***API*** ***Assignment Questions***

***Find the solutions in API Assignment>src>main>java***

1. **Explain the following:**
2. **PUT and PATCH methods**

The PUT and PATCH methods are used in web development to update data on a server, but they work in different ways.

1. PUT: This method completely replaces the entire resource (data). Imagine it like swapping out an entire document. For example, if you're updating a user's profile and send the data `{ "name": "John", "age": 30 }`, the server will replace the existing user information with this new data. If some fields are missing, they might be deleted or set to default values. It’s like saying, "Here’s the full new version."

2. PATCH: This method updates only specific parts of a resource. It’s like editing just a few sentences in a document without changing the rest. For example, if you send `{ "age": 31 }`, only the "age" field will be updated, and everything else will remain unchanged. It’s like saying, "Make only these changes."

### Main Differences:

- PUT: Replaces the entire resource.

- PATCH: Updates only the specified parts.

Use PUT when you need to replace everything.

1. **Headers and Cookies**

Headers are like the instructions attached to a letter. They provide extra details about your request (when you're asking for a webpage) or the server's response (when the website sends the page back). For example:

* Your browser might send headers saying, "I'm using Chrome" or "I understand English."
* The server might reply with headers like, "Here's the webpage in English" or "The content is safe to display."

Headers are temporary and only exist during that specific interaction between your browser and the server.

Cookies are like little notes that a website asks your browser to keep for later. They store information to personalize your experience. For example:

* A website might save a cookie saying, "This user is logged in as Alice."
* The next time you visit, your browser sends that cookie back to the website, so it remembers you.

Cookies stick around until they expire or are deleted, helping websites "remember" you over time, unlike headers that are just for the moment. Together, headers and cookies make your web experience smoother and more personal.

1. **Endpoint and Base URL**

An Endpoint and Base URL are fundamental concepts in APIs (Application Programming Interfaces), which let different systems talk to each other.

Base URL:

Think of the Base URL as the "home address" of a website or service. It’s the common part of a web address that stays the same across all requests to that service. For example, in https://api.weather.com, the https://api.weather.com is the Base URL. It tells your computer where to look for the API.

Endpoint:

An Endpoint is like a specific "room" in that "home" where you can get or send information. It’s an addition to the Base URL that specifies what you want. For example, https://api.weather.com/current is an Endpoint that might provide current weather data. Here, /current is the Endpoint.

Methods:

Methods are the "action" you want to take at an Endpoint. The most common ones are:

* GET: Fetch data (e.g., get the current weather).
* POST: Send data (e.g., submit a form).
* PUT: Update data (e.g., change your profile info).
* DELETE: Remove data (e.g., delete an account).

1. **Query Parameters and Path Parameters**

Path Parameters:

* These are part of the URL itself and usually identify a specific resource.
* Example: https://api.example.com/users/123  
  Here, 123 is a Path Parameter, often used to identify a specific user.

Think of it as giving directions to a specific "room" in a building.

Query Parameters:

* These are added at the end of a URL after a ? and are used to filter or customize the request.
* Example: https://api.example.com/users?age=25&sort=asc  
  Here, age=25 and sort=asc are Query Parameters, used to refine the search.

It’s like specifying the details of what you want when you get to the room (e.g., "Show me people aged 25 and sort them in ascending order").

In Short:

* Path Parameters: Point to "where" (specific resource).
* Query Parameters: Define "what" or "how" (filters, settings).

1. **What are error codes? Explain all series of error codes.**

Error codes are standardized messages used in web and API communication to indicate the success or failure of a request. They’re part of the HTTP protocol and are grouped into five categories, each represented by a series of numbers. Here’s an overview of these series:

1xx: Informational Responses

* These codes indicate that a request was received and is still being processed.
* Example:
  + 100 Continue: The server has received part of the request and asks the client to continue sending the rest.
  + 101 Switching Protocols: The server agrees to switch protocols as requested by the client.

2xx: Success

* These mean the request was successfully received, understood, and processed.
* Example:
  + 200 OK: The request succeeded, and the server is returning the expected data.
  + 201 Created: A resource was successfully created.
  + 204 No Content: The request was successful, but there’s no content to return.

3xx: Redirection

* These codes mean the client needs to take additional actions to complete the request.
* Example:
  + 301 Moved Permanently: The requested resource has a new permanent URL.
  + 302 Found: The resource is temporarily located at a different URL.
  + 304 Not Modified: The resource hasn’t changed since the last request.

4xx: Client Errors

* These indicate issues with the client’s request.
* Example:
  + 400 Bad Request: The server couldn’t understand the request due to invalid syntax.
  + 401 Unauthorized: Authentication is required but missing or incorrect.
  + 403 Forbidden: The client is not allowed to access the resource.
  + 404 Not Found: The server couldn’t find the requested resource.

5xx: Server Errors

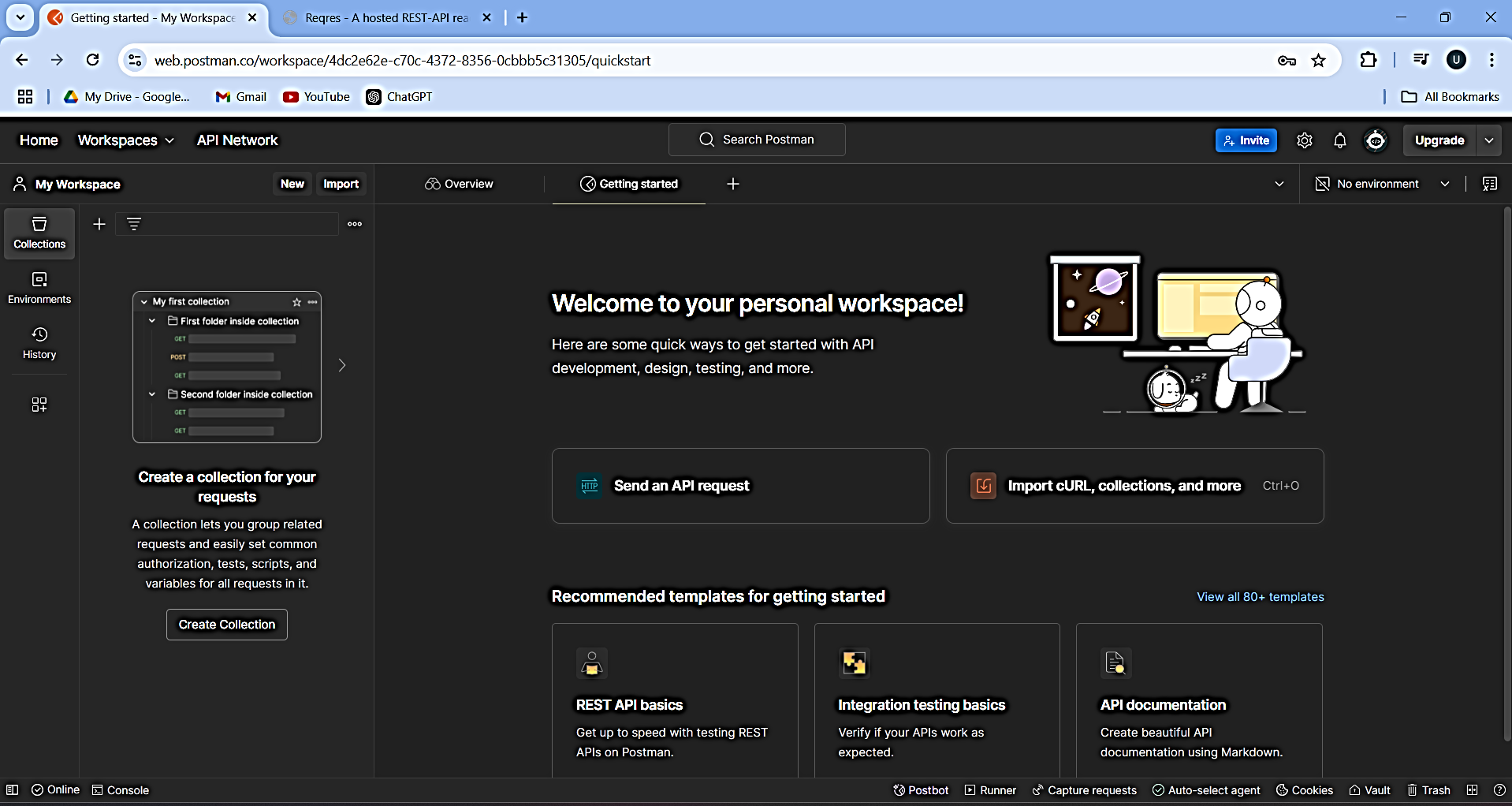
* These indicate the server failed to fulfill a valid request.
* Example:
  + 500 Internal Server Error: The server encountered an unexpected issue.
  + 502 Bad Gateway: The server received an invalid response from an upstream server.
  + 503 Service Unavailable: The server is temporarily unavailable, often due to maintenance.

1. **Create a new collection in Postman named "Sample APIs." Use https://reqres.in/ and implement GET, POST, PUT, PATCH, and DELETE operations.**

**Step 1: Install Postman**

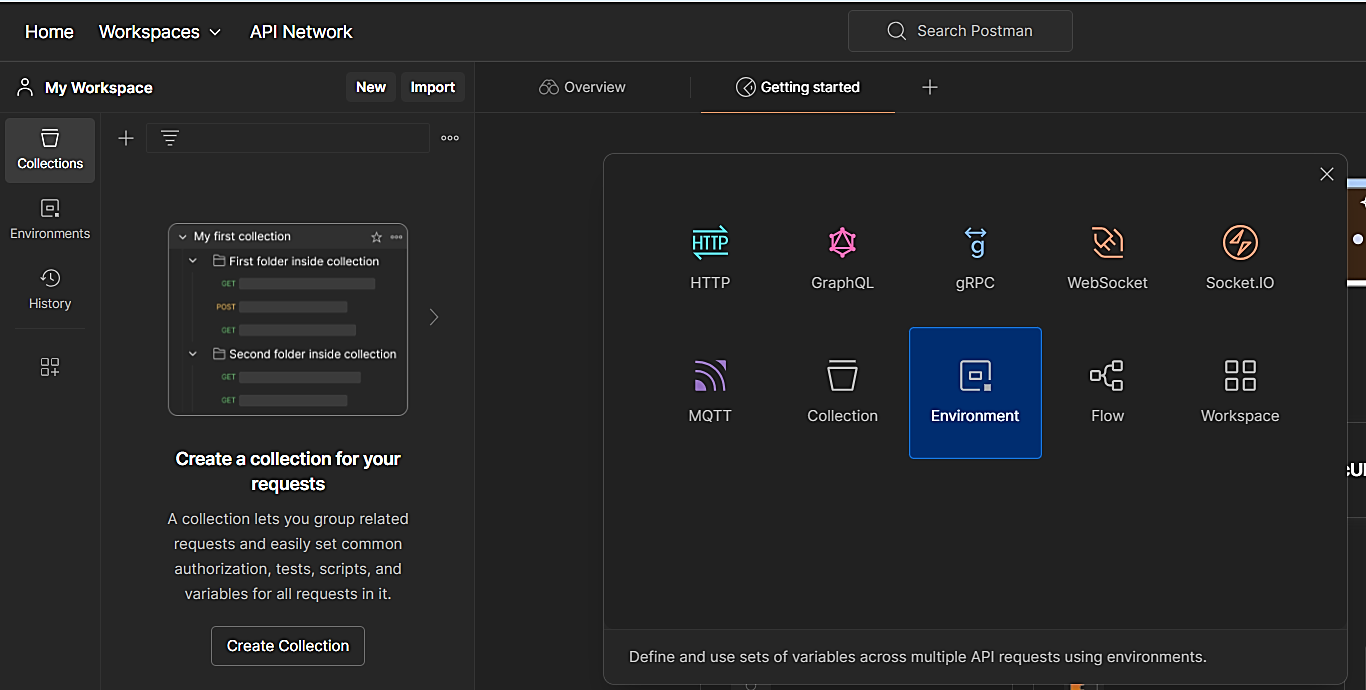
If you haven’t already installed Postman:

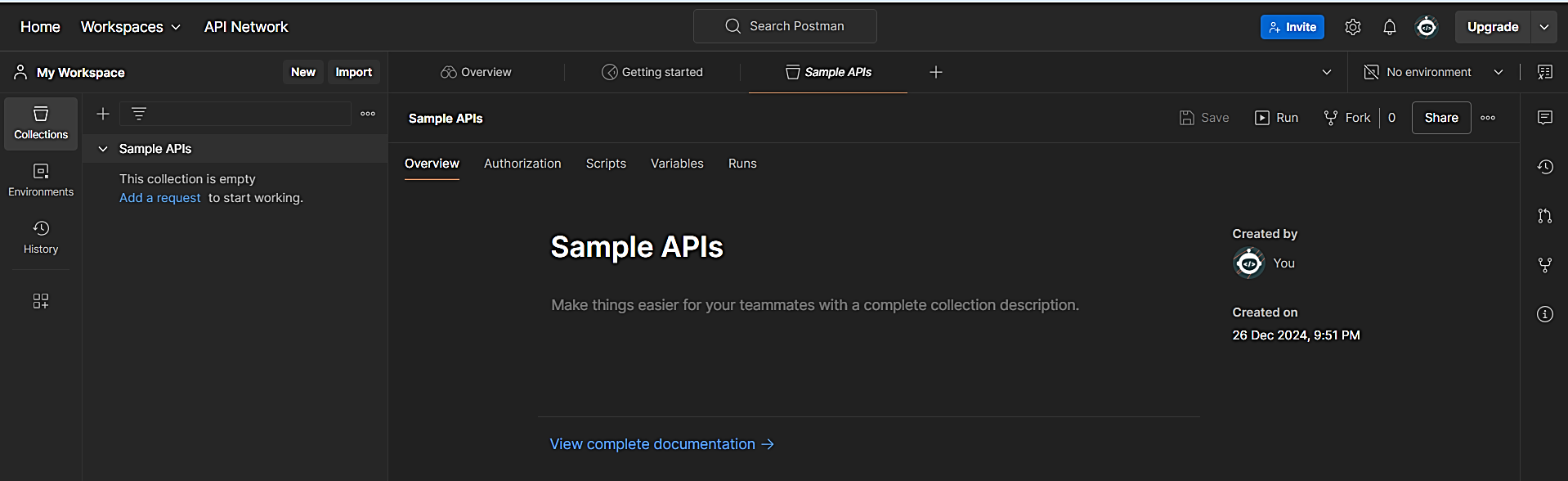
1. Download it from Postman Website.
2. Install and open the application.

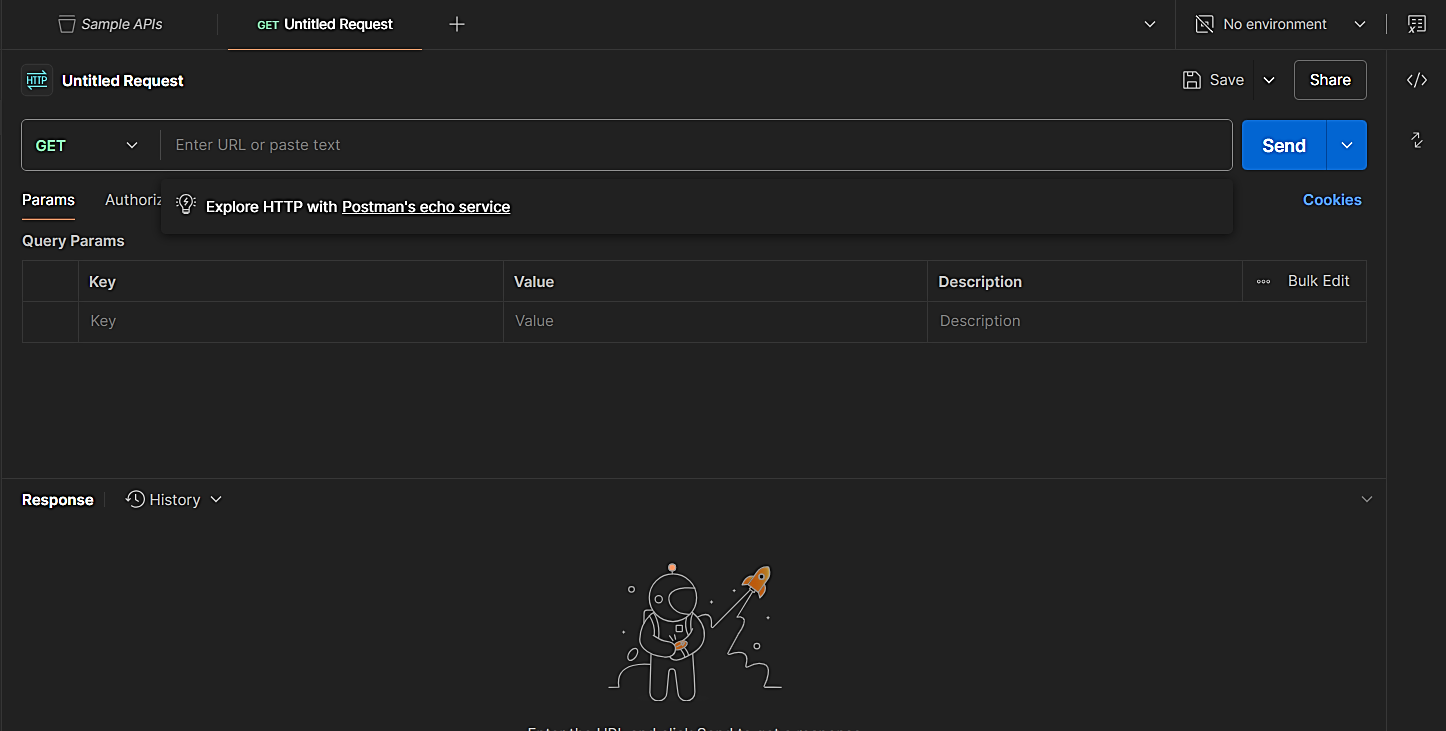


**Step 2: Create a New Collection**

1. Open Postman.
2. Click the Collections tab in the left sidebar.
3. Click the + New Collection button.
4. Name your collection "Sample APIs".
5. Optionally, add a description and save it.







**Step 3:**

Here, we’ll use the website <https://reqres.in> to try different types of operations like **GET**, **POST**, **PUT**, **PATCH**, and **DELETE**.

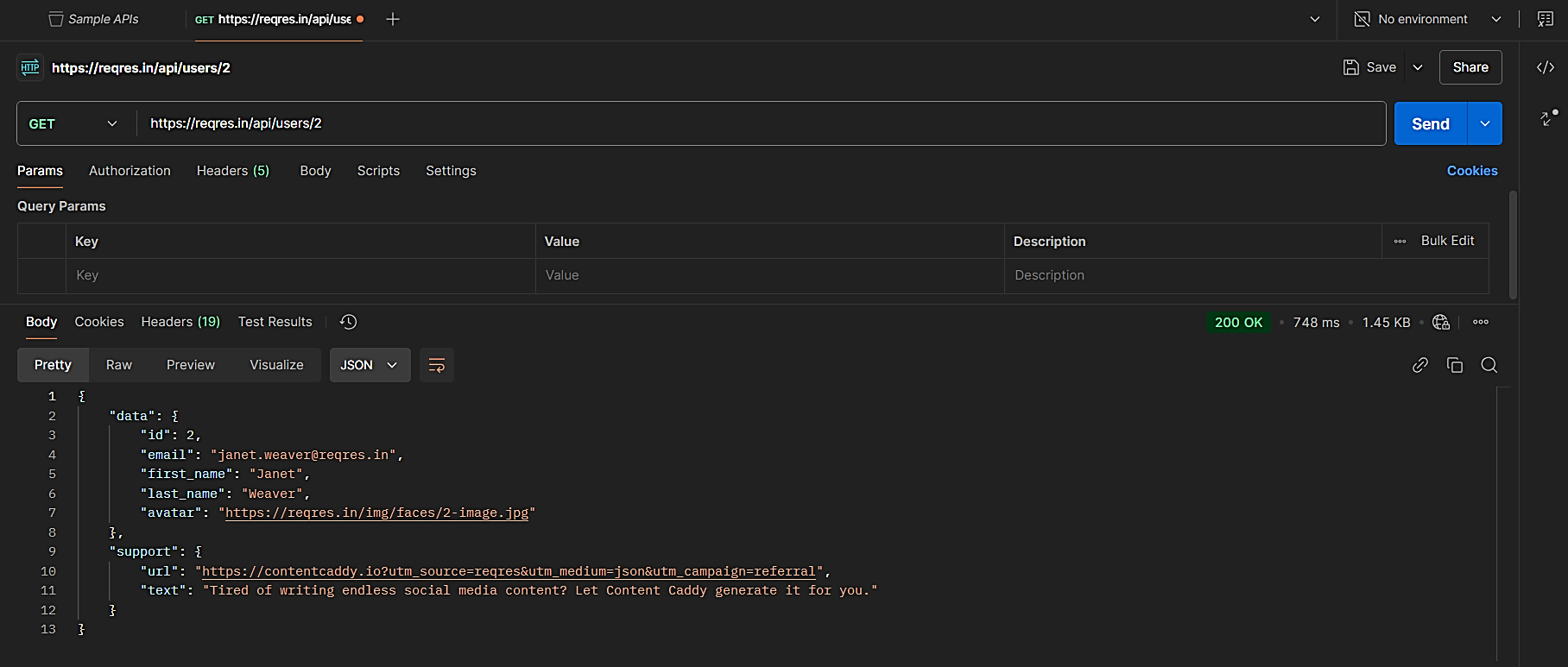
**A. GET (Get Information)**

We’ll get information about a user:

1. Click the **+** button at the top to open a new request tab.
2. Select **GET** from the dropdown (it’s already set by default).
3. In the URL box, type:

https://reqres.in/api/users/2

1. Click **Send**.
2. You’ll see details about a user (like their name and email) in the response box below.



**B. POST (Add New Data)**

We’ll create a new user:

1. Open another new tab (by clicking **+**).
2. Select **POST** from the dropdown.
3. Enter this URL:

https://reqres.in/api/users

1. Click on the **Body** tab below the URL box, then select **raw** and choose **JSON**.
2. Paste this:

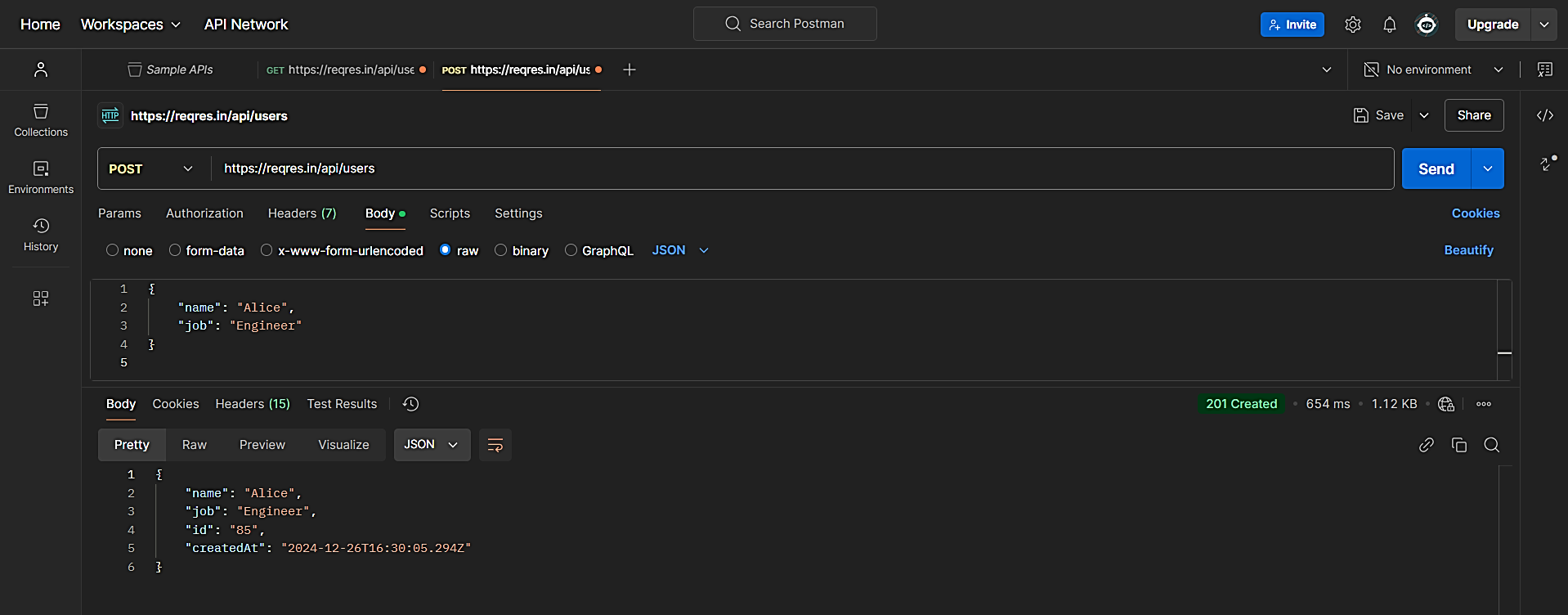
{

"name": "Alice",

"job": "Engineer"

}

1. Click **Send**.
2. You’ll see a response saying the user was created with an ID and a timestamp.



**C. PUT (Update Everything)**

Let’s update all details of a user:

1. Open a new tab and choose **PUT**.
2. Enter this URL:

https://reqres.in/api/users/2

1. In the **Body** tab, paste this:

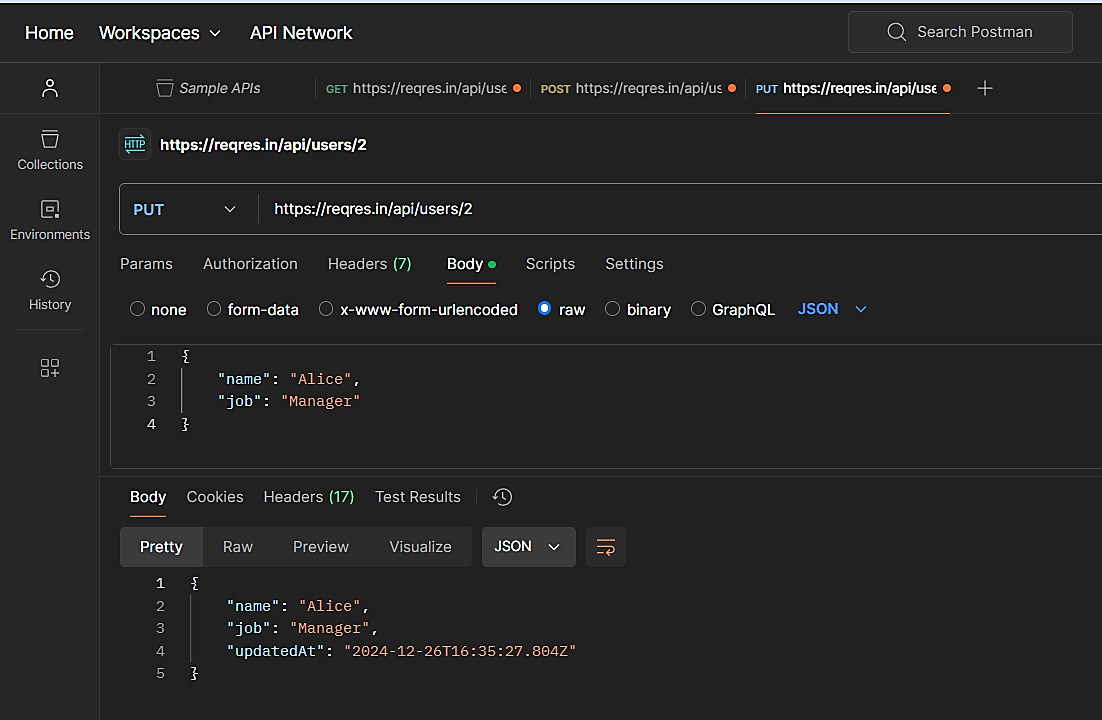
{

"name": "Alice",

"job": "Manager"

}

1. Click **Send**.
2. You’ll see the updated information.



**D. PATCH (Update Some Details)**

Let’s update just one detail:

1. Open a new tab and choose **PATCH**.
2. Enter this URL:

https://reqres.in/api/users/2

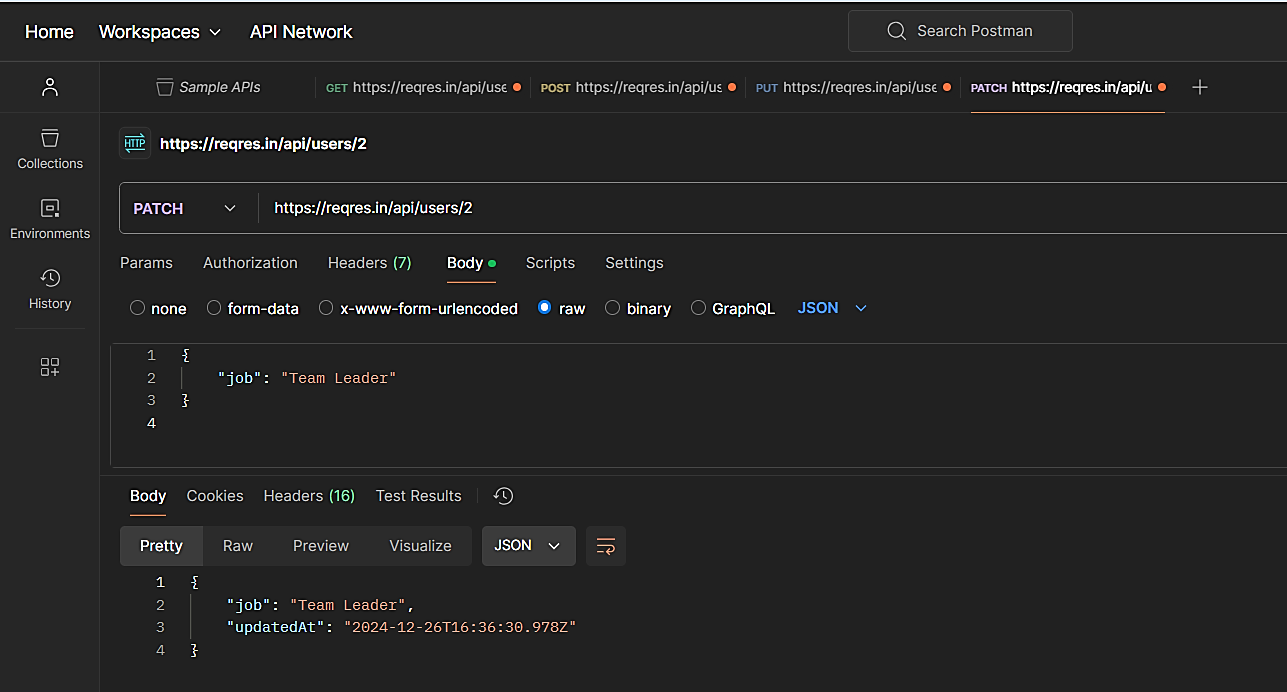
1. In the **Body** tab, paste this:

{

"job": "Team Leader"

}

1. Click **Send**.
2. You’ll see that only the job has been updated.



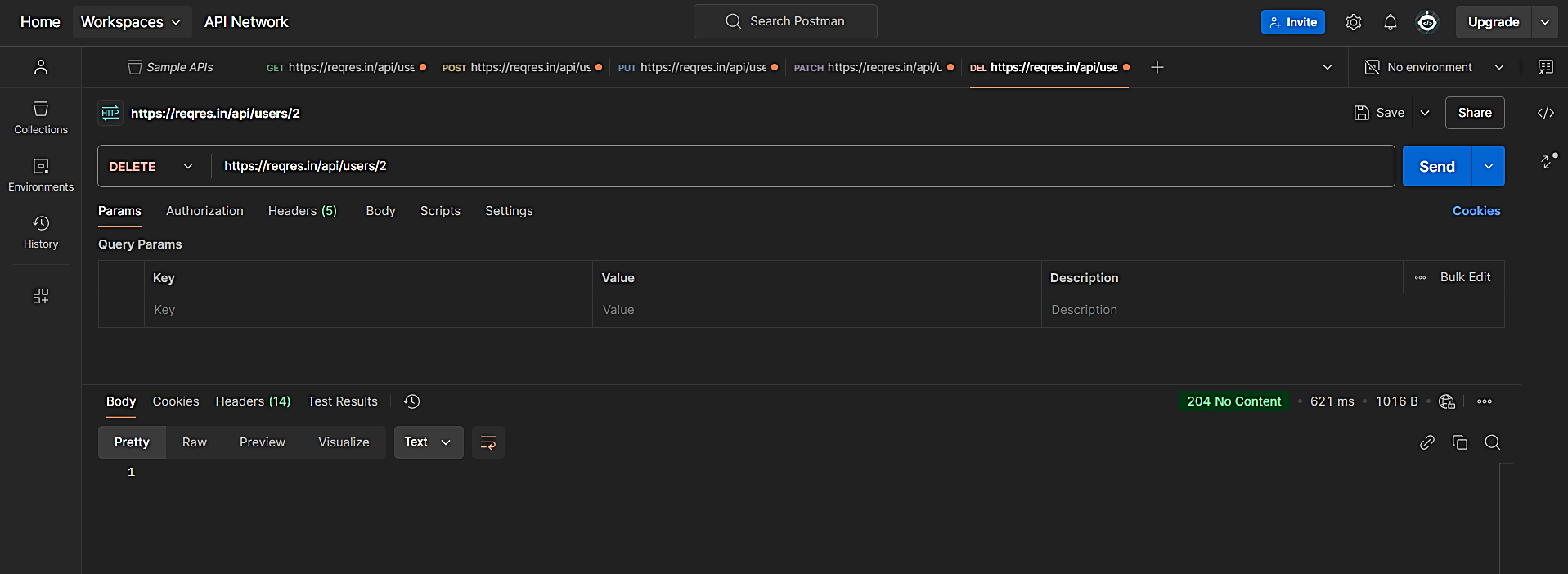
**E. DELETE (Remove Data)**

Let’s delete the user:

1. Open a new tab and choose **DELETE**.
2. Enter this URL:

https://reqres.in/api/users/2

1. Click **Send**.
2. You won’t see much in the response, but it means the user is gone.



**Step 4: Save Everything in Your Collection**

1. For each request, click **Save** and choose the **Sample APIs** collection.
2. Now all your work is neatly saved, and you can revisit it anytime.
3. **What is the difference between** **given(),** **when(), and** **then() in Rest Assured?**

In Rest Assured, a Java library for API testing, given(), when(), and then() are part of a fluent interface used to create readable and structured tests. Here's what each does, in a professional yet simple way:

1. given(): Setting Up the Request

This is where you define the setup for your API request. It includes details like headers, query parameters, request body, authentication, or any preconditions.

Example:  
 given().header("Content-Type", "application/json")

       .body("{\"name\":\"John\"}");

* Think of this step as "What do I need to send or configure for the request?"

2. when(): Making the Request

This is where you specify the HTTP method (GET, POST, PUT, DELETE, etc.) and the endpoint to send the request. It executes the actual API call.

Example:  
 when().post("/users");

* Think of this step as "What action am I performing?"

3. then(): Validating the Response

This is used to define the assertions or verifications for the response. You check status codes, response headers, or body content.

Example:  
 then().statusCode(201).body("id", notNullValue());

* Think of this step as "What do I expect in the response?"

Putting It Together:

A full example combines all three:

given().header("Content-Type", "application/json")

       .body("{\"name\":\"John\"}")

.when().post("/users")

.then().statusCode(201)

       .body("id", notNullValue());

* given(): Prepares the request.
* when(): Sends the request.
* then(): Validates the response.

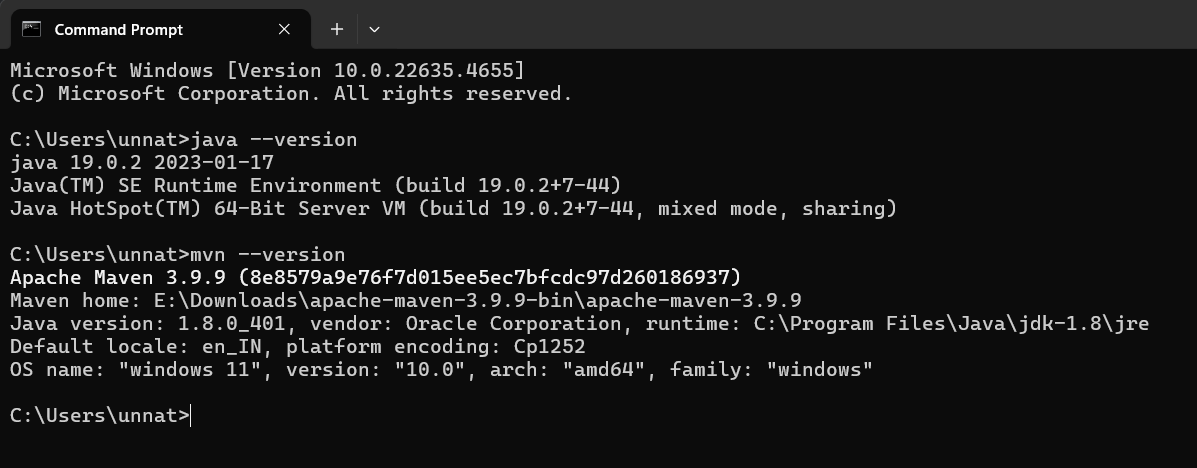
This approach ensures clear and readable tests, mimicking a natural language workflow.

1. **Perform the operations with Rest Assured (use** [**https://reqres.in/**](https://reqres.in/)**):**
2. **Create GET, POST, PUT, PATCH, and DELETE**

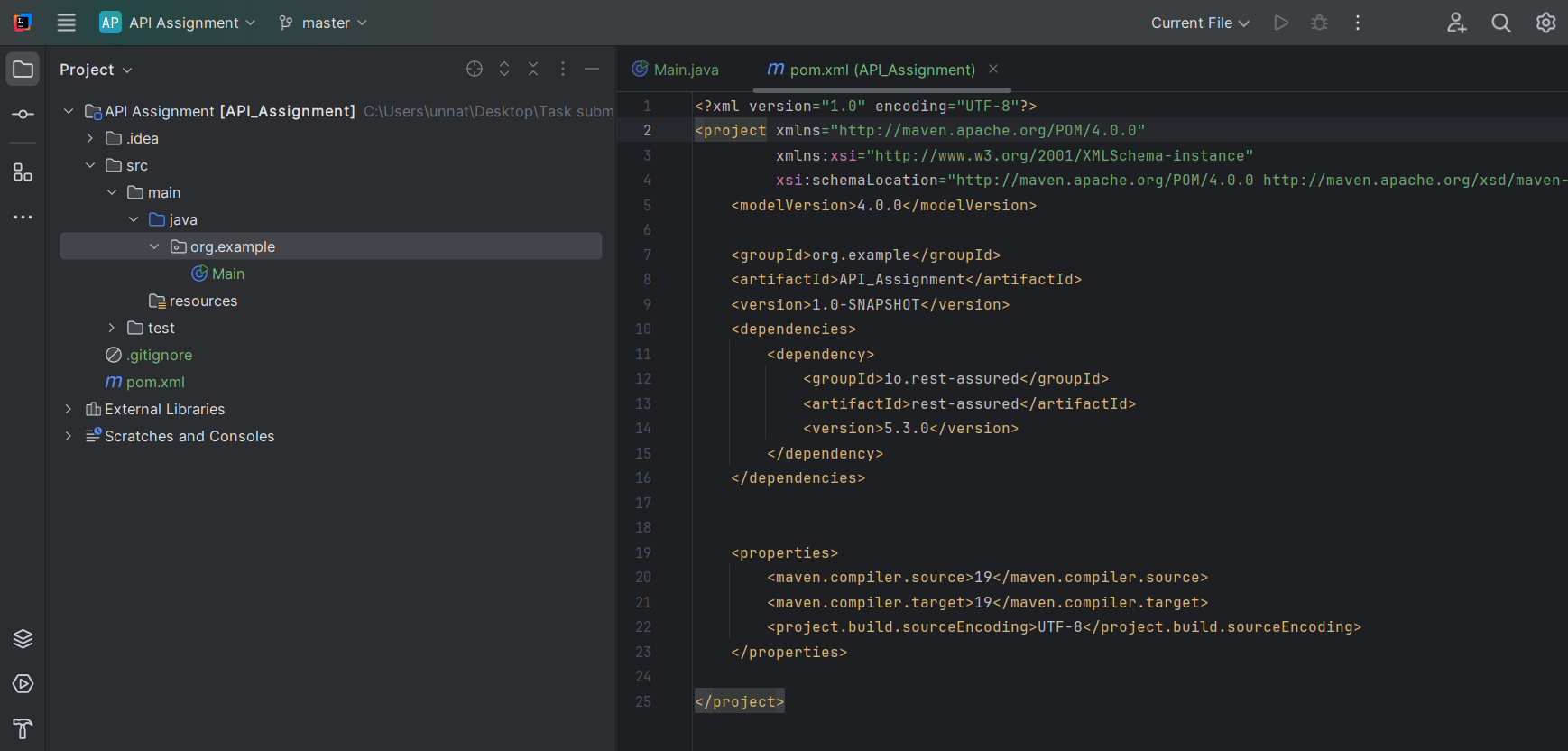
**Using Rest Assured**

**Prerequisites**

1. **Install Java Development Kit (JDK)**: Make sure you have JDK installed (at least Java 8).
2. **Set up a Java IDE**: Use an IDE like **IntelliJ IDEA** or **Eclipse**.



1. **Add Rest Assured Dependency**: Use **Maven** to include Rest Assured in your project by adding this to your pom.xml:



<dependency>

<groupId>io.rest-assured</groupId>

<artifactId>rest-assured</artifactId>

<version>5.3.0</version>

</dependency>

**Task A: Perform GET, POST, PUT, PATCH, and DELETE**

Here's how you can perform these operations with Rest Assured using the https://reqres.in/ API:

**1. GET Request**

import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.\*;

public class GetRequestExample {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// GET Request

Response response = given()

.when()

.get("/users?page=2")

.then()

.statusCode(200)

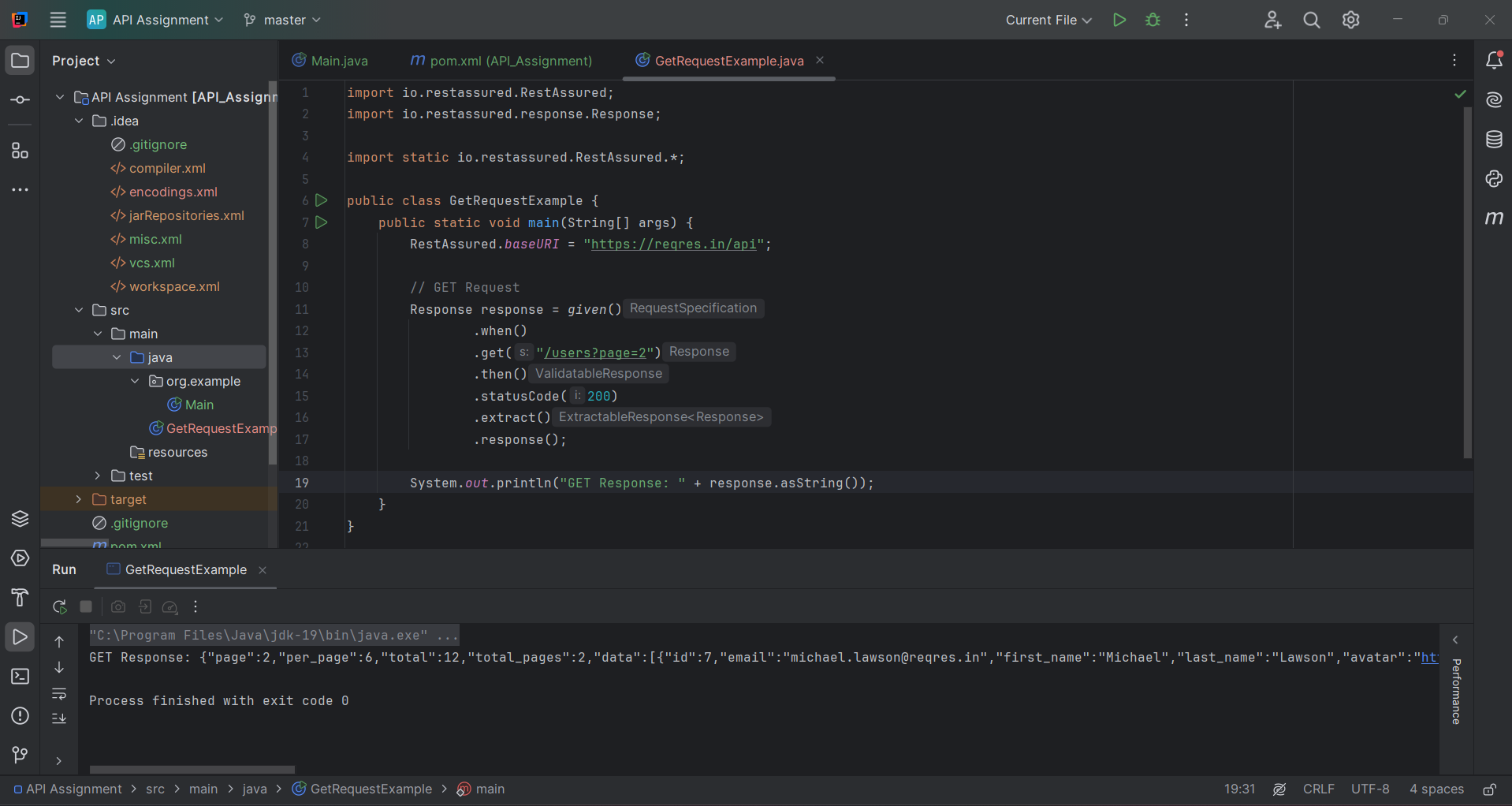
.extract()

.response();

System.out.println("GET Response: " + response.asString());

}

}



**2. POST Request**

import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.\*;

public class PostRequestExample {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// POST Request Body

String postBody = "{ \"name\": \"John\", \"job\": \"developer\" }";

// POST Request

Response response = given()

.header("Content-Type", "application/json")

.body(postBody)

.when()

.post("/users")

.then()

.statusCode(201)

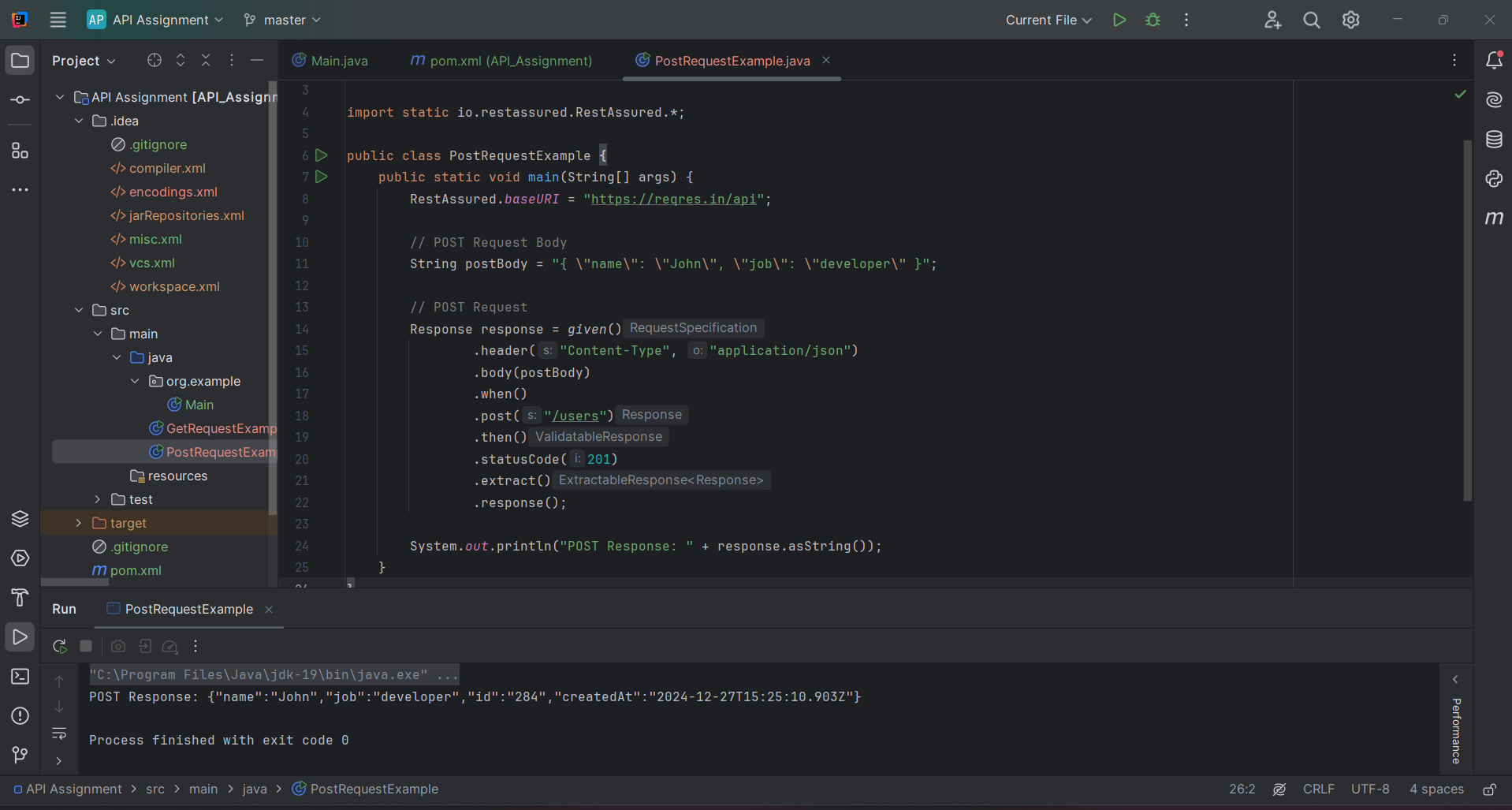
.extract()

.response();

System.out.println("POST Response: " + response.asString());

}

}



**3. PUT Request**

import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.\*;

public class PutRequestExample {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// PUT Request Body

String putBody = "{ \"name\": \"John\", \"job\": \"manager\" }";

// PUT Request

Response response = given()

.header("Content-Type", "application/json")

.body(putBody)

.when()

.put("/users/2")

.then()

.statusCode(200)

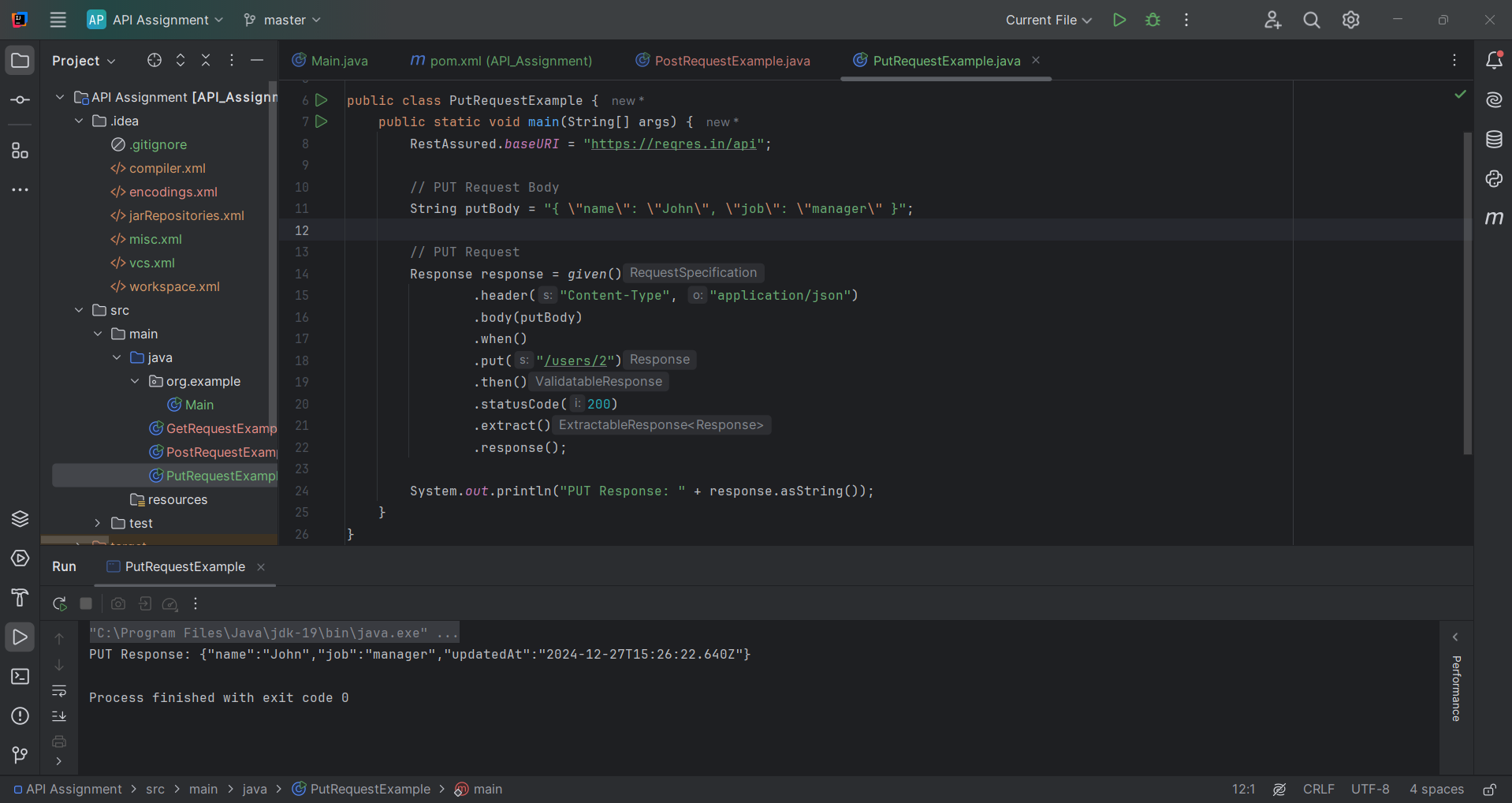
.extract()

.response();

System.out.println("PUT Response: " + response.asString());

}

}



**4. PATCH Request**

import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.\*;

public class PatchRequestExample {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// PATCH Request Body

String patchBody = "{ \"name\": \"John\" }";

// PATCH Request

Response response = given()

.header("Content-Type", "application/json")

.body(patchBody)

.when()

.patch("/users/2")

.then()

.statusCode(200)

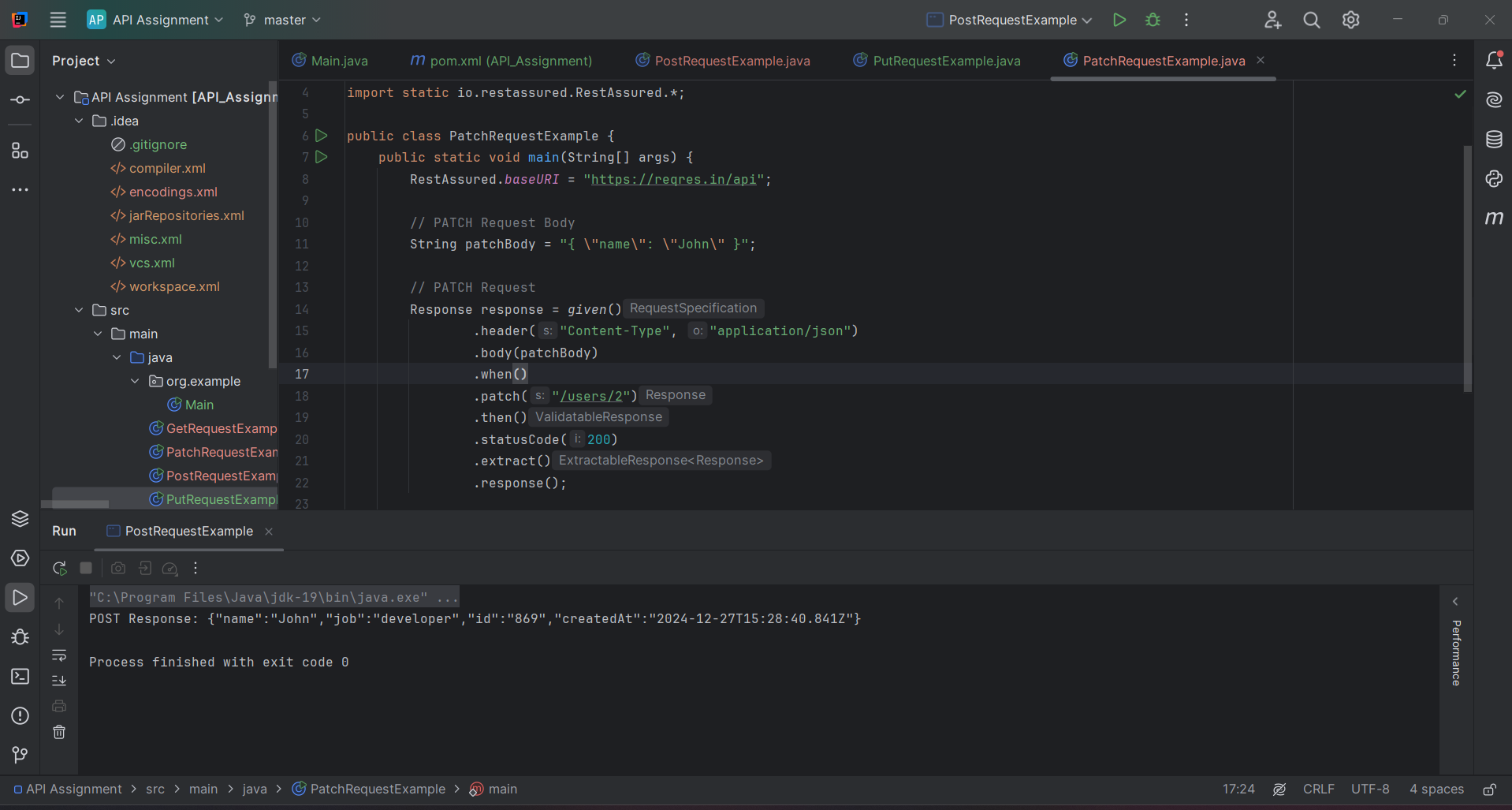
.extract()

.response();

System.out.println("PATCH Response: " + response.asString());

}

}



**5. DELETE Request**

import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.\*;

public class DeleteRequestExample {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// DELETE Request

Response response = given()

.when()

.delete("/users/2")

.then()

.statusCode(204) // No Content

.extract()

.response();

System.out.println("DELETE Response Status Code: " + response.statusCode());

}

}

1. **Assert for 200/201 response**

**Assertion Details**

* **200 Status Code**: Indicates a successful GET, PUT, or PATCH operation.
* **201 Status Code**: Indicates a resource was successfully created via POST.
* If the assertion fails (e.g., the status code isn't 200/201), the test will throw an error.

1. **GET Request with 200 Response Assertion**

import io.restassured.RestAssured;

import static io.restassured.RestAssured.\*;

public class GetRequestWithAssertion {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// Perform GET request and assert for 200 response code

given()

.when()

.get("/users?page=2")

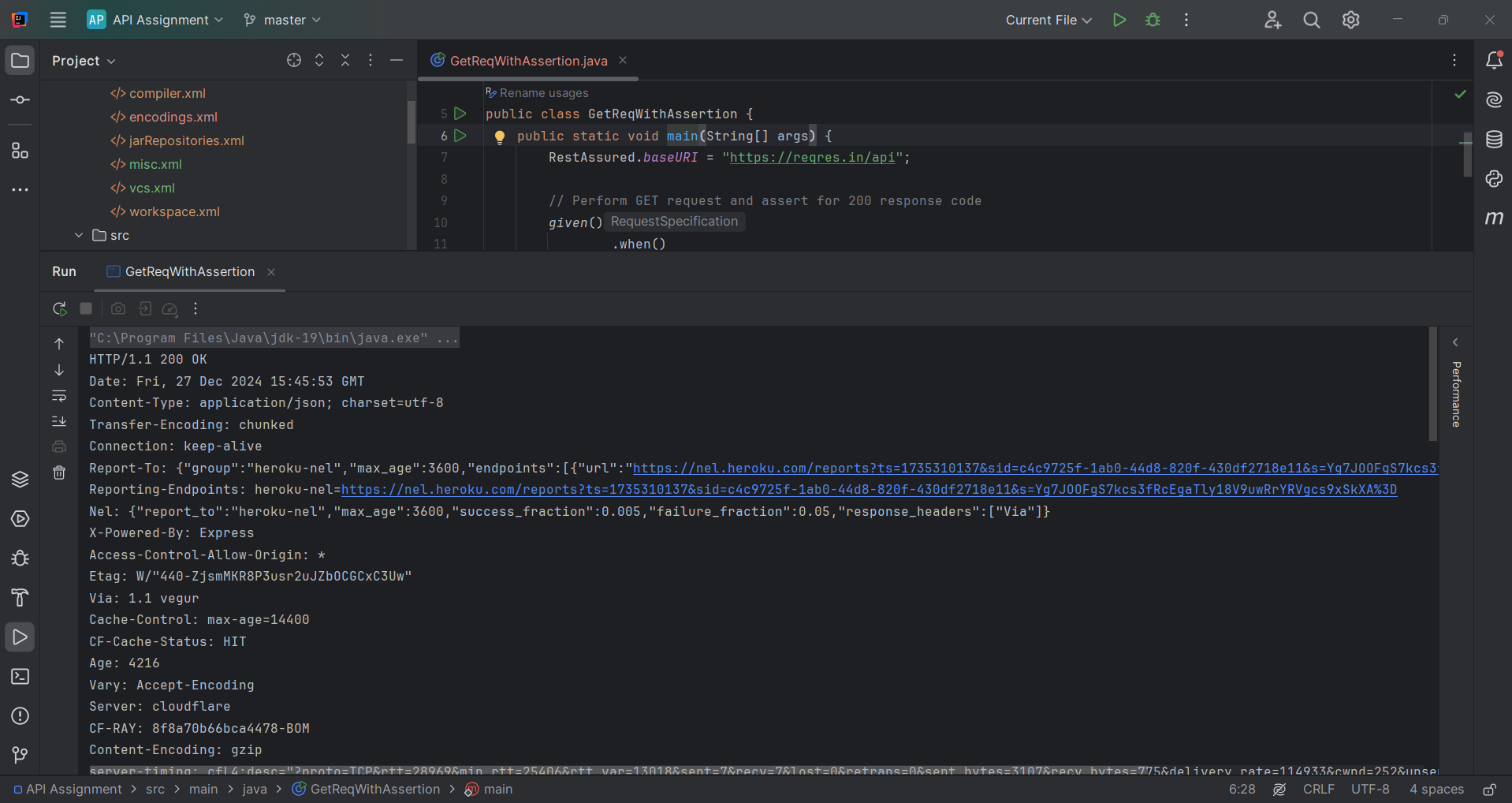
.then()

.statusCode(200) // Assert that the status code is 200

.log().all(); // Log the response to the console

}

}



**2. POST Request with 201 Response Assertion**

import io.restassured.RestAssured;

import static io.restassured.RestAssured.\*;

public class PostRequestWithAssertion {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// JSON Body for POST request

String requestBody = "{ \"name\": \"John\", \"job\": \"developer\" }";

// Perform POST request and assert for 201 response code

given()

.header("Content-Type", "application/json")

.body(requestBody)

.when()

.post("/users")

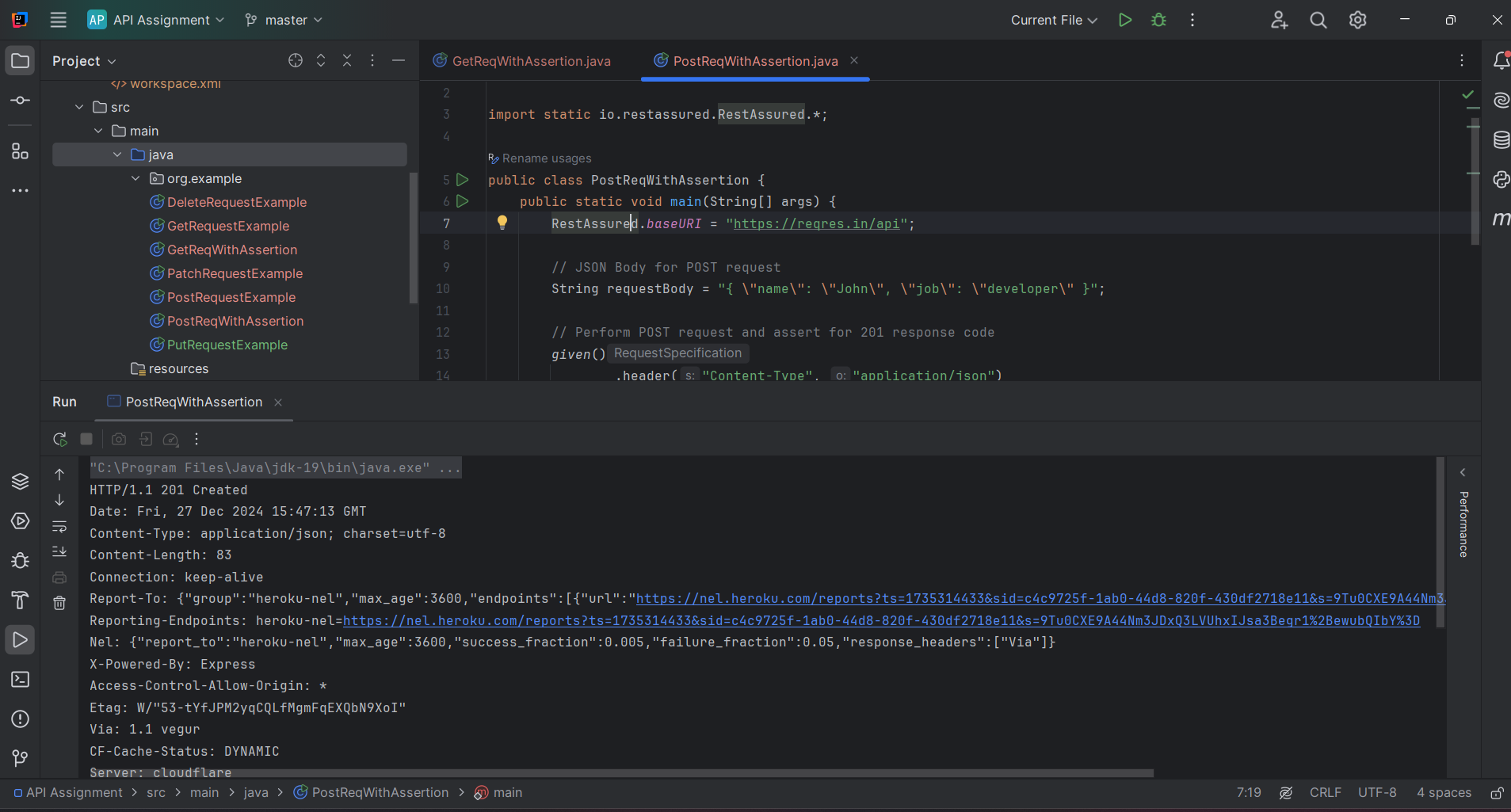
.then()

.statusCode(201) // Assert that the status code is 201

.log().all(); // Log the response to the console

}

}



**3. PUT Request with 200 Response Assertion**

import io.restassured.RestAssured;

import static io.restassured.RestAssured.\*;

public class PutRequestWithAssertion {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// JSON Body for PUT request

String requestBody = "{ \"name\": \"John\", \"job\": \"manager\" }";

// Perform PUT request and assert for 200 response code

given()

.header("Content-Type", "application/json")

.body(requestBody)

.when()

.put("/users/2")

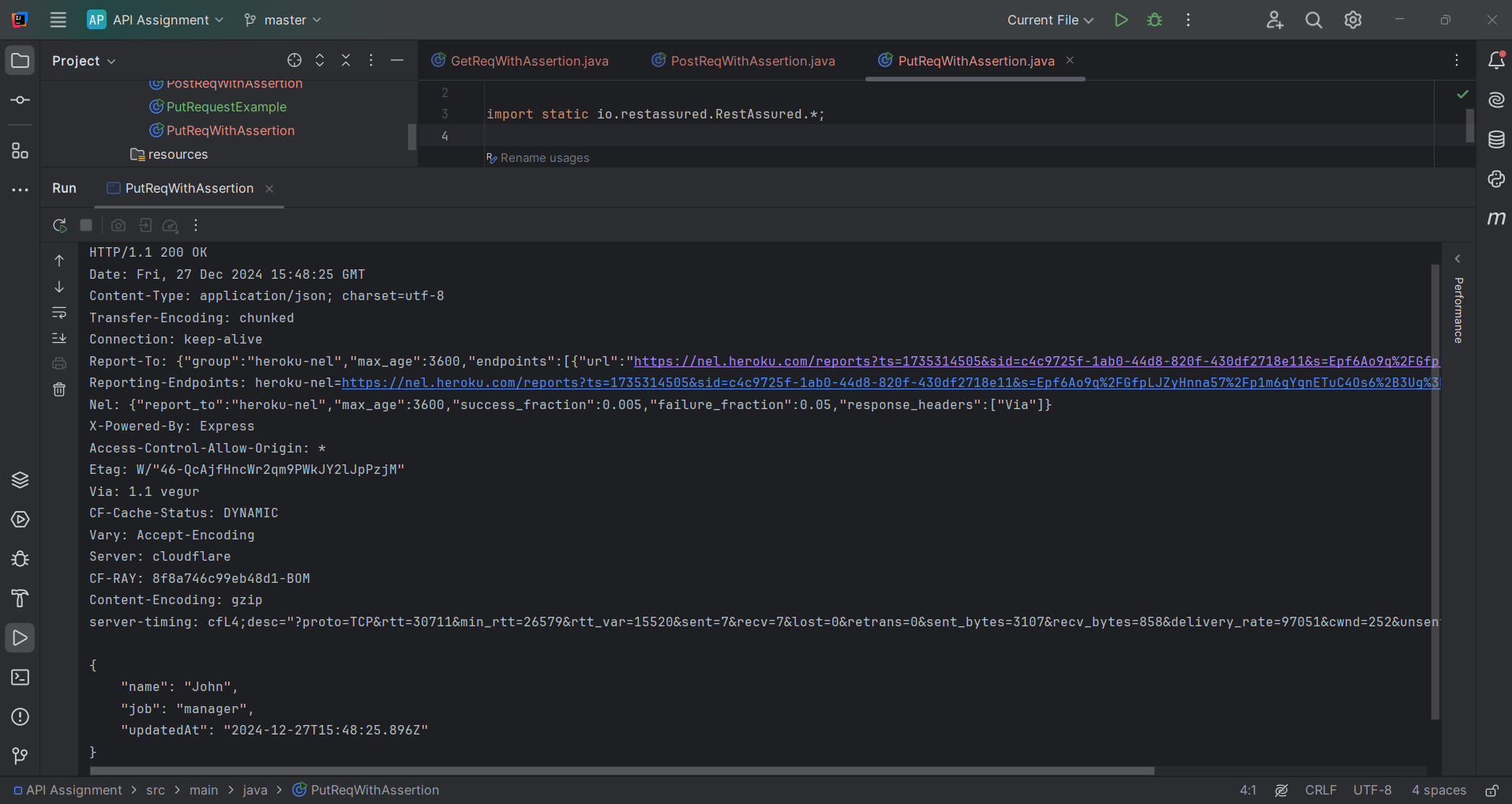
.then()

.statusCode(200) // Assert that the status code is 200

.log().all(); // Log the response to the console

}

}



**4. PATCH Request with 200 Response Assertion**

import io.restassured.RestAssured;

import static io.restassured.RestAssured.\*;

public class PatchRequestWithAssertion {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// JSON Body for PATCH request

String requestBody = "{ \"name\": \"John\" }";

// Perform PATCH request and assert for 200 response code

given()

.header("Content-Type", "application/json")

.body(requestBody)

.when()

.patch("/users/2")

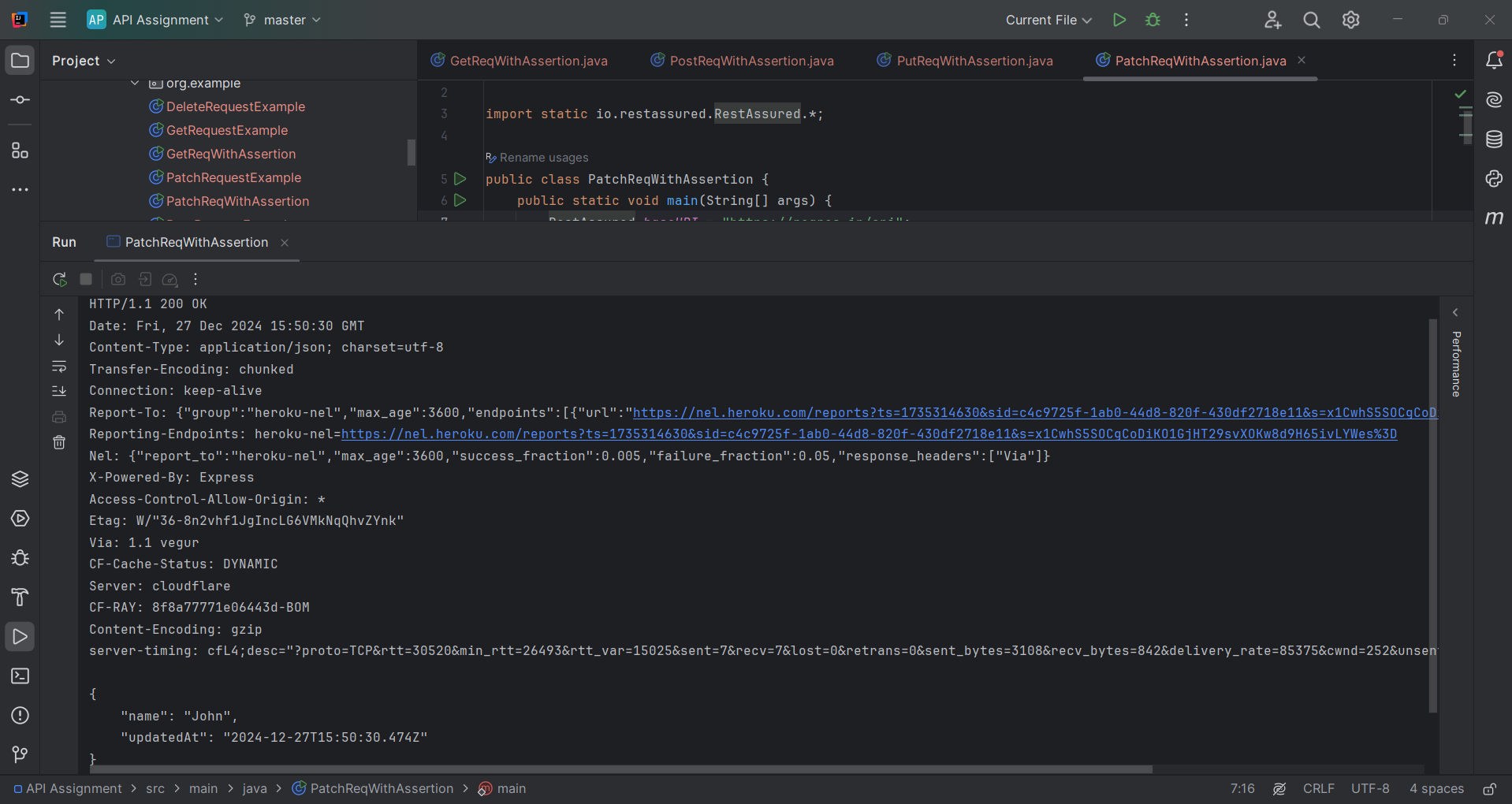
.then()

.statusCode(200) // Assert that the status code is 200

.log().all(); // Log the response to the console

}

}



**5. DELETE Request with 204 Response Assertion**

Although not part of 200/201, DELETE typically returns **204 (No Content)**:

import io.restassured.RestAssured;

import static io.restassured.RestAssured.\*;

public class DeleteRequestWithAssertion {

public static void main(String[] args) {

RestAssured.baseURI = "https://reqres.in/api";

// Perform DELETE request and assert for 204 response code

given()

.when()

.delete("/users/2")

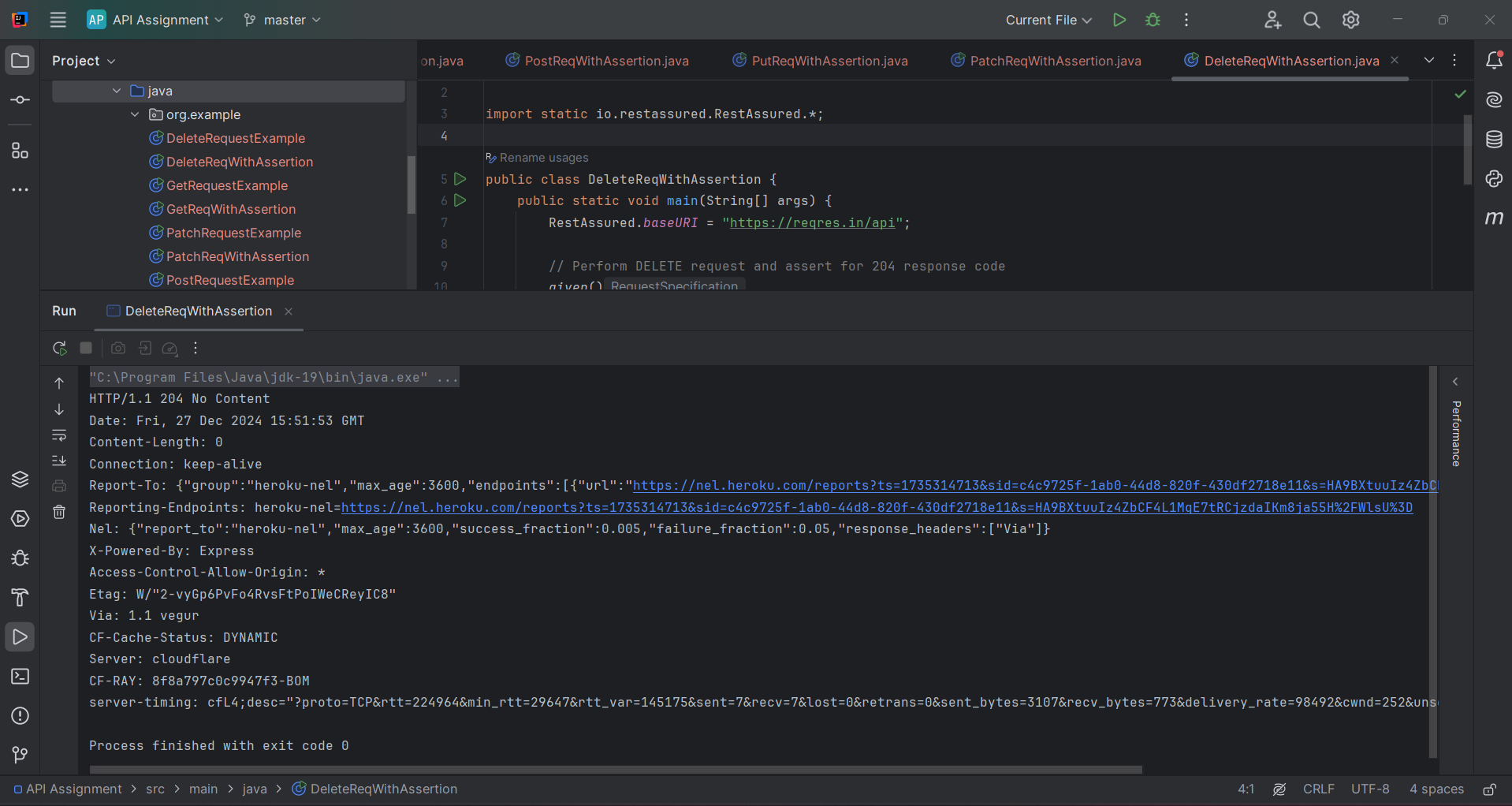
.then()

.statusCode(204) // Assert that the status code is 204

.log().all(); // Log the response to the console

}

}



1. **Verify “first\_name” key value from GET response**

To verify the first\_name key value from the **GET** response using Rest Assured, you can parse the response JSON and assert the expected value using methods like then().body() or extracting the response into a JSON object.

**Verify first\_name Key Value in GET Response**

Here’s the full Java implementation to validate the first\_name key:

import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.\*;

import static org.hamcrest.Matchers.\*;

public class VerifyFirstName {

public static void main(String[] args) {

// Base URI

RestAssured.baseURI = "https://reqres.in/api";

// GET Request

given()

.when()

.get("/users?page=2")

.then()

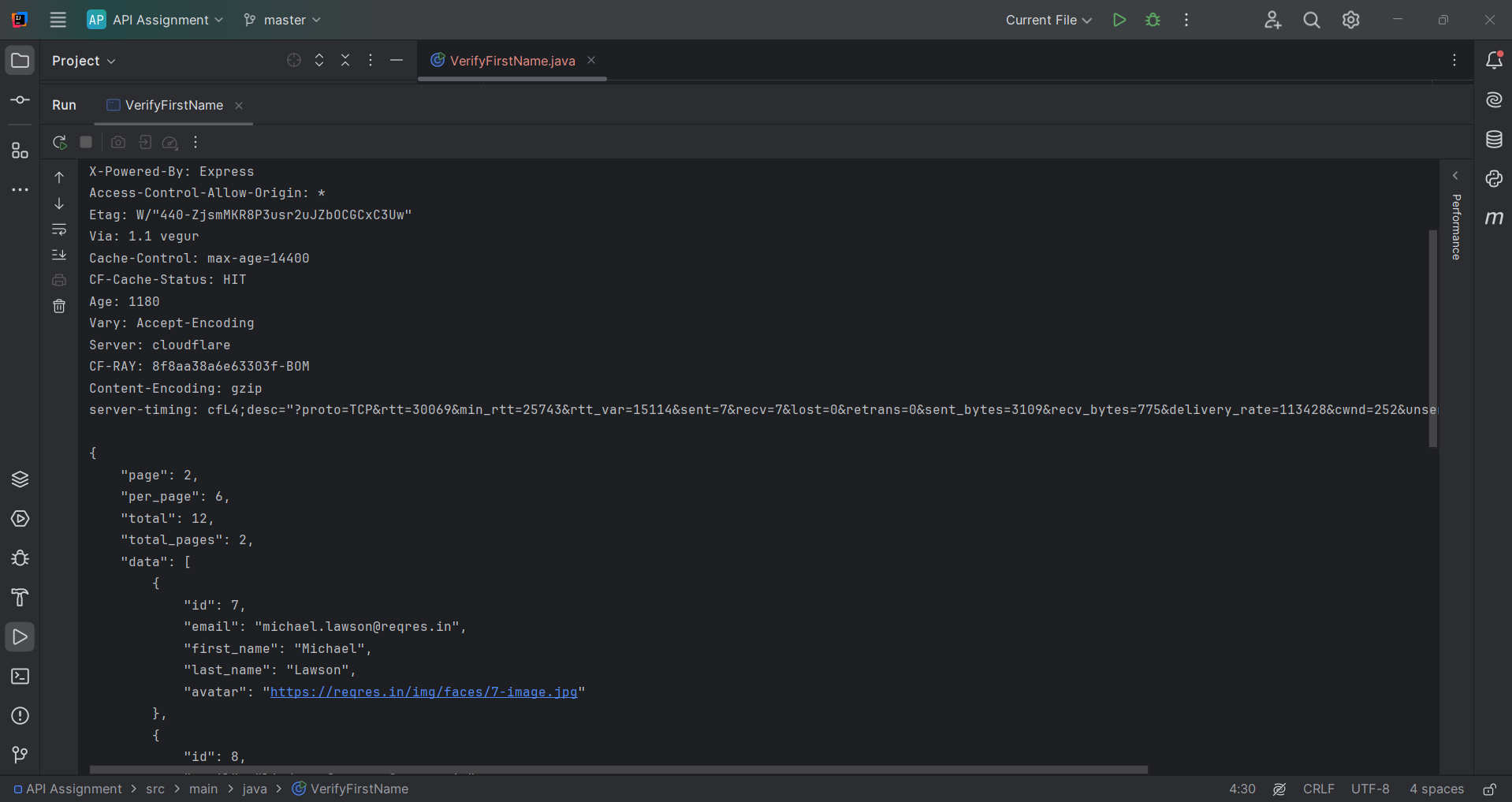
.statusCode(200) // Assert 200 status code

.body("data[0].first\_name", equalTo("Michael")) // Verify "first\_name" key value

.log().all(); // Log response

}

}



**Explanation of the Code**

1. **baseURI**: Sets the base URL for the API.
2. **GET Request**: Fetches the user list from https://reqres.in/api/users?page=2.
3. **Status Code Assertion**: Ensures the response code is 200.
4. **Key Assertion**: Validates that the first\_name of the first user (data[0]) is "Michael".
5. **Log Response**: Outputs the full response to the console for debugging purposes.

**Extract and Print first\_name**

If you want to extract the value of first\_name and print it:

import io.restassured.RestAssured;

import io.restassured.path.json.JsonPath;

import static io.restassured.RestAssured.\*;

public class ExtractFirstName {

public static void main(String[] args) {

// Base URI

RestAssured.baseURI = "https://reqres.in/api";

// GET Request and Extract Response

String response = given()

.when()

.get("/users?page=2")

.then()

.statusCode(200)

.extract()

.asString();

// Parse JSON and Extract First Name

JsonPath jsonPath = new JsonPath(response);

String firstName = jsonPath.getString("data[0].first\_name");

// Print the first name

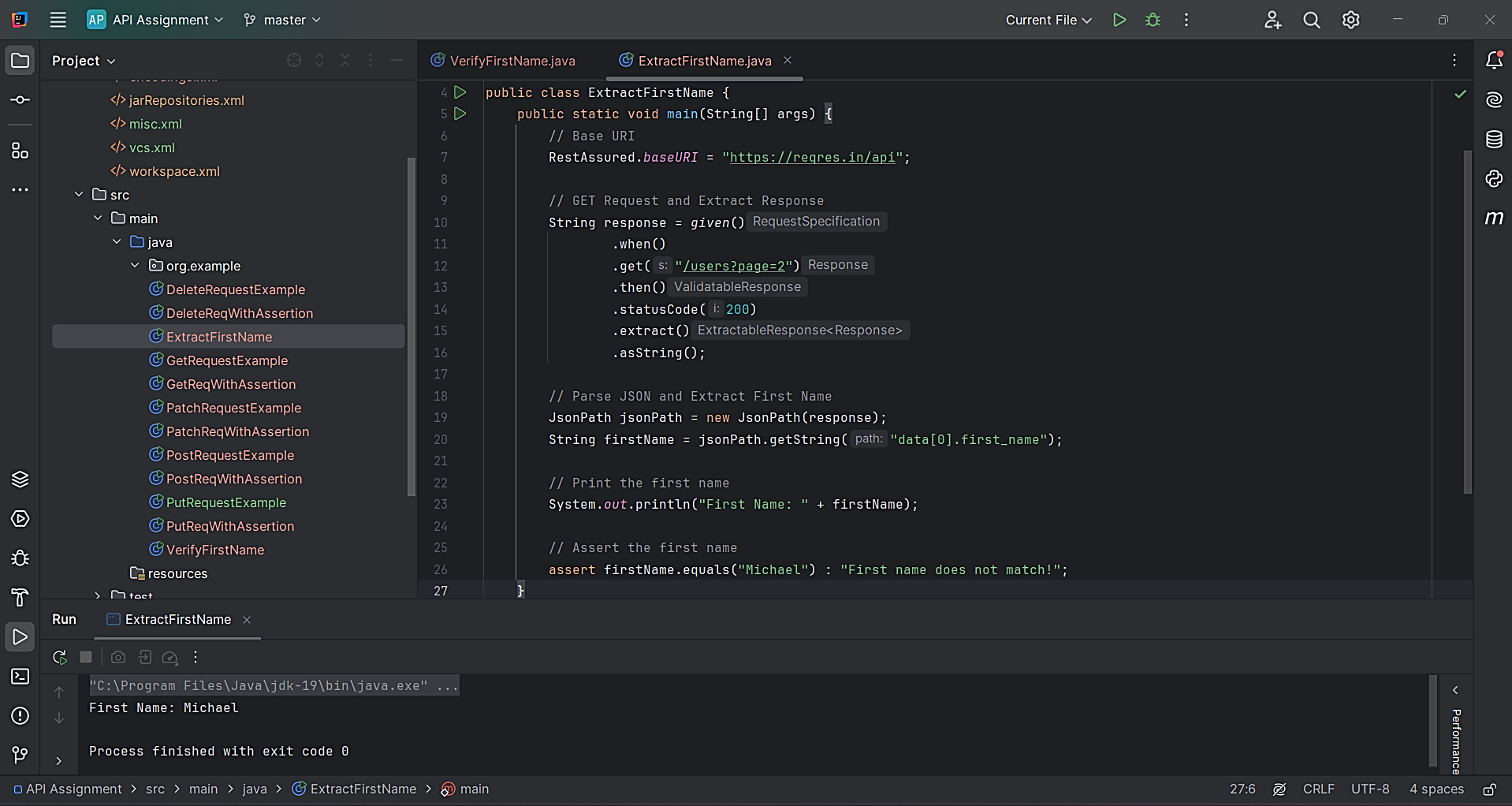
System.out.println("First Name: " + firstName);

// Assert the first name

assert firstName.equals("Michael") : "First name does not match!";

}

}



If the assertion passes, it confirms that the first\_name value is correct. If it fails, an error will be displayed.

1. **Extract “email” from GET request and add it to the subsequent POST request payload and hit the request**

**Steps:**

1. **GET Request**:
   * Fetch the email from the response of the GET request.
   * Extract the email value from the response JSON.
2. **POST Request**:
   * Use the extracted email in the body of the POST request.
   * Send the POST request with the dynamically generated payload.

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.path.json.JsonPath;

import static io.restassured.RestAssured.\*;

public class ExtractAndPost {

public static void main(String[] args) {

// Set Base URI

RestAssured.baseURI = "https://reqres.in/api";

// Step 1: Perform GET request to extract email

Response getResponse = given()

.when()

.get("/users?page=2")

.then()

.statusCode(200) // Assert that the response status is 200

.extract()

.response();

// Extract email from the first user

JsonPath jsonPath = getResponse.jsonPath();

String email = jsonPath.getString("data[0].email"); // Extract "email" from the first user

System.out.println("Extracted Email: " + email);

// Step 2: Use the extracted email in the POST request payload

String postBody = "{ \"name\": \"John\", \"job\": \"developer\", \"email\": \"" + email + "\" }";

// Step 3: Perform POST request with the payload

Response postResponse = given()

.header("Content-Type", "application/json")

.body(postBody)

.when()

.post("/users")

.then()

.statusCode(201) // Assert that the response status is 201

.log().all() // Log the response for debugging

.extract()

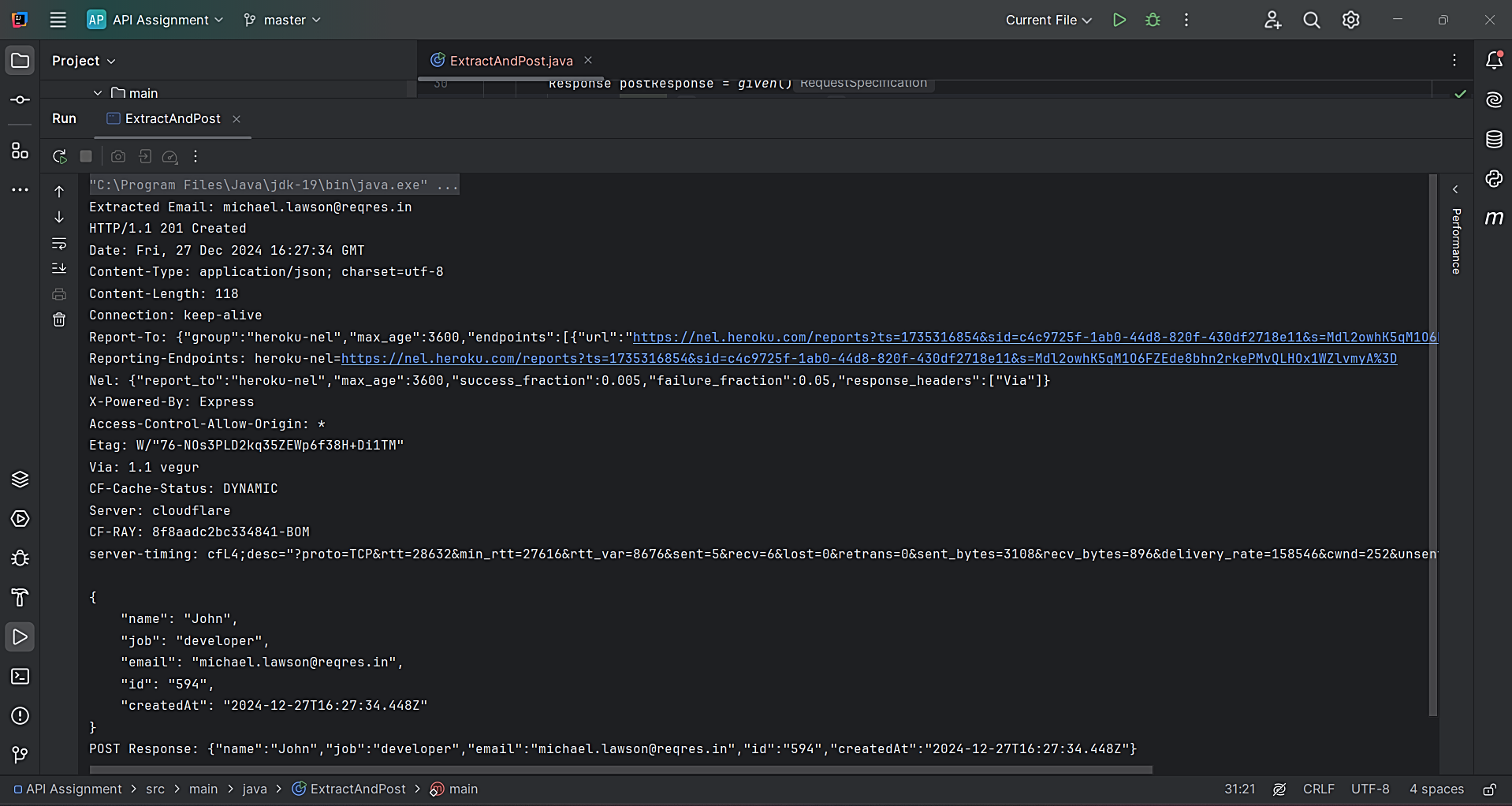
.response();

// Output the POST response

System.out.println("POST Response: " + postResponse.asString());

}

}



**Explanation:**

1. **GET Request**:
   * Fetches the list of users from https://reqres.in/api/users?page=2.
   * Extracts the email value of the first user (data[0].email) using JsonPath.
2. **POST Request**:
   * Constructs the POST request payload dynamically, incorporating the extracted email.
   * Sends the POST request to https://reqres.in/api/users with the payload.
3. **Dynamic Payload**:
   * The email from the GET response is inserted into the POST request JSON body as:

{

"name": "John",

"job": "developer",

"email": "extracted\_email@example.com"

}

1. **Response Assertions**:
   * Asserts that the GET request status is 200 and the POST request status is 201.
   * Logs the POST response to the console.
2. **Write a function to verify presence of Message in response body**

To verify the presence of a specific **message** in the response body using Rest Assured, you can write a function that checks if the response body contains a given substring or matches a specific key-value pair.

**Code Implementation**

import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.\*;

import static org.hamcrest.Matchers.\*;

public class VerifyMessageInResponse {

public static void main(String[] args) {

// Base URI

RestAssured.baseURI = "https://reqres.in/api";

// Perform GET request

Response response = given()

.when()

.get("/users?page=2")

.then()

.statusCode(200) // Assert status code

.extract()

.response();

// Convert response body to String

String responseBody = response.asString();

// Verify the presence of a message or specific text in the response body

verifyMessage(responseBody, "Michael");

}

/\*\*

\* Function to verify if a message is present in the response body

\*

\* @param responseBody The response body as a String

\* @param message The message to verify

\*/

public static void verifyMessage(String responseBody, String message) {

if (responseBody.contains(message)) {

System.out.println("Message \"" + message + "\" is present in the response body.");

} else {

System.out.println("Message \"" + message + "\" is NOT present in the response body.");

}

}

}

**Explanation**

1. **response.asString()**:
   * Converts the response body to a String for easy searching or substring operations.
2. **verifyMessage Function**:
   * Takes the response body and the message to be verified as arguments.
   * Checks if the response body contains the given message using the contains() method.
   * Prints whether the message is present or not.
3. **Example Use Case**:
   * Verifying the presence of a name like "Michael" or a specific message in the response body.

