Testing HTTP request methods on Localhost web Server

To test all HTTP request methods (`GET`, `POST`, `PUT`, `DELETE`, `PATCH`, `HEAD`, and `OPTIONS`) and their parameters in Python, we can set up a simple server on `localhost` along with a client. The server will respond to each type of request, allowing us to experiment with all methods and parameters from the client side.

Below are the instructions for creating both the server and client in Python, followed by detailed explanations of how to use each HTTP method and parameter.

Step 1: Setting up a Simple Server

We'll create a local server using Python's `Flask` framework. Flask is lightweight and easy to use, making it ideal for this task.

1. **Install Flask** (if not already installed):

   ```
   pip install Flask
   ```

2. **Create the Server Code** (`server.py`):

   ```
   from flask import Flask, request, jsonify

   app = Flask(__name__)

   # Sample data to simulate a resource on the server
   data_store = {
       "name": "Sample Item",
       "description": "This is a sample resource",
       "status": "active"
   }

   @app.route("/", methods=["GET"])
   def home():
       return jsonify({"message": "Server is running on localhost"}), 200

   @app.route("/resource", methods=["GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS",
   "HEAD"])
   def resource():
       if request.method == "GET":
           return jsonify(data_store), 200

       elif request.method == "POST":
           data = request.get_json()
           data_store.update(data)
           return jsonify({"message": "Resource created/updated with POST", "data":
   data_store}), 201

       elif request.method == "PUT":
           data = request.get_json()
   ```

```
        data_store.update(data)
        return jsonify({"message": "Resource replaced with PUT", "data": data_store}),
200

    elif request.method == "DELETE":
        data_store.clear()
        return jsonify({"message": "Resource deleted"}), 200

    elif request.method == "PATCH":
        data = request.get_json()
        data_store.update(data)
        return jsonify({"message": "Resource partially updated with PATCH", "data":
data_store}), 200

    elif request.method == "OPTIONS":
        return jsonify({"allow": ["GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS",
"HEAD"]}), 200

    elif request.method == "HEAD":
        return '', 200

if __name__ == "__main__":
    app.run(port=5000)
```

- **Explanation**:
  - The server has a single route, `/resource`, which handles all the HTTP methods.
  - Each method (`GET`, `POST`, etc.) performs an operation on the `data_store` dictionary, simulating CRUD operations.

3. **Run the Server**:
   - The server will start running on `http://127.0.0.1:5000`.

Step 2: Creating the Client Code

The client code will interact with the server using various HTTP request methods and parameters. Here is the client code to test each method with `requests`.

1. **Create the Client Code** (`client.py`):

```
import requests

url = "http://127.0.0.1:5000/resource"

# GET Request
def test_get():
    params = {"query": "test"}
    headers = {"User-Agent": "my-client"}
    response = requests.get(url, params=params, headers=headers)
    print("GET Response:", response.status_code, response.json())

# POST Request
def test_post():
```

```
    json_data = {"name": "New Item", "description": "Created with POST"}
    headers = {"Content-Type": "application/json"}
    response = requests.post(url, json=json_data, headers=headers)
    print("POST Response:", response.status_code, response.json())

# PUT Request
def test_put():
    json_data = {"name": "Updated Item", "description": "Replaced with PUT"}
    headers = {"Content-Type": "application/json"}
    response = requests.put(url, json=json_data, headers=headers)
    print("PUT Response:", response.status_code, response.json())

# DELETE Request
def test_delete():
    headers = {"User-Agent": "my-client"}
    response = requests.delete(url, headers=headers)
    print("DELETE Response:", response.status_code, response.json())

# PATCH Request
def test_patch():
    json_data = {"description": "Partially updated with PATCH"}
    headers = {"Content-Type": "application/json"}
    response = requests.patch(url, json=json_data, headers=headers)
    print("PATCH Response:", response.status_code, response.json())

# HEAD Request
def test_head():
    response = requests.head(url)
    print("HEAD Response:", response.status_code, response.headers)

# OPTIONS Request
def test_options():
    response = requests.options(url)
    print("OPTIONS Response:", response.status_code, response.json())

# Execute each test
test_get()
test_post()
test_put()
test_delete()
test_patch()
test_head()
test_options()
```

2. **Explanation of Each Method**:
    - **`GET`**: Retrieves data from the server.
        - Uses `params` to send query parameters in the URL.
        - Custom `headers` like `User-Agent` can be sent.
    - **`POST`**: Creates a new resource or updates an existing one.
        - Uses `json` to send JSON data, with `Content-Type` set to `application/json`.
    - **`PUT`**: Replaces the entire resource with new data.
        - Similar to `POST`, but intended for full replacements.
    - **`DELETE`**: Deletes the resource.
        - Can use headers for authentication or other purposes.

- **`PATCH`**: Partially updates a resource.
  - Sends only the fields to update.
- **`HEAD`**: Fetches headers without the body.
  - Useful for checking metadata like content length.
- **`OPTIONS`**: Returns allowed methods.
  - Helpful to see supported HTTP methods for the endpoint.

3. **Run the Client Code**:
   - This will execute each function and print the response details for every HTTP method.

Summary of HTTP Methods and Parameters

Method	Usage	Parameters
`GET`	Retrieve data without changing server state	`params`, `headers`, `cookies`, `timeout`, `allow_redirects`, `proxies`, `stream`, `verify`
`POST`	Send new data to create/update a resource	`data`, `json`, `headers`, `auth`, `cookies`, `files`, `timeout`, `allow_redirects`, `proxies`, `stream`, `verify`
`PUT`	Replace a resource completely	Similar to `POST` but used to fully replace data
`DELETE`	Delete a resource	`headers`, `auth`, `cookies`, `timeout`, `allow_redirects`, `proxies`, `stream`, `verify`
`PATCH`	Partially update a resource	Similar to `POST` but only for partial updates
`HEAD`	Fetch metadata without the body	Same parameters as `GET`
`OPTIONS`	Check allowed HTTP methods	Same parameters as `GET`

This setup provides a complete testing environment for all HTTP methods and parameters in Python's `requests` library.