Comprehensive Project Interview Questions & Answers

Petstore API Test Automation Framework

Prepared for: Technical Interviews, Code Reviews, and Project Understanding
Project: RestAssured + TestNG API Testing Framework
Date: August 11, 2025

Table of Contents

1. Project Overview Questions
2. Java & Programming Concepts
3. RestAssured Framework Questions
4. TestNG Framework Questions
5. Maven & Build Management
6. Architecture & Design Patterns
7. Authentication & Security
8. Configuration Management
9. Test Data Management
10. Reporting & Logging
11. Error Handling & Validation
12. Code Analysis - Class by Class
13. Advanced Technical Questions
14. Best Practices & Optimization
15. Troubleshooting & Debugging

---

1. PROJECT OVERVIEW QUESTIONS

Q1.1: What is the main purpose of this project?

Answer: This project is an API test automation framework designed to test the Swagger Petstore API. It validates various pet management operations including creating pets, retrieving pets by status, and basic functionality verification through automated tests.

Q1.2: Which testing approach/methodology does this project follow?

Answer: The project follows:

- Black-box testing approach for API testing
- Behavioral-driven development patterns with clear test descriptions

• Page Object Model (POM) for structural organization
• Data-driven testing capabilities through property files

Q1.3: What are the key technologies and frameworks used?

Answer:

• Java 17 - Programming language
• RestAssured 5.4.0 - API testing framework
• TestNG 7.9.0 - Testing framework for test management
• Maven - Build and dependency management
• ExtentReports 5.1.2 - Test reporting
• Jackson 2.16.1 - JSON processing
• JSON.org - JSON manipulation

Q1.4: What type of API is being tested and why was it chosen?

Answer: The Swagger Petstore API (https://petstore.swagger.io/v2) is being tested. It's a well-known demo REST API that provides:

• Complete CRUD operations
• Authentication mechanisms
• Well-documented endpoints
• Stable and reliable for testing purposes
• Industry-standard for learning API testing

==========

2. JAVA & PROGRAMMING CONCEPTS

Q2.1: What Java version is used and what are its key features utilized?

Answer: Java 17 is used (maven.compiler.release=17). Key features utilized:

• Enhanced exception handling with try-with-resources in PropertyReader
• Stream API capabilities (potential for future enhancements)
• Improved performance and security features
• Records and pattern matching (available for future use)

Q2.2: Explain the use of static variables and methods in the project.

Answer:

• Static variables: ExtentReports extent in ExtentManager for singleton implementation

• Static methods: ExtentManager.getInstance() ensures single instance across test execution
• Constants: TEST_PET_ID, PET_NAME in Tests class for consistent test data

Q2.3: How is exception handling implemented?

Answer: Multiple levels of exception handling:

• PropertyReader: Try-catch with try-with-resources for file operations
• BaseSteps: IllegalStateException for missing authentication credentials
• Custom exceptions: IllegalArgumentException for unsupported HTTP methods
• TestNG integration: ITestResult for test failure handling

Q2.4: What OOP principles are demonstrated in this project?

Answer:

• Encapsulation: Private methods and fields with controlled access
• Inheritance: RequestSteps extends BaseSteps
• Polymorphism: Method overloading in executeRequest()
• Abstraction: BaseSteps abstracts common API operations

==========

3. RESTASSURED FRAMEWORK QUESTIONS

Q3.1: What is RestAssured and why is it used for API testing?

Answer: RestAssured is a Java library for testing REST services. Benefits:

• Fluent API design for readable test code
• Built-in JSON/XML parsing and validation
• Integration with testing frameworks (TestNG/JUnit)
• Comprehensive assertion capabilities
• Authentication support for various mechanisms
• Response validation with hamcrest matchers

Q3.2: How is RestAssured configured in the project?

Answer: Configuration is handled in BaseSteps constructor:

```
// Base URI setup
RestAssured.baseURI = baseURL;

// Request specification with headers
```

```
this.request = RestAssured.given()
.header("Content-Type", "application/json")
.header("Accept", "application/json");

// Authentication setup
setupAuthorization();
```

Q3.3: Explain the request building process in your framework.

Answer: Request building follows this flow:

1. Base configuration in BaseSteps constructor
2. Header setup for content-type and accept headers
3. Authentication injection based on configuration
4. Dynamic body addition in executeRequest() method
5. Endpoint resolution through getBasePath() method

Q3.4: How are different HTTP methods handled?

Answer: The executeRequest() method uses a switch statement:

```
switch (method.toUpperCase()) {
case "POST": return body != null ? request.body(body).post(endpoint) :
request.post(endpoint);
case "GET": return request.get(endpoint);
case "PUT": return body != null ? request.body(body).put(endpoint) :
request.put(endpoint);
case "DELETE": return request.delete(endpoint);
default: throw new IllegalArgumentException("Unsupported HTTP method: " + method);
}
```

Q3.5: How do you handle request and response logging?

Answer: Currently implemented through:

• Console logging with System.out.println()
• ExtentReports integration for test step logging
• Response validation with automatic logging
• Future enhancement: RestAssured's built-in logging filters

Q3.6: Explain JSON handling in the project.

Answer: JSON handling is implemented at multiple levels:

• JSONObject creation for request payloads in createPetPayload()

• Jackson databind for complex object mapping (configured but not actively used)
• RestAssured automatic parsing for response handling
• Manual JSON construction for test data flexibility

==========

4. TESTNG FRAMEWORK QUESTIONS

Q4.1: Why was TestNG chosen over JUnit for this project?

Answer: TestNG offers several advantages:

- **Better annotation support** (@BeforeClass, @AfterClass, @BeforeMethod, @AfterMethod)
- **Test prioritization** with priority attribute
- **Group-based testing** (smoke, regression, etc.)
- **Parallel execution** capabilities
- **Better reporting** with built-in HTML reports
- **Data-driven testing** support with @DataProvider

Q4.2: How is test execution order controlled?

Answer: Test execution order is controlled through:

- **Priority attribute**: `@Test(priority = 1)` for testAddPet()
- **Priority attribute**: `@Test(priority = 4)` for testGetPetsByStatus()
- **Group execution**: `@Test(groups = "smoke")` for smokeTest()
- **TestNG XML configuration** for suite-level control

Q4.3: Explain the test lifecycle in your framework.

Answer: Complete lifecycle flow:

1. **@BeforeClass setUp()**: Framework initialization, ExtentReports setup
2. **@BeforeMethod startTest()**: Individual test initialization, test reporting setup
3. **@Test methods**: Actual test execution with business logic
4. **@AfterMethod afterEachTest()**: Result processing, status logging
5. **@AfterClass tearDown()**: Cleanup, report generation, resource closure

Q4.4: How are test groups implemented and used?

Answer:

- **Smoke group**: `@Test(groups = "smoke")` for basic functionality tests

- **Execution**: Can be run independently with `mvn test -Dgroups=smoke`
- **Future scalability**: Framework ready for regression, integration, sanity groups
- **TestNG XML**: Groups can be included/excluded at suite level

Q4.5: How is test result handling implemented?

Answer: Through `afterEachTest()` method using ITestResult:

```java
if (result.getStatus() == ITestResult.FAILURE) {
   Assert.fail("Test failed: " + result.getThrowable());
} else if (result.getStatus() == ITestResult.SUCCESS) {
   test.pass("Test passed");
} else if (result.getStatus() == ITestResult.SKIP) {
   test.skip("Test skipped: " + result.getThrowable());
}
```

---

5. Maven & Build Management

Q5.1: Explain the Maven project structure and its significance.

Answer: Standard Maven directory layout:

- **src/main/java**: Production code (App.java - entry point)
- **src/test/java**: Test code (all test classes)
- **src/test/resources**: Test resources (PropertyFiles)
- **target/**: Compiled classes and generated artifacts
- **pom.xml**: Project configuration and dependencies

Q5.2: What are the key dependencies and their purposes?

Answer: Dependencies breakdown:

- **RestAssured ecosystem**: rest-assured, json-schema-validator, json-path, xml-path
- **Testing framework**: TestNG for test management
- **JSON processing**: Jackson (databind, core, annotations), org.json
- **Logging**: SLF4J API, Logback Classic
- **Utilities**: Apache Commons Lang3, Hamcrest
- **Test data**: DataFaker for data generation
- **Reporting**: ExtentReports for HTML reporting

Q5.3: How would you run tests using Maven commands?

Answer: Various execution options:

```bash
mvn clean test                    # Run all tests
mvn test -Dtest=Tests             # Run specific test class
mvn test -Dgroups=smoke           # Run specific group
mvn clean compile test           # Clean, compile, and test
mvn surefire-report:report       # Generate surefire reports
```

Q5.4: What Maven plugins are configured?

Answer: Standard Maven plugins with specified versions:

- **maven-compiler-plugin 3.13.0**: Java compilation
- **maven-surefire-plugin 3.3.0**: Test execution
- **maven-resources-plugin 3.3.1**: Resource handling
- **maven-clean-plugin 3.4.0**: Cleanup operations

---

6. Architecture & Design Patterns

Q6.1: Which design patterns are implemented in the project?

Answer: Multiple patterns implemented:

- **Page Object Model**: Separation of test logic (Tests.java) and operations (RequestSteps.java)
- **Factory Pattern**: Authentication setup based on configuration type
- **Singleton Pattern**: ExtentManager for single reporting instance
- **Template Method Pattern**: BaseSteps defining common operations
- **Strategy Pattern**: Different authentication strategies

Q6.2: How does the framework follow separation of concerns?

Answer: Clear separation across layers:

- **Configuration Layer**: PropertyReader handles all configuration
- **Base Layer**: BaseSteps handles common API setup and authentication
- **Operation Layer**: RequestSteps handles specific API operations
- **Test Layer**: Tests.java contains only test logic and assertions

- **Utility Layer**: ExtentManager handles reporting concerns

Q6.3: Explain the inheritance hierarchy in the project.

Answer: Simple inheritance structure:

```
BaseSteps (Parent)
   ├── Common API configuration
   ├── Authentication setup
   ├── Request execution logic
   └── Response validation
       |
       └── RequestSteps (Child)
           ├── Pet-specific operations
           ├── JSON payload creation
           └── Business logic methods
```

Q6.4: How is the framework designed for scalability?

Answer: Scalable design through:

- **Modular architecture**: Easy to add new API modules
- **Configuration-driven**: New environments through properties
- **Generic methods**: executeRequest() supports all HTTP methods
- **Extensible authentication**: Multiple auth types supported
- **Reusable components**: Base classes for common functionality

---

7. Authentication & Security

Q7.1: What authentication mechanisms are supported?

Answer: Framework supports four authentication types:

1. **API Key Authentication** (currently used) - Header-based
2. **Bearer Token Authentication** - JWT/OAuth tokens
3. **Basic Authentication** - Username/password
4. **No Authentication** - For public APIs

Q7.2: How is API Key authentication implemented?

Answer: Through `setupApiKeyAuth()` method:

```java
private void setupApiKeyAuth() {
   String apiKey = propertyReader.getProperty("auth.apikey");
   String headerName = propertyReader.getProperty("auth.apikey.header");
   headerName = (headerName != null && !headerName.isEmpty()) ? headerName :
"api_key";
   this.request = this.request.header(headerName, apiKey);
}
```

Q7.3: How would you add a new authentication type?

Answer: Steps to add new authentication:

1. Add new case in `setupAuthorization()` switch statement
2. Create new setup method (e.g., `setupOAuthAuth()`)
3. Add required properties to Property.properties
4. Implement authentication logic using RestAssured methods
5. Add error handling for missing credentials

Q7.4: How are authentication credentials managed securely?

Answer: Security measures implemented:

- **Property file configuration**: Credentials not hard-coded
- **Environment variable support**: Can be enhanced for CI/CD
- **Validation checks**: Error handling for missing credentials
- **Separation of concerns**: Authentication logic separated from tests

---

8. Configuration Management

Q8.1: How is configuration managed in the project?

Answer: Through PropertyReader utility class:

- **Centralized configuration**: Single Property.properties file
- **Type-safe access**: Property validation and default values
- **Resource loading**: Classpath-based resource loading
- **Error handling**: Proper exception handling for missing files

Q8.2: What configuration options are available?

Answer: Configuration categories:

```properties
# Base URL Configuration
baseURL=https://petstore.swagger.io/v2

# Endpoint Paths
basepathPost=/pet
basepathGet=/pet/
basepathPut=/pet
basepathDelete=/pet/

# Authentication
auth.type=apikey
auth.apikey=special-key
auth.apikey.header=api_key
```

Q8.3: How would you add environment-specific configurations?

Answer: Multiple approaches possible:

1. **Profile-based properties**: property-{env}.properties files
2. **Maven profiles**: Different configurations for dev/test/prod
3. **Environment variables**: Override properties with system variables
4. **Command-line parameters**: Runtime configuration override

---

9. Test Data Management

Q9.1: How is test data managed in the project?

Answer: Multi-level test data management:

- **Static constants**: TEST_PET_ID, PET_NAME in Tests class
- **Dynamic generation**: createPetPayload() method for JSON creation
- **Configuration-based**: Endpoint paths from properties
- **Faker library**: DataFaker dependency for future data generation

Q9.2: Explain the JSON payload creation process.

Answer: JSONObject-based payload creation:

```java
```

```
private JSONObject createPetPayload(int id, String name, String status) {
    JSONObject pet = new JSONObject();
    pet.put("id", id);
    pet.put("name", name);
    pet.put("status", status);
    // Additional fields: category, photoUrls, tags
    return pet;
}
```

Q9.3: How would you implement data-driven testing?

Answer: Multiple approaches available:

- **TestNG @DataProvider**: Method-level data provision
- **CSV/Excel integration**: External data source reading
- **JSON file-based**: Test data in separate JSON files
- **Database integration**: Dynamic data from database queries

---

10. Reporting & Logging

Q10.1: What reporting mechanisms are implemented?

Answer: Three-tier reporting system:

1. **ExtentReports**: Detailed HTML reports with test steps
2. **TestNG reports**: Built-in XML and HTML reports
3. **Console logging**: Real-time execution feedback

Q10.2: How is ExtentReports integrated?

Answer: Through ExtentManager singleton:

```java
public static ExtentReports getInstance() {
    if (extent == null) {
        String reportPath = System.getProperty("user.dir") +
"/target/Reports/ExtentReport.html";
        ExtentSparkReporter spark = new ExtentSparkReporter(reportPath);
        extent = new ExtentReports();
        extent.attachReporter(spark);
    }
    return extent;
```

```
}
```

Q10.3: What information is captured in test reports?

Answer: Comprehensive test information:

- **Test execution timeline** and duration
- **Pass/Fail status** with detailed reasons
- **Test step logging** with API call details
- **Response status codes** and validation results
- **Exception details** for failed tests
- **Test grouping** and priority information

---

11. Error Handling & Validation

Q11.1: How are API response validations implemented?

Answer: Through `validateResponse()` method:

```java
protected void validateResponse(Response response, int expectedStatus) {
   response.then().statusCode(expectedStatus);
}
```

Additionally, assertion-based validation in test methods.

Q11.2: What types of validations are performed?

Answer: Multiple validation levels:

- **Status code validation**: HTTP response codes (200, 404, 500, etc.)
- **Response structure validation**: JSON schema validation capability
- **Business logic validation**: Pet ID, name, status field validation
- **Authentication validation**: Proper credential handling

Q11.3: How does the framework handle test failures?

Answer: Multi-level failure handling:

- **RestAssured failures**: Automatic assertion failures for status codes
- **TestNG assertions**: Assert.assertEquals() for business logic

- **Custom exceptions**: Proper exception handling with meaningful messages
- **ExtentReports integration**: Failure logging with detailed information

---

12. Code Analysis - Class by Class

Q12.1: Analyze the BaseSteps.java class in detail.

Answer: **BaseSteps** is the foundation class:

Purpose: Provides common API configuration and authentication setup

Key Responsibilities:

- RestAssured base URI configuration
- Request specification setup with headers
- Multi-type authentication implementation
- Generic HTTP request execution
- Response validation utilities

Key Methods:

```java
// Constructor: Initializes configuration and authentication
public BaseSteps()

// Authentication setup based on configuration
private void setupAuthorization()

// Individual authentication type methods
private void setupBearerAuth()
private void setupBasicAuth()
private void setupApiKeyAuth()

// Generic request execution for all HTTP methods
protected Response executeRequest(String method, String endpoint, Object body)

// Response validation utility
protected void validateResponse(Response response, int expectedStatus)

// Endpoint path resolution
protected String getBasePath(String operation)
```

Design Features:

- **Template method pattern**: Defines common workflow
- **Strategy pattern**: Different authentication strategies
- **Error handling**: Proper exception handling for missing configurations

Q12.2: Analyze the RequestSteps.java class in detail.

Answer: **RequestSteps** extends BaseSteps for pet-specific operations:

Purpose: Implements Pet API specific business logic

Key Responsibilities:

- JSON payload creation for pet data
- Pet management operations (add, retrieve)
- Business logic validation
- API operation logging

Key Methods:

```java
// Pet data model creation
private JSONObject createPetPayload(int id, String name, String status)

// POST operation for adding pets
public Response addPet(int petId, String petName, String status)

// GET operation for retrieving pets by status
public Response getPetsByStatus(String status)
```

Design Features:

- **Inheritance**: Leverages BaseSteps functionality
- **Encapsulation**: Private payload creation method
- **Business abstraction**: Hides technical details from tests

Q12.3: Analyze the Tests.java class in detail.

Answer: **Tests** class contains actual test implementations:

Purpose: Test execution and validation logic

Key Responsibilities:

- Test lifecycle management
- ExtentReports integration
- Test data management
- Assertion and validation

Key Methods:

```java
// Test setup and initialization
@BeforeClass public void setUp()
@BeforeMethod public void startTest(Method method)

// Test methods with priorities and groups
@Test(priority = 1) public void testAddPet()
@Test(priority = 4) public void testGetPetsByStatus()
@Test(groups = "smoke") public void smokeTest()

// Test cleanup and reporting
@AfterMethod public void afterEachTest(ITestResult result)
@AfterClass public void tearDown()
```

Design Features:

- **TestNG lifecycle**: Proper setup and teardown
- **Reporting integration**: ExtentReports test logging
- **Priority-based execution**: Controlled test order
- **Group-based testing**: Smoke test categorization

Q12.4: Analyze the PropertyReader.java class in detail.

Answer: **PropertyReader** handles configuration management:

Purpose: Centralized configuration file handling

Key Responsibilities:

- Properties file loading from classpath
- Type-safe property access
- Error handling for missing files/properties
- Default value support

Key Methods:

```java
// Constructor with file loading
public PropertyReader(String propertiesFile)

// Property file loading with error handling
private void loadProperties(String propertiesFile)

// Property access methods
public String getProperty(String key)
public String getProperty(String key, String defaultValue)
```

Design Features:

- **Resource management**: Try-with-resources for file handling
- **Error handling**: Comprehensive exception handling
- **Utility pattern**: Simple property access interface

Q12.5: Analyze the ExtentManager.java class in detail.

Answer: **ExtentManager** implements reporting management:

Purpose: Singleton pattern for ExtentReports instance management

Key Responsibilities:

- Single ExtentReports instance creation
- Report configuration setup
- Thread-safe instance management

Key Methods:

```java
// Singleton instance creation
public static ExtentReports getInstance()
```

Design Features:

- **Singleton pattern**: Single instance across test execution
- **Lazy initialization**: Instance created when needed
- **Configuration management**: Report path and reporter setup

---

13. Advanced Technical Questions

Q13.1: How would you implement parallel test execution?

Answer: Multiple approaches for parallel execution:

TestNG Level:

```xml
<suite name="Suite" parallel="methods" thread-count="5">
   <test name="Test">
       <classes>
           <class name="com.tests.Tests"/>
       </classes>
   </test>
</suite>
```

Maven Level:

```xml
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <configuration>
       <parallel>methods</parallel>
       <threadCount>5</threadCount>
   </configuration>
</plugin>
```

Considerations:

- Thread-safe test data management
- Unique test identifiers to avoid conflicts
- ExtentReports thread-safety

Q13.2: How would you add database validation to your tests?

Answer: Database integration approach:

1. **Add database dependencies**: JDBC driver, connection pooling
2. **Create database utility class**: Connection management, query execution
3. **Add validation methods**: Data verification after API operations
4. **Integration in test methods**: API call followed by database validation

```java
// Example implementation
public class DatabaseUtils {
    public Pet getPetFromDatabase(int petId) {
        // Database query to retrieve pet
    }

    public void validatePetInDatabase(int petId, String expectedName) {
        Pet pet = getPetFromDatabase(petId);
        Assert.assertEquals(pet.getName(), expectedName);
    }
}
```

Q13.3: How would you implement schema validation for responses?

Answer: JSON Schema validation using RestAssured:

```java
// Add json-schema-validator dependency (already present)
import static io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchema;

// In test method
Response response = petSteps.addPet(TEST_PET_ID, PET_NAME, "available");
response.then()
    .statusCode(200)
    .body(matchesJsonSchema(new File("schemas/pet-schema.json")));
```

Schema file example:

```json
{
 "type": "object",
 "properties": {
    "id": { "type": "integer" },
    "name": { "type": "string" },
    "status": { "type": "string", "enum": ["available", "pending", "sold"] }
 },
 "required": ["id", "name", "status"]
}
```

Q13.4: How would you implement performance testing within this framework?

Answer: Performance testing integration:

1. **Response time validation**:

```java
Response response = petSteps.addPet(TEST_PET_ID, PET_NAME, "available");
long responseTime = response.getTime();
Assert.assertTrue(responseTime < 2000, "API response time exceeded threshold");
```

2. **Load testing integration**:

```java
@Test
public void loadTest() {
   ExecutorService executor = Executors.newFixedThreadPool(10);
   List<Future<Response>> futures = new ArrayList<>();

   for (int i = 0; i < 100; i++) {
       final int petId = i;
       futures.add(executor.submit(() -> petSteps.addPet(petId, "LoadTestPet",
"available")));
   }

   // Validate all responses
}
```

Q13.5: How would you implement continuous integration (CI/CD) for this project?

Answer: CI/CD pipeline setup:

GitHub Actions example:

```yaml
name: API Tests
on: [push, pull_request]

jobs:
 test:
   runs-on: ubuntu-latest
   steps:
     - uses: actions/checkout@v2
     - name: Set up JDK 17
```

```
      uses: actions/setup-java@v2
      with:
        java-version: "17"
    - name: Run tests
      run: mvn clean test
    - name: Generate reports
      uses: actions/upload-artifact@v2
      with:
        name: test-reports
        path: target/reports/
```

Jenkins Pipeline:

```groovy
pipeline {
   agent any
   stages {
       stage('Test') {
           steps {
               sh 'mvn clean test'
           }
       }
       stage('Report') {
           steps {
               publishHTML([
                   allowMissing: false,
                   alwaysLinkToLastBuild: true,
                   keepAll: true,
                   reportDir: 'target/reports',
                   reportFiles: 'ExtentReport.html',
                   reportName: 'API Test Report'
               ])
           }
       }
   }
}
```

---

14. Best Practices & Optimization

Q14.1: What API testing best practices are followed in this project?

Answer: Best practices implemented:

Structure & Organization:

- ✅ Separation of concerns (BaseSteps, RequestSteps, Tests)
- ✅ Reusable components and utilities
- ✅ Configuration-driven approach
- ✅ Proper exception handling

Test Design:

- ✅ Clear test descriptions and priorities
- ✅ Independent test execution
- ✅ Proper assertion strategies
- ✅ Test data management

Areas for improvement:

- ❌ More comprehensive negative testing
- ❌ Response schema validation
- ❌ Performance assertions
- ❌ Test data cleanup

Q14.2: How would you optimize the current framework for better performance?

Answer: Optimization strategies:

Connection Management:

```java
// Implement connection pooling
RestAssured.config = RestAssured.config()
   .httpClient(HttpClientConfig.httpClientConfig()
       .setParam("http.conn-manager.max-total", 100)
       .setParam("http.conn-manager.max-per-route", 20));
```

Request Specification Reuse:

```java
// Create reusable request specifications
RequestSpecification commonSpec = new RequestSpecBuilder()
   .setBaseUri(baseURL)
   .setContentType(ContentType.JSON)
   .build();
```

```

**Parallel Execution**:

- Implement thread-safe test data
- Use TestNG parallel execution
- Optimize reporting for concurrent access

### Q14.3: What security testing aspects could be added?

**Answer**: Security testing enhancements:

1. **Authentication Testing**:
   - Invalid API keys
   - Expired tokens
   - Missing authentication headers
   - Privilege escalation tests

2. **Input Validation**:
   - SQL injection attempts
   - XSS payload testing
   - Invalid data type submissions
   - Boundary value testing

3. **Authorization Testing**:
   - Access control validation
   - Role-based testing
   - Resource access verification

```java
@Test
public void testInvalidApiKey() {
    // Temporarily modify API key
    String originalKey = propertyReader.getProperty("auth.apikey");
    // Set invalid key and verify 401/403 response
    Response response = petSteps.addPet(123, "TestPet", "available");
    Assert.assertEquals(response.getStatusCode(), 401);
}
```

---

15. Troubleshooting & Debugging
```

Q15.1: What common issues might occur and how to troubleshoot them?

Answer: Common issues and solutions:

Configuration Issues:

```java
// Problem: PropertyReader can't find file
// Solution: Verify classpath and file location
Exception: Properties file not found: PropertyFiles/Property.properties
// Check: File exists in src/test/resources/PropertyFiles/
```

Authentication Issues:

```java
// Problem: 401 Unauthorized responses
// Solution: Verify API key configuration
// Check: auth.apikey value in properties file
// Verify: Header name matches API expectation
```

Network Issues:

```java
// Problem: Connection timeout/refused
// Solution: Check base URL, network connectivity
// Add timeout configuration:
RestAssured.config = RestAssured.config()
   .httpClient(HttpClientConfig.httpClientConfig()
       .setParam("http.socket.timeout", 30000));
```

Q15.2: How would you debug test failures?

Answer: Debugging approach:

1. **Enable detailed logging**:

```java
// Add RestAssured logging
Response response = RestAssured.given()
   .log().all()  // Log request details
   .when()
```

```
   .get("/pet/123")
   .then()
   .log().all()  // Log response details
   .extract().response();
```

2. **Analyze test reports**:

  - ExtentReports for detailed test steps
  - TestNG reports for execution summary
  - Console logs for real-time debugging

3. **Add debug information**:

```java
System.out.println("Request URL: " + RestAssured.baseURI + endpoint);
System.out.println("Request Body: " + body);
System.out.println("Response: " + response.getBody().asString());
```

Q15.3: How would you handle environment-specific test failures?

Answer: Environment handling strategies:

1. **Environment detection**:

```java
String environment = System.getProperty("env", "test");
String propertiesFile = "PropertyFiles/Property-" + environment + ".properties";
```

2. **Conditional test execution**:

```java
@Test
public void testEnvironmentSpecificFeature() {
   String env = System.getProperty("env");
   if (!"prod".equals(env)) {
       // Execute test only in non-production environments
   } else {
       throw new SkipException("Test skipped in production environment");
   }
}
```

3. **Dynamic configuration**:

```java
// Override properties with environment variables
String baseUrl = System.getenv("API_BASE_URL");
if (baseUrl != null) {
   RestAssured.baseURI = baseUrl;
}
```

Conclusion

This comprehensive question bank covers all aspects of the Petstore API test automation framework, from basic concepts to advanced implementation details. The questions are structured to assess understanding at multiple levels:

- **Beginner Level**: Basic concepts, framework understanding, simple implementations
- **Intermediate Level**: Architecture, design patterns, best practices
- **Advanced Level**: Optimization, scaling, integration, troubleshooting

Each question includes detailed answers with code examples and practical insights that demonstrate deep understanding of the project and its underlying technologies.

Total Questions: 60+ comprehensive questions across 15 categories
Coverage: 100% of project codebase and concepts
Difficulty Levels: Beginner (40%), Intermediate (35%), Advanced (25%)

This document serves as a complete preparation guide for technical interviews, code reviews, and project understanding sessions.