COM661 Full Stack Strategies and Development

BE03. API Design and Implementation

Aims

- To introduce the concept of RESTful APIs
- To identify HTTP verbs and response codes
- To present a framework for API design
- To implement API endpoints to modify a data set
- To introduce Postman as a tool for testing of API endpoints
- To present sample code to respond to POST, PUT and DELETE requests
- To demonstrate sub-document manipulation in a dataset

Table of Contents

3.1 API Design

In the previous practical we created a first Flask API to return information from a collection of businesses by using a GET request via a web browser. In this session we will consider more formally the construction of a RESTful API and identify principles of good practice in API design, before continuing with our first example to enhance its functionality.

3.1.1 What is a RESTful API?

REST is an abbreviated form of "**RE**presentational **S**tate **T**ransfer" and describes a software model in which a resource-based architecture communicates representational state information between client and server. By resource-based, we mean that the architecture is designed around resources or objects (nouns) rather than operations (verbs). Hence, we implement URLs for businesses, students, cars, or whatever other objects our application is dealing with – rather than "new", "edit", "delete" or whatever other operations are being implemented.

Typical URL patterns in a RESTful architecture (for example in our sample list of businesses application at http://www.bizlist.com) might therefore be (e.g.)

- http://www.bizlist.com/api/v1.0/businesses
  when referring to the collection of businesses
- http://www.bizlist.com/api/v1.0/businesses/101
  when referring to a specific business by its identifier
- http://www.bizlist.com/api/v1.0/businesses/101/reviews
  when referring to the collection of reviews for a specific business
- http://www.bizlist.com/api/v1.0/users
  when referring to the collection of users
- http://www.bizlist.com/api/v1.0/users/101
  when referring to a specific user by identifier

Note:	Remember that the **/api** path is to separate our API functionality from other public-facing content of the business website at http://www.bizlist.com, while the **/v1.0** path is to support the possibility of multiple versions of the API in simultaneous use

The REST architecture describes six constraints as follows.

- **Uniform interface between client and server**
  This is fundamental to RESTful design and has three main elements; (1) the client indicates a request to the server by using one of the HTTP verbs such as GET, POST, PUT or DELETE; (2) The client request is in the form of a URL; (3) The server response consists of an HTTP status code and a body represented in JSON or XML

- **Statelessness**
  This is the concept that the server contains no state information about the client – each client request is self-contained and provides enough information for it to be processed. Any state information that is required (for example cookies, session variables, etc.) are held on the client.

- **Cacheable**
  All server responses are cacheable

- **Client-server**
  The server and client are disconnected with no shared resources. The uniform interface (above) is the link between the two.

- **Layered system**
  Connected to cacheability and client-server organization, the client cannot assume any direct connection with the server as there may be software or hardware intermediaries. For example, a request could be satisfied by a cache rather than by the server, or the request may be re-directed to an alternative server.

- **Code on demand**
  The server can temporarily extend the client by providing executable code as part of the response, which is then run by the client. For example, the response from the server may contain JavaScript to be run on the browser.

3.1.2 HTTP Methods and Responses

We have seen how the URL pattern in a RESTful application identifies the data object with which we want to perform some operation, but how do we denote the actual operation that we want to execute? For example, a call to the URL

http://www.bizlist.com/api/v1.0/businesses/101/reviews

might mean that we want to see the collection of user reviews for the business identified by the **id** value **101**. However, it might equally mean that we want to add a review to the collection of reviews for that business.

The way in which we communicate our intent to the server is by selection of the appropriate HTTP method (verb). You may already be familiar with the **GET** and **POST** methods from previous work with HTML forms, but there is a wider selection of methods available, each with their own specific purpose, corresponding to the standard CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) operations on databases and described in Table 3.1 below.

HTTP Method	Action requested
GET	Return a resource
POST	Create a new resource
PUT	Update a resource
DELETE	Delete a resource

Table 3.1 HTTP Methods

Therefore, a **GET** request to the previous URL http://www.bizlist.com/api/v1.0/businesses/101/reviews will indicate that we want to fetch the list of reviews for the business with **id** 101, while a **POST** request to the same URL will indicate that we want to add a new review to the collection for that business.

In addition, the RESTful model suggests recommended HTTP Status Codes to be used for each combination of URL and method. Again, you will have already used some of these (e.g. 200 OK, 404 Page not found, etc.), but the codes that we will use are are presented in full in Table 3.2, below.

Method	Collection (e.g. /api/v1.0/businesses)	Specific Item (e.g. /api/v1.0/businesses/101)
GET	**200** (OK); List of businesses. Use pagination, sorting and filtering to navigate large collections	**200** (OK) or **404** (Not found) if **id** not found or invalid
POST	**201** (Created); Location header with a link to the newly created business	**404** (Not found) or **409** (Conflict) if the resource already exists
PUT	**404** (Not found) – unless you provide functionality to modify every element in the collection	**200** (OK) or **404** (Not found) if **id** not found or invalid
DELETE	**404** (Not found) – unless you intend functionality to delete the entire collection	**200** (OK) or **204** (No content) if no data returned or **404** (Not found) if **id** not found or invalid.

Table 3.2 HTTP Methods and Status Codes

3.1.3 API Design

The previous discussions on RESTful services and HTTP methods and responses lead us towards a design pattern for the endpoints (URLs) in our API. We can illustrate this by compiling a list of the functionality we intend to provide in our demonstration application, specifying the HTTP method and URL for each. This list is presented in Table 3.3 below – note how a single URL can be used to indicate multiple actions, with the HTTP Method used to distinguish between them.

Method	URL	Action
GET	/api/v1.0/businesses	Get all (or multiple) businesses
POST	/api/v1.0/businesses	Create a new business
GET	/api/v1.0/businesses/123	Get a specific business (with ID=123)
PUT	/api/ v1.0/businesses/123	Update a specific business
DELETE	/api/ v1.0/businesses/123	Delete a specific business
GET	/api/ v1.0/businesses/123/reviews	Get all (or multiple) reviews for a specific business
POST	/api/ v1.0/businesses/123/reviews	Add a review for a specific business
GET	/api/ v1.0/businesses/123/reviews/321	Get a specific review (with ID=321) for a specific business
PUT	/api/ v1.0/businesses/123/reviews/321	Update a specific review for a specific business
DELETE	/api/ v1.0/businesses/123/reviews/321	Delete a specific review

Table 3.3 API Design for the Sample Application

Note:	If the GET requests on collections of businesses or reviews are intended to fetch a subset of the data (i.e. multiple businesses/reviews but not all), then the request will also need additionality information to specify which subset to return. We will examine this case later in this practical.

3.2 API Endpoints to Update the Dataset

In the previous practical we built a pair of API endpoints (routes) to retrieve the entire set of businesses and to retrieve a single business. However, from the discussion above, we now know that we can also implement endpoints to add an item to the data set, modify an item and delete individual items. We will add these endpoints in this section.

3.2.1 Adding a New Element

To add a new business, we receive values POSTed from an HTML **`<form>`** element and create a new object that can be appended to the list as follows.

- First, we determine the **`id`** for the new business by finding the **`id`** of the business currently at the end of the list and incrementing that value.

- Next, we create the new dictionary object by providing values for each of the fields. Note the use of the **`flask request`** object to retrieve the POSTed values as (e.g.) **`request.form["name"]`** which in turn means that we need to add the **`request`** object to the list of **`flask`** elements **`import`**ed in the first line of the application. The **`id`** is set to the newly calculated value, while **`name`**, **`town`** and **`rating`** are retrieved from the form. Finally, we initialize the **`reviews`** to an empty list.

- Once the object has been created, we simply append it to the list of businesses

- Finally, we return the **201** status code for a successful create operation, along with a copy of the newly created element. Note that it is common practice to also return the URL from which the new object can be retrieved, but we will simply return the object for now.

File: app.py

```
from flask import Flask, jsonify, make_response, request

...

@app.route("/api/v1.0/businesses", methods=["POST"])
def add_business():
    next_id = businesses[-1]["id"] + 1
    new_business = { "id": next_id,
                     "name" : request.form["name"],
                     "town" : request.form["town"],
                     "rating" : request.form["rating"],
                     "reviews" : []
                    }
    businesses.append(new_business)
    return make_response( jsonify( new_business ), 201 )

if __name__ == "__main__":
   app.run(debug=True)
```

Do it now!	Add the new route and function to your **first** application that you created in Practical BE02. Remember to also add the **`request`** object to the **`import`** list on the first line.

3.2.2 Introducing Postman

It is easy to test **GET** requests in the browser, but other HTTP methods are not so convenient. For example, to test a **POST** request, we would have to create an HTML **`<form>`** to send the data to the server in order to check if it is being received and interpreted correctly.

Fortunately, there is a free app called **Postman** that will allow us to easily check all HTTP request types. Postman can be downloaded from https://www.postman.com/downloads/ and is available as a standalone app for Windows, MacOS, Linux, or as an extension for the Chrome browser. Postman is also available for you on the lab machines.

When launched, Postman appears as illustrated in Figure 3.1 below.

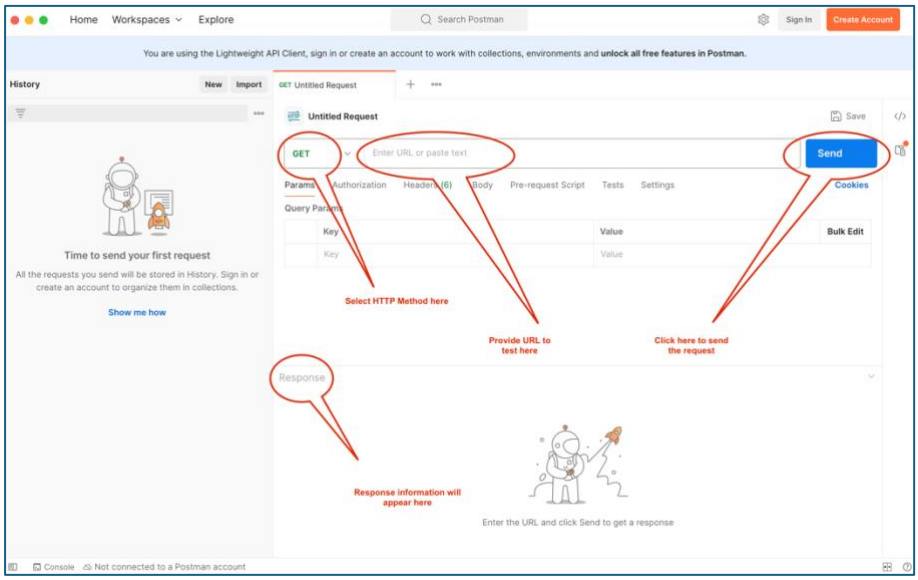

Figure 3.1 Postman

We can see the operation of Postman by sending a simple GET request to http://localhost:5000/api/v1.0/businesses (make sure that your Flask application is running). Choose GET from the list of HTTP methods and provide the URL in the "*Enter request URL*" box and click "*Send*". You should see output as illustrated in Figure B2.2 that verifies that the request has been fulfilled and that Postman is operating correctly.

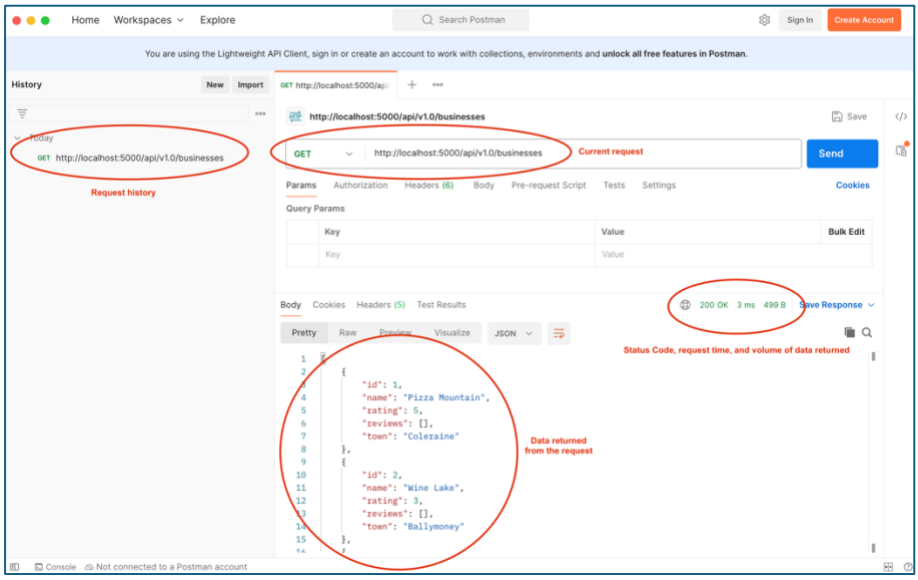

Figure 3.2 Using Postman

The real value in a tool such as Postman is in how it helps us easily check HTTP methods other than GET. For example, we can validate our **/api/v1.0/businesses** POST route by selecting **POST** from the list of methods, choosing **x-www-form-urlencoded** from the options in the ***body*** menu and providing name and value entries for each of the fields that we want to submit as illustrated in Figure 3.3 below.

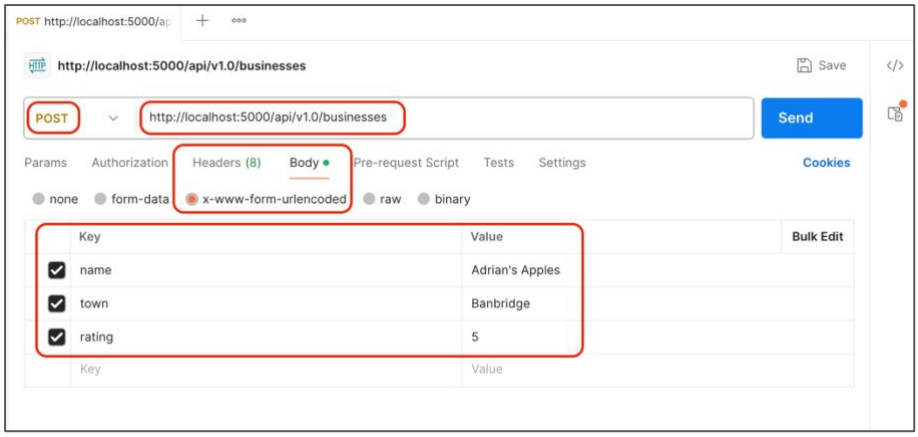

Figure 3.3 POSTing Values to the API

Clicking "*Send*" generates output, as shown in Figure 3.4, that verifies the operation of our POST route for **/api/v1.0/businesses**. We can see that the status code **201** has been provided and that the newly specified object has been returned.

Figure 3.4 Response from POST Request

Finally, we re-issue the **GET** request to return all businesses to verify that the new element has been added to the collection.

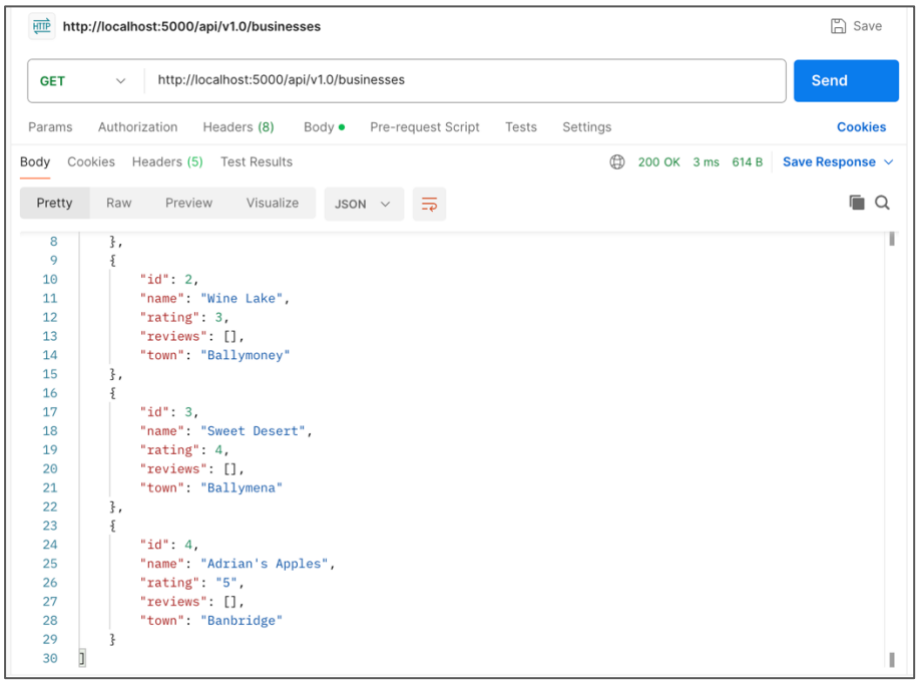

Figure 3.5 New Business Added

Do it now!	Follow the sequence of operations illustrated above to add a new business of your choice to the collection.

Try it now!	Obtain the `id` value of a newly created business (copy it from the data returned by the **GET all businesses** request) and use Postman to send the **GET** command that retrieves just that business.

3.2.3 Editing an Element

To edit an element, we need to pass its **`id`** value in the URL and then use that value to locate the business to be edited. Later, we will see how to do this in conjunction with a back-end database, but for now we simply use a **`for`** loop to iterate across the collection testing the **`id`** value of each until a match is found.

Once the business to be amended is found, we simply assign the values passed from the client to the corresponding fields of that business. Note that we assume that all field values are provided – this is the typical case, but we could add additional code to check for the presence of a form value and only update those fields where a new value is provided. The code for the new route and function are provided below.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<int:id>", methods=["PUT"])
def edit_business(id):
    for business in businesses:
        if business["id"] == id:
            business["name"] = request.form["name"]
            business["town"] = request.form["town"]
            business["rating"] = request.form["rating"]
            break
    return make_response( jsonify( business ), 200 )

if __name__ == "__main__":
   app.run(debug=True)
```

Do it now!	Add the new route and function and test it in Postman by making a **PUT** request to **/api/v1.0/businesses/1** to update the first business in the collection.

Note:	Remember that each time the application is re-loaded, the collection of businesses is reset. Hence, the new business that you added in the previous operation will not be available for editing.

3.2.4 Deleting an Element

Removing an element is a similar process to editing. Again, we accept the **`id`** value passed as the URL variable parameter and loop through the business objects until a match is found. Once the business to be deleted is located, we delete it by using the **`remove()`** method that can be applied to lists. Once done, we return status code **200** with an empty response body.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<int:id>", methods=["DELETE"])
def delete_business(id):
    for business in businesses:
        if business["id"] == id:
            businesses.remove(business)
            break
    return make_response( jsonify( {} ), 200)

if __name__ == "__main__":
   app.run(debug=True)
```

Do it now!	Add the new route and function and test it in Postman by making a **DELETE** request to **/api/v1.0/businesses/2** to remove the second business in the collection.

3.3 Working with Sub-document Collections

Sub-document collections arise when one of the fields of our top-level object (i.e. a business) is itself a collection of documents. In our example we specified reviews as one of the fields of a business and we can further develop this now.

To this point, we have set all reviews fields to empty lists, but we will now create a definition of a review object and produce the endpoints to retrieve the collection of reviews from a specified business and to add a review to a business's collection.

First, we provide the function to serve a GET request to **/api/v1.0/businesses/<id>/reviews** which returns the collection of reviews for the business with the specified **`id`**. This is a very straightforward operation which involves finding the specified business and returning its **`reviews`** element with a status code of **200**.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<int:id>/reviews",
            methods=["GET"])
def fetch_all_reviews(id):
    for business in businesses:
        if business["id"] == id:
            break
    return make_response( jsonify( business["reviews"] ), 200 )

if __name__ == "__main__":
   app.run(debug=True)
```

Adding a new review is slightly more complex. First, we loop through the **businesses** collection as usual, but when we locate the one to which we want to add a review, we need to know whether any reviews already exist in that collection. If the **reviews** element is currently empty, we set the **id** value for the new review to 1, otherwise we retrieve the **id** of the last review in the collection and increment that value to set the new review **id**.

Now, we can create the new review object as a Python dictionary, retrieving the values for **username**, **comment** and **stars** from the **request.form** object, and **append** it to the collection for that business. Once complete, we return the new review with a status code of 201.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<int:b_id>/reviews",
           methods=["POST"])
def add_new_review(b_id):
    for business in businesses:
        if business["id"] == b_id:
            if len(business["reviews"]) == 0:
                new_review_id = 1
            else:
                new_review_id =
                    business["reviews"][-1]["id"] + 1
            new_review = {
                "id": new_review_id,
                "username" : request.form["username"],
                "comment" : request.form["comment"],
                "stars" : request.form["stars"]
            }
            business["reviews"].append(new_review)
            break
    return make_response( jsonify( new_review ), 201 )

if __name__ == "__main__":
    app.run(debug=True)
```

Do it now!	Add the new routes and functions and test in Postman by making **GET** and **POST** requests to **/api/v1.0/businesses/1/reviews** to manipulate the set of reviews for the first business. N.B. Remember to be very careful with Python indentation so that the **`return`** statement in **`add_new_review()`** is outside the scope of the **`for`** loop.

Now that we have implemented the ability to add a collection of reviews to a business, we can begin to provide routes to manipulate individual reviews. First, we will add functionality to retrieve a specific review for a specific business.

This time, the URL requires two variable parameters – one for the business **`id`** and another for the review **`id`**. Both of these are then made available to the code through the parameter list of the function. In the function, we first loop through the collection of

businesses and once a match is found, we loop through the collection of reviews for that business. When the matching review is located, we **`break`** out of both **`for`** loops and return the review object with a status code of **200**.

File: app.py

```
...

@app.route( \
  "/api/v1.0/businesses/<int:b_id>/reviews/<int:r_id>",
  methods=["GET"])
def fetch_one_review(b_id, r_id):
    for business in businesses:
        if business["id"] == b_id:
            for review in business["reviews"]:
                if review["id"] == r_id:
                    break
            break
    return make_response( jsonify( review ), 200)

if __name__ == "__main__":
    app.run(debug=True)
```

Do it now!	Add the new route and function and test in Postman by making **POST** requests to **/api/v1.0/businesses/1/reviews** to add a set of reviews for the first business and then a series of **GET** requests to **/api/v1.0/businesses/1/reviews/<id>** with a range of **<id>** values matching the **`id`** properties of the reviews.

Finally, in this section, we present the endpoints to edit or delete a specified review. These are quite similar in structure and can be presented together.

In both cases, we loop through the businesses looking for the entry that matches the business **`id`** value provided. Then, we loop through the collection of reviews for that business, looking for the matching review **`id`**. Once the review **`id`** is found, the **`edit_review()`** function updates the fields with the values provided by the user, while the **`delete_review()`** function uses the **`remove()`** method to strike the selected entry from the list. When the action is complete, we break out of both **`for`** loops and generate the appropriate response and status code.

File: app.py

```
...

@app.route(
  "/api/v1.0/businesses/<int:b_id>/reviews/<int:r_id>",
  methods=["PUT"])
def edit_review(b_id, r_id):
    for business in businesses:
        if business["id"] == b_id:
            for review in business["reviews"]:
               if review["id"] == r_id:
                  review["username"] = request.form["username"]
                  review["comment"] = request.form["comment"]
                  review["stars"] = request.form["stars"]
                  break
            break
    return make_response( jsonify( review ), 200)

@app.route(
        "/api/v1.0/businesses/<int:b_id>/reviews/<int:r_id>",
        methods=["DELETE"])
def delete_review(b_id, r_id):
    for business in businesses:
        if business["id"] == b_id:
            for review in business["reviews"]:
                if review["id"] == r_id:
                    business["reviews"].remove(review)
                    break
            break
    return make_response( jsonify( {} ), 200)

if __name__ == "__main__":
   app.run(debug=True)
```

Do it now!	Add the new routes and functions and test in Postman by making **POST** requests to **/api/v1.0/businesses/1/reviews** to add a set of reviews for the first business and then a series of **PUT** and **DELETE** requests to **/api/v1.0/businesses/1/reviews/<id>** with a range of **<id>** values matching the **`id`** properties of the reviews.

Note:	Ultimately, we will work with information stored in a back-end database rather than a hard-coded data structure. This will result in much of the internal code in the functions changing, but the general structure of the API will remain unchanged.

3.4 Further Information

- https://en.wikipedia.org/wiki/Representational_state_transfer
  Wikipedia – Representational State Transfer

- https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9
  RESTful API Design Guidelines – the best practice

- https://restfulapi.net/rest-api-design-tutorial-with-example/
  How to design a REST API

- https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask
  Designing a RESTful API with Python and Flask

- https://kite.com/blog/python/flask-restful-api-tutorial/
  Tutorial – Building a RESTful API with Flask

- https://www.postman.com/
  Postman home page

- https://www.youtube.com/watch?v=juldrxDrSH0
  YouTube – Postman Beginner Tutorial

- https://www.youtube.com/watch?v=CjYKrbq8BCw
  YouTube – Creating a RESTful API with Flask (series of 4 videos)

- https://www.youtube.com/watch?v=s_ht4AKnWZg
  YouTube – Build a REST API with Python and Flask