Designing and Building an API in Flask

In this tutorial, you'll discover how to get started with building a Flask API using the basic Flask and Flask-RESTful extensions. You'll also find out how to add authentication to the APIs, how to version APIs, and how to add API documentation.

Designing and Building an API in Flask

Jay Raj

Chapter 1

In this tutorial, you'll learn how to design and build an API using Flask. The tutorial assumes you have a basic knowledge of the Python programming language.

Flask is a web application framework. You can get started with creating your first Flask web app by installing Flask and Python.

Introducing APIs

An API (application programming interface) provides an interface through which you can add, modify, or delete data from an application. For example, the Gmail API provides an interface for reading and sending emails, as well as changing Gmail settings, from an external application.

How to Communicate with an API

An API helps in exchanging information between applications. An HTTP URL represents an API. You can communicate with an HTTP URL using HTTP methods, or verbs (GET, POST, DELETE, PUT, PATCH). We'll get into the details of each HTTP verb at a later point.

Data is exchanged from an API using JSON (JavaScript Object Notation). Earlier XML used to be the preferred format for data exchange, but JSON has become a popular alternative.

Prerequisites

Firstly, do both of the following:

- download and install Python 3
- download and install virtualenv — which helps in creating isolated Python environments

Once you have the above dependencies installed, you can get started with setting up the Flask API project.

Getting Started

Let's start by creating a virtual environment for our Flask app:

```
python3 -m venv venv
```

Based on the Python version you have installed, you need to create the virtual environment. Once the virtual environment has been created, you need to activate it:

```
source env/bin/activate
```

Create a project folder called `python-flask-api` . This will be the root folder for the APIs:

```
mkdir python-flask-api
```

Install Flask using `pip` , which is the Python package installer. `pip` is installed while installing Python 3.x:

```
pip install flask
```

Now that you have everything set up, let's write our first API.

Writing Your First Flask API

Create a root file called `api.py` inside `python-flask-api` . Inside `api.py` , import the `Flask` and `jsonify` modules. The `jsonify` module is used to create a JSON response:

```
from flask import Flask, jsonify
```

Create an app using the Flask module:

```
app = Flask(__name__)
```

Define an API endpoint and supporting method using the @app decorator:

```
@app.route("/employees", methods = ['GET'])
```

Next, define the API endpoint request handler, which returns the response:

```
def getAllEmployees():
    return jsonify([{"name" : "James"},{"name" : "Jackson"}])
```

The `getAllEmployees` request handler makes use of the `jsonify` module to return a `JSON` response. Here's how the complete `api.py` file looks:

```
# api.py
from flask import Flask, jsonify
app = Flask(__name__)
@app.route("/employees", methods = ['GET'])
def getAllEmployees():
    return jsonify([{"name" : "James"},{"name" : "Jackson"}])
```

Save the above changes and export the `FLASK_APP` environment variable:

```
export FLASK_APP=api.py
```

Flask requires the `FLASK_APP` environment variable to know the root file. You can start the server by using the following command:

```
flask run
```

You'll have the API running at http://localhost:5000/employees . Try making a GET request to the API and you'll get the JSON response:

```
[
  {
    "name": "James"
  },
  {
    "name": "Jackson"
  }
]
```

You just created an API using the basic Flask web framework.

Flask-RESTful is an extension to Flask that's is used for building REST APIs. Next, let's have a look at how to build API using Flask-RESTful.

Writing an API Using Flask-RESTful

Creating an API using Flask-RESTful follows a more RESTful way of creating APIs. Let's start by installing Flask-RESTful using pip:

```
pip install flask-restful
```

Using Flask-RESTful, you create a class for the resource at hand. In our case, the resource being Employee, create a class called `Employee` . Create a file called `rest-api.py` inside `python-flask-api` and add the following code:

```
from flask_restful import Resource, Api
class Employee(Resource):
    def get(self):
        return [{"name":"James"},{"name":"Jackson"}]
```

Create an API reference using the flask_restul `Api` and add the resource Employee to the API endpoints:

```
api = Api(app)
api.add_resource(Employee, '/employees')
```

Here's how the complete `rest-api.py` file looks:

```
from flask import Flask
from flask_restful import Resource, Api

class Employee(Resource):
    def get(self):
        return [{"name":"James"},{"name":"Jackson"}]

app = Flask(__name__)
api = Api(app)
api.add_resource(Employee, '/employees')
```

Save the changes and start the server. Try a GET request to the http://localhost:5000/employees endpoint and you'll have the API response.

In order to add the POST, DELETE and PUT request handlers, you need to add new `post` , `delete` and `put` methods respectively to the Employee class.

Why I Prefer the Flask API to Flask-RESTful API

Let's take a look at an example:

```
class Employee(Resource):
    def get(self): return {}
    def post(self): return {}
    def delete(self): return {}
    def getFullName(self): return {}
```

Flask-RESTful implements the APIs by interpreting the HTTP request methods. In the above example, `getFullName` is not an HTTP method. So the only way to create an API route for `/employee/getFullName` is to create a separate resource:

```
class EmployeeFullName(Resource):
    def get(self): return {}
api.add_resource(EmployeeFullName, '/employee/getFullName')
```

The above scenario makes it necessary to write a lot of classes. If you only stick to REST principles with no custom or fancy APIs, that shouldn't be a problem.

Next, let's have a look at how to authenticate an API.

How to Handle Authentication

Authentication needs to be added to the API to prevent unauthorised access. In the above example, using `api.py` file, we created an API that returns JSON data on making a GET request. Let's add a layer of authentication to route in `api.py` , so that the request with a certain privilege can access the API.

Authentication Using an API Key

Assuming the user registered for an API key to access the data, let's validate the API request based on the API key.

Here's the API that you have at hand:

```
@app.route("/employees", methods = ['GET'])
def getAllEmployees():
    response = make_response(jsonify([{"name" : "James"},{"name" : "Johnson"}]))
    response.headers['Accept-version'] = 'v1'
    return response
```

To authenticate the above route, let's make use of function decorators. A decorator is function that wraps around the original function and replaces it. It keeps the original function information and executes it once the decorator function is executed.

You'll be adding a decorator function called `authorize` to the route `/employee` . Let's define the decorator function:

```
from flask import request
from functools import wraps

def authorize(f):
    @wraps(f)
    def decorated_function():
        key = request.args.get('api_key')
        if key == 'abc123':
            return f()
        return jsonify({"statusCode":401, "message":"Un authorised access"})
    return decorated_function
```

The decorator function checks for the `api_key` from the HTTP request. If the received key is same as the expected key, it returns the original function. The request is aborted if the `api_key` doesn't match.

Add the decorator function to the route:

```
@app.route("/employees", methods = ['GET'])
@authorize
def getAllEmployees():
    response = make_response(jsonify([{"name" : "James"},{"name" : "Johnson"}]))
    response.headers['Accept-version'] = 'v1'
    return response
```

Save the above changes and run the Flask server. Try accessing the API without the `api_key` and you'll receive an error message:

```
{"message":"Un authorised access","statusCode":401}
```

On trying to access the API with the `api_key` http://localhost:5000/employees?api_key=abc123 , a proper response is returned.

In a real application, the `api_key` will be compared against a key from a database. You can set up a check from a database as an exercise.

How to Handle Invalid Requests

Let's take a look at how to handle invalid requests.

While developing an API using Flask, you can use `@app` decorator to specify an error handler for 404 (not found) routes:

```
@app.errorhandler(404)
def route_not_found(e):
    return jsonify({"error" : "Invalid route","statusCode" : 404})
```

So whenever an invalid route is requested, it gets handled by the `route_not_found` method.

API Versioning

Versioning is an important part of API development. If your system is constantly evolving, then changes in the API are inevitable. To manage this change, version your API.

Popular ways of versioning APIs are through custom headers or the URI.

Using Custom Headers

You can add custom headers to the response returned from the API to specify the version. Let's have a look at some Flask code to understand how you can add custom headers.

Flask provides a module called `make_response` , which is used for creating an API response. You need to import it to create custom responses:

```
from flask import make_response
```

Update the existing `/employees` route to add a custom versioning header. Instead of returning the JSON data, make use of `make_response` :

```
response = make_response(jsonify([{"name" : "James"},{"name" : "Johnson"}]))
```

Add a custom header for versioning and return the response:

```
@app.route("/employees", methods = ['GET'])
def getAllEmployees():
    response = make_response(jsonify([{"name" : "James"},{"name" : "Johnson"}]))
    response.headers['Accept-version'] = 'v1'
    return response
```

Check the browser console, under the *Network* tab, to look for the versioning custom header.

Using the URI

Versioning of an API can also be achieved by adding version numbers in the URI. For example, `http://api.example.com/v1/employees` .

When a new version of the API is released, the URI becomes `http://api.example.com/v2/employees` . It prevents breaking changes at the client end using the old API versions. And once the latest version changes become stable, you can deprecate the old API version separately.

API Documentation

Documenting your API so that the client users will understand you API is important. We'll use Flasgger to document the API, which uses Swagger under the hoods.

Swagger is a suite of API developer tools that help in development across the API design phase to testing and deployment. From the official documentation :

> *Swagger is a powerful yet easy-to-use suite of API developer tools for teams and individuals, enabling development across the entire API lifecycle, from design and documentation, to test and deployment.*
>
> *Swagger consists of a mix of open source, free and commercially available tools that allow anyone, from technical engineers to street smart product managers to build amazing APIs that everyone loves.*

The Open API Specification (OAS), or Swagger specification, introduces an interface to APIs that helps in understanding the API service. OAS is a language-agnostic way of describing an API, and the input and output values of each endpoints are specified as YAML. For detailed information, I recommend reading the official documentation on OAS .

Flasgger supports version 2 and 3 of Swagger although version 3 is experimental.

To start using Flasgger, you need to install it using pip:

```
pip install flasgger
```

Once you have it installed, import `Swagger` from `flasgger` :

```
from flasgger import Swagger
```

Initialize swagger using the Flask app:

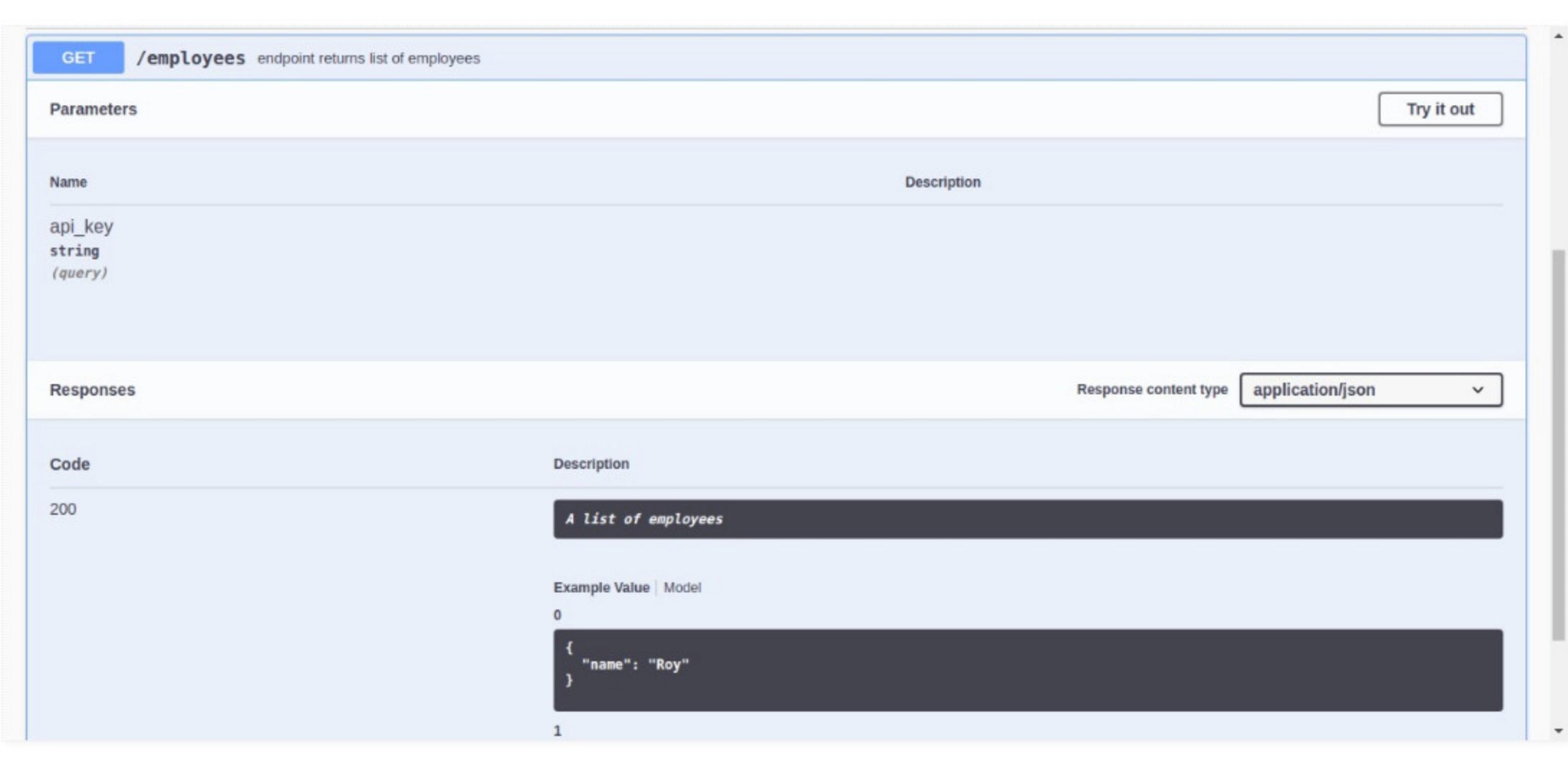
```
swagger = Swagger(app)
```

Flasgger creates the API documentation from the info provided within the API route's docstring. Let's add the specification inside docstring:

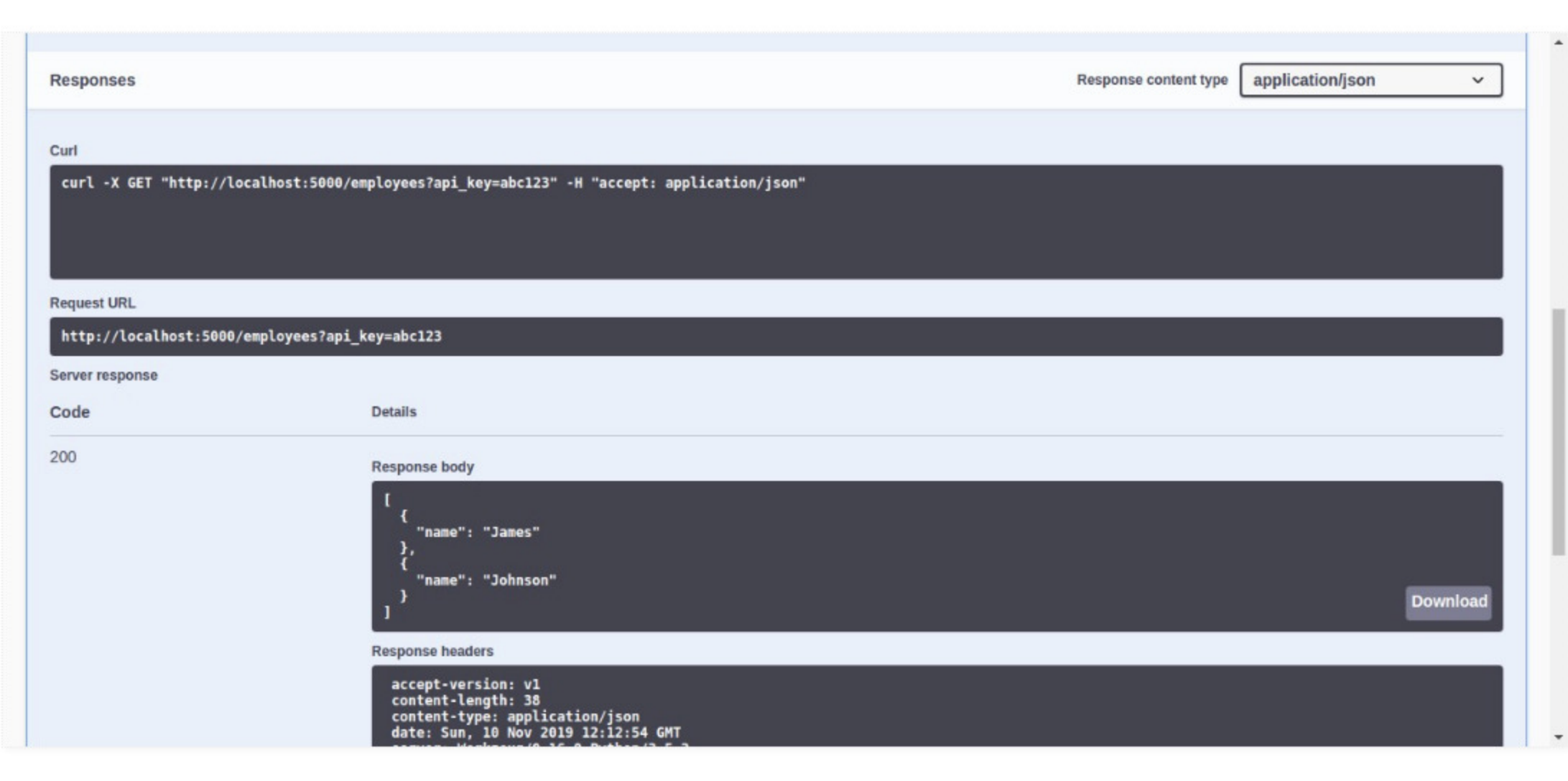
```
def getAllEmployees():
    """
    endpoint returns list of employees
    ---
    parameters:
      - name: api_key
        in: query
    responses:
      200:
        description: A list of employees
        examples:
          [{"name" : "Roy"},{"name" : "Sam"}]
    """
    response = make_response(jsonify([{"name" : "James"},{"name" : "Johnson"}]))
    response.headers['Accept-version'] = 'v1'
    return response
```

`parameters` defines the parameters passed to the API. `in` defines where the parameter is passed, in query or in URL path. `responses` defines the response description and a sample of response.

Save the changes and run the Flask app. Point your browser to http://localhost:5000/apidocs/ and you'll have the API documentation. Click on the API to get detailed information.



You can click on the `Try it out` button to execute the API by entering the `api_key` .



Wrapping Up

In this tutorial, we've looked at how to get started with building a Flask API using the basic Flask and Flask-RESTful extensions. We also looked at how to add authentication to the APIs, how to version APIs, and how to add API documentation.

The source code from this tutorial is available on GitHub .