End to End API Testing & Interview Questions Notes

End to End API Testing, Automation API Test With Cucumber, Mock Interview Question for API Testing for Backend Engineer, SDET, Software Quality Engineer, Software Quality Assurance, Software Test Engineer, and Test Engineer

Curated by Lamhot Siagian

PART 1:

Version 1.0

Published date: 12/18/2024

End to End API Testing

Automation API Test With Cucumber

Mock Interview Question for API Testing

Contact Information:

Email: lamhot.id@gmail.com
Site: https://lamhotjm.github.io
Linkedin: https://www.linkedin.com/in/lamhotsiagian
Tips or Buy me a coffee: https://buymeacoffee.com/lamhot

TABLE OF CONTENTS

Preface

Welcome to *End to End API Testing & Interview Questions Notes*, a comprehensive guide designed to assist you in mastering API testing and preparing for technical interviews in API testing roles. With the increasing reliance on APIs in modern software architectures, understanding how to test APIs is becoming more crucial than ever for QA professionals and software testers.

This book is structured to provide both practical knowledge and insights into real-world API testing scenarios. The chapters are meticulously organized to take you from the basics of API testing to advanced topics such as security, performance testing, and automation. Here's a brief overview of the chapters:

1. **Introduction to APIs and API Testing**
   Understand the fundamentals of APIs and the significance of testing in ensuring the reliability and security of software applications.
2. **Setting Up Your API Testing Environment**
   Learn how to configure and prepare your environment for efficient API testing, focusing on tools and technologies.
3. **Understanding API Requests and Responses**
   Gain a detailed understanding of API communication, including request methods, headers, parameters, and response structures.
4. **Security Testing of APIs**
   Explore key concepts in API security testing, such as authentication, authorization, and vulnerability detection.
5. **Writing Basic API Test Cases**
   Start writing your first API test cases, including functional and negative test cases.
6. **Automating API Tests**
   Delve into automating API test cases with tools such as Cucumber and Maven, enabling fast and consistent test execution.
7. **Performance Testing of APIs using JMeter**
   Learn how to evaluate the performance and load capacity of your APIs using tools like JMeter.
8. **Organizing a Test Framework for API Testing**
   Discover how to structure a scalable and maintainable API test framework, with practical tips on test organization and data management.
9. **Continuous Integration and Continuous Deployment (CI/CD) with API Testing**
   Explore how to integrate API testing into a CI/CD pipeline for rapid development cycles, using tools such as GitHub Actions and Jenkins.
10. **Test Driven Development for Spring Microservices with Cucumber**
    Get hands-on experience with TDD for microservices, learning how to write tests for Spring-based APIs using Cucumber.

Interview Preparation

Each chapter contains a set of interview questions and sample answers, helping you prepare for API testing roles in Backend Engineering, SDET (Software Development Engineer in Test), Software Quality Assurance, and Test Engineering positions. These questions are inspired by my own experiences from years of technical interviews in the United States and other resources.

Over the past seven years, I've meticulously gathered notes and questions from each interview, summarizing them into this guide. Whether you are a fresh graduate or an experienced tester, these notes will help you solidify your understanding of key concepts while giving you the confidence to face any API testing interview.

Who This Notes is For

This book is tailored for:

- End-to-end API testers seeking practical insights and automation tips
- QA professionals looking to enhance their API testing skills with Cucumber and Java
- Candidates preparing for backend testing roles, including SDET, Software Quality Engineers, and Test Engineers
- Individuals seeking mock interview questions and answers to prepare for technical interviews

API testing has been increasingly recognized for its efficiency, flexibility, and integration capabilities, making it a vital part of any software testing strategy. I hope this book helps you gain a strong foundation in API testing, and I trust that the included interview questions and answers will be a valuable resource for your career progression.

Happy learning, and good luck with your API testing journey!

Lamhot Siagian

Software Engineer in Test Consultant

Chapter 1

Introduction to APIs and API Testing

Understanding APIs

APIs (Application Programming Interfaces) is fundamental for modern software development, as they enable different software systems to communicate with each other. Here's a detailed overview to help you understand what APIs are, how they work, and why they are important.

What is an API?

An API, or application programming interface, is a set of rules or protocols that enables software applications to communicate with each other to exchange data, features and functionality..

How Do APIs Work?

It's useful to think about API communication in terms of a request and response between a client and server. The application submitting the request is the client, and the server provides the response. The API is the bridge establishing the connection between them.

A simple way to understand how APIs work is to look at a common example—third-party payment processing. When a user purchases a product on an e-commerce site, the site might prompt the user to "Pay with PayPal" or another type of third-party system. This function relies on APIs to make the connection.

- When the buyer clicks the payment button, an API call is sent to retrieve information. This is the request. This request is processed from an application to the web server through the API's Uniform Resource Identifier (URI) and includes a request verb, headers, and sometimes, a request body.
- After receiving a valid request from the product webpage, the API calls to the external program or web server, in this case, the third-party payment system.
- The server sends a response to the API with the requested information.
- The API transfers the data to the initial requesting application, in this case, the product website [1]

Components of an API

1. **Endpoints**: URLs that provide access to specific resources or functionalities.
2. **Methods/HTTP Verbs**: Actions that can be performed (e.g., GET, POST, PUT, DELETE).
3. **Requests**: Data sent by the client to the server.
4. **Responses**: Data sent back by the server to the client.
5. **Headers**: Metadata about the request or response (e.g., authentication tokens, content type).

[1] https://www.ibm.com/topics/api

6. **Payload/Body**: Data being sent with the request (typically in JSON or XML format).

Benefits of Using APIs

APIs offer significant benefits, including:

1. **Automation**: Streamlines repetitive tasks, boosting productivity.
2. **Innovation**: Enables external teams to innovate by leveraging existing functionalities.
3. **Security**: Adds a layer of protection by requiring authentication and authorization.
4. **Cost Efficiency**: Reduces expenses by using third-party tools instead of developing in-house systems.[2]

API Use Cases

1. **Integration**: Connecting different applications and services, such as integrating payment gateways into e-commerce platforms.
2. **Automation**: Automating repetitive tasks, such as deploying code, managing resources, or data synchronization.
3. **Data Access**: Providing access to data from different sources, such as retrieving weather data or financial information.
4. **Third-Party Services**: Enabling third-party developers to build applications that interact with your services, such as social media apps using Facebook or Twitter APIs.
5. **Microservices Architecture**: Facilitating communication between microservices in a distributed system.

API Documentation

Good API documentation is essential for developers to understand how to use the API effectively. It typically includes:

1. **Endpoint Descriptions**: Detailed information about available endpoints and their purposes.
2. **HTTP Methods**: The methods supported by each endpoint (e.g., GET, POST).
3. **Parameters**: Required and optional parameters for each endpoint.
4. **Request and Response Formats**: Examples of request payloads and expected responses.
5. **Authentication**: Information about authentication and authorization mechanisms.
6. **Error Handling**: Common error codes and messages, along with troubleshooting information.

[2] https://www.postman.com/what-is-an-api/#benefits-of-apis

Importance of API Testing

API testing is a crucial aspect of software development, offering numerous benefits and addressing key challenges that can significantly impact the quality, reliability, and performance of software applications. Here are the primary reasons why API testing is important:

1. Ensures Functionality

- **Verification of Requirements**: API testing ensures that the API functions according to the specified requirements. This involves checking that the endpoints are working correctly, the data is being processed as expected, and the correct responses are returned.
- **Detection of Errors Early**: By testing APIs early in the development cycle, errors can be detected and fixed before they become more difficult and costly to resolve.

2. Improves Reliability

- **Consistent Performance**: Regular API testing helps ensure that the API performs consistently under various conditions, reducing the likelihood of unexpected failures in production environments.
- **Regression Testing**: API tests can be automated and included in regression testing to ensure that new changes do not break existing functionality.

3. Enhances Security

- **Identification of Vulnerabilities**: API testing can reveal security vulnerabilities such as SQL injection, cross-site scripting (XSS), and other common exploits.
- **Validation of Authentication and Authorization**: Ensures that security mechanisms like authentication and authorization are working correctly, protecting sensitive data and functionality from unauthorized access.

4. Boosts Performance

- **Load and Stress Testing**: API testing includes performance tests such as load and stress testing, which help determine how well the API performs under heavy traffic and identify potential bottlenecks.
- **Scalability Assessment**: Helps in assessing whether the API can scale effectively to handle increased load as the user base grows.

5. Enhances User Experience

- **Ensures Smooth Integration**: For APIs that are consumed by external developers or third-party applications, thorough testing ensures that these integrations work smoothly, providing a better user experience.
- **Reduces Downtime**: By identifying and resolving issues early, API testing helps reduce the chances of downtime, ensuring that end-users experience fewer disruptions.

6. Facilitates Continuous Integration/Continuous Deployment (CI/CD)

- **Automated Testing**: API tests can be automated and integrated into CI/CD pipelines, providing quick feedback to developers and ensuring that changes do not introduce new issues.
- **Continuous Monitoring**: Ongoing API testing helps monitor the API's health and performance continuously, allowing for rapid response to issues.

7. Supports Development and Debugging

- **Improved Debugging**: API tests provide detailed information about failures, making it easier for developers to debug and resolve issues.
- **Documentation and Clarity**: Writing API tests often helps clarify the API's functionality and expected behavior, which can improve the quality of documentation and assist other developers in understanding how to use the API.

8. Reduces Costs

- **Lower Maintenance Costs**: Identifying and fixing issues early in the development process reduces the costs associated with post-release maintenance and support.
- **Fewer Production Issues**: Well-tested APIs are less likely to cause issues in production, leading to reduced costs related to bug fixes and customer support.

9. Ensures Compliance and Standardization

- **Adherence to Standards**: API testing ensures that the API complies with industry standards and protocols, which is particularly important for APIs in regulated industries.
- **Contract Testing**: Validates that the API contracts (the agreed-upon schema and behavior between API consumers and providers) are maintained, ensuring compatibility and reliability.

Types of APIs

APIs come in various forms, each with its own strengths and purposes. Understanding these differences is essential for picking the right API for your project and ensuring your application is effective, scalable, and reliable. Let's look at the different types of APIs, exploring what makes each unique and how they fit into modern software development.

This table provides a high-level overview of various API types, their descriptions, and examples.[3]

API type	Description	Examples
Library-based APIs	Part of software libraries, language-specific, range from basic utilities to complex GUI components	Java API for Android app development, .NET libraries used in Windows applications
Operating system APIs	Provide interfaces for OS interaction, manage hardware resources and processes	Windows API for Windows OS interaction, POSIX API for UNIX-like systems
Database APIs	Enable interaction with database management systems, facilitate data querying and manipulation	SQL API for relational databases, Oracle's OCI for Oracle databases
Hardware APIs	Allow communication with hardware devices, direct control over hardware functions	IoT device APIs for smart home systems, Printer APIs for document processing
Cloud APIs	Provided by cloud service platforms, enable interaction with cloud-based resources and services	Amazon Web Services (AWS) API, Microsoft Azure API for cloud-based solutions
HTTP APIs (web API)	Facilitate client-server communication over the web, use standard HTTP methods, lightweight and flexible	Web services, mobile apps, IoT devices, social media platforms, content management systems
REST APIs (web API)	Use HTTP requests for data operations, stateless and separate client-server concerns	Web services accessible via the web, social media APIs, cloud services
SOAP APIs (web API)	Use service interfaces to expose business logic, high security, suitable for enterprise-level organizations	Enterprise-level services like banking, healthcare, where security and transactions are critical
GraphQL APIs (web API)	Clients request only needed data, reduce data transfer over the network, support multiple responses in one request	Complex data-driven web and mobile applications, ad-hoc queries by the client

[3] https://www.nylas.com/api-guide/types-of-apis/

Open APIs (public APIs)	Accessible by third-party developers, typically for external users, require API keys	Twitter API for tweets, Stripe API for payments, providing data or services to external developers
Internal APIs (private APIs)	Designed for internal use within an organization, enhance integration between systems	Linking HR systems to internal employee directories, improving internal efficiency and data security
Partner APIs	Available to strategic business partners, require specific entitlements	Integration of supply chain systems, service expansion to partners, creating revenue channels, controlled data sharing with trusted partners
Composite APIs	Combine different data and service APIs, access multiple endpoints in one call	Aggregating various services in a single call for smoother user experience, commonly used in microservices architectures
JSON-RPC and XML-RPC APIs	Encode data as JSON or XML for remote procedure calls, send lists of commands	Remote procedure calls with simple request/response model, suitable for basic request/response scenarios
Synchronous and asynchronous	Synchronous: Request-response model, Asynchronous: Non-blocking, no immediate response required	Synchronous: Essential data retrieval, web page loading, Asynchronous: Background tasks, real-time data streams
Library-based APIs	Part of software libraries, language-specific, range from basic utilities to complex GUI components	Java API for Android app development, .NET libraries used in Windows applications

Overview of API Testing Techniques

1. Unit Testing

- **Objective**: Test individual API components in isolation.
- **Tools**: JUnit (Java), NUnit (C#), pytest (Python).
- **Description**: Unit tests are typically written by developers to test the functionality of specific methods or functions in the API, ensuring that each part works as intended.

2. Functional Testing

- **Objective**: Verify that the API performs its intended functions correctly.
- **Tools**: Postman, SoapUI, REST Assured.
- **Description**: Functional tests validate the API against the functional requirements and specifications. This includes testing endpoints, methods (GET, POST, PUT, DELETE), and responses.

3. Integration Testing

- **Objective**: Ensure that the API interacts correctly with other components and systems.
- **Tools**: Postman, SoapUI, JUnit (with integration test configurations).
- **Description**: Integration tests evaluate the interactions between different parts of the API and other services or databases to ensure that integrated parts work together as expected.

4. Performance Testing

- **Objective**: Assess the API's performance under various conditions.
- **Tools**: JMeter, LoadRunner, Gatling.
- **Description**: Performance tests include load testing (to check API behavior under expected load), stress testing (to determine the API's breaking point), and endurance testing (to evaluate performance over an extended period).

5. Security Testing

- **Objective**: Identify vulnerabilities and ensure the API is secure.
- **Tools**: OWASP ZAP, Burp Suite, Postman (with security extensions).
- **Description**: Security testing involves checking for common vulnerabilities like SQL injection, cross-site scripting (XSS), and ensuring proper authentication and authorization mechanisms are in place.

6. Usability Testing

- **Objective**: Ensure the API is easy to use and well-documented.
- **Tools**: Swagger, Postman.
- **Description**: Usability testing focuses on the API's user experience, ensuring that the documentation is clear, the endpoints are intuitive, and error messages are helpful.

7. Validation Testing

- **Objective**: Validate the API's functionality, performance, and security comprehensively.
- **Tools**: Postman, SoapUI.
- **Description**: Validation testing ensures that the API meets the business requirements and expectations, combining functional, performance, and security testing aspects.

8. Compliance Testing

- **Objective**: Ensure the API complies with industry standards and regulations.
- **Tools**: Postman, SoapUI, custom scripts.
- **Description**: Compliance testing checks if the API adheres to legal and regulatory requirements, such as GDPR, HIPAA, and other industry-specific standards.

9. Mocking and Virtualization

- **Objective**: Test the API in isolated environments by simulating dependencies.
- **Tools**: WireMock, MockServer, Postman (mock servers).
- **Description**: Mocking and virtualization allow testers to simulate the behavior of the API's dependencies, enabling testing in scenarios where real dependencies are unavailable or difficult to configure.

10. Regression Testing

- **Objective**: Ensure new changes do not negatively affect existing functionality.
- **Tools**: Postman, SoapUI, automated test scripts.
- **Description**: Regression testing involves re-running previously conducted tests to verify that new code changes have not introduced any new bugs or issues.

Best Practices for API Testing

- **Automation**: Automate as many tests as possible to ensure efficiency and repeatability.
- **Version Control**: Keep track of API versions and ensure tests are updated accordingly.
- **Environment Management**: Use consistent environments for testing to avoid discrepancies.
- **Data Management**: Use realistic data for testing to uncover potential issues.
- **Continuous Integration/Continuous Deployment (CI/CD)**: Integrate API testing into the CI/CD pipeline to catch issues early.

Common Interview Questions and Answers Related to Basic APIs

1. What is an API?

Answer: An API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other. APIs define the methods and data formats that applications can use to request and exchange information.

2. Can you explain the difference between REST and SOAP APIs?

Answer:

- **REST (Representational State Transfer)**:
    - Uses standard HTTP methods (GET, POST, PUT, DELETE).
    - Stateless, meaning each request from a client to a server must contain all the information needed to understand and process the request.
    - Typically uses JSON or XML for data interchange.
    - It is easier to implement and more scalable for web services.
- **SOAP (Simple Object Access Protocol)**:
    - A protocol-based approach that relies on XML for message format.
    - Includes built-in error handling and supports more complex operations.
    - Uses stricter standards and can work over several protocols (HTTP, SMTP, TCP).
    - Generally more secure with built-in security features.

3. What are the main HTTP methods used in RESTful APIs and what are their purposes?

Answer:

- **GET**: Retrieve data from the server.
- **POST**: Submit data to the server to create a new resource.
- **PUT**: Update an existing resource on the server.
- **DELETE**: Remove a resource from the server.
- **PATCH**: Apply partial modifications to a resource.

4. What is an API endpoint?

Answer: An API endpoint is a specific URL that provides access to a particular resource or functionality of the API. It represents one of the discrete units of interaction in an API and is typically a combination of the base URL and a resource path.

5. Explain the concept of RESTful API statelessness.

Answer: Statelessness in RESTful APIs means that each request from a client to a server must contain all the information needed to understand and process the request. The server does not store any context or

session information about the client between requests, making each request independent and self-contained.

6. What is an API key and why is it used?

Answer: An API key is a unique identifier used to authenticate a client accessing an API. It helps track and control how the API is used, ensuring that only authorized users can make requests. API keys are commonly used for security purposes to prevent misuse and limit the number of requests from a client.

7. What is CORS and why is it important in web APIs?

Answer: CORS (Cross-Origin Resource Sharing) is a security feature implemented by web browsers to prevent web pages from making requests to a different domain than the one that served the web page. It is important in web APIs because it enables servers to specify who can access their resources, ensuring that only trusted domains can make cross-origin requests.

8. What are some common status codes returned by APIs, and what do they mean?

Answer:

- **200 OK**: The request was successful.
- **201 Created**: A new resource was successfully created.
- **204 No Content**: The request was successful, but there is no content to return.
- **400 Bad Request**: The request was invalid or cannot be processed.
- **401 Unauthorized**: Authentication is required and has failed or not been provided.
- **403 Forbidden**: The server understands the request but refuses to authorize it.
- **404 Not Found**: The requested resource could not be found.
- **500 Internal Server Error**: An error occurred on the server side.

9. What is the purpose of API versioning and how can it be implemented?

Answer: API versioning ensures that changes in the API do not break existing client applications. It allows developers to introduce new features and improvements without disrupting the current functionality for existing users. Versioning can be implemented in several ways:

- URL Path: Including the version number in the URL (e.g., /v1/users).
- Query Parameters: Adding a version parameter in the query string (e.g., ?version=1).
- Headers: Using a custom header to specify the version (e.g., Accept: application/vnd.myapi.v1+json).

10. What are the differences between synchronous and asynchronous API calls?

Answer:

- **Synchronous API Calls**:

    - The client sends a request and waits for the server to respond.
    - The client is blocked until the response is received.
    - Simple and straightforward but can lead to inefficiencies if the server takes a long time to respond.
- **Asynchronous API Calls**:
    - The client sends a request and continues processing other tasks.
    - The client is notified (usually via a callback or promise) when the response is ready.
    - More efficient for long-running operations, as it doesn't block the client.

11. What is rate limiting in APIs and why is it important?

Answer: Rate limiting is a mechanism to control the number of requests a client can make to an API within a certain time period. It is important for:

- **Preventing Abuse**: Protects the API from being overwhelmed by too many requests from a single client.
- **Ensuring Fair Usage**: Ensures that all clients have fair access to the API.
- **Maintaining Performance**: Helps maintain the API's performance and availability by avoiding excessive load.

12. Explain the concept of API throttling.

Answer: API throttling is a technique used to control the usage of an API by limiting the number of requests that can be made in a specific time frame. It helps in managing the load on the server, ensuring service availability, and preventing abuse. Throttling can be implemented by setting a limit on the number of requests per minute, hour, or day.

Chapter 2

Setting Up Your API Testing Environment

Choose Your API Testing Tool

Manual API testing tools are designed to allow testers to interact with APIs in an intuitive, often graphical interface, providing the capability to send requests, inspect responses, and validate functionality without writing code. Here are some popular manual API testing tools along with their key features:

1. Postman

Postman is one of the most widely used tools for API testing.

Features:

- **User-Friendly Interface**: Intuitive interface for creating, saving, and organizing API requests.
- **Support for Various HTTP Methods**: Supports GET, POST, PUT, DELETE, PATCH, and more.
- **Environment Variables**: Allows setting up different environments (e.g., development, testing, production) and switching between them easily.
- **Collections**: Organize requests into collections and folders.
- **Automated Testing**: Basic automated testing capabilities using JavaScript.
- **Mock Servers**: Create mock servers to simulate endpoints and responses.
- **Collaboration**: Team collaboration features to share collections and environments.

Website: Postman

2. Insomnia

Insomnia is another popular API client that emphasizes simplicity and ease of use.

Features:

- **Simple Interface**: Clean and straightforward interface for creating and managing requests.
- **Environment Variables**: Support for environment variables to manage different configurations.
- **GraphQL Support**: Native support for GraphQL queries and mutations.
- **Plugins**: Extend functionality with community plugins.
- **Scripting**: Pre-request scripts and response templating with Nunjucks.

Website: Insomnia

3. Swagger UI

Swagger UI allows you to visualize and interact with API documentation.

Features:

- **Interactive API Documentation**: Automatically generate interactive API documentation from OpenAPI specifications.
- **Try It Out**: Provides an interface to send requests directly from the documentation.
- **Customization**: Customizable look and feel.

Website: Swagger UI

4. SoapUI

SoapUI is a comprehensive tool for testing SOAP and RESTful APIs.

Features:

- **Extensive Protocol Support**: Supports REST, SOAP, GraphQL, and other protocols.
- **Data-Driven Testing**: Supports data-driven testing using various data sources.
- **Assertions**: Rich set of assertions for validating responses.
- **Automation**: Scripting support with Groovy.
- **Load Testing**: Built-in capabilities for performance and load testing.

Website: SoapUI

5. HTTPie

HTTPie is a command-line HTTP client designed to make CLI interaction with web services as human-friendly as possible.

Features:

- **User-Friendly CLI**: Simple syntax for making requests and viewing responses.
- **JSON Support**: Automatically formats JSON responses.
- **Plugins**: Extensible with plugins.
- **Scripting**: Suitable for inclusion in shell scripts for automated testing.

Website: HTTPie

6. Paw

Paw is a full-featured API client for Mac.

Features:

- **Mac Integration**: Explicitly designed for macOS, with native look and feel.
- **Environment Variables**: Support for environments and variables.
- **Extensions**: Create custom Paw extensions using JavaScript.
- **Response Rendering**: View responses in various formats (e.g., JSON, XML, HTML).

Website: Paw

7. JMeter

JMeter, an open-source project, is renowned for its performance testing capabilities, though it also effortlessly accommodates API testing. It features a user-friendly interface that simplifies the creation of complex tests.

Features of JMeter:

- **Easy to set up:** Setting up JMeter is straightforward. It just requires downloading the JMeter jar to start testing.
- **Open-source**: Being open source, JMeter offers all its features for free, making it an accessible tool for everyone.
- **Powerful CLI tool:** Beyond its UI, JMeter can execute tests via its command-line interface, providing users extensive options, including reporting functionalities.
- **Extensible:** The tool's capabilities can be expanded through numerous plugins, which users can easily download and integrate.
- **Multiple operating systems support:** As a Java (jar) application, JMeter is compatible with any operating system that supports Java, broadening its usability.

Pricing:

JMeter is an open-source tool; hence, it is available for free. Users can leverage all of its features without cost, making it an economical choice for individuals and small organizations.

Who's It For?

JMeter is suitable for developers and testers who require a robust performance and API testing tool. Its extensibility and support for multiple operating systems make it a versatile choice for many users. However, due to its complexity and high memory consumption, it might be more suited for those with more experience in testing or those willing to invest time in learning the tool.

Install and Set Up the Tool

We'll use Postman for this guide due to its widespread use and comprehensive features.

1. Installing Postman

1. **Download Postman** from the official website: Postman Download.
2. **Install** the application by following the installation instructions for your operating system (Windows, macOS, Linux).
3. **Sign up** for a Postman account (optional but recommended for saving and syncing your work).

Install the Postman CLI[4]

You can download and install the Postman CLI manually or programmatically (with a script or the command line).

System requirements

The Postman CLI supports the exact operating system requirements as the Postman desktop app. For a complete list of system requirements, see Installing and updating Postman.

Windows installation

Run the following commands to install the Postman CLI for Windows. This will download an install script and run it. The install script creates a %USERPROFILE%\AppData\Local\Microsoft\WindowsApps directory if it doesn't exist yet, then installs a postman binary there.

```
powershell.exe -NoProfile -InputFormat None -ExecutionPolicy AllSigned -Command "[System.Net.ServicePointManager]::SecurityProtocol = 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://dl-cli.pstmn.io/install/win64.ps1'))"
```

Mac (Apple silicon) installation

Run the following command to install the Postman CLI for Macs with an Apple silicon processor. This will download an install script and run it. The install script creates a **/usr/local/bin** directory if it doesn't exist yet, then installs a postman binary there.

```
Copycurl -o- "https://dl-cli.pstmn.io/install/osx_arm64.sh" | sh
```

Mac (Intel) installation

Run the following command to install the Postman CLI for Macs with Intel chips. This will download an install script and run it. The install script creates a **/usr/local/bin** directory if it doesn't exist yet, then installs a postman binary there.

[4] https://learning.postman.com/docs/postman-cli/postman-cli-installation/

Copycurl -o- "https://dl-cli.pstmn.io/install/osx_64.sh" | sh

Linux installation

Run the following command to install the Postman CLI for the latest Linux version. This will download an install script and run it. The install script creates a /usr/local/bin directory if it doesn't exist yet, then installs a postman binary there.

Copycurl -o- "https://dl-cli.pstmn.io/install/linux64.sh" | sh

Update your CLI installation

To update your Postman CLI installation to the latest version, run the same command you used to install. The new version will overwrite the earlier version.

2. Configuring Postman

1. **Open Postman**.
2. **Create a New Workspace**:
    - Click on your workspace name in the top left.
    - Select "Create Workspace".
    - Name your workspace and provide a description (optional).
    - Choose the visibility (Personal, Team, Public).
3. **Set Up Environments**:
    - Click on the gear icon in the top right corner and select "Manage Environments".
    - Click "Add" to create a new environment.
    - Add variables such as `baseUrl`, `apiKey`, etc.
    - Save the environment.

3. Organize Your Requests

Using Collections

1. **Create a New Collection**:
    - Click on the Collections tab on the left sidebar.
    - Click "New Collection".
    - Name your collection and add a description (optional).
2. **Add Requests to the Collection**:
    - Click on your collection.
    - Click "Add Request".
    - Name your request and specify the HTTP method (GET, POST, etc.).

    - Enter the request URL using environment variables if needed (e.g., `{{baseUrl}}/endpoint`).
3. **Organize Requests**:
    - Create folders within collections to group related requests.
    - Drag and drop requests into folders for better organization.

4. Configure and Send Requests

1. **Set Up Request Headers**:
    - Click on the "Headers" tab in the request editor.
    - Add necessary headers, such as Content-Type, Authorization, etc.
    - Use environment variables for dynamic values (e.g., {{apiKey}}).
2. **Set Up Request Body** (for POST, PUT, PATCH requests):
    - Click on the "Body" tab.
    - Choose the appropriate format (raw, form-data, x-www-form-urlencoded).
    - Enter the request payload. Select "raw" and "JSON" for JSON payloads and input the JSON data.
3. **Send the Request**:
    - Click the "Send" button.
    - Review the response in the "Response" section, which includes the status code, headers, and body.

5. Validate Responses

1. **Check Status Codes**: Ensure the API returns the expected HTTP status code (e.g., 200 for success, 404 for not found).
2. **Validate Response Body**:
    - Check that the response body contains the expected data.
    - Use Postman's built-in JSON viewer for easy navigation.
3. **Use Tests for Validation**:
    - Click on the "Tests" tab in the request editor.
    - Write JavaScript code to perform assertions on the response.

Example:

```
pm.test("Status code is 200", function () {
   pm.response.to.have.status(200);
});
pm.test("Response body has userId", function () {
   var jsonData = pm.response.json();
   pm.expect(jsonData).to.have.property("userId");
});
```

6. Documentation and Collaboration

1. **Document Your API**:
    - Add detailed descriptions to your requests and collections.
    - Use Markdown in descriptions for better formatting.
2. **Share Collections**:
    - Click on the ellipsis (three dots) next to your collection.
    - Select "Share Collection".
    - Share via a link, export to a file, or use Postman's built-in collaboration features if you're in a team workspace.

7. Advanced Features

1. **Environment and Global Variables**: Manage different environments (e.g., development, staging, production) using environment variables. Define global variables for values that are used across multiple environments.
2. **Pre-Request Scripts**: Write JavaScript code to execute before a request is sent. Useful for setting dynamic variables.
3. **Mock Servers**: Create mock servers to simulate API endpoints and responses.

Common Interview Questions & Answers Related to API Test Tools

Here are common interview questions related to manual API testing tools, along with brief answers to help you prepare for interviews:

Basic Questions

1. **What is Postman?**

   Postman is a popular API development and testing tool that allows users to create, test, document, and share APIs.

2. **How do you install Postman?**

   You can install Postman by downloading it from the Postman website and following the installation instructions for your operating system.

3. **What is a Postman Collection?**

   A collection in Postman is a group of related API requests that can be organized into folders and subfolders. It helps in organizing and managing your API tests efficiently.

4. **How do you create a new request in Postman?**

   Click on the "New" button, select "Request," enter a request name, and specify the HTTP method and URL.

5. **What HTTP methods can you use in Postman?**

   Postman supports all standard HTTP methods, including GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS.

Intermediate Questions

6. **How do you manage environments in Postman?**

   Click the gear icon, select "Manage Environments," and create new environments by defining variables like `baseUrl`, `apiKey`, etc.

7. **What are environment variables in Postman?**

   Environment variables are placeholders for values that can change depending on the environment (e.g., development, staging, production). They make requests reusable across different environments.

8. **How do you add headers to a request in Postman?**

In the request editor, go to the "Headers" tab and add key-value pairs for headers such as `Content-Type`, `Authorization`, etc.

9. **How do you handle authentication in Postman?**

   Use the "Authorization" tab in the request editor to set up authentication mechanisms like Basic Auth, Bearer Token, OAuth, API Key, etc.

10. **What is the use of the "Tests" tab in Postman?**

    The "Tests" tab allows you to write JavaScript code to perform assertions on the response, validate data, and automate test scripts.

Advanced Questions

11. **How do you use Postman for automated testing?**

    Write test scripts in the "Tests" tab and use the Postman CLI tool, Newman, to run collections from the command line for automated testing.

12. **What is Newman?**

    Newman is a command-line collection runner for Postman, allowing you to run and test Postman collections directly from the CLI.

13. **How do you run a collection in Newman?**

    Install Newman using

    ```
    npm install -g newman
    ```

    then run a collection using

    ```
    newman run <collection-file>.
    ```

14. **How do you use pre-request scripts in Postman?**

    Use the "Pre-request Script" tab to write JavaScript code that executes before the request is sent, helpful in setting dynamic variables and pre-request logic.

15. **What are Postman variables?**

    Variables in Postman can be defined at different scopes: global, environment, collection, and local. They store values that can be reused in requests and scripts.

Practical Questions

16. **How do you export a Postman collection?**

    Click on the collection, select "Export," and choose the format (v2.1 or v2.0) to save it as a JSON file.

17. **How do you import a collection into Postman?**

    Click on the "Import" button, select the file or paste the raw text of the collection JSON, and import it.

18. **What is a Postman monitor?**

    Postman monitors are automated runs of your Postman collections at scheduled intervals to check the performance and health of your APIs.

19. **How do you create a mock server in Postman?**

    Navigate to the "Mock Server" section, create a new mock server, and associate it with an existing collection to simulate endpoints and responses.

20. **How can you document APIs using Postman?**

    Add detailed descriptions to requests and collections, use Markdown for formatting, and generate documentation using Postman's built-in documentation feature.

Troubleshooting Questions

21. **How do you debug a failing request in Postman?**

    Check the response status code, headers, and body. Use console logs in test scripts and the Postman console to debug issues.

22. **What are some common HTTP status codes you might encounter?**

    Common status codes include 200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, and 500 Internal Server Error.

23. **How do you handle dynamic data in Postman?**

    Use pre-request scripts and environment variables to generate and store dynamic data, such as timestamps or random values.

24. **How do you validate the structure of a JSON response?**

    Write JavaScript test scripts to check if the response body contains the expected JSON structure and values.

25. **What is the difference between global and environment variables in Postman?**

Global variables are available across all collections and environments, while environment variables are specific to a particular environment.

Integration Questions

26. **How do you integrate Postman with CI/CD pipelines?**

    Use Newman to run Postman collections in CI/CD pipelines by integrating it with tools like Jenkins, GitLab CI, Travis CI, etc.

27. **What is the purpose of the Postman API?**

    The Postman API allows you to programmatically access and manage Postman resources, such as collections, environments, and monitors.

28. **How do you handle rate limiting in Postman?**

    Implement delay logic in test scripts or pre-request scripts to wait between requests, or configure Postman monitors to run at intervals within the rate limits.

29. **How can you share Postman collections with your team?**

    Share collections by generating a shareable link, exporting the collection file, or using Postman's built-in team workspaces for collaboration.

30. **What are Postman workspaces, and how do they help in collaboration?**

    Workspaces in Postman are collaborative spaces where team members can share collections, environments, and other resources, making it easier to work together on API development and testing.

Chapter 3

Understanding API Requests and Responses

HTTP Methods

REST stands for Representational State Transfer and is an architectural pattern for creating web services. Web services that adhere to REST principles are known as RESTful web services. These services are widely used by application developers due to their simplicity in enabling communication between servers on different machines.

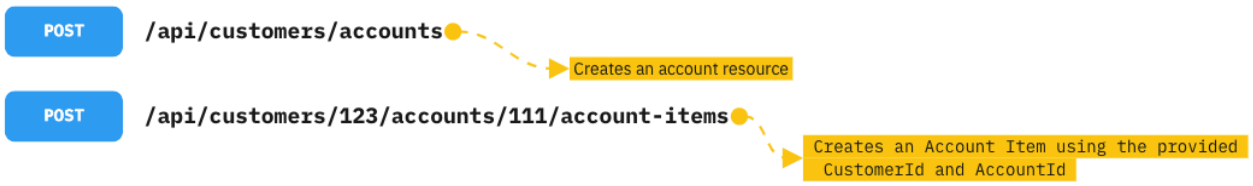
REST facilitates easy data sharing between clients and servers. RESTful applications utilize HTTP methods such as GET, POST, DELETE, and PUT to perform CRUD (Create, Read, Update, Delete) operations.

By now, you should understand how the HTTP protocol operates, including the various HTTP methods and their appropriate uses. This section will delve deeper into each process, starting with GET, the most commonly used HTTP method. So, let's dive in!

1. GET

- **Purpose:** Retrieve data from a server.
- **Usage:** Requests data from a specified resource.
- **Idempotent:** Yes, calling GET multiple times does not change the resource state.

2. POST



- **Purpose:** Send data to the server to create a new resource.
- **Usage:** Submits data to be processed to a specified resource, often causing a change in state or side effects on the server.
- **Idempotent:** No, multiple POST requests may result in multiple resource creations.

3. PUT

- **Purpose:** Update or replace an existing resource or create a new one if it does not exist.
- **Usage:** Sends data to a specified resource, often replacing the resource entirely.
- **Idempotent:** Yes, multiple PUT requests with the same data result in the same resource state.

4. DELETE

- **Purpose:** Remove a specified resource from the server.
- **Usage:** Deletes the specified resource.
- **Idempotent:** Yes, deleting a resource multiple times has the same effect as deleting it once.

5. PATCH

- **Purpose:** Apply partial modifications to a resource.
- **Usage:** Sends partial data to update an existing resource.
- **Idempotent:** Yes, applying the same PATCH multiple times will result in the same state.

6. HEAD

- **Purpose:** Retrieve headers for a resource, similar to GET but without the response body.
- **Usage:** Used to obtain metadata about a resource, such as headers.
- **Idempotent:** Yes, calling HEAD multiple times does not change the resource state.

7. OPTIONS

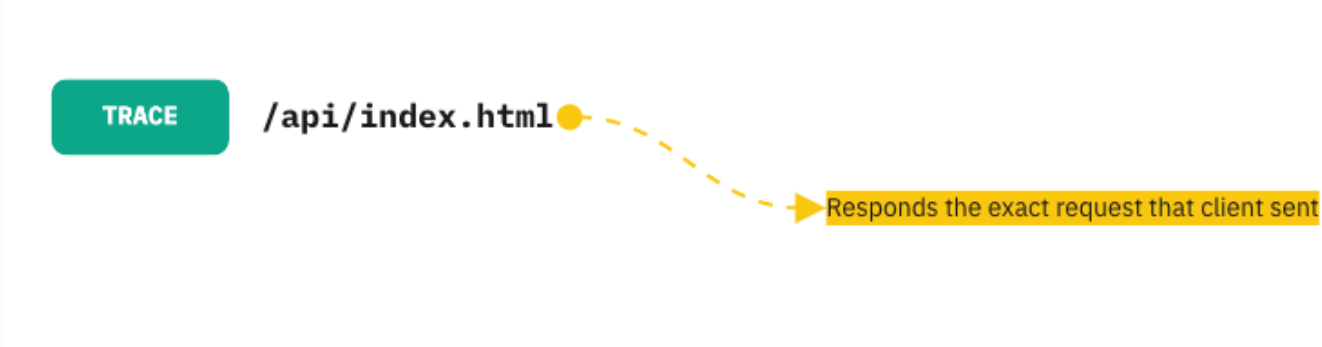
- **Purpose:** Describe the communication options for the target resource.
- **Usage:** Used to determine the HTTP methods and other options supported by a web server.
- **Idempotent:** Yes, it does not change the resource state.

8. CONNECT

- **Purpose:** Establish a tunnel to the server identified by the target resource.

- **Usage:** Mainly used for tunneling, such as establishing a VPN or SSL connection through an HTTP proxy.
- **Idempotent:** Yes, it establishes a connection but does not change the resource state.

9. TRACE



- **Purpose:** Perform a message loop-back test along the path to the target resource.
- **Usage:** Used for diagnostic purposes to trace the path of a request.
- **Idempotent:** Yes, it is a diagnostic method and does not alter the resource state.

Comparison of HTTP methods[5]

Request method	RFC	Request has payload body	Response has payload body	Safe	Idempotent	Cacheable
GET	RFC 9110	Optional	Yes	Yes	Yes	Yes
HEAD	RFC 9110	Optional	No	Yes	Yes	Yes
POST	RFC 9110	Yes	Yes	No	No	Yes
PUT	RFC 9110	Yes	Yes	No	Yes	No
DELETE	RFC 9110	Optional	Yes	No	Yes	No
CONNECT	RFC 9110	Optional	Yes	No	No	No
OPTIONS	RFC 9110	Optional	Yes	Yes	Yes	No
TRACE	RFC 9110	No	Yes	Yes	Yes	No
PATCH	RFC 5789	Yes	Yes	No	No	

[5] https://en.wikipedia.org/wiki/HTTP

Request Headers and Parameters

Request headers and parameters are essential components of an HTTP request. They convey additional information about the request and the client making it. Here's a breakdown of each:

Request Headers

HTTP headers are key-value pairs sent between the client and the server along with the request or response. They provide metadata about the request or the data being sent. Some common request headers include:

- **User-Agent**: Identifies the client making the request, often containing information about the user agent's software and operating system.
- **Host**: Specifies the domain name of the server being requested.
- **Accept**: Informs the server about the types of media that the client can process.
- **Content-Type**: Specifies the media type of the request body, typically used in POST and PUT requests.
- **Authorization**: Contains credentials for authenticating the client with the server, often used with Basic or Bearer authentication.
- **Cookie**: Contains stored HTTP cookies previously sent by the server with the Set-Cookie header.

Request Parameters

Request parameters are key-value pairs sent in the URL query string or in the request body. They provide additional data to the server, often used for filtering, sorting, or providing context to the request. Some common types of request parameters include:

- **Query Parameters**: Appended to the end of the URL after a question mark (?), query parameters are key-value pairs separated by ampersands (&). For example, `https://example.com/search?q=term&page=1`.
- **Path Parameters**: Used to parameterize parts of the URL path, typically denoted by placeholders in the URL. For example, `https://example.com/users/{id}`.
- **Form Data**: Sent in the body of POST requests, form data consists of key-value pairs encoded in either `application/x-www-form-urlencoded` or `multipart/form-data`.
- **JSON/XML Data**: In RESTful APIs, data can be sent in JSON or XML format in the request body.

Example Request:

Here's an example of an HTTP request with headers and parameters:

```
GET /search?q=example HTTP/1.1

Host: example.com

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8

Accept-Language: en-US,en;q=0.9
```

Query Parameter

Query parameters are key-value pairs appended to the end of a URL, typically after a question mark (?). They provide additional information to a web server about the request being made. Here's an overview:

Structure:

- Query parameters are separated by an ampersand (&) if there are multiple parameters.
- Each parameter consists of a key and a value, separated by an equals sign (=).

Example:

Consider the following URL:

```
https://api.example.com/search?q=example&limit=10&page=2
```

In this URL:

- `https://api.example.com/search` is the base URL.
- `q=example`, `limit=10`, and `page=2` are query parameters.
- The key-value pairs are `q=example`, `limit=10`, and `page=2`.
- The keys are `q`, `limit`, and `page`, and their respective values are `example`, `10`, and `2`.

Usage:

- **Filtering**: Query parameters are commonly used for filtering data. For example, `?category=books` to filter products by category.
- **Pagination**: Parameters like `page` and `limit` are frequently used for pagination to retrieve a subset of results.

- **Search**: Parameters like `q` are used to specify search queries.

Fragment Parameters

Fragment parameters, also known as fragment identifiers or hash fragments, are components of a URL that appear after a hash symbol (#). They are commonly used to specify a specific section or resource within a web page, especially in client-side scripting and navigation.

Structure:

- Fragment parameters are appended to the end of a URL after a hash symbol (#).
- They typically consist of key-value pairs or identifiers that denote a specific section or resource within a document.
- Fragment parameters do not get transmitted to the server; they are processed exclusively by the client-side browser.

Example:

Consider the following URL:

`https://example.com/page#section1`

In this URL:

- `https://example.com/page` is the base URL.
- `#section1` is the fragment parameter.
- It denotes that the browser should navigate to or highlight the section with the identifier `section1` within the document.

Character Encoding

Special characters are encoded in the URL, by a mechanism called “percent encoding”. In this mechanism any character can be replaced by the percent symbol, followed by a two-digit hexadecimal value of the encoded character. If special characters (such as the hash character) need to be sent as actual data, they must be encoded. All other characters can optionally be encoded.

Character encoding is a process used to represent characters in a computer's memory or storage by assigning each character a unique numerical value. This allows computers to store, transmit, and display textual data in various human-readable languages and scripts. Here's an overview:

Unicode:

Unicode is the most commonly used character encoding standard today. It assigns unique numerical values (code points) to characters from virtually all writing systems and symbols used worldwide. Unicode supports over 143,000 characters, covering various scripts, symbols, emoji, and special characters.

UTF-8:

UTF-8 (Unicode Transformation Format-8) is a variable-width character encoding that represents Unicode characters using 8-bit code units. It's the most widely used encoding on the internet due to its compatibility with ASCII (American Standard Code for Information Interchange) and its ability to represent all Unicode characters efficiently.

Other Encodings:

- **UTF-16**: Represents Unicode characters using 16-bit code units. It's commonly used in environments that require fixed-width characters, such as Windows operating systems.
- **ISO 8859-1 (Latin-1)**: A widely used single-byte encoding that supports Western European languages. It's not as versatile as Unicode and may not support characters from other writing systems.
- **ASCII**: A 7-bit character encoding standard that represents basic English characters, symbols, and control characters. It's compatible with UTF-8, making it a subset of Unicode.

Size Limits

While the URI standard remains silent on prescribing a maximum URL size, many clients impose an arbitrary limit of 2000 characters. When handling data that's intricate or exceeds this threshold, it's advisable to transmit it within the body of the request. This is particularly relevant when dealing with data that's challenging to represent hierarchically.

As we explored in the SOAP vs. REST comparison, the request's body offers a versatile canvas for structuring data, accommodating various machine-readable formats. While XML and JSON formats are prevalent, the flexibility also extends to other formats.

By leveraging the request body for transmitting sizable or complex data, APIs can sidestep the constraints posed by URL character limits. This approach not only fosters cleaner, more maintainable APIs but also enhances interoperability and scalability.

3.3. Response Codes and Their Meanings

HTTP status codes are standardized numerical codes that indicate the outcome of an HTTP request. They provide information about the result of the request and whether it was successful or encountered an error. Here are some common HTTP status codes and their meanings:

1xx Informational Responses

- **100 Continue**: The server has received the initial part of the request and is willing to accept further processing.
- **101 Switching Protocols**: The server is indicating a change in protocol requested by the client (e.g., switching from HTTP to WebSocket).
- **102 Processing** (WebDAV; RFC 2518): A WebDAV request may contain many sub-requests involving file operations, requiring a long time to complete the request.
- **103 Early Hints** (RFC 8297): Used to return some response headers before final HTTP message.

2xx Success

- **200 OK**: The request was successful.
- **201 Created**: The request has been fulfilled, and a new resource has been created.
- **202 Accepted:** The request has been accepted for processing, but the processing has not been completed.
- **204 No Content:** The server successfully processed the request but is not returning any content.
- **205 Reset Content**: The server successfully processed the request, asks that the requester reset its document view, and is not returning any content. This status code is typically used in conjunction with form submissions.
- **206 Partial Content**: The server is delivering only part of the resource (byte serving) due to a range header sent by the client. This status code is commonly used for resuming interrupted downloads or splitting a download into multiple simultaneous streams.
- **207 Multi-Status**: This status code, defined in WebDAV (RFC 4918), indicates that the message body contains an XML message by default and can contain several separate response codes, depending on how many sub-requests were made.
- **208 Already Reported**: Defined in WebDAV (RFC 5842), this status code indicates that the members of a DAV binding have already been enumerated in a preceding part of the response and are not being included again.
- **226 IM Used**: Defined in RFC 3229, this status code indicates that the server has fulfilled a request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance. It's used in conjunction with instance-manipulation applied to the resource.

3xx Redirection

- **300 Multiple Choices**: Indicates multiple options for the resource that the client may follow.
- **301 Moved Permanently**: The requested resource has been permanently moved to a new URL.
- **302 Found**: The requested resource temporarily resides under a different URL.
- **304 Not Modified**: The client's cached copy of the requested resource is still valid. This status code indicates that the resource has not been modified since the last request, and the client can use its cached copy.

- **305 Use Proxy**: This status code is provided in the response to indicate that the requested resource is only available through a proxy server. However, many HTTP clients do not obey this status code for security reasons.

- **306 Switch Proxy**: This status code is no longer used. It was originally intended to instruct subsequent requests to use the specified proxy.

- **307 Temporary Redirect**: In this case, the request should be repeated with another URI, but future requests should still use the original URI. Unlike the 302 status code, the request method is not allowed to change when reissuing the original request. For example, if the original request was a POST request, it should be repeated using another POST request.

- **308 Permanent Redirect**: This status code, like 307, indicates that future requests should be directed to the given URI. However, unlike 307, 308 does not allow the HTTP method to change. So, for example, submitting a form to a permanently redirected resource may continue smoothly without changing the method.

4xx Client Errors

- **400 Bad Request**: The server could not understand the request due to invalid syntax.
- **401 Unauthorized**: The request requires user authentication.
- **403 Forbidden**: The server understood the request but refuses to authorize it.
- **404 Not Found**: The requested resource could not be found on the server.
- **405 Method Not Allowed**: The request method is not supported for the specified resource.
- **405 Method Not Allowed**: The request method is not supported for the requested resource. For example, trying to perform a GET request on a resource that only allows POST requests.
- **406 Not Acceptable**: The requested resource is capable of generating only content that is not acceptable according to the Accept headers sent in the request.
- **407 Proxy Authentication Required**: The client must first authenticate itself with the proxy.
- **408 Request Timeout**: The server timed out waiting for the request. The client may repeat the request without modifications at any later time.
- **409 Conflict**: Indicates that the request could not be processed because of a conflict in the current state of the resource.
- **410 Gone**: Indicates that the resource requested was previously in use but is no longer available and will not be available again.

- **411 Length Required**: The request did not specify the length of its content, which is required by the requested resource.
- **412 Precondition Failed**: The server does not meet one of the preconditions that the requester put on the request header fields.
- **413 Payload Too Large**: The request is larger than the server is willing or able to process.
- **414 URI Too Long**: The URI provided was too long for the server to process.
- **415 Unsupported Media Type**: The request entity has a media type which the server or resource does not support.
- **416 Range Not Satisfiable**: The client has asked for a portion of the file, but the server cannot supply that portion.
- **417 Expectation Failed**: The server cannot meet the requirements of the Expect request-header field.
- **418 I'm a teapot**: A humorous status code, not intended to be implemented by actual HTTP servers.
- **421 Misdirected Request**: The request was directed at a server that is not able to produce a response.
- **422 Unprocessable Content**: The request was well-formed but could not be processed.
- **423 Locked**: The resource that is being accessed is locked.
- **424 Failed Dependency**: The request failed because it depended on another request and that request failed.
- **425 Too Early**: Indicates that the server is unwilling to risk processing a request that might be replayed.
- **426 Upgrade Required**: The client should switch to a different protocol given in the Upgrade header field.
- **428 Precondition Required**: The origin server requires the request to be conditional.
- **429 Too Many Requests**: The user has sent too many requests in a given amount of time.
- **431 Request Header Fields Too Large**: The server is unwilling to process the request because an individual or all header fields are too large.
- **451 Unavailable For Legal Reasons**: A server operator has received a legal demand to deny access to the resource.

5xx Server Errors

- **500 Internal Server Error**: A generic error message indicating that the server encountered an unexpected condition.
- **501 Not Implemented**: The server does not support the functionality required to fulfill the request.
- **502 Bad Gateway**: The server received an invalid response from an upstream server while acting as a gateway or proxy.
- **503 Service Unavailable**: The server is currently unable to handle the request due to temporary overloading or maintenance of the server.
- **504 Gateway Timeout**: The server, acting as a gateway or proxy, did not receive a timely response from the upstream server.
- **505 HTTP Version Not Supported**: The server does not support the HTTP version used in the request.

- **506 Variant Also Negotiates**: Transparent content negotiation for the request results in a circular reference.
- **507 Insufficient Storage**: The server is unable to store the representation needed to complete the request.
- **508 Loop Detected**: The server detected an infinite loop while processing the request.
- **510 Not Extended**: Further extensions to the request are required for the server to fulfill it.
- **511 Network Authentication Required**: The client needs to authenticate to gain network access. This is often used by intercepting proxies to control access to the network, such as "captive portals" requiring agreement to Terms of Service before granting full Internet access via a Wi-Fi hotspot.

How to Test HTTP Method

How to test an API with a GET method?

When we want to test an API, the most popular method that we would use is the GET method. Therefore, We expect the following to happen.

- If the resource is accessible, the API returns the 200 Status Code, which means OK.
- Along with the 200 Status Code, the server usually returns a response body in XML or JSON format. So, for example, we expect the [GET] /members endpoint to return a list of members in XML or JSON.
- If the server does not support the endpoint, the server returns the 404 Status Code, which means Not Found.
- If we send the request in the wrong syntax, the server returns the 400 Status Code, which means Bad Request.
- curl -v -X GET https://api.example.com/resource

How to test a POST endpoint?

Since the POST method creates data, we must be cautious about changing data; testing all the POST methods in APIs is highly recommended. Moreover, make sure to delete the created resource once your testing is finished.

Here are some suggestions that we can do for testing APIs with POST methods:

- Create a resource with the POST method, and it should return the 201 Status Code.
- Perform the GET method to check if it created the resource was successfully created. You should get the 200 status code, and the response should contain the created resource.
- Perform the POST method with incorrect or wrong formatted data to check if the operation fails.
- curl -v -X POST https://api.example.com/resource -H "Content-Type: application/json" -d '{"key":"value"}'

How to test an API with a PUT method?

The PUT method is idempotent, and it modifies the entire resources, so to test that behavior, we make sure to do the following operations:

- Send a PUT request to the server many times, and it should always return the same result.
- When the server completes the PUT request and updates the resource, the response should come with 200 or 204 status codes.

- After the server completes the PUT request, make a GET request to check if the data is updated correctly on the resource.
- If the input is invalid or has the wrong format, the resource must not be updated.
- curl -v -X PUT https://api.example.com/resource/123 -H "Content-Type: application/json" -d '{"key":"value"}'

How to test an API with a PATCH method?

To test an API with the PATCH method, follow the steps discussed in this article for the testing API with the PUT and the POST methods. Consider the following results:

- Send a PATCH request to the server; the server will return the 2xx HTTP status code, which means: the request is successfully received, understood, and accepted.
- Perform the GET request and verify that the content is updated correctly.
- If the request payload is incorrect or ill-formatted, the operation must fail.
- curl -v -X PATCH https://api.example.com/resource/123 -H "Content-Type: application/json" -d '{"key":"new value"}'

How to test a DELETE endpoint?

When it comes to deleting something on the server, we should be cautious. We are deleting data, and it is critical. First, make sure that deleting data is acceptable, then perform the following actions.

- Call the POST method to create a new resource. Never test DELETE with actual Data. For example, first, create a new customer and then try to delete the customer you just created.
- Make the DELETE request for a specific resource. For example, the request [DELETE] /customers/{customer-id} deletes a customer with thee specified customer Id.
- Call the GET method for the deleted customer, which should return 404, as the resource no longer exists.
- curl -v -X DELETE https://api.example.com/resource/123
- Testfully's Multi-step tests allow you to create resources on the fly and use them for testing DELETE endpoints.

How to test a HEAD endpoint?

One of the advantages of the HEAD method is that we can test the server if it is available and accessible as long as the API supports it, and it is much faster than the GET method because it has no response body. The status code we expect to get from the API is 200. Before every other HTTP method, we can first test API with the HEAD method.

- curl -I https://api.example.com/resource

How to test an OPTIONS endpoint?

Depending on whether the server supports the OPTIONS method, we can test the server for the times of FATAL failure with the OPTIONS method. To try it, consider the following.

- Make an OPTIONS request and check the header and the status code that returns.
- Test the case of failure with a resource that doesn't support the OPTIONS method.
- curl -v -X OPTIONS https://api.example.com/resource

How to test an API with a TRACE method?

- Make a standard HTTP request like a GET request to /api/status
- Replace GET with the TRACE and send it again.
- Check what the server returns. If the response has the same information as the original request, the TRACE ability is enabled in the server and works correctly.
- curl -v -X TRACE https://api.example.com/resource

How to test an API with a CONNECT method?

Testing the HTTP CONNECT method can be done using tools that support raw HTTP requests, such as cURL, Telnet, or specific network diagnostic tools. The CONNECT method is primarily used to establish a network connection to a resource, often through a proxy server. Here's how you can test it using different tools:

- curl -v -X CONNECT https://example.com:443

Analyzing Response Body

Analyzing response bodies during testing ensures that APIs function correctly and return the expected data. Here's how to approach this aspect of API testing:

1. **Validate Structure**: Check that the response body follows the expected structure defined in the API documentation or specifications. This includes verifying the presence of required fields, nested objects, arrays, and data types.

2. **Parse JSON/XML**: If the response body is in JSON or XML format, parse it programmatically to extract relevant data for validation. Use libraries or tools appropriate for your programming language to parse and manipulate the response.

3. **Verify Data Integrity:** Ensure that the data returned in the response body is accurate and consistent with the input provided in the request. Compare the response data against the expected values or data retrieved from the database.

4. **Handle Pagination:** If the API supports pagination, analyze the response body to ensure that pagination parameters such as page numbers, page size, and total count are correctly returned. Test scenarios involving navigating through multiple pages of results.

5. **Test Edge Cases:** Evaluate how the API handles edge cases and boundary conditions in the response body. This includes scenarios such as empty responses, null values, maximum and minimum values, and unexpected input data.

6. **Check Error Responses:** Verify that error responses include meaningful error messages and appropriate HTTP status codes. Analyze the response body for error details such as error codes, descriptions, and troubleshooting instructions.

7. **Evaluate Performance:** Analyze the size and complexity of the response body to assess performance implications, especially for large datasets or complex object structures. Monitor response times and network traffic during performance testing.

8. **Handle Localization:** If the API supports localization or internationalization, analyze the response body to ensure that localized content is correctly returned based on the client's language preferences specified in the request.

9. **Test Content Negotiation:** If the API supports content negotiation, test scenarios where clients request specific media types (e.g., JSON, XML) in the Accept header. Analyze the response body to verify that the requested content type is returned.

10. **Validate Hypermedia Controls:** If the API follows the HATEOAS principle (Hypermedia as the Engine of Application State), analyze the response body to ensure that hypermedia links and controls are correctly included, allowing clients to navigate the API dynamically.

By thoroughly analyzing response bodies during testing, you can identify issues, validate data correctness, and ensure that the API behaves as expected in various scenarios. This helps maintain the reliability, usability, and performance of the API for end-users.

Common Interview Questions & Answers Related to API Request and Response

1. What is a Safe HTTP method?

A safe method is a method that doesn't change data on the server. For example, GET and HEAD are safe methods because the user or an application does not request side effects on the server when calling them.

2. What is an Idempotent method?

An Idempotent method means it returns the same response no matter how many times it runs. For example, The PUT and DELETE methods share this property. However, it is possible that a sequence of several requests is not idempotent, even if all of the methods executed in that sequence are idempotent. So, a series of requests is idempotent if a single execution of the entire series always returns the same result.

3. What are HTTP methods and their significance in API development?

Answer: HTTP methods (GET, POST, PUT, DELETE, PATCH, etc.) specify the action to be performed on a resource. They are essential for defining the behavior of APIs and determining how clients interact with resources.

4. What is the difference between GET and POST requests?

Answer: GET requests are used to retrieve data from the server, while POST requests are used to submit data to the server for processing or storage.

5. Explain the purpose of HTTP headers in an API request.

Answer: HTTP headers provide additional information about the request or the data being sent, such as content type, authentication credentials, and caching directives.

6. What is a query parameter in an API request?

Answer: A query parameter is a key-value pair appended to the end of a URL that provides additional information to the server about the request, such as filtering or pagination criteria.

7. What is the role of HTTP status codes in API responses?

Answer: HTTP status codes indicate the outcome of an API request, such as success, failure, or redirection. They help clients understand how to process the response.

8. Explain the purpose of HTTP headers in an API response.

Answer: HTTP headers provide metadata about the response, such as content type, caching directives, and server information.

9. What is content negotiation in API responses?

Answer: Content negotiation is the process of selecting the appropriate representation of a resource based on the client's preferences specified in the request headers.

10. What are common HTTP status codes for error responses?

Answer: Common HTTP status codes for error responses include 400 Bad Request, 404 Not Found, 500 Internal Server Error, etc.

11. How do you handle errors in API responses?

Answer: Errors in API responses should be handled gracefully by providing informative error messages, appropriate HTTP status codes, and troubleshooting guidance to the client.

Chapter 4

Security Testing of APIs

API Security Testing goes a long way in avoiding API breaches by preventing security vulnerabilities from ever reaching production environments. Noname Active Testing focuses on finding and remediating API security vulnerabilities during the development phase of the SDLC, before they can be exploited.

Security testing of APIs (Application Programming Interfaces) is crucial to ensure that the data and functionalities exposed by these interfaces are protected from various threats and vulnerabilities. Here is a comprehensive overview of security testing for APIs:

Types API security testing

Dynamic Application Security Testing (DAST)

DAST testing, or dynamic application security testing, is different than SAST in that API testing take place in production. Testers identify problems that occur during use and then trace them back to their origins in the software design, rather than detecting issues linked to a code module.

Software Composition Analysis (SCA)

SCA, or Software composition analysis, is a software engineering technique that helps to identify the software components and their relationships. It can be used for analyzing the design of an application, identifying code smells, or finding out how much code is needed for a given task.

Authentication

Authentication is the process of verifying the identity of a user or device, and it is used to access a system, service, or network. It's an important part of any application and can be done in many ways such as username and password authentication, two-factor authentication, and API authentication.

API authentication uses an API key to verify the identity of the user. This type of authentication can be used for both public and private APIs.

Objective: Ensure only authorized users can access the API.

Methods: Check the implementation of authentication mechanisms such as OAuth, JWT, API keys, and basic auth.

Tests: Validate token expiration, token revocation, replay attacks, and credential brute-forcing.

Authorization

The API authorization is a process of checking the identity of the user and authorizing them to access the application. It's a common practice in web applications and can be done by sending an HTTP request with the appropriate header and token in it. The API will then return a response with information about whether or not the request was successful or not.

Objective: Verify that users have permission to access resources and perform actions.

Methods: Role-based access control (RBAC), attribute-based access control (ABAC).

Tests: Check for privilege escalation, verify access control rules, and ensure that users cannot access data belonging to other users.

Common API Security Risks

Despite their benefits, APIs can be vulnerable to various security risks. Some common vulnerabilities include:

Injection Attacks:

APIs can be vulnerable to injection attacks, where malicious code or commands are injected into API requests. This can lead to unauthorized data exposure, compromised systems, or a takeover.

Broken Authentication and Session Management:

Weak authentication mechanisms, improper session handling, or inadequate access controls can expose APIs to authentication and session-related vulnerabilities. Attackers may exploit these weaknesses to impersonate legitimate users, hijack sessions, or gain unauthorized access to sensitive data.

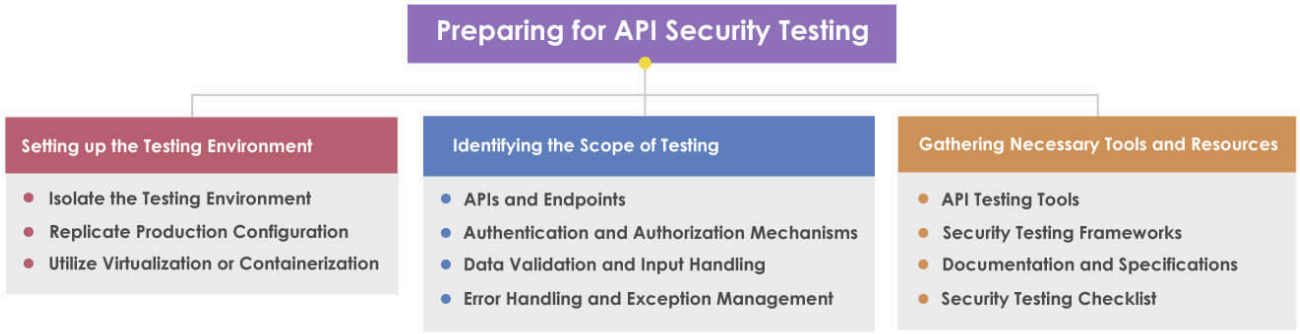
Insecure Direct Object References (IDOR):

APIs that expose internal references, such as database IDs or file paths, without proper authorization checks can be prone to IDOR vulnerabilities. Attackers can manipulate these references to access unauthorized resources or sensitive information.

Denial-of-Service (DoS) Attacks:

APIs can be targeted with DoS attacks, where attackers overwhelm the API infrastructure with a flood of requests, rendering the API unresponsive or unavailable. This disrupts services, impacts user experience, and potentially leads to financial losses.

4.3.Preparing for API Security Testing



Ensuring the security of APIs is crucial to protect sensitive data and prevent potential breaches. API security testing plays a vital role in identifying vulnerabilities and weaknesses in API implementations. Following are some pointers to consider when preparing for API security testing:

Setting up the Testing Environment

Establishing a robust testing environment that closely resembles the production environment is essential. Here are key considerations:

Isolate the Testing Environment:

Create a separate, isolated environment dedicated explicitly to API security testing. This prevents any accidental impact on the production systems and ensures a controlled testing environment.

Replicate Production Configuration:

Replicate the configuration of the production environment as closely as possible, including the server setup, network architecture, and infrastructure components. This ensures that the security tests accurately reflect real-world scenarios.

Utilize Virtualization or Containerization:

Leverage virtualization technologies like virtual machines or containerization platforms (e.g., Docker) to create a scalable and reproducible testing environment. This enables the easy setup of multiple testing instances and facilitates efficient testing of different API configurations.

Identifying the Scope of Testing

Defining the scope of API security testing is vital to ensure focused efforts and comprehensive coverage. Consider the following factors:

APIs and Endpoints:

Determine which APIs and specific endpoints will be included in the testing. It's essential to consider internal and external-facing APIs and any public APIs that may expose sensitive data or critical functionalities.

Authentication and Authorization Mechanisms:

Assess the APIs' various authentication and authorization mechanisms. Include scenarios like API keys, tokens, or user credentials to evaluate the security measures thoroughly.

Data Validation and Input Handling:

Analyze how APIs handle data validation and input handling. Assess how they respond to input formats, including invalid or unexpected data. Pay special attention to potential injection vulnerabilities.

Error Handling and Exception Management:

Evaluate how APIs handle errors and exceptions. Test their response to different error conditions and ensure that sensitive information is not leaked in error messages.

Gathering Necessary Tools and Resources

Equipping your API security testing efforts with the right tools and resources is essential for effective testing. Consider the following:

API Testing Tools:

Explore and select appropriate API testing tools that support security testing, such as OWASP ZAP, Burp Suite, or Postman. These tools offer features like vulnerability scanning, fuzz testing, and API traffic interception for comprehensive testing.

Security Testing Frameworks:

Familiarize yourself with security testing frameworks, such as the OWASP API Security Top 10, which guides the most critical API security risks. These frameworks serve as invaluable references throughout the testing process.

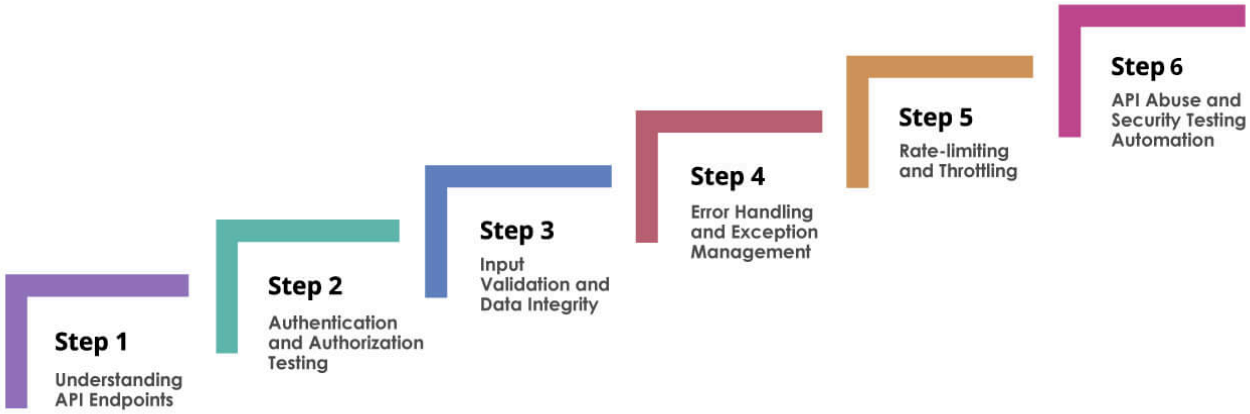
Documentation and Specifications:

Obtain the API documentation, specifications, and relevant security requirements. Thoroughly review them to understand the expected behavior, expected security measures, and any specific test cases.

Security Testing Checklist:

Develop a comprehensive security testing checklist encompassing various aspects of API security, including authentication, authorization, input validation, error handling, and encryption. This checklist will serve as a roadmap for your testing efforts.

Steps to Follow for API Security Testing



Step 1: Understanding API Endpoints

Objective: Ensure endpoints are secured and not publicly exposed unnecessarily.

Methods: Use API gateway, limit endpoint exposure.

Tests: Enumerate API endpoints, verify the necessity of exposed endpoints, and check for hidden or undocumented endpoints.

API endpoints serve as the entry points for interactions with an API. To conduct effective security testing:

Define API Endpoints:

Identify and document all API endpoints. Categorize them based on functionality, sensitivity, and potential security risks.

Identify Sensitive Endpoints and Vulnerabilities:

Determine which endpoints handle sensitive data, perform critical operations, or involve user authentication. These endpoints may be more prone to security vulnerabilities and require rigorous testing.

Map API Endpoints:

Create a comprehensive map of API endpoints, including the request and response types, expected behavior, and associated security controls. This map will serve as a reference during testing to ensure thorough coverage.

Step 2: Authentication and Authorization Testing

Authentication and authorization mechanisms play a vital role in securing APIs. When testing, consider the following:

Evaluate Authentication Effectiveness:

Assess the strength and effectiveness of authentication mechanisms, such as API keys, tokens, or multifactor authentication. Verify that only authenticated users can access protected resources.

Examine Authorization Controls:

Test the authorization controls to ensure only authorized users or roles can perform specific actions. Pay attention to privilege escalation risks, such as bypassing authorization checks or gaining unauthorized access to sensitive data.

Test for Improper Access Controls:

Identify potential security misconfigurations or improper access controls that may allow unauthorized access to sensitive endpoints or operations. Thoroughly examine access control rules and configurations.

Step 3: Input Validation

Objective: Protect the API from malicious input.

Methods: Implement whitelisting, sanitization, and validation of inputs.

Tests: Inject various forms of input such as SQL injection, XSS (cross-site scripting), command injection, and XML external entity (XXE) attacks.

Here some tips on how to implement input validation for your APIs to make sure it's as secure as possible.

1. Validate Content-Type Header and Data Format

The Content-Type header indicates what media type will be transmitted from an HTTP request or response. Verifying the Content-Type to ensure that the posted data matches the expected format is one of the most straightforward ways to protect an API from invalid or malicious data.

Imagine an API endpoint that expects a JSON object for a specific application. Verifying the header and that the request is in JSON format protects your API from data corruption or injection attacks, among other things.

2. Prevent Entity Expansion

Denial of service (DoS) attacks like A Billion Laughs or XML bomb attacks rely on vulnerable XML parsers. The attack involves sending an XML file with a large number of nested entities, resulting in the XML parser expanding each entity and consuming an excessive amount of resources. Simply limiting the number of entities that can be opened by the parser eliminates this type of DoS attack entirely.

3. Limit the Size of Posted Data

When users can interact with your API via input forms, file uploads, or POST requests, it's important to restrict how much data can be sent. Prohibitively large files can consume excess resources, increase processing time, or even, in some circumstances, cause an API to crash.

4. Compare User Input Against Injection Flaws

Injection flaws are pervasive in SQL, LDAP, or NoSQL queries, OS commands, XML parsers, and ORM. They're very easy to discover, as well, using tools like scanners or fuzzers. Injection flaws are when an attacker can run malicious code through an external application. This can result in both a backend being compromised as well as third-party clients connected to the affected application.

To help prevent the risk due to injection flaws, you should validate input from every potentially untrusted source. This includes everything from internet-facing web clients to backend feeds over extranets and data from suppliers, vendors, or regulators. Any of these could be compromised, putting your system at risk.

5. Validate All Levels

To help ensure API security, input validation should be performed on both the syntactical and semantic levels. Syntactic level validation should check to see that structured fields are formatted correctly, such as ensuring the right currency symbols are used, for example, or that the proper hyphenated structure is followed for things like phone or social security numbers. Semantic input validation checks to ensure data falls within a specific context, such as a particular date or price range.

6. Choose The Right Implementation For Your Programming Language

Most programming languages and frameworks have some method for input validation. Django supports Django Validators, for instance, while Apache Commons Validators are useful for Java-based applications. Java also allows for type conversion, using functions like Integer.parseInt(), as does Python, using int(). Input can be checked against JSON and XML schema, as well. Arrays can be used for small sets of parameters, like days of the week or time of day.

7. Use Allow Lists Instead Of Block Lists

Many developers use block lists to try and prevent common attack patterns, such as the apostrophe ' character, the 1=1 string, or <script> tags. This approach is easy for attackers to circumvent, though. Valid inputs can trigger the filter, too, as in the case of last names like O'Grady.

Using allow lists is a better approach for user inputs. They let you specify what is allowed rather than trying to prevent any potential risk. This is easy to implement for structured data, like addresses or social security numbers. Input fields with limited options, like a drop-down menu, are even easier to validate, as the selection needs to exactly match one of the available options.

8. Validate Free-Form UNICODE Input Properly

Free-form inputs, such as text, have a reputation of being the most difficult to validate, as there's such a wide range of possible variables. To validate free-form UNICODE inputs, you should practice normalization to ensure no invalid characters are present. You should create an allow-list for acceptable

character categories, such as Arabic or Cyrillic alphabets. You can also create allow-lists for individual characters, like the apostrophe in a name field.

9. Validate on the Server Side As Well As the Client Side

JavaScript input validation is easily bypassed by attackers, who can either disable JavaScript or use a Web Proxy. Validating input on the client side adds an invaluable extra layer of protection.

10. Manage User Uploads Properly

We already mentioned how attackers can use uploads for DoS attacks. Putting size limits on user uploads is just one approach for validating user uploads, though. If your API only accepts certain file types, you can use input validation to make sure the uploaded file matches the expected file type. Once the file's uploaded, though, make sure to change the file's name in your system. They should also be analyzed for malicious content, such as malware. The user shouldn't be able to dictate where the file is stored, either.

Step 4: Ensure Data Integrity

Verify that the API performs proper validation, sanitization, and encoding of user-supplied data to prevent data corruption or tampering. Validate the integrity of data transmitted between the client and server.

Test for Data Leakage and Exposure Risks:

Identify potential data leakages risks, such as inadvertently disclosing sensitive information in responses or error messages. Test for scenarios where sensitive data may be unintentionally exposed.

Step 4: Error Handling and Exception Management

Objective: Ensure errors and logs do not expose sensitive information.

Methods: Implement proper error messages and logging practices.

Tests: Generate various errors and examine responses for sensitive data leakage, verify logging configurations, and check for proper log sanitization.

Proper error handling and exception management improve an API's overall security and robustness. When conducting testing:

Assess Error Handling Mechanisms:

Evaluate how the API handles errors and exceptions. Test for proper error codes, informative messages, and appropriate logging practices.

Test for Information Disclosure Vulnerabilities:

Look for potential information disclosure vulnerabilities in error responses or stack traces. Ensure that error messages do not expose sensitive information that could aid attackers.

Evaluate Exception Management Practices:

Assess how the API handles unexpected situations, such as unhandled exceptions or denial-of-service attacks. Verify that the API gracefully handles exceptions and does not expose system vulnerabilities.

Step 5: Rate-limiting and Throttling

Objective: Prevent abuse of API services through excessive requests.

Methods: Implement rate limiting, quota management, and throttling policies.

Tests: Simulate high request rates, verify the enforcement of rate limits, and check for DoS (Denial of Service) vulnerabilities. Rate limiting and throttling mechanisms protect APIs against abuse and denial-of-service attacks. During testing:

Understand the Importance of Rate Limiting:

Recognize the significance of rate limiting and throttling to prevent abuse, brute-force attacks, or DoS scenarios. Familiarize yourself with industry best practices.

Test for Bypassing Rate Limits:

Attempt to bypass rate limits and verify if the API enforces them consistently. Check for potential vulnerabilities that allow attackers to circumvent rate limits and overload the system.

Verify Effectiveness of Rate-limiting Mechanisms:

Test the API under various load conditions to ensure that rate-limiting and throttling mechanisms function as expected. Measure the API's response time and stability during high-volume traffic.

API Rate Limiting vs. API Throttling: Comparison Table

FEATURE	API RATE LIMITING	API THROTTLING
Primary Objective	To prevent abuse and ensure equitable access to resources by limiting the number of requests a user or service can make within a specified time frame.	To maintain system performance and stability by dynamically adjusting the rate of incoming requests based on current system load or predefined rules.
Control Mechanism	Implements fixed thresholds for the number of requests that can be made over a set time period (e.g., 1000 requests per hour per user).	Utilizes algorithms to dynamically adjust request limits in real time, based on system performance metrics and predefined priorities.

Operational Focus	Focuses on enforcing fairness and preventing system overload by setting clear, predefined limits on API usage.	Aims to optimize the flow of requests to prevent system overloads while accommodating fluctuating demand and ensuring critical requests are processed efficiently.
User Impact	Once the rate limit is reached, further requests are blocked until the limit resets, potentially leading to service interruptions for the user.	Throttling may slow down the processing of requests rather than outright blocking them, leading to increased response times but generally avoiding complete service interruptions.
Flexibility	Generally less flexible, as limits are fixed and do not adapt to changes in system load or demand.	More flexible and adaptive, as it can adjust allowed request rates in real time based on the current system load, demand, and predefined rules.
Implementation Detail	Often implemented using a token bucket or fixed window counter algorithm, which allows for a simple and straightforward enforcement of rate limits.	May use complex algorithms that take into account current system load, user priorities, and other factors to dynamically adjust request handling, such as weighted fair queuing.
Feedback Mechanisms	Typically provides feedback to the user via HTTP headers, informing them of their current rate limit status, including the remaining number of requests and time until the limit resets.	Feedback mechanisms may include HTTP headers indicating current request processing times and expected delays, allowing users to adjust their request patterns accordingly.
Use Cases	Ideal for APIs where it is crucial to prevent abuse and ensure that all users or services have equitable access to resources, such as public web services or multi-tenant applications.	Best suited for systems where load can vary dramatically and where maintaining performance and availability is critical, such as real time applications or services with highly variable demand.

Limit Adjustment	Rate limits are typically configured statically but can be adjusted manually as needed based on long-term trends in usage or system capacity.	Throttling rules can be adjusted automatically in real-time, allowing the system to respond immediately to changes in load or performance conditions without manual intervention

Step 6: API Abuse and Security Testing Automation

Leverage automation techniques for API security testing to maximize efficiency and coverage. When testing, consider the following:

Explore Techniques to Identify and Prevent API Abuse:

Learn about common API abuse scenarios, such as parameter tampering, replay attacks, or API key exposure. Develop test cases to identify and mitigate these risks.

Implement Automated Security Testing:

Utilize automated tools and frameworks, such as OWASP ZAP or Burp Suite, to streamline security testing efforts. Automate vulnerability scanning, fuzz testing, and security checks to achieve comprehensive coverage.

Leverage Tools and Frameworks:

Leverage open-source tools and frameworks tailored for API security testing. These resources provide a wealth of knowledge, best practices, and test scripts to enhance the effectiveness of your testing efforts.

Tools for API Security Testing:

- **Burp Suite**: Comprehensive web vulnerability scanner with plugins for API testing.
- **OWASP ZAP**: Open-source tool for finding vulnerabilities in web applications and APIs.
- **Postman**: API development environment with security testing capabilities.
- **SoapUI**: Functional testing tool for SOAP and REST APIs.
- **Insomnia**: API client for testing and debugging with security testing extensions.
- **Fiddler**: Web debugging proxy for capturing and manipulating API requests and responses.
- **JMeter**: Performance testing tool with capabilities for API security testing.

Step 7: Session Management Testing

Objective: Ensure secure session handling mechanisms.

Methods: Implement secure cookie attributes, session timeouts.

Tests: Verify session expiration, test session fixation and hijacking, and ensure proper session termination.

Step 8: Business Logic Testing

Objective: Ensure the API's business logic is secure and robust.

Methods: Review business logic for flaws.

Tests: Perform scenario-based testing to identify logic flaws, race conditions, and bypasses.

Best Practices for API Security Testing

- **Shift Left Security**: Integrate security testing early in the development lifecycle.
- **Use Security Standards**: Follow OWASP API Security Top 10 guidelines.
- **Regular Audits and Penetration Testing**: Conduct periodic security audits and penetration tests.
- **Secure Development Practices**: Adopt secure coding practices and conduct code reviews.
- **Following Industry Standards and Guidelines:** To ensure robust API security, it is vital to follow industry standards and guidelines. These standards provide a framework for implementing adequate security controls and mitigating common vulnerabilities. By adhering to these standards, organizations can align their security practices with industry best practices and reduce the risk of potential breaches.
- **Keeping Up with Evolving Threats and Security Practices:** It is another critical aspect of API security. The threat landscape constantly evolves, with new attack vectors and techniques emerging regularly. Staying updated on the latest threats allows organizations to identify and address vulnerabilities proactively before they can be exploited. By actively participating in security communities, attending conferences, and leveraging threat intelligence sources, organizations can stay one step ahead of attackers and implement timely security measures.
- **Continuous Monitoring and Retesting for Ongoing Security**: These are essential for maintaining ongoing security. It is not enough to perform security testing once and consider the job done. APIs and their associated threats evolve. Organizations can detect and respond to potential security incidents in real-time by implementing continuous monitoring. Additionally, regular retesting helps identify new vulnerabilities that may have been introduced due to system updates or changes in the threat landscape. This iterative approach ensures that APIs remain secure and protected against emerging risks. : Automate security tests within CI/CD pipelines.

Common Interview Questions and answers related to API security testing

Authentication Testing

1. **Q: What is the difference between authentication and authorization?**

   A: Authentication verifies the identity of a user, while authorization determines what resources and actions the authenticated user is allowed to access.

2. **Q: How can you test if an API properly handles JWT (JSON Web Tokens)?**

   A: Validate token structure, check for proper expiration handling, ensure the token signature is verified, and test for token revocation.

3. **Q: What are common methods for API authentication?**

   A: OAuth, JWT, API keys, and basic authentication.

Authorization Testing

4. **Q: How do you test for privilege escalation in an API?**

   A: Attempt to access higher-privilege resources with lower-privilege credentials and verify that access is denied.

5. **Q: What is role-based access control (RBAC) in API security?**

   A: RBAC assigns permissions to users based on their roles within an organization, ensuring they can only access resources necessary for their role.

6. **Q: How can you ensure API endpoints are not publicly accessible?**

   A: Use API gateways, apply IP whitelisting, and implement proper access control policies.

Input Validation Testing

7. **Q: Why is input validation important for APIs?**

   A: It prevents malicious input that can lead to security vulnerabilities like SQL injection, XSS, and command injection.

8. **Q: What are common injection attacks you should test against in APIs?**

   A: SQL injection, XSS, command injection, and XML external entity (XXE) attacks.

9. **Q: How do you test for SQL injection vulnerabilities in an API?**

A: Inject SQL payloads into API parameters and observe if the API behaves unexpectedly or returns database errors.

Rate Limiting and Throttling Testing

10. **Q: What is rate limiting in API security?**

    A: Rate limiting controls the number of requests a client can make to the API within a specified time period to prevent abuse.

11. **Q: How can you test the effectiveness of rate limiting?**

    A: Simulate high request rates from a single client and verify that the API enforces rate limits by returning appropriate error responses (e.g., HTTP 429 Too Many Requests).

12. **Q: What is the difference between rate limiting and throttling?**

    A: Rate limiting restricts the number of requests over time, while throttling limits the bandwidth or processing capacity available to a client.

Error Handling and Logging Testing

13. **Q: Why is proper error handling important in APIs?**

    A: To avoid exposing sensitive information through error messages and ensure the API gracefully handles unexpected conditions.

14. **Q: How can you test for information leakage in error messages?**

    A: Intentionally cause errors and analyze the API responses for detailed error messages that reveal implementation details or sensitive information.

15. **Q: What should you check in API logs for security?**

    A: Ensure logs do not contain sensitive information like passwords or tokens, and verify that logging complies with privacy regulations.

Data Protection and Privacy Testing

16. **Q: How do you ensure data in transit is protected in an API?**

    A: Use encryption protocols such as TLS/SSL to encrypt data during transmission.

17. **Q: What are common methods for encrypting data at rest in an API?**

    A: Use encryption algorithms like AES to encrypt sensitive data stored in databases or file systems.

18. **Q: How can you test if data is properly encrypted in transit?**

    A: Inspect network traffic using tools like Wireshark to verify that data is encrypted during transmission.

API Endpoint Testing

19. **Q: How do you discover hidden or undocumented API endpoints?**

    A: Use tools like Burp Suite or OWASP ZAP to spider the API and look for endpoints that are not documented.

20. **Q: Why is it important to limit endpoint exposure? A:** To reduce the attack surface and prevent unauthorized access to sensitive resources.
21. **Q: What should you verify in API endpoint testing?**

    A: Ensure only necessary endpoints are exposed, endpoints require proper authentication and authorization, and input is properly validated.

Session Management Testing

22. **Q: How do you test for session fixation vulnerabilities in an API?**

    A: Attempt to set or reuse session identifiers and verify that the API properly generates new session IDs after authentication.

23. **Q: What is a secure cookie attribute and how is it tested?**

    A: Attributes like Secure and HttpOnly ensure cookies are only transmitted over HTTPS and are not accessible via JavaScript. Test by examining cookie properties in browser developer tools.

24. **Q: Why is session timeout important and how do you test it?**

    A: To minimize the risk of session hijacking. Test by leaving a session inactive for a period and verifying that the session is invalidated after the timeout.

Business Logic Testing

25. **Q: What is business logic testing in API security?**

    A: Testing the API's functionality to ensure it enforces business rules and processes correctly without security flaws.

26. **Q: How do you identify business logic vulnerabilities in an API?**

A: Perform scenario-based testing to simulate real-world usage and identify logic flaws, race conditions, and bypasses.

27. **Q: What are examples of business logic flaws in APIs?**

    A: Incorrectly processed transactions, unauthorized data modification, and failure to enforce workflow constraints.

General Security Practices

28. **Q: What is OWASP API Security Top 10?**

    A: A list of the top ten security risks for APIs published by the Open Web Application Security Project (OWASP), providing guidelines for securing APIs.

29. **Q: How can you integrate security testing into the development lifecycle?**

    A: Use a shift-left approach by incorporating security tests in CI/CD pipelines, conducting code reviews, and performing regular security assessments.

30. **Q: Why is continuous monitoring important for API security?**

    A: To detect and respond to security incidents in real-time, ensuring the API remains secure against emerging threats.

31. **Q:When is the best time to perform API security testing?**

    A: So when should you conduct API security testing? The best answer is "as early as possible." This means testing pre-production. Like other security testing done in software development, API security testing should "shift left," meaning it should move to the earliest possible stage in the development cycle. That's the time when developers are most likely to be familiar with the recent code they wrote rather than the code they wrote a month or 6 months ago. This way, testers can catch and remediate security issues before they go into production.

    Once an application is in production, it becomes more expensive and disruptive to fix a security problem. With CI/CD, a new vulnerability can go into production every hour, so it's really helpful to be on top of API security testing before code reaches the end of the CI/CD pipeline. A further best practice is to follow up with post-production API security testing. Security testing at this juncture catches security flaws that arise in production, but which may be difficult to detect in pre-production, such as production configuration issues.

32. **Q: Why you need API Security Testing?**

    A:The earlier you catch security vulnerabilities, the better. From both a cost perspective and remediation angle, it is much easier to correct issues during the development process of the API than after it has been released into production and is being actively used. It allows organizations to

more confidently and efficiently deliver applications to the business and remain competitive securely.

33. **Q: How to perform API security testing**?

    A: Developers, security teams and more, can now avail themselves of a new generation of API security testing tools. As exemplified by Noname Security Active Testing, they can run numerous dynamic API security tests on an application. Active Testing offers a purpose-built API security testing solution that takes into account the user's unique business logic. It provides comprehensive coverage of API-specific vulnerabilities, including the OWASP API Top Ten security issues. The suite can help align API security tests with business objectives and team structures. These latter two factors are important in making the "shift left" approach viable because making API security testing part of the dev cycle takes people and processes.

    API security testing is critical for protecting modern web applications in this era of CI/CD. It should occur as "far to the left" as possible in the development process. API testing should entail scanning for known API vulnerabilities, such as those referenced in the OWASP list, as well as other security problems. With the right testing tools, it is possible to conduct thorough API security testing early in development—detecting and remediating problems before they go into production.

Chapter 5

Writing Basic API Test Cases

Best Practices To Write A Good Test Case

With those test case templates downloaded, you can now start to document all of the test cases you are working on for a more structured and comprehensive view. Here are some best practices and tips to help you best utilize the template we provide:

- You can clone the template and have separate test case sheets for different areas of the software
- Follow a consistent naming convention for test cases to make them easily searchable.
- You can group similar test cases together under a common feature/scenario
- Familiarize yourself with the requirement or feature you're testing before creating the test case so that you'll know what information to include
- Use action verbs at the start of each test step like "Click", "Enter" or "Validate". If needed, you may even create a semantic structure to describe your test case. You can check out how it is done in BDD testing.
- Include any setup or prerequisites needed before executing the test.
- Ensure that the test cases you included are not only the "common" scenarios but also the negative scenarios that users don't typically face but do happen in the system
- Use formatting to make your test cases easier to read and follow
- Make sure to update your test cases regularly

Test Cases For API Functional Testing

Functionality is the core of any Application Under Test (AUT), and API is no exception. Their most basic and foundational functionality is data retrieval and data sending, and API functional testing should revolve around those 2 domains. Check out the following functional test cases and see how you can apply them to your own testing project:

1. **Status Code Validation for Valid Requests**: Verify that the API consistently returns the expected response status code, such as "200 OK," for valid and properly formatted requests.
2. **Authentication Handling with Invalid Credentials**: Test the API's response when provided with invalid authentication credentials, ensuring it consistently returns a "401 Unauthorized" status code as expected.
3. **Graceful Handling of Missing or Invalid Parameters:** Verify that the API handles missing or invalid request parameters gracefully and returns clear and user-friendly error messages that aid in troubleshooting.
4. **Input Data Validation with Malformed Data:** Test the API's input validation by submitting various forms of malformed data, such as invalid email formats, and confirm that it properly rejects and responds to these inputs.

5. **Timeout Handling under Load:** Confirm that the API correctly handles timeouts by simulating requests that take longer to process, ensuring that it remains responsive and does not hang.
6. **Pagination Functionality Verification:** Test the API's pagination functionality by requesting specific pages of results and verifying that the responses contain the expected data and pagination information.
7. **Concurrency Testing without Data Corruption:** Verify that the API handles concurrent requests from multiple users without data corruption or conflicts, ensuring data integrity.
8. **Response Format Adherence (JSON/XML):** Ensure that the API consistently returns responses in the specified format (e.g., JSON or XML) and adheres to the defined schema for data structure.
9. **Caching Mechanism Evaluation with Repeated Requests**: Evaluate the API's caching mechanism by making repeated requests and verifying that the cache headers are correctly set and honored.
10. **Rate Limiting Assessment:** Test the API's rate limiting by sending requests at a rate that exceeds the defined limits and checking for the expected rate-limiting responses, ensuring that limits are enforced.
11. **HTTP Method Support for CRUD Operations:** Verify that the API supports a variety of HTTP methods (GET, POST, PUT, DELETE) for Create, Read, Update, and Delete operations, and that it returns appropriate responses for each.
12. **Error Handling Capabilities for Meaningful Messages:** Evaluate the API's error-handling capabilities by intentionally causing errors, such as invalid inputs or unexpected situations, and confirm that it consistently returns meaningful error messages for troubleshooting.
13. **Conditional Request Handling (If-Modified-Since, If-None-Match):** Test the API's support for conditional requests using headers like If-Modified-Since and If-None-Match, ensuring that responses are handled appropriately.
14. **Sorting and Filtering Validation for Resource Listings:** Verify that the API correctly sorts and filters resource listings based on specified parameters, maintaining data accuracy.
15. **Handling Long or Complex Data without Data Corruption:** Ensure that the API properly handles long or complex strings, such as URLs or text fields, without truncating or corrupting the data.
16. **Content Negotiation Support for Multiple Formats:** Test the API's support for content negotiation by specifying different Accept headers (e.g., JSON, XML) and verifying that the response format matches the requested format.
17. **Resource Not Found Handling (404 Not Found):** Confirm that the API consistently returns the appropriate "404 Not Found" response when attempting to access a non-existent resource.
18. **Response Time Measurement for Various Requests:** Measure the API's response time for different types of requests to assess its performance and responsiveness.
19. **Handling Large Payloads (File Uploads):** Verify that the API can handle large payloads, such as file uploads, without encountering errors or significant performance degradation.
20. **Compatibility with Client Libraries and SDKs:** Evaluate the API's compatibility with different client libraries or SDKs to ensure seamless integration with various platforms and programming languages.

Introduction to Behavior-Driven Development (BDD)

Behavior-Driven Development (BDD) is a software development process that encourages collaboration among developers, testers, and business stakeholders. BDD extends Test-Driven Development (TDD) by writing test cases in a natural language that non-technical stakeholders can understand. Cucumber is one of the most popular tools for BDD, which allows you to write tests in a human-readable Gherkin syntax.

Writing Tests for CRUD Operations using BDD

POST

Postive Test

```
Feature: User Management
 Scenario: Create a new user
  Given the API endpoint is set to "/api/users"
  When I send a POST request with the following details
   | name  | job       |
   | John  | Developer |
  Then the response status code should be 201
  And the response body should contain the following
   | name  | job       |
   | John  | Developer |
```

Negative Test

```
Feature: Invalid Payload Handling for POST API
  In order to ensure the API correctly handles invalid payloads
  As an API consumer
  I want to validate various invalid payload scenarios

  Scenario: Missing required fields
    Given the API endpoint is set to "/api/resource"
    When I send a POST request with missing required fields
    Then the response status code should be 400
    And the response should contain an error message "Missing required fields"

  Scenario: Invalid data types
    Given the API endpoint is set to "/api/resource"
    When I send a POST request with invalid data types
    Then the response status code should be 400
    And the response should contain an error message "Invalid data types"

  Scenario: Extra unexpected fields
```

```
  Given the API endpoint is set to "/api/resource"
  When I send a POST request with extra unexpected fields
  Then the response status code should be 400
  And the response should contain an error message "Unexpected fields"

 Scenario: Empty payload
  Given the API endpoint is set to "/api/resource"
  When I send a POST request with an empty payload
  Then the response status code should be 400
  And the response should contain an error message "Empty payload"

 Scenario: Malformed JSON
  Given the API endpoint is set to "/api/resource"
  When I send a POST request with malformed JSON
  Then the response status code should be 400
  And the response should contain an error message "Malformed JSON"
```

PUT

Posive Test

```
Feature: User Management
 Scenario: Update an existing user
  Given the API endpoint is set to "/api/users/2"
  When I send a PUT request with the following details
    | name  | job      |
    | John  | Manager  |
  Then the response status code should be 200
  And the response body should contain the following
    | name  | job      |
    | John  | Manager  |
```

Negative Test

```
Feature: Invalid Payload Handling for PUT API
 In order to ensure the API correctly handles invalid payloads for PUT requests
 As an API consumer
 I want to validate various invalid payload scenarios

 Scenario: Missing required fields
  Given the API endpoint is set to "/api/resource/1"
  When I send a PUT request with missing required fields
  Then the response status code should be 400
  And the response should contain an error message "Missing required fields"
```

```
Scenario: Invalid data types
  Given the API endpoint is set to "/api/resource/1"
  When I send a PUT request with invalid data types
  Then the response status code should be 400
  And the response should contain an error message "Invalid data types"

Scenario: Extra unexpected fields
  Given the API endpoint is set to "/api/resource/1"
  When I send a PUT request with extra unexpected fields
  Then the response status code should be 400
  And the response should contain an error message "Unexpected fields"

Scenario: Empty payload
  Given the API endpoint is set to "/api/resource/1"
  When I send a PUT request with an empty payload
  Then the response status code should be 400
  And the response should contain an error message "Empty payload"

Scenario: Malformed JSON
  Given the API endpoint is set to "/api/resource/1"
  When I send a PUT request with malformed JSON
  Then the response status code should be 400
  And the response should contain an error message "Malformed JSON"

Scenario: Invalid resource ID
  Given the API endpoint is set to "/api/resource/invalidID"
  When I send a PUT request with valid payload
  Then the response status code should be 404
  And the response should contain an error message "Resource not found"

Scenario: Unauthorized access
  Given the API endpoint is set to "/api/resource/1"
  When I send a PUT request without authentication
  Then the response status code should be 401
  And the response should contain an error message "Unauthorized"
```

GET

Positive Test

```
Feature: User Management
  Scenario: Get an existing user
    Given the API endpoint is set to "/api/users/2"
    When I send a GET request
    Then the response status code should be 200
```

```
  And the response body should contain the following
    | id   | email              |
    | 2    | lamhot.siagian@reqres.ap |
```

Negative Test

```
Feature: Invalid Payload Handling for GET API
  In order to ensure the API correctly handles invalid scenarios for GET requests
  As an API consumer
  I want to validate various invalid GET request scenarios

  Scenario: Invalid resource ID
    Given the API endpoint is set to "/api/resource/invalidID"
    When I send a GET request
    Then the response status code should be 404
    And the response should contain an error message "Resource not found"

  Scenario: Unauthorized access
    Given the API endpoint is set to "/api/resource/1"
    When I send a GET request without authentication
    Then the response status code should be 401
    And the response should contain an error message "Unauthorized"

  Scenario: Forbidden access
    Given the API endpoint is set to "/api/resource/1"
    When I send a GET request with insufficient permissions
    Then the response status code should be 403
    And the response should contain an error message "Forbidden"

  Scenario: Invalid query parameters
    Given the API endpoint is set to "/api/resource"
    When I send a GET request with invalid query parameters
    Then the response status code should be 400
    And the response should contain an error message "Invalid query parameters"

  Scenario: Non-existent resource
    Given the API endpoint is set to "/api/resource/99999"
    When I send a GET request
    Then the response status code should be 404
    And the response should contain an error message "Resource not found"

  Scenario: Malformed request
    Given the API endpoint is set to "/api/resource/%G"
    When I send a GET request
    Then the response status code should be 400
    And the response should contain an error message "Malformed request"
```

PATCH

Positive Test

```
Feature: User Management
  Scenario: Update an existing user partially
    Given the API endpoint is set to "/api/users/2"
    When I send a PATCH request with the following details
      | name  | job       |
      | Jane  | Developer |
    Then the response status code should be 200
    And the response body should contain the following
      | name  | job       |
      | Jane  | Developer |
```

Negative Test

```
Feature: Invalid Payload Handling for PATCH API
  In order to ensure the API correctly handles invalid payloads for PATCH requests
  As an API consumer
  I want to validate various invalid payload scenarios

  Scenario: Missing required fields
    Given the API endpoint is set to "/api/resource/1"
    When I send a PATCH request with missing required fields
    Then the response status code should be 400
    And the response should contain an error message "Missing required fields"

  Scenario: Invalid data types
    Given the API endpoint is set to "/api/resource/1"
    When I send a PATCH request with invalid data types
    Then the response status code should be 400
    And the response should contain an error message "Invalid data types"

  Scenario: Extra unexpected fields
    Given the API endpoint is set to "/api/resource/1"
    When I send a PATCH request with extra unexpected fields
    Then the response status code should be 400
    And the response should contain an error message "Unexpected fields"

  Scenario: Empty payload
    Given the API endpoint is set to "/api/resource/1"
```

```
  When I send a PATCH request with an empty payload
  Then the response status code should be 400
  And the response should contain an error message "Empty payload"

Scenario: Malformed JSON
  Given the API endpoint is set to "/api/resource/1"
  When I send a PATCH request with malformed JSON
  Then the response status code should be 400
  And the response should contain an error message "Malformed JSON"

Scenario: Invalid resource ID
  Given the API endpoint is set to "/api
```

DELETE

Positive Test

```
Feature: User Management
  Scenario: Delete an existing user
    Given the API endpoint is set to "/api/users/2"
    When I send a DELETE request
    Then the response status code should be 204
```

Negative Test

Scenario 1: Attempting to Delete a Non-Existent Resource

Given a resource with ID `12345` does not exist in the system
When I send a DELETE request to `/resources/12345`
Then the response status code should be `404 Not Found`
And the response message should indicate that the resource was not found

Scenario 2: Deleting a Resource without Authorization

Given I am not authenticated
When I send a DELETE request to `/resources/67890`
Then the response status code should be `401 Unauthorized`
And the response message should indicate that authentication is required

Scenario 3: Deleting a Resource with Insufficient Permissions

Given I am authenticated as a user with insufficient permissions
When I send a DELETE request to `/resources/13579`
Then the response status code should be `403 Forbidden`
And the response message should indicate that I do not have permission to delete the resource

Scenario 4: Invalid Resource ID Format

Given the resource ID format is invalid
When I send a DELETE request to `/resources/invalid-id`
Then the response status code should be `400 Bad Request`
And the response message should indicate that the resource ID is invalid

Scenario 5: Missing Resource ID

Given the resource ID is missing in the request URL
When I send a DELETE request to `/resources/`
Then the response status code should be `404 Not Found`
And the response message should indicate that the resource ID is required

Scenario 6: Server Error During Deletion

Given there is a server error when trying to delete the resource
When I send a DELETE request to `/resources/24680`
Then the response status code should be `500 Internal Server Error`
And the response message should indicate that an error occurred on the server

Scenario 7: Deleting a Resource with Dependencies

Given the resource has dependent resources that prevent deletion
When I send a DELETE request to `/resources/97531`
Then the response status code should be `409 Conflict`
And the response message should indicate that the resource has dependencies and cannot be deleted

Scenario 8: Deleting a Resource After Timeout

Given the resource deletion request exceeds the allowable timeout period
When I send a DELETE request to `/resources/86420`
Then the response status code should be `504 Gateway Timeout`
And the response message should indicate that the request timed out

Scenario 9: Unsupported Media Type in Request

Given the DELETE request contains an unsupported media type in the header
When I send a DELETE request to `/resources/11223` with `Content-Type: application/xml`
Then the response status code should be `415 Unsupported Media Type`
And the response message should indicate that the media type is not supported

Scenario 10: Too Many Requests

Given I have exceeded the rate limit for DELETE requests
When I send a DELETE request to `/resources/44556`
Then the response status code should be `429 Too Many Requests`
And the response message should indicate that the rate limit has been exceeded

Response Format Validation Using BDD

```
Feature: User Management Response Format Validation
  In order to ensure the API returns the correct response format
  As an API consumer
  I want to validate the response format for user management endpoints

  Scenario: Validate the response format of GET user request
    Given the API endpoint is set to "/api/users/2"
    When I send a GET request
    Then the response status code should be 200
    And the response should have the following structure
      | id        | Integer  |
      | email     | String   |
      | first_name| String   |
      | last_name | String   |
      | avatar    | String   |

  Scenario: Validate the response format of POST user request
    Given the API endpoint is set to "/api/users"
    When I send a POST request with the following details
      | name  | job       |
      | John  | Developer |
    Then the response status code should be 201
    And the response should have the following structure
      | id        | Integer  |
      | name      | String   |
      | job       | String   |
      | createdAt | String   |
```

Sorting Validation using BDD

```
Feature: User Management Sorting Validation
  In order to ensure the API returns sorted data correctly
  As an API consumer
  I want to validate the sorting functionality for user management endpoints

  Scenario: Validate sorting of users by first name in ascending order
    Given the API endpoint is set to "/api/users"
    When I send a GET request with query parameter "sort=first_name&order=asc"
    Then the response status code should be 200
    And the users should be sorted by "first_name" in ascending order

  Scenario: Validate sorting of users by last name in descending order
    Given the API endpoint is set to "/api/users"
    When I send a GET request with query parameter "sort=last_name&order=desc"
    Then the response status code should be 200
    And the users should be sorted by "last_name" in descending order
```

Pagination Validation Using BDD

```
Feature: User Management Pagination Validation
  In order to ensure the API handles pagination correctly
  As an API consumer
  I want to validate the pagination functionality for user management endpoints

  Scenario: Validate pagination of users
    Given the API endpoint is set to "/api/users"
    When I send a GET request with query parameters "page=1&per_page=3"
    Then the response status code should be 200
    And the response should contain 3 users
    And the pagination metadata should be correct
```

Authentication Handling Validation Using BDD

```
Feature: Authentication Handling
  In order to protect resources
  As an API consumer
  I want to ensure authentication mechanisms are working correctly
```

```
Scenario: Successful login
  Given the API endpoint is set to "/api/login"
  When I send a POST request with the following credentials
    | username | password |
    | testuser | password123 |
  Then the response status code should be 200
  And the response should contain a token

Scenario: Login with invalid credentials
  Given the API endpoint is set to "/api/login"
  When I send a POST request with the following credentials
    | username | password |
    | testuser | wrongpassword |
  Then the response status code should be 401
  And the response should contain an error message "Invalid credentials"

Scenario: Access protected resource with valid token
  Given I have a valid authentication token
  And the API endpoint is set to "/api/protected"
  When I send a GET request with the token
  Then the response status code should be 200
  And the response should contain the protected data

Scenario: Access protected resource without token
  Given the API endpoint is set to "/api/protected"
  When I send a GET request without a token
  Then the response status code should be 401
  And the response should contain an error message "Unauthorized"

Scenario: Access protected resource with expired token
  Given I have an expired authentication token
  And the API endpoint is set to "/api/protected"
  When I send a GET request with the expired token
  Then the response status code should be 401
  And the response should contain an error message "Token expired"
```

Authorization Validation Using BDD

```
Feature: Authorization Handling
  In order to protect resources based on user roles
  As an API consumer
  I want to ensure authorization mechanisms are working correctly

  Scenario: Access admin resource with admin role
    Given the API endpoint is set to "/api/admin/resource"
    And I have a valid admin authentication token
    When I send a GET request with the token
```

```
  Then the response status code should be 200
  And the response should contain the admin resource data

Scenario: Access admin resource with user role
  Given the API endpoint is set to "/api/admin/resource"
  And I have a valid user authentication token
  When I send a GET request with the token
  Then the response status code should be 403
  And the response should contain an error message "Forbidden"

Scenario: Access user resource with user role
  Given the API endpoint is set to "/api/user/resource"
  And I have a valid user authentication token
  When I send a GET request with the token
  Then the response status code should be 200
  And the response should contain the user resource data

Scenario: Access user resource with admin role
  Given the API endpoint is set to "/api/user/resource"
  And I have a valid admin authentication token
  When I send a GET request with the toke
```

Input Validation Using DDD

For a POST API where a field, such as `name`, requires input validation, it's important to cover a range of invalid input scenarios to ensure data integrity and robust error handling. Here are various invalid input validation scenarios for the `name` field in a BDD (Behavior-Driven Development) format:

Scenario 1: Missing Name Field

Given the request body is missing the `name` field
When I send a POST request to `/resources` with an empty `name` field
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field is required

Scenario 2: Name Field Exceeds Maximum Length

Given the `name` field has a maximum length of 50 characters
When I send a POST request to `/resources` with a `name` that is 51 characters long
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field exceeds the maximum length

Scenario 3: Name Field Contains Special Characters

Given the `name` field should only contain alphabetic characters
When I send a POST request to `/resources` with a `name` that contains special characters like `@`, `#`, `$`
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field contains invalid characters

Scenario 4: Name Field is Numeric

Given the `name` field should not be numeric
When I send a POST request to `/resources` with a `name` that is a number like `12345`
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field should be alphabetic

Scenario 5: Name Field is Null

Given the `name` field should not be null
When I send a POST request to `/resources` with a `name` field set to `null`
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field cannot be null

Scenario 6: Name Field is an Empty String

Given the `name` field should not be an empty string
When I send a POST request to `/resources` with a `name` field that is an empty string
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field cannot be empty

Scenario 7: Name Field Contains HTML/Script Tags

Given the `name` field should not contain HTML or script tags
When I send a POST request to `/resources` with a `name` field that contains
`<script>alert('XSS')</script>`
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field contains invalid characters

Scenario 8: Name Field Contains SQL Injection Attempt

Given the `name` field should be validated against SQL injection attempts
When I send a POST request to `/resources` with a `name` field that contains `'; DROP TABLE users; --`
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field contains invalid characters

Scenario 9: Name Field is Too Short

Given the `name` field should have a minimum length of 3 characters
When I send a POST request to `/resources` with a `name` field that is 2 characters long

Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field is too short

Scenario 10: Name Field Contains Leading or Trailing Spaces

Given the `name` field should not contain leading or trailing spaces
When I send a POST request to `/resources` with a `name` field that contains leading or trailing spaces
Then the response status code should be `400 Bad Request`
And the response message should indicate that the `name` field cannot contain leading or trailing spaces

These scenarios ensure that various invalid inputs for the `name` field are thoroughly tested, helping to maintain data integrity and security in the system.

Common Interview Questions and answers related to Writing Basic API Test Cases

Here are 30 popular interview questions and answers related to POST, PUT, GET, PATCH, DELETE, response format validation, sorting validation, pagination validation, authentication handling validation, and input validation for API testing:

Questions and Answers on HTTP Methods (POST, PUT, GET, PATCH, DELETE)

1. **Q: What is the difference between POST and PUT methods?**
   - **A:** POST is used to create a new resource, while PUT is used to update an existing resource or create a resource if it does not exist. POST requests are not idempotent, meaning subsequent identical requests will result in different outcomes, whereas PUT requests are idempotent.
2. **Q: When should you use the PATCH method?**
   - **A:** PATCH is used to make partial updates to a resource. Unlike PUT, which requires the entire resource data for the update, PATCH allows updating only the fields that need to be changed.
3. **Q: How does the DELETE method work in RESTful APIs?**
   - **A:** DELETE is used to remove a specified resource from the server. The server should respond with a status code indicating the result of the operation, typically `204 No Content` for a successful deletion.
4. **Q: What is the purpose of the GET method in RESTful services?**
   - **A:** The GET method is used to retrieve data from the server. It should not alter the state of the resource, making it a safe and idempotent operation.
5. **Q: Can you explain idempotency with an example?**
   - **A:** Idempotency means that making the same request multiple times will result in the same outcome. For example, sending a PUT request to update a user's email address to "example@example.com" will result in the email being set to "example@example.com" regardless of how many times the request is made.

Questions and Answers on Response Format Validation

6. **Q: How do you validate the response format of an API?**
    - **A:** To validate the response format, you can check the Content-Type header to ensure it matches the expected format (e.g., `application/json`). Additionally, you can use JSON schema validation to ensure the structure and data types in the response match the expected schema.
7. **Q: What is the significance of the Content-Type header in API responses?**
    - **A:** The Content-Type header indicates the media type of the resource. It helps the client understand how to process the data in the response, for example, `application/json` for JSON data.
8. **Q: How would you handle a response that is not in the expected format?**
    - **A:** If the response is not in the expected format, you can log the error, notify the client with an appropriate error message, and potentially raise an exception. It's also important to validate the API implementation and ensure the server is configured to return the correct format.

Questions and Answers on Sorting Validation

9. **Q: How do you test sorting functionality in an API?**
    - **A:** To test sorting, you can send requests with different sort parameters (e.g., `?sort=name&order=asc`) and verify that the returned data is correctly sorted according to the specified fields and order.
10. **Q: What are common issues you might encounter with sorting in APIs?**
    - **A:** Common issues include incorrect sorting order, case sensitivity affecting the sort order, improper handling of null values, and performance degradation with large datasets.

Questions and Answers on Pagination Validation

11. **Q: What is pagination and why is it important in APIs?**
    - **A:** Pagination is the process of dividing a large dataset into smaller, manageable chunks. It is important in APIs to improve performance and reduce the load on the server and client by only retrieving a subset of the data at a time.
12. **Q: How would you validate pagination in an API?**
    - **A:** To validate pagination, you can verify the response metadata (e.g., total items, total pages), check the consistency of data across pages, ensure correct handling of page limits and offsets, and validate edge cases like the first and last pages.

Questions and Answers on Authentication Handling Validation

13. **Q: What are common methods for API authentication?**
    - **A:** Common methods include Basic Authentication, Token-based Authentication (e.g., JWT), OAuth, and API Keys.
14. **Q: How do you test API endpoints that require authentication?**

- **A:** To test authenticated endpoints, you can include valid and invalid credentials in the request, check for the appropriate status codes (`200 OK` for successful authentication, `401 Unauthorized` for missing/invalid credentials), and validate the response data.

15. **Q: What is the difference between authentication and authorization?**
    - **A:** Authentication verifies the identity of a user or client, while authorization determines the permissions and access rights of the authenticated user to specific resources or actions.

Questions and Answers on Input Validation

16. **Q: Why is input validation important in APIs?**
    - **A:** Input validation ensures that the data received by the API is in the correct format, prevents security vulnerabilities like SQL injection, and improves data integrity by ensuring only valid data is processed.
17. **Q: How would you validate an input field in an API request?**
    - **A:** You can validate input fields by checking data types, ensuring required fields are present, enforcing value ranges and formats, and using regular expressions for pattern matching (e.g., email addresses).
18. **Q: How do you handle invalid input in an API?**
    - **A:** When invalid input is detected, the API should return a `400 Bad Request` status code with a descriptive error message indicating which fields are invalid and why.

Questions and Answers on General API Testing

19. **Q: What is the purpose of API testing?**
    - **A:** API testing ensures that the API behaves as expected, meets functional and non-functional requirements, handles edge cases gracefully, and provides secure and reliable interactions between different software components.
20. **Q: What tools can you use for API testing?**
    - **A:** Popular tools for API testing include Postman, SoapUI, JMeter, RestAssured, and Swagger.
21. **Q: How do you ensure your API tests are comprehensive?**
    - **A:** Ensure comprehensive testing by covering various scenarios such as happy paths, edge cases, negative tests, performance tests, security tests, and testing different combinations of input parameters.
22. **Q: What is a mock API and when would you use it?**
    - **A:** A mock API simulates the behavior of a real API. It is used during development and testing when the real API is not available, to test client-side code or to validate the integration between components.
23. **Q: How do you handle rate limiting in API testing?**
    - **A:** To handle rate limiting, you can include test cases to validate the API's response to excessive requests, check the appropriate status code (`429 Too Many Requests`), and verify the Retry-After header for the wait time before subsequent requests can be made.
24. **Q: What are common security issues to test for in APIs?**

- **A:** Common security issues include SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), improper authentication/authorization, and data leakage through improper error handling.

Questions and Answers on Specific Use Cases

25. **Q: How do you validate a POST request with a complex JSON payload?**
    - **A:** Validate a POST request with a complex JSON payload by checking each field's presence, data type, value range, format, and ensuring nested structures conform to the expected schema.
26. **Q: What status code should be returned for a successful POST request?**
    - **A:** A successful POST request should return a `201 Created` status code, indicating that the resource has been successfully created.
27. **Q: How do you handle versioning in APIs?**
    - **A:** Handle versioning by including version information in the URL (e.g., `/v1/resources`), in the request header, or as a query parameter. This allows clients to specify which version of the API they want to use.
28. **Q: What is the purpose of the OPTIONS HTTP method?**
    - **A:** The OPTIONS HTTP method is used to describe the communication options for the target resource. It is commonly used for CORS (Cross-Origin Resource Sharing) preflight requests.
29. **Q: How do you test the resilience of an API?**
    - **A:** Test the resilience of an API by simulating network issues, server failures, high loads, and other adverse conditions. Validate the API's ability to handle these scenarios gracefully and recover.
30. **Q: What is HATEOAS and why is it important in RESTful APIs?**
    - **A:** HATEOAS (Hypermedia As The Engine Of Application State) is a constraint of RESTful APIs where the client interacts with the application entirely through hypermedia provided dynamically by application servers. It is important because it provides discoverability and navigability of the API, allowing clients to dynamically follow links to other resources and actions.

These questions and answers should provide a solid foundation for understanding various aspects of API testing, including HTTP methods, response validation, sorting, pagination, authentication, and input validation.

Chapter 6

Automating API Tests

TestNG BDD API Testing

BDD (Behavior-Driven Development) is a software development approach that enhances collaboration between developers, testers, and business stakeholders. It involves writing test cases in a natural language that non-programmers can read. Combining TestNG with BDD for API testing provides a structured approach to testing while leveraging the powerful features of TestNG.

In this guide, we'll integrate BDD into API testing using TestNG and the Rest Assured library.

Prerequisites

1. **Java**: Ensure Java is installed and configured on your machine.
2. **Maven**: Maven should be installed for managing dependencies.
3. **IDE**: Use an IDE like IntelliJ IDEA or Eclipse.

Setting Up the Project

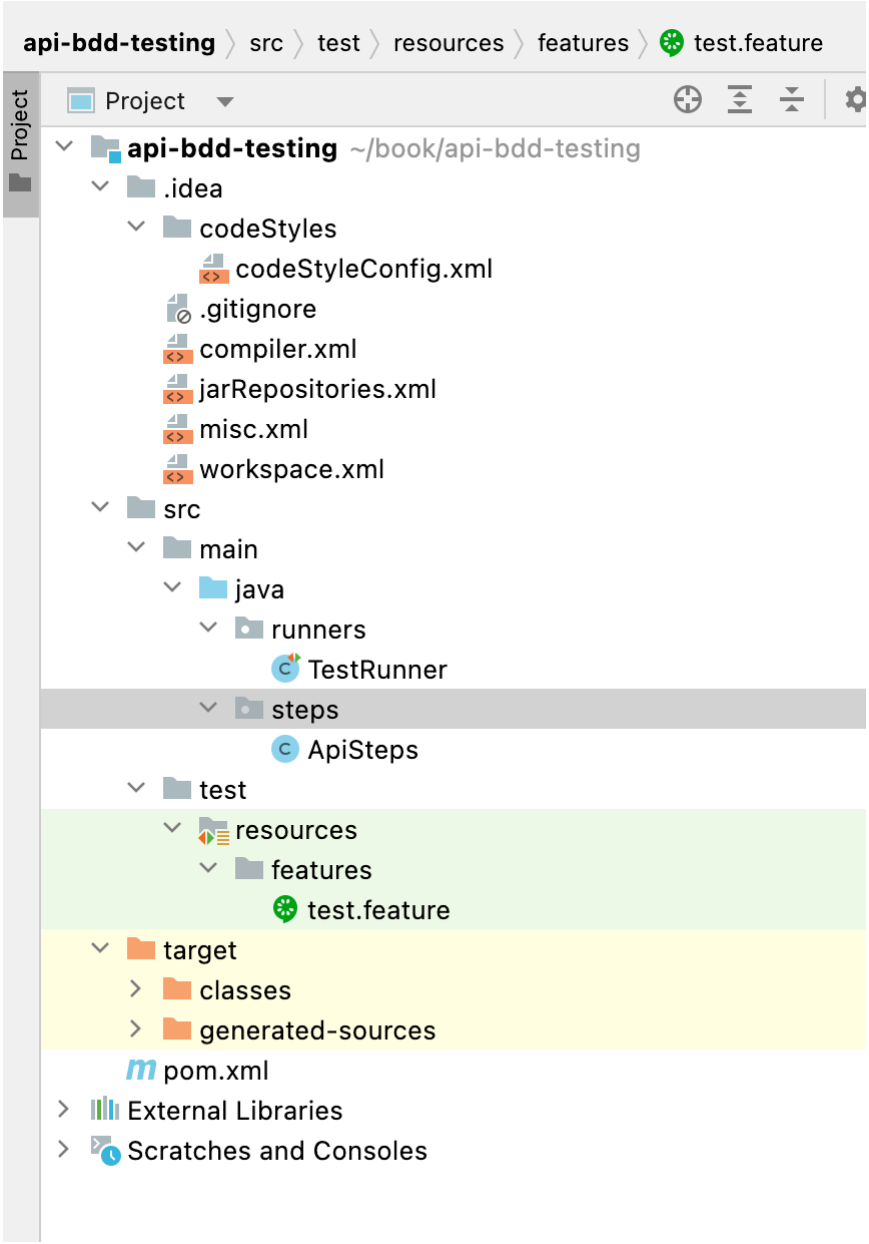
1. **Create a Maven Project**: Open your IDE and create a new Maven project.
2. **Add Dependencies**: Update your `pom.xml` with the necessary dependencies for TestNG, Rest Assured, and Cucumber (for BDD).

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>bdd-api-automation</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- TestNG Dependency -->
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>7.4.0</version>
            <scope>test</scope>
        </dependency>
        <!-- Rest Assured Dependency -->
        <dependency>
```

```
            <groupId>io.rest-assured</groupId>
            <artifactId>rest-assured</artifactId>
            <version>4.3.3</version>
        </dependency>
        <!-- Cucumber Dependencies -->
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-java</artifactId>
            <version>6.10.4</version>
        </dependency>
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-testng</artifactId>
            <version>6.10.4</version>
        </dependency>
        <!-- JSON Schema Validator Dependency -->
        <dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>json-schema-validator</artifactId>
            <version>4.3.3</version>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>7.4.0</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>
</project>
```

Project Structure

1. **src/test/java**: For Java test classes.
2. **src/test/resources**: For feature files.

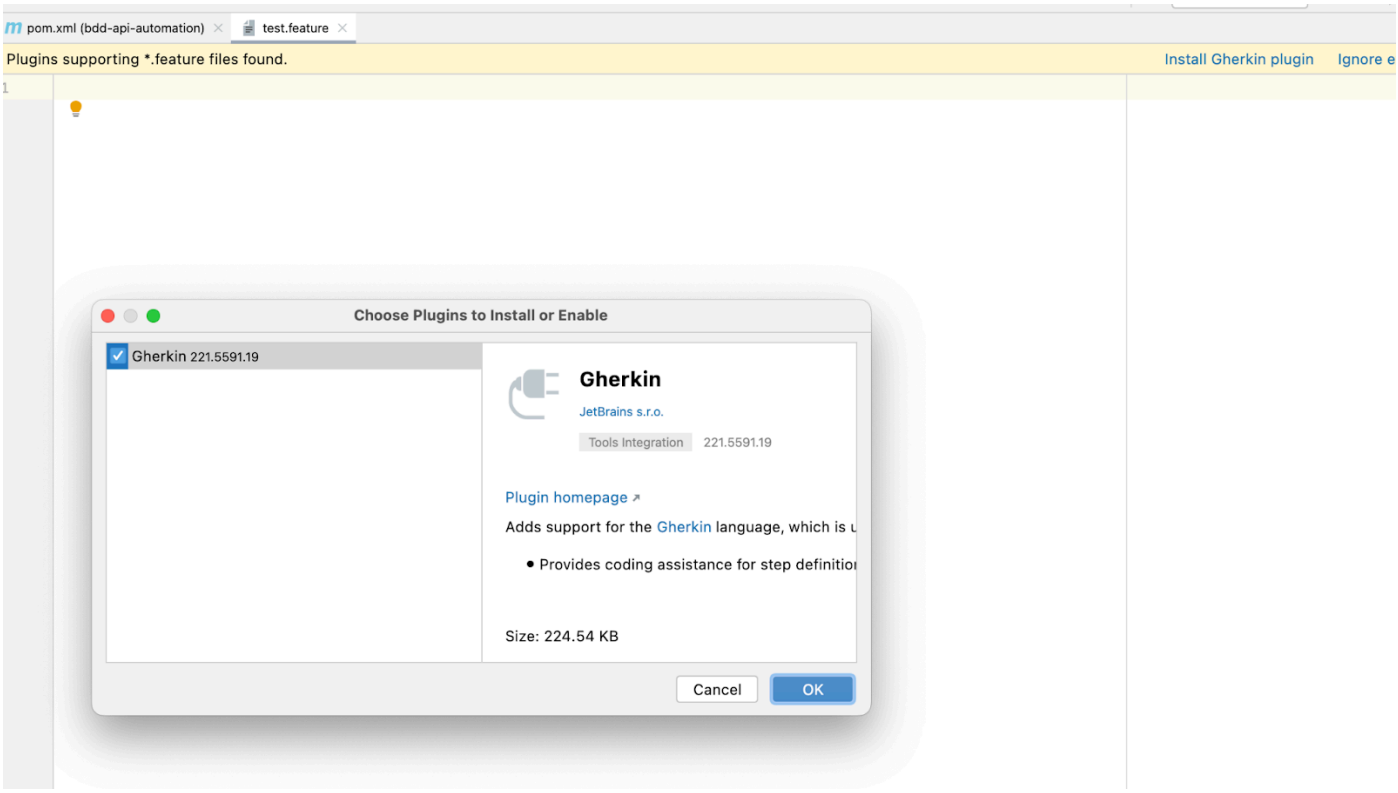


Writing Feature Files

Create a feature file in `src/test/resources/features`.

```
Feature: User API Testing

 Scenario: Get Users
   Given the API endpoint is "https://reqres.in/api/users?page=2"
   When I send a GET request to the endpoint
   Then the response status code should be 200
   And the response should contain "page"
```

Install Gherkin plugin:



Writing Step Definitions

Create a new Java class for step definitions in `src/test/java/steps`.

```
package steps;

import io.restassured.RestAssured;
import io.restassured.response.Response;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import org.testng.Assert;

public class ApiSteps {

   private String endpoint;
   private Response response;

   @Given("the API endpoint is {string}")
   public void theApiEndpointIs(String url) {
       this.endpoint = url;
```

```
    }

    @When("I send a GET request to the endpoint")
    public void iSendAGetRequestToTheEndpoint() {
        response = RestAssured
                .given()
                .get(endpoint);
    }

    @Then("the response status code should be {int}")
    public void theResponseStatusCodeShouldBe(int statusCode) {
        Assert.assertEquals(response.getStatusCode(), statusCode);
    }

    @Then("the response should contain {string}")
    public void theResponseShouldContain(String content) {
        Assert.assertTrue(response.asString().contains(content));
    }
}
```

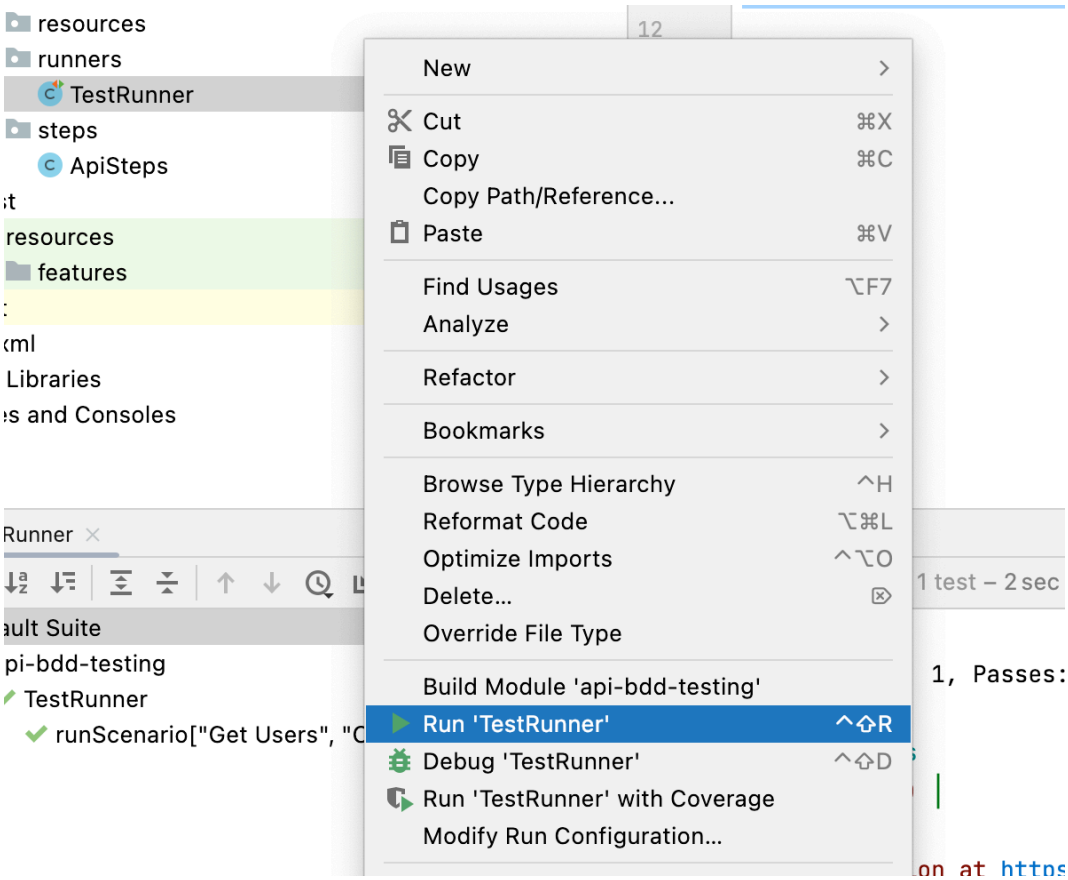
Writing the Test Runner

Create a test runner class in `src/test/java/runners`.

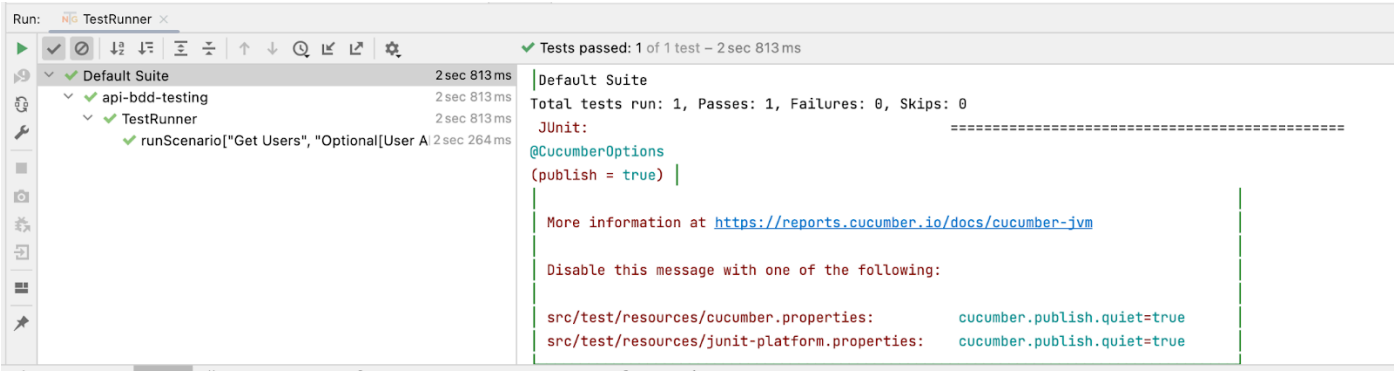
```
package runners;

import io.cucumber.testng.AbstractTestNGCucumberTests;
import io.cucumber.testng.CucumberOptions;

@CucumberOptions(
    features = "src/test/resources/features",
    glue = "steps"
)
public class TestRunner extends AbstractTestNGCucumberTests {
}
```

Running the Tests

1. **Run from IDE**: Right-click on the `TestRunner` class and select "Run."



You will see this result:



2. **Run from Maven**: You can also run the tests using Maven with the following command: `mvn test`

Common Interview Questions and Answers Related to Automation API Test

Why Choose TestNG for API Automation?

1. **Annotations**: TestNG provides a wide range of annotations which makes it easier to write tests and understand them.

2. **Test Configuration**: TestNG allows you to configure your test suites, groups, and test parameters easily through XML files.
3. **Parallel Execution**: TestNG supports parallel execution of tests, which is essential for speeding up the testing process.
4. **Data-driven Testing**: It supports data providers for data-driven testing.
5. **Integration**: TestNG can be easily integrated with other tools and frameworks like Maven, Jenkins, and various reporting tools.

How do you handle authentication in API testing?

Answer: Handling authentication in API testing depends on the type of authentication used:

- **Basic Authentication**: Include the username and password in the request header.
- **OAuth**: Use tokens obtained through authorization servers.
- **API Keys**: Include the API key in the request header or URL parameter.
- **JWT (JSON Web Tokens)**: Include the JWT token in the request header.

```
      // Basic Authentication
       given().auth().basic("username", "password").when().get("/endpoint");

// OAuth
       given().auth().oauth2("access_token").when().get("/endpoint");

// API Key
       given().header("API-Key", "your_api_key").when().get("/endpoint");

// JWT
       given().header("Authorization", "Bearer
your_jwt_token").when().get("/endpoint");
```

How do you validate the response of an API?

Answer: Validation involves checking the following:

- **Status Code**: Ensure the response code is as expected (e.g., 200 OK, 404 Not Found).
- **Response Body**: Validate the data returned in the response body.
- **Headers**: Check if the required headers are present and correct.
- **Schema Validation**: Ensure the response follows a predefined schema.

```
// Status Code
       Assert.assertEquals(response.getStatusCode(), 200);
```

```
// Response Body
        Assert.assertTrue(response.asString().contains("expectedValue"));

// Header
        Assert.assertEquals(response.getHeader("Content-Type"), "application/json");

// Schema Validation

response.then().assertThat().body(matchesJsonSchemaInClasspath("schema.json"));
```

How do you perform data-driven testing in API testing?

Answer: Data-driven testing involves running the same test case with multiple sets of data. This can be done using external data sources like Excel, CSV files, or databases.

Example with TestNG and DataProvider:

```
@DataProvider(name = "userData")
public Object[][] createUserData() {
    return new Object[][]{
            {"John", "Developer"},
            {"Jane", "Tester"}
    };
}

@Test(dataProvider = "userData")
public void testCreateUser(String name, String job) {
    given().body("{\"name\":\"" + name + "\", \"job\":\"" + job + "\"}")
            .when().post("/users")
            .then().statusCode(201);
}
```

How do you handle errors in API testing?

Answer: Handling errors involves:

- **Validating Error Responses**: Ensure the API returns appropriate error codes and messages (e.g., 400 Bad Request, 401 Unauthorized).
- **Boundary Testing**: Test the API with edge cases and invalid inputs.
- **Exception Handling**: Ensure the API gracefully handles exceptions and returns user-friendly error messages.

Example with Rest Assured:

```
Response response = given().when().get("/invalid-endpoint");
```

```
Assert.assertEquals(response.getStatusCode(), 404);
Assert.assertTrue(response.asString().contains("Not Found"));
```

Explain the concept of pagination in API testing.

Answer: Pagination is used to divide large sets of data into smaller chunks to improve performance and manageability. It involves:

- **Limit and Offset**: Specify the number of records to return and the starting point.
- **Page and Page Size**: Define the page number and the number of records per page.
- **Cursor-Based**: Use a pointer to keep track of the current position in the data set.

During testing, validate that pagination parameters work correctly and that the API returns the expected number of records and navigates through pages properly.

```
Response response = given().queryParam("page", 2).queryParam("pageSize",
10).when().get("/users");
Assert.assertEquals(response.getStatusCode(), 200);
Assert.assertTrue(response.jsonPath().getList("data").size() <= 10);
```

What is BDD and how does it relate to API testing?

Answer: Behavior-Driven Development (BDD) is a software development approach that encourages collaboration between developers, testers, and business stakeholders. It involves writing test cases in a natural language that is understandable by all stakeholders. In API testing, BDD helps in defining the behavior of APIs in a readable format using tools like Cucumber.

What are the benefits of using BDD for API testing?

Answer:

- **Improved Communication**: Ensures clear communication between technical and non-technical stakeholders.
- **Readability**: Tests are written in plain language which is easy to understand.
- **Living Documentation**: Test cases serve as documentation for the system's behavior.
- **Early Bug Detection**: Helps identify issues early in the development cycle.

What tools are commonly used for BDD API testing?

Answer:

- **Cucumber**: For writing BDD tests in Gherkin syntax.
- **Rest Assured**: For making HTTP requests and validating responses.
- **Serenity BDD**: For generating comprehensive test reports.
- **TestNG**: For running tests.
- **JBehave**: Another BDD framework for Java.

How do you write a feature file for a POST request in BDD?

Answer: A feature file for a POST request might look like this:

```
Feature: Create a new user

 Scenario: Successful creation of a new user
   Given the API endpoint is "https://reqres.in/api/users"
   When I send a POST request to the endpoint with the following details
     | name     | job       |
     | John Doe | Developer |
   Then the response status code should be 201
   And the response should contain "name"
   And the response should contain "job"
```

How do you implement step definitions for a POST request in BDD?

Answer: Step definitions for the above feature file:

```
import io.restassured.RestAssured;
import io.restassured.response.Response;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import org.testng.Assert;

import java.util.Map;

public class ApiSteps {

   private String endpoint;
   private Response response;

   @Given("the API endpoint is {string}")
   public void theApiEndpointIs(String url) {
       this.endpoint = url;
   }

   @When("I send a POST request to the endpoint with the following details")
   public void iSendAPostRequestToTheEndpointWithTheFollowingDetails(Map<String,
String> userDetails) {
```

```
        response = RestAssured
                .given()
                .contentType("application/json")
                .body(userDetails)
                .post(endpoint);
    }

    @Then("the response status code should be {int}")
    public void theResponseStatusCodeShouldBe(int statusCode) {
        Assert.assertEquals(response.getStatusCode(), statusCode);
    }

    @Then("the response should contain {string}")
    public void theResponseShouldContain(String content) {
        Assert.assertTrue(response.asString().contains(content));
    }
}
```

How do you configure a test runner for BDD tests?

Answer: A test runner using Cucumber and TestNG:

```
package runners;

import io.cucumber.testng.AbstractTestNGCucumberTests;
import io.cucumber.testng.CucumberOptions;

@CucumberOptions(
    features = "src/test/resources/features",
    glue = "steps"
)
public class TestRunner extends AbstractTestNGCucumberTests {
}
```

What is Gherkin syntax?

Answer: Gherkin is a domain-specific language for writing BDD test cases. It uses plain English language to define test cases in a structured manner using keywords like `Feature`, `Scenario`, `Given`, `When`, `Then`, and `And`.

How do you perform data-driven testing in BDD?

Answer: Data-driven testing can be done using the `Scenario Outline` keyword in Gherkin.

```
Feature: User creation
```

```
Scenario Outline: Create multiple users
  Given the API endpoint is "https://reqres.in/api/users"
  When I send a POST request to the endpoint with the following details
    | name     | job       |
    | <name>   | <job>     |
  Then the response status code should be 201
  And the response should contain "name"
  And the response should contain "job"

  Examples:
    | name     | job       |
    | John Doe | Developer |
    | Jane Doe | Tester    |
```

How do you validate JSON response in BDD API testing?

Answer: Rest Assured can validate JSON responses in your step definitions.

```
@Then("the response should contain {string}")
public void theResponseShouldContain(String content) {
   Assert.assertTrue(response.asString().contains(content));
}

@Then("the response should contain the field {string} with value {string}")
public void theResponseShouldContainTheFieldWithValue(String field, String value) {
   Assert.assertEquals(response.jsonPath().getString(field), value);
}
```

How do you handle authentication in BDD API tests?

Answer: Handle authentication by including necessary headers or tokens in your request.

```
@When("I send a POST request to the endpoint with the following details and token
{string}")
public void iSendAPostRequestToTheEndpointWithTheFollowingDetailsAndToken(String
token, Map<String, String> userDetails) {
   response = RestAssured
            .given()
            .header("Authorization", "Bearer " + token)
            .contentType("application/json")
            .body(userDetails)
            .post(endpoint);
}
```

How do you ensure the idempotency of API requests?

Answer: Ensure idempotency by making repeated requests and verifying the response is consistent.

```
@When("I send the same POST request multiple times")
public void iSendTheSamePostRequestMultipleTimes(Map<String, String> userDetails) {
    response = RestAssured
            .given()
            .contentType("application/json")
            .body(userDetails)
            .post(endpoint);

    Response responseRepeated = RestAssured
            .given()
            .contentType("application/json")
            .body(userDetails)
            .post(endpoint);

    Assert.assertEquals(response.getStatusCode(), responseRepeated.getStatusCode());
    Assert.assertEquals(response.asString(), responseRepeated.asString());
}
```

How do you handle rate limiting in API testing?

Answer: Handle rate limiting by sending requests in a loop and checking the response for rate limit errors.

```
@When("I send multiple requests to the endpoint")
public void iSendMultipleRequestsToTheEndpoint() {
    for (int i = 0; i < 10; i++) {
        response = RestAssured.given().get(endpoint);
        if (response.getStatusCode() == 429) { // 429 Too Many Requests
            break;
        }
    }
}

@Then("I should receive a rate limit error")
public void iShouldReceiveARateLimitError() {
    Assert.assertEquals(response.getStatusCode(), 429);
    Assert.assertTrue(response.asString().contains("rate limit"));
}
```

How do you perform schema validation in BDD API tests?

Answer: Use JSON schema validation to ensure the response matches the expected schema.

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

@Then("the response should match the schema {string}")
public void theResponseShouldMatchTheSchema(String schemaPath) {

response.then().assertThat().body(JsonSchemaValidator.matchesJsonSchemaInClasspath(sc
hemaPath));
}
```

How do you handle different environments in API testing?

Answer: Use environment variables or configuration files to handle different environments.

```
@Given("the API endpoint is set for {string} environment")
public void theApiEndpointIsSetForEnvironment(String environment) {
    if (environment.equalsIgnoreCase("production")) {
        this.endpoint = "https://api.production.com/users";
    } else if (environment.equalsIgnoreCase("staging")) {
        this.endpoint = "https://api.staging.com/users";
    }
}
```

How do you manage test data for API testing?

Answer: Manage test data using external files (CSV, JSON, Excel) or in-memory data providers.

```
@DataProvider(name = "userData")
public Object[][] createUserData() {
    return new Object[][]{
            {"John Doe", "Developer"},
            {"Jane Doe", "Tester"}
    };
}

@Test(dataProvider = "userData")
public void testCreateUser(String name, String job) {
    given().body("{\"name\":\"" + name + "\", \"job\":\"" + job + "\"}")
            .when().post("/users")
            .then().statusCode(201);
}
```

Chapter 7

Performance Testing of APIs using JMeter

Install Apache JMeter

Install Java

JMeter requires Java to be installed on your machine. Ensure you have the Java Development Kit (JDK) installed.

1. **Check if Java is already installed:** Open a terminal (Command Prompt on Windows) and type: `java -version`; you will see a version number if Java is installed.
2. **Download and install Java:**
    - If Java is not installed, download the JDK from the Oracle website or OpenJDK.
    - Follow the installation instructions specific to your operating system.

Download JMeter

1. Go to the Apache JMeter download page.
2. Download the binary (zip or tgz) file from the "Binaries" section.

Install JMeter

Extract the downloaded archive:

For Windows: Right-click the downloaded `.zip` file and select "Extract All..."

For Mac/Linux: Use the `tar` command in the terminal:

```
tar -xvzf apache-jmeter-<version>.tgz
```

Move the extracted folder to a preferred location:

You can place the extracted `apache-jmeter-<version>` folder anywhere on your system.

Set Up Environment Variables (Optional)

Setting up environment variables allows you to run JMeter from any command prompt or terminal directory.

Windows:

- Open the Control Panel, go to System and Security > System > Advanced system settings.
- Click on "Environment Variables".
- Under "System variables", find the `Path` variable, select it, and click "Edit".
- Add the path to the `bin` directory of the extracted JMeter folder (e.g., `C:\path\to\apache-jmeter-<version>\bin`).

Mac/Linux:

- Open your terminal and edit the `~/.bashrc` (or `~/.bash_profile` or `~/.zshrc` if using zsh) file:

  ```
  nano ~/.bashrc
  ```

- Add the following line:
  `export PATH=$PATH:/path/to/apache-jmeter-<version>/bin`
- Save the file and reload the configuration:
  `source ~/.bashrc`

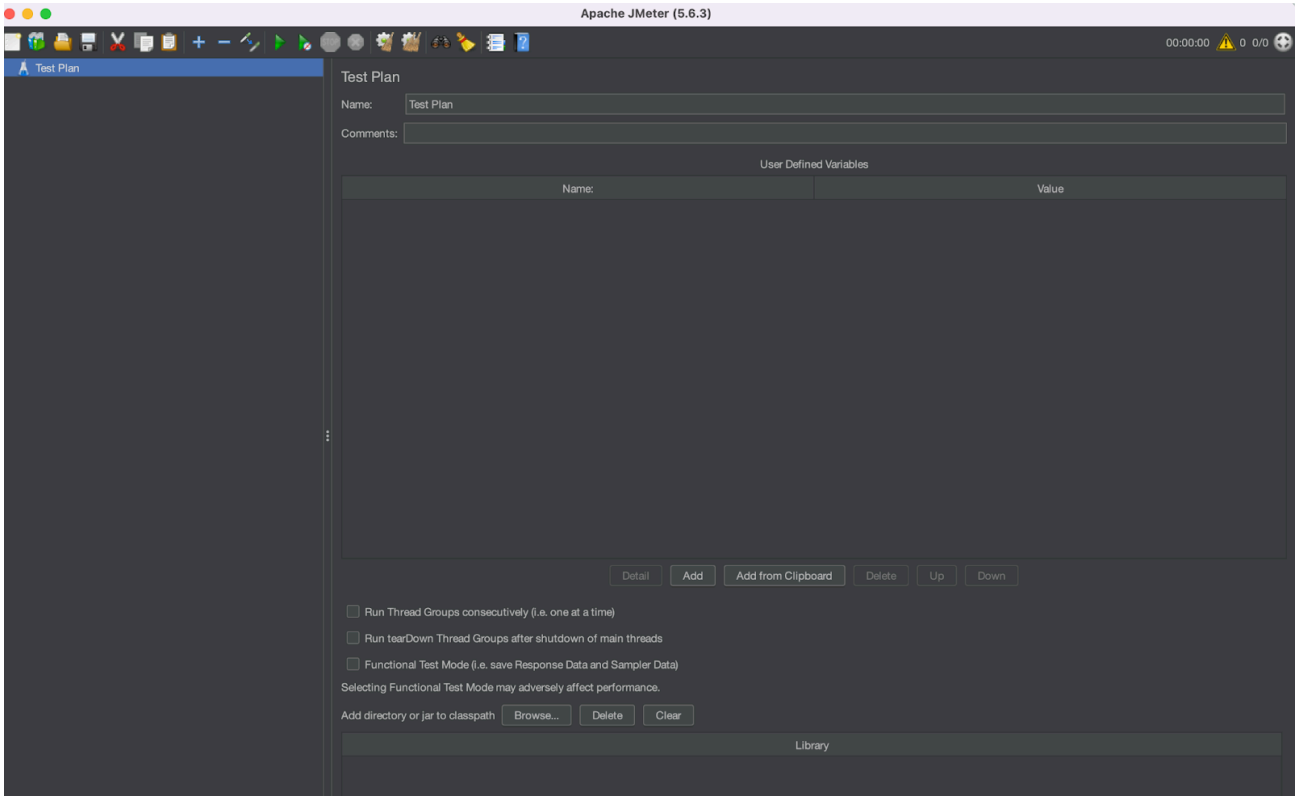
Launch JMeter

1. **Windows:**
   - Navigate to the `bin` directory inside the JMeter folder.
   - Double-click `jmeter.bat` to launch JMeter.
2. **Mac/Linux:**
   - Open a terminal.
   - Navigate to the `bin` directory inside the JMeter folder.
   - Run the following command: `./jmeter.sh`

```
lamhot@Lamhots-MacBook-Pro bin %
lamhot@Lamhots-MacBook-Pro bin % ./jmeter.sh
WARNING: package sun.awt.X11 not in java.desktop
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a future release
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a future release
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a future release
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a future release
================================================================================
Don't use GUI mode for load testing !, only for Test creation and Test debugging.
For load testing, use CLI Mode (was NON GUI):
   jmeter -n -t [jmx file] -l [results file] -e -o [Path to web report folder]
& increase Java Heap to meet your test requirements:
   Modify current env variable HEAP="-Xms1g -Xmx1g -XX:MaxMetaspaceSize=256m" in the jmeter batch file
Check : https://jmeter.apache.org/usermanual/best-practices.html
================================================================================
```

Verify Installation

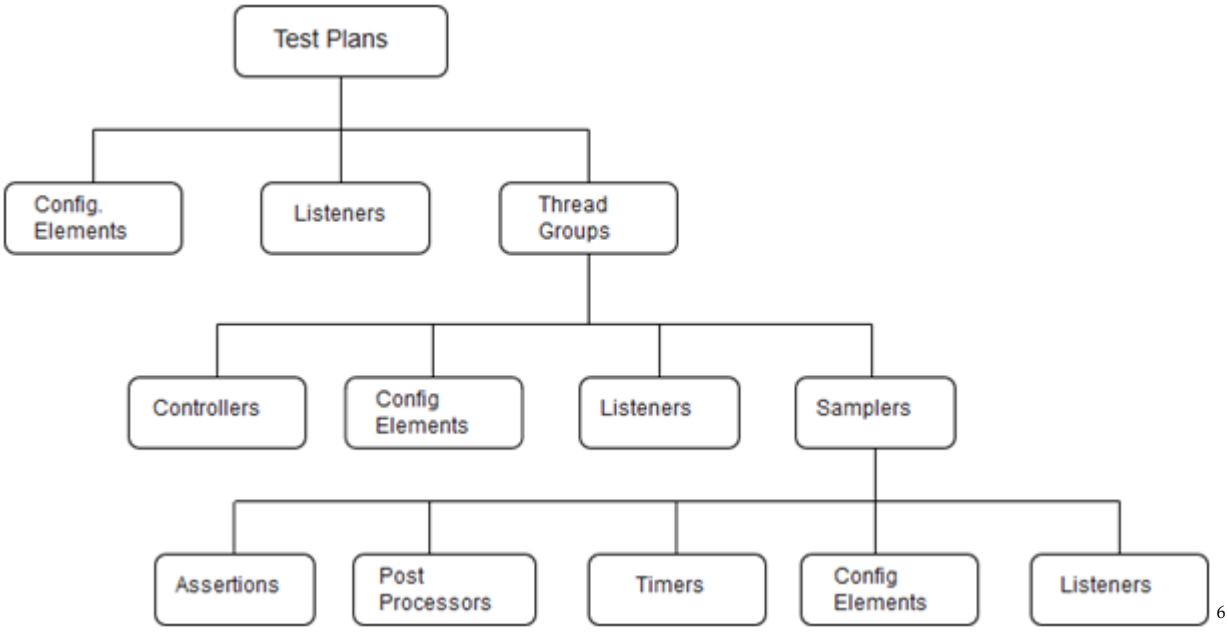
After launching JMeter, you should see the JMeter GUI, which indicates that the installation succeeded.



Tips:

- **Updating JMeter:** To update JMeter, download the latest version and replace the old folder with the new one. Ensure that the `Path` variable (if set) is updated accordingly.
- **Plugins:** Consider installing JMeter plugins for extended functionality. Visit the JMeter Plugins website for more information.

JMeter Components



[6]

Apache JMeter has several key components that you can use to design and execute your performance tests. Here's an overview of the main components:

1. Test Plan

- **Description:** A Test Plan is the container for your entire test script. It includes all the other components like Thread Groups, Samplers, Listeners, and more.
- **Usage:** Define the overall structure of your performance test. Add other components to build your test scenario.

2. Thread Group

- **Description:** A Thread Group represents a group of virtual users that will execute the test scripts. It defines the number of users, the ramp-up period, and the loop count.
- **Usage:** Configure how many users will run the test, how quickly they will start, and how many times they will repeat the test.

3. Samplers

- **Description:** Samplers are requests that JMeter sends to the server. They can be of various types, including HTTP requests, FTP requests, JDBC requests, and more.
- **Types of Samplers:**
    - **HTTP Request:** Used to send HTTP/HTTPS requests.
    - **FTP Request:** Used to send FTP commands.

[6] https://www.devstringx.com/use-jmeter-for-api-load-testing

    - **JDBC Request:** Used to execute SQL queries.
    - **SOAP/XML-RPC Request:** Used to send SOAP or XML-RPC requests.

4. Logic Controllers

- **Description:** Logic Controllers determine the order in which Samplers are processed.
- **Types of Logic Controllers:**
    - **Simple Controller:** Organizes the samplers and other controllers.
    - **Loop Controller:** Runs the samplers a specified number of times.
    - **If Controller:** Executes samplers only if a certain condition is true.
    - **Transaction Controller:** Measures the overall time taken to complete a set of nested samplers.
    - **While Controller:** Repeats the nested samplers while a condition is true.

5. Listeners

- **Description:** Listeners collect and display the results of the test execution. They provide different views of the performance test results.
- **Types of Listeners:**
    - **View Results Tree:** Shows the results of each sample in a tree format.
    - **Summary Report:** Provides a summary of performance metrics.
    - **Graph Results:** Displays the test results in a graphical format.
    - **Aggregate Report:** Summarizes statistics like average, min, max, and standard deviation for the test samples.

6. Timers

- **Description:** Timers introduce delays between requests. They help simulate real-world user interactions by adding think time.
- **Types of Timers:**
    - **Constant Timer:** Introduces a fixed delay.
    - **Gaussian Random Timer:** Introduces a random delay based on a Gaussian distribution.
    - **Uniform Random Timer:** Introduces a random delay within a specified range.
    - **Constant Throughput Timer:** Controls the throughput by introducing pauses to maintain a specified number of requests per minute.

7. Assertions

- **Description:** Assertions are used to validate the responses received from the server. They ensure that the responses contain the expected data.
- **Types of Assertions:**
    - **Response Assertion:** Checks for patterns in the response text.
    - **Duration Assertion:** Verifies that the response time is within a specified limit.
    - **Size Assertion:** Ensures that the response is of a certain size.

- **XML Assertion:** Validates the response against an XML schema.

8. Configuration Elements

- **Description:** Configuration Elements provide support data for Samplers and modify the requests they send.
- **Types of Configuration Elements:**
  - **CSV Data Set Config:** Reads data from a CSV file for parameterization.
  - **HTTP Request Defaults:** Sets default values for HTTP requests.
  - **User Defined Variables:** Defines variables that can be used throughout the test plan.
  - **JDBC Connection Configuration:** Configures database connections.

9. Pre-Processors and Post-Processors

- **Pre-Processors:**
  - **Description:** Pre-Processors are actions that are performed before the sampler request is sent.
  - **Example:** HTTP URL Re-writing Modifier, Regular Expression Extractor.
- **Post-Processors:**
  - **Description:** Post-Processors are actions that are performed after the sampler request has been sent.
  - **Example:** Regular Expression Extractor, JSON Path Post-Processor.

10. Test Fragments

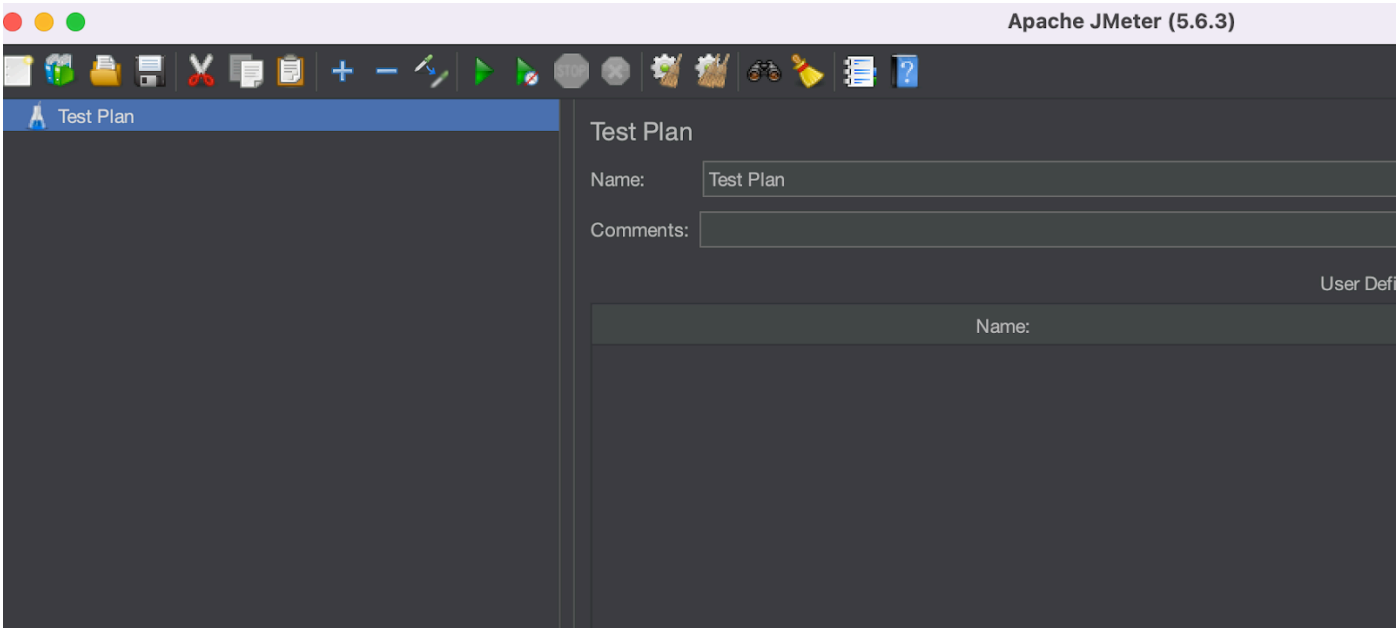
- **Description:** Test Fragments are used to define parts of the test that can be reused in other parts of the test plan.
- **Usage:** Useful for modularizing the test plan and reusing components.

These components allow you to create comprehensive and flexible performance tests in JMeter. By combining these elements, you can simulate complex scenarios and gather detailed performance metrics for your application.

Guide to Creating Performance Test Script

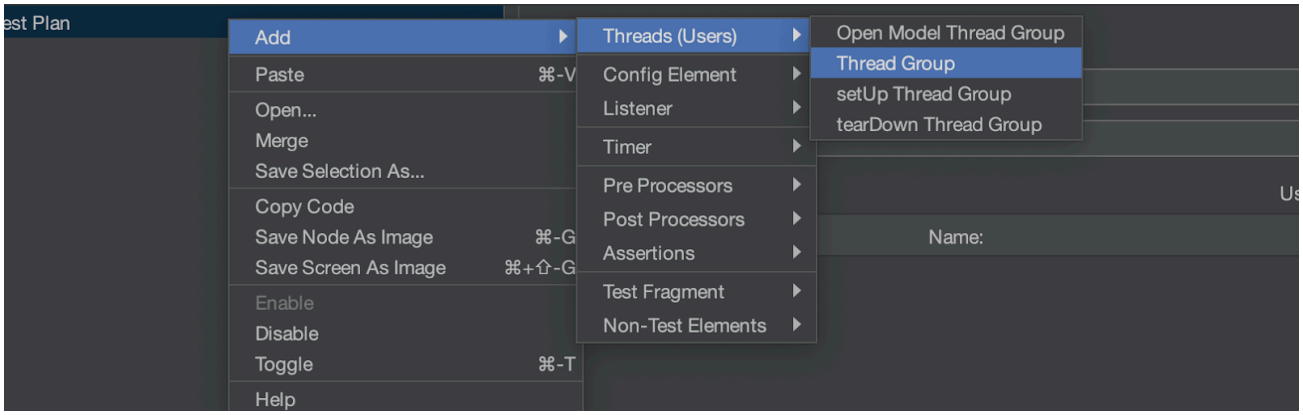
Create a Test Plan

1. In JMeter, create a new test plan by selecting `File > New`.



Add a Thread Group

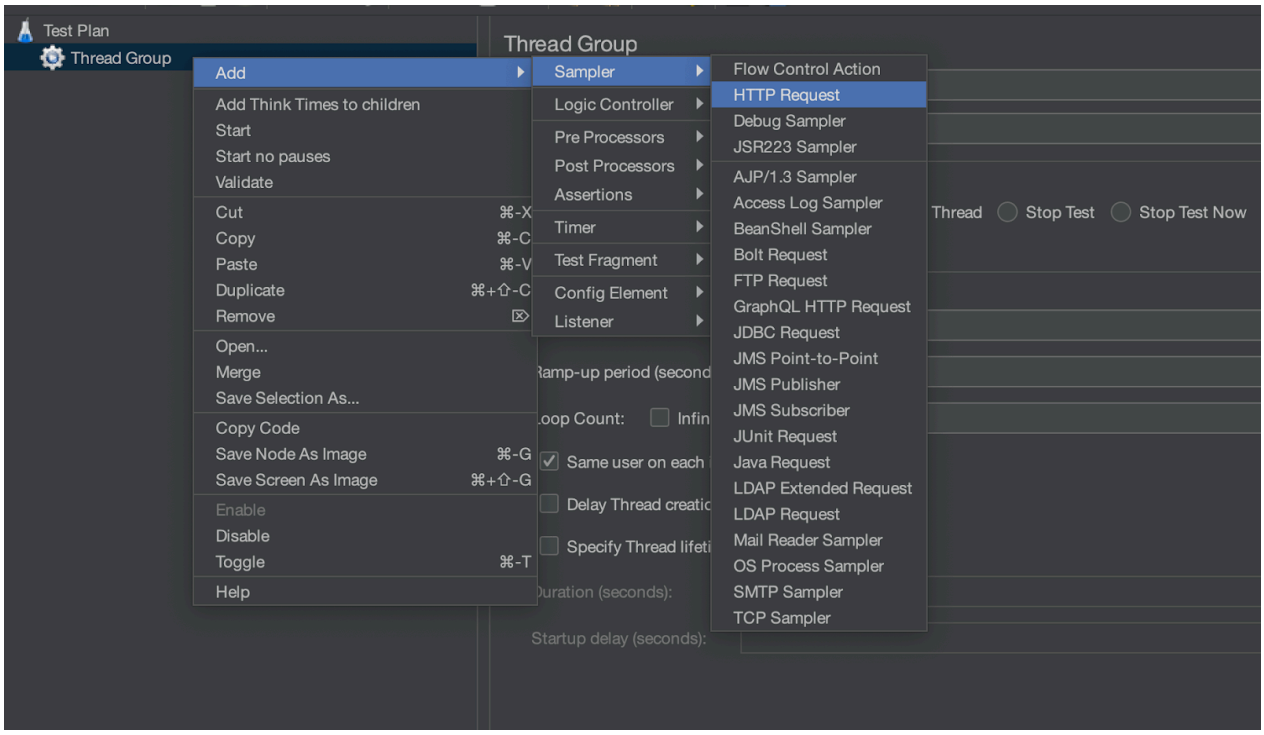
1. Right-click on the Test Plan and select `Add > Threads (Users) > Thread Group`.



2. Configure the Thread Group:
   - **Number of Threads (Users):** This specifies the number of virtual users.
   - **Ramp-Up Period (in seconds):** This is the time JMeter should take to get all the threads up and running.
   - **Loop Count:** This specifies how many times to execute the test.

Add an HTTP Request

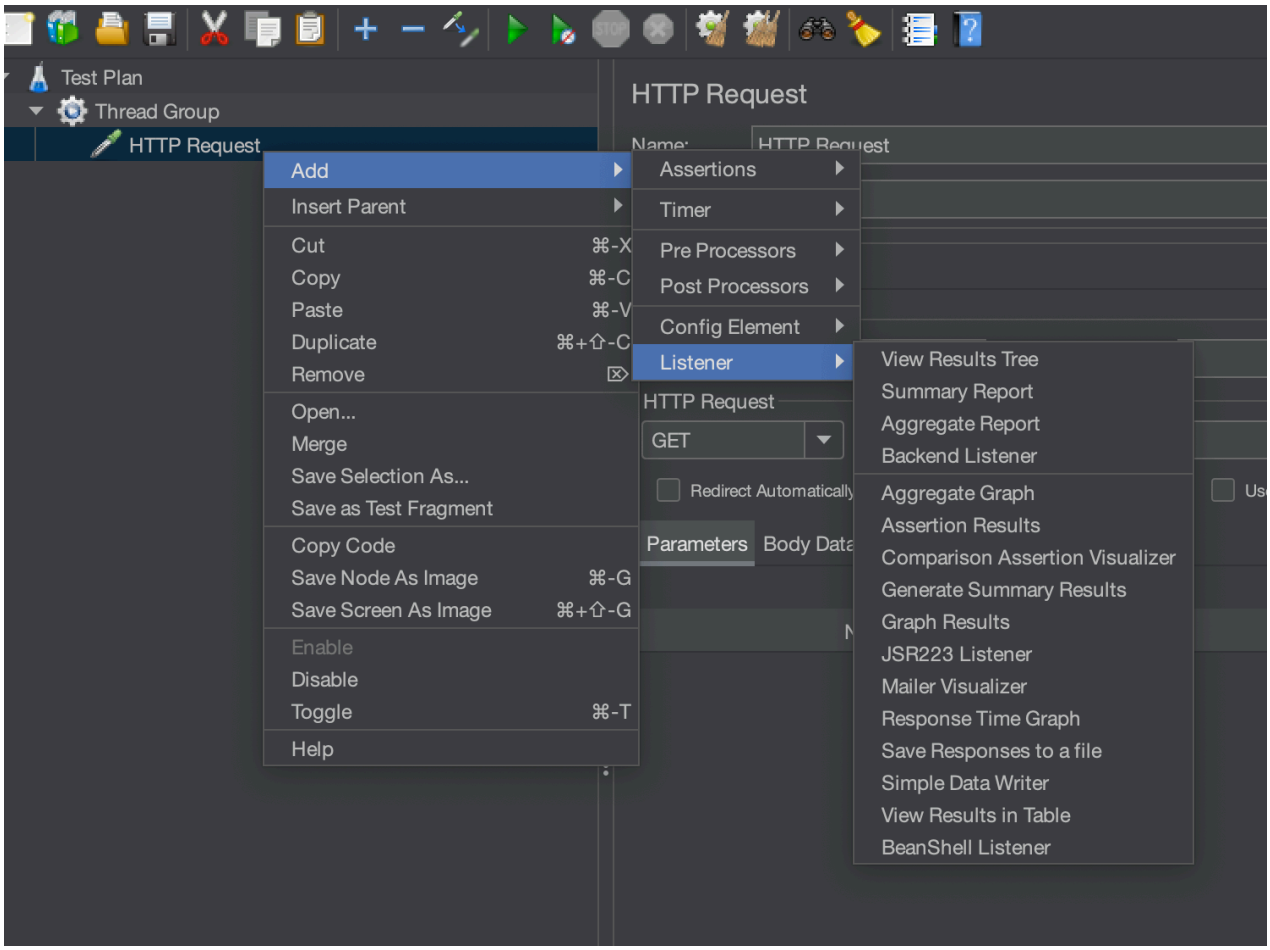
1. Right-click on the Thread Group and select `Add > Sampler > HTTP Request`.



2. Configure the HTTP Request:
    - **Server Name or IP:** Enter the hostname or IP address of the API server.
    - **Port Number:** Enter the port number if required (default is 80 for HTTP and 443 for HTTPS).
    - **HTTP Request:** Specify the method (GET, POST, etc.) and the path (e.g., `/api/v1/resource`).
    - **Parameters:** Add any necessary parameters if the API requires them.

Add a Listener

1. Right-click on the Thread Group and select `Add > Listener > View Results Tree` or any other listener of your choice (e.g., `Summary Report`, `Graph Results`).



Run the Test

1. Save the test plan by selecting `File > Save`.
2. Click the green "Start" button (a triangle) in the JMeter toolbar to run the test.
3. View the results in the Listener you added.

Example Configuration

Here's an example configuration for a simple GET request to `http://example.com/api/v1/resource`:

- **Thread Group:**
  - Number of Threads (Users): 10
  - Ramp-Up Period (in seconds): 5
  - Loop Count: 100
- **HTTP Request:**
  - Server Name or IP: `example.com`
  - Path: `/api/v1/resource`
  - Method: `GET`

Common Interview Questions and Answers Related to Performance Testing of APIs

What is API performance testing?

Answer: API performance testing involves evaluating the speed, responsiveness, and stability of an API under various conditions. It helps ensure the API can handle the expected load and perform efficiently.

Why is performance testing important for APIs?

Answer: Performance testing ensures that the API can handle high traffic, provides a good user experience, identifies potential bottlenecks, and meets the performance criteria before going live.

What tools can be used for API performance testing?

Answer: Common tools include Apache JMeter, Postman, SoapUI, LoadRunner, and Gatling.

What is the difference between load testing and stress testing?

Answer: Load testing measures the system's performance under expected user loads, while stress testing evaluates how the system behaves under extreme or beyond expected loads.

How do you set up a basic API performance test in JMeter?

Answer: Create a Test Plan, add a Thread Group, configure the number of threads and ramp-up period, add an HTTP Request sampler, specify the API details, and add a Listener to capture the results.

What are some common metrics to measure in API performance testing?

Answer: Key metrics include response time, throughput, error rate, latency, and resource utilization (CPU, memory).

What is a throughput in performance testing?

Answer: Throughput refers to the number of requests processed by the API per unit of time, usually measured in transactions per second (TPS).

What is a latency in performance testing?

Answer: Latency is the time taken for a request to travel from the client to the server and back. It is the delay between the request and the first byte of the response.

How do you handle dynamic parameters in JMeter?

Answer: Use the Regular Expression Extractor to capture dynamic values from the response and store them in variables for use in subsequent requests.

What is the purpose of a Listener in JMeter?

Answer: Listeners capture and display the results of the test execution. They provide various reports and visualizations to analyze the performance data.

How do you simulate a heavy load in JMeter?

Answer: Increase the number of threads (users) in the Thread Group, adjust the ramp-up period, and configure the loop count to simulate a high load on the API.

What is the significance of the ramp-up period in JMeter?

Answer: The ramp-up period defines the time taken to start all threads. It helps in gradually increasing the load on the server to avoid sudden spikes.

How do you identify performance bottlenecks in an API?

Answer: Analyze metrics such as response time, CPU and memory usage, throughput, and error rates. Use profiling tools to pinpoint slow-performing code or database queries.

What is a distributed testing in JMeter?

Answer: Distributed testing involves using multiple machines to generate a higher load than a single machine can handle. It allows for testing larger-scale scenarios.

What are the different types of performance testing?

Answer: The main types are load testing, stress testing, endurance testing, spike testing, and volume testing.

How do you ensure the accuracy of performance test results?

Answer: Ensure a controlled environment, use realistic test data, run multiple iterations, monitor system resources, and analyze results to identify any anomalies.

What is a correlation in performance testing?

Answer: Correlation involves capturing and reusing dynamic values from the server response in subsequent requests. It ensures that the test script accurately simulates real-world scenarios.

What is the purpose of the 'Think Time' in JMeter?

Answer: Think Time simulates the real user's wait time between actions. It adds pauses in the test script to mimic user interactions more accurately.

How do you perform a spike test in JMeter?

Answer: Increase the number of threads abruptly in a short period to simulate a sudden spike in traffic and observe how the API handles the load.

What are the best practices for API performance testing?

Answer: Best practices include:

- Define clear performance criteria and goals.
- Use realistic test data and scenarios.
- Run tests in a production-like environment.
- Monitor and analyze system resources.
- Perform tests iteratively and continuously.

Chapter 8

Organizing a test framework for API Testing

To create a comprehensive and well-organized test framework for API testing with Cucumber, we need to consider several aspects: request handling, response verification, exception management, configurations, user authentication, processing, modeling, test framework setup, test assertions, logging, utility functions, test execution, debugging configurations, and a test driver.

Here's a detailed structure and guide for setting up such a framework:

Project Structure

Organize your project structure to keep it modular and maintainable:

```
src
├── main
│   └── java
│       └── com/yourcompany/yourproject
│           ├── config
│           │   └── Config.java
│           ├── exceptions
│           │   └── ApiException.java
│           ├── model
│           │   └── ApiResponse.java
│           ├── processor
│           │   └── ResponseProcessor.java
│           ├── util
│           │   ├── ApiClient.java
│           │   ├── AuthenticationUtil.java
│           │   ├── LoggerUtil.java
│           │   └── TestUtil.java
│           └── driver
│               └── TestDriver.java
└── test
    └── java
        └── com/yourcompany/yourproject
            ├── features
            │   └── api.feature
            ├── runners
            │   └── TestRunner.java
            └── stepdefinitions
                └── ApiStepDefinitions.java
```

Dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>bdd-api-automation</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- TestNG Dependency -->
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>7.4.0</version>
            <scope>test</scope>
        </dependency>
        <!-- Rest Assured Dependency -->
        <dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>rest-assured</artifactId>
            <version>4.3.3</version>
        </dependency>
        <!-- Cucumber Dependencies -->
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-java</artifactId>
            <version>6.10.4</version>
        </dependency>
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-testng</artifactId>
            <version>6.10.4</version>
        </dependency>
        <!-- JSON Schema Validator Dependency -->
        <dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>json-schema-validator</artifactId>
            <version>4.3.3</version>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>7.4.0</version>
            <scope>compile</scope>
        </dependency>
        <!-- JUnit for running tests -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
        <!-- Logging -->
        <dependency>
            <groupId>org.slf4j</groupId>
```

```
            <artifactId>slf4j-api</artifactId>
            <version>1.7.30</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.30</version>
        </dependency>
    </dependencies>
    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>
</project>
```

Configuration

Create a configuration class to manage configurations in `src/main/java/com/yourcompany/yourproject/config/Config.java`.

```
package config;

public class Config {
    public static final String BASE_URL = "http://api.yourservice.com";
    public static final String AUTH_TOKEN = "your-auth-token";
}
```

Exception Handling

Create a custom exception class for API exceptions in `src/main/java/com/yourcompany/yourproject/exceptions/ApiException.java`.

```
package exceptions;

public class ApiException extends RuntimeException {
    public ApiException(String message) {
        super(message);
    }

    public ApiException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Models

Model refers to the representation of the data structures used by the API. These models are usually Java classes that mirror the JSON or XML payloads exchanged between the client and the server. By creating

these model classes, you can easily serialize and deserialize API request and response bodies, making the test code cleaner and more maintainable.

Create a response model class in `src/main/java/com/yourcompany/yourproject/model/ApiResponse.java`.

```
package model;

public class ApiResponse {
    private int statusCode;
    private String body;

    // Getters and setters
    public int getStatusCode() {
        return statusCode;
    }

    public void setStatusCode(int statusCode) {
        this.statusCode = statusCode;
    }

    public String getBody() {
        return body;
    }

    public void setBody(String body) {
        this.body = body;
    }
}
```

Utility Classes

Utility classes in an automation framework provide common functionalities that can be reused across different parts of the test scripts. They help to avoid code duplication and make the test scripts cleaner and more maintainable.

API client Util

Create utility classes for API client, authentication, logging, and test utility in `src/main/java/com/yourcompany/yourproject/util`.

```
package util;

import io.restassured.response.Response;

import static io.restassured.RestAssured.*;

public class ApiClient {

    public static Response get(String endpoint) {
        return given().when().get(endpoint);
    }

    public static Response post(String endpoint, Object body) {
        return given().body(body).when().post(endpoint);
    }

    // Add other methods (put, delete) as needed
}
```

Authentication Util

User authentication is typically integrated within the request component. However, it should be made explicit and visible to facilitate future modifications without affecting other components. This clarity also simplifies adding support for testing different authentication types.

```
package util;

import config.Config;

public class AuthenticationUtil {

    public static String getAuthToken() {
        // Return auth token from config or fetch dynamically
        return Config.AUTH_TOKEN;
    }
}
```

Logger Util

Logger support is beneficial for writing test scripts, though it's not strictly mandatory. Log4j is widely used in Java projects for logging purposes and is especially helpful during debugging.

```
package util;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LoggerUtil {

   private static final Logger logger = LoggerFactory.getLogger(LoggerUtil.class);

   public static void logInfo(String message) {
       logger.info(message);
   }

   public static void logError(String message, Throwable throwable) {
       logger.error(message, throwable);
   }
}
```

Test Util

In test scripts, there are often requirements to generate time values in various units (milliseconds, seconds, minutes, hours, etc.), sort responses, or handle other routine tasks essential for application testing. These functionalities can be efficiently managed within the test framework's utility component.

```
package util;

import org.testng.Assert;

public class TestUtil {

   public static void assertStatusCode(int actual, int expected) {
       Assert.assertEquals(expected, actual);
   }

   public static void assertResponseContains(String response, String expectedValue) {
       Assert.assertTrue(response.contains(expectedValue));
   }
}
```

Response Processor

You may need to execute tests individually, as part of a suite, or by running different test methods within a test class. Ideally, the test framework should provide support for all these scenarios. If it doesn't, you'll need to implement the necessary functionality to handle them.

Create a response processor class in `src/main/java/com/yourcompany/yourproject/processor/ResponseProcessor.java`.

```
package processor;

import io.restassured.response.Response;
import model.ApiResponse;

public class ResponseProcessor {

   public static ApiResponse processResponse(Response response) {
       ApiResponse apiResponse = new ApiResponse();
       apiResponse.setStatusCode(response.getStatusCode());
       apiResponse.setBody(response.getBody().asString());
       return apiResponse;
   }
}
```

Feature File

Create a feature file in `src/test/java/com/yourcompany/yourproject/features/api.feature`.

```
Feature: API Testing

 Scenario Outline: Test API with multiple data sets
   Given I set up the API endpoint "<endpoint>"
   When I send a GET request
   Then the response status code should be <statusCode>
   And the response should contain "<expectedValue>"

   Examples:
     | endpoint      | statusCode | expectedValue |
     | /api/test/1   | 200        | value1        |
     | /api/test/2   | 404        | error         |
     | /api/test/3   | 200        | value3        |
```

Step Definitions

Implement step definitions in `src/test/java/com/yourcompany/yourproject/stepdefinitions/ApiStepDefinitions.java`.

```
package steps;

import io.cucumber.java.en.*;
import model.ApiResponse;
import processor.ResponseProcessor;
import util.ApiClient;
import util.TestUtil;

public class ApiStepDefinitions {

   private String endpoint;
   private ApiResponse apiResponse;

   @Given("I set up the API endpoint {string}")
   public void i_set_up_the_api_endpoint(String endpoint) {
       this.endpoint = endpoint;
   }

   @When("I send a GET request")
   public void i_send_a_get_request() {
       apiResponse = ResponseProcessor.processResponse(ApiClient.get(endpoint));
   }

   @Then("the response status code should be {int}")
   public void the_response_status_code_should_be(int statusCode) {
       TestUtil.assertStatusCode(apiResponse.getStatusCode(), statusCode);
   }

   @Then("the response should contain {string}")
   public void the_response_should_contain(String expectedValue) {
       TestUtil.assertResponseContains(apiResponse.getBody(), expectedValue);
   }
}
```

Test Runner

Create a test runner class in `src/test/java/com/yourcompany/yourproject/runners/TestRunner.java`.

```
package runners;


import org.junit.runner.RunWith;


import io.cucumber.testng.AbstractTestNGCucumberTests;
import io.cucumber.testng.CucumberOptions;
```

```
@CucumberOptions(
        features = "src/test/resources/features",
        glue = "steps"
)
public class TestRunner extends AbstractTestNGCucumberTests {
}
```

Test Driver

Create a test driver class if needed for custom test execution in `src/main/java/com/yourcompany/yourproject/driver/TestDriver.java`.

```
package driver;

public class TestDriver {

   public static void main(String[] args) {
       // Custom test execution logic if needed
   }
}
```

Common Interview Questions and Answers Related to Organizing a Test Framework

General Framework Design

1. **Question:** How would you design an API test framework using Cucumber? **Answer:** I would design the framework by organizing the project structure into separate packages for features, step definitions, and utilities. The features package would contain the Cucumber feature files written in Gherkin syntax. The step definitions package would map Gherkin steps to Java methods, leveraging libraries like RestAssured for API interactions. The utilities package would include common helper classes, such as configuration managers and custom assertions.
2. **Question:** What are the benefits of using Cucumber for API testing? **Answer:** Cucumber allows for Behavior-Driven Development (BDD), making test scenarios readable and understandable by non-technical stakeholders. It facilitates collaboration between developers, testers, and business analysts. The use of Gherkin syntax ensures that the test scenarios are written in plain language, promoting clear communication of requirements and test cases.

Request and Response Handling

3. **Question:** How do you structure your request and response classes in a Cucumber-based API test framework? **Answer:** I create model classes that represent the JSON payloads for requests and responses. These model classes are used for serialization and deserialization using libraries like Jackson or Gson. The step definitions interact with these model classes to send requests and validate responses, ensuring a clean separation of concerns.
4. **Question:** How would you handle different HTTP methods (GET, POST, PUT, DELETE) in your framework? **Answer:** I would create a utility class, such as `ApiClientUtil`, which provides methods for different HTTP methods. Each method would use RestAssured to send the respective HTTP request, passing necessary headers, query parameters, and body data. This utility class would be reusable across different step definitions.

Exception Handling

5. **Question:** How do you handle exceptions in your API test framework? **Answer:** I handle exceptions by implementing custom exception classes and using try-catch blocks within the step definitions and utility classes. Additionally, I use logging to capture detailed information about exceptions for debugging purposes. Custom exception handling ensures that meaningful error messages are provided, improving test maintainability and troubleshooting.
6. **Question:** Can you provide an example of how to implement custom exception handling in a Cucumber step definition? **Answer:** Sure, here's an example:

```
@When("^I send a GET request to the endpoint \"([^\"]*)\"$")
public void sendGetRequest(String endpoint) {
    try {
```

```
        Response response = ApiClientUtil.get(endpoint);
        scenarioContext.setResponse(response);
    } catch (Exception e) {
        LoggerUtil.logError("Failed to send GET request", e);
        throw new CustomException("Error occurred while sending GET request to "
+ endpoint, e);
    }
}
```

Configurations

7. **Question:** How do you manage different configurations (e.g., environment URLs, credentials) in your test framework? **Answer:** I use a configuration utility class, such as `ConfigUtil`, that loads properties from a configuration file (e.g., `config.properties`). This class provides methods to retrieve configuration values, making it easy to manage environment-specific settings and credentials.
8. **Question:** How would you load configuration properties dynamically in your tests? **Answer:** I load configuration properties dynamically by reading the properties file at the start of the test execution. This can be done in a setup method annotated with `@Before` in Cucumber or using a static initializer block in the configuration utility class. This ensures that the properties are available throughout the test execution.

User Authentication

9. **Question:** How do you handle user authentication in your API tests? **Answer:** I handle user authentication by creating a utility class, such as `AuthUtil`, that manages the generation and management of authentication tokens. This class provides methods to obtain tokens for different authentication types (e.g., Basic Auth, OAuth) and includes these tokens in the request headers as needed.
10. **Question:** How would you manage different types of authentication (e.g., Basic Auth, OAuth) in your test framework? **Answer:** I create separate methods in the `AuthUtil` class for each authentication type. For Basic Auth, I encode the credentials and add them to the request headers. For OAuth, I implement token retrieval and refresh logic. The step definitions call these methods to include the appropriate authentication tokens in the API requests.

Processor and Models

11. **Question:** What role do model classes play in an API test framework? **Answer:** Model classes represent the structure of the data being sent and received in API requests and responses. They enable easy serialization and deserialization of JSON or XML payloads, ensuring that the data structures are correctly handled and validated within the tests.
12. **Question:** How would you design a processor class to handle complex API responses? **Answer:** A processor class would encapsulate logic for processing and validating complex API responses. This

class would take the raw response as input, deserialize it into the corresponding model classes, and perform necessary validation or transformation. This design promotes reusability and separation of concerns.

Test Assertions

13. **Question:** What assertion libraries do you use in your test framework, and why? **Answer:** I use assertion libraries like JUnit or TestNG for standard assertions and Hamcrest or AssertJ for more expressive and readable assertions. These libraries provide a rich set of assertion methods that help in writing clear and concise test validations.
14. **Question:** How do you implement custom assertions in your API tests? **Answer:** I implement custom assertions by creating utility methods in a class, such as `TestUtil`. These methods encapsulate complex validation logic, allowing reuse across multiple tests. Custom assertions improve test readability and maintainability by abstracting repetitive validation logic.

Logger

15. **Question:** Why is logging important in an API test framework? **Answer:** Logging is important because it provides detailed information about the test execution flow, including request and response data, errors, and other significant events. This information is crucial for debugging and troubleshooting issues that arise during testing.
16. **Question:** How do you integrate a logging framework (e.g., Log4j, SLF4J) into your Cucumber-based test framework? **Answer:** I integrate a logging framework by including the necessary dependencies in the project (e.g., Log4j or SLF4J). I then create a utility class, such as `LoggerUtil`, that wraps the logging framework's methods. This utility class is used throughout the step definitions and utility classes to log messages.

Utilities

17. **Question:** What are some common utility classes you have created for your API test framework? **Answer:** Common utility classes include `ConfigUtil` for configuration management, `ApiClientUtil` for sending API requests, `AuthUtil` for handling authentication, `LoggerUtil` for logging, and `TestUtil` for custom assertions and test-related utilities.
18. **Question:** How do you manage reusable utility methods in your test framework? **Answer:** I manage reusable utility methods by organizing them into dedicated utility classes. Each utility class focuses on a specific area of functionality, such as configuration management or API requests. This modular approach promotes code reuse and maintainability.

Test Execution

19. **Question:** How do you organize and execute your Cucumber tests? **Answer:** I organize my Cucumber tests by placing feature files in the `src/test/resources` directory and step definitions in the `src/test/java` directory. I use a Cucumber runner class annotated with

`@RunWith(Cucumber.class)` to execute the tests. The runner class specifies the location of feature files and step definitions.

20. **Question:** How do you run your tests in parallel, and what are the challenges associated with parallel test execution? **Answer:** I run tests in parallel using tools like Cucumber-JVM Parallel Plugin or JUnit's parallel execution features. The challenges include managing shared resources, ensuring thread safety, and handling test data dependencies. To address these challenges, I use techniques like thread-local variables and isolate test data for each parallel test instance.

Chapter 9

Continuous Integration and Continuous Deployment (CI/CD) with API Testing

Continuous Integration (CI) and Continuous Deployment (CD) are key practices in modern software development that help teams deliver code changes more frequently and reliably.

Introduction to CI/CD

Continuous Integration (CI): Continuous Integration (CI) is a software development practice where developers frequently commit code changes to a shared repository. Each commit triggers an automated build and testing process, ensuring that the new code integrates smoothly with the existing codebase. The goal of CI is to identify and fix integration issues early in the development cycle, improving code quality and reducing the time to deliver new features.

Continuous Deployment (CD): Continuous Deployment (CD) extends CI by automating the release of code changes to production environments. Once the code passes all stages of testing in the CI pipeline, it is automatically deployed to production. Continuous Deployment aims to minimize manual intervention and accelerate the delivery of new features to users, maintaining high quality and reliability.

Importance of CI/CD for API Testing

API testing is crucial in ensuring that the communication between different software systems is reliable, efficient, and secure. Integrating API testing into a CI/CD pipeline brings several benefits:

1. **Early Detection of Issues**: Automated API tests run with every code commit, identifying problems early in the development cycle. This reduces the risk of deploying faulty APIs to production.
2. **Consistency and Reliability**: CI/CD ensures that API tests are executed consistently, leading to more reliable and stable APIs. Automated tests can cover a wide range of scenarios and edge cases, which might be missed during manual testing.
3. **Faster Feedback Loop**: Developers receive immediate feedback on the impact of their changes on the API, allowing them to address issues promptly. This speeds up the development process and reduces the time spent on debugging and fixing errors.
4. **Improved Collaboration**: CI/CD fosters better collaboration among development, QA, and operations teams. Automated tests provide a common ground for verifying code quality, reducing the chances of miscommunication and misunderstandings.
5. **Scalability**: As the number of APIs and their complexity grow, manual testing becomes impractical. CI/CD pipelines can scale to handle large volumes of API tests, ensuring comprehensive coverage without increasing manual effort.

6. **Regression Testing**: Automated API tests ensure that new code changes do not break existing functionality. Regression tests run automatically in the CI/CD pipeline, providing confidence that the application remains stable over time.
7. **Continuous Improvement**: CI/CD pipelines facilitate continuous improvement by integrating new tests and updating existing ones as the API evolves. This iterative approach helps maintain high-quality APIs and adapts to changing requirements.
8. **Reduced Deployment Risks**: By automating the deployment process, CD reduces the risk of human errors during releases. APIs are deployed to production only after passing all automated tests, ensuring a higher level of reliability.
9. **Enhanced Security**: Automated API tests can include security tests to identify vulnerabilities and ensure compliance with security standards. CI/CD pipelines can integrate tools for static and dynamic analysis, improving the overall security posture of the API.

CD Tools

CD (Continuous Delivery) tools are software solutions designed to automate and streamline the process of delivering software updates and features to production environments. They help teams build, test, and deploy code more efficiently and reliably. Here are some popular CD tools:

Jenkins

Pros:

1. **Open Source and Free:** Jenkins is open-source and free to use, making it accessible for all types of organizations.
2. **Extensive Plugin Ecosystem:** Jenkins has a vast plugin ecosystem, allowing integration with many other tools and services.
3. **Highly Customizable:** Due to its open-source nature, Jenkins can be customized to meet almost any CI/CD requirement.
4. **Large Community Support:** Jenkins has a large, active community that contributes to its plugins and provides support.

Cons:

1. **Complex Setup and Maintenance:** Jenkins requires manual setup and ongoing maintenance, which can be time-consuming.
2. **Steep Learning Curve:** The configuration and management of Jenkins can be complex, especially for beginners.
3. **Performance Issues:** Jenkins can experience performance issues when scaling, especially with large pipelines.
4. **User Interface:** The UI is considered outdated compared to newer CI/CD tools.

GitLab CI/CD

Pros:

1. **Integrated with GitLab:** GitLab CI/CD is integrated with GitLab, providing a seamless experience from code repository to CI/CD.
2. **Ease of Use:** GitLab CI/CD is easier to set up and use compared to Jenkins, with a more modern UI.
3. **Built-in Features:** GitLab offers a wide range of built-in features such as version control, issue tracking, and CI/CD.
4. **Auto DevOps:** GitLab provides Auto DevOps, which automates CI/CD pipeline creation for common use cases.

Cons:

1. **Resource Intensive:** Running GitLab CI/CD, especially on self-hosted instances, can be resource-intensive.
2. **Complexity for Advanced Use Cases:** While GitLab CI/CD is easy to start with, complex pipelines and integrations can become challenging.
3. **Cost:** While GitLab offers free tiers, advanced features, and larger pipelines may require a paid subscription.
4. **Runner Management:** Managing GitLab Runners for CI/CD can be cumbersome, especially at scale.

AWS CodePipeline

Pros:

1. **Integration with AWS Services:** AWS CodePipeline integrates seamlessly with other AWS services, making it ideal for AWS-centric environments.
2. **Scalability:** As a managed service, AWS CodePipeline scales automatically based on the workload.
3. **Pay-as-You-Go:** Pricing is based on usage, which can be cost-effective for smaller projects.
4. **Security and Compliance:** AWS provides strong security features and compliance with various standards.

Cons:

1. **AWS Lock-in:** Using AWS CodePipeline can lead to vendor lock-in, as it's deeply integrated with AWS services.
2. **Learning Curve:** Familiarity with AWS services and concepts is required to effectively use CodePipeline.
3. **Less Flexibility:** Compared to Jenkins, AWS CodePipeline has fewer customization options and a smaller ecosystem of plugins.
4. **Cost for Larger Pipelines:** Costs can accumulate quickly for larger projects or more complex pipelines, especially if additional AWS services are required.

Travis CI

Travis CI is a continuous integration service used to build and test software projects hosted on GitHub and Bitbucket. Here are some pros and cons of using Travis CI:

Pros

1. Ease of Use: Travis CI is straightforward to set up and use, especially for open-source projects. It integrates seamlessly with GitHub and Bitbucket repositories.
2. Configuration as Code: Configuring builds is done via a `.travis.yml` file, which is stored in the repository. This makes it easy to version and review build configurations.
3. Language Support: Travis CI supports a wide range of programming languages and frameworks, including JavaScript, Ruby, Python, PHP, Java, Go, and more.
4. Free for Open Source: Travis CI offers free unlimited builds for open-source projects, which is a significant advantage for the open-source community.
5. Extensive Documentation: Travis CI provides comprehensive documentation and a robust community, making it easier to find solutions to common issues.
6. Third-Party Integrations: It integrates well with other tools and services, such as Slack, HipChat, and email for notifications, and can deploy to various cloud services.
7. Parallel Testing: Travis CI allows running multiple builds in parallel, which can speed up the CI process significantly.

Cons

1. Limited Free Plan for Private Repositories: For private repositories, the free plan has limitations, and higher usage requires a paid subscription.
2. Build Speed: Build times can sometimes be slower compared to other CI services, especially during peak times for free users.
3. Complex Configuration: While `.travis.yml` is powerful, it can become complex and hard to manage for larger projects with many dependencies and build steps.
4. Resource Limitations: There are limitations on build times and resources (CPU, memory) for free plans, which can be restrictive for resource-intensive projects.
5. Lack of Advanced Features: Compared to some newer CI/CD tools, Travis CI may lack some advanced features and flexibility, such as more sophisticated deployment workflows or more granular control over build environments.
6. Vendor Lock-In: As with any CI service, there can be some level of vendor lock-in. Migrating to a different CI/CD platform can be time-consuming and require significant changes to the build configuration.

CircleCI

Pros:

1. **Ease of Use**: CircleCI is user-friendly with an intuitive interface, making it easy for new users to get started quickly.
2. **Scalability**: Offers excellent scalability options, allowing teams to run a large number of parallel jobs.
3. **Integration**: Integrates well with many third-party tools and services like GitHub, Bitbucket, Docker, and more.

4. **Customization**: Highly configurable pipelines with support for custom scripts and workflows.
5. **Cloud and On-Premises**: Provides both cloud-based and self-hosted solutions, giving flexibility based on the team's needs.

Cons:

1. **Pricing**: Can get expensive, especially for larger teams or projects with high concurrency needs.
2. **Configuration Complexity**: YAML-based configuration files can become complex and difficult to manage for very large projects.
3. **Support**: Users sometimes report slow support response times.

Bamboo

Pros:

1. **Integration with Atlassian Suite**: Seamlessly integrates with other Atlassian products like Jira and Bitbucket, providing a cohesive development environment.
2. **Deployment Projects**: Offers robust deployment projects that allow for complex release management.
3. **Customizable Agents**: Allows customization of build agents to fit specific needs.
4. **Flexibility**: Highly flexible in terms of configurations and scripting languages.

Cons:

1. **User Interface**: The UI can be less intuitive compared to other modern CI/CD tools.
2. **Performance**: Some users report performance issues, particularly with larger builds.
3. **Cost**: Requires a license, which can be costly, particularly for small teams or startups.
4. **Resource Intensive**: Self-hosted solution can require significant resources and maintenance.

GitHub Actions

Pros:

1. **Integration with GitHub**: Provides seamless integration with GitHub repositories, making it easy to set up and manage CI/CD pipelines directly within the platform.
2. **Flexibility and Customization**: Highly flexible with a wide range of pre-built actions and the ability to create custom actions.
3. **Community and Marketplace**: Large community and extensive marketplace with reusable workflows and actions.
4. **Cost**: Offers a generous free tier, particularly beneficial for open-source projects.
5. **Ease of Use**: Simple to get started with GitHub's YAML-based workflows.

Cons:

1. **Complex Workflows**: Managing complex workflows can become challenging, especially with extensive YAML configurations.
2. **Limited to GitHub**: Primarily designed for GitHub repositories, so it might not be the best fit for teams using other VCS platforms.
3. **Performance**: Some users report slow performance on the free tier, particularly for resource-intensive tasks.
4. **Support**: Support can be limited, with users relying heavily on community resources for troubleshooting.

Step-by-Step Guide to Setting Up a Basic CI Pipeline with Maven and BDD

1. Connecting to a VCS

Version Control Systems (VCS): Importance in CI

- **Collaboration**: VCS like Git enables multiple developers to work on the same codebase simultaneously, facilitating team collaboration and reducing conflicts.
- **History and Tracking**: It maintains a complete history of all changes, allowing developers to track modifications, identify contributors, and understand the context of changes.
- **Branching and Merging**: VCS supports branching, enabling developers to work on features or fixes in isolation and merge them back into the main codebase when ready.
- **Backup and Recovery**: VCS acts as a backup system, allowing the recovery of previous versions of the codebase in case of errors or issues.
- **Integration with CI Tools**: CI tools use VCS to trigger automated builds and tests whenever changes are committed, ensuring continuous integration and delivery.
- **Example with GitHub**:
    - Create a new repository on GitHub.
    - Push your local repository to GitHub using the remote URL provided.

2. Setting Up Your Maven Project for BDD

Ensure your Maven project is configured for BDD. This typically involves adding dependencies for Cucumber and JUnit in your `pom.xml` file.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>bdd-api-automation</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- TestNG Dependency -->
```

```
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>7.4.0</version>
            <scope>test</scope>
        </dependency>
        <!-- Rest Assured Dependency -->
        <dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>rest-assured</artifactId>
            <version>4.3.3</version>
        </dependency>
        <!-- Cucumber Dependencies -->
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-java</artifactId>
            <version>6.10.4</version>
        </dependency>
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-testng</artifactId>
            <version>6.10.4</version>
        </dependency>
        <!-- JSON Schema Validator Dependency -->
        <dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>json-schema-validator</artifactId>
            <version>4.3.3</version>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>7.4.0</version>
            <scope>compile</scope>
        </dependency>
        <!-- JUnit for running tests -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
        <!-- Logging -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.30</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.30</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.1</version>
```

```
            <scope>compile</scope>
        </dependency>
    </dependencies>
    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.0.0-M5</version>
                <configuration>
                    <includes>
                        <include>**/runners/RunCucumberTest.java</include>
                    </includes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

3. Creating Feature Files and Step Definitions

Create your feature files in the `src/test/resources` directory and your step definitions in the `src/test/java` directory.

Example Feature File (`src/test/resources/features/api.feature`):

```
Feature: API Testing

 Scenario: Get user details
   Given I set the API endpoint to "/user"
   When I send a GET request
   Then I should receive a 200 status code
   And the response should contain user details
```

Example Step Definitions (`src/test/java/steps/APISteps.java`):

```
package steps;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;
```

```
import io.restassured.RestAssured;
import io.restassured.response.Response;

public class APISteps {
    private Response response;

    @Given("I set the API endpoint to {string}")
    public void i_set_the_API_endpoint_to(String endpoint) {
        RestAssured.baseURI = "https://api.example.com" + endpoint;
    }

    @When("I send a GET request")
    public void i_send_a_GET_request() {
        response = RestAssured.get();
    }

    @Then("I should receive a {int} status code")
    public void i_should_receive_a_status_code(int statusCode) {
        assertThat(response.getStatusCode(), equalTo(statusCode));
    }

    @Then("the response should contain user details")
    public void the_response_should_contain_user_details() {
        // Add assertions to verify the response body
    }
}
```

Example Test Runner (`src/test/java/RunCucumberTest.java`):

```
import org.junit.runner.RunWith;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)
@CucumberOptions(features = "src/test/resources/features", glue = "steps")
public class RunCucumberTest {
}
```

4. Configuring the CI Pipeline

1. **Create a Workflow File**:
   - In your repository, create a directory `.github/workflows`.
   - Create a file `ci.yml` in this directory.
2. **Define the Workflow**:

```yaml
name: CI/CD Pipeline

on:
 push:
   branches:
     - master
 pull_request:
   branches:
     - master

jobs:
 build:
   runs-on: ubuntu-latest

   steps:
     - name: Checkout code
       uses: actions/checkout@v2

     - name: Set up JDK 17
       uses: actions/setup-java@v2
       with:
         distribution: 'temurin'
         java-version: '17'

     - name: Cache Maven packages
       uses: actions/cache@v2
       with:
         path: ~/.m2/repository
         key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
         restore-keys: ${{ runner.os }}-maven

     - name: Build with Maven
       run: mvn clean install --no-transfer-progress

     - name: Run tests
       run: mvn test --no-transfer-progress

     - name: Archive test results
       if: always()
       uses: actions/upload-artifact@v2
       with:
         name: test-results
         path: target/surefire-reports/*.xml

 deploy:
   needs: build
   runs-on: ubuntu-latest
   if: github.ref == 'refs/heads/master'
```

```
  steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up JDK 17
      uses: actions/setup-java@v2
      with:
        distribution: 'temurin'
        java-version: '17'

    - name: Deploy to production
      run: echo "Deploying to production..."
      # Add your deployment steps here
```

5. Running Automated Tests

- **Example in the Workflow File**:

```
- name: Run tests
 run: mvn test --no-transfer-progress
```

6. Handling Build Artifacts

- **Example with GitHub Actions**:

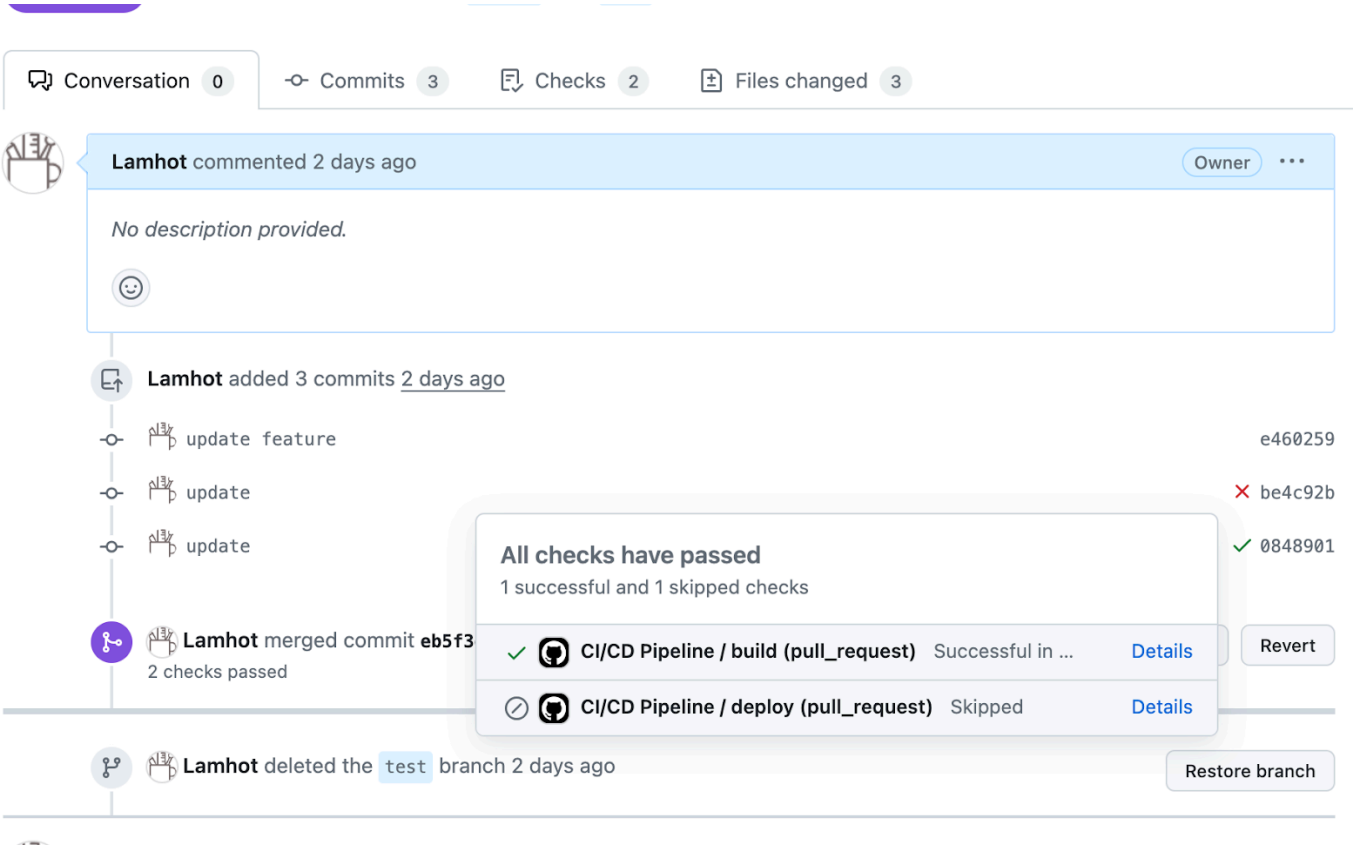
```
- name: Archive test results
 if: always()
 uses: actions/upload-artifact@v2
 with:
   name: test-results
   path: target/surefire-reports/*.xml
```

7. Push your code to GitHub

After making changes, you can push your changes to Git Hub.

Here is the sample PR

https://github.com/Lamhot/api-bdd-testing/pull/1



Common Interview Questions and Answers Related to CI/CD

1. What is CI/CD?

Answer:
CI/CD stands for Continuous Integration and Continuous Deployment (or Continuous Delivery). CI is a development practice where developers integrate code into a shared repository frequently. Each integration is verified by an automated build and automated tests. CD is the next step, where code changes are automatically built, tested, and deployed to production.

2. Why is CI/CD important?

Answer:
CI/CD helps ensure code quality, reduce bugs, and improve software delivery speed. By automating the build, test, and deployment processes, teams can catch issues early and deliver updates more frequently and reliably.

3. What tools are commonly used for CI/CD?

Answer:
Common tools include Jenkins, Travis CI, CircleCI, GitLab CI, Bamboo, and Azure DevOps for CI/CD, along with Docker and Kubernetes for containerization and orchestration.

4. What is containerization, and how does it relate to CI/CD?

Answer:
Containerization involves packaging an application and its dependencies into a container that can run consistently across different environments. Tools like Docker are used in CI/CD pipelines to ensure consistent deployment and simplify scaling and orchestration.

5. How do you measure the success of a CI/CD pipeline?

Answer:
Success can be measured by metrics such as build frequency, build success rate, deployment frequency, mean time to recovery (MTTR), and lead time for changes. Monitoring these metrics helps identify bottlenecks and improve the pipeline.

6. What are some common challenges in implementing CI/CD?

Answer:
Challenges include integrating various tools, managing complex dependencies, handling environment differences, ensuring test coverage and reliability, and dealing with cultural and organizational resistance to change.

7. How do you ensure quality in a CI/CD pipeline?

Answer:
Ensure quality by implementing comprehensive automated testing, including unit, integration, and acceptance tests. Incorporate static code analysis, code reviews, and continuous monitoring. Regularly review and improve pipeline configurations and processes.

8. How would you integrate API tests into a CI/CD pipeline?

Answer:
API tests can be integrated into a CI/CD pipeline using tools like Jenkins or GitLab CI by defining steps in the pipeline configuration file to run the tests automatically after the build phase. This can include using testing frameworks like Postman, Newman, or RestAssured.

9. What is the purpose of a build pipeline?

Answer:
A build pipeline automates the process of building, testing, and deploying code. It ensures that code changes are consistently integrated, tested, and delivered, helping to catch issues early and streamline the development process.

10. What are the common stages in a CI/CD pipeline?

Answer:
Common stages include:

- **Source**: Code is committed and pushed to a version control system.
- **Build**: Code is compiled and built.
- **Test**: Automated tests are run.
- **Deploy**: Code is deployed to staging/production environments.
- **Monitor**: The deployed application is monitored for issues.

11. How do you ensure the security of APIs in CI/CD?

Answer:
Ensure API security by incorporating security testing into the CI/CD pipeline. This can include using tools for vulnerability scanning, static code analysis, and running security tests to identify and fix security issues early in the development process.

12. What is continuous testing?

Answer:
Continuous testing involves running automated tests throughout the software development lifecycle. It aims to provide rapid feedback on the quality of the code, ensuring that new code changes do not break existing functionality.

13. What is the difference between unit testing and integration testing in the context of APIs?

Answer:
Unit testing focuses on testing individual components or functions in isolation. Integration testing examines how different components or services work together, ensuring they interact correctly and produce the expected outcomes.

14. What is the role of a version control system in CI/CD?

Answer:
A version control system (VCS) like Git allows multiple developers to work on code simultaneously while keeping track of changes. It is integral to CI/CD as it triggers pipeline actions (like build and test) upon code commits or pull requests.

15. What is a rollback in the context of CI/CD?

Answer:
A rollback is the process of reverting to a previous stable version of the application in case the current deployment causes issues. Automated rollback strategies can be part of a CI/CD pipeline to ensure minimal downtime and quick recovery.

16. How do you handle environment-specific configurations in CI/CD?

Answer:
Environment-specific configurations can be managed using environment variables, configuration files, or secret management tools like Vault or AWS Secrets Manager. CI/CD tools allow for configuring different settings for different environments (development, staging, production).

Chapter 10

Test Driven Development for Spring Micro Service with Cucumber

Introduction to Test Driven Development (TDD)

What is Test Driven Development (TDD)?

Test Driven Development (TDD) is a software development methodology where tests are written before the code that needs to be tested. It follows a repetitive cycle of writing a test, writing the minimum amount of code to pass the test, and then refactoring the code. This process ensures that the codebase remains clean, maintainable, and well-tested.

Principles of TDD

TDD is based on three core principles:

1. **Red**: Write a test that defines a function or improvements of a function, which fails initially because the function doesn't exist.
2. **Green**: Write the minimal code necessary to pass the test.
3. **Refactor**: Clean up the code while ensuring that all tests still pass. Refactoring is crucial to maintain code quality and readability.

Benefits of TDD

- **Improved Code Quality**: Writing tests first ensures that the code meets the requirements and handles edge cases effectively.
- **Less Debugging**: Since tests are written upfront, it's easier to identify and fix issues early in the development process.
- **Documentation**: Tests serve as a form of documentation, providing clear examples of how the code is supposed to work.
- **Design Improvement**: TDD encourages better design decisions as it requires developers to think about the code's behavior before implementation.

Overview of the TDD Cycle

The TDD cycle consists of three main steps:

1. **Red Phase**:
    - Write a test for a new feature or a piece of functionality.

    - Run the test to see it fail. This step ensures that the test is valid and the functionality is not yet implemented.
2. **Green Phase**:
    - Write the simplest code possible to make the test pass.
    - Run the tests to ensure the new code passes the test.
3. **Refactor Phase**:
    - Refactor the code to improve its structure and readability.
    - Ensure that all tests still pass after refactoring.

Introduction to Microservice

Microservices is an architectural style that structures an application as a collection of small, autonomous services modeled around a business domain. This approach allows for easier scaling, maintenance, and deployment of individual services. Spring Boot, a project from the Spring framework, is widely used to create production-ready, stand-alone applications that run on the JVM. It's particularly well-suited for building microservices due to its simplicity, flexibility, and support for cloud-based deployments.

Getting started with microservices using Spring Boot

1. Setup Your Development Environment

Before you start, ensure you have the following tools installed:

- **Java Development Kit (JDK) 8 or higher**
- **Maven or Gradle** for dependency management and build
- **An Integrated Development Environment (IDE)** like IntelliJ IDEA, Eclipse, or VS Code

2. Create a Spring Boot Application

You can create a Spring Boot project using the Spring Initializr:

1. Go to start.spring.io
2. Select **Project**: Maven Project (or Gradle)
3. Select **Language**: Java
4. Select **Spring Boot** version (e.g., 2.7.1)
5. Provide **Project Metadata**: Group, Artifact, Name, Description, Package name, Packaging (Jar/War), Java Version
6. Select dependencies, such as:
    - **Spring Web** for RESTful web services
    - **Spring Boot Actuator** for monitoring and management
    - **Spring Data JPA** for database interaction (optional)
7. Click **Generate** to download a ZIP file containing your new project.

Unzip the downloaded file and open it in your IDE.

3. Building Your First Microservice

1. Set Up Your Spring Boot Application

a. Create a new Spring Boot project using Spring Initializr or your preferred method. Include the following dependencies:

- Spring Web
- Spring Data JPA
- H2 Database
- Spring Boot DevTools (optional for hot reloading)
- Spring Boot Test
- Cucumber

b. Your `pom.xml` should include these dependencies:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-parent</artifactId>
     <version>3.3.1</version>
     <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>spring.bdd.api</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>spring.bdd.api</name>
  <description>Demo project for Spring Boot with BDD</description>
  <url/>
  <licenses>
     <license/>
  </licenses>
  <developers>
     <developer/>
  </developers>
  <scm>
     <connection/>
     <developerConnection/>
     <tag/>
     <url/>
  </scm>
  <properties>
     <java.version>17</java.version>
  </properties>
  <dependencies>
     <dependency>
        <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
  </build>

</project>
```

2. Create the User Entity

Create a `User` entity class:

```
package com.example.spring.bdd.api.entity;


import jakarta.persistence.*;

@Entity
@Table(name = "users")
public class User {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
   private String email;
   private String password;

   // Getters and setters

   public Long getId() {
       return id;
   }

   public void setId(Long id) {
       this.id = id;
   }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

3. Create the User Repository

Create a repository interface:

```
package com.example.spring.bdd.api.repository;

import com.example.spring.bdd.api.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```

4. Create the User Service

Create a service class to handle business logic:

```
package com.example.spring.bdd.api.service;

import com.example.spring.bdd.api.entity.User;
import com.example.spring.bdd.api.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
```

```
import java.util.List;
import java.util.Optional;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User createUser(User user) {
        return userRepository.save(user);
    }

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }

    public User updateUser(Long id, User userDetails) {
        User user = userRepository.findById(id).orElseThrow(() -> new
RuntimeException("User not found"));
        user.setName(userDetails.getName());
        user.setEmail(userDetails.getEmail());
        user.setPassword(userDetails.getPassword());
        return userRepository.save(user);
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

5. Create the User Controller

Create a REST controller to handle HTTP requests:

```
package com.example.spring.bdd.api.controller;

import com.example.spring.bdd.api.entity.User;
import com.example.spring.bdd.api.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/users")
public class UserController {
```

```
    @Autowired
    private UserService userService;

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        Optional<User> user = userService.getUserById(id);
        return user.map(ResponseEntity::ok).orElseGet(() ->
ResponseEntity.notFound().build());
    }

    @PutMapping("/{id}")
    public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User
userDetails) {
        try {
            User updatedUser = userService.updateUser(id, userDetails);
            return ResponseEntity.ok(updatedUser);
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        try {
            userService.deleteUser(id);
            return ResponseEntity.noContent().build();
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }
}
```

6. Configure H2 Database

Add H2 configuration in `application.properties`:

```
spring.application.name=spring.bdd.api
# src/main/resources/application.properties
spring.h2.console.enabled=true
# default path: h2-console
spring.h2.console.path=/h2-ui

spring.datasource.url=jdbc:h2:file:./testdb
```

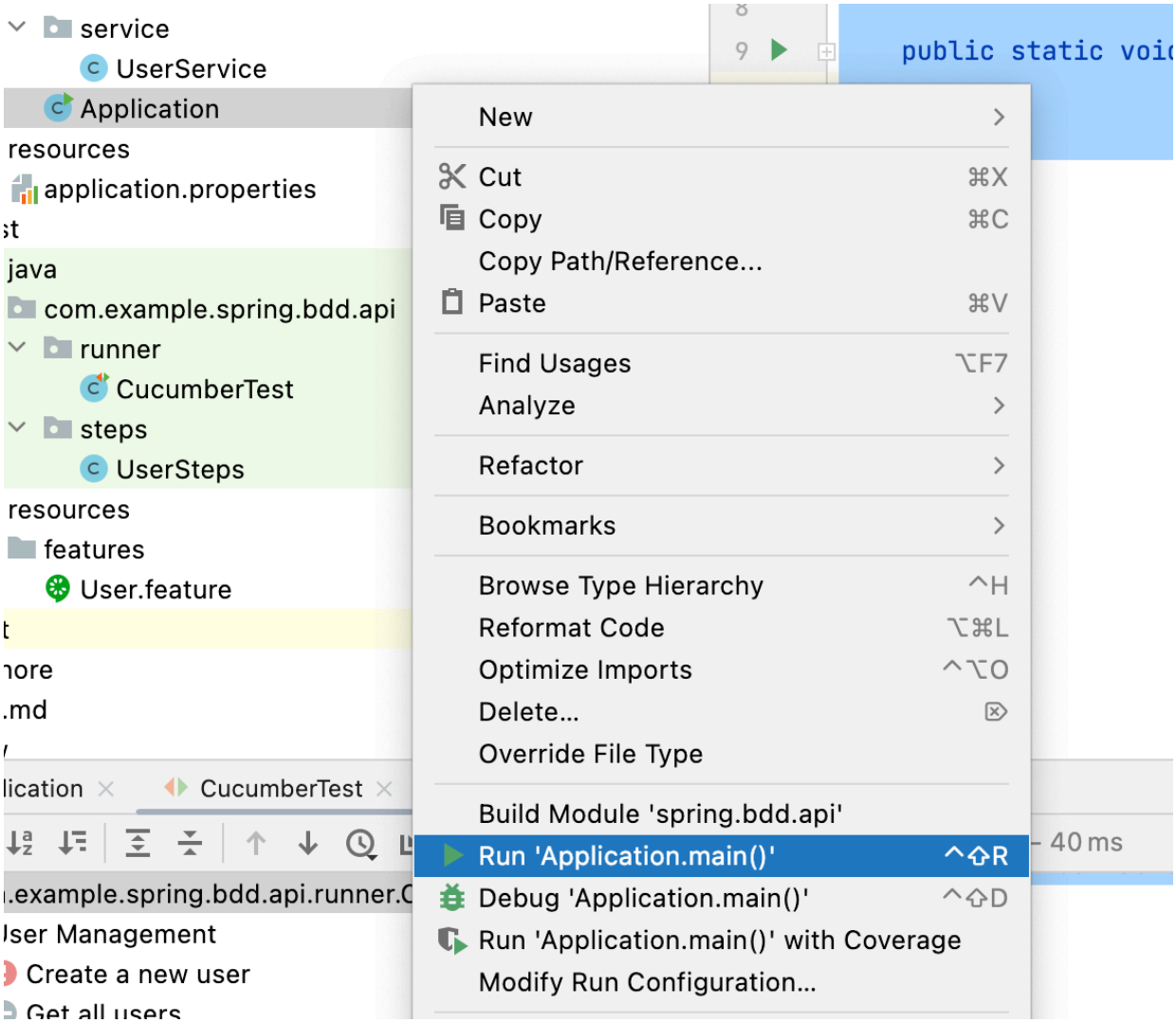
```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto= update
```

7. Define the Application Class

The entry point for your Spring Boot application is the `@SpringBootApplication` annotated class. Here's an example:

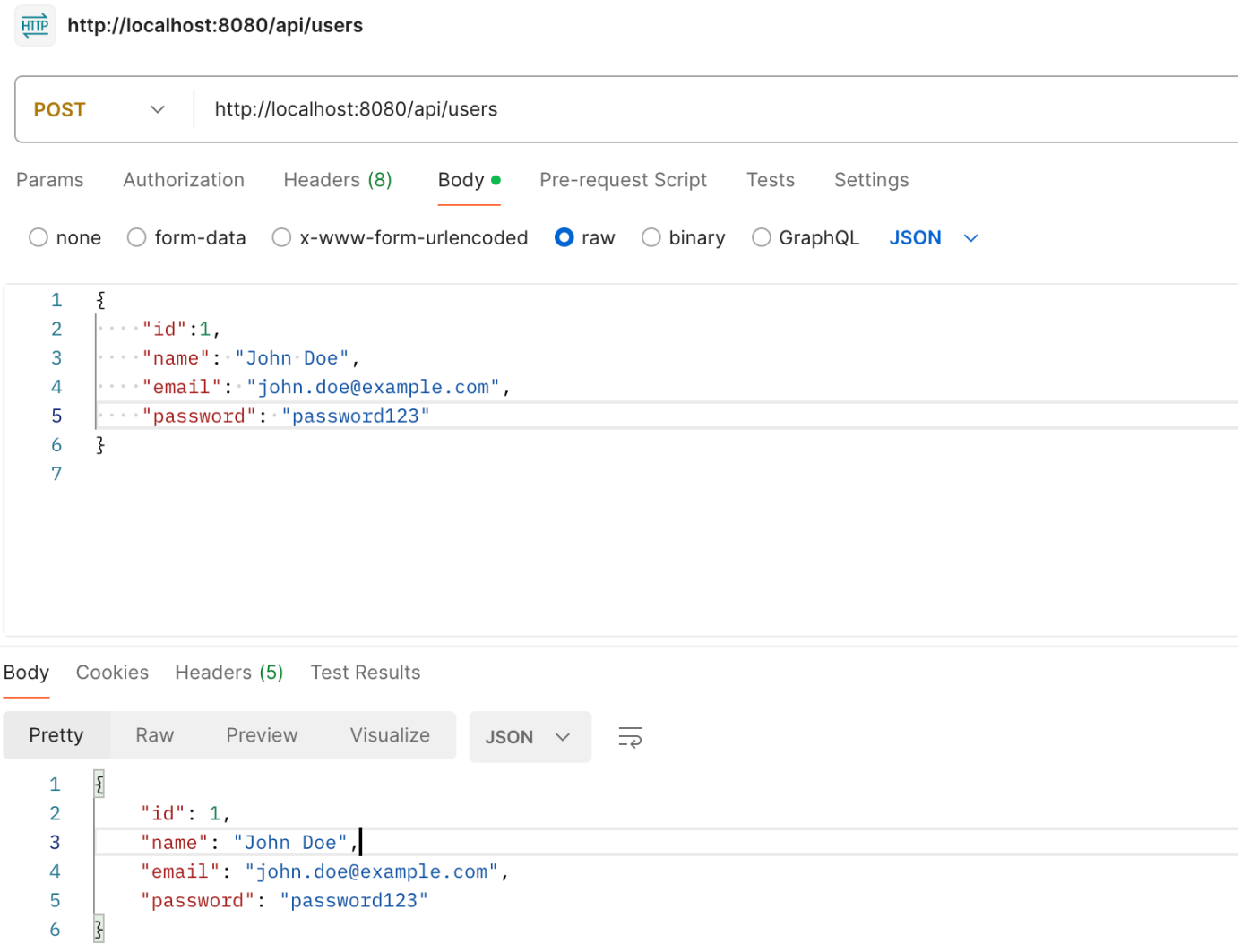
```
package com.example.spring.bdd.api;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

  public static void main(String[] args) {
     SpringApplication.run(Application.class, args);
  }

}
```

8. Run your app and make sure the app is running.



Use this curl request to test your endpoint.

```
curl --location 'http://localhost:8080/api/users' \
--header 'Content-Type: application/json' \
--data-raw '{
   "id":1,
   "name": "John Doe",
   "email": "john.doe@example.com",
   "password": "password123"
}
'
```


http://localhost:8080/api/users
POST
http://localhost:8080/api/users
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"id":1,
"name": "John Doe",
"email": "john.doe@example.com",
"password": "password123"
}
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"id": 1,
"name": "John Doe",
"email": "john.doe@example.com",
"password": "password123"
}

Write BDD Tests

1. Update your pom.xml

You can use a testing framework like Cucumber along with Spring Boot Test to write BDD tests.

```
<dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-java</artifactId>
        <version>7.0.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-spring</artifactId>
        <version>7.0.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-junit</artifactId>
        <version>7.0.0</version>
        <scope>test</scope>
    </dependency>
```

2. Create a feature file `src/test/resources/features/user.feature`:

```
Feature: User Management

 Scenario: Create a new user
   Given I create a user with name "John Doe" and email "john.doe@example.com"
   Then the user is created successfully

 Scenario: Get all users
   When I get all users
   Then I receive a list of users

 Scenario: Get a user by ID
   Given a user with name "John Doe" and email "john.doe@example.com" exists
   When I get the user by ID
   Then I receive the user details

 Scenario: Update a user
   Given a user with name "Jane Doe" and email "jane.doe@example.com" exists
   When I update the user's name to "Jane Smith" and email to
"jane.smith@example.com"
   Then the user is updated successfully

 Scenario: Delete a user
   Given a user with name "John Doe" and email "john.doe@example.com" exists
   When I delete the user by ID
   Then the user is deleted successfully
```

3. Create Step Definitions

Create step definitions
`src/test/com.example.spring.bdd.api.steps/steps/UserSteps.java:`

```
package com.example.spring.bdd.api.steps;

import com.example.spring.bdd.api.Application;
import com.example.spring.bdd.api.entity.User;
import com.example.spring.bdd.api.repository.UserRepository;
import io.cucumber.java.Before;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.jupiter.api.Assertions;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;

import java.util.List;

@SpringBootTest(classes = Application.class, webEnvironment =
SpringBootTest.WebEnvironment.DEFINED_PORT)
public class UserSteps {

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private UserRepository userRepository;

    private ResponseEntity<User> responseEntity;
    private ResponseEntity<User[]> responseEntityList;
    private User createdUser;

    @Before
    public void setUp() {
        userRepository.deleteAll();
    }

    @Given("I create a user with name {string} and email {string}")
    public void i_create_a_user_with_name_and_email(String name, String email) {
        User user = new User();
        user.setName(name);
        user.setEmail(email);
        responseEntity = restTemplate.postForEntity("http://localhost:8080/api/users",
user, User.class);
        createdUser = responseEntity.getBody();
    }

    @Then("the user is created successfully")
    public void the_user_is_created_successfully() {
        Assertions.assertNotNull(createdUser.getId());
        Assertions.assertEquals(HttpStatus.CREATED, responseEntity.getStatusCode());
```

```
    }

    @When("I get all users")
    public void i_get_all_users() {
        responseEntityList =
restTemplate.getForEntity("http://localhost:8080/api/users", User[].class);
    }

    @Then("I receive a list of users")
    public void i_receive_a_list_of_users() {
        Assertions.assertTrue(responseEntityList.getBody().length > 0);
        Assertions.assertEquals(HttpStatus.OK, responseEntityList.getStatusCode());
    }

    @Given("a user with name {string} and email {string} exists")
    public void a_user_with_name_and_email_exists(String name, String email) {
        User user = new User();
        user.setName(name);
        user.setEmail(email);
        createdUser = userRepository.save(user);
    }

    @When("I get the user by ID")
    public void i_get_the_user_by_id() {
        responseEntity = restTemplate.getForEntity("http://localhost:8080/api/users/"
+ createdUser.getId(), User.class);
    }

    @Then("I receive the user details")
    public void i_receive_the_user_details() {
        User user = responseEntity.getBody();
        Assertions.assertNotNull(user);
        Assertions.assertEquals(createdUser.getId(), user.getId());
        Assertions.assertEquals(createdUser.getName(), user.getName());
        Assertions.assertEquals(createdUser.getEmail(), user.getEmail());
        Assertions.assertEquals(HttpStatus.OK, responseEntity.getStatusCode());
    }

    @When("I update the user's name to {string} and email to {string}")
    public void i_update_the_user_s_name_to_and_email_to(String newName, String
newEmail) {
        createdUser.setName(newName);
        createdUser.setEmail(newEmail);
        restTemplate.put("http://localhost:8080/api/users/" + createdUser.getId(),
createdUser);
    }

    @Then("the user is updated successfully")
    public void the_user_is_updated_successfully() {
        User updatedUser = userRepository.findById(createdUser.getId()).orElse(null);
        Assertions.assertNotNull(updatedUser);
        Assertions.assertEquals(createdUser.getId(), updatedUser.getId());
        Assertions.assertEquals(createdUser.getName(), updatedUser.getName());
        Assertions.assertEquals(createdUser.getEmail(), updatedUser.getEmail());
    }

    @When("I delete the user by ID")
```

```
    public void i_delete_the_user_by_id() {
        restTemplate.delete("http://localhost:8080/api/users/" + createdUser.getId());
    }

    @Then("the user is deleted successfully")
    public void the_user_is_deleted_successfully() {
        boolean userExists = userRepository.existsById(createdUser.getId());
        Assertions.assertFalse(userExists);
    }
}
```

4. Create Test Runner

Create a test runner `src/test/java/com/example/demo/CucumberTest.java`:

```
package com.example.spring.bdd.api.runner;

import com.example.spring.bdd.api.Application;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import io.cucumber.spring.CucumberContextConfiguration;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ContextConfiguration;

@RunWith(Cucumber.class)
@CucumberContextConfiguration
@ContextConfiguration(classes = {Application.class})
@CucumberOptions(features = "src/test/resources/features", glue =
"com.example.spring.bdd.api.steps")
public class CucumberTest {
}
```

References

1. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
2. https://toolsqa.com/postman/response-in-postman/
3. https://blog.postman.com/what-are-http-methods/
4. https://testfully.io/blog/http-methods/
5. https://en.wikipedia.org/wiki/HTTP
6. https://katalon.com/resources-center/blog/test-case-template-examples
7. https://www.guru99.com/testing-rest-api-manually.html
8. https://www.softwaretestinghelp.com/rest-api-testing-with-bdd-cucumber/
9. https://nonamesecurity.com/learn/what-is-api-security-testing/
10. https://nonamesecurity.com/wp-content/uploads/2023/06/Product-Brief_SecurityTesting.pdf
11. https://nordicapis.com/how-to-implement-input-validation-for-apis/
12. https://apidog.com/blog/rate-limiting-vs-throttling/
13. https://www.testingxperts.com/blog/api-security-testing#What%20is%20API%20Security%20Testing