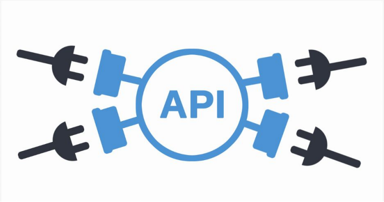
API Testing

Course Content

- Introduction to API testing
- Introduction to Postman
- API testing with Postman
- Advanced Postman concepts
- Introduction to REST Assured
- API testing with REST Assured
- Advanced REST Assured concepts
- API testing best practices and tips

Application Programming Interface (API)



What is an API?

- **Definition**:
  - API stands for **Application Programming Interface**.
  - Acts as an **intermediary** that allows two applications to communicate with each other.
- **Key Characteristics**:
  - Enables integration between software systems.
  - Promotes reusability of services.
- **Real-World Example**:
  - Using an app to check weather data:
    - App (client) communicates with a weather service (API) to fetch and display data.

Client-Server Architecture

What is Client-Server Architecture?

- A model where **clients** (users or applications) request resources or services, and **servers** provide them.
- The foundation for web-based communication, including API interactions.

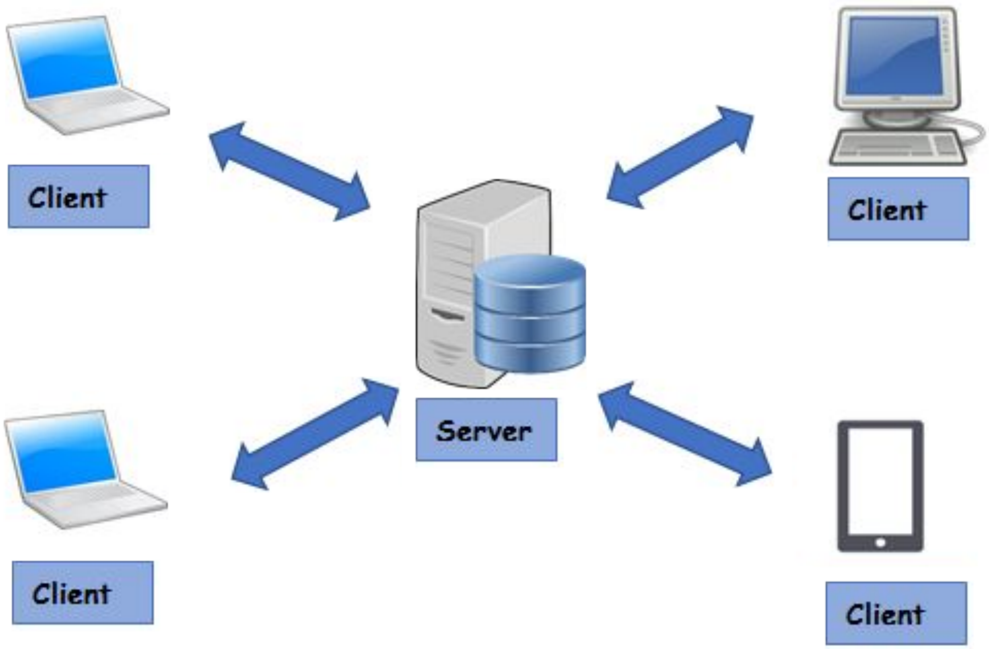
Key Components

1. **Client**:
   - Initiates the request.
   - Examples: Web browsers, mobile apps, Postman.
2. **Server**:
   - Processes client requests and sends responses.
   - Examples: Web servers, database servers.

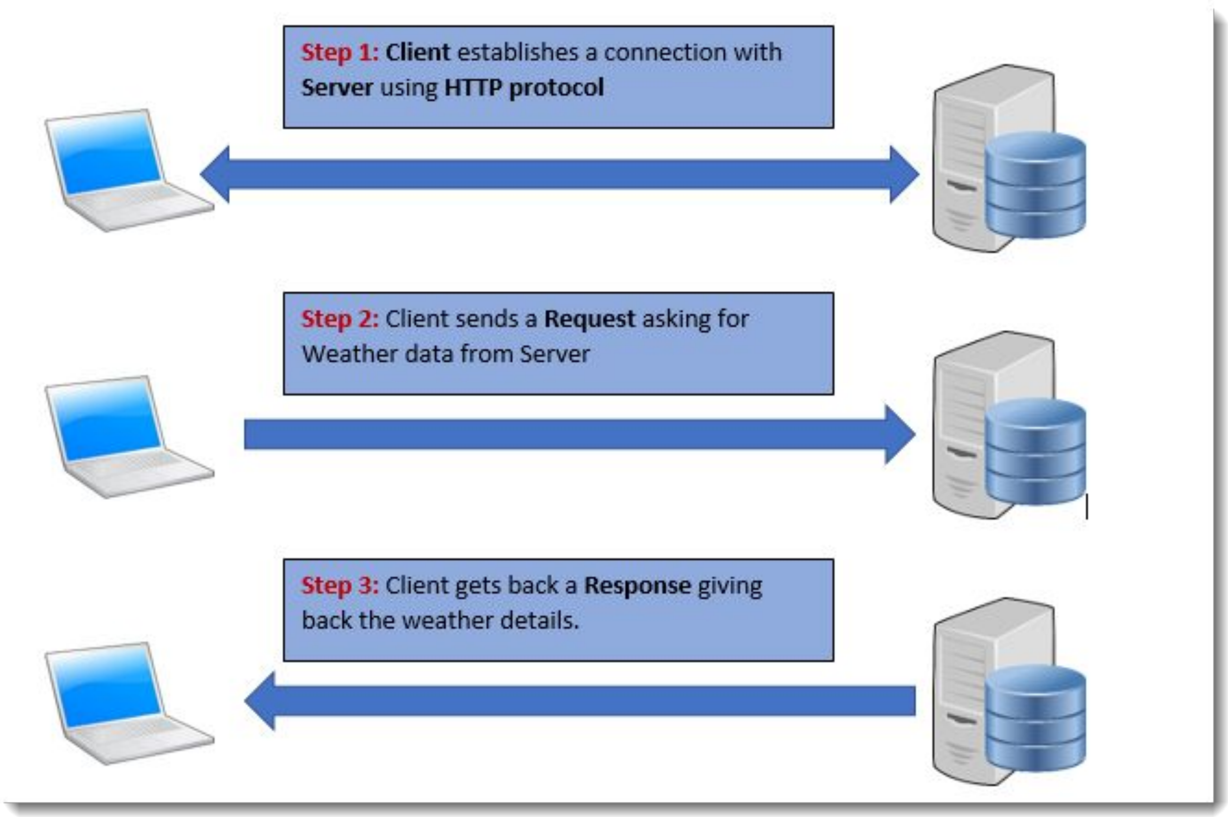
Benefits of Client-Server Architecture

- **Scalability**: Multiple clients can connect to a centralized server.
- **Flexibility**: Clients can use different platforms or devices.
- **Security**: Centralized control of data and services.

Client-Server Architecture



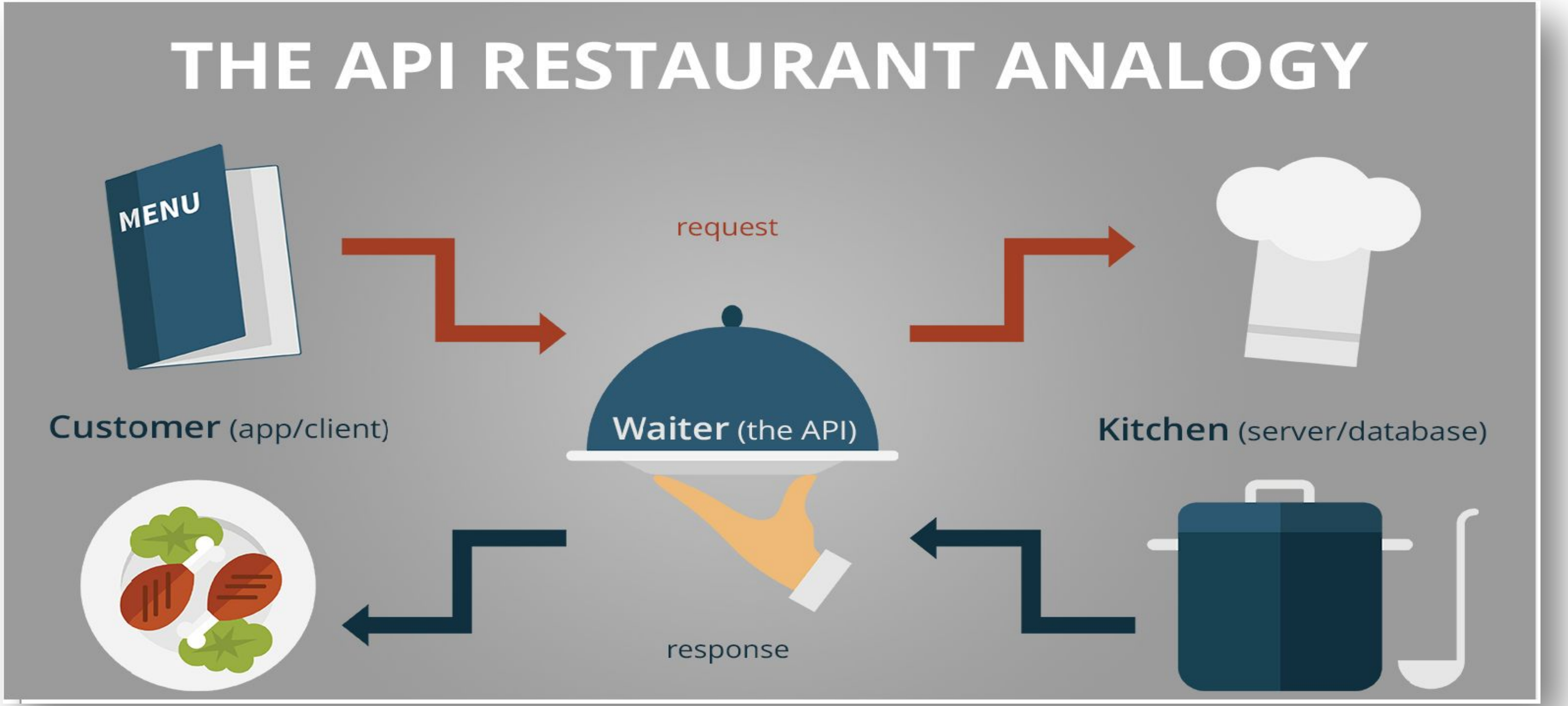
Client-Server Architecture



Step 1: **Client** establishes a connection with **Server** using **HTTP protocol**

Step 2: Client sends a **Request** asking for Weather data from Server

Step 3: Client gets back a **Response** giving back the weather details.


THE API RESTAURANT ANALOGY
MENU
request
Customer (app/client)
Waiter (the API)
Kitchen (server/database)
response

REST Architecture

Definition: **Re**presentational **S**tate **T**ransfer (REST), an architectural style for building web APIs.

Key Principles:

1. **Statelessness**: Each request contains all information needed for processing.
2. **Client-Server Separation**: Clear division between the client and server responsibilities.
3. **Uniform Interface**: Resources are identified by URIs, and actions are performed using standard HTTP methods.
4. **Cacheability**: Responses can be cached to improve performance.

REST API Example:

- **Base URL**: https://api.example.com/users
  - GET: Fetch user data.
  - POST: Create a new user.

HTTP Request

Definition: A packet of information sent from the client to the server for communication.

Parts of an HTTP Request:

- **Request Line**:
  - Contains the **Method** (e.g., GET, POST, PUT, DELETE).
  - **Request URI**: The resource being requested.
  - **HTTP Version**: The protocol version used (e.g., HTTP/1.1).
- **Headers**:
  - Contains additional metadata or information about the request.
  - Can have zero or more headers.
- **Request Body**:
  - Contains data sent to the server (optional).
  - Common formats include JSON, XML, etc.
  - Not always present (e.g., in GET requests).

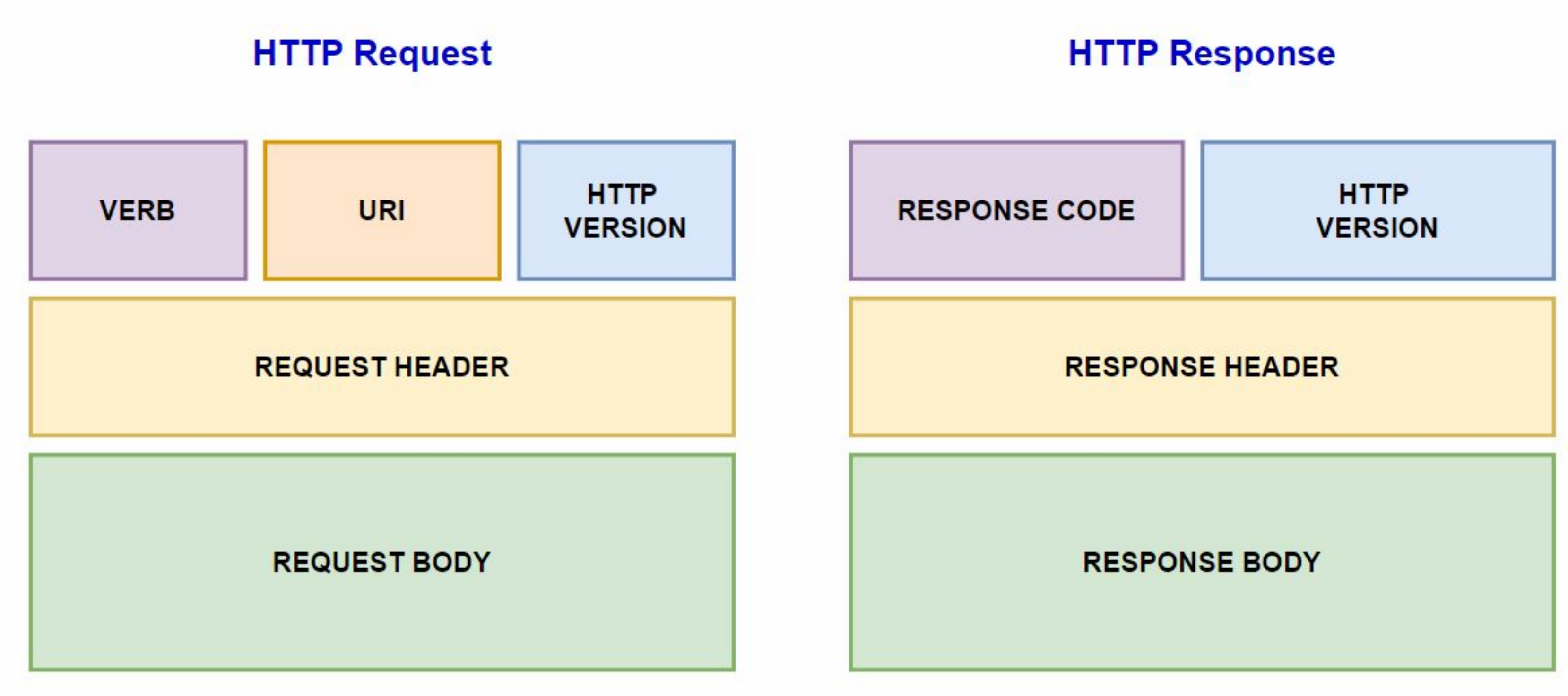
HTTP Response

Definition: A packet of information sent by the server back to the client in response to their request.

Parts of an HTTP Response:

- **Status Line**:
  - **HTTP Version**: Protocol used (e.g., HTTP/1.1).
  - **Status Code**: Numeric code indicating the result (e.g., 200 OK, 404 Not Found).
  - **Reason Phrase**: A brief description of the status (e.g., "OK").
- **Response Headers**:
  - Contains additional metadata about the response.
  - Can have zero or more headers (though responses typically have at least one).
- **Response Body**:
  - Contains the resource or data requested by the client (e.g., weather data).
  - May include various details (e.g., temperature, humidity, weather description).

HTTP Request & Response



HTTP Request

VERB

URI

HTTP VERSION

REQUEST HEADER

REQUEST BODY

HTTP Response

RESPONSE CODE

HTTP VERSION

RESPONSE HEADER

RESPONSE BODY

HTTP Methods

An **HTTP method** (also known as an **HTTP verb**) is an action or operation that a client (typically a web browser or other HTTP client) can request the server to perform on a resource.

- HTTP methods define the type of interaction the client wants to have with the resource on the server.
- Each HTTP method has a specific purpose, and they are part of the HTTP request sent from the client to the server.
- These methods are standardized and provide a way for clients and servers to communicate in a clear and consistent manner.

Common HTTP Methods

- **GET**: Retrieves data from the server. It does not modify or affect the resource.
- **POST**: Sends data to the server, usually to create or update a resource.
- **PUT**: Replaces an existing resource with the new data provided.
- **DELETE**: Deletes the specified resource from the server.

HTTP Methods Contd.

HTTP Method	**Description**	**Use Case**
GET	Retrieves data from the server (e.g., a web page or API data).	Used to fetch resources without making changes to the server.
POST	Submits data to the server (e.g., form data, file upload).	Used to create resources or submit data for processing.
PUT	Replaces the current resource with the new data provided.	Used to update an existing resource or create it if it doesn't exist.
DELETE	Deletes the specified resource from the server.	Used to remove a resource (e.g., deleting a user or record).
PATCH	Partially updates the resource with the given data.	Used to apply partial modifications to a resource.
HEAD	Retrieves the headers of a resource without the body (like GET but without data).	Used to check metadata about a resource (e.g., to check existence or last-modified date).
OPTIONS	Describes the communication options for the target resource.	Used to determine allowed operations or request methods on a resource.
CONNECT	Establishes a tunnel to the server, typically used for SSL/TLS connections.	Used for proxying connections, often in HTTPS communications.
TRACE	Echoes back the received request for diagnostic purposes (useful for debugging).	Used to trace the request path to the server (e.g., for debugging).

HTTP Response Codes

CATEGORY	DESCRIPTION
1xx: Informational	Communicates transfer protocol-level information.
2xx: Success	Indicates that the REST web-service successfully carried out whatever action the client requested
3xx: Redirection	Indicates that the client must take some additional action in order to complete their request.
4xx: Client Error	This category of error status codes points the finger at clients.
5xx: Server Error	The server takes responsibility for these error status codes.

HTTP Response Codes Contd.

- **200**—For Successful request.
- **201**—For successful request and data was created.
- **204**—For Empty Response.
- **400**—For Bad Request. The request could not be understood or was missing any required parameters.
- **401**—For Unauthorized access. Authentication failed or user does not have permissions for the requested operation.
- **403**—For Forbidden, Access denied.
- **404**—For data not found.
- **405**—For Method Not Allowed or Requested method is not supported.
- **500**—For Internal Server Error.
- **503**—For Service Unavailable.

HTTP URI and Resources

- REST uses URI to identify resources

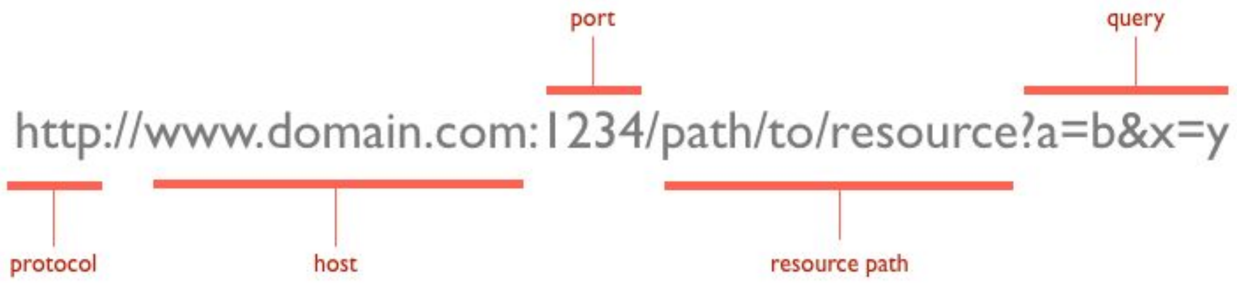
  http://localhost/books/

  http://localhost/books/ISBN-0011

  http://localhost/books/ISBN-0011/authors

- As you traverse the path from more generic to more specific, you are navigating the data

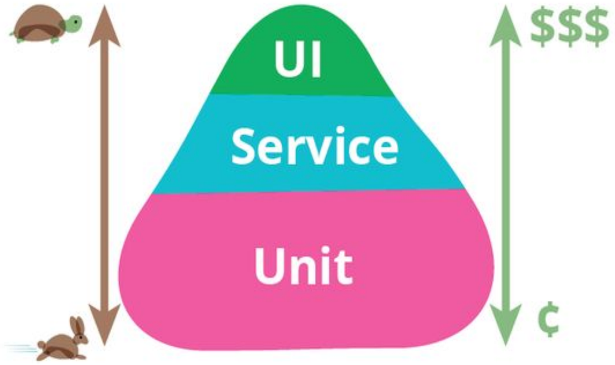
URI:



Why Test APIs?

Benefits:

- Test the core functionality
- Improved test coverage
- Time Effective – faster regression cycles
- Find defects early
- Language independent



Types of API Testing:

- **Functional Testing**: Verify the API performs expected operations.
- **Performance Testing**: Measure response times under different loads.
- **Security Testing**: Assess vulnerabilities such as unauthorized access.

Common Tools for API Testing

- **Postman**: User-friendly interface for manual and automated testing.
- **REST Assured**: Java library for testing REST APIs programmatically.
- **Swagger**: For API documentation and testing.
- **JMeter**: Load testing tool.

Introduction to Postman

Postman is a popular API testing tool that provides a user-friendly interface for designing, testing, and documenting APIs.

Why Use Postman?

- Simplifies API testing with an intuitive GUI.
- Supports automation through scripting and collections.
- Works seamlessly with team collaboration features.

Key Features of Postman

1. **API Requests**: Easily send GET, POST, PUT, DELETE, and other HTTP requests.
2. **Collections**: Group related API requests for organized testing.
3. **Environment Variables**: Manage dynamic data like API keys or base URLs.
4. **Pre-request and Test Scripts**: Add JavaScript code for request preprocessing or validation.
5. **Newman**: CLI tool to run Postman collections in automated pipelines.

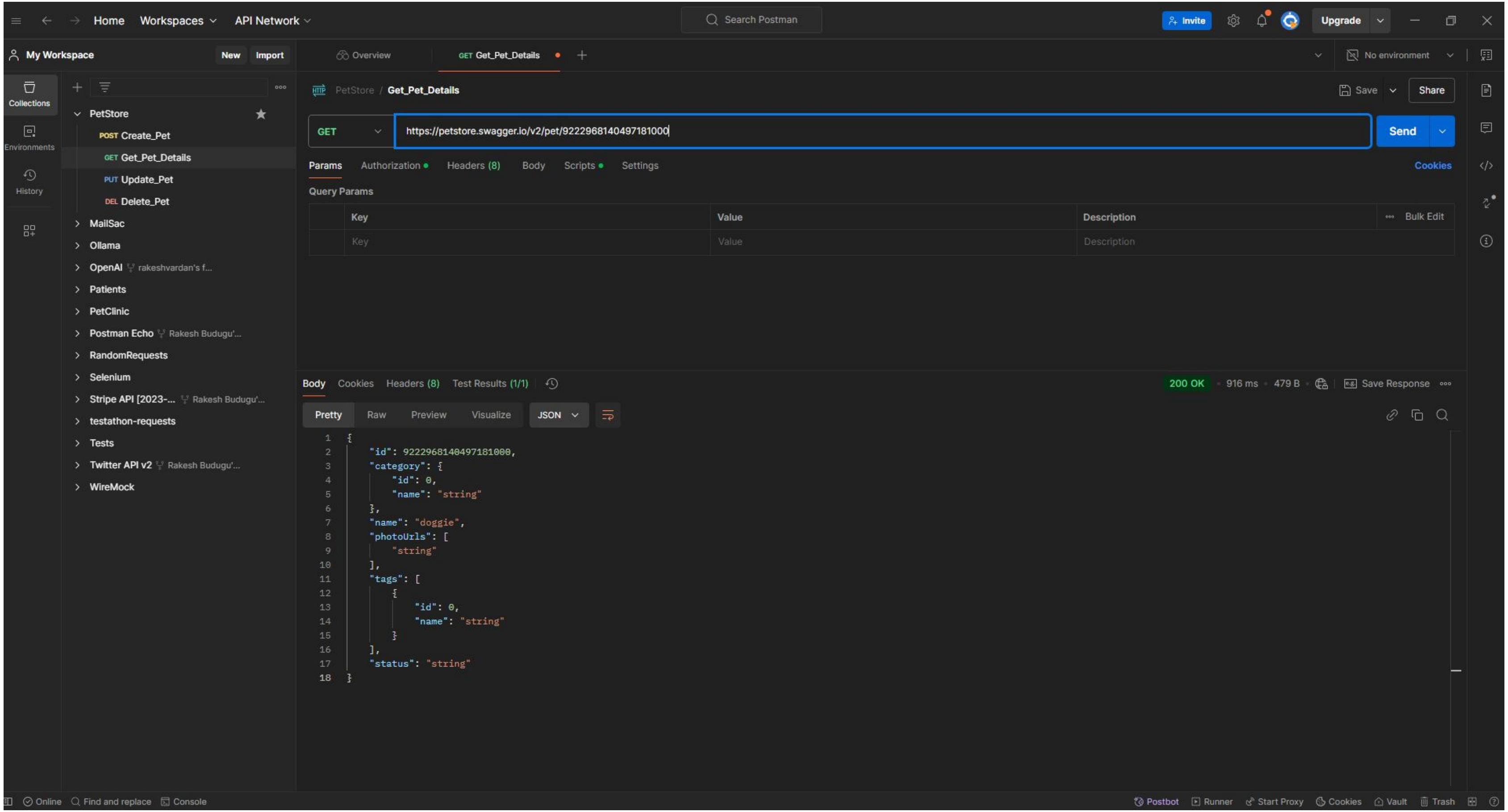
Installing Postman

1. **Visit the official website**: https://www.postman.com/
2. **Choose your platform**: Windows, macOS, or Linux.
3. **Install the application**: Follow the installer prompts.
4. **Sign in or create an account** (optional for collaboration features).

Overview of the Postman Interface

1. **Workspace**: Central dashboard for managing requests and collections.
2. **Request Builder**:
    a. Enter the HTTP method, URL, headers, and body for API requests.
3. **Response Viewer**: Displays the server response, including status codes, headers, and body.
4. **Collections Tab**: Manage grouped API requests.
5. **History Tab**: Track previously executed requests.
6. **Environment and Variables**: Manage data for dynamic API testing.

Overview of the Postman Interface



GET https://petstore.swagger.io/v2/pet/9222968140497181000

Key	Value	Description
Key	Value	Description

200 OK · 916 ms · 479 B

```
{
    "id": 9222968140497181000,
    "category": {
        "id": 0,
        "name": "string"
    },
    "name": "doggie",
    "photoUrls": [
        "string"
    ],
    "tags": [
        {
            "id": 0,
            "name": "string"
        }
    ],
    "status": "string"
}
```

API Testing with Postman

Understanding the Petstore API

- **API Documentation**: Swagger Petstore API (https://petstore.swagger.io/)
- **Base URL**: https://petstore.swagger.io/v2
- **Endpoints**:
    1. GET /pet/{petId}: Retrieve pet details.
    2. POST /pet: Add a new pet.
    3. PUT /pet: Update pet details.
    4. DELETE /pet/{petId}: Delete a pet.

Performing CRUD Operations in Postman - GET

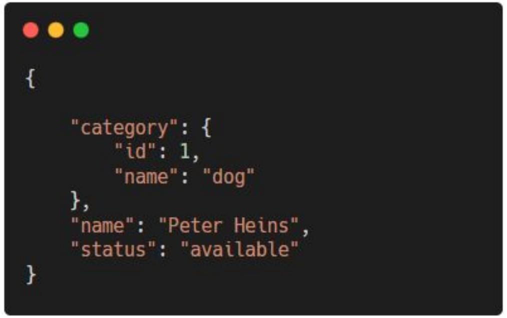
1. GET Request: Retrieve Pet Details

- **Endpoint**: GET /pet/{petId}
- **Steps**:
  1. Open Postman, create a new request, and set the method to GET.
  2. Enter the URL: https://petstore.swagger.io/v2/pet/1.
  3. Click **Send**.
- **Expected Response**: Status code 200 OK and return the pet data.

Performing CRUD Operations in Postman - POST

2. POST Request: Add a New Pet

- **Endpoint**: POST /pet
- **Steps**:
  1. Create a new request, and set the method to POST.
  2. Enter the URL: https://petstore.swagger.io/v2/pet.
  3. Go to the **Body** tab, select **raw**, and set the format to JSON.
  4. Enter the payload:



```
{

    "category": {
        "id": 1,
        "name": "dog"
    },
    "name": "Peter Heins",
    "status": "available"
}
```

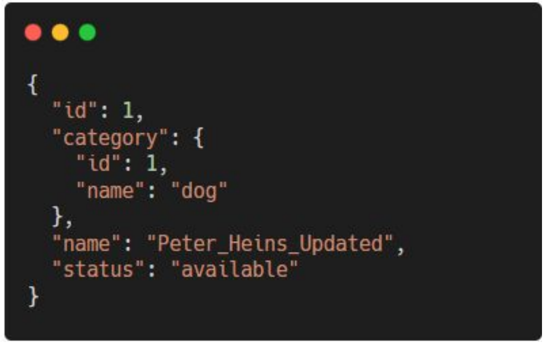
5. Click **Send**.

Expected Response: Status code 200 OK and the new pet data.

Performing CRUD Operations in Postman - PUT

3. PUT Request: Update Pet Details

- **Endpoint**: PUT /pet
- **Steps**:
  1. Create a new request, and set the method to PUT.
  2. Enter the URL: https://petstore.swagger.io/v2/pet.
  3. In the **Body** tab, provide updated data:



```
{
  "id": 1,
  "category": {
    "id": 1,
    "name": "dog"
  },
  "name": "Peter_Heins_Updated",
  "status": "available"
}
```

  4. Click **Send**.
- **Expected Response**: Status code 200 OK and updated pet details.

Performing CRUD Operations in Postman - DELETE

4. DELETE Request: Delete a Pet

- **Endpoint**: DELETE /pet/{petId}
- **Steps**:
  1. Create a new request, and set the method to DELETE.
  2. Enter the URL: https://petstore.swagger.io/v2/pet/2.
  3. Click **Send**.
- **Expected Response**: Status code 200 OK and confirmation message.

Validating API Responses

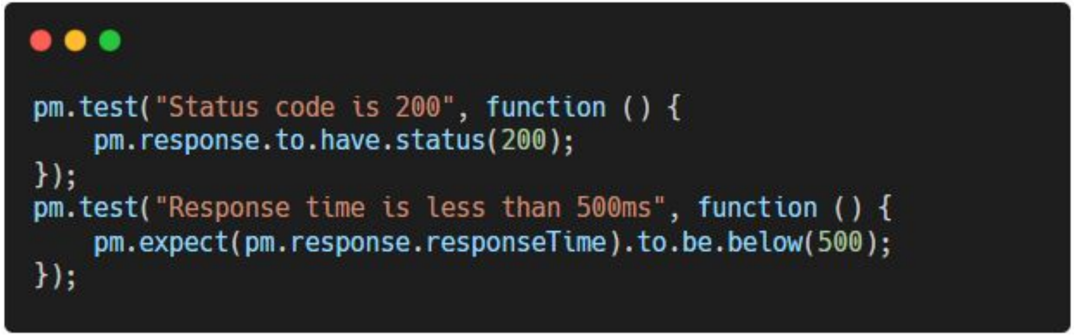
Key Elements to Verify:

1. **Status Codes**:
   - 200 OK: Success.
   - 404 Not Found: Resource doesn’t exist.
   - 500 Internal Server Error: Server issue.
2. **Response Time**: Ensure responses are within acceptable limits (e.g., <500ms).
3. **Response Body**:
   - Verify the structure and values of the response.
   - Example: Check if "status": "available" for the GET request.
4. **Headers**: Ensure correct content type (application/json).

Adding Test Scripts

Example Script: Validate Status Code and Response Time

1. Go to the **Scripts** tab in Postman.
2. Add this script in the **Post-Response** tab:



```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Response time is less than 500ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(500);
});
```

3. Click **Send** to execute and verify.

Saving Requests into a Collection

Why Save?

- Organize related requests for reuse.
- Run collections as a batch (e.g., using Newman).

Steps:

1. Create a new collection in Postman.
2. Save each CRUD request into the collection.

Key Takeaways

- Postman simplifies manual and automated API testing.
- CRUD operations are fundamental to API testing workflows.
- Test scripts and collections enhance productivity and collaboration.

Advanced Postman Concepts

Step 1: Using Variables in Postman

What are Variables?

- **Definition**: Placeholders for dynamic data like API keys, base URLs, or test data.
- **Types of Variables**:
  1. **Global Variables**: Accessible across all workspaces.
  2. **Environment Variables**: Tied to specific environments (e.g., dev, test, prod).
  3. **Collection Variables**: Specific to a collection of requests.
  4. **Local Variables**: Specific to a single request.

How to Use Variables

1. **Define Variables**:
   - Navigate to **Environment** > Add a new environment.
   - Add key-value pairs (e.g., `baseUrl = https://petstore.swagger.io/v2`).
2. **Use Variables in Requests**:
   - Replace static values with `{{variableName}}`.
   - Example:
     - URL: `{{baseUrl}}/pet/1`.

Advanced Postman Concepts Contd.

Step 2: Organizing Requests with Collections

What are Collections?

- Group related requests into a single container for better organization.
- Benefits:
  - Manage test cases easily.
  - Enable reusability and automation.

Creating and Using Collections

1. **Create a New Collection**:
   - Click on **Collections** > New Collection.
2. **Add Requests**:
   - Drag and drop existing requests into the collection.
3. **Folder Structure**:
   - Organize requests into folders within collections for modular testing.

Advanced Postman Concepts Contd.

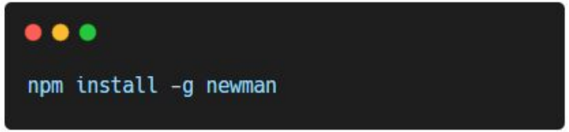
Step 3: Automating Tests with Newman

What is Newman?

- A command-line tool to run Postman collections.
- Ideal for integrating Postman tests into CI/CD pipelines.

Installing Newman

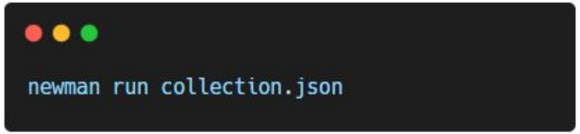
Install via Node.js:



```
npm install -g newman
```

Running a Collection

1. Export your Postman collection (.json file).
2. Run the collection with Newman:



```
newman run collection.json
```

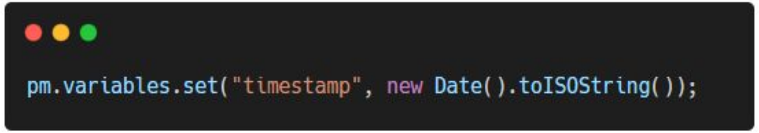
Advanced Postman Concepts Contd.

Step 4: Writing Pre-request and Test Scripts

Pre-request Scripts

- Scripts executed **before sending a request**.

Example: Adding a timestamp to the request:

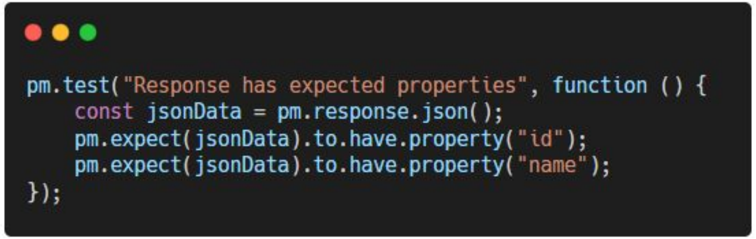


```
pm.variables.set("timestamp", new Date().toISOString());
```

Post-Response Scripts

- Scripts executed **after receiving the response**.

Example: Validate a JSON response structure:



```
pm.test("Response has expected properties", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property("id");
    pm.expect(jsonData).to.have.property("name");
});
```

Introduction to REST Assured

What is REST Assured?

- A **Java-based library** for automating REST API testing.
- Simplifies testing with a fluent interface for making API requests and validating responses.
- Works seamlessly with testing frameworks like **JUnit** and **TestNG**.

Why Use REST Assured?

- Automates API testing directly in code.
- Provides powerful features for:
  - Validating status codes, response bodies, headers, and cookies.
  - Supporting different authentication mechanisms (Basic, Token, OAuth).
- Ideal for integrating API tests into CI/CD pipelines.

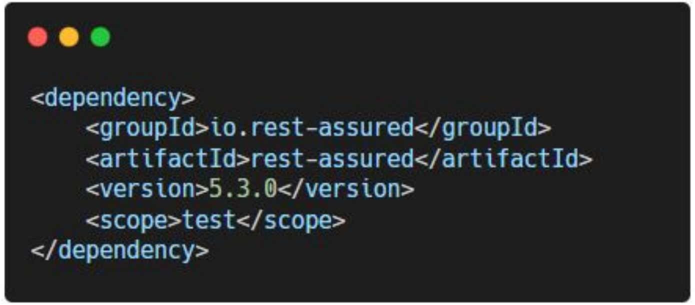
Setting Up REST Assured

Prerequisites:

1. **Install Java**: Ensure JDK is installed.
2. **Set Up a Java IDE**: Use IntelliJ IDEA, Eclipse, or VS Code.
3. **Install a Build Tool**: Use Maven or Gradle for dependency management.

Adding REST Assured Dependency:

For **Maven**: Add the following to pom.xml:

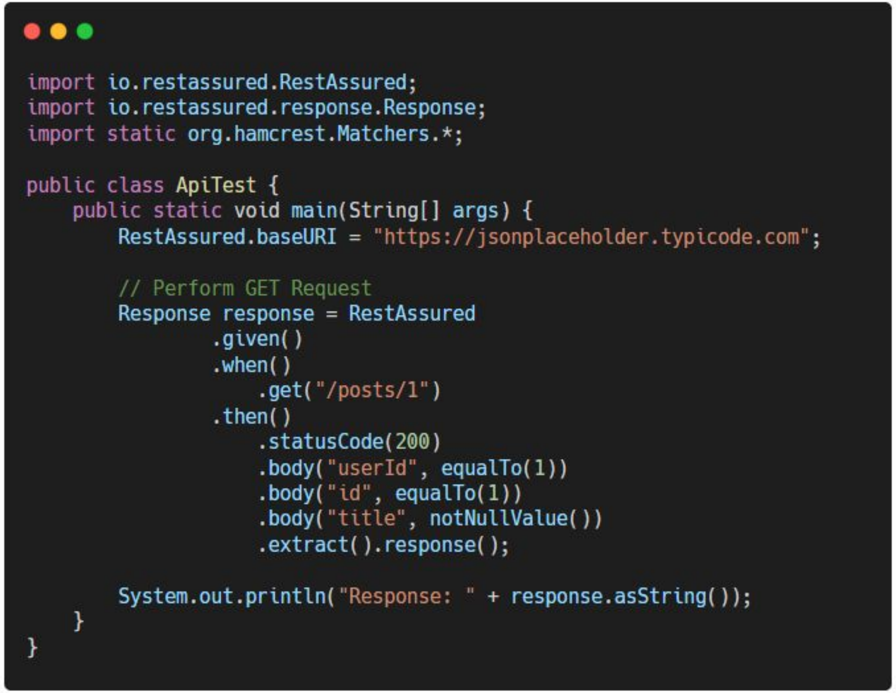


```xml
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>5.3.0</version>
    <scope>test</scope>
</dependency>
```

Basic REST Assured Test Example

Scenario: Verify a GET request on a sample API

Code Example



```java
import io.restassured.RestAssured;
import io.restassured.response.Response;
import static org.hamcrest.Matchers.*;

public class ApiTest {
    public static void main(String[] args) {
        RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

        // Perform GET Request
        Response response = RestAssured
                .given()
                .when()
                    .get("/posts/1")
                .then()
                    .statusCode(200)
                    .body("userId", equalTo(1))
                    .body("id", equalTo(1))
                    .body("title", notNullValue())
                    .extract().response();

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

Explanation

- **Base URI**: Sets the root of the API (https://jsonplaceholder.typicode.com).
- **GET Request**: Retrieves the resource at /posts/1.
- **Assertions**:
    - Validates the status code is 200.
    - Checks the userId and id fields.
    - Ensures the title is not null.

Advantages of REST Assured

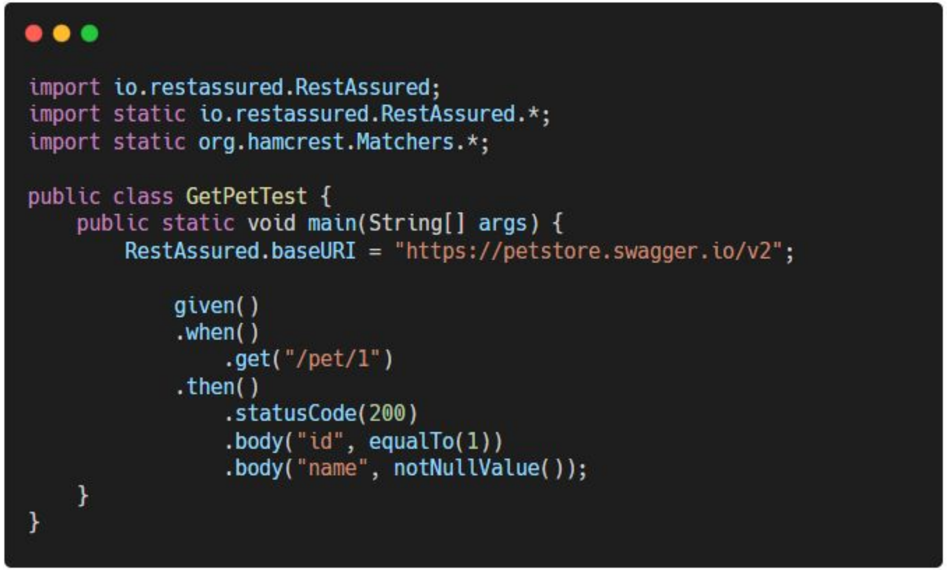
1. **Ease of Use**: Write concise tests with a readable syntax.
2. **Seamless Validation**: Directly validate status codes, response times, and body content.
3. **Authentication Support**: Test APIs with various authentication methods.
4. **Integration**: Combine with JUnit/TestNG for structured testing and reporting.

API Testing with REST Assured

Performing CRUD Operations

1. GET Request: Retrieve Pet Details

Code Example



```java
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class GetPetTest {
    public static void main(String[] args) {
        RestAssured.baseURI = "https://petstore.swagger.io/v2";

            given()
            .when()
                .get("/pet/1")
            .then()
                .statusCode(200)
                .body("id", equalTo(1))
                .body("name", notNullValue());
    }
}
```

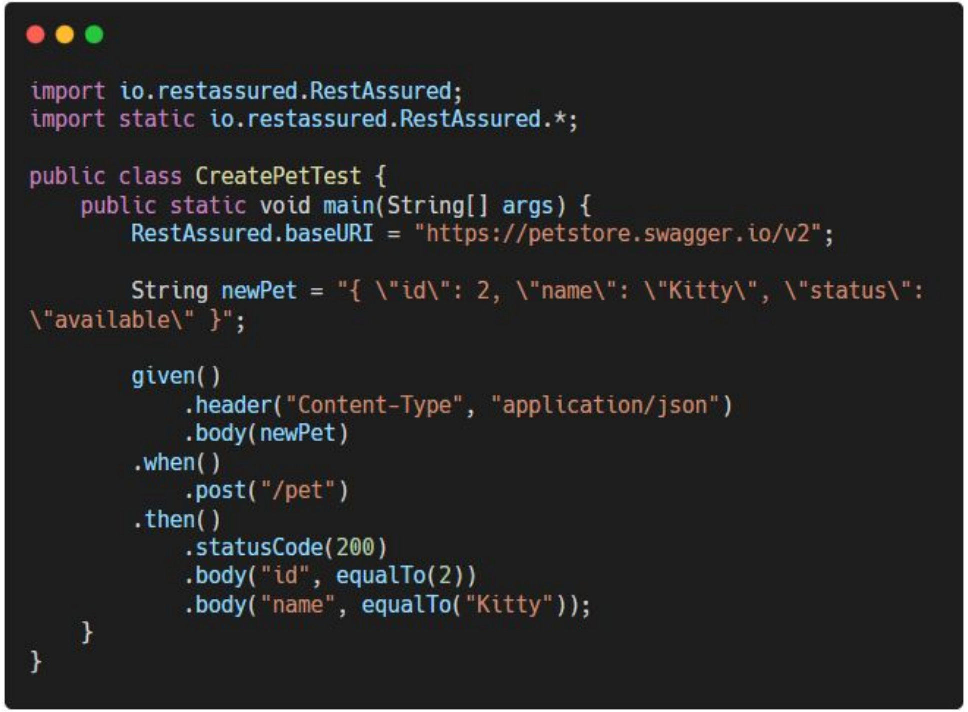
Explanation

- **given()**: Sets up the request.
- **when()**: Sends the GET request.
- **then()**: Validates the response.

API Testing with REST Assured Contd.

2. POST Request: Add a New Pet

Code Example



```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class CreatePetTest {
    public static void main(String[] args) {
        RestAssured.baseURI = "https://petstore.swagger.io/v2";

        String newPet = "{ \"id\": 2, \"name\": \"Kitty\", \"status\":
\"available\" }";

        given()
            .header("Content-Type", "application/json")
            .body(newPet)
        .when()
            .post("/pet")
        .then()
            .statusCode(200)
            .body("id", equalTo(2))
            .body("name", equalTo("Kitty"));
    }
}
```

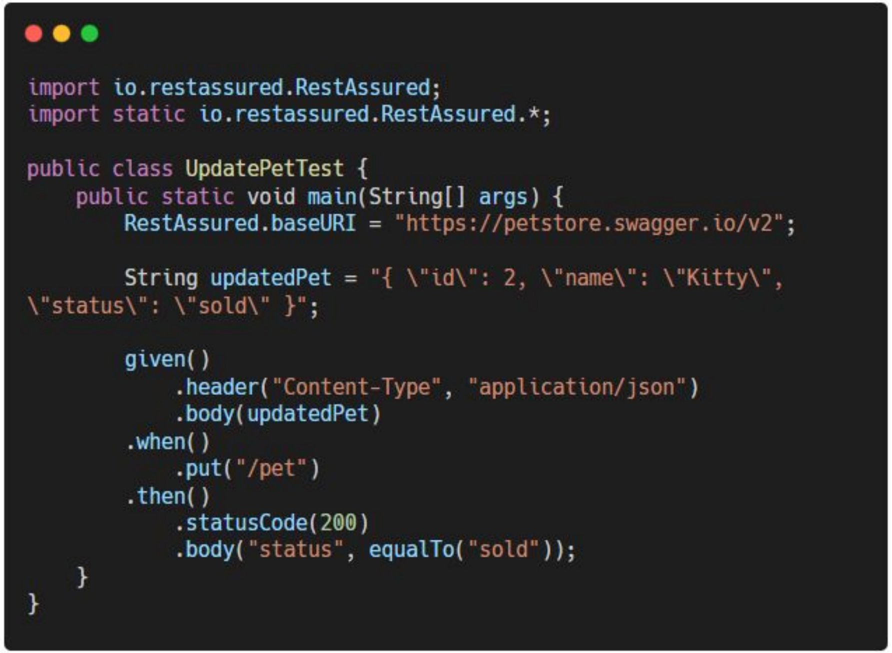
Explanation

- **body()**: Sets the payload for the POST request.
- **header()**: Defines the Content-Type as JSON.

API Testing with REST Assured Contd.

3. PUT Request: Update Pet Details

Code Example



```java
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class UpdatePetTest {
    public static void main(String[] args) {
        RestAssured.baseURI = "https://petstore.swagger.io/v2";

        String updatedPet = "{ \"id\": 2, \"name\": \"Kitty\",
\"status\": \"sold\" }";

        given()
            .header("Content-Type", "application/json")
            .body(updatedPet)
        .when()
            .put("/pet")
        .then()
            .statusCode(200)
            .body("status", equalTo("sold"));
    }
}
```

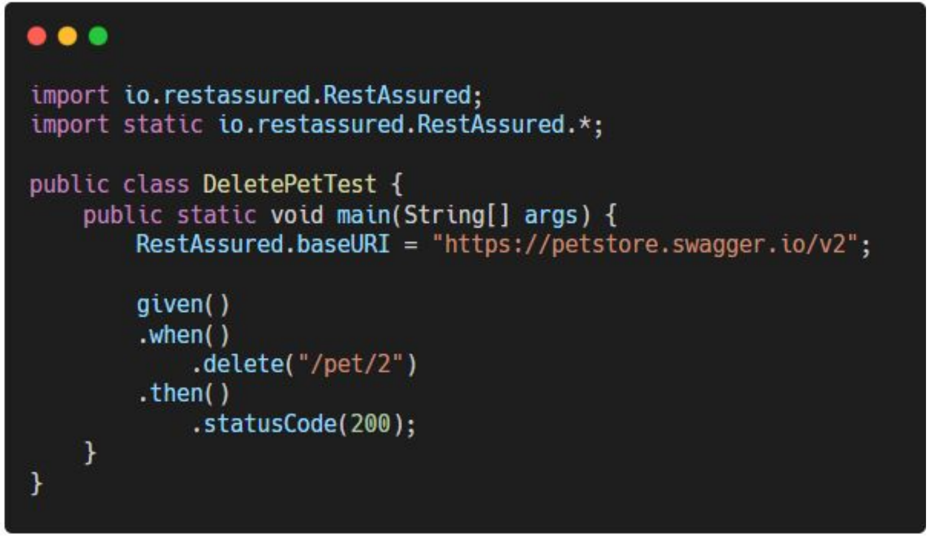
Explanation

- Updates the status field of the existing pet.
- Validates the new status in the response body.

API Testing with REST Assured Contd.

4. DELETE Request: Delete a Pet

Code Example



```java
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class DeletePetTest {
    public static void main(String[] args) {
        RestAssured.baseURI = "https://petstore.swagger.io/v2";

        given()
        .when()
            .delete("/pet/2")
        .then()
            .statusCode(200);
    }
}
```

Explanation

- Deletes the pet with ID 2.
- Verifies the response status code is 200.

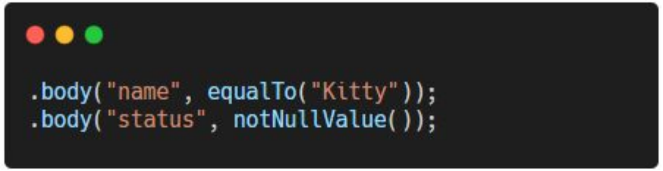
Key Validations

Status Code Validation

- Always verify the expected status code.
- Example: **statusCode(200).**

Response Body Validation

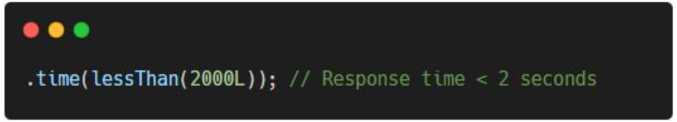
- Use **Hamcrest matchers** to validate response fields.
- Example:



```
.body("name", equalTo("Kitty"));
.body("status", notNullValue());
```

Response Time Validation

- Ensure the API response time is within acceptable limits.
- Example:

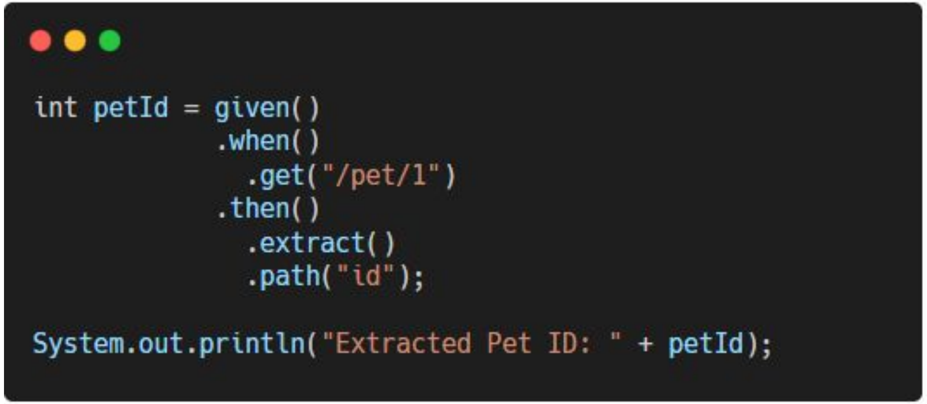


```
.time(lessThan(2000L)); // Response time < 2 seconds
```

Extracting Data from Responses

REST Assured allows extracting data for further validations or usage.

Example: Extract the id from a response:



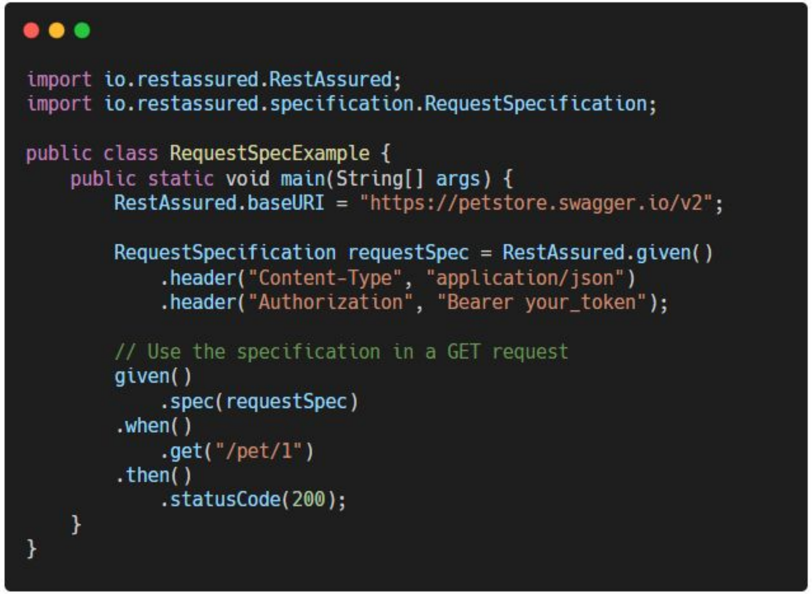
```
int petId = given()
            .when()
              .get("/pet/1")
            .then()
              .extract()
              .path("id");

System.out.println("Extracted Pet ID: " + petId);
```

Request and Response Specifications

Why Use Specifications?

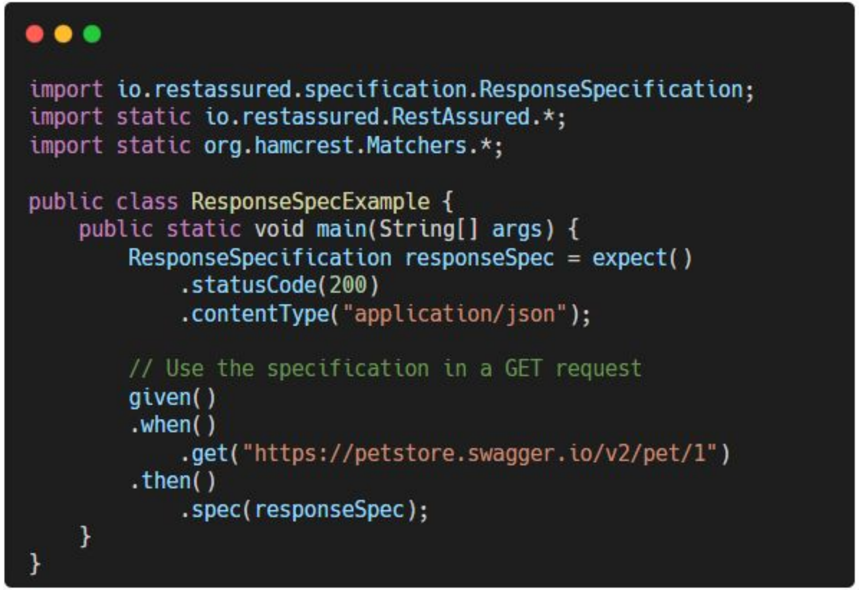
- Reduce redundancy by defining common properties for requests or responses.
- Improve test readability and maintainability.

Creating Request Specification



```java
import io.restassured.RestAssured;
import io.restassured.specification.RequestSpecification;

public class RequestSpecExample {
    public static void main(String[] args) {
        RestAssured.baseURI = "https://petstore.swagger.io/v2";

        RequestSpecification requestSpec = RestAssured.given()
            .header("Content-Type", "application/json")
            .header("Authorization", "Bearer your_token");

        // Use the specification in a GET request
        given()
            .spec(requestSpec)
        .when()
            .get("/pet/1")
        .then()
            .statusCode(200);
    }
}
```

Creating Response Specification



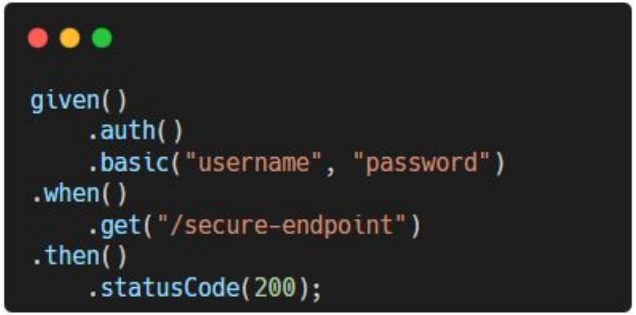
```java
import io.restassured.specification.ResponseSpecification;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class ResponseSpecExample {
    public static void main(String[] args) {
        ResponseSpecification responseSpec = expect()
            .statusCode(200)
            .contentType("application/json");

        // Use the specification in a GET request
        given()
        .when()
            .get("https://petstore.swagger.io/v2/pet/1")
        .then()
            .spec(responseSpec);
    }
}
```

Advanced Concepts: Authentication Mechanisms

1. Basic Authentication

- Add a username and password to the request.

Code Example

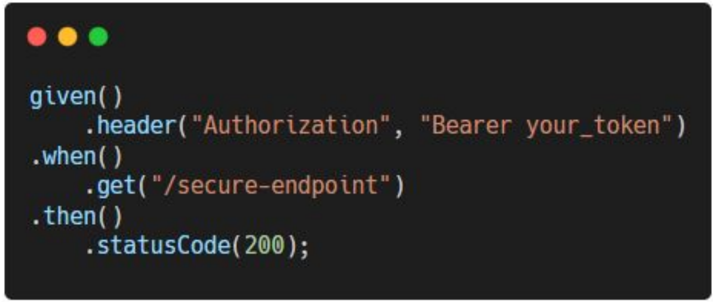


```
given()
    .auth()
    .basic("username", "password")
.when()
    .get("/secure-endpoint")
.then()
    .statusCode(200);
```

2. Token-Based Authentication

- Pass an access token in the headers.

Code Example



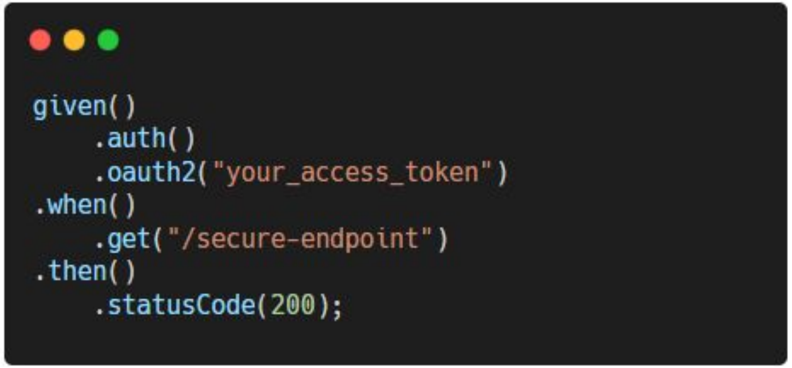
```
given()
    .header("Authorization", "Bearer your_token")
.when()
    .get("/secure-endpoint")
.then()
    .statusCode(200);
```

Advanced Concepts: Authentication Mechanisms Contd.

3. OAuth 2.0

- Use REST Assured’s OAuth support for token-based APIs.

Code Example

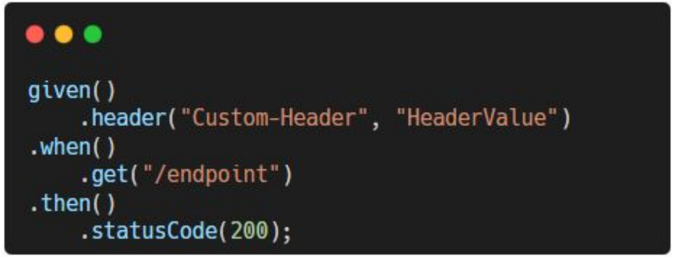


```
given()
    .auth()
    .oauth2("your_access_token")
.when()
    .get("/secure-endpoint")
.then()
    .statusCode(200);
```

Advanced Concepts: Handling Headers and Cookies

Adding Custom Headers

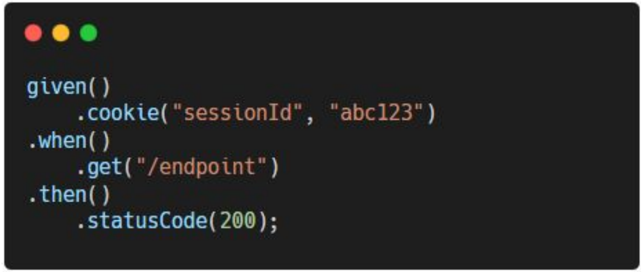
Code Example



```
given()
    .header("Custom-Header", "HeaderValue")
.when()
    .get("/endpoint")
.then()
    .statusCode(200);
```

Handling Cookies

Code Example



```
given()
    .cookie("sessionId", "abc123")
.when()
    .get("/endpoint")
.then()
    .statusCode(200);
```

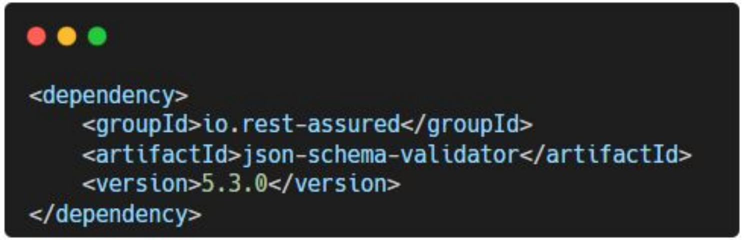
Advanced Concepts: Schema Validation

Why Validate Schema?

- Ensure the API response adheres to the expected structure.
- Use **JSON Schema** or **XML Schema** for validation.

Validating JSON Schema

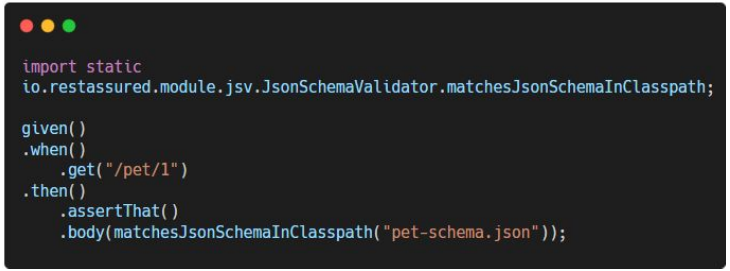
1. Add the **json-schema-validator** dependency:



```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>json-schema-validator</artifactId>
    <version>5.3.0</version>
</dependency>
```

2. Validate the schema in your test:

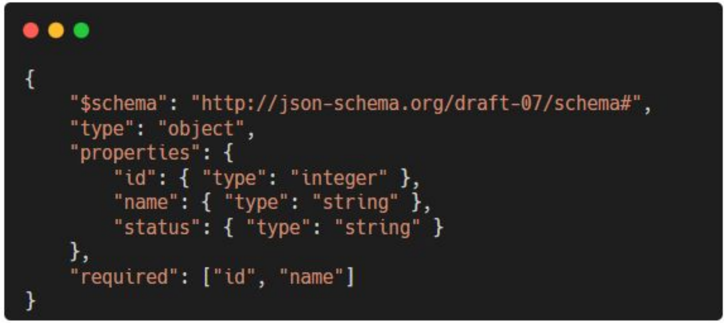
Code Example



```
import static
io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchemaInClasspath;

given()
.when()
    .get("/pet/1")
.then()
    .assertThat()
    .body(matchesJsonSchemaInClasspath("pet-schema.json"));
```

Creating a Schema File

- Example of a JSON schema (pet-schema.json):



```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties": {
        "id": { "type": "integer" },
        "name": { "type": "string" },
        "status": { "type": "string" }
    },
    "required": ["id", "name"]
}
```

Best Practices for API Testing

1. Understand the API

- Review API documentation thoroughly before writing tests.
- Understand endpoints, request payloads, and expected responses.
- Examples of documentation tools: **Swagger**, **Postman API Docs**.

2. Use Proper Test Design

- Categorize tests into:
  - **Positive Tests**: Validate correct inputs.
  - **Negative Tests**: Validate invalid inputs.
  - **Boundary Tests**: Test edge cases for parameters (e.g., min/max values).
- Avoid testing everything in one test case; use smaller, atomic tests.

3. Validate All Response Aspects

- **Status Codes**: Ensure appropriate codes (e.g., 200 OK, 400 Bad Request).
- **Headers**: Verify content type, caching directives, etc.
- **Response Body**: Validate fields, data types, and values.
- **Response Time**: Ensure performance criteria are met (e.g., < 2 seconds).

Best Practices for API Testing Contd.

4. Automate Tests Effectively

- Integrate API tests into your **CI/CD pipelines** for frequent test execution.
- Use tools like Jenkins, GitHub Actions, or GitLab CI for automation.

5. Use Data-Driven Testing

- Test with multiple input sets using external data sources (e.g., CSV, JSON, databases).

6. Mock APIs for Early Testing

- Use mocking tools like **WireMock** or **MockServer** to test APIs under development.
- Create mock responses for endpoints that are not yet available.

7. Handle Rate Limits and Timeouts

- Simulate real-world scenarios like high traffic or delayed responses.
- Respect API rate limits during tests to avoid being blocked.

Best Practices for API Testing Contd.

8. Secure Sensitive Information

- Avoid hardcoding sensitive information (e.g., tokens, passwords).
- Use environment variables or encrypted configuration files.
- Example in Postman:
    - Use **Environment Variables** for tokens.
    - Access via `{{variable_name}}`.

9 . Test APIs Independently

- Minimize dependencies between API tests.
- Mock dependent services if necessary.

Best Practices for API Testing Contd.

10. Log Detailed Test Results

- Log requests, responses, and assertion failures for debugging.

11. Version Control Your Tests

- Store test scripts in a version control system like Git.
- Use branching strategies for collaborative development.

12. Keep Tests Up-to-Date

- Update test cases to reflect API changes.
- Regularly review and remove redundant or outdated tests.