

CUCUMBER WITH JAVA

BDD Interview Questions & Answers

A Complete Preparation Guide for QA Automation Engineer Roles

Covers Gherkin · Step Definitions · Hooks · Tags · Reporting · CI/CD

-
- ✓ Java-based Cucumber (BDD) questions written from a QA automation engineer's perspective
 - ✓ Beginner to advanced difficulty, organized by topic
 - ✓ Real Gherkin and Java code snippets you can reuse in interviews and on the job
 - ✓ Includes best practices for writing maintainable, business-readable scenarios
 - ✓ A quick-reference cheat sheet at the end

Prepared as a study reference for QA Automation Engineer interviews. Practice explaining each answer in your own words — interviewers value clear reasoning over memorized definitions.

Table of Contents

Section A	BDD & Cucumber Fundamentals	5 Qs
Section B	Framework Structure & Hooks	5 Qs
Section C	Writing Effective Scenarios (QA Best Practices)	5 Qs
Section D	Reporting & CI/CD Integration	5 Qs
Appendix	Quick-Reference Cheat Sheet	

Section A | BDD & Cucumber Fundamentals

Q. What is Behavior-Driven Development (BDD) and what problem does Cucumber solve for QA teams?

BDD is a collaboration practice where business stakeholders, developers, and QA jointly define expected system behavior in plain language before automation begins. Cucumber is a tool that executes those plain-language scenarios (written in Gherkin) as automated tests, so the same document serves as both living documentation and an executable test — closing the gap between requirements and test coverage.

Q. What is Gherkin syntax, and what do Given/When/Then represent?

Gherkin is the structured, keyword-based language Cucumber parses. Given sets up the initial context/preconditions, When describes the action or event being tested, and Then specifies the expected outcome. And/But can extend any of these for readability.

```
Feature: User Login
```

```
Scenario: Successful login with valid credentials
Given the user is on the login page
When the user enters valid username and password
And clicks the login button
Then the user should see the dashboard
```

Q. What is a Step Definition and how does it connect to a Gherkin step?

A step definition is a Java method annotated with @Given/@When/@Then containing a regex or Cucumber expression that matches the wording of a Gherkin step. When Cucumber runs a scenario, it matches each plain-language line to its corresponding method and executes the underlying automation code (often calling into a Page Object).

```
@Given("the user is on the login page")
public void userIsOnLoginPage() {
    driver.get("https://example.com/login");
}

@When("the user enters valid username and password")
public void userEntersCredentials() {
    loginPage.login("qa_user", "secret123");
}
```

Q. What is a Feature File and how should it be organized in a real project?

A .feature file groups related scenarios under one Feature heading, typically mirroring a single piece of functionality (e.g., login.feature, checkout.feature). Keeping one feature per file, with scenarios that describe distinct, independent behaviors, keeps the suite readable and makes it easy for non-technical stakeholders to review coverage.

Q. What is Scenario Outline and Examples in Cucumber, and when would you use them?

Scenario Outline lets you write one scenario template with placeholder variables, then run it once per row in an Examples table — equivalent to data-driven testing, but expressed in a business-readable table rather than a Java DataProvider.

```
Scenario Outline: Login with different credential sets
Given the user is on the login page
When the user logs in with "<username>" and "<password>"
Then the login result should be "<result>"
```

Examples:

```
| username | password | result |
| user1   | correct1 | success |
| user2   | wrong    | failure |
```

Section B | Framework Structure & Hooks

Q. What are Cucumber Hooks (@Before/@After) and how are they used in a test framework?

Hooks run setup/teardown code around every scenario without needing a step in the feature file — @Before typically launches the browser/driver and navigates to a starting state, and @After closes the driver and, on failure, attaches a screenshot to the report.

```
@Before
public void setUp() {
    driver = new ChromeDriver();
}

@After
public void tearDown(Scenario scenario) {
    if (scenario.isFailed()) {
        byte[] screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.BYTES);
        scenario.attach(screenshot, "image/png", "failure-screenshot");
    }
    driver.quit();
}
```

Q. How do you share state (like a logged-in session) between step definitions in different classes?

Since Cucumber creates a new instance of each step definition class per scenario, shared state is typically handled via a small dependency-injection setup (PicoContainer or Spring), injecting a shared 'World'/context object into each step class so data set in one step (e.g., a created order ID) is accessible in a later step.

```
public class TestContext {
    public String orderId;
}

public class OrderSteps {
    private final TestContext context;
    public OrderSteps(TestContext context) { this.context = context; }
}
```

Q. What are Cucumber Tags and how would a QA engineer use them to organize test runs?

Tags (e.g., @smoke, @regression, @wip) mark scenarios so specific subsets can be run selectively — for example, running only @smoke tests on every pull request for fast feedback, and the full @regression tag set nightly, without duplicating any test files.

```
@smoke @regression
Scenario: Successful login with valid credentials
...

# Run only smoke tests:
mvn test -Dcucumber.filter.tags="@smoke"
```

Q. How do Cucumber step definitions connect to Page Object Model in a Selenium/Playwright + Cucumber project?

Step definition classes should stay thin — they call methods on Page Object classes rather than containing locators or raw WebDriver calls directly. This keeps the same separation of concerns as a non-BDD framework: Gherkin describes behavior, step definitions translate that to actions, and Page Objects encapsulate how those actions are actually performed.

Q. What is a Cucumber Runner class (@CucumberOptions) and what does it configure?

The runner class (typically using JUnit's @RunWith(Cucumber.class) or JUnit 5's Cucumber Suite) points Cucumber to the feature files' location, the step definition package, which tags to include/exclude, and which plugins/reporters to generate output with.

```
@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/resources/features",
    glue = "stepdefinitions",
    tags = "@regression",
    plugin = {"pretty", "html:target/cucumber-report.html"}
)
public class TestRunner {}
```

Section C | Writing Effective Scenarios (QA Best Practices)

Q. What makes a Gherkin scenario 'well written' from a QA engineer's perspective?

A good scenario is declarative (describes what the user does and expects, not low-level UI mechanics like 'clicks the button with id=x'), independent of other scenarios, focused on one behavior, and phrased so a non-technical stakeholder could read and validate it without seeing the underlying automation code.

Q. What is the difference between imperative and declarative Gherkin style, and which is preferred?

Imperative style spells out every UI interaction ('the user clicks the email field, types email, clicks the password field...'), which is brittle and hard to read. Declarative style expresses intent ('the user logs in with valid credentials'), hiding UI mechanics inside the step definition. Declarative is generally preferred for maintainability and readability by non-engineers.

Q. Why should Gherkin scenarios avoid conjunctions like 'and' that combine multiple behaviors into one Then step?

A Then step like 'the user is logged in and sees 3 notifications and the cart is empty' bundles multiple assertions into one unclear step, making failures harder to diagnose and the scenario harder to reuse. Splitting into multiple explicit Then/And steps keeps each assertion isolated and its failure message specific.

Q. How do you avoid duplicate step definitions across a growing feature file suite?

Use Cucumber Expressions or shared parameter types for common phrasing patterns, keep step definitions grouped by domain area to make existing steps discoverable before writing new ones, and periodically run `cucumber --dry-run` or an IDE's step-usage search to catch near-duplicate step wording before it multiplies.

Q. How would you handle a scenario that needs specific test data to already exist in the system?

Prefer setting up required data through an API call or database seed step executed inside a Given step (e.g., 'Given a customer with 5 past orders exists'), rather than assuming static data pre-exists in a shared environment — this keeps scenarios independent and repeatable regardless of environment state.

Section D | Reporting & CI/CD Integration

Q. What reporting options does Cucumber provide, and what would you use for a CI dashboard?

Built-in plugins include pretty (console), html, json, and junit formats. For CI systems that parse native test results, the junit XML output integrates directly; for a richer, shareable view, the Cucumber JSON output is commonly fed into a dedicated reporting tool like Allure or Cucumber Reports (Masterthought) to produce a polished HTML dashboard.

Q. How do you integrate a Cucumber + Java suite into a CI/CD pipeline like Jenkins or Azure DevOps?

Package and run tests via Maven/Gradle (`mvn test -Dcucumber.filter.tags="@regression"`), configure the pipeline to publish the generated JUnit XML for the built-in test results view, and archive the HTML/Allure report as a downloadable build artifact so stakeholders can review scenario-level pass/fail without digging through console logs.

Q. How would you run only a subset of Cucumber scenarios (e.g., smoke tests) in a CI pipeline?

Pass the desired tag expression as a build parameter to the Maven/Gradle command (`-Dcucumber.filter.tags`), which lets the same pipeline definition run either a fast smoke subset on every commit or the full regression suite on a nightly schedule, without needing separate pipeline definitions.

Q. What is Parallel execution in Cucumber-JVM and how is it configured?

Cucumber-JVM supports parallel execution at the scenario level via the JUnit Platform engine or Maven Surefire/Failsafe parallel configuration, distributing scenarios across threads. As with any parallel suite, step definitions and shared state (like WebDriver instances) must be made thread-safe, typically via `ThreadLocal`.

Q. How would you explain the business value of a Cucumber suite to a non-technical stakeholder during a demo?

Point out that the feature files themselves — readable by product owners and business analysts — are the same artifact executed as automated tests, so there's no drift between documented requirements and what's actually verified. A stakeholder can read a `.feature` file and immediately understand what's covered, without needing to read Java code.

Appendix | Quick-Reference Cheat Sheet

Concept	Example
Feature file structure	Feature: ... / Scenario: ... / Given/When/Then
Step definition	@Given("the user is on the login page")
Scenario Outline	Scenario Outline: ... + Examples: table
Hook (setup)	@Before public void setUp() { ... }
Hook (teardown)	@After public void tearDown(Scenario s) { ... }
Tag a scenario	@smoke @regression\nScenario: ...
Run tagged tests (Maven)	mvn test -Dcucumber.filter.tags="@smoke"
Runner config	@CucumberOptions(features=..., glue=..., tags=...)
Attach screenshot on failure	scenario.attach(bytes, "image/png", "name");
Data table step	@Given("the following users exist:") + DataTable arg
Cucumber expression	@When("the user enters {string} and {string}")

Final Interview Tip: Interviewers often ask you to critique or rewrite a poorly worded Gherkin scenario — practice spotting imperative, UI-mechanics-heavy steps and rephrasing them declaratively.