🔥 Interview Questions & Answers for beginners

Download Now

Question 1:

Explain Software Testing in your own words

Answer:

1. Software testing ensures the product works not just technically, but practically in real-world situations.

2. It validates how the system behaves when real users interact — including mistakes and unexpected actions.

3. It helps identify risks before they impact users, saving both reputation and cost.

4. It builds trust by ensuring the system is reliable, stable, and consistent.

5. A true tester doesn't just verify — they challenge the system and think beyond requirements.

👉 **Example:** Login works with correct data, but fails when user enters special characters — testing catches this before users face it.

💡 **Testing is not about checking code — it is about protecting user experience.**

Question 2:

Can you give a real example of testing?

Answer:

1. Real testing focuses on complete user journeys, not just individual features.

2. It validates each step and ensures smooth transitions between them.

3. It checks both success and failure paths.

4. It ensures the final goal of the user is achieved.

5. It reflects real-world usage instead of theoretical flows.

👉 **Example:** E-commerce flow → Add to cart → Apply coupon → Payment → Order confirmation. If payment fails, entire journey fails.

💡 **Always test how users use the system — not how developers built it.**

Question:3

How is testing different from debugging?

Answer:

Testing	Debugging
1. Testing is the process of identifying defects in the system.	1. Debugging is the process of fixing the identified defects.
2. It focuses on validating system behavior from a user perspective.	2. It focuses on finding the root cause in the code.
3. It is performed by testers or QA team.	3. It is performed by developers.
4. It ensures the product works as expected.	4. It ensures the issue is resolved permanently.
5. It happens before the product reaches users.	5. It happens after a defect is found.

👉 **Example:** Tester finds login failure → Developer analyzes code → fixes issue → that fixing process is debugging.

💡 **Testing finds the problem, debugging removes it.**

Question 4:

When do you start testing in SDLC?

Answer:

1. Testing starts from requirement phase by identifying gaps.

2. During design, it validates system logic and flow.

3. During development, it prepares test scenarios.

4. During execution, it validates actual behavior.

5. Early testing reduces cost and prevents rework.

👉 **Example:** Requirement unclear → tester questions early → avoids future defects.

💡 **The earlier you test, the cheaper the bug.**

Question 5:

Do testers only find bugs?

Answer:

1. Testers identify defects, but their role is much broader.

2. They improve usability and user experience.

3. They challenge unclear requirements.

4. They identify risks early.

5. They help build a better product overall.

👉 **Example:** Suggesting better UI flow instead of just reporting issue.

💡 **A tester adds value, not just defects.**

Question 6:

What happens if testing is skipped?

Answer:

1. Bugs directly reach users.

2. User trust gets damaged.

3. Business losses can occur.

4. Fixing issues becomes expensive.

5. Product becomes unreliable.

👉 **Example:** Payment failure in production → revenue loss.

💡 **Skipping testing is not saving time — it is creating future problems.**

Question 7:

Can testing guarantee 100% quality?

Answer:

1. Not all scenarios can be tested.

2. Unknown edge cases always exist.

3. User behavior is unpredictable.

4. Testing reduces risk significantly.

5. It builds confidence, not perfection.

👉 **Example:** App works in testing, but crashes under real user load.

💡 **Testing builds confidence, not guarantees perfection.**

Question 8:

What is your approach when testing a new feature?

Answer:

1. Understand the purpose and user expectation.

2. Identify positive, negative, and edge cases.

3. Validate functionality step by step.

4. Check usability and performance.

5. Think about real-world usage and failures.

👉 **Example:** Login feature → valid login, invalid login, empty fields, multiple clicks.

💡 **Don't just test what is written — test what can go wrong.**

Question 9:

Why do we need testing if developers already test their code?

Answer:

1. Developers test code correctness, but testers validate real-world behavior.

2. Developers focus on "how it should work", testers explore "how it can fail".

3. Testing brings an independent perspective, removing bias.

4. It ensures system works under unexpected conditions.

5. It validates usability, reliability, and user experience.

👉 **Example:** Developer tests successful login, but tester checks invalid inputs, multiple clicks, and slow network.

💡 **Developers make it work. Testers make it reliable.**

Question 10:

What is the difference between developer testing and QA testing?

Answer:

Developer Testing	QA Testing
1. Focus on code correctness	1. Focus on user experience
2. Tests expected scenarios	2. Tests edge & negative cases
3. Limited to functionality	3. Covers real-world behavior
4. Done during development	4. Done independently

👉 **Example:** Developer checks feature works. Tester checks how it breaks.

💡 **Developers verify logic, testers validate reality.**

Question 11:

Can developers miss bugs? Why?

Answer:

1. Developers are closely connected to their code, which creates bias.
2. They assume ideal usage instead of unpredictable user behavior.
3. Focus is on building, not breaking the system.
4. Edge cases and unusual scenarios may be overlooked.
5. Familiarity makes defects harder to notice.

👉 **Example:** Developer tests valid input flow, but misses invalid or unexpected inputs.

💡 **Familiarity hides defects.**

Question 12:

Have you seen a case where developer testing was not enough?

Answer:

1. Developer testing ensures functionality works in ideal scenarios.
2. It often misses real-world conditions like network issues.
3. User behavior is unpredictable and not always considered.
4. System may fail under edge or unusual conditions.
5. QA testing fills these real-world gaps.

👉 **Example:** Payment works normally, but fails when internet disconnects during transaction.

💡 **Real-world conditions expose real bugs.**

Question 13:

Who is responsible for quality — developer or tester?

Answer:

Developer	Tester
1. Builds the product	1. Validates product behavior
2. Ensures correct implementation	2. Ensures usability & reliability
3. Writes efficient code	3. Challenges real-world scenarios

👉 **Example:** Developer builds feature, tester ensures user can trust it.

💡 **Quality is a team responsibility, not a role.**

Question 14:

What kind of bugs developers usually miss?

Answer:

1. Edge cases and boundary conditions.

2. Usability and user experience issues.

3. Unexpected user behavior scenarios.

4. Performance under stress conditions.

5. UI inconsistencies and interaction issues.

👉 **Example:** Double click causing duplicate actions or empty input causing crash.

💡 **Bugs live where expectations end.**

Question 15:

Can testing be completely handled by developers?

Answer:

1. Developers focus on building, testers focus on breaking.

2. Independent perspective is required for unbiased validation.

3. Developers may miss real-world scenarios.

4. Testing requires different mindset and approach.

5. Both roles complement each other but cannot replace each other.

👉 **Example:** Developer ensures feature works, tester ensures it works in all conditions.

💡 **Independence creates objectivity.**

Question 16:

What is independent testing and why is it important?

Answer:

1. Independent testing is done by someone not involved in development.

2. It removes bias and assumptions.

3. It brings fresh perspective to identify hidden issues.

4. It improves test coverage and quality.

5. It ensures unbiased validation of the product.

👉 **Example:** QA team testing feature built by developers and finding gaps missed earlier.

💡 **Fresh eyes find hidden problems.**

Question 17:

What makes a good QA tester?

Answer:

1. A good tester focuses on mindset rather than tools.

2. They question functionality instead of blindly accepting it.

3. They think like a real user and explore unexpected behavior.

4. They identify risks beyond obvious scenarios.

5. They focus on what can fail, not just what works.

👉 **Example:** Instead of checking button click, tester checks rapid clicks, invalid inputs, and unusual behavior.

💡 **Great testers don't follow steps — they challenge them.**

Question 18:

What skills do you have as a QA tester?

Answer:

1. Strong analytical thinking to understand system behavior.

2. Attention to detail to identify small but critical issues.

3. Communication skills to clearly explain defects.

4. Risk-based thinking to prioritize important areas.

5. User-focused mindset to validate real-world usage.

👉 **Example:** Identifying issues like incorrect error messages or confusing navigation.

💡 **Skills help you test, mindset helps you stand out.**

Question 19:

How do you improve your testing skills?

Answer:

1. By working on real applications and understanding behavior.

2. By analyzing defects and learning root causes.

3. By studying different testing techniques.

4. By observing user behavior patterns.

5. By continuously learning from mistakes.

👉 **Example:** Finding usability gaps in real apps and improving testing approach.

💡 **Every bug teaches something new.**

Question 20:

How do you think differently from a developer?

Answer:

Developer Thinking	Tester Thinking
1. Focus on building functionality	1. Focus on breaking system
2. Assumes correct usage	2. Assumes unpredictable usage
3. Validates logic	3. Validates behavior
4. Thinks how it should work	4. Thinks how it can fail

👉 **Example:** Developer checks success flow, tester checks failure scenarios.

💡 **Developers build systems, testers test reality.**

Question 21:

What is your biggest strength as a tester?

Answer:

1. Ability to think from user perspective.

2. Identifying hidden risks in simple features.

3. Exploring beyond defined requirements.

4. Asking the right questions.

5. Focusing on real-world scenarios.

👉 **Example:** Detecting issues when user enters unexpected data.

💡 **Thinking like a user reveals real defects.**

Question 22:

How do you handle pressure in testing?

Answer:

1. By focusing on critical functionalities first.

2. By prioritizing based on risk and impact.

3. By avoiding unnecessary testing.

4. By staying calm and structured.

5. By delivering maximum value in limited time.

👉 **Example:** Testing login and payment first under tight deadlines.

💡 **Smart prioritization reduces pressure.**

Question 23:

What do you do if you don’t understand a feature?

Answer:

1. Ask questions to stakeholders.

2. Review requirements and documentation.

3. Explore similar features.

4. Clarify assumptions before testing.

5. Ensure complete understanding before execution.

👉 **Example:** Discussing feature with BA before writing test cases.

💡 **Clarity prevents incorrect testing.**

Question 24:

How do you ensure you don’t miss bugs?

Answer:

1. Test from multiple perspectives.

2. Cover positive, negative, and edge cases.

3. Think about real user behavior.

4. Review test scenarios thoroughly.

5. Continuously improve testing approach.

👉 **Example:** Testing login with valid, invalid, empty, and boundary inputs.

💡 **Broader thinking leads to better coverage.**

Question 25:

Difference between “testing” and “quality”

Answer:

Testing	Quality
1. It is an activity performed during development lifecycle.	1. It is the final outcome experienced by users.
2. Focus is on identifying defects.	2. Focus is on user satisfaction and reliability.
3. It verifies system behavior against requirements.	3. It is built across the entire lifecycle.
4. It is limited to validation process.	4. It reflects real-world usability.

👉 **Example:** A feature passes all test cases but still confuses users — it is tested but not high quality.

💡 **Testing checks quality, but users define it.**

Question 26:

Can we achieve 100% quality through testing?

Answer:

1. It is impossible to test every possible scenario.
2. Real-world user behavior is unpredictable.
3. Unknown edge cases can still occur.
4. Testing reduces risk but does not eliminate it.
5. It builds confidence, not perfection.

👉 **Example:** Application works in testing but fails under unexpected user behavior.

💡 **Testing improves quality, it doesn’t guarantee it.**

Question 27:

Who is responsible for quality in a team?

Answer:

Individual Roles	Reality
1. Developers write code.	1. Quality is owned by the entire team.
2. Testers validate behavior.	2. It requires collaboration.

3. Designers handle user experience.

3. Every role contributes to final outcome.

👉 **Example:** A production bug reflects team responsibility, not just QA.

💡 **Quality is a shared responsibility.**

Question 28:

How do you measure quality?

Answer:

1. By analyzing defect count and severity.

2. By checking system stability and performance.

3. By evaluating user satisfaction and feedback.

4. By monitoring production issues.

5. By observing real user interaction.

👉 **Example:** Low defects + smooth performance + happy users indicate high quality.

💡 **Users are the real quality metric.**

Question 29:

Is testing enough to ensure quality?

Answer:

1. Testing alone cannot ensure quality.

2. Quality starts from requirements and design.

3. Poor design cannot be fixed by testing.

4. Testing only identifies defects, not builds quality.

5. Quality must be built throughout development.

👉 **Example:** A confusing UI passes tests but still frustrates users.

💡 **Quality is built, not just tested.**

Question 30:

What are examples of poor quality but tested software?

Answer:

1. Software that works but is slow.

2. Systems with confusing user interface.

3. Features that meet requirements but lack usability.

4. Applications that pass tests but frustrate users.

5. Products that are correct but not convenient.

👉 **Example:** A slow-loading app that works correctly but annoys users.

💡 **Working software is not always good software.**

Question 31:

How does QA improve quality before testing?

Answer:

1. By reviewing requirements early.

2. By identifying gaps and ambiguities.

3. By suggesting improvements before development.

4. By preventing defects instead of detecting them.

5. By ensuring clarity in system behavior.

👉 **Example:** Suggesting validation rules before development avoids future bugs.

💡 **Early QA prevents future defects.**

Question 32:

What is the role of process in quality?

Answer:

1. Process ensures consistency in development and testing.

2. It reduces human errors.

3. It improves predictability of results.

4. It standardizes workflows.

5. It makes quality repeatable and reliable.

👉 **Example:** Following SDLC process reduces defects and improves delivery.

💡 **Strong process builds strong quality.**

Question 33:

Have you ever found a bug in real life?

Answer:

1. Yes, I encountered a critical bug in an e-commerce checkout flow.

2. Payment was successful but order was not created.

3. This created a gap between system response and user expectation.

4. It directly impacted user trust and business revenue.

5. Such bugs are not technical only — they are business critical.

👉 **Example:** Payment deducted but no order confirmation received.

💡 **Real bugs are those that damage user trust.**

Question 34:

Explain Payment deducted but no order confirmation received bug in detail

Answer:

1. User initiates payment and transaction is successfully completed at payment gateway.

2. System receives payment success response but fails to create order in backend.

3. User account shows no order while money is deducted.

4. This creates inconsistency between payment system and application.

5. It results in confusion, frustration, and loss of trust.

👉 **Example:** Bank shows successful transaction but order history is empty.

💡 **Broken flow hurts more than broken feature.**

Question 35:

What was the expected vs actual behavior of Payment gateway bug?

Answer:

Expected Behavior	Actual Behavior
1. Payment success	1. Payment success
2. Order created	2. Order not created
3. Confirmation shown	3. No confirmation shown
4. Order visible in history	4. Order missing

👉 **Example:** User paid successfully but system failed to complete order flow.

💡 **Expected builds trust, actual reveals gaps.**

Question 36:

How would you report that bug?

Answer:

1. Write clear steps to reproduce the issue.

2. Include test data and environment details.

3. Attach screenshots or logs.

4. Mention expected vs actual behavior clearly.

5. Highlight business impact for better prioritization.

👉 **Example:** Steps: Add item → Checkout → Pay → Observe missing order.

💡 **A clear bug report speeds up resolution.**

Question 37:

What would be the severity and priority of Payment gateway bug??

Answer:

Severity	Priority
1. Critical	1. High
2. Financial impact	2. Immediate fix required
3. Data inconsistency	3. Direct user impact
4. System failure	4. Business risk involved

👉 **Example:** Payment issue affecting real users needs urgent fix.

💡 **Critical bugs demand immediate attention.**

Question 38:

How would you reproduce that Payment gateway bug??

Answer:

1. Perform checkout with valid payment details.

2. Simulate network delay or interruption.

3. Retry payment under different conditions.

4. Observe order creation after payment success.

5. Repeat steps to confirm consistency.

👉 **Example:** Simulating slow network during payment processing.

💡 **Reproducible bugs are fixable bugs.**

Question 39:

What could be the root cause Payment gateway bug??

Answer:

1. Failure in communication between payment gateway and backend.

2. Order creation API failure.

3. Timeout during transaction processing.

4. Missing retry or rollback mechanism.

5. Database inconsistency.

👉 **Example:** Payment success callback received but order API failed.

💡 **Root cause analysis leads to real solution.**

Question 40:

How would you test Payment gateway bug further?

Answer:

1. Perform regression testing on checkout flow.

2. Test multiple payment methods.

3. Validate retry and failure scenarios.

4. Test under different environments.

5. Ensure fix works consistently.

👉 **Example:** Testing UPI, card, wallet, and failure cases.

💡 **Fix is complete only when confidence returns.**

Question 41:

What other areas would you test after finding that Payment gateway bug?

Answer:

1. Order history module.

2. Payment logs and transaction records.

3. Refund and reversal mechanism.

4. Notification system.

5. Integration with external services.

👉 **Example:** Checking if refund triggered when order failed.

💡 **One bug can reveal multiple hidden risks.**

Question 42:

If you get incomplete requirements, what will you do?

Answer:

1. First, I pause execution instead of blindly starting testing.

2. I analyze what is missing in terms of inputs, outputs, and validations.

3. I avoid assumptions and try to understand the actual intent.

4. I prepare structured questions before approaching stakeholders.

5. I ensure clarity before writing or executing test cases.

👉 **Example:** If password rules are missing, I don't test randomly — I first clarify expected behavior.

💡 **Testing without clarity is not testing — it's guessing.**

Question 43:

How do you identify that the requirements are incomplete?

Answer:

1. When expected behavior is not clearly defined.

2. When inputs or validations are missing.

3. When I am unable to write clear test scenarios.

4. When edge cases are not covered.

5. When different interpretations are possible.

👉 **Example:** Login requirement says "user should login" but doesn't define error handling.

💡 **If you can't test it clearly, it's not clearly defined.**

Question 44:

What steps do you take before raising a question that requirements is incomplete?

Answer:

1. I thoroughly review requirement documents.

2. I check similar features or previous implementations.

3. I analyze logically what the system should do.

4. I prepare clear and structured questions.

5. I avoid asking vague or unprepared questions.

👉 **Example:** Reviewing previous module behavior before asking about new validation logic.

💡 **Good questions come from preparation, not confusion.**

Question 45:

Whom will you approach first — BA, developer, or client when the requirements are incomplete ?

Answer:

I Will coonect BA First due to the following reasons

BA (Business Analyst)	Developer / Client
1. First point of contact	1. Developer for technical clarification
2. Understands business requirements	2. Client for business confirmation
3. Provides requirement clarity	3. Used when deeper clarity is needed
4. Helps resolve ambiguity	4. Helps finalize understanding

👉 **Example:** Clarifying validation rules with BA before discussing implementation with developer.

💡 **Ask the right person to get the right answer.**

Question 46:

What if no one is available to clarify that the requirements are incomplete?

Answer:

1. I document assumptions clearly before proceeding.

2. I perform exploratory testing cautiously.

3. I mark areas of uncertainty in test cases.

4. I avoid final conclusions based on assumptions.

5. I validate once stakeholders are available.

👉 **Example:** Writing “Assumed behavior” in test case comments.

💡 **Assumptions are temporary — clarity is permanent.**

Question 47:

Will you start testing with assumptions when the requirements are incomplete?

Answer:

1. Yes, but only for exploratory understanding.

2. I clearly document all assumptions.

3. I avoid marking results as final.

4. I validate assumptions later.

5. I treat it as temporary learning, not final testing.

👉 **Example:** Testing login with assumed validation rules before confirmation.

💡 **Assumptions guide exploration, not decisions.**

Question 48:

How do you document missing requirements?

Answer:

1. I clearly highlight missing areas in documentation.

2. I maintain a list of open questions.

3. I document assumptions separately.

4. I add comments in test cases.

5. I ensure visibility to stakeholders.

👉 **Example:** “Validation rule not defined” added in test case notes.

💡 **Documentation turns confusion into clarity.**

Question 49:

What risks are involved if you proceed without clarity of the requirements ?

Answer:

1. Incorrect testing scenarios.

2. Missing critical defects.

3. Wrong assumptions.

4. Rework and delays.

5. Poor user experience in production.

👉 **Example:** Missing validation leads to production bug.

💡 **Lack of clarity creates hidden failures.**

Question 50:

Can you give a real example where requirement was unclear?

Answer:

1. I worked on a form where validation rules were not defined.

2. Different fields behaved inconsistently.

3. Some accepted invalid data while others rejected it.

4. I raised the issue and clarified requirements.

5. Proper validation rules were implemented later.

👉 **Example:** Email field accepting invalid formats due to missing requirement.

💡 **Unclear requirements create inconsistent systems.**

Question 51:

How do you ensure you understood the requirement correctly?

Answer:

1. Break requirement into clear scenarios and flows.

2. Validate each scenario logically before testing.

3. Ensure there is no ambiguity in expected behavior.

4. Confirm that test cases can be written confidently.

5. Cross-check understanding with stakeholders if needed.

👉 **Example:** Converting requirement into scenarios and verifying all paths.

💡 If you can explain it clearly, you understand it correctly.

Anand Hooda

Question 52:

Do you document your understanding? If yes, how?

Answer:

1. Document understanding using test scenarios and notes.

2. Capture assumptions clearly.

3. Add comments in test cases.

4. Maintain structured documentation.

5. Share with stakeholders for alignment.

👉 **Example:** Writing expected behavior and assumptions in test cases.

💡 Writing clarifies thinking.

Question 53:

What is your approach to requirement validation?

Answer:

1. Analyze requirement from functional perspective.

2. Identify edge cases and boundary conditions.

3. Check logical consistency.

4. Validate completeness.

5. Ensure requirement is testable.

👉 **Example:** Checking if validations and error handling are defined.

💡 Good validation prevents future defects.

Question 54:

Do you create test cases immediately or after confirmation?

Answer:

1. I do not create final test cases immediately — I first ensure requirement clarity.

2. I may create rough scenarios early to understand the flow and identify gaps.

3. But I finalize test cases only after confirming requirements with stakeholders.

4. This helps avoid rework and ensures accuracy.

5. It ensures my testing aligns with actual business expectations.

👉 **Example:** I first draft scenarios for a login feature, but finalize test cases only after confirming validation rules.

💡 **Draft early, finalize after clarity — that's a smart tester's approach.**

Question 55:

How do you confirm your understanding with stakeholders?

Answer:

1. Discuss scenarios with stakeholders.

2. Ask targeted questions.

3. Validate expected behavior.

4. Share understanding for confirmation.

5. Align before execution.

👉 **Example:** Sharing test scenarios with BA.

💡 **Communication removes confusion.**

Question 56:

What if your understanding is different from the developer?

Answer:

1. Discuss differences openly.

2. Refer requirement and business logic.

3. Align based on correct interpretation.

4. Avoid assumptions.

5. Confirm final understanding.

👉 **Example:** Validation expected by tester but missed by developer → resolved via discussion.

💡 **Alignment is more important than assumption.**

Question 57:

Do you use any techniques like scenarios or flow diagrams?

Answer:

1. Use scenarios to understand behavior.

2. Use flow diagrams to visualize system.

3. Identify missing paths.

4. Validate complete journey.

5. Improve clarity through visualization.

👉 **Example:** Drawing login flow with success and failure paths.

💡 **Visualization improves understanding.**

Question 58:

How do you avoid misinterpretation?

Answer:

1. Validate assumptions.

2. Ask questions proactively.

3. Confirm scenarios with stakeholders.

4. Avoid guess-based testing.

5. Ensure clarity before execution.

👉 **Example:** Confirming edge case behavior before testing.

💡 **Validate before you assume.**

Question 59:

Can you explain a situation where you misunderstood a requirement?

Answer:

1. Assumed validation rules without confirmation.

2. Created test cases based on assumptions.

3. Later found mismatch in expected behavior.

4. Required rework.

5. Learned importance of validation.

👉 **Example:** Assuming mandatory fields without confirmation.

💡 **Assumption without validation leads to rework.**

Question 60:

What questions will you ask before starting testing?

Answer:

1. What is the purpose of this feature?

2. What is the expected behavior?

3. What are the validations and rules?

4. What are the edge cases and failure scenarios?

5. What is the user flow and business impact?

👉 **Example:** What should happen if user enters invalid input?

💡 **Right questions define right testing.**

Question 61:

What are the most important questions you ask?

Answer:

1. What is the expected outcome?

2. What are the failure scenarios?

3. What are the boundaries and limits?

4. What happens in edge cases?

5. What is the business impact?

👉 **Example:** What should happen if payment fails after deduction?

💡 **Important questions reveal hidden requirements.**

Question62:

Do you focus more on functional or non-functional aspects?

Answer:

1. My primary focus is on functional aspects because they ensure the core behavior of the application works correctly.

2. If functionality itself is broken, non-functional testing has no value.

3. Once functional stability is achieved, I shift focus to non-functional aspects like performance, usability, and reliability.

4. I follow a risk-based approach — critical business flows always get priority first.

5. Ultimately, I balance both, but functional testing comes first, followed by non-functional validation.

👉 **Example:** First I ensure login works correctly (functional), then I check how fast it responds and how smooth the user experience is (non-functional).

💡 **First make it work, then make it better — that's a smart testing approach.**

Question 63:

How do you prioritize questions?

Answer:

1. Based on risk and impact.

2. Focus on critical business flows.

3. Prioritize unclear requirements.

4. Consider user impact.

5. Ignore low-impact questions initially.

👉 **Example:** Payment flow questions prioritized over UI styling.

💡 **Prioritize what impacts users most.**

Question 64:

What if stakeholders get irritated with too many questions?

Answer:

1. Ask only relevant and meaningful questions.

2. Prepare questions in structured format.

3. Avoid repetitive or random questions.

4. Group related questions together.

5. Respect stakeholder time.

👉 **Example:** Sending a well-organized list instead of asking randomly.

💡 **Quality of questions matters more than quantity.**

Question65:

Do you ask questions in meetings or offline?

Answer:

1. I use both approaches based on the situation and type of question.

2. For quick clarifications or discussions that need immediate alignment, I prefer asking in meetings.

3. For detailed, complex, or multiple questions, I prefer asking offline in a structured format.

4. This ensures clarity without interrupting flow or overwhelming stakeholders.

5. My goal is not just to ask questions, but to get clear and well-documented answers.

👉 **Example:** I ask simple doubts in meetings, but send detailed requirement questions via email or Jira.

💡 **Smart testers choose the right time and channel — not just ask questions.**

Question 66:

How do you track answers given by stakeholders?

Answer:

1. Document answers in test cases.

2. Maintain notes or tracking sheet.

3. Update requirement documents.

4. Use tools like Jira or Confluence.

5. Ensure visibility for team.

👉 **Example:** Updating confirmed behavior in test cases.

💡 **Documentation ensures consistency.**

Question 67:

Can you give examples of critical questions you asked in a project?

Answer:

1. What happens if payment fails after deduction?

2. What is retry behavior?

3. How are edge cases handled?

4. What is expected user behavior?

5. What are system limitations?

👉 **Example:** Asking about payment failure scenarios prevented major bug.

💡 **Critical questions prevent critical bugs.**

Question68:

What happens if you don’t ask enough questions?

Answer:

1. Lack of questions leads to assumptions instead of clarity.

2. Important scenarios and edge cases are easily missed.

3. Testing may be done on wrong understanding of requirements.

4. This results in defects reaching production.

5. It ultimately impacts user experience and increases rework for the team.

👉 **Example:** If validation rules are not clarified, tester may miss negative scenarios, leading to production defects.

💡 **Not asking questions doesn't save time — it creates bigger problems later.**

Question 69:

What if client changes requirement at the last moment?

Answer:

1. I stay calm and avoid reacting emotionally to last-minute changes.

2. I first analyze what exactly has changed and what areas are impacted.

3. I focus on high-risk and critical functionalities first.

4. I adjust testing strategy based on time and impact.

5. I communicate clearly with stakeholders about risks and coverage.

👉 **Example:** Payment flow changed before release → focus on validation + regression of critical paths.

💡 **Calm analysis is more powerful than panic reaction.**

Question 70:

How do you handle re-testing in limited time?

Answer:

1. I prioritize high-impact and high-risk areas first.

2. I avoid trying to test everything blindly.

3. I focus on core functionalities and affected modules.

4. I apply risk-based testing approach.

5. I ensure maximum coverage in minimum time.

👉 **Example:** Retesting login, payment, and checkout instead of full regression.

💡 **In limited time, smart testing beats complete testing.**

Question 71:

What will you prioritize first?

Answer:

1. Critical business flows that directly impact users.

2. Recently changed or modified modules.

3. High-risk areas where failure is costly.

4. Core functionalities like login, payment, checkout.

5. Areas with maximum user interaction.

👉 **Example:** Payment and checkout prioritized over UI styling.

💡 **Priority should always follow business impact.**

Question72:

Will you update test cases immediately after requirement change?

Answer:

1. Yes, I update test cases after requirement changes, but only after confirming the updated requirement clearly.

2. I first understand what exactly has changed and what areas are impacted.

3. Then I update only the relevant test cases instead of modifying everything blindly.

4. This ensures accuracy and avoids unnecessary rework.

5. It also keeps test cases aligned with the latest business expectations.

👉 **Example:** If validation rules change in a form, I update only the related test cases after confirming the new rules.

💡 **Don't update blindly — update with clarity and impact awareness.**

Question 73:

How do you manage regression impact?

Answer:

1. I identify impacted modules based on the change.

2. I focus regression only on related areas.

3. I avoid unnecessary full regression.

4. I validate integration points.

5. I ensure existing functionality remains stable.

👉 **Example:** Payment change → test order, refund, and transaction flow.

💡 **Smart regression is targeted, not exhaustive.**

Question 74:

What if change breaks existing functionality?

Answer:

1. I report the issue immediately with proper impact analysis.

2. I highlight risk to stakeholders.

3. I ensure defect is prioritized correctly.

4. I validate related areas for further impact.

5. I ensure fix does not introduce new issues.

👉 **Example:** Login update breaks forgot password functionality.

💡 **A new change should not break existing stability.**

Question 75:

How do you communicate risk to stakeholders?

Answer:

1. I communicate risks clearly and without ambiguity.

2. I explain impact on users and business.

3. I mention testing coverage and limitations.

4. I avoid technical jargon and keep it simple.

5. I ensure stakeholders can make informed decisions.

👉 **Example:** “Limited testing done, risk remains in edge cases.”

💡 **Clear communication builds trust and confidence.**

Question 76:

Have you faced last-minute requirement changes in real life?

Answer:

1. Yes, I have faced last-minute changes in real projects.

2. I stayed calm and avoided panic.

3. I prioritized critical areas first.

4. I adjusted testing strategy based on time.

5. I ensured quality was maintained despite pressure.

👉 **Example:** UI change before release → focused testing on impacted screens.

💡 **Real experience builds real confidence.**

Question77:

How do you avoid panic in such situations when last-minute requirement changes in real life?

Answer:

1. I stay calm and avoid reacting emotionally, because panic leads to poor decisions.

2. I break the situation into smaller, manageable tasks.

3. I quickly identify what has changed and what areas are impacted.

4. I prioritize critical functionalities instead of trying to do everything at once.

5. I follow a structured approach and focus on one step at a time.

👉 **Example:** If a last-minute change happens in payment flow, I list impacted areas first and test them step by step instead of testing randomly.

💡 **Panic creates confusion, but structured thinking creates solutions.**

Question 78:

How do you identify missing requirements, what techniques do you use to find gaps?

Answer:

1. I break the feature into multiple scenarios and flows.

2. I question each step to identify undefined or unclear behavior.

3. I use techniques like scenario analysis, boundary thinking, and user journey mapping.

4. I check for missing validations, edge cases, and alternate flows.

5. I focus on what is not written as much as what is written.

👉 **Example:** Login requirement exists but validation and error handling are not defined.

💡 **Gaps are not visible — they are discovered through structured thinking.**

Question 79:

Do you compare missing requirements with similar systems or past projects?

Answer:

1. Yes, I use past experience and similar systems as a reference point.

2. It helps identify common patterns and expected behaviors.

3. I compare flows, validations, and user journeys.

4. This helps in quickly identifying missing scenarios.

5. Experience makes gap identification faster and more accurate.

👉 **Example:** Comparing checkout flow with previous e-commerce systems to find missing validations.

💡 **Experience helps you see what others miss.**

Question 80:

How do you validate business logic completeness?

Answer:

1. I map requirements to real-world business scenarios.

2. I check if all conditions and outcomes are defined.

3. I validate logic for different user types and situations.

4. I ensure all possible paths are covered.

5. If a scenario cannot be logically explained, I treat it as incomplete.

👉 **Example:** Verifying discount logic for different users, conditions, and edge cases.

💡 **If logic fails in real life, requirement is incomplete.**

Question 81:

What is your approach to edge cases?

Answer:

1. I actively think beyond normal or happy path scenarios.

2. I test boundary conditions like minimum, maximum, and limits.

3. I validate unusual and unexpected inputs.

4. I explore negative scenarios and failure conditions.

5. I focus on areas where system behavior can break.

👉 **Example:** Testing empty input, maximum values, and invalid formats.

💡 **Real bugs live in edge cases, not in happy paths.**

Question 82:

How do you involve stakeholders in gap analysis?

Answer:

1. I discuss scenarios and flows with stakeholders.

2. I ask targeted and structured questions.

3. I validate assumptions before proceeding.

4. I share identified gaps for confirmation.

5. I ensure alignment before finalizing testing approach.

👉 **Example:** Reviewing requirement scenarios with BA before writing test cases.

💡 **Collaboration turns gaps into clarity.**

Question 83:

Can you give a real example where you found a missing requirement?

Answer:

1. I encountered a form where validation rules were not defined.

2. During testing, inconsistent behavior was observed.

3. Some fields accepted invalid inputs while others rejected them.

4. I raised the issue and clarified requirements.

5. Proper validation rules were implemented later.

👉 **Example:** Email field accepting invalid formats due to missing requirement.

💡 **Missing requirements create inconsistent systems.**

Question 84:

What happens if missing requirement is found late?

Answer:

1. It leads to rework and delays in delivery.

2. It increases project cost and effort.

3. It impacts timelines and release schedules.

4. It increases risk of production defects.

5. Fixing late-stage issues is always more complex.

👉 **Example:** Adding validation after development causes redesign and retesting.

💡 **Late clarity is always expensive.**

Question 85:

How do you document and track missing requirements?

Answer:

1. I document missing requirements as gaps, assumptions, and open questions.

2. I use tools like Jira, test cases, or requirement documents.

3. I clearly mention impact and affected areas.

4. I ensure visibility for stakeholders.

5. I track them until they are resolved.

👉 **Example:** Logging missing requirement in Jira with clear description and impact.

💡 **Documented gaps become actionable tasks.**

Question 86:

How do you write a test case for a login page?

Answer:

1. I start by understanding the complete user flow of login.

2. I identify inputs, validations, and expected outcomes.

3. I design positive, negative, and edge test scenarios.

4. I define clear steps, test data, and expected results.

5. I ensure test cases are easy to understand and executable.

👉 **Example:** Enter valid credentials → Click login → User should be redirected to dashboard.

💡 **A good test case reflects real user behavior, not just system logic.**

Question 87:

What fields will you consider for login page testing?

Answer:

1. Username or email field.

2. Password field and masking behavior.

3. Login button functionality.

4. Error messages and validation.

5. Additional links like “Forgot Password”.

👉 **Example:** Email format validation and password masking.

💡 **Every field is a potential failure point.**

Question 88:

How will you write negative test cases for login page?

Answer:

1. I test invalid credentials combinations.

2. I check empty and null inputs.

3. I validate special characters and invalid formats.

4. I test boundary conditions.

5. I ensure system handles errors gracefully.

👉 **Example:** Enter wrong password → System should show proper error message.

💡 **Negative testing reveals system weaknesses.**

Question 89:

How do you handle validation messages for login page?

Answer:

1. I ensure messages are clear and user-friendly.

2. I avoid technical or confusing language.

3. I verify message consistency and placement.

4. I check timing of message display.

5. I ensure messages guide user correctly.

👉 **Example:** “Invalid credentials” instead of “Error 401”.

💡 **Good messages improve user experience.**

Question 90:

Will you include security test cases for login page?

Answer:

1. Yes, security is critical for login functionality.

2. I test SQL injection and malicious inputs.

3. I validate password encryption and storage.

4. I check brute force protection.

5. I ensure no sensitive data is exposed.

👉 **Example:** Enter SQL query → System should block it.

💡 **Security is essential, not optional.**

Question 91:

How do you test login with invalid credentials?

Answer:

1. I test wrong username and password combinations.

2. I verify system rejects invalid inputs consistently.

3. I check proper error messages.

4. I ensure no sensitive information is exposed.

5. I validate system stability under invalid attempts.

👉 **Example:** Wrong username + correct password → Login should fail.

💡 **Controlled failure defines system reliability.**

Question 92:

What happens if user clicks login multiple times?

Answer:

1. I verify system handles multiple clicks gracefully.

2. I check duplicate requests are not created.

3. I validate no multiple sessions are generated.

4. I ensure button is disabled or controlled.

5. I confirm system stability under repeated actions.

👉 **Example:** Multiple clicks trigger only one login request.

💡 **Systems should handle human impatience.**

Question 93:

How do you test login session handling?

Answer:

1. I verify session creation after login.

2. I check session timeout and expiry.

3. I validate logout functionality.

4. I ensure session invalidation after logout.

5. I test security of session management.

👉 **Example:** Session expires after inactivity.

💡 **Session control ensures security and usability.**

Question 94:

Do you consider performance or load scenarios for login page?

Answer:

1. Yes, login performance is critical.

2. I test system under multiple user load.

3. I validate response time.

4. I check system stability under load.

5. I ensure smooth user experience.

👉 **Example:** 1000 users logging in simultaneously.

💡 **First impression matters — login must be fast.**

Question 95:

How do you test login with different browsers/devices?

Answer:

1. I test login across multiple browsers.

2. I validate functionality on different devices.

3. I check UI consistency.

4. I ensure performance remains stable.

5. I confirm consistent user experience.

👉 **Example:** Login works on desktop but fails on mobile.

💡 **Compatibility ensures usability across platforms.**

Question 96:

Can you explain your test case structure?

Answer:

1. I define clear test steps.

2. I specify input data.

3. I define expected results.

4. I record actual results during execution.

5. I ensure test case is clear and reusable.

👉 **Example:** Step: Enter valid credentials → Expected: Login success.

💡 **Clear structure ensures consistent testing.**

Question 97:

What makes a test case effective?

Answer:

1. Clearly defines what needs to be tested and why.

2. Focuses on real user behavior and meaningful scenarios.

3. Covers positive, negative, and edge cases.

4. Is simple, structured, and easy to execute.

5. Provides clear expected outcomes to avoid ambiguity.

👉 **Example:** Login test case covering valid, invalid, and edge inputs.

💡 **A good test case doesn't just test — it guides thinking.**

Question 98:

What elements must every test case contain?

Answer:

1. Clear test steps.

2. Defined input data.

3. Expected result.

4. Actual result (during execution).

5. Status (Pass/Fail).

👉 **Example:** Step → Input → Expected Result → Actual Result → Status.

💡 **Structure brings consistency, clarity brings confidence.**

Question 99:

How do you ensure clarity in test cases?

Answer:

1. Use simple and precise language.

2. Write clear step-by-step instructions.

3. Avoid vague or ambiguous terms.

4. Define expected results clearly.

5. Review from another person's perspective.

👉 **Example:** "Click login button" instead of "Perform login action".

💡 **If it needs explanation, it is not clear enough.**

Question 100:

How do you avoid duplication in test cases?

Answer:

1. Review existing test cases before writing new ones.

2. Identify reusable steps and flows.

3. Avoid repeating the same logic.

4. Create modular and reusable test cases.

5. Maintain proper documentation.

👉 **Example:** Reusing login steps instead of rewriting them.

💡 **Smart reuse is better than repeated work.**

Question 101:

What is the role of expected result?

Answer:

1. Defines the expected system behavior.

2. Acts as a benchmark for validation.

3. Removes ambiguity during testing.

4. Helps identify defects accurately.

5. Ensures consistency across testers.

👉 **Example:** User should be redirected to dashboard after login.

💡 **No expected result = no real testing.**

Question 102:

How do you write reusable test cases?

Answer:

1. Keep test cases generic and flexible.

2. Avoid hardcoding values.

3. Create modular steps.

4. Separate common flows.

5. Design for reuse across scenarios.

👉 **Example:** Common login steps reused across modules.

💡 **Reusability saves time and improves consistency.**

Question 103:

What is the difference between good and bad test case?

Answer:

Good Test Case	Bad Test Case
1. Clear and structured	1. Vague and confusing
2. Covers real scenarios	2. Missing steps
3. Easy to execute	3. No expected result
4. Has defined expected result	4. Hard to understand
5. Adds value	5. Wastes effort

💡 **A bad test case wastes time, a good one saves it.**

Question 104:

How do you make test cases easy to understand for others?

Answer:

1. Use simple and clear language.

2. Write step-by-step instructions.

3. Avoid assumptions.

4. Maintain proper structure.

5. Think from reader's perspective.

👉 **Example:** Clear steps with defined expected results.

💡 **Write for the team, not just for yourself.**

Question 105:

Do you prioritize test cases? How?

Answer:

1. Yes, based on risk and business impact.

2. Focus on critical functionalities first.

3. Prioritize high user-impact areas.

4. Consider time constraints.

5. Test core flows before secondary features.

👉 **Example:** Testing checkout before UI alignment.

💡 **Priority decides effectiveness.**

Question 106:

Can you show an example of a poorly written test case?

Answer:

1. Lacks clarity and structure.

2. Missing steps and expected results.

3. Difficult to understand.

4. Leads to inconsistent execution.

5. Wastes time instead of adding value.

👉 **Example:** “Check login functionality” ❌ vs detailed steps ✓

💡 **Poor test cases create poor testing.**

Question 107:

How do you ensure coverage through test cases?

Answer:

1. Map test cases with requirements.

2. Cover positive, negative, and edge scenarios.

3. Use techniques like boundary value and equivalence.

4. Track using RTM.

5. Ensure no critical scenario is missed.

👉 **Example:** Testing login with valid, invalid, and boundary inputs.

💡 **Coverage builds confidence, not just reports.**

Question 108:

What is more important: test cases or scenarios?

Answer:

1. It is not about choosing one, but using both based on need.

2. Test cases provide depth, precision, and repeatability.

3. Scenarios provide speed, flexibility, and broader coverage.

4. I choose based on project stage, risk, and time constraints.

5. A smart tester balances both instead of blindly choosing one.

👉 **Example:** Scenarios for early testing, test cases for regression.

💡 **Smart QA chooses approach based on impact, not habit.**

Question 109:

What is the difference between test case and test scenario?

Answer:

Test Case	Test Scenario
1. Detailed step-by-step execution	1. High-level testing idea
2. Defines inputs and expected outputs	2. Focuses on user journey
3. Ensures accuracy and repeatability	3. No detailed steps
4. Used in regression and critical flows	4. Used for quick coverage and exploration

💡 **Scenario tells WHAT to test, test case defines HOW to test.**

Question 110:

When do you prefer scenarios over test cases?

Answer:

1. When requirements are still evolving.

2. During exploratory testing.

3. When quick coverage is needed.

4. When time is limited.

5. When flexibility is more important than structure.

👉 **Example:** Smoke testing before release using scenarios.

💡 **Scenarios are for speed, not for detail.**

Question 111:

Can you give a real example where scenario is more useful?

Answer:

1. In exploratory testing scenarios provide flexibility.

2. They allow testing multiple paths quickly.

3. They help uncover unexpected behavior.

4. They are not restricted by predefined steps.

5. They enable creative and dynamic testing.

👉 **Example:** Testing checkout flow with different user behaviors.

💡 **Scenarios unlock creativity in testing.**

Question 112:

Do you always convert scenarios into test cases?

Answer:

1. No, only when detailed validation is required.

2. Critical flows need structured test cases.

3. Scenarios are enough for exploratory testing.

4. Conversion depends on repeatability and risk.

5. I avoid unnecessary documentation.

👉 **Example:** Payment flow converted into detailed test cases.

💡 **Detail only where it matters most.**

Question 113:

Which is better for time-constrained projects?

Answer:

1. Scenarios are more practical in limited time.

2. They provide faster execution.

3. They give broader coverage quickly.

4. Test cases are used only for critical areas.

5. Focus is on maximum value in minimum time.

👉 **Example:** Smoke testing using scenarios before release.

💡 **In limited time, test smart — not everything.**

Question 114:

How do scenarios help in exploratory testing?

Answer:

1. They provide direction without restriction.

2. Allow testers to explore freely.

3. Help uncover hidden defects.

4. Enable dynamic testing.

5. Encourage creative thinking.

👉 **Example:** Exploring login with unexpected inputs and flows.

💡 **Scenarios guide thinking, not limit it.**

Question 115:

What is the impact of skipping detailed test cases?

Answer:

1. Increases risk of missing edge cases.

2. Reduces accuracy and consistency.

3. Creates gaps in validation.

4. Impacts traceability.

5. May lead to production defects.

👉 **Example:** Missing validation due to scenario-only testing.

💡 **Speed without structure creates gaps.**

Question 116:

Can scenarios miss edge cases?

Answer:

1. Yes, because they are high-level.

2. They do not enforce detailed validation.

3. Edge cases require structured testing.

4. Boundary conditions may be missed.

5. Balance is required to avoid gaps.

👉 **Example:** Missing boundary values in input testing.

💡 **High-level testing can miss low-level defects.**

Question 117:

How do you balance both test cases and scenarios?

Answer:

1. Use scenarios for early exploration.

2. Use test cases for critical flows.

3. Combine both for better coverage.

4. Adjust based on project needs.

5. Focus on both speed and accuracy.

👉 **Example:** Scenarios first, then test cases for regression.

💡 **Balance creates a powerful testing strategy.**

Question 118:

What do you use in Agile projects more?

Answer:

1. I use scenarios more due to fast-changing requirements.

2. They allow quick feedback.

3. Test cases are used for critical flows.

4. Avoid over-documentation.

5. Focus on speed and adaptability.

👉 **Example:** Scenarios for sprint testing, test cases for regression.

💡 **Agile needs speed with smart structure.**

Question 119:

How many test cases are enough?

Answer:

1. There is no fixed number of test cases that can be considered enough.

2. The focus should be on meaningful and complete coverage.

3. I ensure all critical functionalities and high-risk areas are tested.

4. I include edge cases and real user scenarios.

5. “Enough” means confidence that no major gap is left untested.

👉 **Example:** 10 strong test cases covering edge cases are better than 50 repetitive ones.

💡 **Effective coverage matters more than count.**

Question 120:

How do you decide test coverage is complete?

Answer:

1. When all requirements are covered.

2. When all critical business flows are tested.

3. When edge cases and negative scenarios are validated.

4. When no major gap is identified.

5. When testing gives confidence in system behavior.

👉 **Example:** Covering valid, invalid, and boundary scenarios.

💡 **Coverage builds confidence, not perfection.**

Question 121:

Do you follow any technique (like boundary value, equivalence)?

Answer:

1. Yes, I use Boundary Value Analysis and Equivalence Partitioning.

2. These techniques help reduce unnecessary test cases.

3. They focus on critical input ranges and boundaries.

4. They increase efficiency without reducing coverage.

5. They help find defects where they are most likely.

👉 **Example:** Range 1–100 → test 0, 1, 100, 101.

💡 **Smart techniques improve efficiency.**

Question 122:

What is risk-based testing?

Answer:

1. Risk-based testing focuses on high-impact areas first.

2. It prioritizes features where failure is costly.

3. It ensures critical functionality is always tested.

4. It is useful when time or resources are limited.

5. It improves efficiency and effectiveness.

👉 **Example:** Testing payment before UI design.

💡 **Test where failure hurts the most.**

Question 123:

Do you stop when all requirements are covered?

Answer:

1. No, requirement coverage is just the starting point.

2. Real users behave differently from documented requirements.

3. I test edge cases and unexpected scenarios.

4. I explore beyond defined behavior.

5. I ensure real-world usage is validated.

👉 **Example:** Testing unexpected inputs not mentioned in requirements.

💡 **Bugs often exist beyond requirements.**

Question 124:

What if time is limited to write test cases?

Answer:

1. I focus on critical and high-risk scenarios.

2. I avoid writing unnecessary test cases.

3. I use scenarios or exploratory testing.

4. I ensure key functionality is covered.

5. I maximize value within limited time.

👉 **Example:** Testing login and payment first.

💡 **In limited time, test what matters most.**

Question 125:

How do you prioritize test cases?

Answer:

1. Based on business impact.

2. Based on risk level.

3. Based on frequency of use.

4. Based on recent changes.

5. Based on user impact.

👉 **Example:** Checkout flow before profile settings.

💡 **Priority follows impact and risk.**

Question 126:

Can more test cases reduce quality?

Answer:

1. Yes, too many unnecessary test cases reduce efficiency.

2. They increase execution time.

3. They create confusion and duplication.

4. They shift focus away from important scenarios.

5. Testing should be meaningful, not overloaded.

👉 **Example:** Repeating same test without value.

💡 **More test cases ≠ better testing.**

Question 127:

How do you avoid over-testing?

Answer:

1. I eliminate duplicate scenarios.

2. I focus on meaningful coverage.

3. I use smart techniques like equivalence partitioning.

4. I avoid repetitive testing without value.

5. I ensure every test adds insight.

👉 **Example:** Avoid testing same scenario multiple times.

💡 **Test smart, not blindly.**

Question 128:

What is minimum vs optimal coverage?

Answer:

Minimum Coverage	Optimal Coverage
1. Covers basic functionality	1. Includes edge cases
2. Focus on critical flows	2. Covers risk areas
3. Quick validation	3. Provides higher confidence

👉 **Example:** Minimum = login works, Optimal = login + edge + security.

💡 **Optimal coverage builds real confidence.**

Question 129:

How do you measure coverage?

Answer:

1. By mapping test cases to requirements using RTM.

2. By tracking tested vs untested scenarios.

3. By validating edge cases and risk areas.

4. By ensuring all flows are covered.

5. By identifying gaps in testing.

👉 **Example:** Tracking coverage using RTM.

💡 What you track, you can improve.

Question 130:

Write test cases for a “Password Reset” feature

Answer:

1. I design test cases covering complete user journey from request to login.

2. I include positive, negative, and edge scenarios.

3. I validate OTP/email verification and token handling.

4. I ensure security validations like expiry and reuse restrictions.

5. I focus on both functionality and security aspects.

💡 Password reset is a high-risk entry point — test it like security, not just functionality.

Question 131:

What are the main scenarios in password reset for a Password Reset feature?

Answer:

1. Successful password reset flow.

2. Invalid email or OTP scenarios.

3. Expired or reused links/tokens.

4. Incorrect password inputs.

5. Multiple requests and retry behavior.

👉 **Example:** User enters wrong OTP and system blocks access.

💡 A strong system handles both users and attackers.

Question 132:

How do you test OTP or email verification for a Password Reset feature?

Answer:

1. Verify OTP/email is generated correctly.

2. Validate timely delivery.

3. Test valid, invalid, and expired OTP.

4. Ensure OTP cannot be reused.

5. Validate secure backend verification.

👉 **Example:** Enter expired OTP → system should reject.

💡 **OTP validation is the gatekeeper of identity.**

Question 133:

What edge cases will you include for a Password Reset feature?

Answer:

1. Empty and null inputs.

2. Boundary password lengths.

3. Special characters and invalid formats.

4. Multiple reset requests.

5. Interrupted flows or browser close scenarios.

👉 **Example:** User closes browser mid-reset and resumes later.

💡 **Edge cases reveal real-world failures.**

Question 134:

How do you test expired links or tokens for a Password Reset feature?

Answer:

1. Verify link expires after defined time.

2. Ensure expired links cannot be reused.

3. Validate proper error message is shown.

4. Ensure user is redirected to request new link.

5. Confirm no sensitive data is exposed.

👉 **Example:** Clicking expired link shows “Link expired”.

💡 **Expired links must fail safely.**

Question 135:

What security aspects will you test for a Password Reset feature?

Answer:

1. Secure token generation and encryption.

2. OTP validation and expiry.

3. Brute-force protection.

4. Rate limiting.

5. No sensitive data exposure.

👉 **Example:** SQL injection attempt blocked in reset flow.

💡 **Security protects user trust.**

Question 136:

How do you test invalid or reused reset links for a Password Reset feature?

Answer:

1. Validate system rejects invalid links.

2. Ensure reused links are blocked.

3. Verify clear error message is shown.

4. Check backend validation is strict.

5. Ensure no access is granted.

👉 **Example:** Reused link shows “Invalid or expired”.

💡 **One-time links are critical for security.**

Question 137:

What happens if user requests reset multiple times for a Password Reset feature?

Answer:

1. Only latest link should be valid.

2. Previous links should be invalidated.

3. Rate limiting should be applied.

4. Prevent spam or misuse.

5. Maintain smooth user experience.

👉 **Example:** Only last email link works.

💡 **Control prevents misuse.**

Question 138:

How do you test password rules for a Password Reset feature?

Answer:

1. Validate min/max length.

2. Test uppercase, lowercase, special characters.

3. Check invalid formats.

4. Validate boundary values.

5. Ensure proper validation messages.

👉 **Example:** Password without special char → reject.

💡 **Strong rules ensure strong security.**

Question 139:

Will you test session behavior after reset for a Password Reset feature?

Answer:

1. Yes, verify all sessions are invalidated.

2. Ensure user must login again.

3. Validate session security.

4. Prevent unauthorized access.

5. Ensure proper session reset behavior.

👉 **Example:** Old session expires after reset.

💡 **Reset must remove old access.**

Question 140:

How do you validate error messages for a Password Reset feature?

Answer:

1. Ensure messages are clear and user-friendly.

2. Avoid exposing system details.

3. Validate correct triggering of messages.

4. Ensure consistency across flows.

5. Guide user with proper information.

👉 **Example:** “Invalid OTP” instead of system error code.

💡 **Good messages help users, not attackers.**

Question 141:

What will you do if developer says “it’s not a bug”?

Answer:

1. I first stay calm and avoid taking it personally.

2. I re-check the issue from requirement and user perspective.

3. I explain the issue clearly with proper steps and impact.

4. I discuss it logically instead of arguing emotionally.

5. If needed, I involve BA or stakeholders for clarification.

👉 **Example:** Developer says validation is correct, but requirement says otherwise → discussed with BA and clarified.

💡 **Focus on facts, not ego.**

Question 142:

How do you prove that the bug you created is actually a bug?

Answer:

1. I refer to requirement or acceptance criteria.

2. I show clear steps to reproduce the issue.

3. I attach screenshots or recordings.

4. I explain expected vs actual behavior.

5. I highlight user and business impact.

👉 **Example:** Showing mismatch between expected validation and actual system behavior.

💡 **Evidence is stronger than opinion.**

Question 143:

Will you escalate immediately if developer says “it’s not a bug”?

Answer:

1. No, I do not escalate immediately.

2. I first try to resolve through discussion.

3. I ensure I fully understand the developer’s perspective.

4. I validate with requirements again.

5. Escalation is the last option when clarity is not achieved.

👉 **Example:** Discussed internally before involving BA.

💡 **Escalate for clarity, not for conflict.**

Question 144:

How do you handle the situations when developer says “it’s not a bug” without conflict?

Answer:

1. I keep communication respectful and professional.

2. I focus on facts instead of opinions.

3. I listen to developer’s reasoning.

4. I present my understanding with logic.

5. I aim for solution, not winning the argument.

👉 **Example:** Calm discussion with supporting data instead of argument.

💡 **Collaboration solves problems, conflict creates them.**

Question 145:

Do you refer to requirement or business logic when developer says “it’s not a bug”?

Answer:

1. Yes, requirement is the primary source of truth.

2. I validate against acceptance criteria.

3. I consider business logic and real user behavior.

4. I avoid personal assumptions.

5. Decision should always align with requirement.

👉 **Example:** Requirement says mandatory field, but system allows empty input.

💡 **Requirement is stronger than assumption.**

Question 146:

How do you involve BA or product owner when developer says “it’s not a bug”?

Answer:

1. I involve BA when requirement is unclear or disputed.

2. I present both perspectives clearly.

3. I provide evidence and scenarios.

4. I seek clarification, not validation of my opinion.

5. I ensure final decision is documented.

👉 **Example:** BA confirmed expected behavior and resolved conflict.

💡 **Clarity removes confusion.**

Question 147:

Can you give a real example when developer says “it’s not a bug”?

Answer:

1. I found a validation issue in a form.

2. Developer said it was expected behavior.

3. I checked requirement and found mismatch.

4. I discussed with BA.

5. It was confirmed as a bug and fixed.

👉 **Example:** Email field accepting invalid format.

💡 **Real clarity comes from validation, not assumption.**

Question 148:

What if developer is senior and still disagree with the bug you created?

Answer:

1. I respect experience but rely on facts.

2. I validate with requirement and business logic.

3. I present evidence calmly.

4. I avoid ego-based arguments.

5. I involve stakeholders if needed.

👉 **Example:** Senior developer disagreed but requirement proved otherwise.

💡 **Respect people, but trust facts.**

Question 149:

How do you document such discussions when developer says “it’s not a bug”?

Answer:

1. I document discussion in defect or tracking tool.

2. I include developer comments and my observations.

3. I attach evidence like screenshots or logs.

4. I mention final decision clearly.

5. I ensure traceability for future reference.

👉 **Example:** Updating Jira ticket with discussion and final resolution.

💡 **Documentation turns confusion into clarity.**

Question 150:

What if your bug is rejected?

Answer:

1. I stay calm and do not take rejection personally.

2. I review the reason for rejection carefully.

3. I re-check the issue against requirements and expected behavior.

4. I discuss with the developer to understand their perspective.

5. I treat it as a learning opportunity instead of a failure.

👉 **Example:** Bug rejected as expected behavior → validated with requirement and learned better understanding.

💡 **Rejection is feedback, not failure.**

Question 151:

How do you respond to a rejected bug?

Answer:

1. I respond professionally without emotional reaction.

2. I understand the reason for rejection.

3. I verify the issue again.

4. I provide additional evidence if needed.

5. I focus on resolution, not argument.

👉 **Example:** Added screenshots and logs to clarify bug.

💡 **Professional response builds trust.**

Question 152:

Will you reopen the rejected bug? On what basis?

Answer:

1. Yes, I reopen only if I have valid evidence.

2. I ensure issue still exists.

3. I validate against requirement or business logic.

4. I provide clear reproduction steps.

5. I reopen with proper justification, not assumption.

👉 **Example:** Bug reopened after verifying incorrect validation behavior.

💡 **Reopen with proof, not opinion.**

Question 153:

What if rejection of bug is valid?

Answer:

1. I accept the rejection positively.

2. I understand where my understanding was incorrect.

3. I update my knowledge of requirements.

4. I avoid repeating same mistake.

5. I treat it as a learning experience.

👉 **Example:** Logged issue based on assumption → learned correct requirement.

💡 **Learning matters more than being right.**

Question 154:

How do you avoid emotional reaction when our bug is rejected??

Answer:

1. I focus on facts instead of feelings.

2. I remind myself that rejection is part of the process.

3. I keep communication professional.

4. I avoid ego-based thinking.

5. I focus on improving quality, not proving myself right.

👉 **Example:** Calmly discussing issue instead of arguing.

💡 **Strong testers control emotions, not situations.**

Question 155:

Do you verify before logging the bug?

Answer:

1. Yes, verification is mandatory before logging.

2. I reproduce the issue multiple times.

3. I validate against requirements.

4. I check environment and test data.

5. I ensure it is not expected behavior.

👉 **Example:** Retesting issue before raising bug.

💡 **Verify before you report.**

Question 156:

What if bug is marked as “won’t fix”?

Answer:

1. I understand the reason behind the decision.

2. I check if it is low priority or acceptable risk.

3. I ensure it is documented properly.

4. I confirm stakeholders are aware.

5. I move forward with clarity.

👉 **Example:** Minor UI issue marked as “won’t fix”.

💡 **Not every bug needs fixing.**

Question 157:

How do you learn from rejected bugs?

Answer:

1. I analyze why the bug was rejected.

2. I identify gaps in my understanding.

3. I improve requirement analysis skills.

4. I refine bug reporting quality.

5. I apply learning in future testing.

👉 **Example:** Improved clarity after repeated rejection.

💡 **Every rejection improves your skill.**

Question 158:

Can you share a real example of a rejected bug?

Answer:

1. I reported a validation issue in a form.

2. Developer marked it as expected behavior.

3. After discussion, requirement supported developer.

4. I updated my understanding.

5. I avoided similar mistakes later.

👉 **Example:** Optional field treated as mandatory by mistake.

💡 **Experience teaches what theory cannot.**

Question 159:

How do you improve bug quality after rejection?

Answer:

1. I write clearer bug descriptions.

2. I include proper steps and evidence.

3. I validate requirement before logging.

4. I improve clarity of expected vs actual.

5. I focus on impact and reproducibility.

👉 **Example:** Adding logs and screenshots.

💡 **Better bugs lead to better fixes.**

Question 160:

Do you discuss with team before logging critical bugs?

Answer:

1. Yes, for critical bugs I prefer quick discussion.

2. It helps validate understanding.

3. It avoids false positives.

4. It improves clarity before reporting.

5. It ensures correct severity and priority.

👉 **Example:** Discussing payment failure before logging critical bug.

💡 **Quick discussion prevents wrong reporting.**

Question 161:

How do you decide severity vs priority?

Answer:

1. I decide severity based on impact on system functionality.

2. I decide priority based on urgency and business impact.

3. Severity is technical impact, priority is business importance.

4. I evaluate both before assigning values.

5. I ensure alignment with business needs and release timelines.

👉 **Example:** Minor UI issue (low severity) but on homepage before release → high priority.

💡 **Severity = impact, Priority = urgency.**

Question 162:

Who decides severity and who decides priority?

Answer:

Severity	Priority
1. Decided by tester	1. Decided by business/PO/manager
2. Based on technical impact	2. Based on urgency
3. Focus on system behavior	3. Focus on business impact

💡 **Tester finds impact, business decides urgency.**

Question 163:

Can severity and priority be different? Give example.

Answer:

1. Yes, severity and priority can be different.

2. A bug may have high impact but low urgency.

3. Or low impact but high business urgency.

4. Both are independent decisions.

5. Final decision depends on context.

👉 **Example:** Typo on homepage → low severity but high priority before release.

💡 **Impact and urgency don't always match.**

Question 164:

What factors affect severity?

Answer:

1. Impact on core functionality.

2. Data loss or corruption.

3. System crash or failure.

4. Security impact.

5. User experience impact.

👉 **Example:** Payment failure → high severity.

💡 **Severity depends on system damage.**

Question 165:

What factors affect priority?

Answer:

1. Business impact.

2. Release deadlines.

3. Customer visibility.

4. Frequency of use.

5. Stakeholder decisions.

👉 **Example:** Homepage issue before release → high priority.

💡 **Priority depends on business urgency.**

Question 166:

How do you handle disagreement on priority?

Answer:

1. I stay calm and avoid conflict.

2. I explain impact clearly.

3. I provide real examples.

4. I involve stakeholders if needed.

5. I focus on business value, not personal opinion.

👉 **Example:** Explained risk of delay → priority increased.

💡 **Clarity resolves disagreement.**

Question 167:

Can you give real-life examples for each combination of severity and priority?

Answer:

1. High severity, High priority → Payment failure.

2. High severity, Low priority → Rare crash in edge case.

3. Low severity, High priority → Typo on homepage before release.

4. Low severity, Low priority → Minor UI misalignment.

👉 **Example:** Payment failure fixed immediately, UI issue later.

💡 **All combinations are possible in real projects.**

Question 168:

What happens if severity is high but priority is low?

Answer:

1. Issue has serious impact but low urgency.

2. It may affect rare scenarios.

3. It is scheduled for later fix.

4. Risk is accepted temporarily.

5. It is monitored for future impact.

👉 **Example:** Rare crash in uncommon condition.

💡 **Not all critical issues are urgent.**

Question 169:

How do you communicate critical bugs?

Answer:

1. I communicate immediately.

2. I clearly explain impact and risk.

3. I provide reproduction steps.

4. I highlight urgency.

5. I ensure stakeholders are informed.

👉 **Example:** Payment failure reported instantly to team.

💡 **Critical bugs need immediate visibility.**

Question 170:

Do you ever change severity after discussion?

Answer:

1. Yes, if new information changes impact understanding.

2. I validate with team and requirements.

3. I ensure updated value is correct.

4. I document changes.

5. I focus on accuracy, not ego.

👉 **Example:** Issue downgraded after impact clarified.

💡 **Correctness matters more than consistency.**

Question 171:

Do you ever change priority after discussion?

Answer:

1. Yes, priority can change based on business needs.

2. I align with stakeholders and timelines.

3. I update based on urgency changes.

4. I ensure clarity in decision.

5. I document updated priority.

👉 **Example:** Issue priority increased before release.

💡 **Priority is dynamic, not fixed.**

Question 172:

What if a critical bug is found just before release?

Answer:

1. I immediately assess the impact of the bug on core functionality.

2. I inform stakeholders without delay.

3. I clearly explain the risk and possible consequences.

4. I help evaluate whether release should proceed or not.

5. My focus is to protect user experience and business trust.

👉 **Example:** Payment failure found before release → team paused release and fixed it.

💡 **A critical bug before release is a business risk, not just a defect.**

Question 173:

Will you stop the release if a critical bug is found just before release? On what basis?

Answer:

1. I do not take the decision alone, but I strongly recommend based on impact.

2. If the bug affects core functionality, I suggest stopping the release.

3. I evaluate risk, user impact, and business consequences.

4. I provide clear facts to stakeholders.

5. Final decision depends on business risk tolerance.

👉 **Example:** Login failure → recommended release stop.

💡 **Recommendation is based on impact, decision is business-driven.**

Question 174:

Who takes the final decision to stop /go the release If a critical bug is found just before release?

Answer:

1. Final decision is taken by business stakeholders or product owner.

2. QA provides impact analysis and recommendation.

3. Developers provide technical feasibility.

4. Decision is collaborative but business-driven.

5. QA ensures decision is informed and risk-aware.

👉 **Example:** PO decides after reviewing QA risk analysis.

💡 **QA informs, business decides.**

Question 175:

How do you communicate urgency? If a critical bug is found just before release?

Answer:

1. I communicate immediately without delay.

2. I clearly explain impact and risk.

3. I avoid technical jargon and keep it simple.

4. I highlight urgency and consequences.

5. I ensure stakeholders understand severity.

👉 **Example:** “Payment is failing — users cannot complete transactions.”

💡 **Urgency should be clear, not exaggerated.**

Question 176:

What if business still wants to release even a critical bug is found just before release?

Answer:

1. I clearly document the risk and impact.

2. I ensure stakeholders are fully aware of consequences.

3. I suggest possible mitigation strategies.

4. I respect final business decision.

5. I ensure traceability of decision.

👉 **Example:** Business accepted risk and released with known issue.

💡 **Risk accepted should be risk documented.**

Question 177:

How do you assess risk quickly If a critical bug is found just before release?

Answer:

1. I identify impacted functionality.

2. I evaluate user impact and business loss.

3. I check frequency of occurrence.

4. I assess workaround availability.

5. I estimate worst-case scenario.

👉 **Example:** Payment bug → high risk due to revenue impact.

💡 **Quick risk analysis needs clear thinking.**

Question 178:

What is rollback strategy?

Answer:

1. Rollback strategy is reverting to previous stable version.

2. It is used when release causes critical issues.

3. It ensures system stability.

4. It minimizes business impact.

5. It should be planned before release.

👉 **Example:** Reverting to previous version after failure.

💡 **Always plan rollback before release.**

Question 179:

How do you handle pressure in such situation If a critical bug is found just before release?

Answer:

1. I stay calm and avoid panic.

2. I focus on facts instead of emotions.

3. I prioritize tasks logically.

4. I follow structured approach.

5. I communicate clearly.

👉 **Example:** Listing impacted areas and addressing them step by step.

💡 **Calm mind makes better decisions.**

Question 180:

Have you faced this situation in real life when a critical bug is found just before release?

Answer:

1. Yes, I have faced such situations.

2. A critical issue was found before release.

3. I reported it immediately.

4. Team assessed risk and delayed release.

5. Issue was fixed before deployment.

👉 **Example:** Payment issue discovered before release → release delayed.

💡 **Real experience builds confidence.**

Question 181:

What documentation will you provide to business If a critical bug is found just before release?

Answer:

1. Detailed bug report.

2. Impact analysis document.

3. Risk assessment summary.

4. Possible workaround or mitigation.

5. Recommendation (go/no-go).

👉 **Example:** Document showing impact on payment flow and user experience.

💡 **Clear documentation supports better decisions.**

Question 182:

Have you ever missed a bug? What did you learn?

Answer:

1. Yes, I have missed bugs during testing.

2. I realized that no testing is perfect and gaps can exist.

3. I learned the importance of covering edge cases.

4. I improved my requirement analysis and test coverage.

5. I treated it as a learning opportunity to improve quality.

👉 **Example:** Missed validation issue that appeared in production.

💡 **Missing a bug is a mistake, not learning from it is the real mistake.**

Question 183:

What will be your Answer if your manager asks you that why was the bug missed?

Answer:

1. I take responsibility without blaming others.

2. I explain the actual reason clearly.

3. I identify gaps in testing or understanding.

4. I present corrective actions.

5. I focus on preventing future misses.

👉 **Example:** Edge case not covered in test cases.

💡 **Ownership builds trust.**

Question 184:

How did you identify the root cause of missed bug?

Answer:

1. I analyzed the defect and testing steps.

2. I checked whether scenario was covered.

3. I reviewed requirement understanding.

4. I identified gap in test coverage.

5. I documented root cause for future reference.

👉 **Example:** Boundary case not tested.

💡 **Root cause analysis prevents repetition.**

Question 185:

What steps did you take to avoid missing a bug again?

Answer:

1. I improved test case coverage.

2. I added edge cases and negative scenarios.

3. I reviewed requirements more carefully.

4. I used checklists and techniques.

5. I performed peer review when needed.

👉 **Example:** Added boundary testing cases.

💡 **Improvement is the real success.**

Question 186:

Did you update your test cases when you found that a bug was missed?

Answer:

1. Yes, I updated test cases immediately.

2. I added missed scenarios.

3. I ensured future coverage.

4. I improved clarity in test cases.

5. I maintained documentation properly.

👉 **Example:** Added validation scenario after miss.

💡 **Every missed bug improves future testing.**

Question 187:

How did your team react when a bug was missed?

Answer:

1. Team focused on resolving issue instead of blaming.

2. We analyzed root cause.

3. We improved testing strategy.

4. We ensured better coverage.

5. It became a learning experience.

👉 **Example:** Team added extra validation testing.

💡 **Strong teams learn, weak teams blame.**

Question 188:

How you calculate the missed bug impact on production?

Answer:

1. I check affected functionality.

2. I evaluate number of users impacted.

3. I analyze business loss or revenue impact.

4. I assess severity of issue.

5. I determine urgency of fix.

👉 **Example:** Payment issue affecting transactions.

💡 **Impact defines urgency.**

Question 189:

How do you improve coverage after missing a bug?

Answer:

1. I review existing test cases.

2. I add missing scenarios.

3. I include edge cases.

4. I improve requirement understanding.

5. I use better testing techniques.

👉 **Example:** Added negative testing scenarios.

💡 **Coverage grows from experience.**

Question 190:

What tools or techniques can prevent such misses?

Answer:

1. Boundary value analysis.

2. Equivalence partitioning.

3. Checklists and test reviews.

4. Automation where possible.

5. Requirement traceability matrix.

👉 **Example:** Using RTM to ensure full coverage.

💡 **Smart techniques reduce misses.**

Question 191:

Can you share a real example when you missed a bug?

Answer:

1. I missed a validation issue in a form.

2. It worked in normal cases but failed in edge scenario.

3. It was found in production.

4. I analyzed root cause.

5. I improved testing approach.

👉 **Example:** Empty input accepted due to missing validation test.

💡 **Experience is built through mistakes.**

Question 192:

What do you do daily as a QA tester?

Answer:

1. I start my day by reviewing tasks and priorities.

2. I attend daily standup and share progress.

3. I perform testing on assigned features.

4. I log defects with proper details.

5. I collaborate with developers and team.

👉 **Example:** Testing login module and reporting defects.

💡 **Daily QA work is about consistency and clarity.**

Question 193:

Can you walk me through your typical daily routine as a QA tester?

Answer:

1. I review tasks and requirements.

2. I attend standup meeting.

3. I execute test cases.

4. I log and track bugs.

5. I communicate updates and blockers.

👉 **Example:** Testing feature, logging bug, discussing with developer.

💡 **A structured routine improves productivity.**

Question 194:

As a QA tester how do you start your day — standup or testing?

Answer:

1. I usually start with standup meeting.

2. It helps align with team priorities.

3. After standup, I begin testing tasks.

4. I adjust based on urgent requirements.

5. Flexibility is important in daily workflow.

👉 **Example:** Standup highlights urgent bug → testing starts immediately.

💡 **Start with clarity, then execution.**

Question 195:

Do you attend daily scrum meetings? What do you share there?

Answer:

1. Yes, I attend daily scrum meetings.

2. I share what I worked on yesterday.

3. I mention what I will do today.

4. I highlight any blockers.

5. I keep updates short and clear.

👉 **Example:** Reported bug and waiting for fix.

💡 **Clear communication keeps team aligned.**

Question 196:

How do you track your daily QA tasks?

Answer:

1. I use task management tools like Jira.

2. I track tasks based on priority.

3. I update status regularly.

4. I ensure visibility for team.

5. I follow structured workflow.

👉 **Example:** Updating Jira tickets after testing.

💡 **Tracking ensures accountability.**

Question 197:

What tools do you use daily (Jira, TestRail, etc.)?

Answer:

1. I use Jira for bug tracking.

2. I use TestRail or similar tools for test cases.

3. I use browser tools for testing.

4. I use communication tools like Slack.

5. Tools depend on project requirements.

👉 **Example:** Logging bug in Jira and updating test case in TestRail.

💡 **Tools support process, not replace thinking.**

Question 198:

How much time do you spend on testing vs reporting bugs?

Answer:

1. Most time is spent on testing activities.

2. Bug reporting takes less time but is important.

3. Balance depends on project phase.

4. Quality reporting saves time later.

5. Both are equally important for QA.

👉 **Example:** 70% testing, 30% reporting.

💡 **Testing finds bugs, reporting communicates them.**

Question 199:

How do you handle communication with developers daily?

Answer:

1. I communicate clearly and professionally.

2. I discuss issues with proper evidence.

3. I avoid assumptions and focus on facts.

4. I maintain collaboration.

5. I ensure clarity in discussions.

👉 **Example:** Sharing bug details with screenshots.

💡 **Good communication prevents confusion.**

Question 200:

Do you update test cases daily or only when needed?

Answer:

1. I update test cases when needed.

2. Changes in requirements trigger updates.

3. I avoid unnecessary updates.

4. I ensure accuracy of test cases.

5. I maintain proper documentation.

👉 **Example:** Updating test case after requirement change.

💡 **Update with purpose, not habit.**

Question 201:

What challenges do you face in your daily work?

Answer:

1. Incomplete requirements.

2. Time constraints.

3. Communication gaps.

4. Changing requirements.

5. Handling multiple tasks.

👉 **Example:** Requirement changes during testing phase.

💡 **Challenges improve problem-solving skills.**

Question 202:

Can you give a real example of a busy testing day?

Answer:

1. Multiple features were assigned.

2. Critical bugs were found.

3. Urgent fixes needed validation.

4. Continuous communication with team.

5. Prioritization was key to handle workload.

👉 **Example:** Testing release build with multiple high-priority defects.

💡 **Busy days test your ability to prioritize.**