DISTRIBUTED AND CLOUD COMPUTING

LAB 7: RESTFUL API & OPENAPI SPECIFICATION

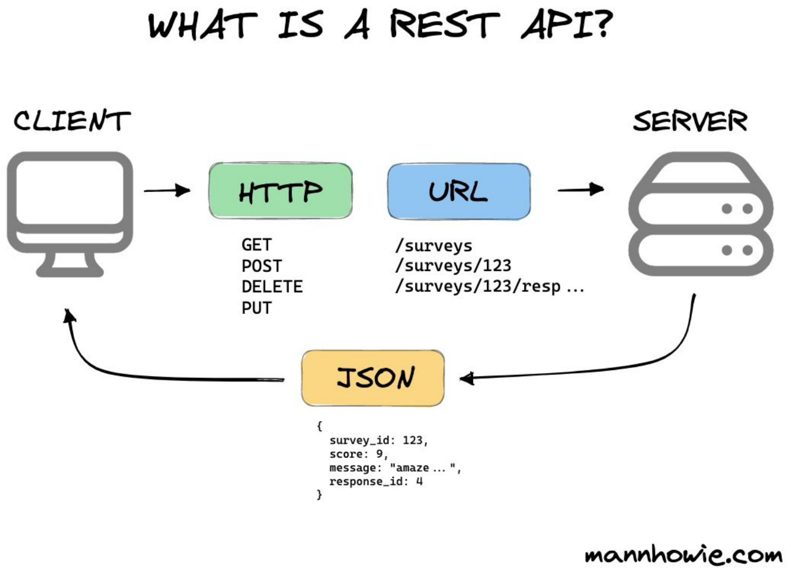
(Module: RPC & RESTFUL API)

REST As an Architecture Style

- **RE**presentational **S**tate **T**ransfer: architecture style for distributed hypermedia systems
- REST Components
  a. Resource
     - Identifiers (URI including URL)
     - Metadata (e.g., source links, etc.)

  b. Representation
     - Metadata (e.g., media type, etc.)

  c. Control data (e.g., Cache-Control, etc.)
- 6 Guiding Principles/Constraints of REST
  a. Client-Server
  b. Stateless
  c. Cacheable
  d. Uniform Interface
  e. Layered System
  f. Code on Demand (Optional)
- **RESTful API (or REST API)**: Web API conforming to the REST architecture style

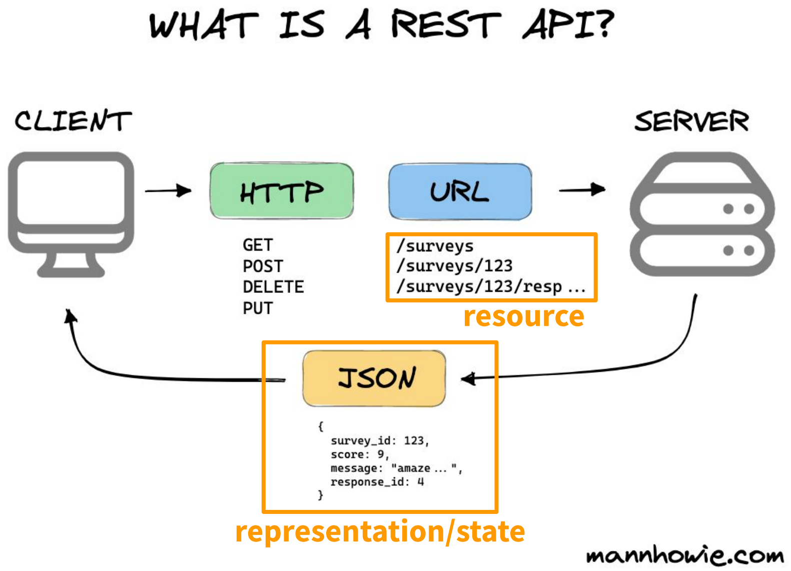


```
{
  survey_id: 123,
  score: 9,
  message: "amaze ... ",
  response_id: 4
}
```

https://mannhowie.com/rest-api

REST Components

- Resource: **abstraction** of information
  - Identifiers
    - URL endpoint…
  - Metadata
    - Source links
    - Alternate data formats
    - Vary: language preferences…
    - …
- Representation: **state** of the resource
  - Metadata
    - Media type
    - Last-modified time
    - …
- Control data: interaction behavior
  - Cache-Control
  - …

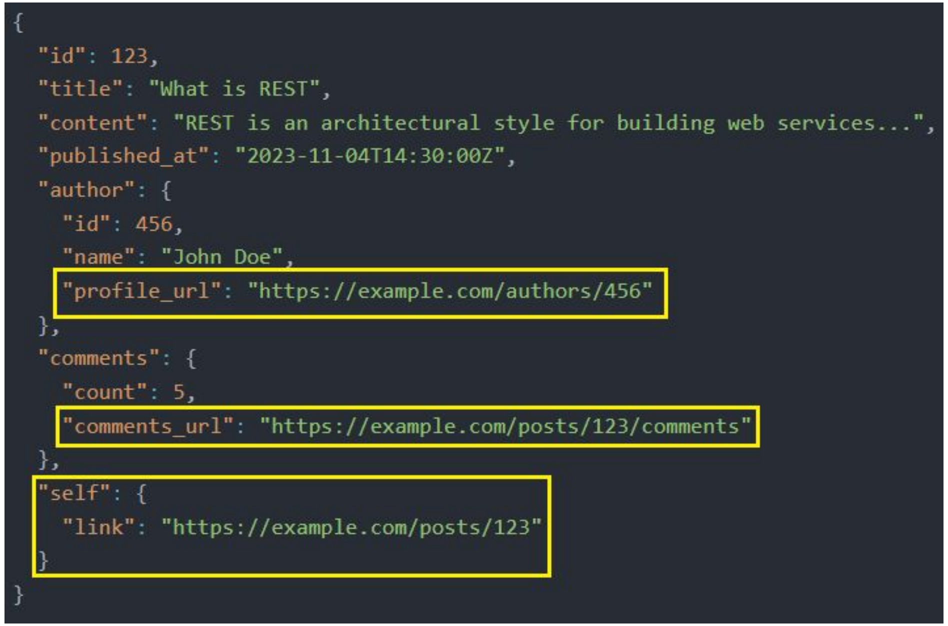


```
{
  survey_id: 123,
  score: 9,
  message: "amaze … ",
  response_id: 4
}
```

https://mannhowie.com/rest-api

REST Components

- Resource: **abstraction** of information
  - Identifiers
    - URL endpoint…
  - Metadata
    - Source links
    - Alternate data formats
    - Vary: language preferences…
    - …
- Representation: **state** of the resource
  - Metadata
    - Media type
    - Last-modified time
    - …
- Control data: interaction behavior
  - Cache-Control
  - …



```
{
  "id": 123,
  "title": "What is REST",
  "content": "REST is an architectural style for building web services...",
  "published_at": "2023-11-04T14:30:00Z",
  "author": {
    "id": 456,
    "name": "John Doe",
    "profile_url": "https://example.com/authors/456"
  },
  "comments": {
    "count": 5,
    "comments_url": "https://example.com/posts/123/comments"
  },
  "self": {
    "link": "https://example.com/posts/123"
  }
}
```

Pressing the links (i.e., GET …) transform from one resource state to another - REST.

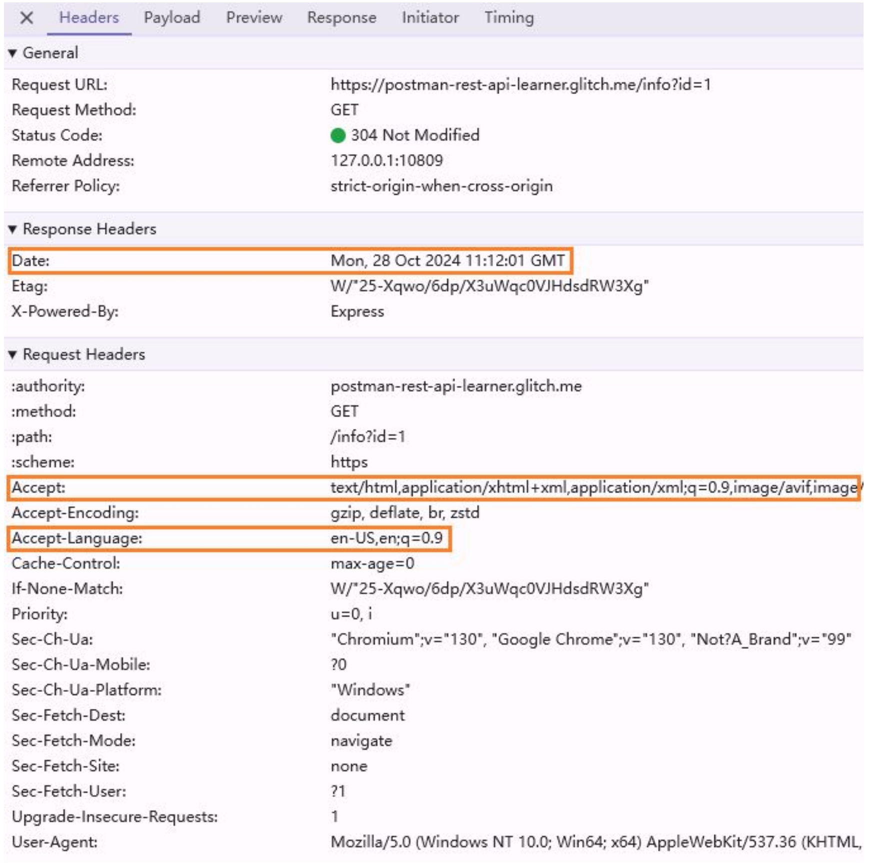
https://restfulapi.net/

REST Components

- Resource: **abstraction** of information
  - Identifiers
    - URL endpoint…
  - Metadata
    - Source links
    - Alternate data formats
    - Vary: language preferences…
    - …
- Representation: **state** of the resource
  - Metadata
    - Media type
    - Last-modified time
    - …
- Control data: interaction behavior
  - Cache-Control
  - …

JSON, XML, …

application/json



Headers | Payload | Preview | Response | Initiator | Timing

▼ General

Request URL:	https://postman-rest-api-learner.glitch.me/info?id=1
Request Method:	GET
Status Code:	● 304 Not Modified
Remote Address:	127.0.0.1:10809
Referrer Policy:	strict-origin-when-cross-origin

▼ Response Headers

Date:	Mon, 28 Oct 2024 11:12:01 GMT
Etag:	W/"25-Xqwo/6dp/X3uWqc0VJHdsdRW3Xg"
X-Powered-By:	Express

▼ Request Headers

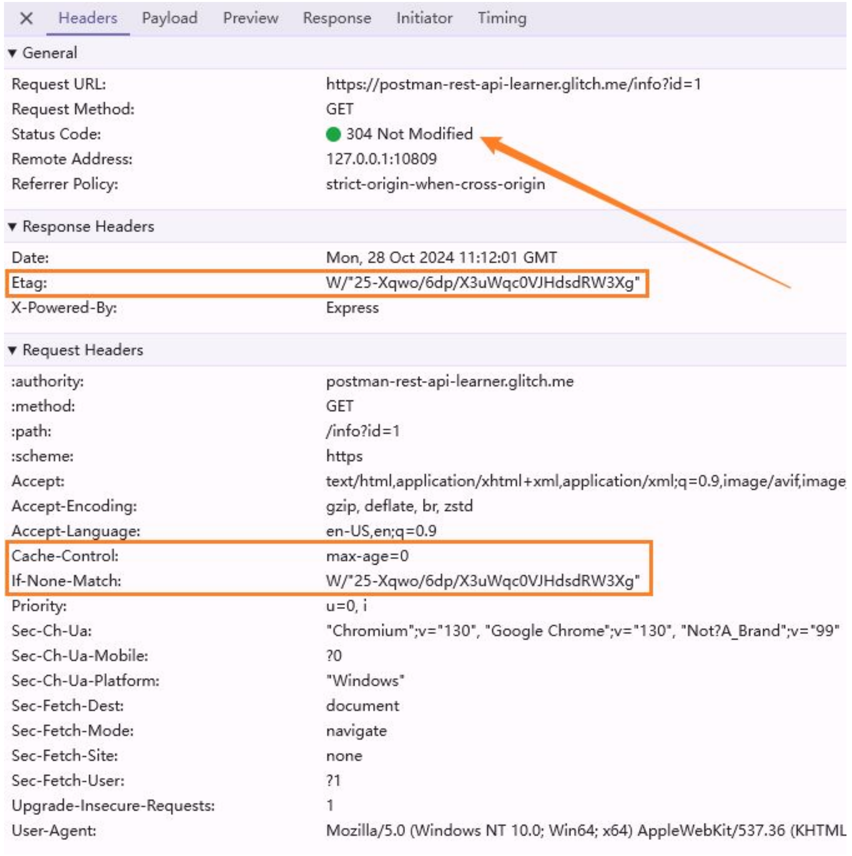
:authority:	postman-rest-api-learner.glitch.me
:method:	GET
:path:	/info?id=1
:scheme:	https
Accept:	text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
Accept-Encoding:	gzip, deflate, br, zstd
Accept-Language:	en-US,en;q=0.9
Cache-Control:	max-age=0
If-None-Match:	W/"25-Xqwo/6dp/X3uWqc0VJHdsdRW3Xg"
Priority:	u=0, i
Sec-Ch-Ua:	"Chromium";v="130", "Google Chrome";v="130", "Not?A_Brand";v="99"
Sec-Ch-Ua-Mobile:	?0
Sec-Ch-Ua-Platform:	"Windows"
Sec-Fetch-Dest:	document
Sec-Fetch-Mode:	navigate
Sec-Fetch-Site:	none
Sec-Fetch-User:	?1
Upgrade-Insecure-Requests:	1
User-Agent:	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,

GET https://postman-rest-api-learner.glitch.me/info?id=1

REST Components

- Resource: **abstraction** of information
  - Identifiers
    - URL endpoint…
  - Metadata
    - Source links
    - Alternate data formats
    - Vary: language preferences…
    - …
- Representation: **state** of the resource
  - Metadata
    - Media type
    - Last-modified time
    - …
- Control data: interaction behavior
  - Cache-Control
  - …

Try hard-reload (Ctrl+F5) ● 200 OK



Headers Payload Preview Response Initiator Timing

▼ General

Request URL:	https://postman-rest-api-learner.glitch.me/info?id=1
Request Method:	GET
Status Code:	● 304 Not Modified
Remote Address:	127.0.0.1:10809
Referrer Policy:	strict-origin-when-cross-origin

▼ Response Headers

Date:	Mon, 28 Oct 2024 11:12:01 GMT
Etag:	W/"25-Xqwo/6dp/X3uWqc0VJHdsdRW3Xg"
X-Powered-By:	Express

▼ Request Headers

:authority:	postman-rest-api-learner.glitch.me
:method:	GET
:path:	/info?id=1
:scheme:	https
Accept:	text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image
Accept-Encoding:	gzip, deflate, br, zstd
Accept-Language:	en-US,en;q=0.9
Cache-Control:	max-age=0
If-None-Match:	W/"25-Xqwo/6dp/X3uWqc0VJHdsdRW3Xg"
Priority:	u=0, i
Sec-Ch-Ua:	"Chromium";v="130", "Google Chrome";v="130", "Not?A_Brand";v="99"
Sec-Ch-Ua-Mobile:	?0
Sec-Ch-Ua-Platform:	"Windows"
Sec-Fetch-Dest:	document
Sec-Fetch-Mode:	navigate
Sec-Fetch-Site:	none
Sec-Fetch-User:	?1
Upgrade-Insecure-Requests:	1
User-Agent:	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML

GET https://postman-rest-api-learner.glitch.me/info?id=1

REST - 6 Guiding Principles

- Client-Server
  - Separation of concerns
  - Modularization
  - Features:
    - Portability of client
    - Scalability of server
- **Stateless**
- Cacheable
- Uniform Interface
- Layered System
- Code on Demand (*)

- Recall Lab4 when we intend to refactor a local program into a modular service.

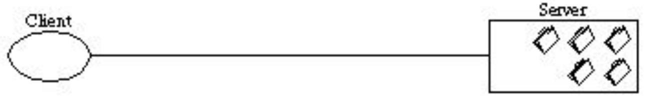
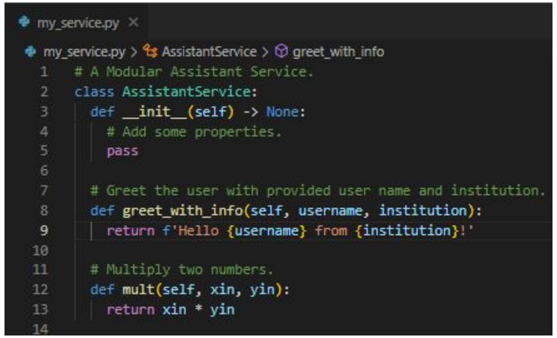
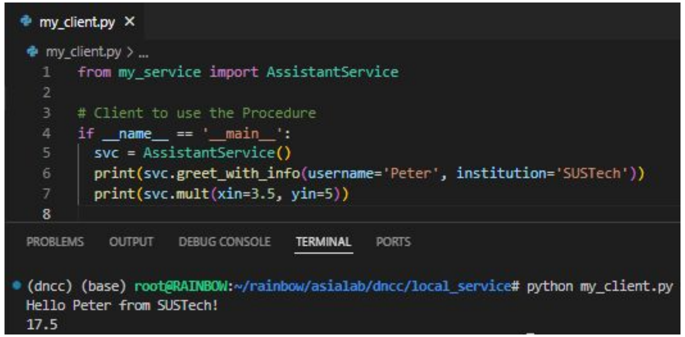


Figure 5-2. Client-Server



```python
# A Modular Assistant Service.
class AssistantService:
    def __init__(self) -> None:
        # Add some properties.
        pass

    # Greet the user with provided user name and institution.
    def greet_with_info(self, username, institution):
        return f'Hello {username} from {institution}!'

    # Multiply two numbers.
    def mult(self, xin, yin):
        return xin * yin
```



```python
from my_service import AssistantService

# Client to use the Procedure
if __name__ == '__main__':
    svc = AssistantService()
    print(svc.greet_with_info(username='Peter', institution='SUSTech'))
    print(svc.mult(xin=3.5, yin=5))
```

```
(dncc) (base) root@RAINBOW:~/rainbow/asialab/dncc/local_service# python my_client.py
Hello Peter from SUSTech!
17.5
```

https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

REST - 6 Guiding Principles

- Client-Server
- **Stateless**
  - No session state on server
  - Request contains all necessary info
  - Requests do not rely on one another
  - Features:
    - Visibility
    - Reliability
    - Scalability
  - Concern: repetitive data overhead
- Cacheable
- Uniform Interface
- Layered System
- Code on Demand (*)

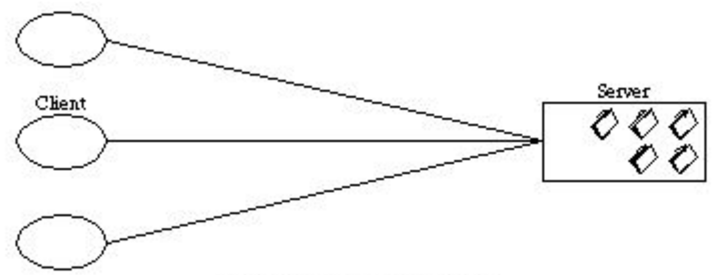


Figure 5-3. Client-Stateless-Server

Request:
1. Add a new book (id=1) in the bookshelf.
2. **Give me that created book.** ✗

Request:
1. Add a new book (id=1) in the bookshelf.
2. **Give me the book (id=1).** √

https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

REST - 6 Guiding Principles

- Client-Server
- **Stateless**
- Cacheable
  - Response data should implicitly or explicitly announce cacheability.
  - If cacheable, client cache is reused.
  - Features:
    - Efficiency
    - …
  - Concern: stale data handling
- Uniform Interface
- Layered System
- Code on Demand (*)

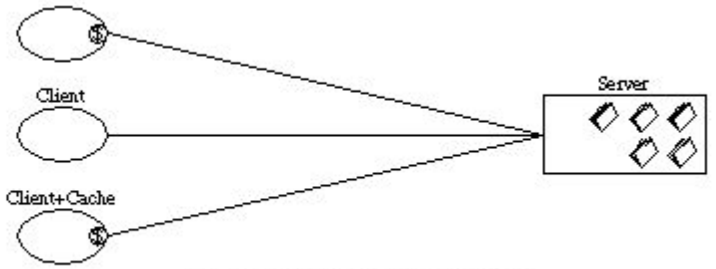
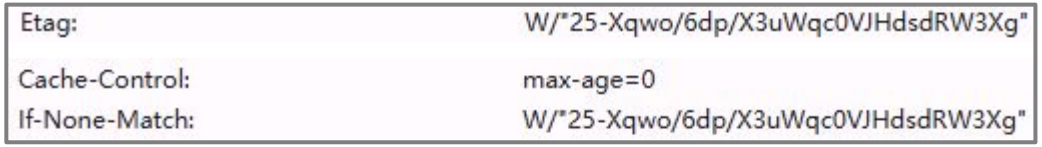


Figure 5-4. Client-Cache-Stateless-Server

Response: ● 200 OK

1. The story is: “Alice was beginning to …”
2. **The story does not change. Check the previous copy.** ● 304 Not Modified

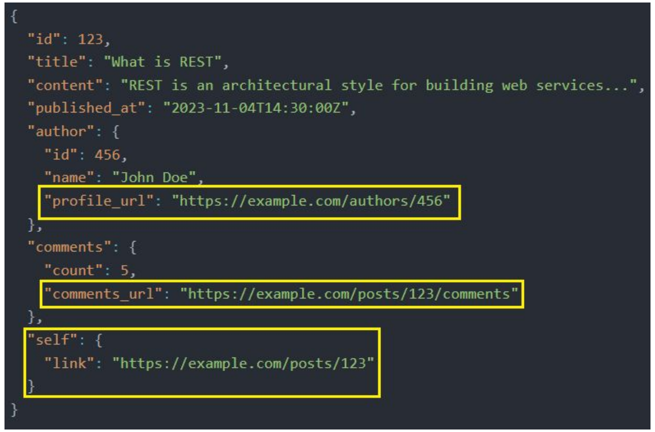


Etag:	W/"25-Xqwo/6dp/X3uWqc0VJHdsdRW3Xg"
Cache-Control:	max-age=0
If-None-Match:	W/"25-Xqwo/6dp/X3uWqc0VJHdsdRW3Xg"

https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

REST - 6 Guiding Principles

- Client-Server
- **Stateless**
- Cacheable
- Uniform Interface
  - Principle of generality
  - **Identification of resources**: uniquely identify each resource
  - **Manipulation of resources through representations**: uniform representations
  - **Self-descriptive messages**: enough info in each resource representation
  - **Hypermedia as the engine of application state**: drive resources via hyperlinks
  - Features:
    - Simplicity
    - Independent evolvability
    - …
  - Concern: degradation of efficiency (general *vs.* optimized)
- Layered System
- Code on Demand (*)



```
{
  "id": 123,
  "title": "What is REST",
  "content": "REST is an architectural style for building web services...",
  "published_at": "2023-11-04T14:30:00Z",
  "author": {
    "id": 456,
    "name": "John Doe",
    "profile_url": "https://example.com/authors/456"
  },
  "comments": {
    "count": 5,
    "comments_url": "https://example.com/posts/123/comments"
  },
  "self": {
    "link": "https://example.com/posts/123"
  }
}
```

NOTE: Principles are just guidelines. It is not compulsory to follow all mentioned guidelines to become a RESTful API.

https://restfulapi.net/

REST - 6 Guiding Principles

- Client-Server
- **Stateless**
- Cacheable
- Uniform Interface
- Layered System
  - Hierarchical layers
  - Example: MVC pattern
  - Features:
    - Reduced coupling
      - Evolvability
      - Reusability
    - Scalability
    - Transparency
  - Concern: complexity & processing overhead
- Code on Demand (*)

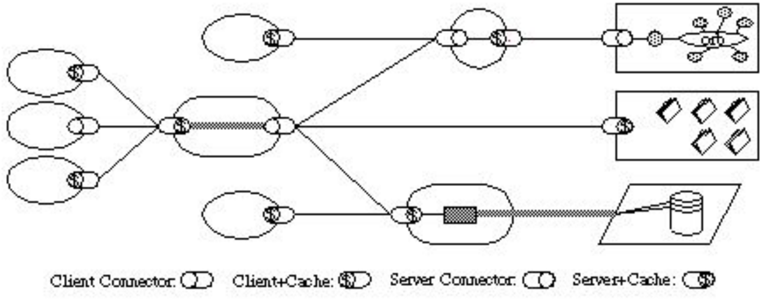
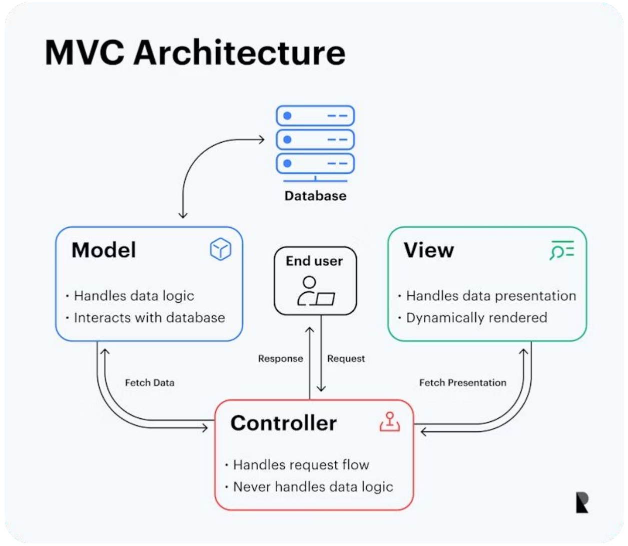


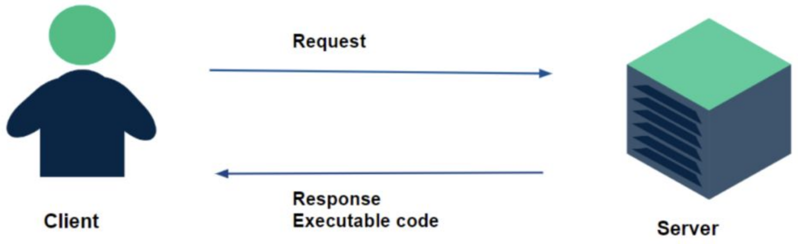
Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server



https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
https://www.ramotion.com/blog/mvc-architecture-in-web-application/

REST - 6 Guiding Principles

- Client-Server
- **Stateless**
- Cacheable
- Uniform Interface
- Layered System
- Code on Demand (*)
    - Extend client functionality by downloading code from server
    - Example: server sends JavaScript code to client to execute
    - Features:
        - Simplicity of client init feature
        - Flexibility
    - Concern: visibility & security

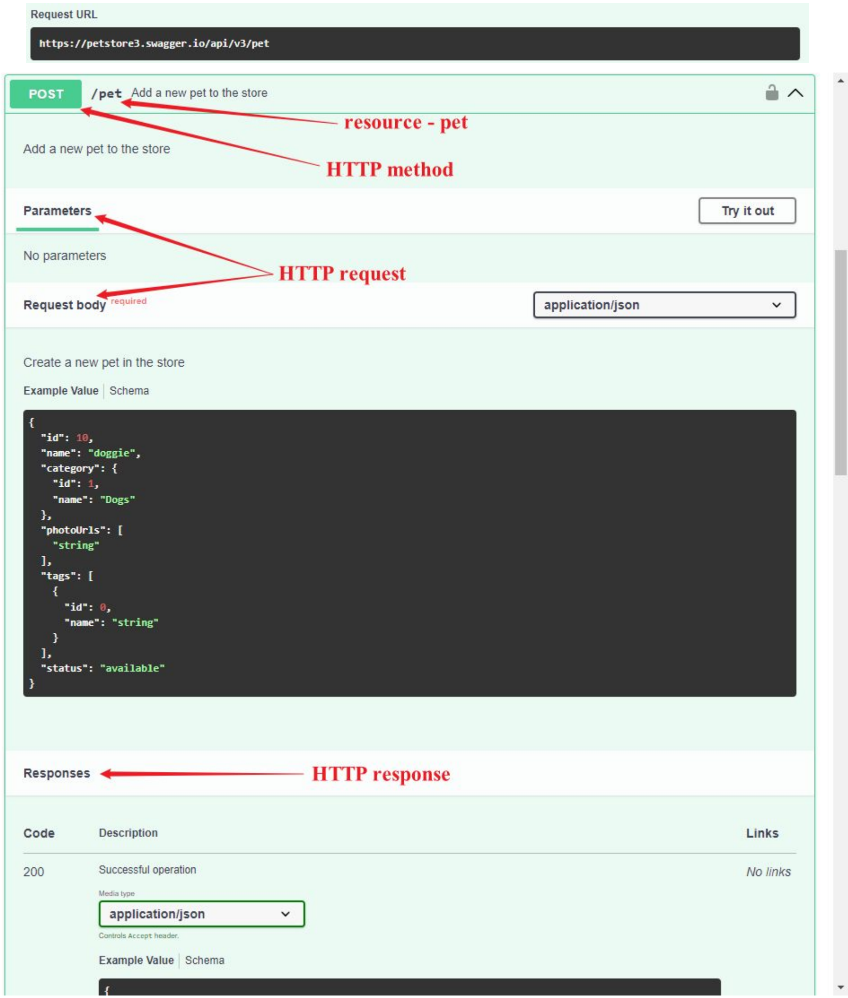


https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

RESTful API Components

Take OpenAPI specification as an example:

- Resource: identified by a URL endpoint.
- HTTP method: specifies an operation on the resource (e.g., GET, POST, etc.).
- HTTP request:
  - Header: metadata
    - `Content-Type = application/json`
  - Query Parameters
    - `/users?age=25&sort=desc`
  - Request Body
- HTTP response:
  - Status code + message
    - `200 OK`
    - `404 Not Found`
    - `...`
  - Header
  - Response Body



Request URL

`https://petstore3.swagger.io/api/v3/pet`

POST /pet Add a new pet to the store

Add a new pet to the store

Parameters

No parameters

Request body required — application/json

Create a new pet in the store

Example Value | Schema

```
{
  "id": 10,
  "name": "doggie",
  "category": {
    "id": 1,
    "name": "Dogs"
  },
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

Responses

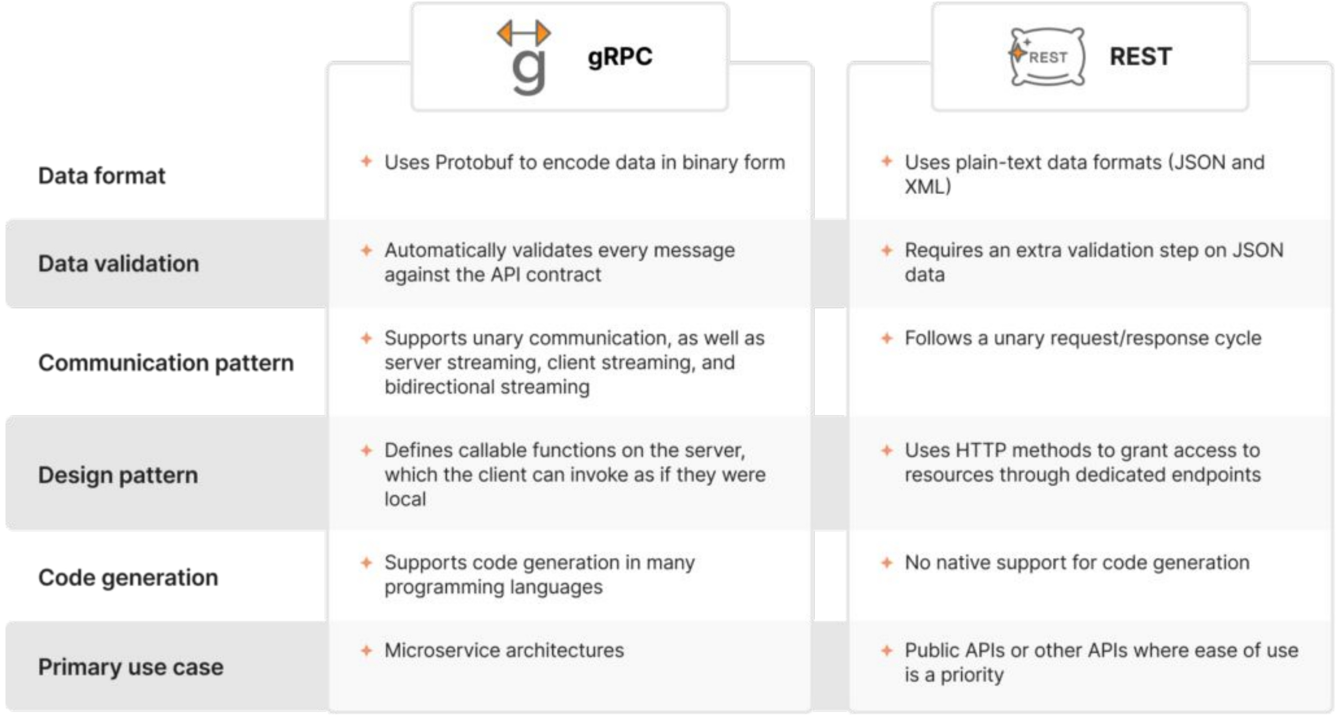
Code	Description	Links
200	Successful operation Media type: application/json Controls Accept header. Example Value | Schema	*No links*

RESTful API - Benefits

- **Scalability**
  a. Client-Server enables multiple server duplicates with load balancing.
  b. Stateless ensures that each request holds the entire context and is independently processable, thus the servers can be easily scaled out.
  c. Cacheable reduces the load of the servers so they can server more clients. Introducing distributed caching further improves the service capacity.
- **Flexibility & Independence**
  a. Client-Server allows the client and the server to be evolved independently without affecting each other.
  b. Layered System further decouples service logic so that each layer can be designed and implemented separately.
  c. Flexibility: Code on Demand allows the client to extend new functionalities when needed provided by the server via downloading code.

- Everything is organized thanks to Uniform Interface.

RESTful API *vs.* gRPC



	gRPC	REST
Data format	✦ Uses Protobuf to encode data in binary form	✦ Uses plain-text data formats (JSON and XML)
Data validation	✦ Automatically validates every message against the API contract	✦ Requires an extra validation step on JSON data
Communication pattern	✦ Supports unary communication, as well as server streaming, client streaming, and bidirectional streaming	✦ Follows a unary request/response cycle
Design pattern	✦ Defines callable functions on the server, which the client can invoke as if they were local	✦ Uses HTTP methods to grant access to resources through dedicated endpoints
Code generation	✦ Supports code generation in many programming languages	✦ No native support for code generation
Primary use case	✦ Microservice architectures	✦ Public APIs or other APIs where ease of use is a priority

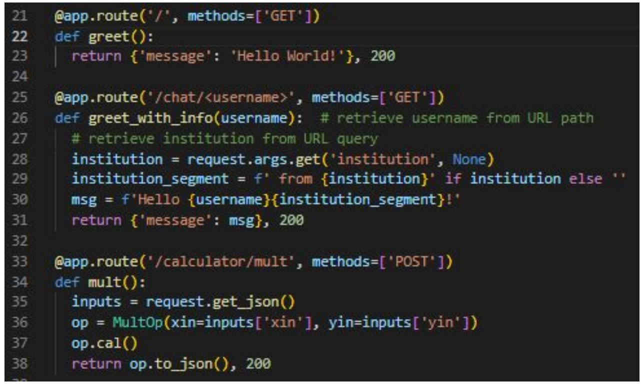
https://blog.postman.com/grpc-vs-rest/

TASK: RESTful API - Hello World with Python Flask

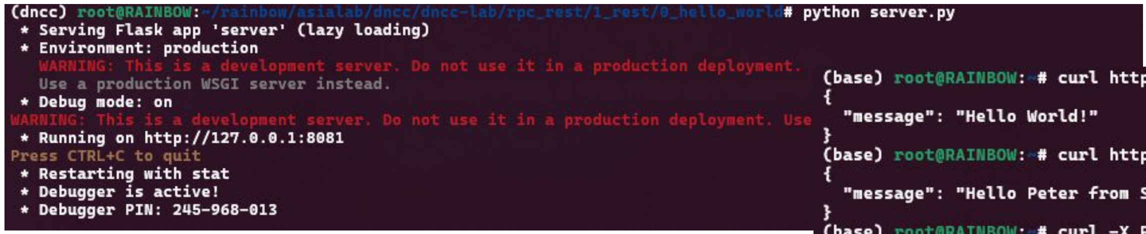
Implement a simple RESTful API server using Python Flask.

> Reference codebase: `rest_hello_world`

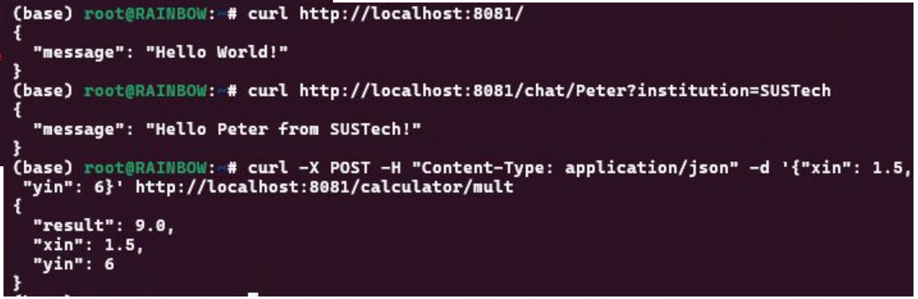
1. Set up Python ([Miniconda]() is recommended).
2. Install Python dependencies into a Conda environment via:
   - `python -m pip install -r requirements.txt`
3. Run the API server via:
   - `python server.py`
4. In another terminal, test the API with HTTP requests.



```python
@app.route('/', methods=['GET'])
def greet():
  return {'message': 'Hello World!'}, 200

@app.route('/chat/<username>', methods=['GET'])
def greet_with_info(username):  # retrieve username from URL path
  # retrieve institution from URL query
  institution = request.args.get('institution', None)
  institution_segment = f' from {institution}' if institution else ''
  msg = f'Hello {username}{institution_segment}!'
  return {'message': msg}, 200

@app.route('/calculator/mult', methods=['POST'])
def mult():
  inputs = request.get_json()
  op = MultOp(xin=inputs['xin'], yin=inputs['yin'])
  op.cal()
  return op.to_json(), 200
```



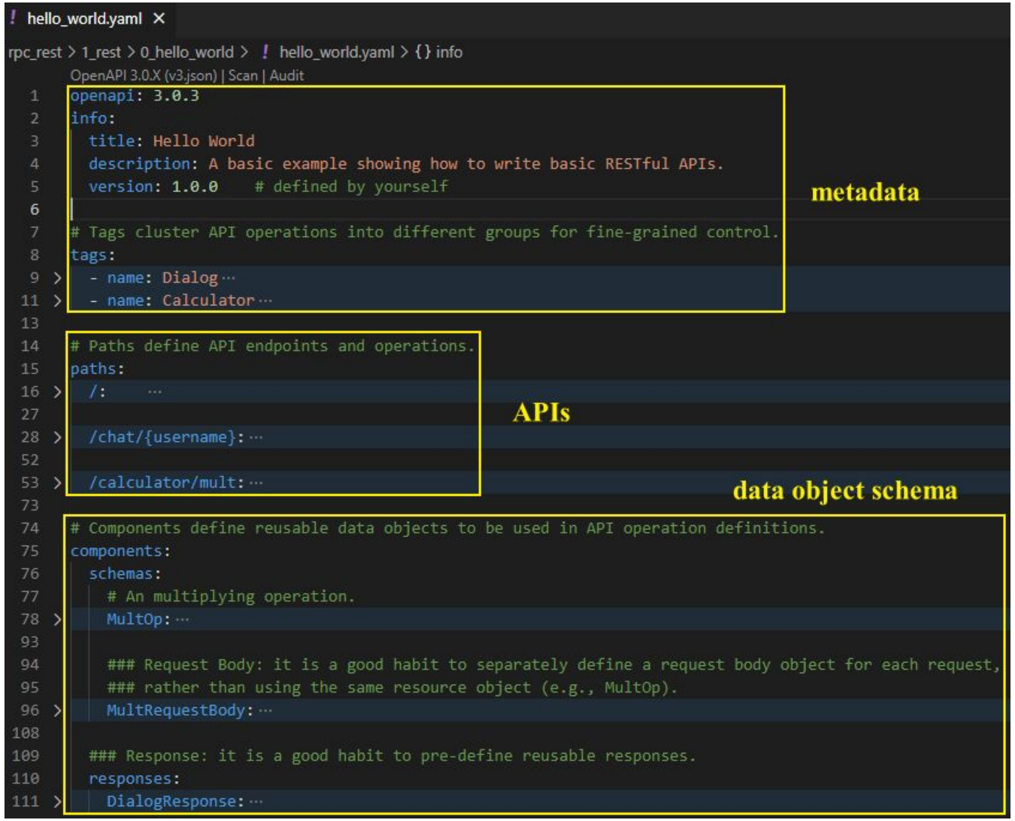
```
(dncc) root@RAINBOW:~/rainbow/asialab/dncc/dncc-lab/rpc_rest/1_rest/0_hello_world# python server.py
 * Serving Flask app 'server' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use
 * Running on http://127.0.0.1:8081
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 245-968-013
```



```
(base) root@RAINBOW:~# curl http://localhost:8081/
{
  "message": "Hello World!"
}
(base) root@RAINBOW:~# curl http://localhost:8081/chat/Peter?institution=SUSTech
{
  "message": "Hello Peter from SUSTech!"
}
(base) root@RAINBOW:~# curl -X POST -H "Content-Type: application/json" -d '{"xin": 1.5,
 "yin": 6}' http://localhost:8081/calculator/mult
{
  "result": 9.0,
  "xin": 1.5,
  "yin": 6
}
```

OpenAPI As a RESTful API Specification

- Refer to OpenAPI Specification from Swagger.
- Specification document file can be either JSON or **YAML**.
- Metadata:
  - OpenAPI version
  - Info:
    - Title
    - API spec version
  - Tags (*)
- API paths & operations
  - URL endpoint
  - Method
  - Request & Response
- Components - Schemas
  - Set of reusable data objects

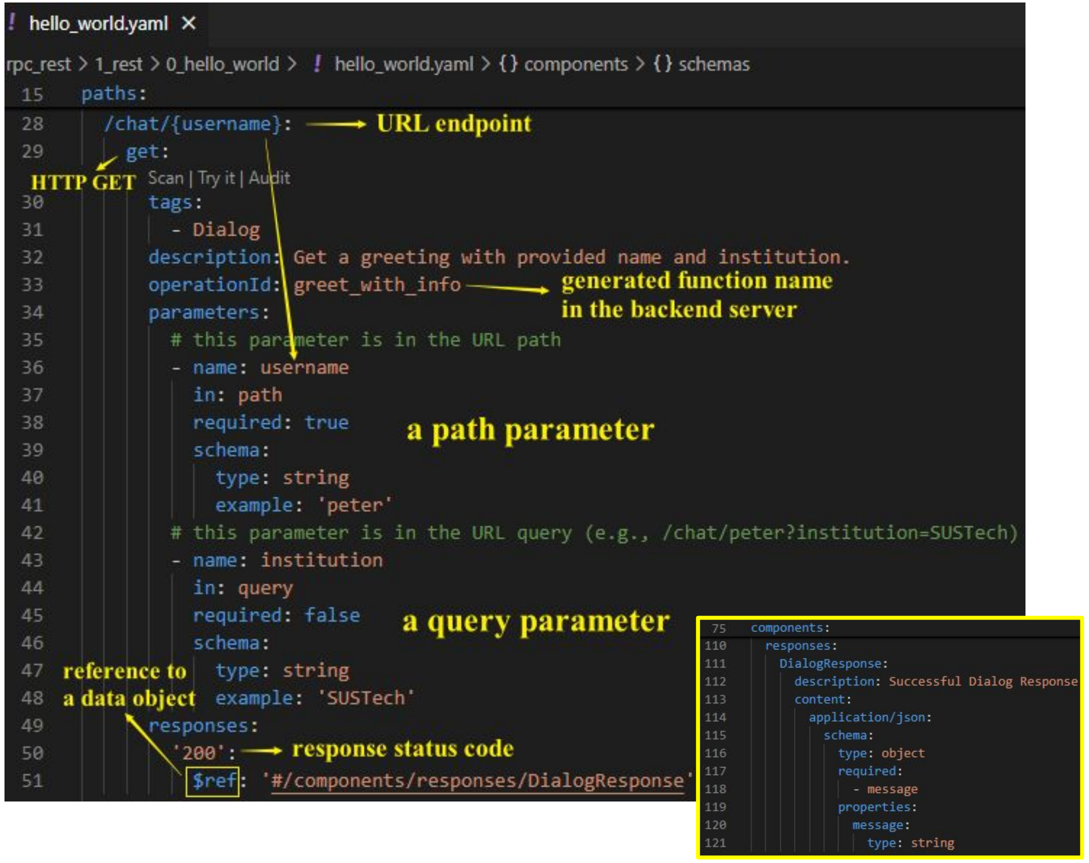

hello_world.yaml ×
rpc_rest > 1_rest > 0_hello_world > ! hello_world.yaml > {} info
OpenAPI 3.0.X (v3.json) | Scan | Audit
metadata
APIs
data object schema

```yaml
openapi: 3.0.3
info:
  title: Hello World
  description: A basic example showing how to write basic RESTful APIs.
  version: 1.0.0    # defined by yourself

# Tags cluster API operations into different groups for fine-grained control.
tags:
  - name: Dialog···
  - name: Calculator···

# Paths define API endpoints and operations.
paths:
  /:    ···

  /chat/{username}:···

  /calculator/mult:···

# Components define reusable data objects to be used in API operation definitions.
components:
  schemas:
    # An multiplying operation.
    MultOp:···

    ### Request Body: it is a good habit to separately define a request body object for each request,
    ### rather than using the same resource object (e.g., MultOp).
    MultRequestBody:···

  ### Response: it is a good habit to pre-define reusable responses.
  responses:
    DialogResponse:···
```

OpenAPI As a RESTful API Specification

- Refer to OpenAPI Specification from Swagger.
- Specification document file can be either JSON or **YAML**.
- Metadata:
  - OpenAPI version
  - Info:
    - Title
    - API spec version
  - Tags (*)
- API paths & operations
  - URL endpoint
  - Method
  - Request & Response
- Components - Schemas
  - Set of reusable data objects

A GET example



hello_world.yaml

rpc_rest > 1_rest > 0_hello_world > hello_world.yaml > {} components > {} schemas

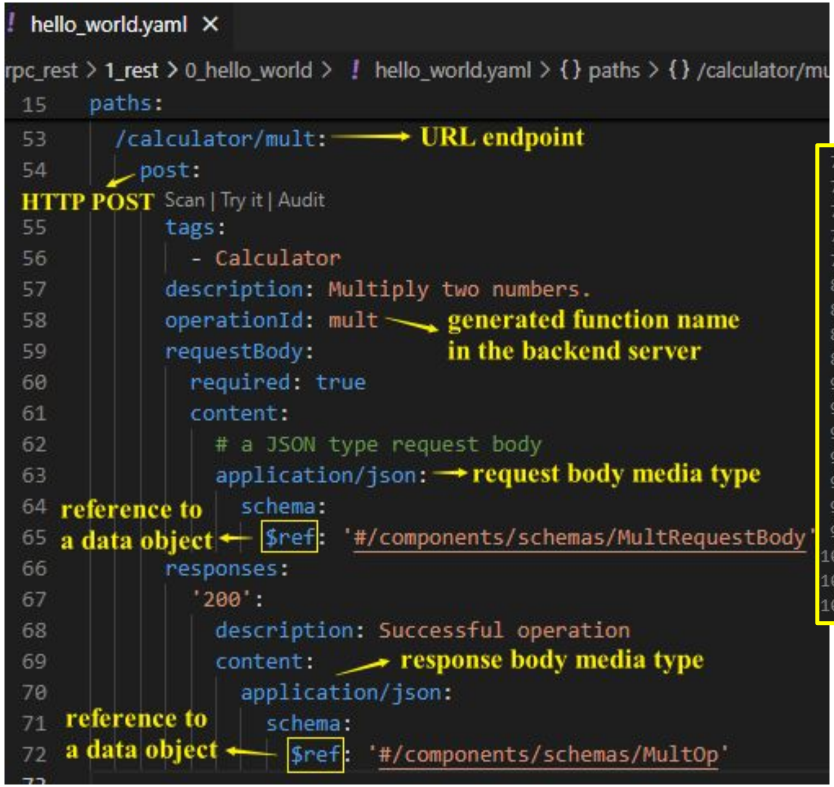
```yaml
paths:
  /chat/{username}:
    get:
      tags:
        - Dialog
      description: Get a greeting with provided name and institution.
      operationId: greet_with_info
      parameters:
        # this parameter is in the URL path
        - name: username
          in: path
          required: true
          schema:
            type: string
            example: 'peter'
        # this parameter is in the URL query (e.g., /chat/peter?institution=SUSTech)
        - name: institution
          in: query
          required: false
          schema:
            type: string
            example: 'SUSTech'
      responses:
        '200':
          $ref: '#/components/responses/DialogResponse'
```

```yaml
components:
  responses:
    DialogResponse:
      description: Successful Dialog Response
      content:
        application/json:
          schema:
            type: object
            required:
              - message
            properties:
              message:
                type: string
```

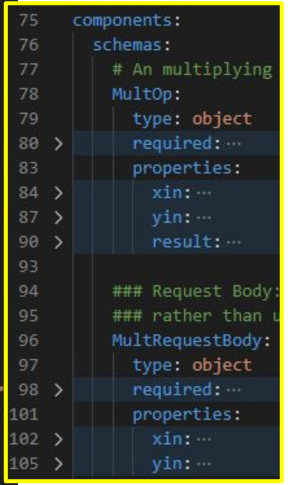
OpenAPI As a RESTful API Specification

- Refer to OpenAPI Specification from Swagger.
- Specification document file can be either JSON or **YAML**.
- Metadata:
  - OpenAPI version
  - Info:
    - Title
    - API spec version
  - Tags (*)
- API paths & operations
  - URL endpoint
  - Method
  - Request & Response
- Components - Schemas
  - Set of reusable data objects

A POST example


hello_world.yaml
rpc_rest > 1_rest > 0_hello_world > hello_world.yaml > {} paths > {} /calculator/mu
URL endpoint
HTTP POST
Scan | Try it | Audit
generated function name in the backend server
request body media type
reference to a data object
response body media type
reference to a data object

```
paths:
  /calculator/mult:
    post:
      tags:
        - Calculator
      description: Multiply two numbers.
      operationId: mult
      requestBody:
        required: true
        content:
          # a JSON type request body
          application/json:
            schema:
              $ref: '#/components/schemas/MultRequestBody'
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/MultOp'
```



```
components:
  schemas:
    # An multiplying
    MultOp:
      type: object
      required: ···
      properties:
        xin: ···
        yin: ···
        result: ···

    ### Request Body:
    ### rather than u
    MultRequestBody:
      type: object
      required: ···
      properties:
        xin: ···
        yin: ···
```

OpenAPI - Benefits

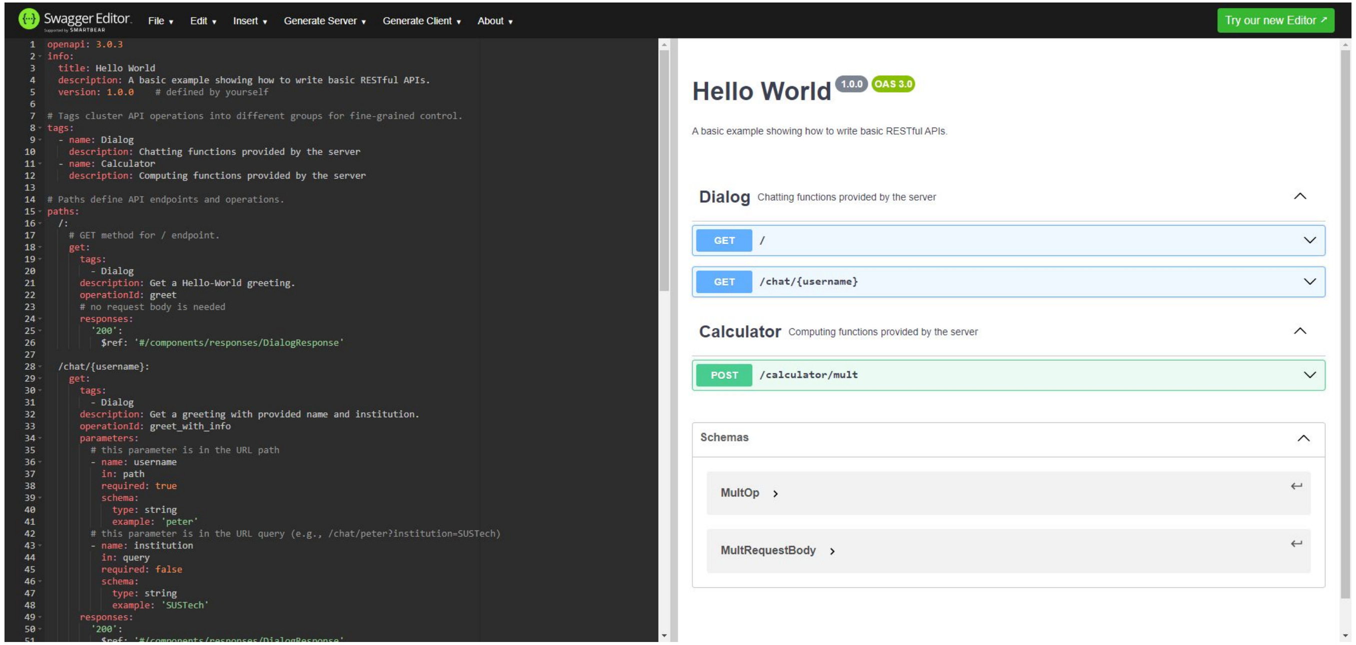
- **Standardization**
  - Client & Server reaches an agreement on available APIs and access requirements.
- **Human-Readable Format**
  - JSON/YAML is both clean & programmatically operable.
- **API Documentation & Visualization**:
  - Swagger UI: visualize & interact with APIs in a web page
  - Swagger Editor: edit API specification file with real-time Swagger UI support
- **Multi-Language Client & Server Code Template Generation**:
  - Client/Server code of different frameworks from different programming languages can be automatically generated as development templates.
  - OpenAPI Generator
- **Integrated Testing Capability**:
  - Variable tools support easy setup for API testing.
  - cURL
  - Swagger Mock Server
  - Postman

TASK: RESTful API - Hello World with Python Flask

Implement a simple RESTful API server using Python Flask.

> Reference codebase: `rest_hello_world`

5. Check the API specification file `hello_world.yaml`
6. Copy the file content into the online Swagger Editor. Explore the generated Swagger UI page.



```yaml
openapi: 3.0.3
info:
  title: Hello World
  description: A basic example showing how to write basic RESTful APIs.
  version: 1.0.0    # defined by yourself

# Tags cluster API operations into different groups for fine-grained control.
tags:
  - name: Dialog
    description: Chatting functions provided by the server
  - name: Calculator
    description: Computing functions provided by the server

# Paths define API endpoints and operations.
paths:
  /:
    # GET method for / endpoint.
    get:
      tags:
        - Dialog
      description: Get a Hello-World greeting.
      operationId: greet
      # no request body is needed
      responses:
        '200':
          $ref: '#/components/responses/DialogResponse'

  /chat/{username}:
    get:
      tags:
        - Dialog
      description: Get a greeting with provided name and institution.
      operationId: greet_with_info
      parameters:
        # this parameter is in the URL path
        - name: username
          in: path
          required: true
          schema:
            type: string
            example: 'peter'
        # this parameter is in the URL query (e.g., /chat/peter?institution=SUSTech)
        - name: institution
          in: query
          required: false
          schema:
            type: string
            example: 'SUSTech'
      responses:
        '200':
          $ref: '#/components/responses/DialogResponse'
```

Hello World 1.0.0 OAS 3.0

A basic example showing how to write basic RESTful APIs.

Dialog Chatting functions provided by the server

GET /

GET /chat/{username}

Calculator Computing functions provided by the server

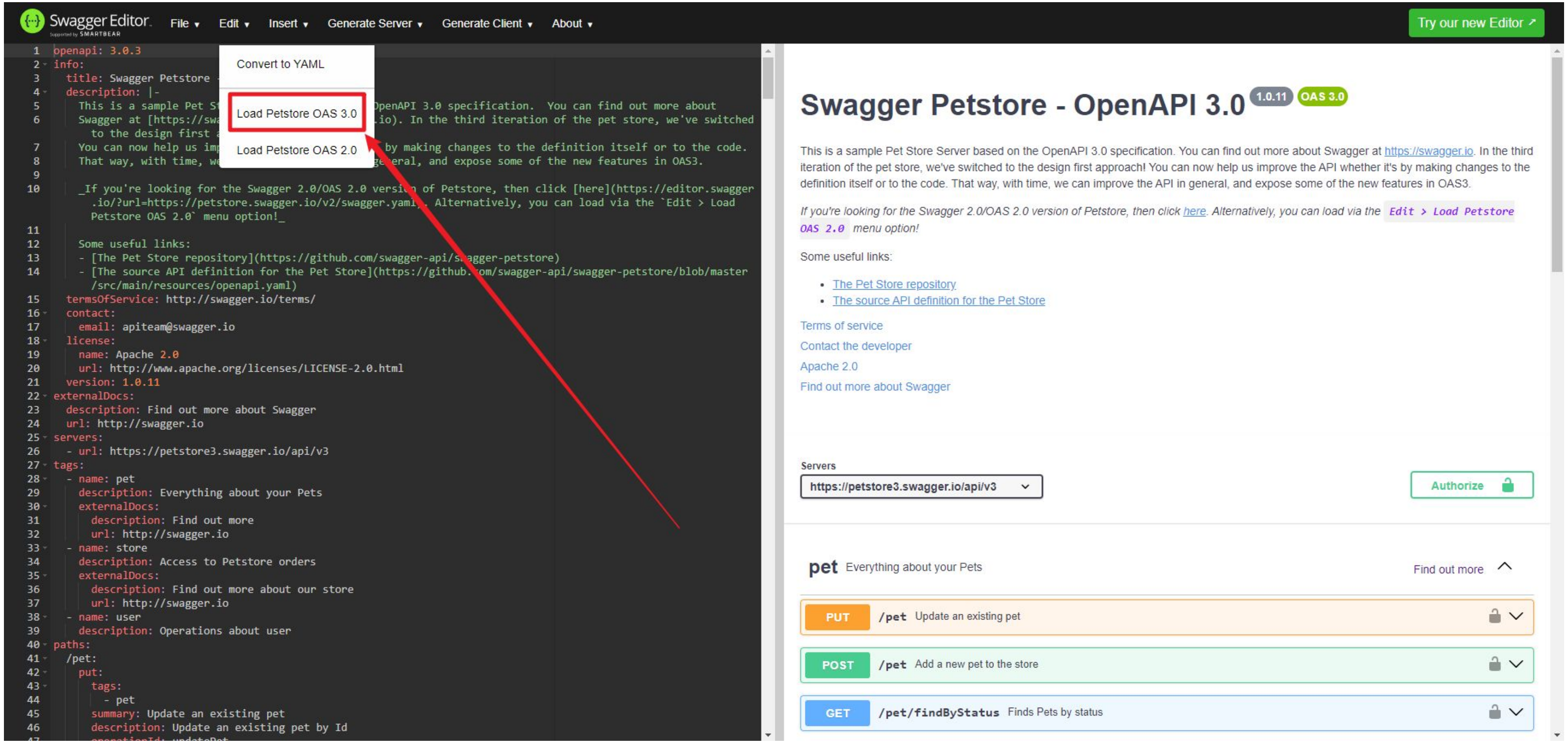
POST /calculator/mult

Schemas

MultOp

MultRequestBody

Another RESTful API Example - Petstore



```yaml
openapi: 3.0.3
info:
  title: Swagger Petstore -
  description: |-
    This is a sample Pet St                              OpenAPI 3.0 specification.  You can find out more about
    Swagger at [https://swa                              .io). In the third iteration of the pet store, we've switched
      to the design first a
    You can now help us imp                              by making changes to the definition itself or to the code.
    That way, with time, we                              general, and expose some of the new features in OAS3.

    _If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click [here](https://editor.swagger
      .io/?url=https://petstore.swagger.io/v2/swagger.yaml). Alternatively, you can load via the `Edit > Load
      Petstore OAS 2.0` menu option!_

    Some useful links:
    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
    - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-petstore/blob/master
      /src/main/resources/openapi.yaml)
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.11
externalDocs:
  description: Find out more about Swagger
  url: http://swagger.io
servers:
  - url: https://petstore3.swagger.io/api/v3
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: http://swagger.io
  - name: store
    description: Access to Petstore orders
    externalDocs:
      description: Find out more about our store
      url: http://swagger.io
  - name: user
    description: Operations about user
paths:
  /pet:
    put:
      tags:
        - pet
      summary: Update an existing pet
      description: Update an existing pet by Id
```

Swagger Petstore - OpenAPI 3.0

This is a sample Pet Store Server based on the OpenAPI 3.0 specification. You can find out more about Swagger at https://swagger.io. In the third iteration of the pet store, we've switched to the design first approach! You can now help us improve the API whether it's by making changes to the definition itself or to the code. That way, with time, we can improve the API in general, and expose some of the new features in OAS3.

If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click here. Alternatively, you can load via the `Edit > Load Petstore OAS 2.0` *menu option!*

Some useful links:

- The Pet Store repository
- The source API definition for the Pet Store

Terms of service

Contact the developer

Apache 2.0

Find out more about Swagger

https://petstore3.swagger.io/api/v3

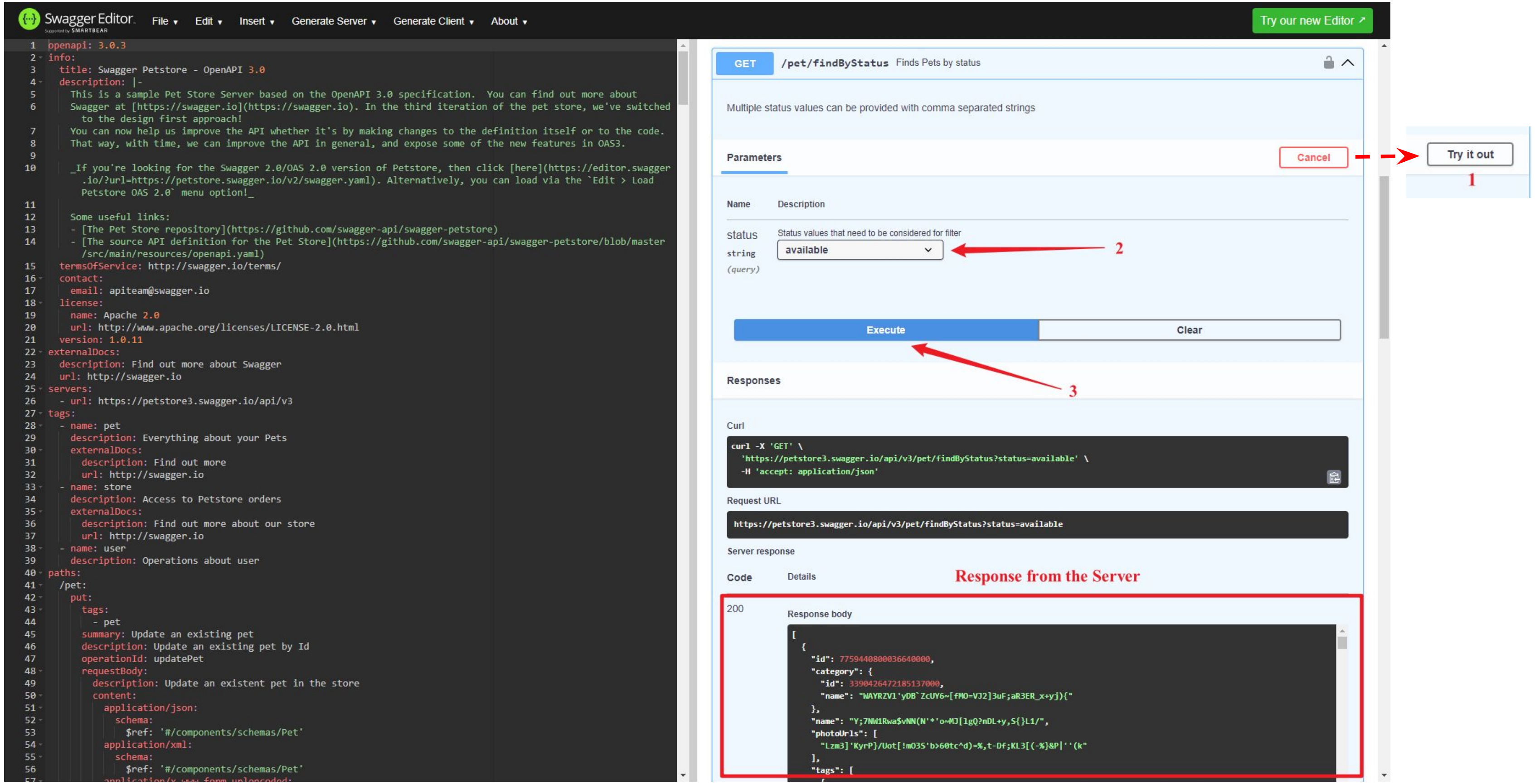
pet Everything about your Pets — Find out more

PUT `/pet` Update an existing pet

POST `/pet` Add a new pet to the store

GET `/pet/findByStatus` Finds Pets by status

Another RESTful API Example - Petstore



```
openapi: 3.0.3
info:
  title: Swagger Petstore - OpenAPI 3.0
  description: |-
    This is a sample Pet Store Server based on the OpenAPI 3.0 specification.  You can find out more about
    Swagger at [https://swagger.io](https://swagger.io). In the third iteration of the pet store, we've switched
      to the design first approach!
    You can now help us improve the API whether it's by making changes to the definition itself or to the code.
    That way, with time, we can improve the API in general, and expose some of the new features in OAS3.

    _If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click [here](https://editor.swagger
      .io/?url=https://petstore.swagger.io/v2/swagger.yaml). Alternatively, you can load via the `Edit > Load
      Petstore OAS 2.0` menu option!_

    Some useful links:
    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
    - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-petstore/blob/master
      /src/main/resources/openapi.yaml)
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.11
externalDocs:
  description: Find out more about Swagger
  url: http://swagger.io
servers:
  - url: https://petstore3.swagger.io/api/v3
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: http://swagger.io
  - name: store
    description: Access to Petstore orders
    externalDocs:
      description: Find out more about our store
      url: http://swagger.io
  - name: user
    description: Operations about user
paths:
  /pet:
    put:
      tags:
        - pet
      summary: Update an existing pet
      description: Update an existing pet by Id
      operationId: updatePet
      requestBody:
        description: Update an existent pet in the store
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
          application/xml:
            schema:
              $ref: '#/components/schemas/Pet'
```

GET `/pet/findByStatus` Finds Pets by status

Multiple status values can be provided with comma separated strings

Parameters

Name	Description
status string *(query)*	Status values that need to be considered for filter available

Responses

Curl

```
curl -X 'GET' \
  'https://petstore3.swagger.io/api/v3/pet/findByStatus?status=available' \
  -H 'accept: application/json'
```

Request URL

```
https://petstore3.swagger.io/api/v3/pet/findByStatus?status=available
```

Server response

Code Details

Response from the Server

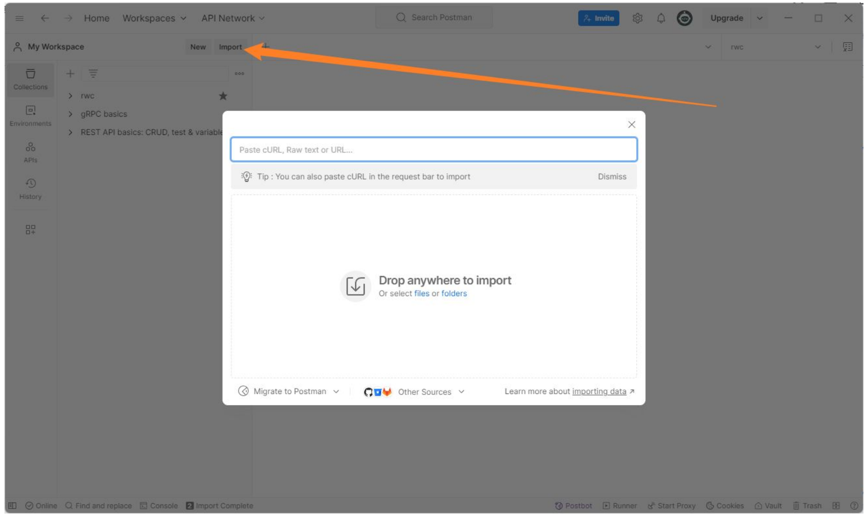
200

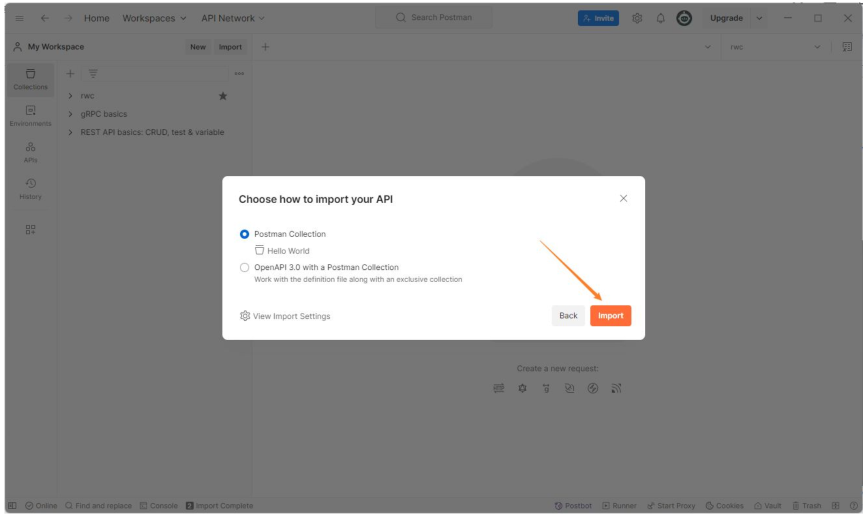
Response body

```
[
  {
    "id": 7759440800036640000,
    "category": {
      "id": 3390426472185137000,
      "name": "WAYRZV1'yDB`ZcUY6~[fMO=VJ2]3uF;aR3ER_x+yj){"
    },
    "name": "Y;7NW1Rwa$vNN(N'*'o~MJ[1gQ?nDL+y,S{}L1/",
    "photoUrls": [
      "Lzm3]'KyrP}/Uot[!mO3S'b>60tc^d)=%,t-Df;KL3[(-%}&P|''(k"
    ],
    "tags": [
```

Postman for RESTful API Testing

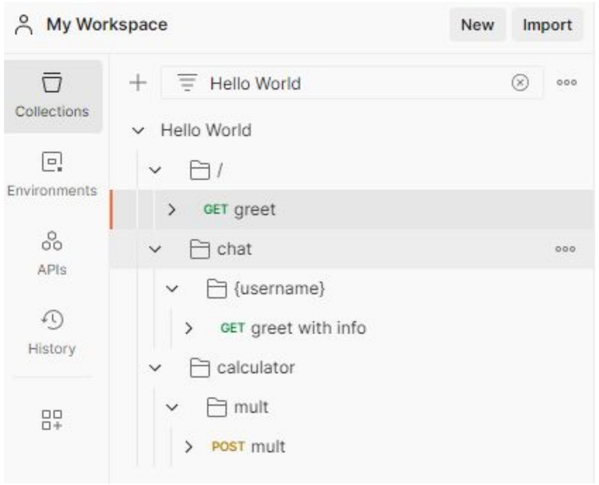
1. Download and install Postman Desktop app.
2. In the workspace, click the “Import” button.
3. Upload the API specification YAML file (e.g., `hello_world.yaml`). Click “Import”.

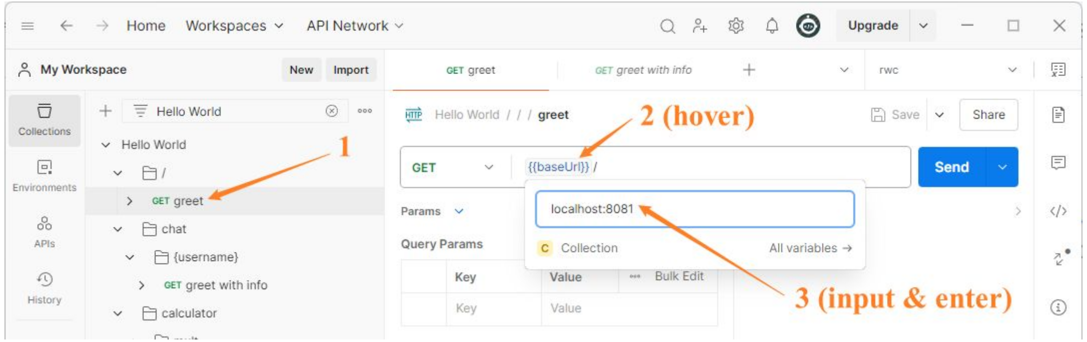




Postman for RESTful API Testing

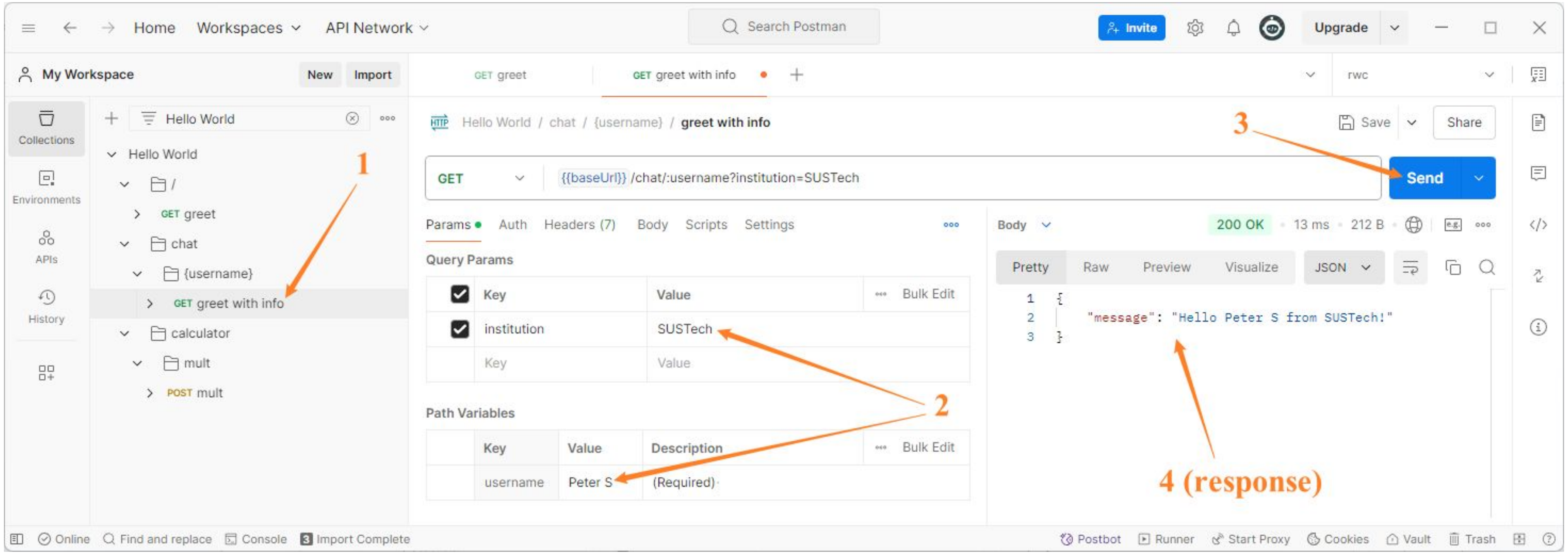
4. A collection of API testing requests will be automatically generated. Note that the URL endpoint contains a base url variable that needs to be configured.
5. Select any request in the collection, hover on the `{{baseUrl}}` variable in the URL endpoint and change its value to the address of the running API backend server.





Postman for RESTful API Testing

6. Now play with these requests with different parameters/body.



GET {{baseUrl}} /chat/:username?institution=SUSTech

Query Params

	Key	Value
✓	institution	SUSTech
	Key	Value

Path Variables

Key	Value	Description
username	Peter S	(Required)

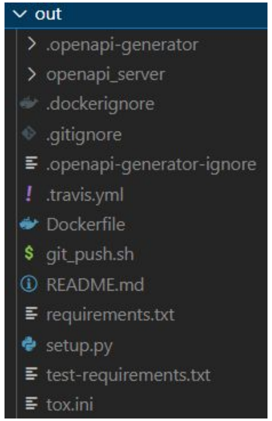
```
{
    "message": "Hello Peter S from SUSTech!"
}
```

TASK: RESTful API via Codegen

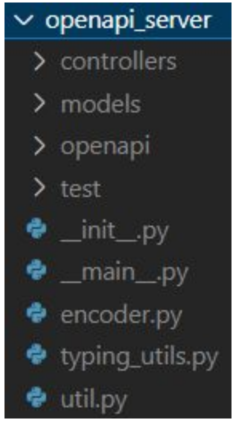
Generate Flask server code template via OpenAPI Generator.

> Reference codebase: `rest_codegen`

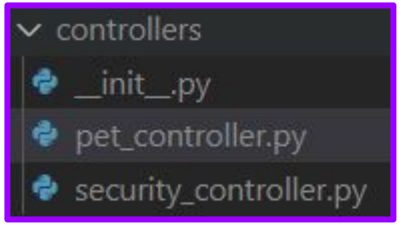
1. Set up Python (Miniconda is recommended).
2. Check the API specification file `petstore.yaml`. Using Docker, generate Python Flask server code template with the OpenAPI Generator CLI:
   - `docker run --rm -v ./:/app/ openapitools/openapi-generator-cli generate -i /app/petstore.yaml -g python-flask -o /app/out/`
3. The server template includes data object models and API controllers.



```
out
  .openapi-generator
  openapi_server
  .dockerignore
  .gitignore
  .openapi-generator-ignore
  .travis.yml
  Dockerfile
  git_push.sh
  README.md
  requirements.txt
  setup.py
  test-requirements.txt
  tox.ini
```

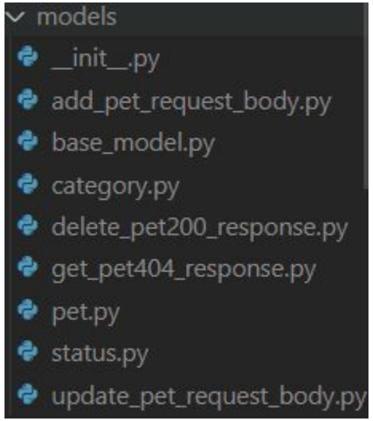


```
openapi_server
  controllers
  models
  openapi
  test
  __init__.py
  __main__.py
  encoder.py
  typing_utils.py
  util.py
```



```
controllers
  __init__.py
  pet_controller.py
  security_controller.py
```

Need to implement

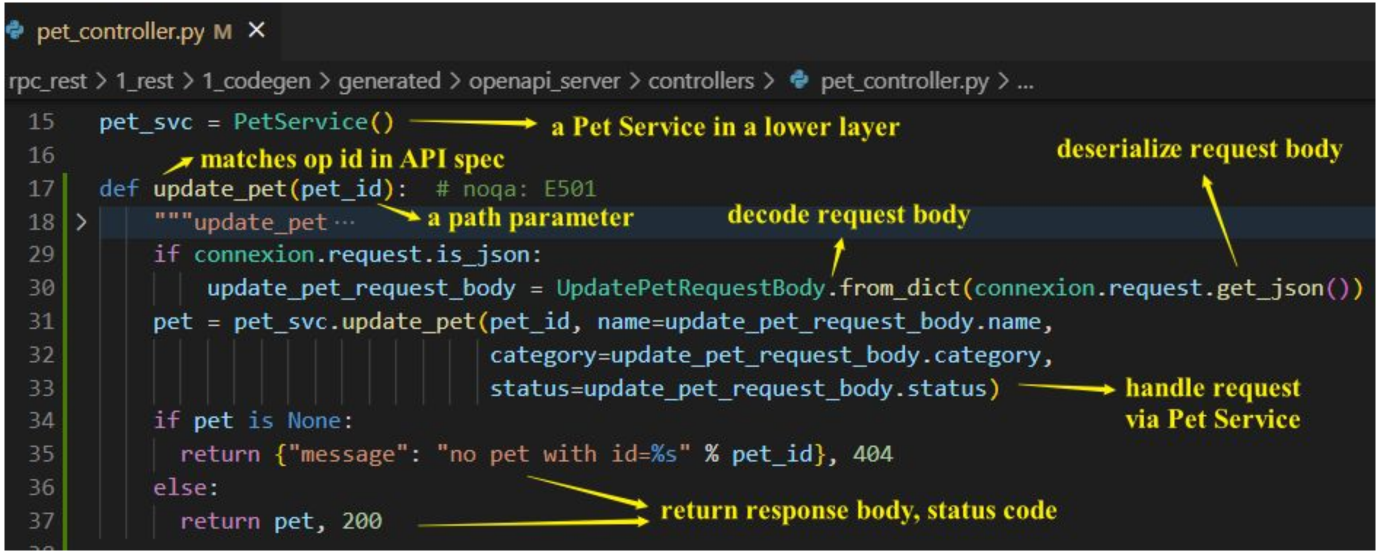


```
models
  __init__.py
  add_pet_request_body.py
  base_model.py
  category.py
  delete_pet200_response.py
  get_pet404_response.py
  pet.py
  status.py
  update_pet_request_body.py
```

TASK: RESTful API via Codegen

Generate Flask server code template via OpenAPI Generator.
> Reference codebase: `rest_codegen`

4. Implement the Pet Controller. The sample implementation follows the MVSC pattern (Model-View-Service-Controller).


pet_controller.py M
rpc_rest > 1_rest > 1_codegen > generated > openapi_server > controllers > pet_controller.py > ...
a Pet Service in a lower layer
matches op id in API spec
a path parameter
decode request body
deserialize request body
handle request via Pet Service
return response body, status code

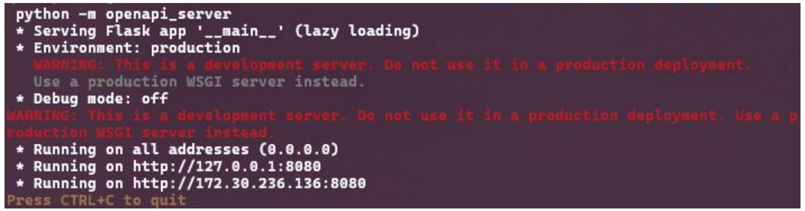
```python
pet_svc = PetService()

def update_pet(pet_id):  # noqa: E501
    """update_pet ...
    if connexion.request.is_json:
        update_pet_request_body = UpdatePetRequestBody.from_dict(connexion.request.get_json())
    pet = pet_svc.update_pet(pet_id, name=update_pet_request_body.name,
                             category=update_pet_request_body.category,
                             status=update_pet_request_body.status)
    if pet is None:
        return {"message": "no pet with id=%s" % pet_id}, 404
    else:
        return pet, 200
```

TASK: RESTful API via Codegen

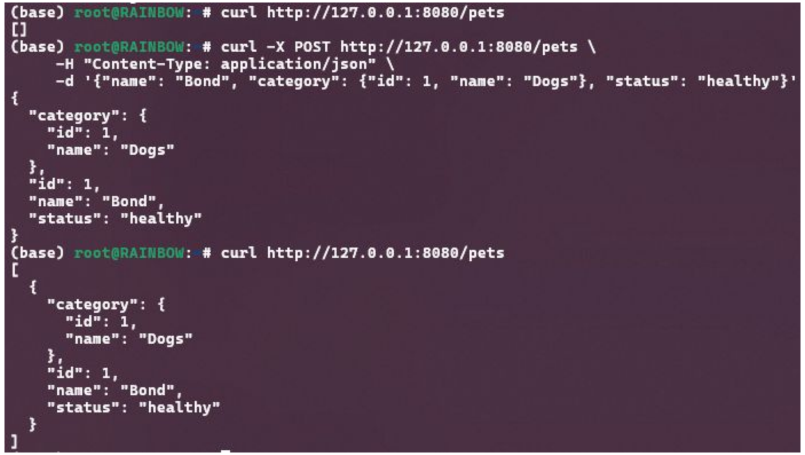
Generate Flask server code template via OpenAPI Generator.

> Reference codebase: `rest_codegen`

5. Configure Python dependencies:
   - `cd out/; python -m pip install -r requirements.txt`
6. Run the server as a Python module:
   - `cd out/; python -m openapi_server`
7. In another terminal, test the API with HTTP requests.



```
python -m openapi_server
 * Serving Flask app '__main__' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a p
roduction WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8080
 * Running on http://172.30.236.136:8080
Press CTRL+C to quit
```



```
(base) root@RAINBOW:~# curl http://127.0.0.1:8080/pets
[]
(base) root@RAINBOW:~# curl -X POST http://127.0.0.1:8080/pets \
     -H "Content-Type: application/json" \
     -d '{"name": "Bond", "category": {"id": 1, "name": "Dogs"}, "status": "healthy"}'
{
  "category": {
    "id": 1,
    "name": "Dogs"
  },
  "id": 1,
  "name": "Bond",
  "status": "healthy"
}
(base) root@RAINBOW:~# curl http://127.0.0.1:8080/pets
[
  {
    "category": {
      "id": 1,
      "name": "Dogs"
    },
    "id": 1,
    "name": "Bond",
    "status": "healthy"
  }
]
```

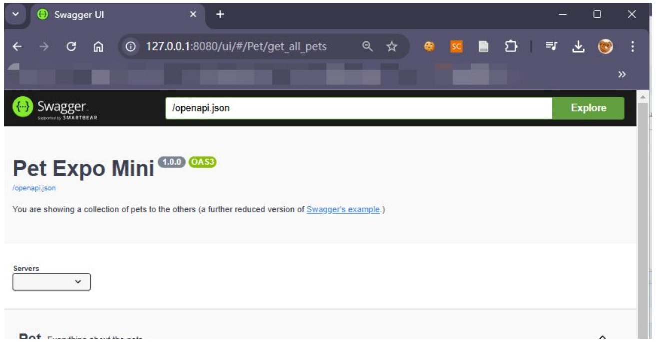
TASK: RESTful API via Codegen

This task explores all the aforementioned benefits.

Generate Flask server code template via OpenAPI Generator.

> Reference codebase: `rest_codegen`

8. The generator naturally supports Swagger UI. The UI web page is serving at the `<BASE_URL>/ui` endpoint. In this task, access http://127.0.0.1:8080/ui in the browser.
9. All test requests invoked from the UI page will be forwarded to the API server.

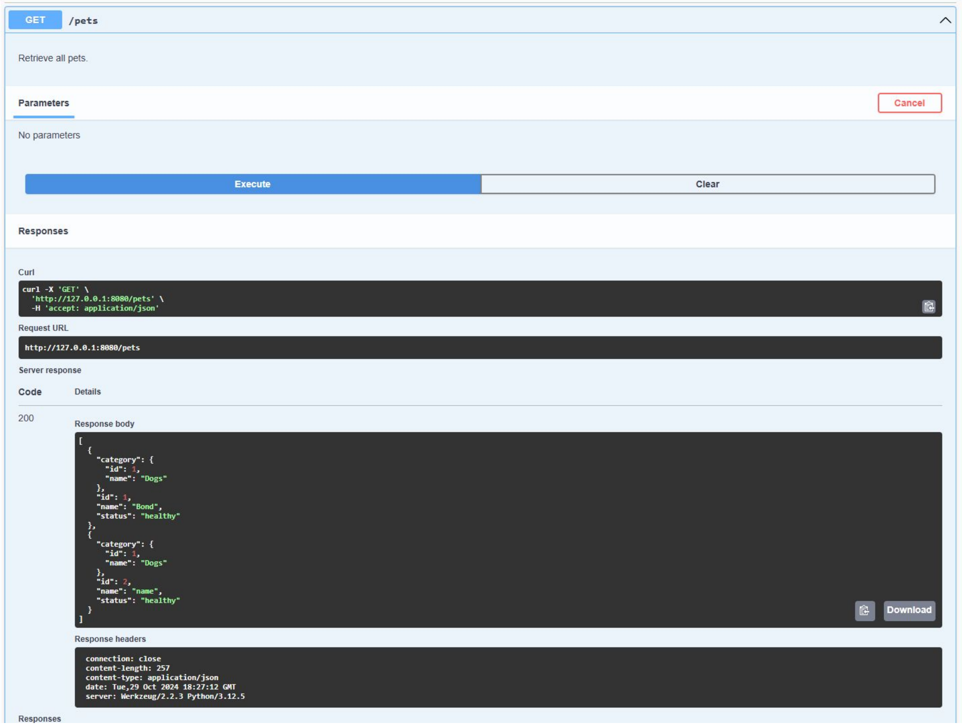


Pet Expo Mini 1.0.0 OAS3

/openapi.json

You are showing a collection of pets to the others (a further reduced version of Swagger's example.)

Servers



Retrieve all pets.

Parameters

No parameters

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8080/pets' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8080/pets
```

Server response

Code Details

200

Response body

```
[
  {
    "category": {
      "id": 1,
      "name": "Dogs"
    },
    "id": 1,
    "name": "Bond",
    "status": "healthy"
  },
  {
    "category": {
      "id": 1,
      "name": "Dogs"
    },
    "id": 2,
    "name": "name",
    "status": "healthy"
  }
]
```

Response headers

```
connection: close
content-length: 257
content-type: application/json
date: Tue,29 Oct 2024 18:27:12 GMT
server: Werkzeug/2.2.3 Python/3.12.5
```

Responses

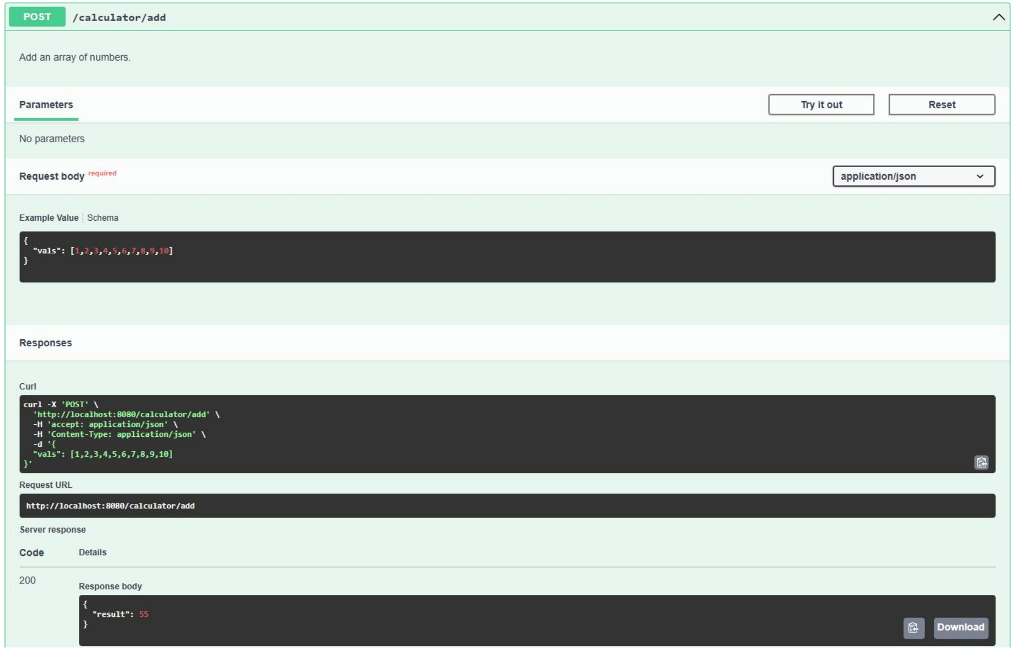
TASK: RESTful API - Codegen Yourself

Implement a RESTful API server yourself with OpenAPI Generator.
> Reference codebase: `rest_codegen_yourself`

Write an addition function to add an array of integers together and return the result. Define a RESTful API with the calculator tag to handle this functionality. Add this API to `hello_world.yaml`.

1. Modify API specification YAML file.
2. Generate via OpenAPI Generator.
3. Implement the generated controllers.
4. Test the new API.

Try starting from scratch to be familiar with the process before Assignment 2.



/calculator/add

Add an array of numbers.

Parameters

No parameters

Request body required

Example Value | Schema

```
{
  "vals": [1,2,3,4,5,6,7,8,9,10]
}
```

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8080/calculator/add' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "vals": [1,2,3,4,5,6,7,8,9,10]
}'
```

Request URL

```
http://localhost:8080/calculator/add
```

Server response

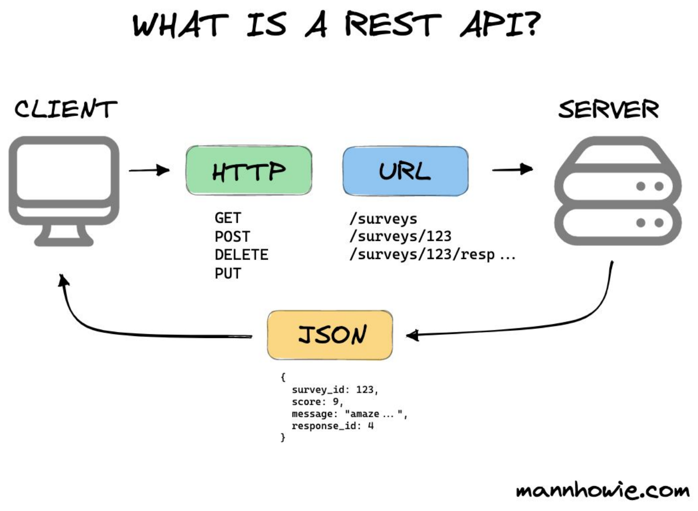
Code Details

200 Response body

```
{
  "result": 55
}
```

Summary

- **REST**
  a. Components: resource, representation, control data
  b. 6 Guiding Principles
     i. Client-Server
     ii. Stateless
     iii. Cacheable
     iv. Uniform Interface
     v. Layered System
     vi. Code on Demand (Optional)
- **RESTful API**
  a. Benefits: Scalability, Flexibility, Independence
  b. RESTful API *vs.* gRPC
- **OpenAPI**
  a. Components: metadata, API paths & operations, data object schemas
  b. Benefits: Standardization, Human-readable, UI, Codegen, Integrated Testing



```
{
  survey_id: 123,
  score: 9,
  message: "amaze ... ",
  response_id: 4
}
```

https://mannhowie.com/rest-api