Building a REST API and using Git Actions
(with Python, Flask and Connexion)

---

1 Simple web server

In this guide we're going to use a REST API providing access to a collection of people with CRUD access to an individual person within that collection and extend it with new functionality. Here's the API design for the people collection:

Action	HTTP Verb	URL Path	Description
Create	`POST`	`/api/people`	Defines a unique URL to create a new person
Read	`GET`	`/api/people`	Defines a unique URL to read a collection of people
Read	`GET`	`/api/people/Farrell`	Defines a unique URL to read a particular person in the people collection
Update	`PUT`	`/api/people/Farrell`	Defines a unique URL to update an existing person
Delete	`DELETE`	`/api/people/Farrell`	Defines a unique URL to delete an existing person

Get the codebase from the RealPython github project:
https://github.com/realpython/materials

First use the Version 1 of the application **flask-connexion-rest** (Flask version).

1.1 Requirements

This application makes use of the Flask Micro Framework and Connexion module. So, make sure that you have already installed **Python 3**.

To install **Flask** (https://flask.palletsprojects.com/en/2.0.x/):

```
$ pip install flask
```

To install **Connexion** (https://connexion.readthedocs.io/en/latest/):

```
$ pip install connexion
```

To install **Swagger UI**:

```
$ pip install swagger-ui-bundle
```

1.2 Checking the codebase

The Python code in `server.py` gets a simple web server up and running, and responding with Hello World for a request for the home page.

In the templates directory you can find the `home.html`. This is the file that will be served to a browser when navigating to the URL '`/`'. The file is not named index.html to avoid collision with the Connexion module.

1.3 Running the server

The server can be run with the following command (when the current directory is the one that contains server.py):

```
$ python server.py
```

When you run this application, a web server will start on port 5000. If you open a browser and navigate to `localhost:5000`, you should see `Hello World!` displayed. Right now, this is useful to see that the web server is running.

1.4 Connexion module

Now change to the Version 2 of the application **flask-connexion-rest** (Connexion version).

The Connexion module allows a Python program to use the Swagger specification. This provides a lot of functionality: validation of input and output data to and from your API, an easy way to configure the API URL endpoints and the parameters expected, and a really nice UI interface to work with the created API and explore it.

All of this can happen when you create a configuration file your application can access. The Swagger site even provides an online configuration editor tool to help create and/or syntax check the configuration file you will create.

There are two parts to adding a REST API URL endpoint to your application with Connexion. You'll add Connexion to the server and create a configuration file it will use.

Check how Connexion was added to the server in `server.py`. The `import connexion` statement adds the module to the program. The next step is creating the application instance using Connexion rather than Flask. Internally, the Flask app is still created, but it now has additional functionality added to it.

Part of the app instance creation includes the parameter `specification_dir`. This informs Connexion what directory to look in for its configuration file, in this case our current directory. Right after this, you've added the line:

```
app.add_api('swagger.yml')
```

This tells the app instance to read the file `swagger.yml` from the specification directory and configure the system to provide the Connexion functionality.

1.5 The Swagger configuration file

The file `swagger.yml` is a YAML or JSON file containing all of the information necessary to configure your server to provide input parameter validation, output response data validation, URL endpoint definition, and the Swagger UI.

This file is organized in a hierarchical manner: the indentation levels represent a level of ownership, or scope.

For example, `paths` defines the beginning of where all the API URL endpoints are defined. The `/people` value indented under that defines the start of where all the `/api/people` URL endpoints will be defined. The `get:` indented under that defines the section of definitions associated with an HTTP GET request to the `/api/people` URL endpoint. This goes on for the entire configuration.

Here are the meanings of the fields in `swagger.yml` file:

In the first part of the global configuration information:

1. `swagger:` tells Connexion what version of the Swagger API is being used
2. `info:` begins a new 'scope' of information about the API being built
3. `description:` a user defined description of what the API provides or is about. This is in the Connexion generated UI system
4. `version:` a user defined version value for the API
5. `title:` a user defined title included in the Connexion generated UI system
6. `consumes:` tells Connexion what MIME type is expected by the API.
7. `produces:` tells Connexion what content type is expected by the caller of the API.
8. `basePath: "/api"` defines the root of the API, kind of like a namespace the REST API will come from.

The next section begins the configuration of the API URL endpoint paths:

1. `paths:` defines the section of the configuration containing all of the API REST endpoints.
2. `/people:` defines one path of your URL endpoint.
3. `get:` defines the HTTP method this URL endpoint will respond to. Together with the previous definitions, this creates the `GET /api/people` URL endpoint.

The following section begins the configuration of the single `/api/people` URL endpoint:

1. `operationId: "people.read"` defines the Python import path/function that will respond to an HTTP `GET /api/people` request. The `"people.read"` portion can go as deep as you need to in order to connect a function to the HTTP request. Something like `"<package_name>.<package_name>.<package_name>.<function_name>"` would work just as well. You'll create this shortly.
2. `tags:` defines a grouping for the UI interface. All the CRUD methods you'll define for the people endpoint will share this tag definition.
3. `summary` defines the UI interface display text for this endpoint.
4. `description:` defines what the UI interface will display for implementation notes.

The final section defines the configuration of a successful response from the URL endpoint:

1. `response:` defines the beginning of the expected response section.
2. `200:` defines the section for a successful response, HTTP status code 200.
3. `description:` defines the UI interface display text for a response of 200.
4. `schema:` defines the response as a schema, or structure.
5. `type:` defines the structure of the schema as an array.
6. `items:` starts the definition of the items in the array.
7. `properties:` defines the items in the array as objects having key/value pairs.
8. `fname:` defines the first key of the object.
9. `type:` defines the value associated with `fname` as a string.
10. `lname:` defines the second key of the object.
11. `type:` defines the value associated with `lname` as a string.
12. `timestamp:` defines the third key of the object.
13. `type:` defines the value associated with `timestamp` as a string.

1.6 Handler for people endpoint

In the `swagger.yml` file, Connexion was configured with the `operationId` value to call the `people` module and the `read` function within the module when the API gets an HTTP request for `GET /api/people`. This means a `people.py` module must exist and contain a `read()` function.

Check the contents of `people.py`.

In this code, a helper function called `get_timestamp()` generates a string representation of the current timestamp. This is used to create your in-memory structure and modify the data when you start modifying it with the API.

The `PEOPLE` dictionary data structure is a simple names database, keyed on the last name. This is a module variable, so its state persists between REST API calls. In a real application, the `PEOPLE` data would exist in a database, file, or network resource, something that persists the data beyond running/stopping the web application.

The `read()` function is called when an HTTP request to `GET /api/people` is received by the server. The return value of this function is converted to a JSON string (recall the `produces:` definition in the `swagger.yml` file). The `read()` function you created builds and returns a list of people sorted by last name.

Running your server code and navigating in a browser to `localhost:5000/api/people` will display the list of people on screen.

2 The Swagger UI

We haven't taken advantage of the input or output validation. All that `swagger.yml` gave us was a definition for the code path connected to the URL endpoint. However, what you also get for the extra work is the creation of the Swagger UI for your API.

Navigate to `localhost:5000/api/ui`

You'll see the initial Swagger interface and shows the list of URL endpoints supported at our `localhost:5000/api` endpoint. This is built automatically by Connexion when it parses the `swagger.yml` file.

If you click on the `/people` endpoint in the interface, the interface will expand to show a great deal more information about your API.

It displays the structure of the expected response, the `content-type` of that response, and the description text you entered about the endpoint in the `swagger.yml` file.

You can even try the endpoint out by clicking the `Try It Out!` button at the bottom of the screen.

This can be extremely useful when the API is complete as it gives you and your API users a way to explore and experiment with the API without having to write any code to do so.

Not only is the Swagger UI useful as a way to experiment with the API and read the provided documentation, but it's also dynamic. Any time the configuration file changes, the Swagger UI changes as well.

In addition, the configuration offers a nice, clean way to think about and create the API URL endpoints.

By separating the code from the API URL endpoint configuration, we decouple one from the other.

3 Full API specification

Now change to the Version 3 of the application **flask-connexion-rest** (Connexion version).

This version has the full API specification and respective implementation. Study it and try it!

4 Git actions Hello World

Consider a repository at GitHub with an Hello World for GitHub action at:
https://github.com/davelms/python-github-actions-demo

Click on the green button “Use this template” to create a new repository with it on your GitHub account.

Clone the project to your computer.

Edit the file `.github/workflows/main.yml`
Remove the second job named “deploy-to-test”. We will not be using AWS to test run this project.

Commit changes and push updated files back to your repository.

Go to your repository web page at GitHub and go to the tab “Actions”. The workflow for the build process should have been triggered. Check its results.

5 Create a build action for the REST service

Now create a repository for the code of Version 3 of the REST service (section 3) and create a build workflow with a simple test.