cisco Live!

Let's go

The bridge to possible

Build your own features by taking your python to the next level

Bryn Pounds, Principal Architect
@brynpounds

Agenda

- Introduction
- What we typically do
- API First
- Swagger & Open API
- Simple UI
- Package it with Docker
- Conclusions

Agenda

- Introduction
- Check out my cool python script!
- API First
- Swagger & Open API
- Simple UI
- Package it with Docker
- Considerations for API hardening and scaling
- Conclusions

Check out my Python script...

(...and some problems that come with it)

We all start somewhere…

After 4 days of python training, this was my first program (2012)…

```python
import paramiko
import time
import socket
import string

chassis_list=['10.0.2.241','10.0.2.242']
chassis_username = 'admin'
chassis_password = 'password'


def show_chassis_status(self):
    '''Sequence for powering on my home servers'''
    delete_table  = string.maketrans(
    string.ascii_lowercase, ' ' * len(string.ascii_lowercase))

    #print "made it to the def with variable " + self
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    try:
        ssh.connect(self,username=chassis_username, password=chassis_password, timeout=10)
        #print "connected"
        c = ssh.invoke_shell()
        c.send('show chassis\n')
        time.sleep(2)
        output = c.recv(2048)
        print "Power Status of {} is {}".format(self, output[213:217].translate(None, delete_table))
    except paramiko.AuthenticationException:
        print "Authentication problem with " + self
    except socket.timeout:
        print "We don't seem able to connect to " + self
```

```python
def power_on_chassis(self):
    '''Sequence for powering on my home servers'''
    delete_table  = string.maketrans(
    string.ascii_lowercase, ' ' * len(string.ascii_lowercase))
    try:
        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh.connect(self,username=chassis_username, password=chassis_password, ti
        print "connected to " + self

        c = ssh.invoke_shell()
        c.send('scope chassis\n')
        time.sleep(2)
        print c.recv(2048)

        c.send('power on\n')
        time.sleep(2)
        print c.recv(2048)

        c.send('y\n')
        time.sleep(2)
        print c.recv(2048)
    except paramiko.AuthenticationException:
        print "Authentication problem with " + self
    except socket.timeout:
        print "We don't seem able to connect to " + self

    ssh.close()
```

We all start somewhere…

After 4 days of python training, this was my first program (2012)…

```python
import paramiko
import time
import socket
import string

chassis_list=['10.0.2.241','10.0.2.242']
chassis_username = 'admin'
chassis_password = 'password'


def show_chassis_status(self):
    '''Sequence for powering on my home servers'''
    delete_table  = string.maketrans(
    string.ascii_lowercase, ' ' * len(string.ascii_lowercase))

    #print "made it to the def with variable " + self
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    try:
        ssh.connect(self,username=chassis_username, password=chassis_password, timeout=10)
        #print "connected"
        c = ssh.invoke_shell()
        c.send('show chassis\n')
        time.sleep(2)
        output = c.recv(2048)
        print "Power Status of {} is {}".format(self, output[213:217].translate(None, delete_table))
    except paramiko.AuthenticationException:
        print "Authentication problem with " + self
    except socket.timeout:
        print "We don't seem able to connect to " + self
```

Hard Coded (everything)!

```python
def power_on_chassis(self):
    '''Sequence for powering on my home servers'''
    delete_table  = string.maketrans(
    string.ascii_lowercase, ' ' * len(string.ascii_lowercase))
    try:
        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh.connect(self,username=chassis_username, password=chassis_password, ti
        print "connected to " + self

        c = ssh.invoke_shell()
        c.send('scope chassis\n')
        time.sleep(2)
        print c.recv(2048)

        c.send('power on\n')
        time.sleep(2)
        print c.recv(2048)

        c.send('y\n')
        time.sleep(2)
        print c.recv(2048)
    except paramiko.AuthenticationException:
        print "Authentication problem with " + self
    except socket.timeout:
        print "We don't seem able to connect to " + self

    ssh.close()
```

EXPECT Scripting

We all start somewhere…

And - What an AMAZING user experience!!!

```
Welcome to Bryn's Handy Dandy UCS-C Series management tool!!!
What would you like to do?

1 = Power them up?
2 = Power them down?
3 = See the Status?
4 = Exit this script

1
Are you sure you want to power them up?
  Y or N?

Y
You're the Boss!!!   Powering them up...
```

Shortly after this, the lab admin heard I had this and asked if she could have it.

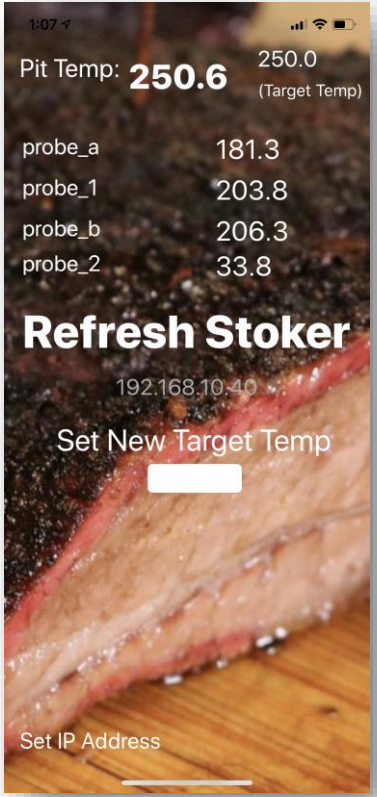
I *proudly* and happily shared my code with her.

She never got it to run.

Oh the memories! Good Times!

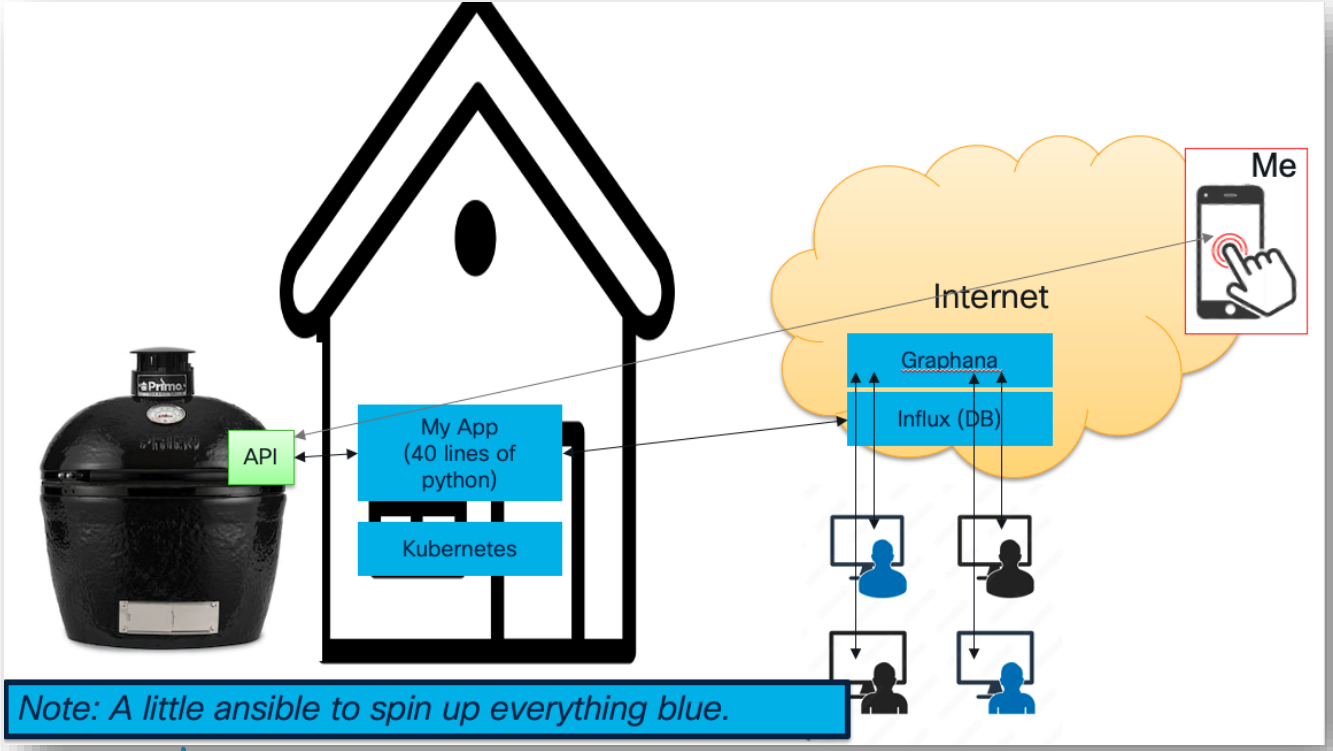
A few years later… BBQ Telemetry

Things got a little better…



Getting Better. Still lots of room for Improvement

Still not thinking like a “Developer”



WHAT ABOUT?

API First?
API Documentation?
API Hardening?
Scale Testing?
Versioning?
Packaging?
Distribution?

Not just for BBQ. This applies to anything.

This is Cisco Live, so how are you programmatically touching the network?

- Interact with Cisco Controllers…
  - Catalyst Center
  - Meraki
  - Nexus Dashboard Fabric Controller
  - ACI
- CI/CD Pipelines
- Digital Twin leveraging Cisco Modeling Labs
- <Your Idea Here>

Try my BoilerPlate...

- Been using this as a quick start for years...

```
git clone https://github.com/brynpounds/bryn_boilerplate
cd bryn_boilerplate/
docker-compose up --build bryn_boilerplate
```

(Save time by skipping the entire build after initial run)

```
docker-compose up bryn_boilerplate
```

Requirements.txt – (*Always* needs updating)

bryn_boilerplate / app / requirements.txt

brynpounds Update requirements.txt

Code | Blame | 15 lines (13 loc) · 197 Bytes | Code 55% faster with GitHub Copilot

```
Flask-RESTful==0.3.10
Flask==2.3.3
gunicorn==21.2.0
python-dotenv==1.0.0
flask-apispec==0.11.4
jinja2
requests=2.31.0
#lxml==4.9.3
#ncclient==0.6.13
#xmltodict==0.13.0
#utils
#pytz>=2015.7
#fpdf
```

Commit

Update requirements.txt

main

brynpounds committed 2 minutes ago Verified

Showing 1 changed file with 12 additions and 13 deletions.

25 app/requirements.txt

```
@@ -1,16 +1,15 @@
- Flask-RESTful==0.3.7
- Flask==1.1.1
- gunicorn==20.0.4
- python-dotenv==0.19.0
- flask-apispec==0.11.0
+ Flask-RESTful==0.3.10
+ Flask==2.3.3
+ gunicorn==21.2.0
+ python-dotenv==1.0.0
+ flask-apispec==0.11.4
  jinja2
- requests
- lxml==4.9.3
- ncclient==0.6.13
- urllib3==1.26.14
- xmltodict==0.13.0
- utils
- pytz>=2015.7
- fpdf
+ requests=2.31.0
+ #lxml==4.9.3
+ #ncclient==0.6.13
+ #xmltodict==0.13.0
+ #utils
+ #pytz>=2015.7
+ #fpdf
```

API First

What is API First?

API First is the way a developer approaches writing apps.

- First step is to think about all the functions that will make up your app.
  - ***"I need to generate a custom report"***
    a) I'll need to log into Catalyst Center
    b) I'll need to pull the inventory
    c) I'll need to be able to parse and filter that inventory
    d) I'll need to format things a specific way.
    e) I'll need to deliver the info via an email
    f) I'll need to deliver the info via a PDF file

What is API First?

API First is the was a developer approaches writing apps.

- First step is to think about all the functions that will make up your app.
  - *“I need to generate a custom report”*
    a) I’ll need to log into Catalyst Center
    b) I’ll need to pull the inventory
    c) I’ll need to be able to parse and filter that inventory
    d) I’ll need to format things a specific way.
    e) I’ll need to deliver the info via an email
    f) I’ll need to deliver the info via a PDF file

This shouldn’t be a big deal at least conceptually.

After we get a little experience, most of us start to leverage “def” functions.

But how do we makes these “def”’s available via an API call?

Introducing Flask

Flask for today's discussion. There are several other options.

What is Flask Python

Flask is a web framework, it's a Python module that lets you develop web applications easily. It's has a small and easy-to-extend core: it's a microframework that doesn't include an ORM (Object Relational Manager) or such features.

It does have many cool features like url routing, template engine. It is a WSGI web app framework.

Related course: Python Flask: Create Web Apps with Flask

What is a Web Framework?

A Web Application Framework or a simply a Web Framework represents a collection of libraries and modules that enable web application developers to write applications without worrying about low-level details such as protocol, thread management, and so on.

What is Flask?

Flask is a web application framework written in Python. It was developed by Armin Ronacher, who led a team of international Python enthusiasts called Poocco. Flask is based on the Werkzeg WSGI toolkit and the Jinja2 template engine.Both are Pocco projects.

https://pythonbasics.org/what-is-flask-python/

Flask is one of the most popular Python based Web Frameworks for building REST API's.

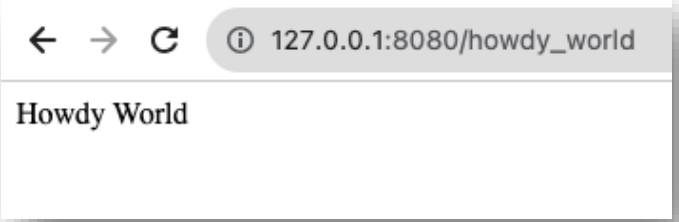
(Many use it for UI work – we will today)

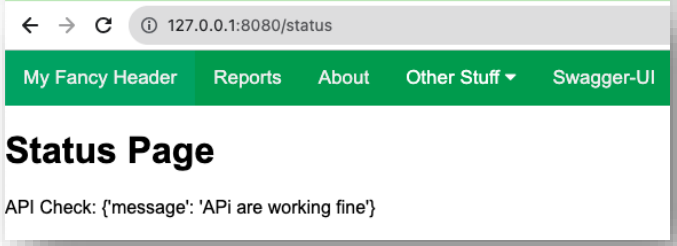
Flask
web development,
one drop at a time

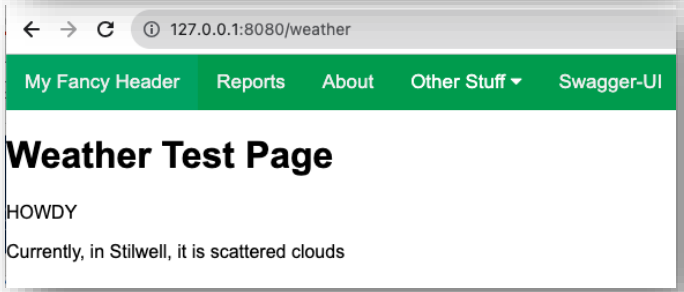
FLASK Example

Flask for today's discussion. There are several other options.

```python
@app.route("/howdy_world")
def howdy():
    return "<p>Howdy World</p>"

@app.route("/test")
def test():
    return render_template("test.html")

@app.route("/status")
def status():
    url = 'http://localhost:8080/health_check'
    response = requests.get(url)
    our_response_content = response.content.decode('utf8')
    proper_json_response = json.loads(our_response_content)
    return render_template("status.html", testing=proper_json_response)

@app.route("/weather")
def weather():
    headers = {
        'accept': 'application/json',
        'Content-Type': 'application/json',
    }
```







https://pythonbasics.org/what-is-flask-python/

FLASK Example

Flask for today's discussion. There are several other options.

```
@app.route("/howdy_world")
def howdy():
    return "<p>Howdy World</p>"

@app.route("/test")
def test():
    return render_template("test.html")

@app.route("/status")
def status():
    url = 'http://localhost:8080/health_check'
    response = requests.get(url)
    our_response_content = response.content.decode('utf8')
    proper_json_response = json.loads(our_response_content)
    return render_template("status.html", testing=proper_json_response)

@app.route("/weather")
def weather():
    headers = {
        'accept': 'application/json',
        'Content-Type': 'application/json',
    }
```

https://pythonbasics.org/what-is-flask-python/

First thing we'll need to do is start thinking about our "def"'s in terms of FLASK API calls…

BUT…. Let's talk about Swagger first!

Swagger & OpenAPI

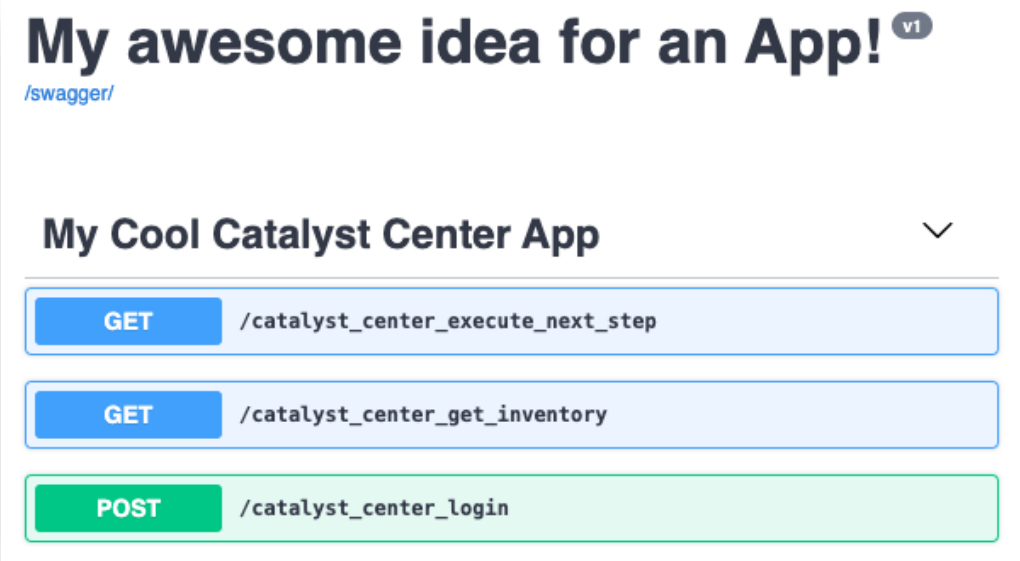
Approaching “API First”

You split out your app’s functions. Let’s make each step available via REST.

- *Stop doing this...*

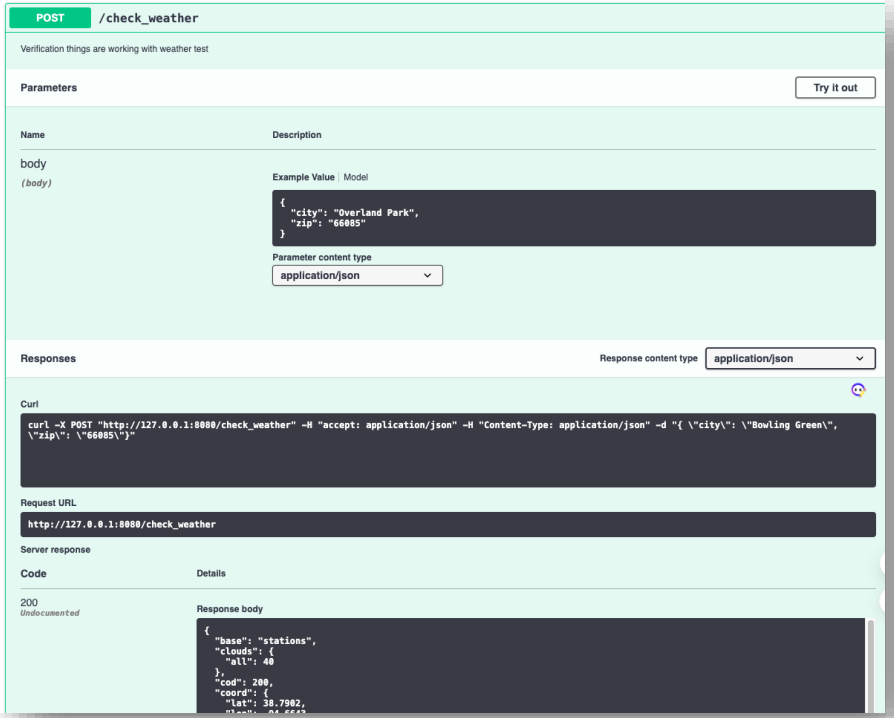
```
Let's log into Catalyst Center...
Please enter your username:
**************
Please enter your passwword:
**************

Calculating BASE64 to get token


SUCCESS!  Token received

Getting Catalyst Center Device Inventory...
It appears we have 239 devices

(execute next step)
(execute next step)
(execute next step)
```

- *Start doing this...*



What is SWAGGER & Open API

Make your “def”’s available as REST API calls



Swagger ”documents” your API’s and makes them available to anyone

(including yourself – important points we’ll consider shortly)

Quick Swagger Tour

Group Your Functions…

My awesome idea for an App! v1

/swagger/

My Cool Catalyst Center App

GET	/catalyst_center_execute_next_step
GET	/catalyst_center_get_inventory
POST	/catalyst_center_login

Health and testing Endpoints

POST	/check_weather
GET	/health_check

Quick Swagger Tour

Try it out – With your own custom payload

POST /check_weather

Verification things are working with weather test

Parameters Try it out

Name	Description
body *(body)*	Example Value | Model `{ "city": "Overland Park", "zip": "66085" }` Parameter content type application/json

Responses Response content type application/json

Quick Swagger Tour

Let’s see if it works…

Execute | Clear

Responses

Response content type: application/json

Curl

```
curl -X POST "http://127.0.0.1:8080/check_weather" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"city\": \"Bowling Green\",
\"zip\": \"66085\"}"
```

Request URL

```
http://127.0.0.1:8080/check_weather
```

Quick Swagger Tour

What did we get back?

Server response

Code	Details
200 *Undocumented*	Response body


```
{
  "base": "stations",
  "clouds": {
    "all": 40
  },
  "cod": 200,
  "coord": {
    "lat": 38.7902,
    "lon": -94.6643
  },
  "dt": 1702147142,
  "id": 0,
  "main": {
    "feels_like": 277.07,
    "humidity": 57,
    "pressure": 1017,
    "temp": 280.48,
    "temp_max": 281.46,
    "temp_min": 279.2
  },
  "name": "Stilwell",
  "sys": {
    "country": "US",
    "id": 2010244,
    "sunrise": 1702128313,
    "sunset": 1702162616,
    "type": 2
  },
```


Download

Tip: Leverage other SWAGGER's for your code

Convert someone else's SWAGGER CURL command to <what you want>

curl command

Examples: GET - POST - JSON - Basic Auth - Files - Form

```
curl -X POST "http://127.0.0.1:8080/check_weather" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"city\": \"Bowling Green\", \"zip\": \"66085\"}"
```

Language: Python + Requests

```python
import requests

headers = {
    'accept': 'application/json',
    'Content-Type': 'application/json',
}

json_data = {
    'city': 'Bowling Green',
    'zip': '66085',
}

response = requests.post('http://127.0.0.1:8080/check_weather', headers=headers, json=json_data)

# Note: json_data will not be serialized by requests
# exactly as it was in the original request.
#data = '{ "city": "Bowling Green", "zip": "66085"}'
#response = requests.post('http://127.0.0.1:8080/check_weather', headers=headers, data=data)
```

Copy to clipboard

Putting your functions in SWAGGER…

3 Files…

- If you start with my boilerplate, there's 3 files that need attention…
  1. ./app/API/BrynCode/views.py
  2. ./app/API/__init__.py
  3. ./app/app.py

- I've included a few examples to get you started. I suggest checking out "Weather" as a generic (practical) function.

Swagger Page Title in app/API/__init__.py

```
app = Flask(__name__)  # Flask app instance initiated
app.config['SECRET_KEY'] = 'C1sc012345'
api = Api(app)  # Flask restful wraps Flask app around it.
app.config.update({
    'APISPEC_SPEC': APISpec(
        title='Super Meraki Lab Control',
        version='v1',
        plugins=[MarshmallowPlugin()],
        openapi_version='2.0.0'
    ),
    'APISPEC_SWAGGER_URL': '/swagger/',  # URI to access API Doc JSON
    'APISPEC_SWAGGER_UI_URL': '/swagger-ui/'  # URI to access UI of API Doc
})
docs = FlaskApiSpec(app)
```

./app/API/BrynCode/views.py

Define your variables in the “Schema” and your function in “Controller”

```
class WeatherControllerSchema(Schema):
    ########  My goto example to verify things are working.  Go grab the weather based on a zip US code.
    ########  For an example, we want swagger to show 2 inputs required for this function.  Could have used Integer, but went with strings for simplicity of example.
    zip = fields.String(required=True, description="zip code",example='66085')
    city = fields.String(required=False, description="city name",example='Overland Park')

class WeatherController(MethodResource, Resource):
    ########  Now just write your code using the 2 inputs.
    import json
    import requests
    ########  Tags is how you group things in swagger.  decription is the label for the specific function.
    @doc(description='Verification things are working with weather test', tags=['Health and testing Endpoints'])
    @use_kwargs(WeatherControllerSchema, location=('json'))
    def post(self, **kwargs):
        #url = """http://192.241.187.136/data/2.5/weather?zip=10001,us&appid=11a1aac6bc7d01ea13f0d2a8e78c227e"""
        ######## insert some default values for ease of testing or demoing.
        url = """http://192.241.187.136/data/2.5/weather?zip=""" + str(kwargs.get("zip", "10001")) + """,us&appid=11a1aac6bc7d01ea13f0d2a8e78c227e"""
        my_response = requests.get(url)
        our_response_content = my_response.content.decode('utf8')
        proper_json_response = json.loads(our_response_content)

        _message = kwargs.get("zip", "10001")
        _message2 = kwargs.get("city", "Overland Park")
        #response = {"message":"Weather JSON response for zip code:" + str(_message) + "\n\n" + str(proper_json_response) + "\n\n" + url + _message2}
        response = proper_json_response
        return response
```

./app/API/BrynCode/views.py (1 of 2)

Variables in the Schema

```
class WeatherControllerSchema(Schema):
    ########  My goto example to verify things are working.  Go grab the weather based on a zip US code.
    ########  For an example, we want swagger to show 2 inputs required for this function.
    zip = fields.String(required=True, description="zip code",example='66085')
    city = fields.String(required=False, description="city name",example='Overland Park')
```

Health and testing Endpoints

POST /check_weather

Verification things are working with weather test

Parameters — Try it out

Name	Description
body *(body)*	**Example Value** | Model

```
{
  "city": "Overland Park",
  "zip": "66085"
}
```

./app/API/BrynCode/views.py (2 of 2)

Leverage the passed variables in your functions…

```
class WeatherController(MethodResource, Resource):
    ########  Now just write your code using the 2 inputs.
    import json
    import requests
    ########  Tags is how you group things in swagger.  decription is the label for the specific function.
    @doc(description='Verification things are working with weather test', tags=['Health and testing Endpoints'])
    @use_kwargs(WeatherControllerSchema, location=('json'))
    def post(self, **kwargs):
        #url = """http://192.241.187.136/data/2.5/weather?zip=10001,us&appid=11a1aac6bc7d01ea13f0d2a8e78c227e"""
        ######## insert some default values for ease of testing or demoing.
        url = """http://192.241.187.136/data/2.5/weather?zip=""" + str(kwargs.get("zip", "10001")) + """,us&appid
        my_response = requests.get(url)
        our_response_content = my_response.content.decode('utf8')
        proper_json_response = json.loads(our_response_content)
```

Your Function

Health and testing Endpoints

POST /check_weather

Verification things are working with weather test

Parameters

Try it out

./app/API/__init__.py (1 of 1)

Import your new Controller

```
try:
    from flask import Flask, render_template
    from flask_restful import Resource, Api
    from apispec import APISpec
    from marshmallow import Schema, fields
    from apispec.ext.marshmallow import MarshmallowPlugin
    from flask_apispec.extension import FlaskApiSpec
    from flask_apispec.views import MethodResource
    from flask_apispec import marshal_with, doc, use_kwargs

    from API.ClusterHealth.views import HeathController
    from API.BrynCode.views import WeatherController

    import requests
    import json
    import subprocess

except Exception as e:
    print("__init Modules are Missing {}".format(e))

app = Flask(__name__)  # Flask app instance initiated
```

Feel free to create your own directory – doesn't have to be "BrynCode"

./app/app.py (1 of 3)

Import your new stuff

```
from flask import Flask, render_template, send_from_directory, request, url_for, flash, redirect
import json
import requests
import time
import subprocess
from subprocess import PIPE
from dotenv import load_dotenv


try:
    from API import (app,
                     api,
                     HeathController,docs,
                     WeatherController
                     )
except Exception as e:
    print("Modules are Missing : {} ".format(e))
```

Don’t forget the comma

./app/app.py (2 of 3)

Create the API mount point.

```
@app.route("/weather")
def weather():
    headers = {
        'accept': 'application/json',
        'Content-Type': 'application/json',
    }

    json_data = {
        'city': 'Overland Park',
        'zip': '66085',
    }

    response = requests.post('http://localhost:8080/check_weather', headers=headers, json=json_data)
    our_response_content = response.content.decode('utf8')
    proper_json_response = json.loads(our_response_content)
    return render_template("weather.html", message="HOWDY", testing=proper_json_response)
```

Important Take Away…
We're calling our own API with our new function.

Anyone else can use this same API because it's now published!

API FIRST!!!

./app/app.py (3 of 3)

Register your new stuff

```
api.add_resource(HeathController, '/health_check')
docs.register(HeathController)

api.add_resource(WeatherController, '/check_weather')
docs.register(WeatherController)
```

And we're off...

Granted, it's not very pretty, but we're moving.

http://127.0.0.1:8080/check_weather


```
{"coord":{"lon":-87.65,"lat":41.85},"weather":[{"id":804,"main":"Clouds","description":"overcast
clouds","icon":"04d"}],"base":"stations","main":
{"temp":274.28,"feels_like":271.74,"temp_min":273.33,"temp_max":274.92,"pressure":1016,"humidity":76
},"visibility":10000,"wind":{"speed":2.24,"deg":269,"gust":3.58},"clouds":
{"all":100},"dt":1702220825,"sys":
{"type":2,"id":2011010,"country":"US","sunrise":1702213631,"sunset":1702246783},"timezone":-21600,"i
d":4887398,"name":"Chicago","cod":200}
```

Import your existing code

Import any existing code

Drop any existing code in boilerplate/app

```
"""Compiled by Bryn Pounds as a 'boilerplate' docker container, swagger 'open API' 2.0 spec,
with a Flask API.

This comes from a project written by Keith Baldwin and Bryn Pounds.
"""
try:
    from flask import Flask
    from flask_restful import Resource, Api
    from apispec import APISpec
    from marshmallow import Schema, fields
    from apispec.ext.marshmallow import MarshmallowPlugin
    from flask_apispec.extension import FlaskApiSpec
    from flask_apispec.views import MethodResource
    from flask_apispec import marshal_with, doc, use_kwargs
    import requests
    import json
    import time
    import urllib3
    import utils
    import os
    import threading
    import sys

    from urllib3.exceptions import InsecureRequestWarning  # for insecure https warnings
    from requests.auth import HTTPBasicAuth  # for Basic Auth
    from SelfServeMeraki01a import *
```

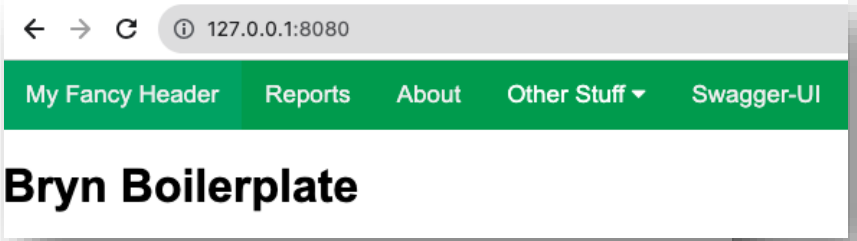
(VERY) Simple UI’s with Flask

Flask Templates

Cascading Style Sheets…

- Beyond the scope for the limited time we have today. I've included a basic example of a navbar template in the boilerplate.

```html
<div class="topnav" id="myTopnav">
  <a href="/" class="active">My Fancy Header</a>
  <a href="/report">Reports</a>
  <a href="/about">About</a>
  <div class="dropdown">
    <button class="dropbtn">Other Stuff
      <i class="fa fa-caret-down"></i>
    </button>
    <div class="dropdown-content">
      <a href="/status">Status</a>
      <a href="/configure_system">Configure System</a>
      <a href="/weather">Kansas City Weather Report</a>
    </div>
  </div>
  <a href="/swagger-ui/" target="_blank">Swagger-UI</a>
  <a href="javascript:void(0);" style="font-size:15px;" class="icon" onclick="myFunction()">&#9776;</a>
</div>
```



Use the header template in our weather HTML UI

Create ./app/API/templates/weather.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Weather Checker</title>
  </head>
  <body>
    {% extends "template.html" %}
    {% block content %}

    <h1> Weather Test Page </h1>
    <p> {% print(message) %} </p>
    <p> Currently, in Kansas City, it is {% print(testing) %} </p>

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

Leverage the CSS

Remember when we mounted our weather API?

We're going to send the HTML page our variables

```
@app.route("/weather")
def weather():
    headers = {
        'accept': 'application/json',
        'Content-Type': 'application/json',
    }

    json_data = {
        'city': 'Overland Park',
        'zip': '66085',
    }

    response = requests.post('http://localhost:8080/check_weather', headers=headers, json=json_data)
    our_response_content = response.content.decode('utf8')
    proper_json_response = json.loads(our_response_content)
    return render_template("weather.html", message="HOWDY", testing=proper_json_response)
```

Hey Flask - Please load up the weather.html template, and send it "**message**" and "**testing**".

Render our weather API in the UI

Create ./app/API/templates/weather.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Weather Checker</title>
  </head>
  <body>
    {% extends "template.html" %}
    {% block content %}

    <h1> Weather Test Page </h1>
    <p> {% print(message) %} </p>
    <p> Currently, in Kansas City, it is {% print(testing) %} </p>

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

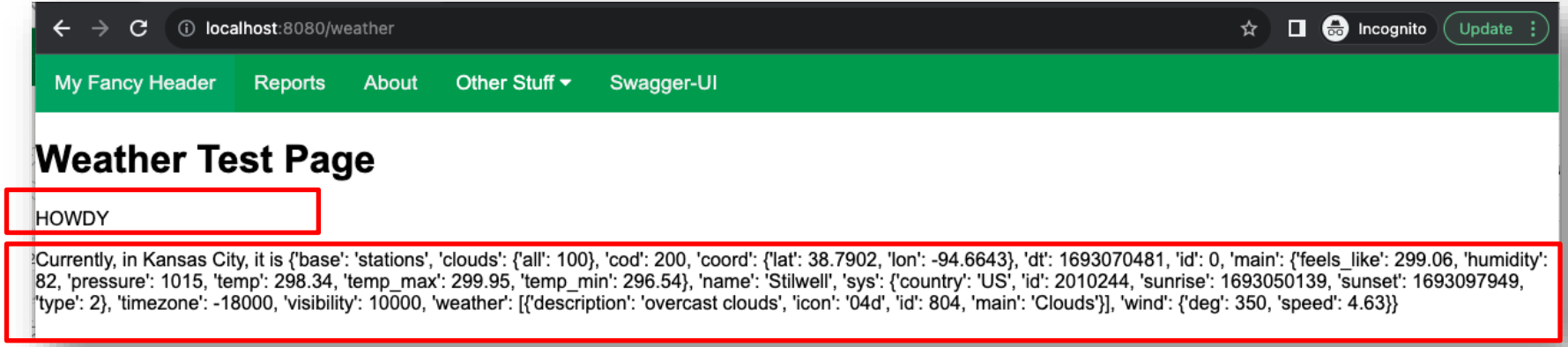
Mixing HTML and code. (Often called "controllers" and "views")

Leverage the CSS

Render the 2 variables we sent to this HTML page

We've successfully rendered our API function

It's working. Let's clean it up a little bit!



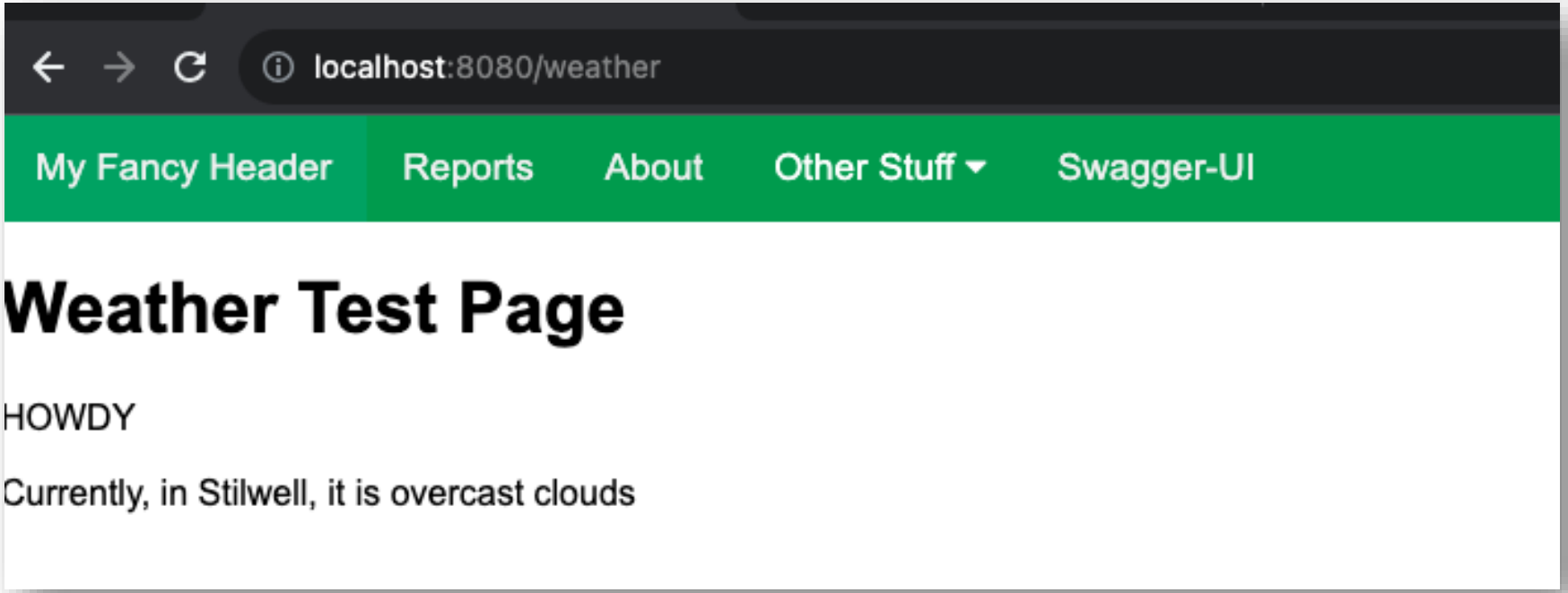
I'm not suggesting you put "Howdy" all over your UI.
Just demonstrating the concepts here

Let’s clean it up…

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Weather Checker</title>
  </head>
  <body>
    {% extends "template.html" %}
    {% block content %}

    <h1> Weather Test Page </h1>
    <p> {% print(message) %} </p>
    <p> Currently, in {% print(testing['name']) %}, it is {% print(testing['weather'][0]['description']) %} </p>

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

We're getting there…

Feel free to add dynamic input fields…. Party On!



Quick note on submit fields

```
@app.route("/report", methods=('GET', 'POST'))
def serve_report():
    if request.method == 'POST':
```

If it's a POST, go do your function you put in swagger.
(Maybe process the values from the input fields)

```
    else:
```

...otherwise, treat it like a GET, and render the template with what you want.
(Maybe render the input fields)

If it is an HTTP GET…

Here: Grab 4 pieces of data, and send it to template “form1a.html”

```
# Display the form for GET requests
# Get the current API Key Value
headers = {'accept': 'application/json','Content-Type': 'application/json',}
json_data = {'file_path': '/app/API/static/meraki_api_key',}
api_key = requests.post('http://127.0.0.1:8080/get_value', headers=headers, json=json_data)

json_data = {'file_path': '/app/API/static/meraki_org_id',}
org_id = requests.post('http://127.0.0.1:8080/get_value', headers=headers, json=json_data)

json_data = {'file_path': '/app/API/static/meraki_base_url',}
meraki_base_url = requests.post('http://127.0.0.1:8080/get_value', headers=headers, json=json_data)

json_data = {'file_path': '/app/API/static/meraki_network_id',}
meraki_network_id = requests.post('http://127.0.0.1:8080/get_value', headers=headers, json=json_data)

return render_template("form1a.html", api_key = len(api_key.json()), org_id=org_id.json(), meraki_bas
```

Render “form1a.html

127.0.0.1:8080/settings

My Fancy Header | Settings | Start Here | About | O

The current API Key is *********40

The current Org ID is 165479

Base URL: https://api.meraki.com/api/v1

Network ID: L_588282701325277227

Would you like to update the API settings?

New API Key:

New Org ID:

New Base URL:

New Network ID:

Submit

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
    {% extends "template.html" %}
    {% block content %}
    <head>
    <title>Flask Static Demo</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <p>The current API Key is *********{{api_key}}</p>
    <p>The current Org ID is {{org_id}}</p>
    <p>Base URL: {{meraki_base_url}}</p>
    <p>Network ID: {{meraki_network_id}}</p><br><br>
    <p><b><big><big>Would you like to update the API settings?
    </b></big></big></p>
    <script src="/static/serve.js" charset="utf-8"></script>
        <form method="post">
          New API Key: <input type="text" name="new_api_key"><br>
          New Org ID: <input type="text" name="new_org_id"><br>
          New Base URL: <input type="text" name="new_base_url"><br>
          New Network ID: <input type="text" name="new_network_id"><br>
          <input type="submit" value="Submit">
        </form>
  </body>
    {% endblock %}
  </body>
</html>
```

But if it's an HTTP POST…

For any populated field, call my "store_value" from our swagger interface

```
@app.route("/settings", methods=('GET', 'POST'))
def form_example():
    if request.method == 'POST':
        # Process the posted form data
        new_api_key = request.form.get('new_api_key')
        new_network_id = request.form.get('new_network_id')
        new_org_id = request.form.get('new_org_id')
        if new_api_key:
            headers = {'accept': 'application/json','Content-Type': 'application/json',}
            json_data = {'api_key': new_api_key,'file_path': '/app/API/static/meraki_api_key',}
            response = requests.post('http://127.0.0.1:8080/store_value', headers=headers, json=json_data)
        if new_network_id:
            headers = {'accept': 'application/json','Content-Type': 'application/json',}
            json_data = {'api_key': new_network_id,'file_path': '/app/API/static/meraki_network_id',}
            response = requests.post('http://127.0.0.1:8080/store_value', headers=headers, json=json_data)
        if new_org_id:
            headers = {'accept': 'application/json','Content-Type': 'application/json',}
            json_data = {'api_key': new_org_id,'file_path': '/app/API/static/meraki_org_id',}
            response = requests.post('http://127.0.0.1:8080/store_value', headers=headers, json=json_data)
        ###  BGP Comment: Note that this section is not complete.  Not updating the base URL at the moment.

        #return f'Hello, {name}!'
        return render_template("form1a.html")
```

Received form data

Review

```
@app.route("/report", methods=('GET', 'POST'))
def serve_report():
    if request.method == 'POST':
```

If it's a POST, go do your function you put in swagger.
(Maybe process the values from the input fields)

```
    else:
```

...otherwise, treat it like a GET, and render the template with what you want.
(Maybe render the input fields)

Let’s Package it up with Docker

Docker and Dockerhub

1 page review…

- Docker is a popular container technology.
  - It's a great way to "package up" everything you need for your awesome app.
  - Include any libraries you've leveraged. Eliminates "wrong version" or "package not installed" type errors
- DockerHub is a public container store
  - Upload your new app, packaged in a container, to dockerhub
  - Any user can run your (complete) app with a single command.

```
docker run -d -p 8080:8080 ciscoautomationstudent1/dnac_compliance_app:latest
```

Packaging up your awesome app (**with Versioning**)

Once you get everything running how you like it…

Here's your awesome app running locally…

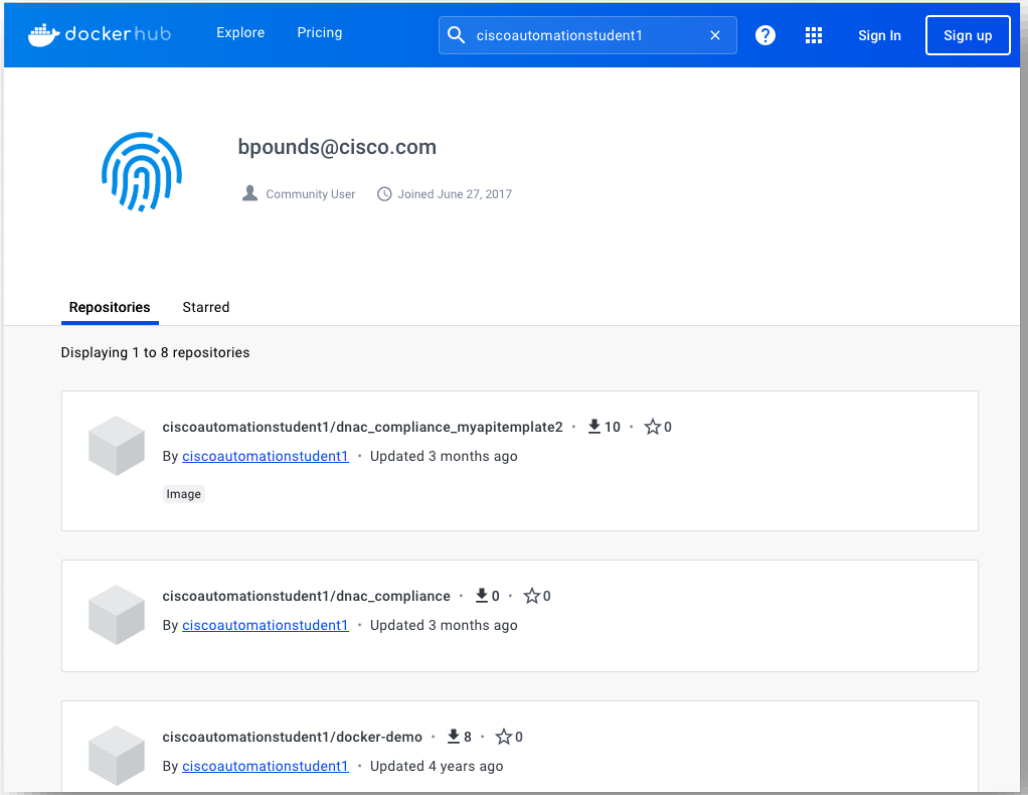
```
bpounds@BPOUNDS-M-X77E app % docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED       STATUS       PORTS                                       NAMES
06dd5c05a1fd   bryn_boilerplate_bryn_boilerplate  "gunicorn --bind 0.0…"   2 hours ago   Up 2 hours   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   bryn_boilerplate
```

```
docker image tag bryn_boilerplate <your dockerhub account name>/<what you want to call your awesome app>
docker push <your dockerhub account name>/<what you want to call your awesome app> :latest
```

Example…

```
docker image tag bryn_boilerplate ciscoautomationstudent1/myapibasedapp
docker push ciscoautomationstudent1/myapibasedapp:latest
```

Now anyone can run your app with a single command




```
docker run -d -p 8080:8080
-v
~/dnac_compliance/config:/
app/API/config
ciscoautomationstudent1/dn
ac_compliance_myapitemplat
e2:latest
```

Run the UI on port 8080.
Map container directory to a local (persistent) directory outside the container

Testing API Scale and Hardening

API Testing – For Scale and Hardening

Tips on testing & verifying your API's…

One of the big advantages of API first and swagger, is you can do focused testing on each individual function of your app.

Example: I'm writing an app to query a populated Catalyst Center with 5000 devices. I only have a test system with 12 devices though.

How can I see how my app runs at scale?

API Testing – For Scale and Hardening

Tips on testing & verifying your API's…

(One) Approach: Use your knowledge of Flask to simulate the DNAC API response. (Skip the authentication – focus on the query only).

Simply have Flask return the exact same response as Catalyst Center.

Scale the simulation (clients or server load) as much as you like per your needs. Load balancers…. You're network people! 8-)

Harden your API, by intentionally inserting errors and timeouts. See how your API's respond when things aren't perfect. Make sure it can handle real world scenarios where things break. (***Recover gracefully***, and give me a useful ***actionable error message***)

Review...

Take your apps to the next level.

Adopt these simple steps, and ”Level Up” your stuff!

- Write API calls. (Functions → API calls)
- Document your API’s with SWAGGER. *(Check Open API spec)*
- Write your apps and UI to leverage your API’s.
- Package your awesome stuff in Docker, post on DockerHub.
- Share with everyone!
  - *1 simple docker command lets anyone run your stuff.*
- Congratulate yourself for being awesome! Sign Autographs!

The bridge to possible

Thank you

cisco Live!

cisco Live!

Let's go