1. What are the different API methods you have tested?

In API testing, we commonly test the following HTTP methods:

- **GET**: Retrieves data from the server. This method is used to request data from a specified resource. It is read-only and does not change the state of the resource.

```
given()
    .when()
    .get("https://api.example.com/users")
    .then()
    .statusCode(200);
```

- **POST**: Sends data to the server to create a new resource. This method submits data to be processed to a specified resource.

```
given()
    .contentType("application/json")
    .body("{ \"name\": \"John\", \"age\": 30 }")
    .when()
    .post("https://api.example.com/users")
    .then()
    .statusCode(201);
```

- **PUT**: Updates an existing resource on the server. If the resource does not exist, it can create it.

```
given()
    .contentType("application/json")
    .body("{ \"name\": \"John\", \"age\": 31 }")
    .when()
    .put("https://api.example.com/users/1")
    .then()
    .statusCode(200);
```

- **DELETE**: Removes a resource from the server.

```
given()
    .when()
    .delete("https://api.example.com/users/1")
    .then()
    .statusCode(204);
```

- **PATCH**: Partially updates an existing resource.

```
given()
    .contentType("application/json")
    .body("{ \"age\": 32 }")
    .when()
    .patch("https://api.example.com/users/1")
    .then()
    .statusCode(200);
```

2. What is the PUT method?

The **PUT** method is used to update an existing resource or create a new resource if it does not exist. It is idempotent, meaning multiple identical requests should have the same effect as a single request.

Example:

```
given()
    .contentType("application/json")
    .body("{ \"name\": \"John\", \"age\": 30 }")
    .when()
    .put("https://api.example.com/users/1")
    .then()
    .statusCode(200);
```

In this example, a user with ID `1` is updated with the name "John" and age `30`.

3. What type of payload does POST support?

The **POST** method supports various types of payloads, including:

- **JSON**: The most common format for APIs.

  ```
  given()
      .contentType("application/json")
      .body("{ \"name\": \"John\", \"age\": 30 }")
      .when()
      .post("https://api.example.com/users")
      .then()
      .statusCode(201);
  ```

- **XML**: Another format, although less common than JSON.

  ```
  given()
      .contentType("application/xml")
      .body("<user><name>John</name><age>30</age></user>")
      .when()
      .post("https://api.example.com/users")
      .then()
      .statusCode(201);
  ```

- **Form Data**: Often used in web forms.

  ```
  given()
      .contentType("application/x-www-form-urlencoded")
      .formParam("name", "John")
      .formParam("age", "30")
      .when()
      .post("https://api.example.com/users")
      .then()
  ```

```
    .statusCode(201);
```

- **Multipart Data**: Used for file uploads.

```
given()
    .multiPart(new File("/path/to/file"))
    .when()
    .post("https://api.example.com/upload")
    .then()
    .statusCode(201);
```

4. Key difference between PUT and POST

- **PUT**:
  - Idempotent: Multiple identical requests have the same effect as a single request.
  - Used to update an existing resource or create a resource if it does not exist.
  - Example:

```
given()
    .contentType("application/json")
    .body("{ \"name\": \"John\", \"age\": 30 }")
    .when()
    .put("https://api.example.com/users/1")
    .then()
    .statusCode(200);
```

- **POST**:
  - Not idempotent: Multiple identical requests can create multiple resources.
  - Used to create a new resource.
  - Example:

```
given()
    .contentType("application/json")
    .body("{ \"name\": \"John\", \"age\": 30 }")
    .when()
    .post("https://api.example.com/users")
    .then()
    .statusCode(201);
```

5. Explain different data types in payload for API request

- **String**: Plain text values.

```
{ "name": "John" }
```

- **Number**: Integers or floating-point numbers.

```
{ "age": 30 }
```

- **Boolean**: `true` or `false`.

```
{ "isActive": true }
```

- **Array**: An ordered list of values.

```
{ "tags": ["developer", "blogger"] }
```

- **Object**: A collection of key-value pairs.

```
{
    "address": {
        "street": "123 Main St",
        "city": "Anytown"
    }
}
```

6. Explain how do you assert uniqueness of ID?

To assert the uniqueness of an ID, you can retrieve the list of existing IDs and ensure that the new ID is not already present in the list.

Example:

```
List<Integer> ids = given()
    .when()
    .get("https://api.example.com/users")
    .jsonPath().getList("id");

int newId = 101;
assertFalse(ids.contains(newId)); // Ensure newId is unique
```

7. Let's say for a bank user with multiple accounts, how do you verify account numbers array?

To verify the account numbers array, you can assert that the array contains the expected account numbers.

Example:

```
given()
    .when()
    .get("https://api.example.com/users/1/accounts")
    .then()
    .statusCode(200)
    .body("accounts", hasItems("123456", "789012", "345678"));
```

8. Represent users API response in the form of JSON (structure of the response).

Here is an example of a JSON response for a user with multiple accounts:

```
{
```

```
    "id": 1,
    "name": "John Doe",
    "email": "john.doe@example.com",
    "accounts": [
        {
            "accountNumber": "123456",
            "balance": 1000.0
        },
        {
            "accountNumber": "789012",
            "balance": 2500.5
        }
    ]
}
```

9. What is a mocking response and how do you achieve mocking?

A mocking response is a simulated response for testing purposes. It is used to test the API client in isolation from the actual server, ensuring that the client can handle various scenarios without relying on the live server.

You can achieve mocking using tools like WireMock.

Example using WireMock:

```
WireMockServer wireMockServer = new WireMockServer();
wireMockServer.start();
wireMockServer.stubFor(get(urlEqualTo("/users/1"))
    .willReturn(aResponse()
        .withHeader("Content-Type", "application/json")
        .withBody("{ \"id\": 1, \"name\": \"John Doe\" }")));

given()
    .when()
    .get("http://localhost:8080/users/1")
    .then()
    .statusCode(200)
    .body("name", equalTo("John Doe"));

wireMockServer.stop();
```

10. What is API automation testing? How does it differ from UI automation testing?

API automation testing involves testing the APIs directly to ensure they meet functionality, performance, and reliability expectations. It focuses on business logic, data responses, and error handling without a GUI.

UI automation testing involves testing the application's user interface to ensure it behaves as expected when interacted with by a user. It focuses on the graphical interface and user interactions.

11. What are the advantages of API automation testing?

- **Faster**: Tests run faster compared to manual testing.
- **Reliable**: Automated tests are more consistent and less prone to human error.
- **Early Issue Detection**: Issues can be detected early in the development cycle.
- **Cost-Effective**: Reduces the cost of testing by automating repetitive tasks.
- **Comprehensive**: Provides better test coverage for business logic.
- **Independent of UI Changes**: Tests are not affected by changes in the user interface.

12. How do you select the appropriate tools and frameworks for API automation testing?

Consider the following when selecting tools and frameworks:

- **Language Compatibility**: Ensure the tool supports the programming language used in your project.
- **Ease of Use**: Tools should be easy to configure and use.
- **Community Support**: Strong community and documentation support are essential.
- **Integration**: The ability to integrate with CI/CD pipelines and other tools.
- **Flexibility**: Support for different data formats (JSON, XML) and authentication methods (OAuth, API keys).

13. Explain the steps involved in testing an API.

1. **Define Requirements**: Understand the API specifications and requirements.
2. **Setup Environment**: Configure the test environment and data.
3. **Create Test Cases**: Design test cases for different scenarios, including positive and negative cases.
4. **Execute Tests**: Run the test cases using an automation tool like Rest Assured.
5. **Validate Responses**: Verify the responses against expected results using assertions.
6. **Report Bugs**: Document any issues found during testing.
7. **Automate Regression Testing**: Ensure new changes do not break existing functionality.

14. What are the commonly used HTTP methods in API testing?

- **GET**: Retrieve data from the server.
- **POST**: Send data to the server to create a new resource.
- **PUT**: Update an existing resource or create a new one if it does not exist.
- **DELETE**: Remove a resource from the server.
- **PATCH**: Partially update an existing resource.

15. What is the difference between GET and POST methods in API testing?

- **GET**:
  - Used to retrieve data from the server.
  - Does not change the state of the resource.

- Idempotent: Multiple identical requests have the same effect.
- Example:

```
given()
    .when()
    .get("https://api.example.com/users")
    .then()
    .statusCode(200);
```

- **POST**:
  - Used to send data to the server to create a new resource.
  - Can change the state of the resource.
  - Not idempotent: Multiple identical requests can create multiple resources.
  - Example:

```
given()
    .contentType("application/json")
    .body("{ \"name\": \"John\", \"age\": 30 }")
    .when()
    .post("https://api.example.com/users")
    .then()
    .statusCode(201);
```