Handout - Web Services with Python and FastAPI

1. Basics of Web Services

A **Web Service** is a way of enabling communication and data exchange between different software systems over the internet. Web services allow different applications to interact with each other, regardless of the platforms or programming languages they are built on. Web services generally use **HTTP** or **HTTPS** protocols, and data is often exchanged in **JSON** or **XML** formats.

RESTful Web Services (Representational State Transfer) have become popular due to their simplicity and ability to work seamlessly with HTTP, making them ideal for web and mobile applications. REST services operate using HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE`.

2. Why Use Web Services?

Web services provide a means to decouple systems, allowing them to be modular and scalable. They allow for easy integration between different systems and technologies. For example:

Use Case:

Imagine an e-commerce platform that needs to interact with third-party payment gateways. By using web services, the platform can send payment requests to a payment provider (like PayPal) and receive the result, all without needing to handle the payment processing itself.

Web services allow the payment gateway to expose a standard interface, which can be consumed by any platform, regardless of the technology it is built on.

3. Technologies Used to Develop a Web Service

To develop a web service, various technologies come into play:

- **HTTP/HTTPS**: These are the core protocols used for communication between client and server.
- **REST**: A software architectural style that uses HTTP to create, read, update, and delete data.
- **JSON and XML**: These are common data formats used for the exchange of information between client and server.
- **Python Frameworks**: FastAPI, Flask, and Django are some of the Python frameworks commonly used to build web services.

4. Introduction to FastAPI

What is FastAPI?

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints. FastAPI is built on **Starlette** for the web parts and **Pydantic** for the data validation.

Benefits of FastAPI:

- **Speed**: One of the fastest Python frameworks, it takes advantage of asynchronous programming.
- **Ease of Use**: FastAPI has a simple interface that makes it accessible for both beginners and experienced developers.
- **Automatic Documentation**: FastAPI automatically generates documentation with Swagger UI and ReDoc, making API development and testing easier.
- **Data Validation**: Uses Python's type hints and Pydantic for automatic request validation.

Installation:

To install FastAPI and the Uvicorn server (an ASGI server to run FastAPI apps), use the following command:

```
pip install fastapi uvicorn
```

5. Basic Codes in FastAPI

Below are basic examples of how to create endpoints using FastAPI.

a. Root Endpoint

This is the root endpoint, which returns a welcome message when accessed.

```
# main.py
from fastapi import FastAPI

# Create an instance of the FastAPI application
app = FastAPI()

# Define the root endpoint
@app.get("/")
def read_root():
    """
    Root endpoint:
    Returns a welcome message.
    """
    return {"message": "Welcome to my FastAPI Web Service!"}
```

Link to reach the Root Endpoint:

http://localhost:8000/

b. Endpoint without Parameters

This endpoint returns static information without needing any parameters.

```
# Endpoint without parameters
@app.get("/info")
def get_info():
```

```
    """
    Endpoint without parameters:
    Returns basic information about the API.
    """
    return {"info": "This is a FastAPI web service."}
```

Link to reach the endpoint:

`http://localhost:8000/info`

c. Endpoint with Integer and String Parameters

These endpoints demonstrate handling parameters, such as an integer (`item_id`) and a string (`username`), passed via the URL.

```
# Endpoint with integer parameter
@app.get("/items/{item_id}")
def read_item(item_id: int):
    """
    Endpoint with integer parameter:
    Returns details for a specific item.
    """
    return {"item_id": item_id}

# Endpoint with string parameter
@app.get("/users/{username}")
def read_user(username: str):
    """
    Endpoint with string parameter:
    Greets a user based on their username.
    """
    return {"message": f"Hello, {username}!"}
```

Links to reach the endpoints:

- Integer parameter: `http://localhost:8000/items/42`
- String parameter: `http://localhost:8000/users/johndoe`

Running the FastAPI Application

You can run the FastAPI application with the following command:

```
uvicorn main:app --reload
```

This will start the server on `localhost:8000`. The `--reload` option allows the server to automatically restart when changes are made to the code.

6. FastAPI Documentation (Swagger and ReDoc)

FastAPI automatically provides interactive documentation for your API, which is extremely useful for testing and debugging.

a. Swagger UI

Swagger UI is an interactive documentation tool that allows users to explore and test API endpoints from a web interface.

To access Swagger UI, visit:

`http://localhost:8000/docs`

Here, you can test each API endpoint, send parameters, and view responses directly from the browser.

b. ReDoc

ReDoc provides a structured and clean documentation for APIs, focusing more on a detailed specification rather than interactivity.

To access ReDoc, visit:

`http://localhost:8000/redoc`

ReDoc is especially useful when you need to understand the detailed API structure and specifications.

7. Difference between `GET`, `POST`, `PUT`, and `DELETE`

a. GET

GET is an HTTP method used to retrieve data from a server. It is the most common method for reading data, and the request contains no body. GET requests are considered safe, meaning they do not modify the state of the server. They are also idempotent, which means multiple GET requests should return the same response each time, as long as the resource does not change.

Use Case:

- Retrieving a list of items, details of a specific item, or reading any data from the server.

b. POST

POST is an HTTP method used to send data to the server, typically to create a new resource. The data is sent in the request body and is usually in JSON format. POST is not idempotent, meaning multiple POST requests could create multiple new resources with the same data.

Use Case:

- Creating a new user, adding a new item to a database, submitting a form, or sending any data that modifies the server state.

c. PUT

PUT is an HTTP method used to update or replace an existing resource on the server. Unlike POST, PUT is idempotent. If you send the same PUT request multiple times, the result will always be the same: the resource will be updated with the provided data.

Use Case:

- Updating the details of an existing user, replacing a file, or modifying an existing record.

d. DELETE

DELETE is an HTTP method used to remove a resource from the server. Like PUT, DELETE requests are idempotent: multiple DELETE requests for the same resource should return the same result.

Use Case:

- Deleting a user, removing an item, or deleting a resource from a database.

2. Example Codes for Each Method in FastAPI

Below are example codes for each HTTP method using the FastAPI framework.

a. Example for `GET` Request

This example demonstrates how to retrieve data from the server using a `GET` request. You can either get a list of items or a specific item by its ID.

```
# main.py
from fastapi import FastAPI

app = FastAPI()

# GET endpoint to retrieve all items
@app.get("/items")
def get_items():
    """
    GET endpoint to return a list of items.
    """
    return {"items": ["item1", "item2", "item3"]}

# GET endpoint to retrieve a specific item by ID
@app.get("/items/{item_id}")
def get_item(item_id: int):
    """
    GET endpoint to return a specific item by ID.
    """
    return {"item_id": item_id, "name": f"Item {item_id}"}
```

Link to access:

- Get all items: `http://localhost:8000/items`

- Get an item by ID: `http://localhost:8000/items/1`

b. Example for `POST` Request

This example demonstrates how to create a new item using a `POST` request. The client sends data (such as an item's name and description) in the request body, and the server creates the new item.

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

# Data model for creating a new item
class Item(BaseModel):
    name: str
    description: str

# POST endpoint to create a new item
@app.post("/items")
def create_item(item: Item):
    """
    POST endpoint to create a new item.
    """
    return {"message": f"Item '{item.name}' created with description
'{item.description}'."}
```

Link to access:

- Create a new item: Use a tool like **Postman** or **curl** to send a `POST` request to `http://localhost:8000/items` with a JSON payload like:

```
{
  "name": "New Item",
  "description": "This is a new item."
}
```

c. Example for `PUT` Request

This example demonstrates how to update an existing item using a `PUT` request. The client sends updated data, and the server updates the item.

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

# Data model for updating an item
class UpdateItem(BaseModel):
```

```
    name: str
    description: str

# PUT endpoint to update an item
@app.put("/items/{item_id}")
def update_item(item_id: int, item: UpdateItem):
    """
    PUT endpoint to update an existing item by ID.
    """
    return {"item_id": item_id, "updated_name": item.name, "updated_description":
item.description}
```

Link to access:

- Update an item: Use a tool like **Postman** or **curl** to send a `PUT` request to `http://localhost:8000/items/1` with a JSON payload like:

```
{
  "name": "Updated Item",
  "description": "This is an updated description."
}
```

d. Example for `DELETE` Request

This example demonstrates how to delete an item using a `DELETE` request. The client sends the ID of the item to be deleted, and the server removes it.

```
from fastapi import FastAPI

app = FastAPI()

# DELETE endpoint to remove an item by ID
@app.delete("/items/{item_id}")
def delete_item(item_id: int):
    """
    DELETE endpoint to remove an item by ID.
    """
    return {"message": f"Item with ID {item_id} has been deleted."}
```

Link to access:

- Delete an item: `http://localhost:8000/items/1`

Running the FastAPI Application

To run the FastAPI app, use the following command:

```
uvicorn main:app --reload
```

This will start the FastAPI server, and the endpoints will be available at `http://localhost:8000`.