Learning Objectives: Distributing with Docker and Heroku

Learners will be able to…

- **Define a container**
- **Describe the benefits of using a container**
- **Explain the importance of the Dockerfile**
- **Identify several important characteristics of Flask app**
- **Identify the benefits of using a platform as a service**

info

Make Sure You Know

Students are comfortable working in the terminal, using Conda to manage packages and virtual environments, as well as have some experience with HTML and CSS.

Limitations

You will create a simple Flask app as this is not a web development course. Docker is discussed only in a cursory manner. The example will be kept to a minimum. Heroku will handle the build and deploy processes of the Flask app.

What is Docker?

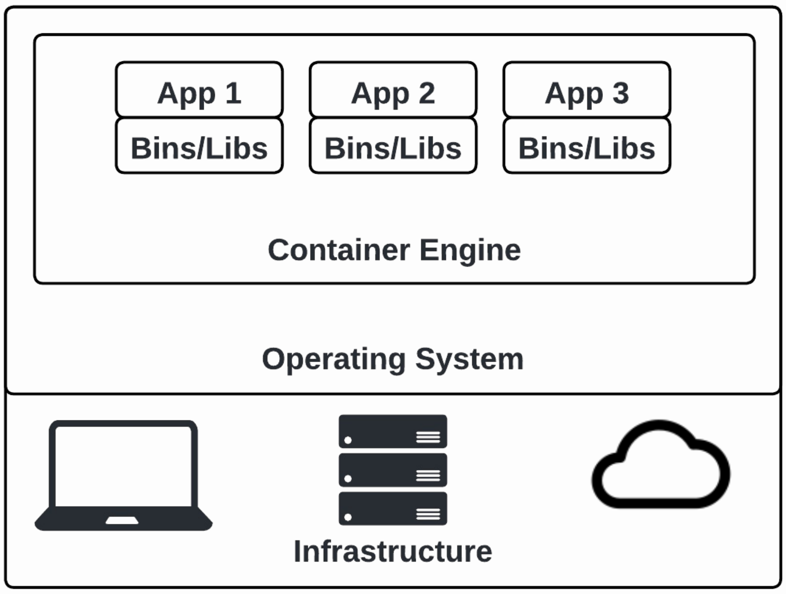
Using Containers to Distribute Python Scripts

In a previous module, we talked about how we can use a program like Anaconda to help with Python development. By creating a virtual environment, we separate our Python project from the rest of our system. Our Python script only interacts with the packages explicitly installed in our virtual environment. If we want to share our work with another person, we can give them our Python code and a way to recreate the virtual environment.

This helps in creating similarity from one programmer to another. But what if I run the Python project on a Linux machine while you run it on a Mac or Windows? The similarities between our work end at the virtual environment. From there on, we see significant differences, namely the underlying OS. How can we guarantee our Python scripts will run identically when there are differences outside the virtual environment?

This is where containers come into play. Containers provide an consistent environment in which our Python script can run. The container is isolated from the rest of the host machine. There are many platforms or tools to help you build containers, but we are going to focus on Docker, one of the most popular platforms.

Containers can run on a variety of machines. You can run one on your local laptop or computer. They can run on a server that you administer. We are going to deploy our Docker container to a cloud service to be hosted on the internet. Regardless of where they run, containers provide an environment separate from the host system. Containers also run on a variety of operating systems as well. Containers provide enough resources to perform a single task. You can create multiple containers if you need to perform multiple tasks.



The benefit of a container is that it provides, from top to bottom, an identical environment in which you can run your Python script. You can install required software and packages in the container. This provide a silo in which the running code is kept separate from the host machine. Therefore, the underlying OS has no effect on a container or the task it performs. Distributing your Python code on a container guarantees a more similar runtime environment than a virtual environment made with traditional Python tools.

But before we can begin creating a Docker container, we first need a Python script to run in it. Over the next few pages, we will use Flask to build a simple website. We will then deploy this site in a Docker container on the internet via a cloud provider.

Flask App

What is Flask?

Flask is a framework used for building web pages with Python. You will still use HTML, CSS, and JavaScript, but Flask (Python) ties it all together on the backend. Web apps work well with Docker, so we are going to build a simple web app with Flask that tells you if today is your birthday (or not).

Flask is built around the idea of templates. For our example, we are going to build a generic template with information used across all our views. This template contains a content block, which allows us to insert custom content we need for a given moment. Our project will have three views — the home view which allows you to enter your birthday, a message view which tells you if today is your birthday, and an about view which is a simple about page for the project.

Let's start by creating the `birthday` virtual environment with Conda.

```
conda create --name birthday -y
```

Then activate the new created environment.

```
conda activate birthday
```

Finally, install the Flask package.

```
conda install flask -y
```

Most Flask apps have a similar structure regarding directories and files. Run the `tree` command to see the structure.

```
tree
```

The `birthday` directory represents the Flask app. The Python logic is in `main.py` (often `app.py` is used instead), the `static` directory contains our CSS to style the site, and the `templates` directory contains all of our HTML templates. You *must* have a directory called templates as Flask expects this to exist.

```
.
└── birthday
    ├── main.py
    ├── static
    │   └── style.css
    └── templates
        ├── about.html
        ├── birthday.html
        ├── check.html
        └── template.html
```

Starting Our Flask App

We will begin with our `main.py` Python file. Click the link below to open it.

Copy and paste the starter code into the file.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def birthday():
    return 'Hello from Flask'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=3000, debug=True)
```

We are importing `Flask` from its page and then instantiating a Flask object. The `app.route` decorator is used to define the different views for our app. The `/` view is the first content you will see when the page loads. For now, return a string with a greeting. Finally, when we call this script, run our Flask app. **Important**, host and port have specific information so they work on Codio. If you are developing a Flask app on your own machine, Flask will provide defaults that will work just fine.

Return to the terminal by clicking on the appropriate tab in the left panel. Then run the Flask app. Remember, the `main.py` is in the `birthday` directory, so we need to specify this with our Python command (or we need to change into the correct directory).

```
python birthday/main.py
```

You will see terminal output about a development server running. It is running port 3000 of our local host, specifically the address `0.0.0.0`. Click the link below to preview our website. You should see our simple greeting and nothing more.

▼ **Did you notice?**

> You are bouncing around a lot from the terminal to files to the running website. This is quite common with web development. Be sure that you are selecting the appropriate tab before making any changes to the code.

challenge

Try this variation:

- Change the greeting in our Flask app by following these steps:
    - Stop the Flask app by going to the terminal and pressing the `Ctrl` and `c` keys at the same time. If you are on a Mac it would be `control` and `c`.
    - Click on the tab for the `main.py` file.
    - Change the message in the `return` statement to be whatever you like.
    - Go back to the terminal and run the Flask app again with:

```
python main.py
```

    - Click on the tab with the Flask preview and click on the icon with the blue arrows to refresh the preview.
    - You should see your new message.

In the terminal, stop the Flask app with `Ctrl + c` and then deactivate the `birthday` virtual environment.

```
conda deactivate
```

Creating Templates

Parent Template

Start the `birthday` virtual environment.

```
conda activate birthday
```

Flask is built around a templating system. So before we render any HTML, we need to design a parent template that will serve as the foundation of our app. Because our app is so simple, our parent template will contain the title `Is it My Birthday?`, which will appear on every view. The most important part of this template is the content block. This block is where information from other views will be displayed. Use the link to open the parent template and copy/paste the following code into the file.

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Parent Template</title>
  </head>
  <body>
    <header>
      <h1 class="title">Is it My Birthday?</h1>
    </header>

    {% block content %}
    {% endblock %}

  </body>
</html>
```

Birthday Template

Now that we have the parent template, we can create children templates that inherit from the parent and provide information in the content block of the parent. The most important parts of a child template is the `extends` statement that inherits from the parent and the block content tags.

Whatever appears in the block content tags of a child will be placed into the block content tag of the parent. Click the link below to open the `birthday.html` template.

Copy and paste the code below into the IDE. This is the `birthday.html` template, which will serve as the first thing you see when loading the page. For now, we are going to keep it simple. There is only a single `<h2>` tag in the content block. That means the parent template has an `<h1>` tag and the child has an `<h2>` tag. Rendering `birthday.html` will show the `<h1>` content from its parent and its `<h2>` content.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Birthday</title>
  </head>
  <body>
    {% extends "template.html" %}
    {% block content %}

    <h2> Enter your birthday to find out </h2>

    {% endblock %}
  </body>
```

Before we can see our templates, we need to make changes to the `main.py` file so that it renders HTML instead of returning a string. Click the link below to open the `main.html` file.

Because Flask is built around templates, this is a rather easy thing to do. We are not doing any data manipulation on the backend, we simply want Flask to render an HTML document. First update our import statement to import `render_template`.

```
from flask import Flask, render_template
```

Then, update the `birthday` function so that it uses `render_template` and pass it `birthday.html`. This will render all of the HTML for `birthday.html` and its parent template as well.

```
@app.route('/')
def birthday():
    return render_template('birthday.html')
```

▼ **Did you notice?**

> We did not have to specify the path to the `birthday.html` file. Remember, Flask *expects* there to be a directory named “templates”. So when we are rendering a template, Flask will look in that directory automatically.

Open the terminal by clicking the link below. Then enter the command to start the Flask app.

```
python birthday/main.py
```

Then preview our app with the following link. You should see the title from the parent template and the text telling you to enter your birthday. The font should have the basic styling of the `<h1>` and `<h2>` tags.

In the terminal, stop the Flask app with `Ctrl + c` and then deactivate the `birthday` virtual environment.

```
conda deactivate
```

Other Templates

Finishing the Birthday Template

Start the `birthday` virtual environment.

```
conda activate birthday
```

The birthday template should provide a way for the user to enter their birthday. To avoid having to clean up user input, we are going to use two drop down menus, one for the month and another for the date. Users click a button to submit their birthday.

Start by creating the first half of the form. **Important**, note that under `action`, we are using `{{ url_for('check_birthday') }}`. This means we are sending the form information to another view called `check_birthday` which we will build later on.

```
{% extends "template.html" %}
{% block content %}

<h2> Enter your birthday to find out </h2>
<form action="{{ url_for('check_birthday') }}"
    method="POST">
    <label for="month">Select a month:</label>
    <select name="month" class="dropdown">
      <option value="January">January</option>
      <option value="February">February</option>
      <option value="March">March</option>
      <option value="April">April</option>
      <option value="May">May</option>
      <option value="June">June</option>
      <option value="July">July</option>
      <option value="August">August</option>
      <option value="September">September</option>
      <option value="October">October</option>
      <option value="November">November</option>
      <option value="December">December</option>
    </select><br>
```

The second half of the form should be another drop down menu, a submit button (has the text “Check”), and a link to the “About” page (which does not yet exist). **Important**, the values for each drop down selection should be a string with a `0` before any single-digit number. This will help us when using Python's `datetime` object to get today's date. Also, verify that the `<form>` tag resides inside the content block in the template.

```
        <label for="day">Select a day:</label>
        <select name="day" class="dropdown">
          <option value="01">1</option>
          <option value="02">2</option>
          <option value="03">3</option>
          <option value="04">4</option>
          <option value="05">5</option>
          <option value="06">6</option>
          <option value="07">7</option>
          <option value="08">8</option>
          <option value="09">9</option>
          <option value="10">10</option>
          <option value="11">11</option>
          <option value="12">12</option>
          <option value="13">13</option>
          <option value="14">14</option>
          <option value="15">15</option>
          <option value="16">16</option>
          <option value="17">17</option>
          <option value="18">18</option>
          <option value="19">19</option>
          <option value="20">20</option>
          <option value="21">21</option>
          <option value="22">22</option>
          <option value="23">23</option>
          <option value="24">24</option>
          <option value="25">25</option>
          <option value="26">26</option>
          <option value="27">27</option>
          <option value="28">28</option>
          <option value="29">29</option>
          <option value="30">30</option>
          <option value="31">31</option>
        </select>
        <br>
        <input type="submit" value="Check" id="check">
    </form>
    <a href="{{ url_for('about') }}">About</a>

    {% endblock %}
```

Because the code above references a URL for `check_birthday` and `about`, we need to modify our `main.py` file. Open the file with the link below. Then copy/paste the beginning of the `check_birthday` and `about` routes. We will complete the functions later.

```
@app.route('/about')
def about():
    pass

@app.route('/check_birthday')
def check_birthday():
    pass
```

Let's see what the Flask app looks like now. Go back to the terminal and run the website.

```
python birthday/main.py
```

Then preview our app with the following link. We should now see our form (two drop down menus and a button) as well as the link. Click on each drop down menu and make the correct months and dates are visible. **Important**, the link does not work as we have not created the `about.html` template. The "Check" button does not work yet either.

▼ **Did you notice?**

> The link for the "About" page and the "Check" button are both broken. We will address these issues shortly.

Finally, stop the Flask app by pressing `Ctrl + c` on the keyboard.

About Page

Most websites have an about page, so we will too. Click the link below to open the `about.html` template.

This is a very simple page that inherits from the parent template and adds an `<h2>` tag, a sentence about the purpose of this project, and a link back to the homepage.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>About</title>
  </head>
  <body>
    {% extends "template.html" %}
    {% block content %}

    <h2>About This Site</h2>
    <p>This is site is a quick example for learning how to get
        dockerized Flask app deployed to Heroku.</p>
    <a href="{{ url_for('birthday') }}">Home</a>

    {% endblock %}
  </body>
```

The about page is a different view, so we need to modify the `main.py` file accordingly. Open the file with the link below.

Since this is a different view than the `birthday.html` template, differentiate it by using `/about` in the `app.route` decorator. Then define the `about` function which renders the `about.html` file.

```
@app.route('/about')
def about():
    return render_template('about.html')
```

We are ready to check our work and test the links. Open the terminal by clicking the link below. Then enter the command to start the Flask app.

```
python birthday/main.py
```

Then preview our app with the following link. As always, Flask loads the `birthday.html` template first. You should be able to click the link at the bottom to view the about page. Make sure the `Home` link takes you back to the `birthday.html` template.

In the terminal, stop the Flask app with `Ctrl + c` and then deactivate the `birthday` virtual environment.

```
conda deactivate
```

Backend Calculations

Working with Form Data

Start the `birthday` virtual environment.

```
conda activate birthday
```

The benefit of using a framework like Flask is that you can use Python for any calculations or manipulation that needs to take place (as opposed to using JavaScript). Use the link below to open `main.py`, so we can add a function to calculate if today is the user's birthday or not.

First update our import statements. We need to use the `datetime` package as well as `request`, which comes from `flask`.

```
from datetime import datetime
from flask import Flask, render_template, request
```

Next, we are going to create a view for the response if today is your birthday or not. Create another `app.route` decorator and name it `/check_birthday`. In addition, allow for the `POST` method so we can send data from the form to Python on the backend. Then create the function `check_birthday`.

```
@app.route('/check_birthday', methods=['POST'])
def check_birthday():
```

Use `request.form` to get information from our form. **Remember**, `'month'` and `'day'` are the names of the `<select>` tags (the drop down menus) in our form.

```
@app.route('/check_birthday', methods=['POST'])
def check_birthday():
    user_month = request.form['month']
    user_day = request.form['day']
    dt_obj = datetime.now()
    today_month = dt_obj.strftime('%B')
    today_day = dt_obj.strftime('%d')
```

Now that we know what today's date and the user's birthday, we can make a comparison. If both the day and month match, then it is the user's birthday. Either way, we need to prepare a message to send back to the website. Create two variables (`msg` and `txt`) with a message for the user. We are splitting the message into two variables so we can style them differently. As you render the `check.html` template, pass it the variable `value` and assign it a tuple with both `msg` and `text`.

```python
@app.route('/check_birthday', methods=['POST'])
def check_birthday():
    user_month = request.form['month']
    user_day = request.form['day']
    dt_obj = datetime.now()
    today_month = dt_obj.strftime('%B')
    today_day = dt_obj.strftime('%d')

    check_month = today_month == user_month
    check_day = today_day == user_day

    if check_month and check_day:
        msg = 'Happy birthday!'
        txt = 'Today is your birthday. Spend it doing something
        you enjoy with those you enjoy being around.'
    else:
        msg = 'It is not your birthday'
        txt = 'Sadly, today is not your birthday. Try again
        later.'

    return render_template('check.html', value=(msg, txt))
```

Message for the User

In our final template, we want to display the two messages stored in the variable `value` as well as have links for the homepage and the about page. Open the `check.html` template by clicking the link below.

Copy and paste the code into the IDE. **Important**, notice how a variable is surrounded by `{{ }}` in the HTML. Since `value` is a tuple, use index `0` to reference `msg` and index `1` to reference `txt`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Check Birthday</title>
  </head>
  <body>
    {% extends "template.html" %}
    {% block content %}

    <h2>{{ value[0] }}</h2>
    <p>{{ value[1] }}</p>
    <a href="{{ url_for('birthday') }}">Home</a><br>
    <a href="{{ url_for('about') }}">About</a>
    {% endblock %}
  </body>
```

Open the terminal by clicking the link below. Then enter the command to start the Flask app.

```
python birthday/main.py
```

Then preview our app with the following link. We need to check our two use cases. Click the “Check” button without making any changes to the month or day. This should produce the message that it is not our birthday. Then enter today’s date and click “Check”. You should see the message about it being our birthday.

In the terminal, stop the Flask app with `Ctrl + c` and then deactivate the `birthday` virtual environment.

```
conda deactivate
```

Styling the Site

Adding CSS

Start the `birthday` virtual environment.

```
conda activate birthday
```

The last thing we are going to do is style the website. This is a rudimentary Flask app, so the CSS will likewise be rather simple. By convention, CSS files are stored in the `static` directory in a Flask app. Use the link below to open the `style.css` file.

Let’s start by having some general styling for the whole site. Change the background color from white to a light gray. Then use the Open Sans font from Google for all text. Color the text a dark blue and set the default size to 22 pixels.

```css
body {
  font-family: "Open Sans", sans-serif;
  text-align: center;
  background-color: #ecf0f1;
  color: #2c3e50;
  font-size: 22px;
}
```

For the `<h1>` and `<h2>` tags, set the color to a light blue. Set the font size to 96 pixels for `<h1>` tags and 60 pixels for `<h2>` tags.

```css
h1 {
  font-size: 96px;
  color: #2980b9;
}

h2 {
  font-size: 60px;
  color: #2980b9;
}
```

Style the button by having a matching font size with the body element. Use the same dark blue text color as well. Give the button a slight border radius. The button itself should have a 2 pixel border with a solid line the same color as the text. Finally, set the background color to the same light gray as the site itself.

```css
input {
  font-size: 22px;
  color: #2c3e50;
  margin-bottom: 10px;
  border-radius: 5px;
  border: 2px solid #2c3e50;
  background-color: #ecf0f1;
}
```

Both drop down menus use the `dropdown` CSS class name so styling them is easier. The menus will use the same color scheme as the rest of the site, as well as the same border and border radius as the “Check” button. Adding a margin and some padding will help space things out a bit.

```css
.dropdown {
  font-size: 16px;
  padding: 5 5px;
  text-align: center;
  border-radius: 5px;
  border: 2px solid #2c3e50;
  font-size: 22px;
  color: #2c3e50;
  margin-bottom: 10px;
  background-color: #ecf0f1;
}
```

CSS Links

Before we can test our site, we need to link to the documents needed for the styling. Since we are using the same styling across all templates, we are going to modify the parent template. Each child template will import the same styling. Open `template.html` with the link below.

In the `<head>` tag, we want to link to our CSS style sheet. Use `url_for` followed by the directory and then the name of the file. This tells Flask where to find our styling. We also need to link to the Open Sans Google font. These fonts are freely available.

```
<head>
  <meta charset="utf-8">
  <link rel="stylesheet"  type="text/css" href="
      {{url_for('.static', filename='style.css')}}">
  <link rel="stylesheet"
      href="https://fonts.googleapis.com/css?
      family=Open%20Sans">
  <title>Parent Template</title>
</head>
```

Open the terminal by clicking the link below. Then enter the command to start the Flask app.

```
python birthday/main.py
```

Then preview our app with the following link. The Flask app should look better with the new colors, font, and other styling aspects.

In the terminal, stop the Flask app with `Ctrl + c` and then deactivate the `birthday` virtual environment.

```
conda deactivate
```

Now that we have a Flask app, we are ready to get it running on Docker and hosted by Heroku.

Dockerfile

Dockerfile, Image, and Container

Start the `birthday` virtual environment.

```
conda activate birthday
```

We previously talked about the advantages using a container has over using a virtual machine. We will be using the Docker container system to distribute our Flask app. First, let's start with some vocabulary. When we say our app is "running in Docker", what we really mean is the **container** is running an instance of a Docker image.

A **Docker image** is the environment that runs our application. We can select things like the operating system, the version of Python, any dependencies for our app, etc. You can be very selective about what is and is not a part of your Docker image. For example, Alpine Linux is a popular choice as the foundation of a Docker image. This OS is tiny and consumes hardly any resources. This helps with performance. Also, the fewer packages that are installed on the image, the fewer potential security holes.

The instructions for building a Docker image are contained in the **Dockerfile**. This is a blueprint that builds the environment that runs our app step by step. Every time you want to use Docker for a project, you need to start with a Dockerfile. Every Dockerfile sets the foundation with an image, then we customize it for our needs. This could be installing additional software, copying files, running commands, etc. Start by creating the file `Dockerfile` itself. Then use the link below to open it.

```
touch Dockerfile
```

▼ **Did you notice?**

> The name for `Dockerfile` starts with a capital "D". This is a requirement. The file does not have an extension at the end either.

We are going to keep things very simple with our Dockerfile. First, we are going to use the latest version (2021.11 at the time of writing) of `anaconda3` as the foundation of our project. Anaconda maintains the `anacoda3` image. They provide us with a pre-made selection of software. In particular, it gives us access to Python and Flask. This means we do not have to tell our Dockerfile to install this dependency. You could argue using the `anaconda3` image is excessive given our needs.

```
FROM continuumio/anaconda3:2021.11
```

Next, we need to copy any relevant files to the Docker image. We want all of our files, which is represented by the `.`. Again, we could probably leave off some files, but we are keeping things simple. Copy our files to `/code` directory on the Docker image. Then set our working directory to `/code`.

```
FROM continuumio/anaconda3:2021.11

ADD . /code

WORKDIR /code
```

Finally, our entry point serves as the command that starts up our Flask app. The first element in the list is the command to run a Python script. Remember, we built our Flask app inside the `birthday` directory. So use `birthday/main.py` as the path for our script.

```
FROM continuumio/anaconda3:2021.11

ADD . /code

WORKDIR /code

ENTRYPOINT ["python", "birthday/main.py"]
```

That is all their is to our simple Dockerfile. When we build our custom Docker image we start with an image from Anaconda. This has several packages already installed, most importantly Flask. We then copy our work into a directory on Docker called `/code`. This directory is set as our working directory. Finally, we give the command and file path to run our Flask app.

In the terminal, deactivate the `birthday` virtual environment.

```
conda deactivate
```

Deploying with Heroku

What is Heroku?

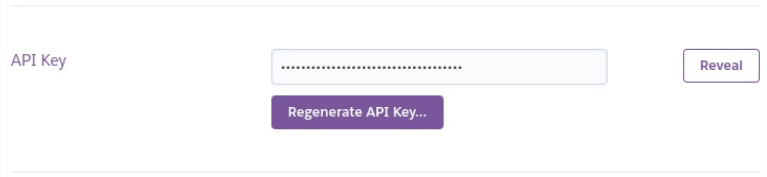
We have our Flask app and Dockerfile, which means we are ready to build and deploy our Docker image. We are going to use Heroku for this. Heroku is one of the first cloud computing systems. They provide a platform as a service (PaaS) to their customers. This means Heroku builds out and maintains an infrastructure that customers can use to host their own applications. Our Flask app needs to run on a web server to be accessible to the wider internet. Heroku provides all of the infrastructure needed to host a website. They provide the servers, they offer a set of installable software, they provide software updates to ensure a secure system, they generate a URL for our Flask app, etc.

The first thing you need to do is go to Heroku and create a free account. You may be asked to enable two-factor authentication for your account. So this process could be a bit longer than you would expect.

Once we have our Heroku account, we need to download and install the Heroku command line interface (CLI). Codio runs on the Ubuntu operating system, so run the following script in the terminal. You should see a success message after installation.

```
curl https://cli-assets.heroku.com/install-ubuntu.sh | sh
```

We need to login to our Heroku account via the CLI before we can go any further. Heroku will ask for your username and password. However, if your account is set up with multifactor authentication, they will not let you authenticate with this information. Instead, you have to use an API key in place of your password. Go to the tab where you are logged in on Heroku. Click your avatar in the top-right corner. Select account settings. Scroll down until you see the section for API Key. Copy the API key. You do not need to click reveal first. You can copy the obscured API key.



The image depicts the Heroku settings page for your account. Toward the bottom there is a section entitled API Key. The key is represented as a series of dots. To the right of the obscured key is a button to reveal the API key's contents.

Once we have the key, we can start the login process. In the terminal, enter the following command.

```
heroku login -i
```

You will be prompted for your email and password for Heroku. **Remember**, use the API key in place of your password. The process will look something like the example below.

```
heroku: Enter your login credentials
Email: your-email@provider.com
Password: ************************************
Logged in as your-email@provider.com
```

Next, we need to create our Heroku application. The name used for the application will also be used in the URL for our Flask app. So the application name needs to be unique to you. The command below creates a Heroku application and sets the buildpack to Python. **Important**, replace `<your initials>` with your initials or some other unique identifier.

▼ **Unique names**

> If the name you chose for your app is not unique, you will see a message that this name is already taken. You will have to run `heroku create` again with a different name. You may have to do this a few times until you find a unique name.

```
heroku create check-birthday-<your initials> --buildpack
        heroku/python
```

Upon successful application creation, you will see a message like the one below. You get a URL for the Heroku app where we can see our Flask app running on the internet. The other URL is for the git repository where our code lives. Take note of the first URL.

```
Creating ⬢ check-birthday... done
https://check-birthday.herokuapp.com/ |
https://git.heroku.com/check-birthday.git
```

By default, Heroku runs Ubuntu for its applications. We want to run a Docker container instead. Set the stack to container and use the `-a` flag to name the application to which this change applies. **Important**, replace `<heroku app name>` with the name of the Heroku app you chose above.

```
heroku stack:set container -a <heroku app name>
```

Heroku YAML File and Flask App

Now that we have setup and configured our Heroku app, we need Heroku to build our Docker image and deploy it to a container. This is done with a YAML file called `heroku.yml`. Enter the following command to create the the YAML file. Then open it with the link below.

```
touch heroku.yml
```

This file serves as a set of instructions for building and deploying our Flask app. The first section is for building. Specify that we are building a Docker image and then use our Dockerfile for the build process. Normally we would follow the build section with a run section. However, by omitting the run section Heroku will default to our `ENTRYPOINT` command in our Dockerfile instead. That's all we need for the YAML file.

```
build:
  docker:
    web: Dockerfile
```

The last few changes come to our `main.py` app. Previously, our Flask app was setup to run in debug mode on a specific port (3000 in our case). We need to turn off debug mode and let Heroku specify what port it wants to use. Open the Python file with the link below. Then, import the `os` module.

```
import os
from datetime import datetime
from flask import Flask, render_template, request
```

In the conditional at the bottom, we want to create the variable `port`. This will get the `PORT` environment variable from Heroku. Then use the `port` variable for setting the port and change `debug` to `False`.

```
if __name__ == '__main__':
    port = os.environ.get("PORT", 5000)
    app.run(host='0.0.0.0', port=port, debug=False)
```

All of our files are finally done. The next step is to put all of this on Heroku, build our Docker image, and deploy the Docker container on the internet. Remember, Heroku provides each application with a git repository. As such, we are going to use git commands to finalize this process. Start by initializing the directory for git.

```
git init
```

Next, we want to add all of our files so they will be pushed to Heroku.

```
git add .
```

We need to commit our changes. Use the `-m` flag to write a message about the commit. Since this is our first commit, “initial commit” is a sufficient message. If you make any future changes, use descriptive commit messages to help you remember the changes you made.

```
git commit -m "initial commit"
```

Finally, push our changes to the git repository. This will kick of a series of events that build our Docker image, create a container, and ultimately lead to our Flask app running on the internet.

```
git push heroku master
```

This process can take a bit of time to complete. Once done, open a new tab and go to the Heroku URL to see the Flask app running.

▼ **What if I forgot my URL?**

> Go to your Heroku dashboard, which has a list of all your applications. Click on your Flask app. Toward the top-right, click the button that says `Open app`. This will open your Flask app in a new tab.