

Updates on Test Implementation for RESTful API Project

To create an aggregation server with consistency management and a RESTful API, several improvements have been made to the project testing. These improvements are aimed at broadening the scope of unit tests to ensure error handling and to test edge cases. Below is a summary of the new tests that have been added, as well as the existing tests that show the project's outputs under different conditions.

Existing Tests

1. PUT Test (PUTTest)

Purpose: To verify the functionality of the PUT request. The `PUTTest` class tests a PUT request by sending data from a specified file to a content server and then comparing the sent data with the data received back from the server. After starting the content server and waiting, it reads the data from both the file used in the request and the server's response file (`LatestWeatherData.json`). It then compares the two, ignoring whitespace, to check if they match. The test outputs whether the sent and received data are identical. This test also shows the integration of the client and server interacting with each other.

Expected Outcome: The server should respond appropriately, confirming the data has been received.

2. GET Test (GETTest)

Purpose: The `GETTest` class tests a GET request by sending multiple requests concurrently using four threads. It retrieves the server's JSON response, parses it, and compares it against expected data stored in the file (`LatestWeatherData.json`). If the response matches the expected data, the test passes, otherwise, it fails. The test outputs the expected and actual request onto the terminal to show whether it passed or failed.

Expected Outcome: The test passes if the correct data is returned as expected.

3. Data Expunge Test (DataExpunge)

Purpose: The DataExpunge class tests the functionality of expunging data after 30 seconds. It starts a ContentServer thread and waits 31 seconds after the server completes. Then, it initiates a GETClient to check if any data remains on the server. If the response is null, indicating the data has been successfully removed, the test passes. The test handles exceptions and prints the result of the expunging process, verifying whether the data was successfully cleared after the wait time.

Expected Outcome: The test passes if expired data is no longer accessible after the designated time.

4. Lamport Clock Test (LamportClockTest)

Purpose: This test checks that each component correctly updates and synchronises its clock.

Expected Outcome: The `LamportClockTest`` class tests Lamport timestamp functionality by starting a `ContentServer`` to handle requests and manage Lamport clocks. After the server finishes processing, it starts a `GETClient`` to send PUT and GET requests. The test waits for both the server and client to complete, handling any interruptions that occur. This test checks whether the Lamport clock system is properly integrated and functioning during client-server interactions.

5. **Response Tests** (e.g., `Response_201`, `Response_400`, `Response_500`, `Response_204`)

Purpose: To verify the correct handling of various HTTP response codes. Each test simulates different scenarios such as creating a resource and handling bad requests and server errors to ensure proper responses are generated.

Expected Outcome: Each test should return the expected HTTP status code and associated messages.

New Tests Implemented

1. **TestInvalidJSONParsing**

Purpose: The `TestInvalidJSONParsing`` class tests the handling of invalid JSON input. It tries to parse an invalid JSON string (missing a closing brace). If parsing succeeds, the test fails because an exception was expected. If a `JSONException`` is caught, the test passes, confirming that invalid JSON input is correctly detected. The error message and the invalid JSON are printed when the exception is caught, indicating the test succeeded in detecting the malformed data. This test checks whether the application correctly throws a `JSONException` when given improperly formatted JSON (missing closing brace in this test specifically).

Expected Outcome: The test passes when the expected exception is thrown, confirming that the application handles invalid input gracefully.

2. **TestMissingStationID**

Purpose: To ensure that the application detects the absence of required fields in the JSON input. The `TestMissingStationID`` class tests whether a JSON input correctly handles the absence of the "stationId" variable. It parses a JSON string that is missing the "stationId" field and checks if the key is absent. If not present the test passes, confirming the expected behaviour. If the key is found or an unexpected exception occurs, the test fails, and an error message is printed.

Expected Outcome: The test should pass if the application identifies the missing field.

3. **TestNoUserAgent**

Purpose: To evaluate the server's response to requests that lack a User-Agent header. This test sends a GET request without the User-Agent and expects a 400 Bad Request response from the server. The `TestNoUserAgent`` class tests a GET request made without a "User-Agent" header to check if the server correctly responds with a `400 Bad Request`` error. It sends a request to a local server and evaluates the response code. If the server returns a `400`` error, the test passes, confirming that the missing "User-

Agent" caused the rejection. If any other response code is received or an exception occurs, the test fails, indicating unexpected behaviour.

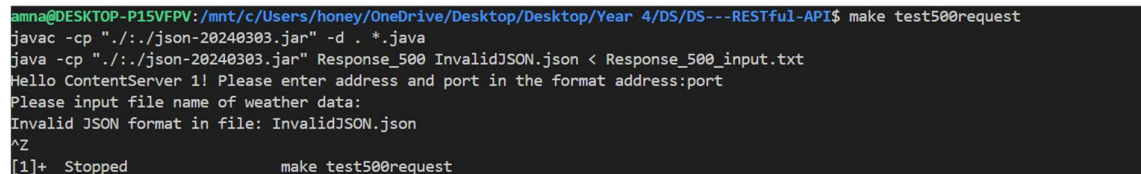
Expected Outcome: The test is successful if the server correctly returns the expected error code, indicating that the User-Agent header is required for processing requests.

Further testing/ Enhancements

1. test500request

Issue Analysis: The test500request is designed to simulate a scenario where the server receives invalid JSON input. The output indicates that the server requests an address and a port but does not effectively handle invalid JSON. The program subsequently becomes unresponsive, suggesting it may be waiting for further input or is caught in a loop waiting for the user to provide valid data (Refer to Figure 1).

The issue with the test500request is that it fails due to a JSON parsing error in the input file (InvalidJSON.json), preventing the test from reaching the server and triggering the intended 500 error response. The program outputs an "Invalid JSON format" error because the file is parsed early, and this error stops the execution before the server can process the request. To address this, the code should be modified, allowing the request to be sent to the server regardless of the format issue. This will ensure that the test properly assesses server-side behaviour for a 500 error. The edit can be made in the RequestHandler module where the run() method can be modified by changing how the invalid JSON is processed and ensuring that the response is sent back to the client, regardless of the validity of the JSON data i.e. the reading response is not stopped. This change was not implemented as it ended up giving unexpected results and affected the functionality of other methods.



```
amna@DESKTOP-P15VFPV: /mnt/c/Users/honey/OneDrive/Desktop/Desktop/Year 4/DS/DS---RESTful-API$ make test500request
javac -cp "./../json-20240303.jar" -d . *.java
java -cp "./../json-20240303.jar" Response_500 InvalidJSON.json < Response_500_input.txt
Hello ContentServer ! Please enter address and port in the format address:port
Please input file name of weather data:
Invalid JSON format in file: InvalidJSON.json
^Z
[1]+  Stopped                  make test500request
```

Figure 1 Test for response 500 hangs waiting for input.

2. test204request

Issue Analysis: The test expected a 204 No Content response, however, the output showed a 200 OK response. This indicates that while the server processed the request correctly, it returned an incorrect HTTP status code (refer to Figure 2). Upon further investigation, it appears that this can be fixed by making a few enhancements in the ProducerConsumer module.

First, the put method should implement logic to return a 204 No Content status when there is no content to update in LatestWeatherData.json, requiring a check for empty data before any file operations. Additionally, the request handling must consistently validate essential values such as User-Agent and Lamport-Timestamp to prevent null pointer exceptions and ensure accurate response codes for client requests. The requestJSONgenerator method should be updated to properly handle the 204 No Content response without including a content length or

body following HTTP specifications. This change was not implemented as it ended up giving unexpected results and affected the functionality of other methods.

```
amna@DESKTOP-P15VFPV:/mnt/c/Users/honey/OneDrive/Desktop/Desktop/Year 4/DS/DS---RESTful-API$ make test204request
javac -cp "./json-20240303.jar" -d . *.java
java -cp "./json-20240303.jar" Response_204 < Response_204_input.txt
Hello ContentServer 1! Please enter address and port in the format address:port
Please input file name of weather data:
Server response for ContentServer 1 : [HTTP/1.1 200 OK, Lamport-Timestamp: 0, Content-Length: 49, Content-Type: application/json, , {'message': 'Successfully updated weather file!'}]

Request successful. Would you like to send another PUT request? ('true' for yes, 'false' for no)
Goodbye ContentServer 1!
Empty data in put request works? false
```

Figure 2 Failed 204 Response test