -X GET "http://0.0.0.0:5000/api" -H "Content-Type:
application/json" --data '{"INDUS":"5.9", "RM":"4.7", "AGE":"80.5",
"DIS":"3.7", "NOX":"0.7", "PTRATIO":"13.6"}'

{
  "ESTIMATE": 18,
  "MISSING_DATA": false
}
```

## Docker

Docker is all the rage these days, and much has been written about its benefits. If you're interested in learning more about Docker and getting a more in-depth intro, read this.

For this tutorial, you'll need to get Docker set up on your computer, follow the instructions here. You'll also need a Docker Hub account.

Once you're all set up, let's dive in! There are two main files you'll need to build a Docker image, `Dockerfile` and `requirements.txt`.

`Dockerfile` includes instructions for creating the environment, installing dependencies, and running the application.

```
# app/Dockerfile

# Start with a base image
FROM python:3-onbuild

# Copy our application code
WORKDIR /var/app
COPY . .
COPY requirements.txt .

# Fetch app specific dependencies
RUN pip install --upgrade pip
RUN pip install -r requirements.txt

# Expose port
EXPOSE 5000
```

```
# Start the app
CMD ["gunicorn", "app:app", "--bind", "0.0.0.0:5000"]
```

`requirements.txt` contains all of the Python packages needed for our app.

```
# app/requirements.txt

Flask==1.0.2
itsdangerous==1.1.0
Jinja2==2.10.1
MarkupSafe==1.1.1
simplejson==3.16.0
Werkzeug==0.15.2
numpy==1.16.4
pandas==0.24.2
scikit-learn==0.19.1
scipy==1.0.0
requests==2.22.0
gunicorn==19.9.0
```

Inside the `app/` directory, run :

```
$ docker build -t <your-dockerhub-username>/model_api .

$ docker run -p 5000:5000 blemi/model_api
```

Your application is now running in a Docker container. Rerun the `curl` command, and you will get the same output!

```
$ curl -X GET "http://0.0.0.0:5000/api" -H "Content-Type:
application/json" --data '{"INDUS":"5.9", "RM":"4.7", "AGE":"80.5",
"DIS":"3.7", "NOX":"0.7", "PTRATIO":"13.6"}'

{
  "ESTIMATE": 18,
  "MISSING_DATA": false
}
```

Run `$ docker push <your-dockerhub-username>/model_api` to push the image to your Docker Hub account. This last step will come in very handy when deploying to AWS Elastic Beanstalk.

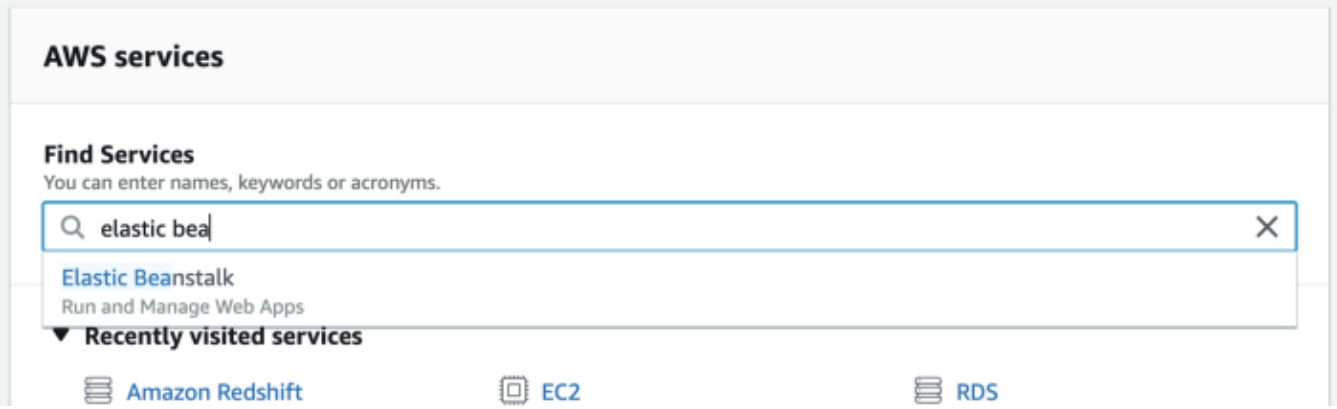
## AWS Elastic Beanstalk

Time to host our API on the web so that our friends and colleagues can access it! Create an AWS account and sign in to the console. ***Note: You'll need to provide a credit card to create your account. If you follow the instructions below, without modifying any of the options, your app will be free-tier eligible, and costs will be minimal. Once you've started your app, navigate to your billing dashboard where you can see your estimated monthly expenses.***

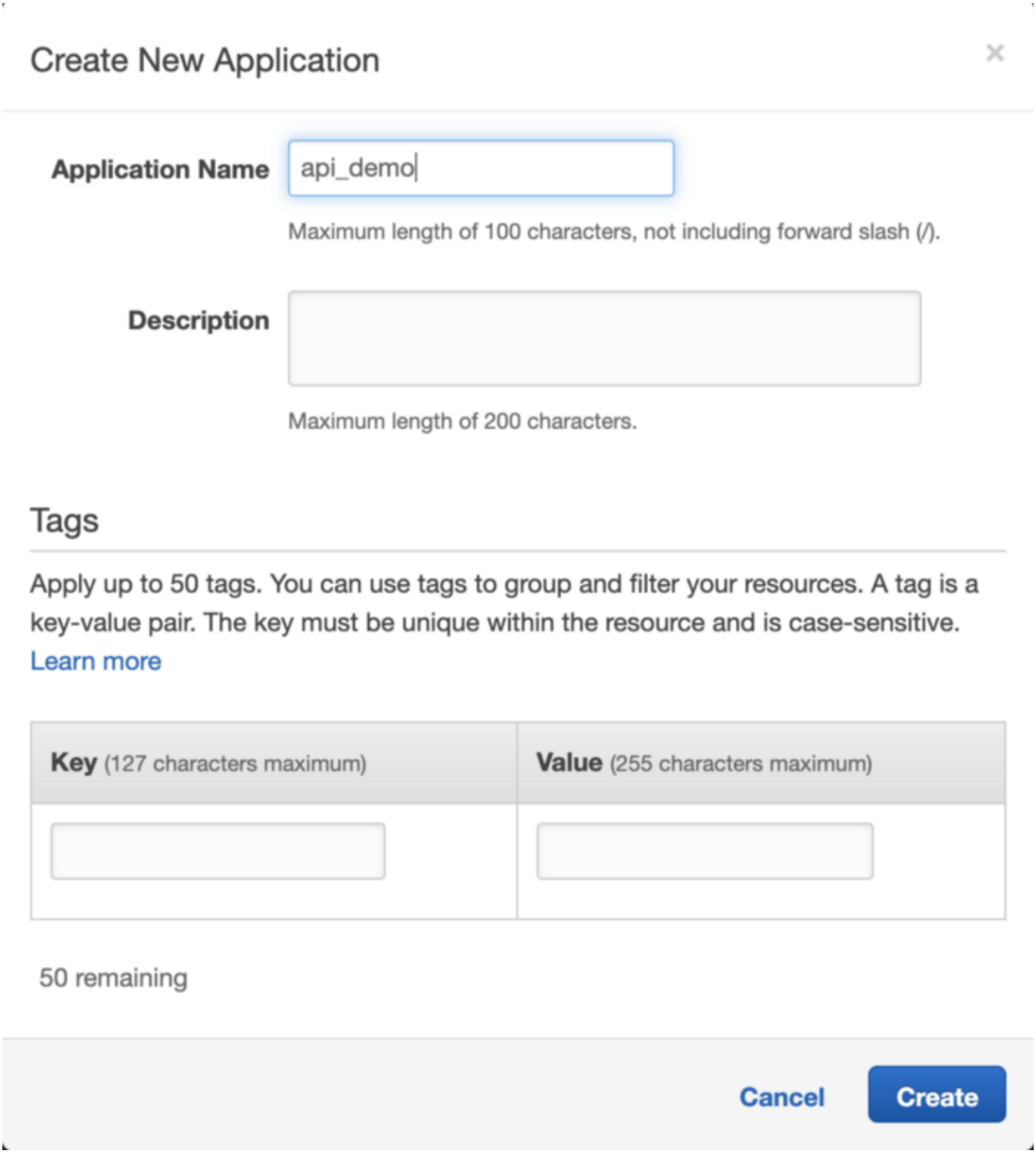
Next, we'll create a file that tells AWS where to access our image. Call this `Dockerrun.aws.json`. The critical piece here is the "Name" value. My Docker Hub username is "blemi", and I've named the image "model_api", so I would put `blemi/model_api:latest` as the "Name" value.

```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "<your-dockerhub-username>/model_api:latest",
    "Update": "true"
  },
  "Ports": [
    {
      "ContainerPort": "5000"
    }
  ],
  "Logging": "/var/log/nginx"
}
```

In the AWS console, search for "Elastic Beanstalk" and select it.

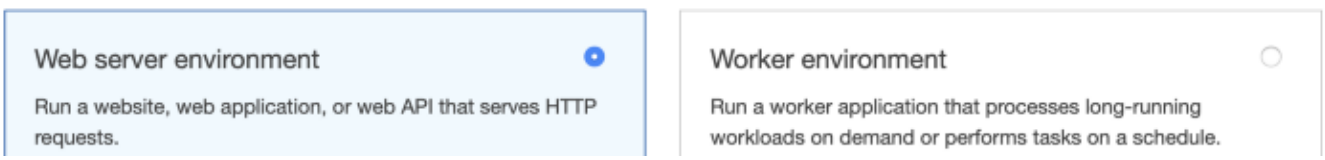


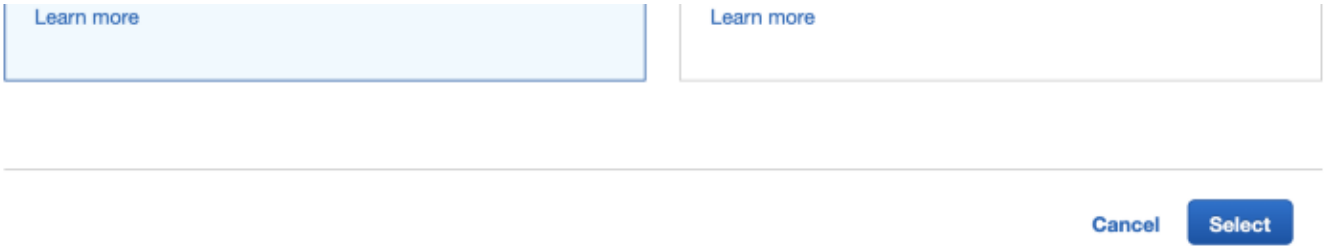

Select "Create New Application" in the top righthand corner, add a name and description to your application, and click "Create."



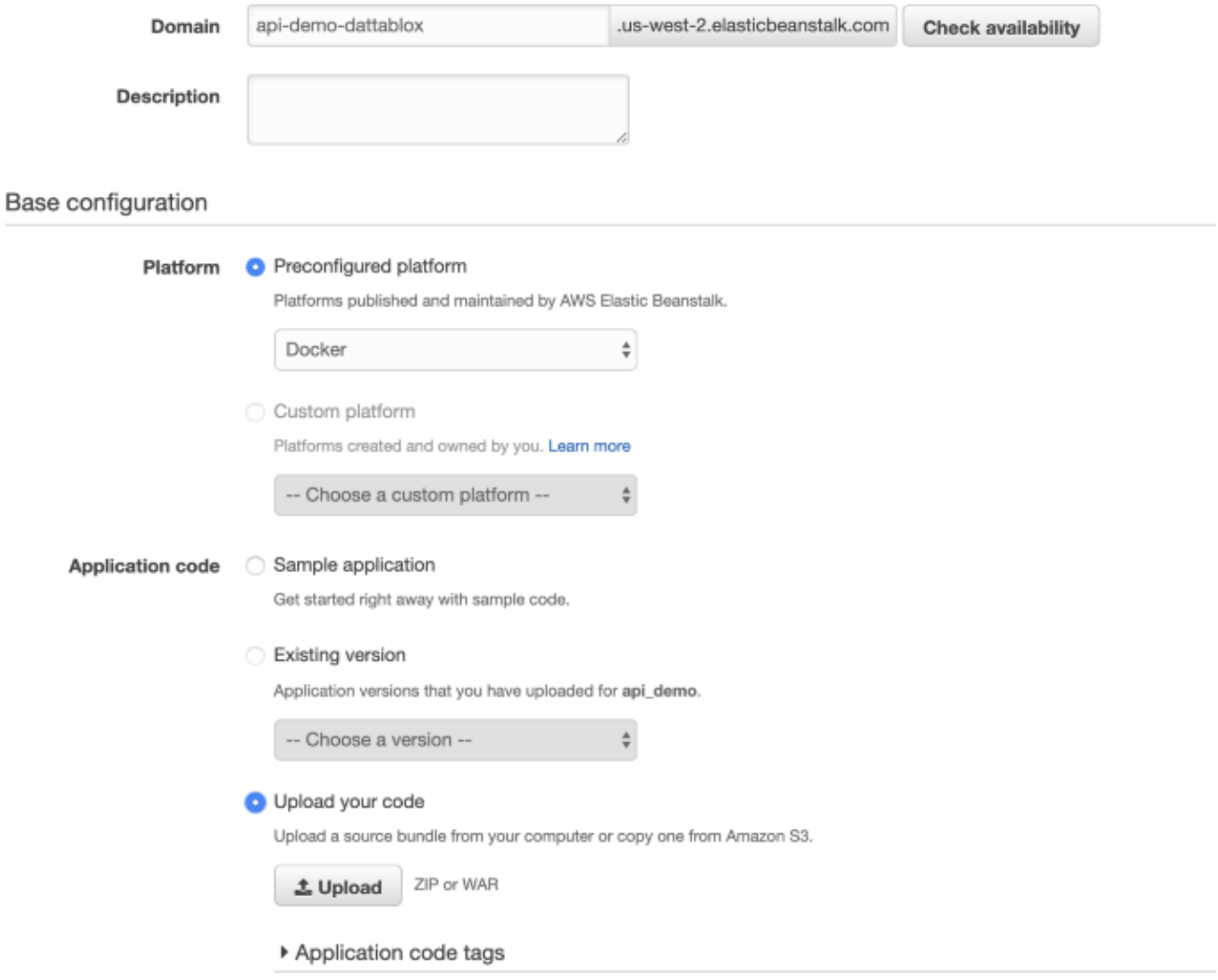

Click “Create one now.”


Leave “Web server environment” selected and click “Select” to continue.

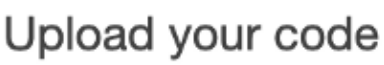


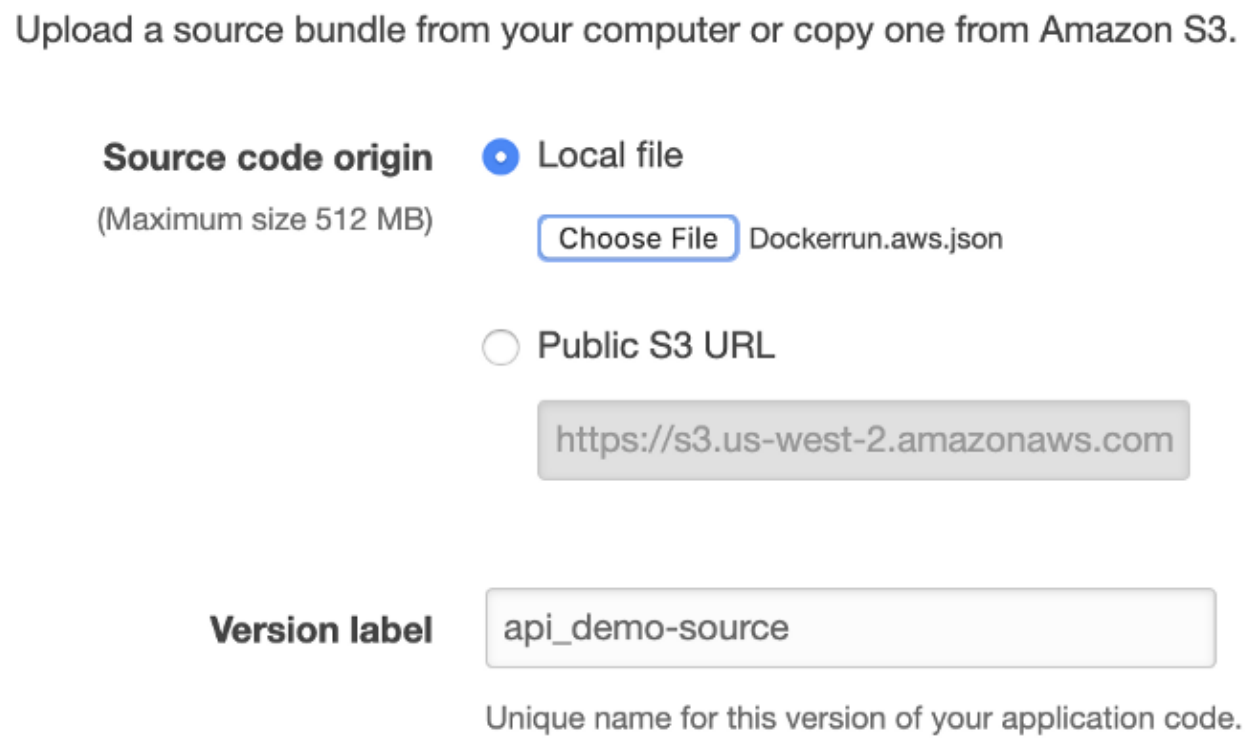



Fill in a custom value for “Domain” and “Description” if you like. For “Platform”, choose “Preconfigured platform” and select “Docker” in the dropdown. For “Application code”, select “Upload your code” and click on the “Upload” button.




Click the “Choose File” button and open the `Dockerrun.aws.json` file that we created above. ***Note: this is only going to work if you’ve pushed your Docker image to Docker Hub.***






Click “Upload” then “Create Environment” to deploy the application.

***Note: If you’re creating a production-grade API, you’ll likely want to select “Configure more options” here before selecting “Create Environment” — if you’re interested in learning about some of the other options to enhance security and scalability, please contact me. My info is at the bottom of this article.***

The app will take a few minutes to deploy, but once it does, you can access it at the URL provided at the top of the screen:


Now, let’s run the `curl` command on our API hosted on the web.

```
$ curl -X GET "http://api-demo-dattablox.us-west-2.elasticbeanstalk.com/api" -H "Content-Type: application/json" --data '{"INDUS":"5.9", "RM":"4.7", "AGE":"80.5", "DIS":"3.7", "NOX":"0.7", "PTRATIO":"13.6"}'

{"ESTIMATE":18,"MISSING_DATA":false}
```

## Summary and final thoughts

We built a simple model API with Python Flask, containerized it with Docker, and deployed it to the web with AWS Elastic Beanstalk. You can now take this knowledge and develop APIs for your models and projects! This makes collaborating with other developers much more accessible. They only need to learn how to use your API to integrate your model into their applications and systems.

There's more that needs to be done to make this API production-ready. Within AWS, and the Flask application itself, there are many configurations you can set to enhance security. There are also many options to help with scalability if usage is heavy. Flask-RESTful is a Flask extension that makes conforming with REST API best practices easy. If you're interested in using Flask-RESTful, check out this great tutorial.

## Get in touch

If you have any feedback or critiques, please share with me. If you found this guide useful, be sure to follow me, so you don't miss future articles.

If you would like to get in touch, **connect with me on LinkedIn**. Thanks for reading!



Docker AWS Machine Learning Data Science Ds In The Real World