Api theory

Q1. What is a RESTful API?

Ans. A RESTful API is a type of API that uses the principles of Representational State Transfer (REST) to communicate between different systems over the internet

Q2. Explain the concept of API specification?

Ans , An API (Application Programming Interface) specification is a blueprint that describes how an API behaves and interacts with other systems. It's essentially a formal document detailing the API's operations, endpoints, input/output formats, and data models, acting as a contract between the API provider and its users

An API specification provides a broad understanding of how an API behaves and how the API links with other APIs. It explains how the API functions and the results to expect when using the API

Q3. What is Flask, and why is it popular for building APIs?

Ans. Flask is a lightweight and flexible micro web framework for Python used to build web applications, APIs, and other web-based projects. It's known for its simplicity and ease of use, making it a good choice for beginners and small to medium-sized projects. Flask allows developers to focus on the application's core logic without being burdened by extensive boilerplate code

Q4.What is routing in Flask?

Ans Routing in Flask is the mechanism that maps specific URLs to Python functions. It allows web applications to respond to different user requests by associating each URL with the appropriate code to be executed.

Q5. How do you create a simple Flask application?

Ans. flask is a web application framework written in Python. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects.

Installation:

We will require two packages to set up your environment. virtualenv for a user to create multiple Python environments side-by-side. Thereby, it can avoid compatibility issues between the different versions of the libraries and the next will be Flask itself.

virtualenv

pip install virtualenv

Create Python virtual environment

virtualenv venv

Activate virtual environment

windows > venv\Scripts\activate

linux > source ./venv/bin/activate

Q6. What are HTTP methods used in RESTful APIs?

Ans. n RESTful APIs, the most commonly used HTTP methods are GET, POST, PUT, PATCH, and DELETE. These methods correspond to the CRUD (Create, Read, Update, Delete) operations, respectively.

Here's a breakdown of each method:

- GET:
  Retrieves data from the server. It's used to read resources or information about a resource.
- POST:
  Creates a new resource on the server. It's typically used to send data for creating new entries or performing actions.
- PUT:
  Updates an existing resource on the server. It replaces the entire resource with the data provided in the request.
- PATCH:
  Partially updates an existing resource. It applies specific changes without replacing the entire resource.
- DELETE:
  Removes a resource from the server.

Q7. What is the purpose of the @app.route() decorator in Flask?

Ans. oute('/') is the decorator that tells Flask to associate the function home() with the root URL ('/'). def home(): is the function that will be executed when a user accesses the root URL. return 'Welcome to the homepage! ' is the response that will be sent back to the user's browser.

Q8. What is the difference between GET and POST HTTP methods?

Ans. The main difference between GET and POST HTTP methods lies in how they handle data transmission. GET retrieves data, and data is included in the URL, while POST sends data to a server to create or update resources, and data is sent in the request bod

Key Differences Summarized:

Feature	GET	POST
Data Transmission	In URL (query parameters)	In request body
Purpose	Retrieving data	Sending data (create/update)
Idempotency	Idempotent (safe)	Non-idempotent (unsafe)
Caching	Often cached	Not usually cached
Security	Less secure (data in URL)	More secure (data in request body)
Data Limit	Limited by URL length	No limit (data in request body)

Q9. How do you handle errors in Flask APIs?

Ans. Error handling in Flask APIs is crucial for creating robust and user-friendly applications. Here's a breakdown of how to effectively manage errors:

1. Built-in Exceptions:

- Flask provides built-in exceptions like BadRequest, Unauthorized, NotFound, etc., that correspond to standard HTTP status codes (400, 401, 404, etc.).
- Raising these exceptions automatically generates appropriate error responses with JSON content and a "detail" key.
- Example: abort(404) will return a 404 error with a JSON response.

2. Custom Exceptions:

- You can define custom exception classes that inherit from Flask's HTTPException.
- This allows you to create specific error types that match your API's needs.

Q10. How do you connect Flask to a SQL database?

Ans. How to connect to SQL database using Python?

Steps to Connect SQL with Python involve:

- Install MySQL Database. Download and Install MySQL database in your system.
- Open Command Prompt and Navigate to the location of PIP. ...
- Test MySQL Connector. ...
- Create Connection.

Q11. M What is the role of Flask-SQLAlchemy?

Ans. Flask SQLAlchemy enables developers to perform tasks like defining models, creating queries, and easily managing database migrations and supports multiple database management systems such as SQLite, MySQL, and PostgreSQL.

Flask SQLAlchemy is a popular ORM tool tailored for Flask applications. It not only simplifies database interactions but also provides a robust platform to define data structures (models), execute queries, and manage database updates (migrations).

Q12. What are Flask blueprints, and how are they useful?

Ans. lask Blueprint is an object that works very similarly to a Flask application. They both can have resources, such as static files, templates, and views that are associated with routes. However, a Flask Blueprint is not actually an application. It needs to be registered in an application before you can run it.

Q13. What is the purpose of Flask's request object?

Ans. The Flask Request Object is used to perform both sending and receiving operations from a user's browser to the server and process the request data from the server. It should be imported from the flask module.

The Flask request object is a crucial component for handling incoming HTTP requests in a Flask web application. It provides access to all the data sent by the client, such as form data, URL parameters, headers, and files

Q14. How do you create a RESTful API endpoint using Flask?

Ans. from flask import Flask, jsonify, request

```
app = Flask(__name__)

@app.route('/items', methods=['GET'])
def get_items():
    items = [{"id": 1, "name": "Item 1"}, {"id": 2, "name": "Item 2"}]
    return jsonify(items)

@app.route('/items', methods=['POST'])
def create_item():
    data = request.get_json()
    new_item = {"id": 3, "name": data['name']}
    return jsonify(new_item), 201

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

Run this -

python app.py

Q15. What is the purpose of Flask's jsonify() function?

Ans. Flask's jsonify() function converts Python dictionaries or lists into JSON format and automatically sets the response's content type to application/json, making it ideal for creating REST APIs. This function simplifies returning JSON data in Flask routes, ensuring proper handling of JSON responses. parses the string and returns a Python dictionary

Q16. Explain Flask's url_for() function.

Ans. The url_for() function in Flask is used to generate URLs dynamically based on view functions. Instead of hardcoding URLs directly in templates or code, url_for() allows you to reference view functions by their name and Flask will generate the correct URL. This function is especially useful when you need to change the URL structure of your application, as you only need to modify the route definitions, not every place in your code where the URL is used

Q17. M How does Flask handle static files (CSS, JavaScript, etc.)?

Ans. Flask handles static files like CSS, JavaScript, and images by serving them from a designated directory, typically named "static," located in the same directory as your main application file.

Here's how it works:

- Static Folder:
  By default, Flask assumes that your static files are in a folder named static at the root of your application.
- URL Mapping:
  Flask automatically maps the /static URL path to this folder. For example, a file named styles.css in the static folder will be accessible at /static/styles.css

Q18. What is an API specification, and how does it help in building a Flask API?

Ans. Method 1: using only Flask

Here, there are two functions: One function to just return or print the data sent through GET or POST and another function to calculate the square of a number sent through GET request and print it.

```
# Using flask to make an api
# import necessary libraries and functions
from flask import Flask, jsonify, request

# creating a Flask app
app = Flask(__name__)

# on the terminal type: curl http://127.0.0.1:5000/
# returns hello world when we use GET.
# returns the data that we send when we use POST.
@app.route('/', methods = ['GET', 'POST'])
def home():
    if(request.method == 'GET'):

        data = "hello world"
        return jsonify({'data': data})

# A simple function to calculate the square of a number
# the number to be squared is sent in the URL when we use GET
# on the terminal type: curl http://127.0.0.1:5000 / home / 10
# this returns 100 (square of 10)
@app.route('/home/<int:num>', methods = ['GET'])
def disp(num):

    return jsonify({'data': num**2})

# driver function
if __name__ == '__main__':
```

app.run(debug = True)

Q19. What are HTTP status codes, and why are they important in a Flask API?

Ans. The whole point of these status codes is to give your users information about the request. 2xx means success, 4xx means the client did something wrong, and 5xx means the server did something wrong. In practice, the user should never see 5xx codes because this means the server has a bug

HTTP response status codes indicate whether a specific HTTP request has been successfully completed.

Q20. How do you handle POST requests in Flask?

Ans. Handling POST requests in Flask involves defining routes that accept POST methods and accessing the data sent within the request. Here's a breakdown:

1. Defining a route for POST requests:

- Use the @app.route() decorator and specify the methods argument to include POST.
- Example: @app.route('/submit', methods=['POST'])
-

2. Accessing form data:

- Use the request.form dictionary to access data submitted through HTML forms.
- Example: name = request.form['name'].
-

3. Accessing JSON data:

- If the request sends JSON data, use request.get_json() to parse it.
- Example: data = request.get_json(), then access values like data['key'].
-

4. Handling file uploads:

- Use request.files to access uploaded files.
- Example: file = request.files['file']. Use file.save() to store the file.

Q21. How would you secure a Flask API?

Ans. 1. Use HTTPS:

- Employ HTTPS to encrypt data in transit, preventing eavesdropping and man-in-the-middle attacks.

2. Authentication and Authorization:

- Token-based authentication: Use JSON Web Tokens (JWTs) for authentication. When a user logs in, generate a token and send it back to the user. The user should send the token with every API request.
- OAuth 2.0: Implement OAuth 2.0 for authorization, allowing users to grant limited access to their resources.
- API Keys: Use API keys for simple authentication, especially for applications that don't require user-specific authorization.
- Flask-Security: Use the Flask-Security extension to handle authentication and authorization.
- Custom Decorators: Create custom decorators to validate tokens and scopes.
- User Management: Handle password recovery, resetting, and username management.

3. Input Validation:

- Validate all user inputs to prevent injection attacks (e.g., SQL injection, cross-site scripting).

4. Rate Limiting:

- Implement rate limiting to protect against brute-force attacks and denial-of-service attacks.

5. Error Handling and Logging:

- Use proper error handling and logging to monitor and troubleshoot issues without exposing sensitive information.

6. Data Encryption:

- Encrypt sensitive data both in transit and at rest.

7. Security Audits:

- Conduct regular security audits and penetration testing to identify vulnerabilities.

8. Environment Variables:

- Store sensitive information like API keys and database credentials in environment variables instead of configuration files.

9. API Versioning:

- Implement proper API versioning and deprecation strategies.

10. Session Security:

- Configure session security settings to avoid common vulnerabilities.

11. CORS:

- Configure Cross-Origin Resource Sharing (CORS) headers to control which domains can access your API.

12. Dependencies:

- Keep all dependencies up to date with the latest security patches.

13. Content Security Policy (CSP):

- Use CSP headers to mitigate cross-site scripting attacks

Q22. What is the significance of the Flask-RESTful extension?

Ans. The Flask RESTful extension significantly simplifies the development of RESTful APIs within Flask applications. It provides a structured way to define resources, manage HTTP methods, and handle data serialization/deserialization, leading to cleaner, more maintainable code. By leveraging the Resource class, it enables developers to define HTTP methods as class methods, making it easier to organize and manage API endpoints.

Q23. What is the role of Flask’s session object?

Ans. In Flask, the session object allows developers to store and retrieve user-specific data across multiple HTTP requests. It's a mechanism to maintain state between requests, enabling features like login persistence and personalized experiences. Think of it as a way to "remember" information about a user as they navigate your web application.