Lab Starter Code

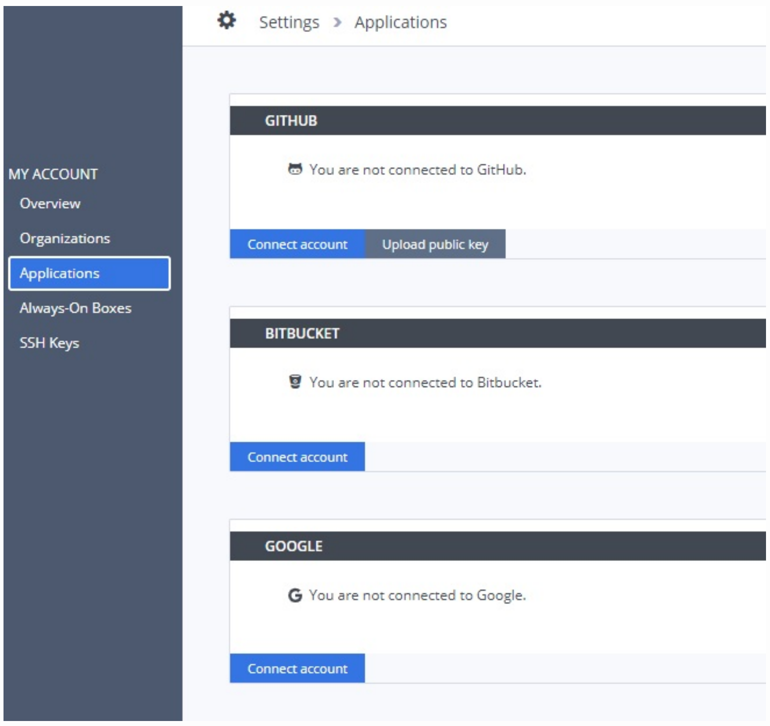
Connecting to GitHub

We are going to build a Flask app that you can upload to your GitHub repository. If you do not yet have an account, please create one now. We are going to clone a repository that will contain the starter code for your Flask app.

Connecting GitHub and Codio

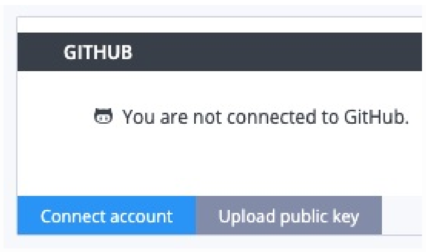
You need to connect GitHub to your Codio account. This only needs to be done one time.

- In your Codio account, click on your username
- Click on **Applications**



The image depicts all of the services you can connect to your Codio account. GitHub is the topmost option.

- Under GitHub, click on **Connect account**



The image depicts the buttons associated with connecting your GitHub account to Codio. The first button says to connect GitHub. The second button says to upload a public key.

- You will be using an SSH connection, so you need to click on **Upload public key**

Fork the Repository

- Go to the ascii-flask repository. This repository is the starting point for your project.
- Click on the "Fork" button in the top-right corner.
- Click the green "Code" button.
- Copy the SSH information. It should look something like this:

```
git@github.com:<your_github_username>/ascii-flask.git
```

info

Important

If you do not use the SSH information, you will have to provide your username and Personal Access Token (PAT) to GitHub each time you push or pull from the repository. See this documentation for setting up a PAT for your GitHub account.

In the Terminal

- Clone the repository. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/ascii-flask.git
```

- You should see a `ascii-flask` directory appear in the file tree.

You are now ready to start the project.

Creating ASCII Art

Setting Up Our Virtual Environment

ASCII art is a representation of words or images using only ASCII characters. In this project, we will take the URL of an image and return an ASCII art rendition of the image. Before we can do this, we need to setup our work space. Create the `ascii` virtual environment with Conda.

```
conda create --name ascii -y
```

Then activate the environment.

```
conda activate ascii
```

Next, install `pillow` an image manipulation library for Python. This needs to be installed from the Conda Forge channel.

```
conda install pillow -c conda-forge -y
```

We also need to install the `requests` package.

```
conda install requests -y
```

Finally, open the `ascii_art.py` file. This is where we will create an algorithm to convert an image from the internet into a list of strings. Each character in the string represents a pixel from the original image.

Transforming an Image into ASCII Art

Start by importing the necessary packages. `PIL` is the Pillow package. The `requests` package is used to fetch data from a website. `BytesIO` is used to store the contents of the image from the internet.

```
from PIL import Image
import requests
from io import BytesIO
```

Define the `create_ascii` function which takes the URL of an image. We need to be able to handle bad input from the user. If not, the Flask app will crash when it cannot open an image URL. There are two main ways in which the app would crash. One, the URL is not given in its full form - `https://...`. Two, the URL provided is not the URL for an image. This project assumes users will enter a URL for an image; i.e., if you enter that URL in a browser an image (and only an image) would load. If we cannot open the webpage or convert it to an image, return a string telling the user to enter a valid URL.

```
def create_ascii(img_url):
  try:
    response = requests.get(img_url)
    pic = Image.open(BytesIO(response.content))
  except:
    return 'Please enter a valid URL for an image'
```

We need to resize the image. Unpack the `width` and `height` from the `pic.size` tuple. Figure out the aspect ratio for the image. Create a new width of `110` and using the aspect ratio, give the image a new height. Resize the image to the new width and height.

```
  width, height = pic.size
  aspect_ratio = height / width
  new_width = 110
  new_height = int(aspect_ratio * new_width)
  img = pic.resize((new_width, new_height))
```

Convert the image to grayscale. Then use the `getdata()` method to get the data for each pixel in the image. This returns a list of numbers each with a value from 0 to 255. 0 represents the color black and 255 represents the color white. Numbers in between are various shades of gray. Create the list `chars`. The first character should have the “most writing” in it. As you move through the list, the characters have less and less “writing”. At the very end, you have a blank space. The progression of characters matches the progression from 0 (black) to 255 (white). Then count the number of characters in the list `chars`.

```
  img = img.convert('L')
  pixels = img.getdata()
  chars = ['@', '#', 'S', '%', '?', '*', '+', ';', ':', ',',
        '.', ' ']
  count = len(chars)
```

This is the most important part of the ASCII art algorithm. For each pixel in the image, find the corresponding character in the `chars` list. Then, use the `join` method to take the list of strings and turn it into one giant string. Calculate the length of this giant string. We need to insert line breaks every 110 characters (the new width of our ASCII art). Create the list `ascii_image`. Using a comprehension, loop over the range from 0 to the length of the `new_pixels` string. Use the interval of `new_width` which is 110. This means the index of the loop is 0, 110, 220, etc. until the end of the string. Use the slice operator to take each chunk of 110 characters and put them in `ascii_image`. Lastly, return the `ascii_image` list.

```
  new_pixels = [chars[int(((count-1)*pixel)/256)] for pixel in
        pixels]
  new_pixels = ''.join(new_pixels)
  new_pixels_count = len(new_pixels)
  ascii_image = [new_pixels[index:index + new_width] for index
        in range(0, new_pixels_count, new_width)]
  return ascii_image
```

Finally, we need to test our code. Add a conditional that checks if we are running `ascii_art.py` directly. Start our testing with the two examples of a bad URL. The first URL (`google.com`) is bad because it is missing `https://...`. The second example is bad because the URL is for a valid webpage, but it is not an image URL. Finally, test the `create_ascii` function with a working URL. Iterate over the list, printing each string in the list.

```
if __name__ == '__main__':
  print(create_ascii('google.com'))
  print(create_ascii('https://www.google.com/'))
  art =
        create_ascii('https://images.pexels.com/photos/5302784/pe
        xels-photo-5302784.jpeg?
        auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1')

  for line in art:
    print(line)
```

Swtich back to the tab with the terminal and enter the following command. You should see two messages about entering a valid image URL as well as the silhouette of an individual printed in the terminal.

```
python ascii-flask/ascii_art.py
```

Deactivate the virtual environment.

```
conda deactivate
```

▼ **Code**

Your code should look like this:

```
from PIL import Image
import requests
from io import BytesIO

def create_ascii(img_url):
  try:
    response = requests.get(img_url)
    pic = Image.open(BytesIO(response.content))
  except:
    return 'Please enter a valid URL for an image'

  width, height = pic.size
  aspect_ratio = height / width
  new_width = 110
  new_height = int(aspect_ratio * new_width)
  img = pic.resize((new_width, new_height))
  img = img.convert('L')
  pixels = img.getdata()
  chars = ['@', '#', 'S', '%', '?', '*', '+', ';', ':', ',', '.', ' ']
  count = len(chars)
  new_pixels = [chars[int(((count-1)*pixel)/256)] for pixel in pixels]
  new_pixels = ''.join(new_pixels)
  new_pixels_count = len(new_pixels)
  ascii_image = [new_pixels[index:index + new_width] for index in range(0, new_pixels_count, new_width)]
  return ascii_image

if __name__ == '__main__':
  print(create_ascii('google.com'))
  print(create_ascii('https://www.google.com/'))
  art = create_ascii('https://images.pexels.com/photos/5302784/pexels-photo-5302784.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1')

  for line in art:
    print(line)
```

HTML Templates

Create the Parent Template

Before we start with Python code, let's create the parent template and then the two children templates. This is a rather simple webpage. It uses the `style.css` file for styling, has an `<h1>` tag with the title, and has a single content block. The content block is inside a `<div>` with the class name `"container"`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
  <link rel="stylesheet"  type="text/css" href="
        {{url_for('.static', filename='style.css')}}">
    <title>Parent Template</title>
  </head>
  <body>
    <header>
      <h1 class="title">Create ASCII Art</h1>
    </header>

    <div class="container">
      {% block content %}
      {% endblock %}
    </div>

  </body>
</html>
```

Input Template

Open the `input.html` page. This will serve as the entry point for the user. They will enter a URL and click a button to convert it to ASCII art. Start by extending `parent.html` and set up a content block. In the block, create an `<h2>` tag with instructions to enter an image URL.

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

    <h2> Enter the URL of an image </h2>

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

Add a form with a text field and a submission button. Under the action, have it use the `/ascii` route (which we will define later). Give the form the name `url` so we can reference it. Set `"Image URL"` as the value. That way this text will appear in the text box. However, we want this text to disappear when the user clicks on the text box. Set `onfocus` to `"this.value=''"`. This will replace the text in the text box with an empty string when clicked. Give each `<input>` tag a unique ID for styling. In addition, add a `<details>` tag (which creates text hidden behind a dropdown menu) explaining how users can right-click on an image to copy its URL. Give this text a `<p>` tag and the `"findImg"` ID for styling.

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

    <h2> Enter the URL of an image </h2>
    <form action="{{ url_for('ascii') }}" method="POST">
      <input type="text" id="urlBox" name="url" value="Image
        URL" onfocus="this.value=''"><br>
      <input type="submit" id="urlButton" value="Create">
    </form>
    <details>
      <summary><strong>How do I get an image URL?</strong>
        </summary>
      <p id="findImg">Right-click on an image. You should see a
        window pop up. The action you select depends on your
        browser. It should say something like "Copy image
        address" or "Copy image link". Select the action that
        appears in your browser and paste into the box below.
        </p>
    </details>

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

Art Template

The `art.html` template is the page users see after clicking the button generate ASCII art. Use the link below to open the art template.

Like the above template, extend `parent.html` and create a content block in the `<body>`. Add a link to the content block that takes the user back “home”, that is the `input.html` page.

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

    <a href="{{ url_for('home') }}">Home</a>

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

What we show here depends on the result of the `create_ascii` function we made at the beginning of the lab. Either it returns a string with a message about entering a valid URL or it returns a list of strings. We can create conditionals in Flask with the `{% if %}` statement. This code assumes Python passes a dictionary to the code block. The dictionary has two key-value pairs. The first key is `'valid'` which contains a boolean value. If the `create_ascii` function returns ASCII art, the value is `True`. Add a for loop to the HTML document that iterates over the list stored in `'img'` key of the dictionary. Add each element from the list to the webpage followed by a `<br>` tag so each row of characters is on its own line. Place the ASCII art in a `<div>` with the ID `ascii-art` for styling purposes.

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

    {% if ascii['valid'] %}
      <h2> Converted Image </h2>
      <div id='ascii-art'>
        {% for line in ascii['img'] %}
          {{line}}<br>
        {% endfor %}
      </div>
    {% endif %}
    <a href="{{ url_for('home') }}">Home</a>

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

If the `valid` key is `False`, then add `{% else %}` before `{% endif %}`. Create a title as an `<h2>` tag and add an `<h3>` tag with the string message from the `create_ascii` function. The key for this information is called `img`. Both of these elements should have the class `urlMsg` for styling purposes.

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

    {% if ascii['valid'] %}
      <h2> Converted Image </h2>
      <div id='ascii-art'>
        {% for line in ascii['img'] %}
          {{line}}<br>
        {% endfor %}
      </div>
    {% else %}
        <h2 class="urlMsg"> Invalid URL </h2>
        <h3 class="urlMsg">{{ ascii['img'] }}</h3>
    {% endif %}
    <a href="{{ url_for('home') }}">Home</a>

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

▼ **Did you notice?**

> Since we were not interacting with Python, we did not start the Conda virtual environment. You must activate the virtual environment when working with Python as it has all of the required packages installed. It is not necessary to edit HTML pages. However, you must activate the virtual environment to run our Flask app and see the HTML pages rendered in the browser.

Coding the Flask App

Installing Flask and Setting the Home Route

With the templates out of the way, we are ready to code the Flack backend. Starting by activating the `ascii` virtual environment.

```
conda activate ascii
```

Then install the Flask package and any dependencies.

```
conda install flask -y
```

Use the link to open `main.py` so we can start coding the Flask backend for our app. Import the `Flask` package including `render_template` and `request`. Then import the `create_ascii` function from the `ascii_art` module. Create our app and the route for `/`. Render the `input.html` page when Flask loads. For now, create the `/ascii` route, but set the body of the function to `pass`. Finally, set the host to `0.0.0.0`, the port to `3000`, and debug to `True`.

```
from flask import Flask, render_template, request
from ascii_art import create_ascii

app = Flask(__name__)

@app.route('/')
def home():
  return render_template('input.html')

@app.route('/ascii')
def ascii():
  pass

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

```
python ascii-flask/main.py
```

Use the link below to preview the Flask app. While the page is running, click on the triangle to toggle the text about right-clicking on an image. In addition, click on the text box. Make sure the pre-set text disappears.

Displaying the ASCII Art

Because this function is interacting with the information in the `url` form, be sure to set `POST` as an allowable method for the `/ascii` route. Use `request.form` to take the information entered in the text box and store it in the `url` variable. Then create `img` and set its value to the `create_ascii` function. Be sure to pass the user URL to this function. If `create_ascii` returns a list, create the response dictionary with the `'valid'` key set to `True` and the `'img'` key set to `img`.

```
@app.route('/ascii', methods=['POST'])
def ascii():
  url = request.form['url']
  img = create_ascii(url)
  if type(img) == list:
    resp = {'valid': True, 'img': img}
```

If `create_ascii` does not return a list (the URL is not valid), create the response dictionary with `'valid'` set to `False` and `'img'` set to `img`. End the function by rendering the `art.html` template and pass it the response dictionary as `ascii`.

```
@app.route('/ascii', methods=['POST'])
def ascii():
  url = request.form['url']
  img = create_ascii(url)
  if type(img) == list:
    resp = {'valid': True, 'img': img}
  else:
    resp = {'valid': False, 'img': img}
  return render_template('art.html', ascii = resp)
```

There is one big change that still needs to be made. Right now, our site will not render the ASCII art properly. That is because font in the terminal is a monospace font. That means every character has a fixed width. The default font for webpages are not monospace. As such, characters will have variable widths and the image will appear distorted. Open the `style.css` document and set the font family for the ASCII art to `monospace`. We will style the rest of the site on the next page.

```css
#ascii-art {
  font-family: monospace;
}
```

Our Flask app is ready to run. Open the tab with the terminal and enter the command below.

```
python ascii-flask/main.py
```

Click the link to view the preview of the site. Use the link below as a test image for the webpage. You should see the same silhouette as before.

```
https://images.pexels.com/photos/5302784/pexels-photo-
5302784.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1
```

challenge

Try these variations:

- Enter invalid image URLs to verify that the site properly displays a message to users.

```
conda deactivate
```

Styling the Flask App

Adding CSS to the Webpage

Start the `ascii` virtual server.

```
conda activate ascii
```

We have already changed the font family for the ASCII art image, but now it is time to style the rest of the Flask app. Let’s start with the body. Change the background color to a dark blue, set the color of the text to a light gray, and change the font family to a sans-serif font.

```
#ascii-art {
  font-family: monospace;
}

body {
  background-color: #34495e;
  color: #ecf0f1;
  font-family: sans-serif;
}
```

In the terminal, run the Flask server. Then click the link to see the preview. Leave the server running.

```
python ascii-flask/main.py
```

Any time you make a change to the CSS file, go to the preview and refresh the page by clicking the two blue arrows in the shape of a circle. The rest of this page assumes you will be refreshing the page as we go.

Next, we want to style the main title for the site. Invoke the `.title` class and center the text and set its size to 60 pixels. The `.container` class contains the content block for each view. Set the sizeof the text to 16 pixels, center the text, set the width of the container to 90% of the page, and horizontally center the container. By making the container smaller than the width of the page (and centering the `<div>`), we give our site margins on either side.

```
.title {
  text-align: center;
  font-size: 60px;
}

.container {
  font-size: 16px;
  text-align: center;
  width: 90%;
  margin: auto;
}
```

The `.urlMsg` class represents the message about an invalid URL. Set this text to a shade of red. Then set the `<h2>` and `<h3>` text to 36 and 24 pixels respectively.

```
.urlMsg {
  color: #e74c3c;
}

h2 {
  font-size: 36px;
}

h3 {
  font-size: 24px;
}
```

Style the `<input>` tags so that the font is 16 pixels, there are no border colors, and there is a slight curve to the button and input box. The padding adds a bit of space around the text inside the elements, and the top margin separates the elements slightly.

```
input {
  font-size: 16px;
  border: none;
  border-radius: 4px;
  padding: 5px;
  margin-top: 5px;
}
```

We want there to be some visual indicator about interacting with the button. Set the background color to a light blue and the text color the same light gray as the other text. When the mouse hovers over the button, change the blue to a slightly darker shade. The text color in the text box should be the same dark blue as the background of the webpage.

```
#urlButton {
  background-color: #3498db;
  color: #ecf0f1;
}

#urlButton:hover {
  background-color: #2980b9;
}

#urlBox {
  color: #34495e;
}
```

The dropdown about finding the image URL needs to be confined to a space narrower than the parent `<div>`. Set its width to 40% and center the text (this is the title of dropdown, not the dropdown text). Center this element with `margin: auto;` and add a 5 pixel margin above this element. For the text inside the dropdown, create a two-pixel border on the left that is light blue. Add 5 pixels of padding between the text and the border. Make the text left justified. Finally, make the link text light blue.

```
details {
  width: 40%;
  text-align: center;
  margin: auto;
  margin-top: 5px;
}

#findImg {
  border-left: 2px solid #3498db;
  padding-left: 5px;
  text-align: left;
}

a {
  color: #3498db;
  font-size: 16px;
}
```

challenge

Try this variation:

- Change the styling for the link so that it changes color once it has been visited. Add this styling so that the link becomes orange after you have clicked it.

```css
a:visited {
  color: #f39c12;
}
```

▼ **Code**

Your code should look like this:

```css
#ascii-art {
  font-family: monospace;
}

body {
  background-color: #34495e;
  color: #ecf0f1;
  font-family: sans-serif;
}

.title {
  text-align: center;
  font-size: 60px;
}

.container {
  font-size: 16px;
  text-align: center;
  width: 90%;
  margin: auto;
}

.urlMsg {
  color: #e74c3c;
}

h2 {
  font-size: 36px;
}
```

```css
h3 {
  font-size: 24px;
}

input {
  font-size: 16px;
  border: none;
  border-radius: 4px;
  padding: 5px;
  margin-top: 5px;
}

#urlButton {
  background-color: #3498db;
  color: #ecf0f1;
}

#urlButton:hover {
  background-color: #2980b9;
}

#urlBox {
  color: #34495e;
}

details {
  width: 40%;
  text-align: center;
  margin: auto;
  margin-top: 5px;
}

#findImg {
  border-left: 2px solid #3498db;
  padding-left: 5px;
  text-align: left;
}

a {
  color: #3498db;
  font-size: 16px;
}
```

Deactivate the virtual environment.

```
conda deactivate
```

Dockerfile

Ignoring Files

We could copy over all of the files from our project to the Docker image. However, there are many files that would be copied but not used by our Flask app. Instead of listing individual copy commands for the files we want, we can make an ignore file. This file tells Docker which files and directories to ignore. Start by creating the `.dockerignore` file. Notice the `.` in front of the file name. This means the file is hidden.

```
touch .dockerignore
```

Use the link below to open the ignore file. The `.git` refers to any files generated by git as we add, commit, and push with git. The `.guides` and `.settings` are directories and files used by Codio to store information like the content of this page and many other things. These files are needed to display the instructional content, but have non bearing on the Flask app.

```
.git
.guides
.settings
```

Now we are ready to create the Dockerfile. When we copy contents of our project over to the Docker image, Docker will only include those files it needs to run the Flask app, build the Docker image, and deploy on Heroku.

Preparing our Dockerfile

Next, we need to create our Dockerfile. Use the link below to return to the terminal. Then enter the command to create the Dockerfile. This file must start with a capital "D".

```
touch Dockerfile
```

Similar to our previous Docker example, we are going to use the continuumio image, which is built and maintained by the team at Anaconda. All of the dependencies for our Flask app (requests, Flask, and

Pillow) are included with this image. There is nothing for us to install. Use the link to open the file. Then copy and paste the first line into it.

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

The remainder of the Dockerfile is three lines. The copy command copies everything (minus the files mentioned in the ignore file) to the Docker image. When we set the working directory to `ascii-flask` which contains our Flask code. Finally, we run the `main.py` script with the container starts.

```
FROM continuumio/anaconda3:2021.11

COPY . /

WORKDIR /ascii-flask

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

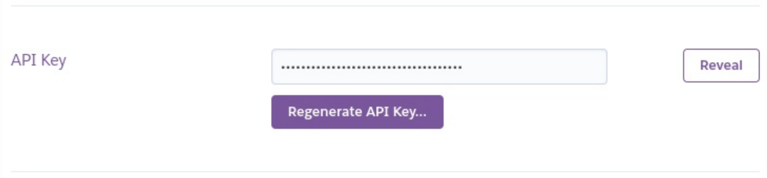
Deploying to Heroku

Creating the Heroku App

You should already have an account with Heroku (the CLI application is already installed). Login to your Heroku account with the following command.

```
heroku login -i
```

Remember, you cannot use your Heroku password with the CLI application. Instead, use the API key, which is found under you account settings on the Heroku site.



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.

Next, create your Heroku application for this project. The name should be unique so adding something like your initials to the template below may help. In addition, we want to add the Python buildpack to our Heroku application. Make a note of the URL Heroku generates for your application. We will need this to check if our Flask app is running properly.

```
heroku create ascii-flask-<your initials> --buildpack
        heroku/python
```

▼ **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.

The last thing we need to do is set the Heroku stack to run a container. Use the following command to do this; be sure to substitute the name of your Heroku application in the template below.

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

Heroku YAML File and Flask App

```
touch heroku.yml
```

Heroku follows the commands set forth in the `heroku.yml` file. Add the snippet of code to the top-left panel. This tells Heroku to build the Docker container according to the Dockerfile. From there, Heroku will use the entry point in the Dockerfile to start the container. Use the link to open the file and copy/paste the code into it.

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

The last thing we need to do is make some changes to our Flask app for deployment on the internet. Use the link below to open this file.

Import the `os` module so we can set an environment variable for the script.

```
import os
from flask import Flask, render_template, request
from ascii_art import create_ascii
```

At the bottom of the Python file, get the `PORT` environment variable and use it when starting the Flask app. Then change the debug mode to `False`.

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

The last steps are to use git to push our code to a Heroku git repository. From there, Heroku will deploy our Flask app to the internet. Use the link to open the terminal. Then add our changes.

```
git add .
```

Next, 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
```

Once this process finishes, your webpage is live. Use the Heroku URL to visit the site. If you cannot remember your URL, use the following command in the terminal.

```
heroku open
```

If you were to run this command on your local computer (and not in Codio), Heroku would open the browser and load your Flask app. However, Heroku throws an error because it cannot open a browser inside Codio. However, it does list the URL. Click on the link, and your webpage will load in a new tab.

```
▸    Error opening web browser.
 ▸    Error: Exited with code 3
 ▸
 ▸    Manually visit https://ascii-flask.herokuapp.com/ in your
browser.
```

Pushing to GitHub

Pushing to GitHub

Before leaving, add this project to your GitHub portfolio. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finished ASCII Flask app"
```

- Push to GitHub:

```
git push origin master
```

▼ **Did you notice?**

> This git command uses `origin master` and not `heroku master`. For this project, we are using two different git repositories. The first one gave us the template code at the very beginning of the lab. The second is the Heroku repository. Using `origin master` saves your Flask app code to your portfolio.

Lab Challenge

Lab Challenge

Problem

Login to the Heroku CLI. **Remember**, use your API key instead of your password.

```
heroku login -i
```

Create a Heroku app called `lab-challenge-<unique id>` where `<unique-id>` is a unique identifier for your app. **Important**, your app must start with `lab-challenge-`. Modify the app such that it will runn off of a container.

Expected Output

Once you have created the Heroku app, list the information about the app and redirect the output to the file `solution.txt`. **Important**, replace `your-app-name` with the name of the app you created. It should start with `lab-challenge-` and end with the unique identifier you gave it.

```
heroku apps:info --app=your-app-name > solution.txt
```

Use the `cat` tool to view the contents of the `solution.txt` file in the terminal.

```
cat solution.txt
```

Your output should be similar to that below. The git and web URLs should be different due to the unique name for you Heroku app. However, they should still start with `lab-challenge-`.

```
=== lab-challenge-codio
Auto Cert Mgmt: false
Dynos:
Git URL:        https://git.heroku.com/lab-challenge-codio.git
Owner:          your-email@provider.com
Region:         us
Repo Size:      0 B
Slug Size:      0 B
Stack:          container
Web URL:        https://lab-challenge-codio.herokuapp.com/
```