Add Security to Your RESTful Service

Skills Network

Estimated time needed: 60 minutes

An important code practice is to ensure that your microservice is sending back security headers and has established cross-origin resource sharing (CORS) policies to avoid potential vulnerability exploitation.

Welcome to the **Add Security to Your RESTful Service** hands-on lab. In this lab, you will implement the second story in Sprint 2, "*Need to add security headers and CORS policies*" by adding `Flask-Talisman` and `Flask-Cors` to your project.

Objectives

In this lab, you will:

- Take the next story from the Sprint Backlog to work on
- Add Flask-Talisman for security headers
- Add Flask-Cors to establish cross-origin resource sharing (CORS) policies
- View the results of your changes
- Make a pull request and merge your changes after the CI tests pass
- Move the story to "Done"

Note: Important Security Information

Welcome to the Cloud IDE with Docker. This is where all your development will take place. It has all the tools you will need to use Docker for deploying a PostgreSQL database.

It is important to understand that the lab environment is **ephemeral**. It only lives for a short while before it is destroyed. This makes it imperative that you push all changes made to your own GitHub repository so that it can be recreated in a new lab environment any time it is needed.

Also note that this environment is shared and therefore not secure. You should not store any personal information, usernames, passwords, or access tokens in this environment for any purposes.

Finally, the environment may get recreated at any time so you may find that you have to preform the **Initialize Development Environment** each time the environment is created.

Note on Screenshots

Throughout this lab, you will be prompted to take screenshots and save them on your device. You will need these screenshots to either answer graded quiz questions or upload as your submission for peer review at the end of this course. Your screenshot must have either the .jpg or .png extension.

To take screenshots, you can use various free screen-capture tools or your operating system's shortcut keys. For example:

- **Mac:** you can use `Shift + Command + 3` (⇧ + ⌘ + 3) on your keyboard to capture your entire screen, or `Shift + Command + 4` (⇧ + ⌘ + 4) to capture a window or area. They will be saved as a file on your Desktop.
- **Windows:** you can capture your active window by pressing `Alt + Print Screen` on your keyboard. This command copies an image of your active window to the clipboard. Next, open an image editor, paste the image from your clipboard to the image editor, and save the image.

Initialize Development Environment

Because the Cloud IDE with Docker environment is ephemeral, it may be deleted at any time. The next time you come into the lab, a new environment may be created. Unfortunately, this means that you will need to initialize your development environment every time it is recreated. This shouldn't happen too often as the environment can last for several days at a time but when it is removed, this is the procedure to recreate it.

Overview

Each time you need to set up your lab development environment you will need to run three commands.

Each command will be explained in further detail, one at a time, in the following section.

`{your_github_account}` represents your GitHub account username.

The commands include:

```
git clone https://github.com/{your_github_account}/devops-capstone-project.git
cd devops-capstone-project
bash ./bin/setup.sh
exit
```

Copied!

Now, let's discuss each of these commands and explain what needs to be done.

Task Details

Initialize your environment using the following steps:

1. Open a terminal with `Terminal` -> `New Terminal` if one is not open already.
2. Next, use the `export GITHUB_ACCOUNT=` command to export an environment variable that contains the name of your GitHub account.

   > Note: Substitute your GitHub username for the `{your_github_account}` placeholder below:

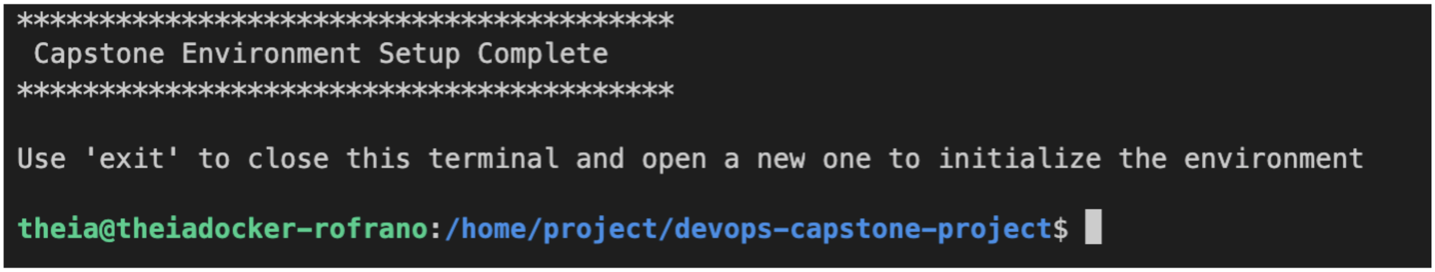
   ```
   export GITHUB_ACCOUNT={your_github_account}
   ```

   Copied!
3. Then use the following commands to clone your repository, change into the `devops-capstone-project` directory, and execute the `./bin/setup.sh` command.

   ```
   git clone https://github.com/$GITHUB_ACCOUNT/devops-capstone-project.git
   cd devops-capstone-project
   bash ./bin/setup.sh
   ```

   Copied! Executed!

   You should see the following at the end of the setup execution:



```
****************************************
 Capstone Environment Setup Complete
****************************************

Use 'exit' to close this terminal and open a new one to initialize the environment

theia@theiadocker-rofrano:/home/project/devops-capstone-project$ 
```

4. Finally, use the `exit` command to close the current terminal. The environment will not be fully active until you open a new terminal in the next step.

   ```
   exit
   ```

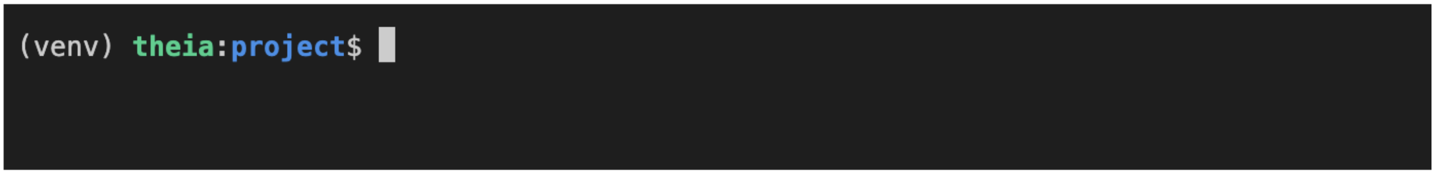
   Copied! Executed!

Validate

In order to validate that your environment is working correctly, you must open a new terminal because the Python virtual environment will only activate when a new terminal is created. You should have ended the previous task by using the `exit` command to exit the terminal.

1. Open a terminal with `Terminal` -> `New Terminal` and check that everything worked correctly by using the `which python` command:

   Your prompt should look like this:



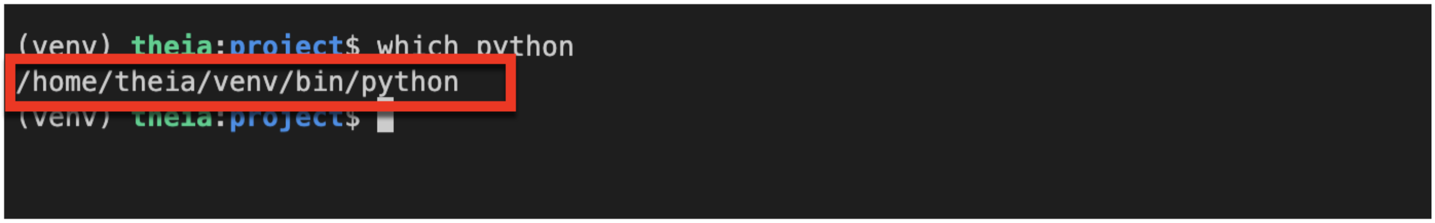
```
(venv) theia:project$ 
```

   Check which Python you are using:

   ```
   which python
   ```

   Copied! Executed!

   You should get back:



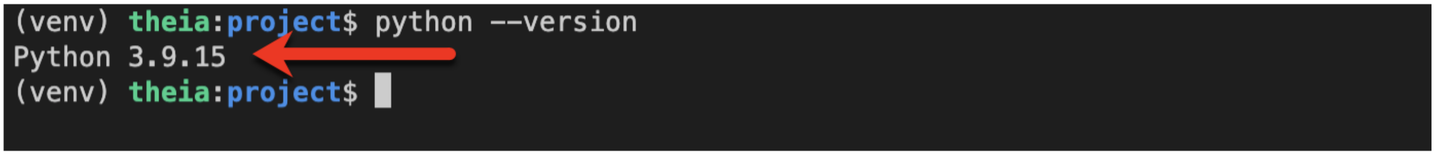
```
(venv) theia:project$ which python
/home/theia/venv/bin/python
(venv) theia:project$ 
```

   Check the Python version:

   ```
   python --version
   ```

   Copied! Executed!

   You should get back some patch level of Python 3.9:



```
(venv) theia:project$ python --version
Python 3.9.15
(venv) theia:project$ 
```

This completes the setup of the development environment. Anytime your environment is recreated, you will need to follow this procedure.

You are now ready to start working.

Exercise 1: Pick Up the Next Story

The first thing you need to do is to go to Zenhub to get a story to work on. Take the next story from the top of the Sprint Backlog, move it to the **In Progress** column, assign it to yourself, and read the contents.

Your Task

1. Go to your kanban board and take the next story from the top of the `Sprint Backlog` column. It should be titled: "*Need to add security headers and CORS policies*".
2. Move the story to `In Progress`.
3. Open the story and assign it to *yourself*.
4. Read the contents of the story.

Results

The story should look like this:

Need to add security headers and CORS policies

As a service provider
I need my service to use security headers and CORS policies
So that my web site is not vulnerable to CORS attacks

Assumptions

- Flask-Talisman will be used for security headers
- Flask-Cors will be used to establish cross-origin resource sharing (CORS) policies

Acceptance Criteria

```
Given the site is secured
When a REST API request is made
Then secure headers and a CORS policy should be returned
```

Copied!

You are now ready to begin working on your story.

Exercise 2: Observe the Current Behavior

In reading your story, you see that the first assumption is:

- Flask-Talisman will be used for security headers

This will be the first change you make. But before you do, it's a good idea to observe the current behavior so that you have something to compare to once you add Talisman.

Your Task

1. Open a terminal and make sure that you are in the `/home/project/devops-capstone-project` folder.

   ```
   cd /home/project/devops-capstone-project
   ```

   Copied! Executed!

2. Use the `honcho start` command to start your microservice running in the terminal listening on port `5000`.

   ```
   honcho start
   ```

   Copied! Executed!

3. Open another terminal (`Terminal` -> `New Terminal`) and use the following `curl` command to see the headers that are being returned.

   ```
   curl -I localhost:5000
   ```

   Copied! Executed!

4. Go back to the first terminal and use `Ctrl+C` to stop the running server.

Results

You should see output similar to the following:

```
HTTP/1.1 200 OK
Server: gunicorn
Date: Thu, 13 Oct 2022 12:07:32 GMT
Connection: close
Content-Type: application/json
Content-Length: 52
```

Copied!

Make a mental note of this output. Once you add Talisman, you will see security headers included.

Exercise 3: Write a Test Case

In reading your story, you see that the first assumption is:

- Flask-Talisman will be used for security headers

In keeping with test driven development (TDD) practices, you want to write a test case to test for the behavior that you are about to implement. While the test case will fail at first, it will also let you know when the implementation is successful.

Flask-Talisman forces your REST API clients to use the `HTTPS` protocol. This means that if you want to test this behavior, your test case must use `https://` in order to work. To get the Flask test client to use `https` you can use the `environ_overrides` attribute.

Your Task

You will be working with the `./tests/test_routes.py` file:

Open **test_routes.py** in IDE

1. Use the `git checkout -b` command to create a new branch to work on in the development environment.

2. Run `nosetests` and make sure that all of the test cases are passing. Fix any that fail before proceeding.

3. Edit `tests/test_routes.py` and add the following line of code toward the top of the file after the line that defines the `BASE_URL` global variable.

   ```
   HTTPS_ENVIRON = {'wsgi.url_scheme': 'https'}
   ```

   Copied!

4. Write a test case that calls the root URL "`/`" passing in `environ_overrides=HTTPS_ENVIRON` as a parameter and asserting the presence of the following headers and their values in the output:

```
'X-Frame-Options': 'SAMEORIGIN'
'X-XSS-Protection': '1; mode=block'
'X-Content-Type-Options': 'nosniff'
'Content-Security-Policy': 'default-src \'self\'; object-src \'none\''
'Referrer-Policy': 'strict-origin-when-cross-origin'
```

Copied!

▼ Click here for the answer.
This is the test case for checking the headers that Talisman adds:

```
    def test_security_headers(self):
        """It should return security headers"""
        response = self.client.get('/', environ_overrides=HTTPS_ENVIRON)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        headers = {
            'X-Frame-Options': 'SAMEORIGIN',
            'X-XSS-Protection': '1; mode=block',
            'X-Content-Type-Options': 'nosniff',
            'Content-Security-Policy': 'default-src \'self\'; object-src \'none\'',
            'Referrer-Policy': 'strict-origin-when-cross-origin'
        }
        for key, value in headers.items():
            self.assertEqual(response.headers.get(key), value)
```

Copied!

5. Run `nosetests` and watch that test case fail with an `AssertionError` because it cannot find the headers.

You are now ready to write the code to add the missing security headers.

Exercise 4: Add Security Headers

In this step, you will add Talisman to your requirements file and add code to use Talisman in your microservice.

You will be working with the `./service/__init__.py` file:

Open **__init__.py** in IDE

Your Task

1. Add `Flask-Talisman` to your `requirements.txt` file.

2. Use `pip` to install the new requirements using `-r requirements.txt`.

3. Open the `service/__init__.py` file and import the `Talisman` class from `flask_talisman`.

   ▼ Click here for the answer.

   ```
   from flask_talisman import Talisman
   ```

   Copied!

4. Next, after the Flask app is created, create an instance of the `Talisman` class called `talisman` passing in the Flask `app` to the class constructor.

   ▼ Click here for the answer.

   ```
   talisman = Talisman(app)
   ```

   Copied!

5. Run `nosetests tests/test_routes.py` to run only the tests for the routes and observe the output.

Results

All of your test cases have failed except for the new one that tests for the security headers. You will fix these errors in the next exercise.

Exercise 5: Disable Forced https

By default, Talisman will force all requests to your REST API to use the `https://` protocol. This is a good thing, except perhaps when testing. Luckily, Talisman gives you a way to turn this behavior on and off.

In this exercise, you will disable forced `https` by setting `force_https = False` on the `talisman` instance from your service.

Your Task

1. Open `tests/test_routes.py` and import `talisman` from the `service` package.

   ▼ Click here for the answer.

   ```
   from service import talisman
   ```

   Copied!

2. Update the `setUpClass()` method by adding the line: `talisman.force_https = False`

   ▼ Click here for the answer.

   ```
   @classmethod
   def setUpClass(cls):
       """Run once before all tests"""
       { other lines of code here ... }
   ```

```
    talisman.force_https = False
```

Copied!

3. Run `nosetests tests/test_routes.py` again and all of the tests should now pass.

4. Finally, when all the tests have passed, use the `git commit -am` to commit your changes with the message `Added security headers`.

Make sure that all of the tests pass before moving on to the next exercise.

Exercise 6: Validate Security Headers

Let's call the REST API again and see how the headers have changed.

1. Go to the first terminal that was running your microservice and start it again using `honcho start`.

2. In the second terminal, use `curl` to see the new headers:

```
curl -I localhost:5000
```

Copied! Executed!

3. Go back to the first terminal that was running your microservice and use `Ctrl+C` to stop the service.

> Note: You may have to press Ctrl+C more than once to stop it.

Results

You should see output similar to this:

```
HTTP/1.1 302 FOUND
Server: gunicorn
Date: Thu, 13 Oct 2022 20:00:01 GMT
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 233
Location: https://localhost:5000/
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Content-Security-Policy: default-src 'self'; object-src 'none'
Referrer-Policy: strict-origin-when-cross-origin
```

Copied!

Notice the new headers that have been added since the previous output. Also notice that you did not get back a `200 OK` but rather `302 FOUND`. This is because `curl` uses `http` by default and the service is sending a `302` return code with a `Location` header to tell the browser to redirect the request to `https://localhost:5000/`.

Since you don't have a proper endpoint configured for `https`, you cannot call it with `curl`, but at least you can see that the new security headers are working.

Exercise 7: Add CORS Policies

Going back to the story that you are working on, you see that the second assumption is

- Flask-Cors will be used to establish cross-origin resource sharing (CORS) policies

This will be the next change you make.

Your Task

1. Write a test case that calls the root URL `"/"` passing in `environ_overrides=HTTPS_ENVIRON` as a parameter and asserting the presence of the header `Access-Control-Allow-Origin: *` in the output.

   ▼ Click here for the answer.
   This is the test case for checking that the CORS headers is present:

```
    def test_cors_security(self):
        """It should return a CORS header"""
        response = self.client.get('/', environ_overrides=HTTPS_ENVIRON)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        # Check for the CORS header
        self.assertEqual(response.headers.get('Access-Control-Allow-Origin'), '*')
```

   Copied!

2. Run `nosetests` and watch that test case fail with an `AssertionError` because it cannot find the headers.

3. Add `Flask-Cors` to your `requirements.txt` file.

4. Use `pip` to install the new requirements using `-r requirements.txt`.

5. Open the `service/__init__.py` file and import the `CORS` class from `flask_cors`.

   ▼ Click here for the answer.

```
from flask_cors import CORS
```

   Copied!

6. Next, after the Talisman class is created, create the `CORS` class passing in the Flask `app`.

   ▼ Click here for the answer.

```
CORS(app)
```

Copied!

7. Run `nosetests` to run all of the unit tests and observe the output.

8. Finally, when all the tests have passed, use `git commit -am` to commit your changes with the message `Added CORS headers`.

Results

All of your test cases should pass.

Exercise 8: Validate CORS Headers

Let's call the REST API again and see how the headers have changed.

1. Go to the first terminal that was running your microservice and start it again using `honcho start`.

2. In the second terminal, use `curl` to see the new headers:

```
curl -I localhost:5000
```

Copied! Executed!

3. Go back to the first terminal that was running your microservice and use `Ctrl+C` to stop the service.

> Note: You may have to press Ctrl+C more than once to stop it.

Results

You should see output similar to this:

```
HTTP/1.1 302 FOUND
Server: gunicorn
Date: Thu, 13 Oct 2022 20:18:54 GMT
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 233
Location: https://localhost:5000/
Access-Control-Allow-Origin: *
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Content-Security-Policy: default-src 'self'; object-src 'none'
Referrer-Policy: strict-origin-when-cross-origin
```

Copied!

Notice that there is one additional header `Access-Control-Allow-Origin: *`. You can, of course, add a policy to CORS and change this header. It all depends on what your application needs.

Exercise 8: Make a Pull Request

Now that you have completed your GitHub Action, you are ready to commit your changes, push code to your GitHub repository, and make a pull request.

Your Task

1. Use `git status` to make sure that you have committed your changes locally in the development environment.

2. Push your local changes to a remote branch.

3. Make a pull request, which should kick off the GitHub Action that is now enabled on your repository.

Evidence

1. Open the `service/__init__.py` file.
2. Take a screenshot and save it as `security-code-done.jpg` (or `security-code-done.png`).
3. Take a screen shot of the output from **Exercise 7** and save it as `security-headers-done.jpg` (or `security-headers-done.png`).
4. Move your story to the `Done` column on your kanban board.
5. Take a screenshot of your kanban board and save it as `security-kanban-done.jpg` (or `security-kanban-done.png`).

Conclusion

Congratulations! You have implemented secure code practices for your microservice. You also experienced how GitHub Actions will run on every pull request and that any code you add will not break the build.

Next Steps

This completes Sprint 2. Next, you will start Sprint 3.

Author

John J. Rofrano

Other Contributor(s)

Change Log

Date	Version	Changed by	Change Description
2022-10-12	0.1	John Rofrano	Initial version created
2022-10-14	0.2	John Rofrano	Updated screenshot image names

Date	Version	Changed by	Change Description
2022-10-25	0.3	Beth Larsen	QA pass
2022-10-28	0.4	John Rofrano	Updated markdown in story text
2023-03-16	0.5	Lavanya Rajalingam	Updated SN Logo