MAGLAWAY, MARK S. ACTIVITY 7
BSIT 2101

Questions (Explain the Code)

1. What does the GET endpoint return, and in what format?

⇒ The GET endpoint at the /products route is solely for retrieving data. When a client sends an HTTP GET request, the get_products() function runs, accessing the product data stored in the Python list of dictionaries named items. This list holds the product catalog information (ID, name, price). The primary goal of this endpoint is to quickly and securely provide this product list to the client application that initiated the request.

The essential step is the call to return jsonify(items) on line 21. The Flask jsonify function is critical because it automatically converts the internal Python structure (the list of dictionaries) into a JSON string. JSON is the industry-standard, universal format for exchanging data across the web. This crucial conversion ensures the data is properly formatted and easily readable by any client application, regardless of its own underlying programming language.

The GET endpoint's final response is the entire product list delivered in JSON format. This text-based structure uses simple key-value pairs, making it highly efficient and easy to parse compared to sending raw Python objects. The client receives this structured JSON, allowing it to immediately access the product's details and display them to the end-user. Adhering to the JSON format ensures the API follows standard practices for reliable web services.

2. Why does the POST endpoint use request.get_json()?

⇒ The POST endpoint at /products is designed to receive and process new data, conceptually adding a product. Unlike GET requests, which pass data via URL query parameters, POST requests carry data entirely within the request body. This is essential for sending complex or large amounts of data, like a new product object, without exposing it in the URL. Therefore, when a client submits the product details, they are sent to the server encoded as a JSON string inside the HTTP message body.

The function request.get_json() is the specific tool Flask uses to handle this incoming data on line 26. This method is critical because it tells Flask exactly where to look for the data—the request body—and what format to expect—JSON. It first checks the request's Content-Type header, which must be set to application/json by the client to confirm the data type. This step ensures the integrity of the data stream before attempting to process it.

After confirming the format, request.get_json() performs the crucial final step: it parses the JSON string from the request body. Parsing converts the text-based JSON data into a usable Python dictionary (assigned to the data variable). This conversion is necessary because Python code operates on native data structures, not raw JSON strings. By converting the data, the server is able to easily access and manipulate the individual fields—like name and price—to complete the conceptual process of adding the new product.

3. What does the POST endpoint send back to the client?

⇒ The POST endpoint, executed by the add_product() function, is explicitly configured to return a JSON object to the client. This object serves as the server's immediate acknowledgment of the transaction. The first key-value pair included is a status message: "message": "Product added". This provides a clear, simple confirmation that the server successfully received, processed, and conceptually accepted the incoming product data, giving the client application immediate feedback on the operation's success.

The second critical component of the response is the inclusion of the received data itself: "product": data. The variable data holds the Python dictionary that was created when Flask parsed the client's incoming JSON payload. By sending this exact data structure back, the server performs an echo confirmation. This allows the client to verify, down to every detail, that the server correctly parsed the submitted fields (ID, name, price) and did not encounter any corruption or misinterpretation during transmission.

In essence, the entire response functions as a digital receipt for the client. The combination of the success message and the echoed data provides a high degree of confidence and reliability for the API interaction. Client applications can interpret this JSON response, coupled with a successful HTTP status code (typically 200), to confirm the validity of their submission and then proceed with subsequent actions, such as updating the user interface or internal state without needing to query the API again.

4. How would you conceptually test this API using Postman?

⇒ The initial step in testing this API involves verifying the GET endpoint at /products, which is designed for data retrieval. Using a tool like Postman, a tester would first select the GET HTTP method and send a request to the appropriate URL (http://<server-address>/products). Upon receiving a response, the tester must verify two crucial elements: the response status code should be 200 OK, indicating successful communication, and the response body must contain the full, expected list of products (Keyboard and Mouse) presented in a correctly structured JSON format.

The second major step is to test the POST endpoint, which handles adding new products. In Postman, the tester must select the POST method and target the same URL: http://<server-address>/products. The most important part of this test is preparing the data. The tester navigates to the Body tab, selects the raw option, and sets the data type dropdown to JSON. They then input a sample product object, such as {"id": 3, "name": "Tablet", "price": 1000}, ensuring it is valid JSON, which is essential for the server's request.get_json() function to work correctly.

After sending the prepared POST request, the tester verifies the server's response to confirm the successful addition of the product. Similar to the GET test, the response must have a 200 OK status code. Furthermore, the tester must check the response body itself. The body should be a JSON object that includes the confirmation message, "message": "Product added", and, crucially, it should also echo the product data that was just submitted. This confirms that the API route is correctly handling the incoming payload and successfully generating the expected receipt.