Test Automation End Term Study Material

**UNIT IV: API Testing with Postman and Rest Assured**

**1. Introduction to API Testing**

1. **What is an API?**
   * An **Application Programming Interface (API)** is a contract or interface that allows two software components to communicate with each other using defined protocols and data formats (e.g., JSON, XML).
   * Common API styles:
     + **REST (Representational State Transfer)**: HTTP-based, stateless, and resource-oriented.
     + **SOAP (Simple Object Access Protocol)**: Uses XML-based messages.
     + **GraphQL**: Query-based, allowing clients to specify exactly what data they need.
     + **gRPC**: High-performance, binary protocol often used for microservices.
2. **Why is API Testing Important?**
   * **Early Defect Detection**: Catch issues in business logic or integrations before UI testing.
   * **Integration Validation**: Ensures services can communicate correctly in a microservices or distributed architecture.
   * **Faster Feedback**: API tests typically run faster than UI tests, providing quicker validation of core functionality.
   * **Reduced Maintenance**: APIs are more stable than GUIs, so API tests often require fewer updates when UI changes.
3. **Key Concepts in API Testing**
   * **Request Methods**: GET, POST, PUT, DELETE, PATCH, etc.
   * **Request Headers**: Content-Type, Authorization, Accept, etc.
   * **Request Body**: Data sent with POST/PUT (often JSON or XML).
   * **Response Codes**: 2xx (success), 4xx (client errors), 5xx (server errors).
   * **Response Body**: JSON, XML, or other formats containing actual data or error details.
   * **Authentication & Authorization**: Basic auth, OAuth2, JWT tokens, API keys.
4. **Common Testing Types**
   * **Functional Testing**: Validate each endpoint’s core functionality against expected requirements.
   * **Regression Testing**: Ensure new changes do not break existing functionality.
   * **Performance & Load Testing**: Check speed, stability, and scalability under various loads.
   * **Security Testing**: Verify authentication, authorization, and data privacy (e.g., testing for SQL injection or XSS).
   * **Contract Testing**: Ensure provider and consumer services adhere to a defined contract (e.g., using Pact).

**2. API Testing with Postman**

**2.1 Introduction to Postman**

1. **Overview**
   * Postman is a graphical (and web-based) tool that helps you craft, organize, and automate API requests.
   * It supports **variables**, **scripting**, **collections**, and **environment management**, making it a robust solution for exploratory and automated API testing.
2. **Key Features**
   * **Collections**: Group related requests, making it easy to manage large sets of tests.
   * **Environments**: Store variables (e.g., base URLs, tokens) for different stages (dev, staging, prod).
   * **Scripting**: Use JavaScript-based scripts in Pre-request and Test script sections.
   * **Collaboration**: Team workspaces, version history, and real-time collaboration for multiple testers.

**2.2 Designing and Sending API Requests with Postman**

1. **Creating a Request**
   * **Method**: Choose GET, POST, PUT, DELETE, etc.
   * **URL**: Enter the endpoint (e.g., https://api.example.com/users).
   * **Headers**: Configure Content-Type: application/json, Accept: application/json, etc.
   * **Body**: For POST/PUT requests, provide JSON/XML in raw or form-data format.
2. **Using Variables and Environments**
   * **Environment Variables**: Store values like {{baseUrl}} or {{authToken}}.
   * **Global Variables**: Useful for data needed across multiple collections or environments.
   * **Local/Collection Variables**: Scope-specific for a single collection or request.
   * **Practical Example**:

{

"baseUrl": "https://dev.example.com/api",

"authToken": "Bearer 123abc"

}

* + **Dynamic Data**: Generate timestamps, random strings, or GUIDs in Pre-request scripts.

1. **Chaining Requests**
   * **Data Transfer**: Capture a value (e.g., user ID) from one response and store it in a variable.
   * **Subsequent Requests**: Use the stored variable in another request’s URL or body.

**2.3 Validating API Responses in Postman**

1. **Test Scripts (Post-Request)**
   * **JavaScript-based Assertions**:

pm.test("Status code is 200", () => {

pm.response.to.have.status(200);

});

let jsonData = pm.response.json();

pm.test("Check user name", () => {

pm.expect(jsonData.name).to.eql("John Doe");

});

* + **Assertions**: Response code, body fields, headers, response time, etc.

1. **Pre-Request Scripts**
   * **Usage**: Generate dynamic data, set or update environment variables, handle authentication tokens.
   * **Example**:
   * pm.environment.set("timestamp", new Date().toISOString());
2. **Error Handling & Edge Cases**
   * **Negative Tests**: Expecting 4xx or 5xx responses when sending invalid data or unauthorized requests.
   * **Boundary Conditions**: Large inputs, empty inputs, or invalid JSON structures.

**2.4 Advanced Postman Features**

1. **Newman CLI**
   * **Purpose**: Run Postman collections from the command line for automation.
   * **Integration with CI**: Plug into Jenkins, GitLab CI, or GitHub Actions for continuous testing.
   * **Data-Driven Testing**: Provide CSV/JSON data files to iterate over multiple test scenarios.
2. **Mock Servers**
   * **Usage**: Simulate API endpoints for development or testing when the real service is unavailable.
   * **Example**: Define sample responses in Postman, then direct your requests to the Mock Server URL.
3. **Collaboration & Version Control**
   * **Postman Cloud**: Team workspaces and version history for real-time collaboration.
   * **Export & Import**: Collections and environments can be stored in Git for versioning.
4. **Best Practices**
   * **Organize Collections**: Separate by feature, endpoint, or user story.
   * **Keep Tests Modular**: Reuse environment variables and scripts.
   * **Automate**: Use Newman or Postman’s built-in monitors to schedule tests regularly.

**3. API Testing with Rest Assured**

**3.1 Overview of Rest Assured**

1. **What is Rest Assured?**
   * A Java-based library that simplifies testing of RESTful APIs by providing a **fluent DSL**.
   * Ideal for backend test automation in the JVM ecosystem.
2. **Advantages**
   * **Readable Tests**: BDD-like syntax (given().when().then()) is easy to understand.
   * **Integration with JUnit/TestNG**: Combine with popular Java test runners for parallelization and reporting.
   * **Supports JSON/XML**: Built-in parsing and validation for common data formats.
   * **Extensive Features**: Authentication, request/response specifications, logging, etc.

**3.2 Setting Up Rest Assured**

1. **Maven Dependency**

<dependency>

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

<artifactId>rest-assured</artifactId>

<version>5.3.0</version>

<scope>test</scope>

</dependency>

1. **Gradle Example**

testImplementation 'io.rest-assured:rest-assured:5.3.0'

1. **Static Imports**
   * Common practice to import Rest Assured methods (e.g., import static io.restassured.RestAssured.\*;).

**3.3 Writing Tests in Rest Assured**

1. **Basic Structure**

import static io.restassured.RestAssured.\*;

import static org.hamcrest.Matchers.\*;

import org.junit.Test;

public class UserApiTest {

@Test

public void createUserTest() {

given().

baseUri("https://api.example.com").

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

body("{\"name\": \"John\", \"role\": \"tester\"}").

when().

post("/users").

then().

statusCode(201).

body("name", equalTo("John"));

}

}

1. **Common Assertions**
   * **Status Code**: statusCode(200)
   * **JSON Fields**: body("fieldName", equalTo("value"))
   * **Headers**: header("Content-Type", "application/json")
2. **Advanced Usage**
   * **Authentication**: auth().basic("user","pass") or oauth2("token").
   * **Logging**: log().all() to debug requests/responses.
   * **Request/Response Specifications**: Reusable configs to avoid repetition.
   * **Path Parameters & Query Parameters**:

given().

baseUri("https://api.example.com").

pathParam("userId", 123).

when().

get("/users/{userId}").

then().

statusCode(200);

**3.4 Best Practices and Patterns**

1. **Reusable Utilities**
   * Abstract common steps (e.g., setting baseUri) into a @Before method or a static utility class.
   * Create custom methods for repeated tasks (login, token retrieval, etc.).
2. **Data-Driven Testing**
   * Integrate with JUnit Parameterized tests or TestNG DataProviders to feed multiple data sets into the same test logic.
3. **Parallel Execution**
   * Configure TestNG or JUnit to run tests in parallel, reducing total runtime.
   * Ensure concurrency-safe usage of shared resources (e.g., unique test data).
4. **Combining Postman & Rest Assured**
   * Postman is excellent for quick exploration and collaboration.
   * Rest Assured is great for deeper automation within Java projects.
   * Teams often use both: Postman for manual checks and collaboration, Rest Assured for formal regression suites.

**4. Additional API Testing Considerations**

1. **Performance Testing**
   * Tools like **JMeter**, **Gatling**, or **k6** can be used for load and stress tests on APIs.
   * Integrate with CI/CD for continuous performance monitoring.
2. **Contract Testing**
   * Tools like **Pact** verify that consumer and provider services adhere to a defined contract, reducing integration issues.
3. **Security and Penetration Testing**
   * Check for vulnerabilities like SQL injection, XSS, or insecure authentication.
   * Tools like **OWASP ZAP** can be integrated to scan APIs automatically.
4. **Reporting and Analytics**
   * Ensure your automation generates clear, actionable reports (HTML, JSON, JUnit XML).
   * Integrate with dashboards (e.g., **Allure**, **Extent Reports**) for visually appealing results.

**UNIT V: Cloud-Based Test Automation Infrastructure**

**1. Cloud Considerations for Test Automation**

1. **Scalability**
   * Run large test suites in parallel across multiple cloud nodes.
   * Auto-scale to handle peak demands (e.g., nightly regressions or large-scale performance tests).
2. **Cost Effectiveness**
   * **Pay-as-You-Go**: Only pay for resources when tests run.
   * Eliminate expensive on-premises hardware and maintenance.
3. **Geographic Distribution**
   * Test from different regions to simulate real-world latency or localized scenarios.
   * Helpful for multi-region failover testing.
4. **Security & Compliance**
   * Cloud providers often have built-in encryption and compliance (e.g., GDPR, HIPAA).
   * Use secure channels (TLS/SSL) and proper IAM policies for test environments.

**2. Setting Up a Cloud-Based Environment for Test Execution**

1. **Infrastructure as Code (IaC)**
   * **Terraform**: Create and manage AWS, Azure, GCP resources via declarative .tf files.
   * **AWS CloudFormation / Azure Resource Manager**: Platform-specific templates to provision VMs, containers, or serverless functions.
2. **Containers vs. Virtual Machines**
   * **Containers**: Faster startup, less overhead, ideal for ephemeral test runners.
   * **VMs**: More isolation, can run different OSes, but heavier and slower to provision.
3. **Test Data & Environment Configuration**
   * Store secrets, tokens, or environment-specific variables in secure vaults (AWS Secrets Manager, HashiCorp Vault).
   * For ephemeral environments, spin up containers or VMs for each test run, then tear them down to save costs.
4. **Examples of Cloud-Based Test Execution**
   * **Selenium Grid on AWS EC2**: Dynamically spin up nodes for browser testing.
   * **Docker Swarm / Kubernetes**: Orchestrate containers that run your test suites in parallel.
   * **Serverless Functions**: AWS Lambda or Azure Functions for lightweight test tasks.

**3. Leveraging AWS for Cloud-Based Test Automation**

1. **AWS EC2**
   * **Elastic Compute Cloud**: Spin up Linux/Windows instances for test runners.
   * Use **Auto Scaling Groups** to add or remove instances based on demand.
2. **AWS CodeBuild**
   * Fully managed build and test service.
   * Define commands in buildspec.yml to install dependencies, run tests, and produce artifacts.
   * Example:

version: 0.2

phases:

install:

commands:

- echo "Installing dependencies..."

- npm install

build:

commands:

- echo "Running tests..."

- npm test

artifacts:

files:

- '\*\*/\*'

1. **AWS CodePipeline**
   * Automates the end-to-end release process, including build, test, and deploy stages.
   * Integrates with CodeBuild, Lambda, S3, and third-party tools (e.g., GitHub).
2. **AWS Device Farm**
   * Test mobile and web apps on a wide range of real iOS and Android devices.
   * Supports Appium, Calabash, Espresso, etc. for UI automation.
3. **Security & Governance**
   * **IAM Roles**: Restrict test runners to minimal privileges.
   * **CloudWatch Logs & Metrics**: Centralize test output and track performance.

**4. Utilizing Azure for Cloud-Based Test Automation**

1. **Azure DevOps**
   * **Azure Repos**: Host code and test scripts (Git-based).
   * **Azure Pipelines**: Configure CI/CD pipelines using YAML or Classic UI.
   * **Azure Test Plans**: Manage manual and automated tests with integrated dashboards.
2. **Hosted vs. Self-Hosted Agents**
   * **Hosted Agents**: Microsoft-maintained VMs with common tools pre-installed.
   * **Self-Hosted Agents**: Custom environments if you need specific dependencies or security constraints.
3. **Azure Container Instances (ACI)**
   * Run Docker containers without managing VMs or orchestrators.
   * Good for lightweight, short-lived test executions.
4. **Azure Kubernetes Service (AKS)**
   * Managed Kubernetes service for larger-scale test orchestration.
   * Automate scaling test containers as needed.
5. **Integration and Reporting**
   * **Azure Artifacts**: Host build artifacts or third-party packages.
   * **Dashboards**: Track pass/fail rates, code coverage, performance trends, etc.

**5. Other Cloud Providers and Tools**

1. **Google Cloud Platform (GCP)**
   * **Compute Engine**: VM instances.
   * **Google Kubernetes Engine (GKE)**: Managed Kubernetes clusters.
   * **Cloud Build**: CI/CD platform for building and testing apps.
2. **Cross-Cloud Strategies**
   * **Cloud-Agnostic Tools** (e.g., Terraform, Jenkins, Kubernetes) allow tests to run on multiple clouds.
   * **Hybrid Environments**: Combine on-prem resources with cloud-based test runners.
3. **Cost Management**
   * Monitor usage and set budgets or alerts to prevent excessive spending.
   * Automate resource cleanup (e.g., ephemeral test environments) to avoid idle costs.

**UNIT VI: DevOps Integration and Test Automation Strategies**

**1. Integrating Test Automation into CI/CD Pipelines**

1. **Continuous Integration (CI)**
   * **Definition**: Every code commit triggers a pipeline that compiles, runs tests, and generates artifacts.
   * **Why It Matters**: Immediate feedback on code quality, preventing regressions from creeping in.
2. **Continuous Deployment (CD)**
   * **Definition**: Automates releasing tested code changes into production (or staging) with minimal human intervention.
   * **Role of Test Automation**: Ensures only code passing all tests is deployed, increasing confidence in releases.
3. **Setting Up CI/CD Pipelines**
   * **Jenkins**: Jenkinsfile (pipeline-as-code) to define stages (build, test, deploy).
   * **GitLab CI**: .gitlab-ci.yml with multiple stages, artifacts, and environment variables.
   * **GitHub Actions**: YAML-based workflows triggered on push, pull requests, or scheduled events.
   * **Azure Pipelines**: YAML or Classic editor to define tasks for build, test, and deployment.

**2. Integrating Test Automation Frameworks**

1. **Unit Tests**
   * Validate individual functions or methods in isolation.
   * Run quickly, providing immediate feedback.
   * Examples: JUnit, NUnit, PyTest, Jest.
2. **Integration Tests**
   * Ensure multiple modules or services interact correctly.
   * Often require a test environment with real or mocked dependencies.
   * Example: Testing a microservice that calls another service’s API.
3. **Automated Regression Tests**
   * Catch new bugs introduced in existing functionality.
   * Typically run after each commit or on a nightly schedule.
   * Keep them stable and well-documented to avoid flaky test suites.
4. **Parallel vs. Pipeline-Based Approaches**
   * **Parallel Execution**: Tests run on multiple agents simultaneously, reducing total time.
   * **Pipeline Stages**: Different types of tests (unit, integration, acceptance) run in separate stages, failing early if something is wrong.
5. **Best Practices**
   * **Fail Fast**: If unit tests fail, do not proceed to integration tests.
   * **Use Docker**: Containerize test environments for consistency.
   * **Clean State**: Ensure each pipeline run starts with a fresh environment.

**3. Prioritizing Test Cases and Continuous Feedback**

1. **Risk-Based Testing**
   * Focus on critical functionality and high-risk areas first.
   * E.g., payment processing, authentication, or data integrity tests.
2. **Test Impact Analysis**
   * Identify which tests are affected by recent code changes to run only relevant subsets.
   * Saves time by skipping tests unrelated to changed components.
3. **Continuous Feedback Loops**
   * Automated notifications (email, Slack, Teams) for test failures.
   * Rapid iteration cycles help developers fix issues quickly.
4. **Reporting and Analytics**
   * Generate HTML or PDF reports summarizing pass/fail rates, coverage, performance.
   * Tools like Allure, Extent Reports, SonarQube for deeper analytics.

**4. Test Automation in DevOps Environments**

1. **Shifting Left**
   * Incorporate testing activities as early as possible in the software delivery pipeline.
   * Developers write unit tests and sometimes integration tests, reducing reliance on late-stage QA phases.
2. **Maintaining Test Automation Scripts and Data**
   * **Version Control**: Store test scripts alongside application code for traceability.
   * **Test Data Management**: Use realistic but sanitized data, or dynamically generate data for each run.
   * **Avoid Hard-Coded Values**: Rely on environment variables or configuration files.
3. **Monitoring Test Results and Failures**
   * **Dashboards**: Tools like Grafana or built-in CI dashboards to visualize pass/fail rates, test duration, trends over time.
   * **Root Cause Analysis**: Automate log collection and environment snapshots for faster debugging.
4. **Early Detection and Best Practices**
   * **Code Reviews**: Ensure new tests are added for new features.
   * **Static Code Analysis**: Tools like SonarQube detect code smells or security vulnerabilities.
   * **Shift Right**: Monitor production performance and logs to identify real-world issues early (e.g., chaos engineering, canary releases).
5. **Ephemeral Test Environments**
   * **On-Demand**: Spin up short-lived environments (containers or VMs) for each feature branch.
   * **Isolation**: Prevent conflicts between multiple test runs.
   * **Cost Optimization**: Tear down automatically after tests finish.

**5. Putting It All Together**

1. **End-to-End Example**
   * **Local Development**: Developer writes new feature + unit tests, uses Postman for quick API checks.
   * **Push to Git**: CI pipeline triggers, running unit tests, integration tests (Rest Assured), and coverage checks.
   * **Cloud Environment**: A test environment is spun up automatically on AWS or Azure.
   * **Automated Test Execution**: Postman collections or Rest Assured tests run in parallel.
   * **Reporting**: Results posted to Slack and stored in a dashboard (e.g., Allure).
   * **Deployment**: If all tests pass, the code is deployed to staging or production.
   * **Monitoring**: Production environment is monitored for errors, performance, and security anomalies.
2. **Key Success Factors**
   * **Collaboration**: Dev, QA, and Ops working together with clear communication.
   * **Automation Coverage**: Comprehensive tests covering unit, integration, API, UI, performance, security.
   * **Continuous Improvement**: Regular retrospectives to refine the pipeline, tests, and processes.