11

Manual Testing Fundamentals

The Bedrock

THE MODERN QA ENGINEER SKILLSET

Manual Testing Fundamentals: The Bedrock

Test Design, Execution, and the Thinking Behind Every Test

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

Manual Testing Fundamentals: The Bedrock

Test Design, Execution, and the Thinking Behind Every Test

Book 11 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction

A Self-Guided Learning Textbook

About This Book

Manual testing is not the absence of automation — it is the discipline of thinking critically about software behavior. Every senior QA engineer, no matter how deeply they work with AI-driven frameworks, relies on manual testing instincts daily: writing clear test cases, exploring software with intent, and communicating defects so developers act on them immediately.

This book teaches you the craft from the ground up. By the end, you will be able to:

- Design test cases using classical techniques that interviewers expect you to know
- Run structured exploratory testing sessions that find what scripts miss
- Write bug reports that get fixed in hours, not weeks
- Build risk-based test plans that focus effort where it matters
- Speak the language of professional QA with confidence

Who This Book Is For

- **Career changers** entering QA with zero testing experience
- **Junior QA engineers** who want to solidify their fundamentals
- **Bootcamp graduates** who learned tools but not the underlying discipline

- **Developers** who want to understand how QA engineers think
- **Senior engineers** brushing up before interviews

Prerequisites

None. This book assumes you have used software (websites, apps) but have never tested it professionally. If you know what a web browser is and can follow step-by-step instructions, you are ready.

How to Use This Book

Each chapter builds on the previous one. Read them in order the first time. After that, use individual chapters as reference.

Every chapter ends with:

- **Key Takeaways** — the essential points to remember
- **Self-Assessment Quiz** — test your understanding
- **Hands-On Exercises** — practice what you learned (rated Beginner / Intermediate / Advanced)

The book concludes with a **Capstone Project** that ties everything together.

Estimated completion time: 4-6 weeks at 1-2 hours per day.

PART I

TEST CASE DESIGN

Chapter 1: Anatomy of a Test Case

A test case is a contract between you and the future. Someone who has never seen the feature — including you in six months — must be able to execute it and reach the same conclusion.

1.1 The Seven Essential Fields

Every test case, regardless of your tooling (TestRail, Zephyr, Google Sheets, or plain Markdown), should include these fields:

Field	Purpose	Example
ID	Unique reference for traceability	TC-LOGIN-004
Title	One-line summary of what is being verified	Verify login fails with expired password

continued on next page

Field	Purpose	Example
Preconditions	State the system must be in before starting	User account exists with password expired 1 day ago
Steps	Numbered, atomic actions	1. Navigate to /login 2. Enter email 3. Enter expired password 4. Click Submit
Expected Result	Observable, measurable outcome	Error message "Your password has expired" is displayed; user is not redirected to dashboard
Actual Result	Filled during execution	(blank until run)
Test Data	Specific values needed	Email: expired@test.com, Password: OldPass123!

The ID Convention

Use a consistent naming pattern. Common formats:

- `TC-MODULE-NNN` — e.g., TC-LOGIN-004, TC-CART-012
- `FEATURE.SCENARIO.NNN` — e.g., AUTH.EXPIRED_PW.001
- Auto-incrementing from your test management tool

The format matters less than consistency. Pick one and enforce it across the team.

Title Best Practices

Start with a verb. Answer "what is being verified?" in one line:

- **Good**: "Verify login fails with expired password"
- **Good**: "Confirm cart total updates after removing an item"
- **Bad**: "Login test" (too vague — which aspect of login?)
- **Bad**: "Test that when a user who has an expired password tries to log in the system shows an error" (too long — this is a description, not a title)

Steps: The Atomic Rule

Each step should describe exactly one user action. If a step contains "and," it should probably be two steps.

```
BAD:
1. Navigate to the login page, enter credentials, and click Submit

GOOD:
1. Navigate to https://staging.example.com/login
2. Enter email: test@example.com
3. Enter password: Test123!
4. Click the "Sign In" button
```

Expected Results: Observable and Measurable

The expected result must be something you can see, read, measure, or verify in the UI, database, API response, or logs. Avoid subjective language.

```
BAD:  "The page should look correct"
GOOD: "The dashboard displays 'Welcome, John' in the header and shows 3 notification badges"

BAD:  "Login works"
GOOD: "User is redirected to /dashboard. The URL changes to /dashboard. The user's avatar appears in the top-right corner."
```

1.2 The Given/When/Then Template

```
GIVEN [precondition / system state]
WHEN  [user action or trigger]
THEN  [expected observable result]
```

This format, from BDD (Behavior-Driven Development), works even outside Cucumber frameworks. It forces you to separate setup from action from assertion.

Examples

```
GIVEN a registered user with an active account
WHEN  the user enters valid email and password and clicks Sign In
THEN  the user is redirected to /dashboard and sees a welcome message

GIVEN a shopping cart with 2 items totaling $50.00
WHEN  the user applies discount code "SAVE10" (10% off)
THEN  the cart total updates to $45.00 and the discount line is visible

GIVEN a user with no upload permissions
WHEN  the user attempts to upload a file via the API
THEN  the API returns HTTP 403 with message "Insufficient permissions"
```

When to Use Each Format

Scenario	Recommended Format
Agile team using BDD / Cucumber	Given/When/Then (mandatory)
Quick internal test documentation	Given/When/Then (concise)
Formal test plans for regulated industries	Traditional table format (detailed)
Exploratory testing charters	Neither — use free-form charters

1.3 Common Mistakes in Test Case Writing

> **Common Mistake #1: Vague Steps**
> "Log in to the app" — Which user? Which environment? What credentials? A new team member cannot execute this.
> **Fix**: "Navigate to https://staging.example.com/login. Enter email: test@example.com. Enter password: Test123!. Click the 'Sign In' button."

> **Common Mistake #2: Missing Preconditions**
> The test assumes a database state nobody sets up. Testing that a discount code works — but the code was created in staging last month and deleted since.
> **Fix**: State every precondition explicitly. Link to setup scripts or describe creation steps.

> **Common Mistake #3: Compound Assertions**
> Checking five things in one test case makes failure ambiguous. If the test fails, which of the five was the problem?
> **Fix**: One test case = one verification point. Separate cases can share preconditions.

> **Common Mistake #4: Platform Assumptions**
> "Click the button" — on mobile there is no click, there is a tap. "Right-click to open context menu" — does not work on touch devices.
> **Fix**: Specify the platform, or write platform-specific variants.

> **Common Mistake #5: Missing Negative Cases**
> Only testing the happy path. You wrote "Verify login succeeds" but never wrote "Verify login fails with empty password."
> **Fix**: For every positive test, ask: what should happen when the user does this wrong?

1.4 Test Case Maintenance

Writing test cases is half the job. Maintaining them is the other half.

Signs Your Test Cases Need Updating

- The feature was redesigned but tests still reference the old UI
- Steps reference environments or URLs that no longer exist
- Test data (user accounts, product IDs) has been cleaned up
- New edge cases were discovered in production but never added

Maintenance Practices

1. **Review during sprint refinement** — if a story modifies an existing feature, flag related test cases
2. **Tag by feature area** — makes it easy to find all tests related to "checkout" or "user management"
3. **Archive, don't delete** — removed features may return; archived tests save rework
4. **Version control** — keep test cases in Git or use your tool's versioning feature

> **Pro Tip**: Set a calendar reminder to audit test cases quarterly. Dead test cases erode team confidence in the entire suite.

Key Takeaways — Chapter 1

- A test case must be executable by someone who has never seen the feature
- Include all seven fields: ID, title, preconditions, steps, expected result, actual result, test data
- Use Given/When/Then to enforce clarity
- Avoid vague steps, compound assertions, and missing negative cases
- Maintain test cases as actively as you maintain code

Self-Assessment Quiz — Chapter 1

1. What are the seven essential fields of a test case?
2. Why is "Login test" a bad test case title? Write a better one.
3. What is the difference between the Expected Result and Actual Result fields?
4. Convert this into Given/When/Then format: "Test that a premium user can access the analytics dashboard after logging in."
5. You find a test case with this step: "Set up the test data and log in and navigate to settings." What's wrong with it, and how would you fix it?

Exercises — Chapter 1

Beginner: Write 3 test cases for a "Forgot Password" feature using the Given/When/Then format. Include at least one positive and one negative case.

Intermediate: Write 5 test cases for an e-commerce product search feature. Cover: successful search, empty results, special characters in query, very long query, and search with filters applied.

Advanced: You are handed a one-page feature spec for a "User Invitation" system (admin invites users by email, invited users receive a link,

link expires in 48 hours, invited users set their password). Write a complete test suite of 10-15 test cases covering positive, negative, and edge cases.

Career Translation

Resume phrasing

- Designed and maintained structured test cases with clear preconditions, atomic steps, and measurable expected results, reducing test execution ambiguity by 40% across QA team members
- Implemented Given/When/Then test case templates that standardized documentation and enabled non-QA stakeholders to review test coverage
- Established test case ID conventions and maintenance workflows that improved traceability from requirements to defects
- Authored 200+ reusable test cases covering positive, negative, and edge-case scenarios for mission-critical features

Cover letter framing

I bring discipline to test case design by treating every test case as a contract: clear enough for any team member to execute without ambiguity, thorough enough to catch defects that matter. My documentation practices ensure teams can scale testing without losing quality, and my focus on preconditions and test data means fewer "works on my machine" failures during regression runs.

Interview framing

"I approach test case writing as a communication exercise. A test case with vague steps like 'log in and check the page' creates different interpretations for every tester. I optimize for atomic steps, explicit test data, and observable expected results so that anyone — a new hire, a developer, or an auditor — reaches the same conclusion. Trade-offs include the time investment in detailed documentation versus faster-but-fragile ad-hoc notes, and I balance that by investing heavily in high-risk areas while keeping lower-risk cases leaner."

What not to say

- "I just follow the test cases someone else wrote" — signals you cannot design tests independently
- "I test whatever seems important" — implies no structured approach or traceability
- "Happy path is usually enough" — reveals ignorance of negative testing and edge cases
- "I keep the test steps in my head" — suggests undocumented, non-repeatable testing
- "Expected result: it should work correctly" — demonstrates inability to define measurable outcomes

Interview Depth Check

Question 1

Prompt: You join a team that has 500 test cases in a spreadsheet, but test execution takes 3 days and many tests frequently fail due to outdated steps. How would you approach improving this test suite?

What a strong answer should cover:

- Audit and triage: identify which test cases are outdated, redundant, or low-value
- Prioritize maintenance based on risk (critical flows first)
- Establish tagging/categorization to enable selective execution
- Introduce a review cadence tied to sprint refinement
- Archive rather than delete obsolete tests

Example answer:

- First, I would categorize all 500 tests by feature area and tag each with a last-verified date
- I would identify which failures are caused by stale test data or outdated steps versus actual product defects
- High-risk, high-frequency tests get updated immediately; low-risk tests get archived if they have not passed in 3+ sprints
- I would propose a quarterly audit cadence and tie test case reviews to sprint refinement so updates happen when features change

Question 2

Prompt: A developer says your test case is too detailed and slows them down when they need to verify a fix. How do you respond?

What a strong answer should cover:

- Acknowledge their perspective — developers need speed for quick verifications
- Explain the audience: test cases serve multiple stakeholders (new testers, auditors, future you)
- Propose tiered documentation: detailed for regression, lighter for smoke/sanity
- Offer to create a quick-verify checklist alongside detailed cases

Example answer:

- I would acknowledge that detailed steps can feel heavy for a developer doing a quick spot check
- I would explain that the detail serves the broader team — a new QA hire or an auditor needs that precision
- I would propose maintaining the full test case for regression while creating a short "smoke checklist" version the developer can use for quick verification
- The key is ensuring both artifacts stay in sync during maintenance

Question 3

Prompt: You are writing test cases for a feature with incomplete requirements. The product manager says final specs will arrive next week, but testing needs to start now. What do you do?

What a strong answer should cover:

- Write test cases based on available information and assumptions
- Explicitly document assumptions in preconditions
- Focus on obvious flows (login, CRUD, error handling) that are unlikely to change
- Flag gaps and schedule a review when full specs arrive

- Use exploratory testing to supplement scripted cases

Example answer:

- I would write test cases for the flows I can infer from mockups, conversations, and similar features in the product
- Each assumption gets documented: "Assumption: password minimum is 8 characters based on existing registration flow"
- I would focus first on structural behaviors (does the form submit? does validation trigger?) that are unlikely to change regardless of final specs
- Simultaneously, I would create exploratory testing charters for areas with the most uncertainty, and schedule a test case review session when the final specs land

Question 4

Prompt: Explain how you decide whether to use the traditional table format versus Given/When/Then for a test case. Give an example where each format is the better choice.

What a strong answer should cover:

- Given/When/Then excels for BDD teams, concise scenarios, and readability
- Traditional table format suits regulated industries, detailed multi-step flows, and audit trails
- The choice depends on audience, tooling, and compliance requirements
- Both formats can coexist in the same project

Example answer:

- I use Given/When/Then when the team practices BDD or when the scenario is a single user action with a clear outcome — for example, "GIVEN a cart with items, WHEN user applies a valid coupon, THEN the total decreases by the coupon amount." It is concise and readable by non-technical stakeholders

- I use the traditional table format when a regulated environment requires step-by-step audit evidence — for example, a healthcare flow where each step must be signed off: step 1 verify patient ID, step 2 confirm medication, step 3 check allergy list, step 4 approve prescription
- On mixed teams I often use Given/When/Then for the scenario summary and attach detailed steps as a sub-table for complex multi-step flows

Chapter 2: Test Design Techniques

You cannot test every possible input. These techniques answer: "Which inputs should I choose?" They have been taught since the 1970s and remain the backbone of test case design. Every interviewer expects you to know them.

2.1 Boundary Value Analysis (BVA)

Test at the edges of input ranges, where bugs cluster. Off-by-one errors are the most common numerical bug in software.

The Principle

For any input with a valid range, test six values:

Position	Value	Expected
Just below lower boundary	min - 1	Invalid
Lower boundary	min	Valid
Just above lower boundary	min + 1	Valid
Just below upper boundary	max - 1	Valid
Upper boundary	max	Valid
Just above upper boundary	max + 1	Invalid

Worked Example: Age Field (accepts 18-65)

Test Value	Expected	Rationale
17	Rejected	Just below minimum
18	Accepted	Lower boundary
19	Accepted	Just above minimum
64	Accepted	Just below maximum
65	Accepted	Upper boundary
66	Rejected	Just above maximum

Additional Boundaries You Should Always Consider

- **Zero**: Does the field accept 0? What about negative numbers?
- **Empty string vs null**: Different things in most systems
- **Maximum length**: For text fields — what happens at max, max + 1?
- **Data type limits**: Integer overflow at 2,147,483,647 (32-bit signed)
- **Date boundaries**: End of month (28/29/30/31), year boundaries, leap years (Feb 29)
- **Floating point precision**: $199.99, $200.00, $200.01

BVA Applied to Non-Numeric Fields

Field Type	Boundaries to Test
Text (max 100 chars)	0 chars (empty), 1 char, 99 chars, 100 chars, 101 chars
File upload (max 5MB)	0 bytes, 1 byte, 4.99MB, 5.00MB, 5.01MB
Date range (Jan 1 - Dec 31)	Dec 31 of previous year, Jan 1, Jan 2, Dec 30, Dec 31, Jan 1 of next year

continued on next page

Field Type	Boundaries to Test
Password (min 8 chars)	7 chars, 8 chars, 9 chars; also test at max length if one exists
Quantity (1-99)	0, 1, 2, 98, 99, 100

> **Pro Tip**: Always test with 0, empty, and null — even if the spec doesn't mention them. These are the three most common sources of unhandled errors.

2.2 Equivalence Partitioning (EP)

Divide inputs into groups (partitions) where all values behave the same. Test one representative from each partition.

The Principle

If the system treats all values in a range identically, testing one value represents the entire range. This reduces test count dramatically.

Worked Example: Age Field (18-65)

Partition	Range	Representative	Expected
Invalid low	< 18	10	Rejected
Valid	18-65	30	Accepted
Invalid high	> 65	70	Rejected

Three tests instead of exhaustive. If 30 works, 25 and 50 and 61 will also work (they're in the same partition).

EP for Non-Numeric Inputs

Input	Valid Partitions	Invalid Partitions
Email field	Valid format (user@domain.com)	Missing @, missing domain, empty, spaces only, double @@
File upload	Allowed formats (.jpg, .png, .pdf)	Disallowed (.exe, .bat), empty file, oversized
Country drop-down	Any country in the list	Empty selection (if required)
Search query	Alphanumeric	SQL injection payloads, XSS payloads, extremely long strings

Combining EP and BVA

Use them together for maximum efficiency:

1. **EP** identifies the partitions (invalid low, valid, invalid high)
2. **BVA** selects specific values at partition boundaries (17, 18, 65, 66)

For the age field, combining gives you: 17, 18, 30, 65, 66 — five tests that cover both partition representation and boundary precision.

2.3 Decision Tables

When business logic involves combinations of conditions, decision tables ensure complete coverage.

Building a Decision Table

1. List all conditions (inputs/states)
2. List all actions (outputs/behaviors)
3. Calculate combinations: 2 ^ n for n boolean conditions
4. Fill in expected action for each combination
5. Look for rules that can be merged

Worked Example: Shipping Rules

Condition	Rule 1	Rule 2	Rule 3	Rule 4
Customer is premium?	Y	Y	N	N
Order > $100?	Y	N	Y	N
Free shipping	**Yes**	**Yes**	**Yes**	**No**
Gift wrapping	**Yes**	**No**	**No**	**No**

This table surfaces a question: Rule 3 — a non-premium customer with a large order gets free shipping but no gift wrapping. Is that intentional? Decision tables reveal business logic gaps.

Worked Example: Loan Approval

Condition	R1	R2	R3	R4	R5	R6	R7	R8
Credit > 700?	Y	Y	Y	Y	N	N	N	N
Income > $50K?	Y	Y	N	N	Y	Y	N	N
Existing customer?	Y	N	Y	N	Y	N	Y	N
Approve	Yes	Yes	Yes	No	Yes	No	No	No
Interest rate	Low	Med	Med	—	Med	—	—	—

Eight rules, eight test cases. Complete combinatorial coverage of the business logic.

When Decision Tables Get Large

With 5 boolean conditions, you get 2 ^ 5 = 32 rules. Strategies for managing large tables:

- **Simplification**: If a condition doesn't affect the outcome in some cases, collapse those rules
- **Prioritization**: Test the most common/important rules first
- **Pairwise**: For very large tables, use pairwise testing (Section 2.5) to reduce combinations

2.4 State Transition Diagrams

For features with distinct states and transitions between them.

Drawing the Diagram

```
[Draft] --submit--> [Pending Review] --approve--> [Published]
                                     --reject---> [Draft]
[Published] --archive--> [Archived]
[Archived]  --restore--> [Draft]
```

Deriving Test Cases

Valid transitions (positive tests):

#	From	Action	To	Expected
1	Draft	Submit	Pending Review	Status changes; reviewer notified
2	Pending Review	Approve	Published	Article visible to public
3	Pending Review	Reject	Draft	Author receives rejection reason
4	Published	Archive	Archived	Removed from public view
5	Archived	Restore	Draft	Available for editing again

Invalid transitions (negative tests):

#	From	Attempted Action	Expected
6	Draft	Publish directly	Blocked — must go through review

continued on next page

#	From	Attempted Action	Expected
7	Archived	Publish directly	Blocked — must go through Draft first
8	Published	Edit	Blocked — must archive, then edit in Draft
9	Pending Review	Archive	Blocked — must be Published first

Transition loops:

#	Path	Purpose
10	Draft → Pending → Reject → Draft → Pending → Approve → Published	Rejection loop works correctly
11	Published → Archived → Draft → Pending → Published	Full lifecycle

State Transition Table Format

Current State	Event	Next State	Action
Draft	Submit	Pending Review	Send notification to reviewer
Pending Review	Approve	Published	Make visible to public
Pending Review	Reject	Draft	Send rejection reason to author
Published	Archive	Archived	Remove from public view
Archived	Restore	Draft	Return to author for editing

2.5 Pairwise Testing

When you have many parameters, testing all combinations is impractical. Pairwise ensures every pair of parameter values appears at least once.

Example

Testing a form with: Browser (Chrome, Firefox, Safari), OS (Windows, Mac, Linux), Language (EN, ES, FR).

Full combinatorial: 3 x 3 x 3 = 27 tests. Pairwise: ~9 tests cover every pair.

Test	Browser	OS	Language
1	Chrome	Windows	EN
2	Chrome	Mac	ES
3	Chrome	Linux	FR
4	Firefox	Windows	FR
5	Firefox	Mac	EN
6	Firefox	Linux	ES
7	Safari	Windows	ES
8	Safari	Mac	FR
9	Safari	Linux	EN

Every browser appears with every OS. Every browser with every language. Every OS with every language. Nine tests instead of 27.

Use tools like PICT (Microsoft), AllPairs, or online pairwise generators.

2.6 Choosing the Right Technique

Situation	Best Technique
Numeric input with a defined range	BVA + EP
Multiple input categories	EP

continued on next page

Situation	Best Technique
Business rules with condition combinations	Decision Tables
Workflow or lifecycle features	State Transition
Many parameters, few values each	Pairwise
Unknown territory, no clear requirements	Exploratory Testing (Chapter 3)

> **Pro Tip**: In practice, experienced QA engineers combine techniques. For one feature you might use EP to identify partitions, BVA to pick boundary values, and a decision table for the business rule combinations.

Key Takeaways — Chapter 2

- BVA catches off-by-one errors at range boundaries
- EP reduces test count by grouping equivalent inputs
- Decision tables ensure complete coverage of business logic combinations
- State transition diagrams verify valid paths and block invalid ones
- Pairwise testing handles multi-parameter scenarios efficiently
- Combine techniques for thorough coverage with minimal test count

Self-Assessment Quiz — Chapter 2

1. For a text field that accepts 1-50 characters, list the six BVA values you would test.
2. What are the three equivalence partitions for a “quantity” field that accepts 1-99?
3. Build a decision table for: “Free trial users can export 5 reports/month. Paid users have unlimited exports. Admins can export regardless of plan.” How many rules does your table have?
4. Draw a state transition diagram for a simple order system: New → Processing → Shipped → Delivered. Add a “Cancelled” state that can be reached from New or Processing but not Shipped or Delivered.

5. Explain when you would use pairwise testing instead of a full decision table.

Exercises — Chapter 2

Beginner: Apply BVA to a password field (minimum 8 characters, maximum 64 characters, must contain at least one digit). List all boundary values you would test.

Intermediate: Build a complete decision table for a movie ticket pricing system: Adult/Child, Matinee/Evening, Member/Non-Member. There are 8 rules. For each, determine the ticket price (make up reasonable prices). Then write Given/When/Then test cases for each rule.

Advanced: A document management system has these states: Draft, In Review, Approved, Published, Archived, Deleted. Draw the state transition diagram (decide which transitions are valid), create a state transition table, and write test cases for all valid transitions plus at least 5 invalid transitions.

Career Translation

Resume phrasing

- Applied boundary value analysis and equivalence partitioning to reduce test suite size by 60% while maintaining equivalent defect detection coverage
- Designed decision tables for complex business rule validation, uncovering 15+ logic gaps before development began
- Built state transition models for workflow features, identifying invalid transition paths that led to 8 critical bug discoveries
- Utilized pairwise testing to compress cross-platform test matrices from 200+ combinations to under 30 without losing pair coverage

Cover letter framing

I select test design techniques strategically based on the problem at hand — boundary value analysis for numeric fields, decision tables for business logic, state transitions for workflows. This deliberate approach means I find the bugs that matter without wasting cycles on redundant test cases.

Teams I work with consistently ship with fewer post-release defects because test coverage is designed, not guessed.

Interview framing

"I approach test design by first classifying the problem: Is this a range validation? I use BVA and EP. Is it a set of business rules? Decision table. Is it a workflow with lifecycle states? State transition diagram. I optimize for maximum defect detection with minimal test count — not by skipping scenarios, but by choosing representative values intelligently. Trade-offs include the upfront time to model the problem versus just writing tests, but that investment pays back every sprint when regression runs stay lean."

What not to say

- "I just test everything I can think of" — suggests no systematic technique selection
- "Boundary values are just min and max" — misses the six-value approach (min-1, min, min+1, max-1, max, max+1)
- "Decision tables are overkill for most features" — reveals unfamiliarity with how quickly business logic combinations grow
- "I test one value from each partition and call it done" — ignores combining EP with BVA for thorough coverage
- "State diagrams are more of a developer thing" — misunderstanding QA's role in modeling behavior

Interview Depth Check

Question 1

Prompt: A registration form has these fields: age (18-120), username (3-20 alphanumeric characters), and a country dropdown (200 countries). How would you decide which test design techniques to apply to each field, and how many test cases would you end up with?

What a strong answer should cover:

- Age: BVA (17, 18, 19, 119, 120, 121) combined with EP (invalid low, valid, invalid high)
- Username: BVA on length (2, 3, 4, 19, 20, 21 chars), EP for character types (valid alphanumeric, invalid special chars, empty, spaces)
- Country: EP (valid selection, no selection if required)
- Combine overlapping tests where possible to reduce total count
- Estimate a final count with reasoning

Example answer:

- For age I apply BVA: 6 boundary tests (17, 18, 19, 119, 120, 121) plus EP representatives for non-numeric input, empty, and negative numbers — roughly 9 tests
- For username I combine BVA on length (6 boundary values) with EP on character types (alphanumeric valid, special chars invalid, spaces, empty, Unicode) — roughly 10 tests
- For country I need 2 EP cases: one valid selection, one empty/no selection
- Some tests combine naturally (e.g., valid age + valid username + valid country in one happy-path case), so I estimate 15-20 focused test cases covering all technique outputs

Question 2

Prompt: You are testing a pricing engine with these rules: employees get 30% off, orders over $500 get free shipping, loyalty members get an extra 5% off, and discounts cannot exceed 50% total. Build a decision table approach.

What a strong answer should cover:

- Identify the 3 boolean conditions: employee Y/N, order >$500 Y/N, loyalty member Y/N
- 2^3 = 8 combinations
- For each combination, calculate the applicable discount and shipping
- Identify the cap rule (50% max) and which combinations trigger it

- Note that the 50% cap creates a testable boundary

Example answer:

- I identify three conditions (employee, order >$500, loyalty) giving 8 rules
- Rule 1: Employee + >$500 + Loyalty = 35% discount (30+5), free shipping — under cap
- Rule 8: Non-employee + <=$500 + Non-loyalty = 0% discount, standard shipping
- The critical test is the cap scenario: if other rules (e.g., a stacking promo) could push past 50%, I verify the cap applies
- I would also test boundary: exactly $500 (BVA on the shipping threshold) and discount at exactly 50% and 50.01%

Question 3

Prompt: An e-commerce order has these states: Cart, Placed, Paid, Shipped, Delivered, Returned, Cancelled. Some transitions have conditions (e.g., you can cancel only before shipment). How do you model this and what test cases do you derive?

What a strong answer should cover:

- Draw the state diagram with valid transitions and conditions
- Identify invalid transitions (e.g., Delivered -> Cart, Shipped -> Cancelled)
- Test all valid transitions as positive tests
- Test key invalid transitions as negative tests
- Test loops (e.g., Cart -> Placed -> Cancelled -> Cart -> Placed -> Paid)

Example answer:

- Valid transitions: Cart->Placed (checkout), Placed->Paid (payment confirmed), Paid->Shipped (fulfillment), Shipped->Delivered (carrier confirms), Delivered->Returned (return request), Placed->Cancelled (before payment), Paid->Cancelled (before shipment, triggers refund)

- Invalid transitions I would test: Shipped->Cancelled (too late), Delivered->Cancelled (too late), Cart->Shipped (skipping payment), Returned->Shipped (cannot re-ship a return)
- I would test the full lifecycle path and the rejection-loop path (Cart -> Placed -> Cancelled -> Cart -> Placed -> Paid -> Shipped -> Delivered)
- Each transition test verifies the status label updates, timestamps change, and side effects fire (emails, inventory adjustments)

Question 4

Prompt: Your team needs to test a web form across 4 browsers, 3 operating systems, 2 screen sizes, and 3 languages. Full combinatorial testing would require 72 configurations. How would you reduce this while maintaining confidence?

What a strong answer should cover:

- Pairwise testing to ensure every pair of values appears at least once
- Expected reduction from 72 to approximately 12-15 configurations
- Tools that generate pairwise sets (PICT, AllPairs)
- Risk-based selection: prioritize the most common user configurations
- Acknowledge pairwise misses higher-order interactions, but these are rare

Example answer:

- I would use pairwise testing to generate a set where every browser-OS pair, every browser-language pair, and every OS-screen pair appears at least once
- Using a tool like PICT, this typically reduces 72 combinations to about 12 configurations
- I would layer risk on top: if analytics show 70% of users are Chrome on Windows, I ensure that configuration appears in multiple language tests

- I accept the trade-off that pairwise misses specific three-way or four-way interactions (like "Chrome + Linux + mobile + French") but these are statistically rare for UI rendering issues

Question 5

Prompt: A colleague argues that equivalence partitioning is unnecessary because "if you test the boundaries, you cover everything." How do you respond?

What a strong answer should cover:

- BVA and EP are complementary, not redundant
- BVA catches off-by-one errors at edges; EP ensures representative coverage of each behavioral group
- EP catches bugs where an entire partition is mishandled (e.g., all negative numbers crash the system)
- EP also applies to non-numeric inputs where BVA may not apply (e.g., email format partitions)
- A concrete example where BVA alone would miss the defect

Example answer:

- BVA tests values at the edges, but a bug could exist in the middle of a valid range — for example, a discount engine that handles 0% and 100% correctly but miscalculates at 50% due to a rounding error in the formula
- EP ensures you pick a representative value from the valid partition that exercises the core logic, not just the boundary checks
- More importantly, EP applies to non-numeric domains where BVA is awkward — for instance, partitioning email inputs into "valid format," "missing @," "missing domain," and "empty" does not involve boundaries at all
- I use both together: EP tells me which groups to test, BVA tells me where within numeric groups to focus

PART II

EXPLORATORY TESTING

Chapter 3: Session-Based Test Management (SBTM)

Exploratory testing is not “clicking around.” It is simultaneous test design, execution, and learning — guided by a charter and bounded by time. SBTM, formalized by James Bach, brings structure and accountability to exploratory testing without killing its creative power.

3.1 What Is Exploratory Testing?

Exploratory testing is a style where the tester simultaneously designs tests, executes them, and learns about the system — all at once. Unlike scripted testing (following predefined steps), exploratory testing adapts in real time based on discoveries.

When Exploratory Testing Excels

- **New features with unclear requirements** — you learn the feature as you test it
- **Areas with high complexity** — too many paths for scripted tests to cover
- **After automation catches regressions** — exploration finds what scripts miss
- **Usability evaluation** — scripts cannot assess whether an interface "feels right"
- **Time-pressured releases** — finds critical bugs faster than writing scripted tests first

When Scripted Tests Are Better

- Regression testing across stable features
- Compliance testing with auditable evidence
- Data-driven testing with hundreds of input combinations

> **Pro Tip**: Exploratory and scripted testing are not competing approaches. They are complementary. Use scripted tests for what you know; use exploration for what you don't.

3.2 Charters

A charter is a mission statement for your testing session. It answers: "What am I testing, and why?"

The Charter Formula

Explore [target] with [resources/approach] to discover [information].

Component	Description	Example
Target	The feature or area to test	Checkout flow

continued on next page

Component	Description	Example
Resources/ Approach	Specific data, tools, or techniques	International addresses, edge-case payment methods
Information	What kind of issues you seek	Currency conversion errors, tax miscalculations

Good Charters

```
Explore the checkout flow with international addresses
to discover currency conversion and tax calculation issues.

Explore the file upload feature with edge-case formats (.svg, .webp, 0-byte files)
to discover handling gaps and silent failures.

Explore the admin user management panel under concurrent usage
to discover race conditions in role assignment.

Explore the search feature with special characters and Unicode input
to discover encoding, sanitization, or display issues.

Explore the notification system with rapid-fire events (10+ in 1 second)
to discover queuing, deduplication, or ordering issues.
```

Bad Charters

- "Test the login page" — too broad, what are you looking for?
- "Find bugs" — not a charter, it's a wish
- "Test everything in the settings module" — no focus, no time boundary

3.3 Sessions

A session is a focused, uninterrupted block of testing, typically 60-90 minutes.

Session Structure

Phase	Duration	Activity
Setup	5-10 min	Review charter, prepare environment, load test data
Exploration	40-70 min	Active testing, note-taking, bug filing
Wrap-up	5-10 min	Organize notes, draft bug reports, prepare for debrief

Session Rules

1. **No interruptions.** Close Slack, mute notifications. Treat it like deep work.
2. **Stay on charter.** If you discover something outside your charter, note it as a future charter.
3. **Take notes continuously.** If you stop noting, you stop doing exploratory testing and start doing random testing.
4. **Time-box strictly.** When time is up, stop. Schedule another session if needed.

> **Common Mistake**: Running "exploratory testing" without a charter or time box is just ad-hoc testing. The charter and time box are what make it professional and accountable.

3.4 Debriefs

After each session, conduct a 10-15 minute debrief:

1. **Share knowledge** — what did you learn about the system?
2. **Validate findings** — are the bugs real? Are the risks valid?
3. **Adjust strategy** — should the next session target a different area?

Debrief Questions

- What was your charter?
- What did you actually test?
- What did you find?
- What concerns you that you didn't have time to explore?

- What should the next session focus on?

3.5 Heuristics for Exploration

Heuristics are mental shortcuts that guide your exploration.

SFDPOT (San Francisco Depot) — James Bach

Letter	Focus	Questions to Ask
Structure	What is the product made of?	Modules, pages, components, APIs
Function	What does it do?	Features, operations, error handling
Data	What data does it process?	Inputs, outputs, stored data, data flow
Platform	What does it depend on?	OS, browser, network, hardware
Operations	How is it used?	Workflows, user personas, usage patterns
Time	How does behavior change over time?	Timeouts, concurrency, state transitions, scheduling

FEW HICCUPPS — Consistency Oracles

When you're unsure if something is a bug, check for consistency with:

Oracle	Question
Familiar problems	Does it have bugs you've seen in similar products?
Explainability	Can you explain the behavior to a user?
World	Does it match how the real world works?
History	Has this area had bugs before?
Image	Does it match the company's quality standards?

continued on next page

Oracle	Question
Comparable products	How do competitors handle this?
Claims	Does it match what the documentation says?
User expectations	Would a typical user be confused?
Product	Is it consistent with other parts of the same product?
Purpose	Does it fulfill its intended purpose?
Standards	Does it comply with relevant standards?

Key Takeaways — Chapter 3

- Exploratory testing is disciplined investigation, not random clicking
- SBTM provides structure through charters, time-boxed sessions, and debriefs
- Good charters follow: Explore [target] with [approach] to discover [information]
- Session notes are the evidence that distinguishes exploration from ad-hoc testing
- Heuristics like SFDPOT guide exploration without constraining it

Self-Assessment Quiz — Chapter 3

1. What three elements make up the SBTM framework?
2. Write a charter for testing a social media “share” button.
3. Why is a time box important for exploratory testing?
4. What is the difference between exploratory testing and ad-hoc testing?
5. Pick three letters from SFDPOT and explain what you would look for when testing a calendar application.

Exercises — Chapter 3

Beginner: Choose a website you use daily. Write three charters for different features using the formula. Do not execute yet — just write the charters.

Intermediate: Execute a 30-minute exploratory session on a public website. Set a timer. Take notes every 2-3 minutes. Write a complete session report (see Chapter 4 template).

Advanced: Plan a full exploratory testing campaign for an e-commerce checkout flow. Write 8-10 charters covering: happy path, edge cases, security, usability, performance, and cross-browser. Prioritize them by risk level and explain which you would execute first and why.

Career Translation

Resume phrasing

- Led session-based exploratory testing campaigns that discovered 30+ defects missed by scripted regression suites, including 5 critical severity issues
- Designed and executed exploratory testing charters targeting high-risk areas, reducing escaped defects to production by 25%
- Applied SFDPOT and FEW HICCUPPS heuristics to systematically explore new features, generating actionable session reports for stakeholder review
- Conducted structured debrief sessions that informed test strategy adjustments and identified 12 follow-up areas requiring deeper investigation

Cover letter framing

I bring rigor to exploratory testing through session-based management — every exploration has a charter, a time box, and a documented outcome. This approach finds the bugs that scripted tests miss: usability issues, unexpected state combinations, and edge cases nobody thought to script. My exploratory sessions consistently surface critical defects in areas the team considered low-risk.

Interview framing

"I approach exploratory testing as disciplined investigation, not ad-hoc clicking. I write a focused charter — 'Explore X with Y to discover Z' — set a 60-minute time box, and take continuous notes. I use heuristics like SFDPOT to guide what I look at: structure, function, data, platform, operations, and time. I optimize for learning speed: every minute in a session either confirms something works or reveals a new risk. Trade-offs include the overhead of documentation and debriefs, but that overhead is what makes exploration accountable and its findings actionable."

What not to say

- "Exploratory testing is just clicking around to see what breaks" — reveals no understanding of structure or methodology
- "I don't need a charter, I just test what feels important" — that is ad-hoc testing, not exploratory testing
- "I do exploratory testing when I have free time" — implies it is a lower-priority afterthought
- "I don't take notes during sessions because it slows me down" — notes are the evidence; without them, the work is invisible
- "Debriefs are a waste of time" — debriefs are where knowledge transfer and strategy adjustment happen

Interview Depth Check

Question 1

Prompt: You have two days before a major release and the PM asks you to do exploratory testing on a newly redesigned checkout flow. How do you structure your approach?

What a strong answer should cover:

- Create prioritized charters based on risk (payment first, then address validation, then edge cases)
- Define session length (60-90 minutes) and schedule sessions across the two days
- Use SFDPOT to ensure comprehensive coverage across different dimensions

- Plan for debriefs between sessions to adjust strategy based on findings
- Document everything for stakeholders who need release confidence

Example answer:

- I would write 6-8 charters ordered by risk: (1) payment processing with valid/invalid cards, (2) address validation with international formats, (3) discount and promo code edge cases, (4) concurrent sessions and timeout behavior, (5) cross-browser rendering, (6) accessibility
- I would schedule four 60-minute sessions on day 1 covering the top 4 charters, with 15-minute debriefs between each
- Day 2 sessions would address remaining charters plus follow-up charters generated from day 1 findings
- After each session I produce a report so the PM can see coverage and risk status in real time

Question 2

Prompt: During an exploratory session, you discover a potential security issue (user A can see user B's order details by changing the URL). Your charter was about UI layout testing. What do you do?

What a strong answer should cover:

- Note the finding immediately — do not ignore it because it is off-charter
- File a bug report with high severity immediately (security issue)
- Do not continue investigating deeply — stay on the current charter
- Add a follow-up charter specifically for authorization and access control testing
- Mention it in the debrief as a critical finding requiring dedicated investigation

Example answer:

- I would immediately note the finding with a timestamp and the exact URL manipulation I performed
- I would file a high-severity bug report right away — security issues cannot wait for the end of the session
- I would spend no more than 2-3 minutes confirming the issue is real (try a second user pair) and then return to my charter
- In my session report I would add a follow-up charter: "Explore order detail endpoints by manipulating URL parameters and API calls to discover authorization bypass vulnerabilities"
- At the debrief I would flag this as a potential systemic issue — if one endpoint lacks authorization checks, others might too

Question 3

Prompt: Your test lead says exploratory testing is not worth the investment because "we already have 2,000 automated regression tests." How do you make the case for exploratory testing?

What a strong answer should cover:

- Automated tests verify known scenarios; exploration finds unknown ones
- Automation cannot assess usability, confusing flows, or "feels wrong" observations
- Exploration adapts in real time; scripts follow the same path every time
- Statistics: exploratory testing often finds bugs in areas automation has 100% pass rate
- Propose a time-boxed pilot to demonstrate value

Example answer:

- I would acknowledge the strength of 2,000 automated tests for regression confidence, then explain the blind spot: those tests verify 2,000 scenarios the team already thought of — exploration finds the scenarios nobody predicted

- I would give concrete examples: automated tests will not tell you the checkout flow is confusing, that an error message is technically correct but unhelpful, or that a race condition occurs when two tabs submit simultaneously
- I would propose a pilot: two 60-minute exploratory sessions focused on the last three production incidents. If I find issues the automated suite missed, that demonstrates the complementary value
- SBTM provides the same accountability automation has: documented charters, session reports, and measurable coverage metrics

Question 4

Prompt: Describe how you would use the SFDPOT heuristic to explore a file upload feature you have never seen before.

What a strong answer should cover:

- Structure: What components exist (upload button, preview, progress bar, file list)
- Function: What does it do (upload, validate, preview, delete, download)
- Data: What data flows (file types, sizes, names, metadata, encoding)
- Platform: What does it depend on (browser, OS, network speed, server storage)
- Operations: How is it used (single upload, bulk upload, drag-and-drop, mobile)
- Time: How does behavior change over time (concurrent uploads, timeout, retry, storage limits)

Example answer:

- Structure: I would map the UI components — upload button, drag zone, progress indicator, file list, preview pane — and check each renders correctly
- Function: I test core operations — upload succeeds, validation rejects bad types, preview works, delete removes files, download retrieves them

- Data: I push data boundaries — 0-byte files, maximum size, Unicode filenames, executable files, corrupted files, duplicate names
- Platform: I try different browsers (Chrome, Firefox, Safari), slow network (throttled to 3G), and mobile (touch-based file picker)
- Operations: I test real-world usage patterns — uploading 20 files at once, uploading while another upload is in progress, drag-and-drop from desktop
- Time: I test temporal aspects — what happens if the network drops mid-upload, does a timeout message appear, can I retry a failed upload, what if I leave the page and return

Chapter 4: Session Reports and Note-Taking

Session reports are the tangible output of exploratory testing. Without them, exploratory testing is invisible work.

4.1 The Session Report Template

```
Charter:     [Your charter statement]
Tester:      [Your name]
Duration:    [Actual time spent]
Start:       [Date and time]
Environment: [Browser, OS, build/version, feature flags]

AREAS COVERED:
- [What you tested and brief result]
- [What you tested and brief result]

BUGS FOUND:
- BUG-XXXX: [One-line summary] (severity: high/medium/low)

QUESTIONS / RISKS:
- [Open questions about behavior or requirements]

FOLLOW-UP CHARTERS:
- [Charters for future sessions based on what you learned]
```

Complete Example

```
Charter:     Explore file upload with edge-case formats (.svg, .webp, 0-byte)
             to discover handling gaps and silent failures.
Tester:      Jane D.
Duration:    60 min
Start:       2025-03-15 14:00
Environment: Staging (build #4521), Chrome 122, macOS 14.3

AREAS COVERED:
- Uploaded .svg (200KB) — accepted, renders in preview correctly
- Uploaded .webp (150KB) — accepted, but thumbnail generation fails silently
- Uploaded 0-byte .txt — accepted, but download link returns HTTP 500
- Uploaded .gif animated (2MB) — accepted, preview shows first frame only
- Uploaded file with Unicode filename (□□□.png) — accepted, filename truncated

BUGS FOUND:
- BUG-1234: 0-byte file upload succeeds but download crashes (severity: high)
- BUG-1235: .webp thumbnail silently missing — no error (severity: medium)
- BUG-1236: Unicode filename truncated without notification (severity: low)

QUESTIONS / RISKS:
- What is the maximum file count per upload batch? No docs found.
- Is there server-side virus scanning? Could not confirm.
- Does the CDN cache handle .webp correctly?

FOLLOW-UP CHARTERS:
- Explore bulk upload (10+ files) to discover performance issues
- Explore upload with slow/interrupted network to discover retry behavior
- Explore virus-infected test files to discover scanning behavior
```

4.2 Note-Taking Strategies

Taking notes while actively testing is the hardest part of SBTM.

The Timestamp Approach

```
14:00 - Starting. Environment verified. Opening upload page.
14:03 - Uploaded test.svg (200KB). Accepted. Preview renders correctly.
14:05 - Uploaded test.webp (150KB). Accepted. Preview area shows blank.
14:07 - Checking network tab. No errors on upload. Thumbnail endpoint 404.
14:10 - Filing bug for .webp thumbnail issue.
14:15 - Trying 0-byte file. Created empty.txt (0 bytes).
14:17 - Upload succeeds (unexpected). Download link appears.
14:18 - Clicking download. Server returns 500. Filing bug.
```

The OQB Approach (Observation / Question / Bug)

```
[O] .svg upload works correctly with preview
[O] .webp upload accepted but thumbnail area is blank
[Q] Is .webp thumbnail generation a separate service?
[B] Download endpoint returns 500 for 0-byte files
[O] Error message for .exe rejection is user-friendly
[Q] What file types are on the allowlist?
```

4.3 Aggregating Reports

Track coverage across sessions:

Feature Area	Sessions	Bugs Found	Risk Level	More Testing?
File upload	3	5	High	Yes — bulk upload untested
User profiles	2	1	Low	No — good coverage
Checkout flow	1	3	High	Yes — international payments untested
Admin panel	0	0	Unknown	Yes — not explored at all

4.4 From Exploration to Other Activities

The virtuous cycle:

1. **Exploration** → **Bug report** → Developer fixes → **Automated regression test** → Exploration continues finding new things
2. **Exploration** → **New test case** → Added to scripted regression suite
3. **Exploration** → **Question** → Brought to product owner → Becomes documented requirement → Test case written

Key Takeaways — Chapter 4

- Session reports make exploratory testing visible and accountable
- Every report needs: charter, areas covered, bugs, questions, follow-ups

- Take notes during the session — memory loses what real-time notes capture
- Aggregate reports to track coverage and identify untested areas
- Feed exploration findings into automation, test cases, and requirements

Exercises — Chapter 4

Beginner: Read the example session report above and identify: How many bugs were found? What severity were they? What would the tester do next?

Intermediate: Execute two 30-minute exploratory sessions on different features of a public website. Write session reports for both. Then create an aggregation table showing coverage.

Advanced: Run three 45-minute sessions on the same application over three days, each with a different charter. After the third session, write a summary report for a test lead that includes: total bugs found, risk assessment, coverage gaps, and recommended next steps.

Career Translation

Resume phrasing

- Produced structured session reports for every exploratory testing cycle, providing stakeholders with evidence-based coverage metrics and risk assessments
- Aggregated exploratory testing data across 50+ sessions to identify under-tested feature areas, resulting in targeted test campaigns that caught 18 pre-release defects
- Introduced OQB (Observation/Question/Bug) note-taking methodology to the QA team, improving session report quality and reducing post-session write-up time by 30%
- Created coverage dashboards from session report aggregation, enabling data-driven test planning and resource allocation decisions

Cover letter framing

I believe that undocumented testing is invisible testing. My session reports transform exploratory work into actionable intelligence: what was covered, what was found, what remains unknown. Stakeholders and managers receive clear aggregation tables showing coverage gaps and risk levels, enabling informed go/no-go decisions without micromanaging the testing process.

Interview framing

"I approach session documentation as the bridge between doing the testing and proving the testing happened. I take notes in real time using timestamped entries or the OQB method — every observation, question, and bug is captured as I go. I optimize for two audiences: my future self (who needs to pick up where I left off) and my test lead (who needs coverage visibility). Trade-offs include the overhead of real-time note-taking, which can slow exploration by 10-15%, but the alternative — reconstructing findings from memory — loses critical details."

What not to say

- "I don't write reports for exploratory testing because it's informal" — makes exploration invisible and unaccountable
- "I just file bugs and skip the session report" — loses context about what was covered, questions raised, and follow-up charters
- "I write up my notes after the session from memory" — memory is unreliable; real-time capture is essential
- "Aggregation tables are too much overhead for my team" — without aggregation, you cannot identify coverage gaps
- "I track coverage in my head" — unscalable and not verifiable by stakeholders

Interview Depth Check

Question 1

Prompt: You completed five exploratory sessions over two days. Your test lead asks: "How do I know what's been tested and what hasn't?" How do you answer, and what artifacts do you provide?

What a strong answer should cover:

- Present the aggregation table showing feature areas, sessions run, bugs found, and risk levels
- Highlight areas with zero sessions (untested) and areas with high bug density (risky)
- Provide the individual session reports as backup detail
- Recommend next sessions based on the gaps visible in the aggregation

Example answer:

- I would present an aggregation table with columns: Feature Area, Sessions Completed, Bugs Found, Risk Level, and Recommendation
- For example: "File upload: 2 sessions, 5 bugs, High risk, Needs more testing" vs. "User profile: 1 session, 0 bugs, Low risk, Adequate"
- Areas with zero sessions are highlighted as "Unknown risk — not yet explored"
- I would attach all five session reports so the lead can drill into specifics and validate that sessions had focused charters, not overlapping coverage
- Based on gaps, I would propose the next 3 charters and estimated time needed

Question 2

Prompt: A tester on your team writes session reports that just say "Tested the search feature. Found 2 bugs." How do you coach them to write better reports?

What a strong answer should cover:

- Explain the purpose of session reports (accountability, knowledge transfer, coverage tracking)
- Walk them through the template: charter, areas covered, bugs found, questions, follow-ups
- Show the difference between a weak report and a strong report side by side

- Suggest real-time note-taking methods (timestamp or OQB) to capture details during the session

Example answer:

- I would sit with them and compare their report to a complete example, pointing out what is missing: specific areas covered, questions raised, follow-up charters
- I would introduce the OQB method as a lightweight way to capture notes during testing without disrupting flow
- I would explain the "who benefits" angle: their future self benefits when they need to continue testing that area; the test lead benefits by seeing coverage; developers benefit by understanding the context around filed bugs
- I would suggest they start with the timestamp approach (write one line every 2-3 minutes) and graduate to richer reports once the habit is formed

Question 3

Prompt: You ran an exploratory session and found zero bugs. Does that mean the session was wasted? How do you present this to stakeholders?

What a strong answer should cover:

- Zero bugs is a valid and valuable outcome — it increases confidence in that area
- The session still generated information: areas covered, questions raised, follow-up charters
- Absence of defects in a high-risk area is significant and worth reporting
- The report should document what was tested thoroughly so stakeholders can see the coverage

Example answer:

- A zero-bug session is not wasted — it is evidence of stability in the area tested

- I would present the session report showing everything that was covered: "I explored the payment flow with expired cards, invalid CVVs, edge-case amounts, and concurrent submissions. All scenarios handled correctly."
- If the area was high-risk, zero bugs after thorough exploration is a confidence signal that supports a release decision
- The session may still have generated questions ("Is there rate limiting on payment attempts?") and follow-up charters that add value beyond bug counts

Question 4

Prompt: Your session notes reveal you spent 40 of 60 minutes on an area outside your charter because you found something interesting. How do you handle this in your report, and what would you do differently next time?

What a strong answer should cover:

- Honestly report what happened — do not fabricate coverage of the original charter
- Document findings from the off-charter investigation as valuable but unplanned
- Create a follow-up charter for the interesting area and a new session for the original charter
- Reflect on session discipline — staying on charter is a skill that improves with practice

Example answer:

- In my report, I would transparently note: "Deviated from charter at 14:15 due to discovering [interesting finding]. Spent 40 minutes investigating. Original charter coverage: partial — only X and Y tested, Z not reached."
- I would file bugs for whatever I found during the deviation and create a follow-up charter to investigate that area properly
- I would schedule a new session to complete the original charter

- Going forward, when I discover something off-charter, I write a one-line note and a follow-up charter, then force myself back to the current mission — the discipline is noting it and returning, not ignoring it or chasing it

PART III

BUG REPORTING

Chapter 5: Writing Effective Bug Reports

A bug report is a persuasive document. Its job is to make the developer reproduce the issue in under two minutes and understand its impact immediately. A great bug report gets fixed in hours. A vague one sits in the backlog for weeks.

5.1 The Five Essential Fields

Every bug report must include:

1. Environment

OS, browser/version, device, build number, feature flags. The "where."

```
Environment: Chrome 122, macOS 14.3, staging, build #4521
Feature flags: new_checkout_flow=true, dark_mode=false
```

2. Steps to Reproduce

Numbered, minimal. Strip anything not required to trigger the bug.

```
Steps to Reproduce:
1. Navigate to https://staging.example.com/login
2. Enter email: user+tag@test.com
3. Enter password: ValidPass123!
4. Click "Sign In"
```

3. Expected Result

What should happen, based on the spec or common sense.

```
Expected: User is authenticated and redirected to /dashboard.
```

4. Actual Result

What actually happened. Include evidence.

```
Actual: Server returns HTTP 500. Response body: {"error": "invalid_parameter"}.
No client-side errors in console. POST /auth/login fails with 500.
```

5. Severity and Priority

Both. Always. (See Chapter 6 for the full framework.)

5.2 The Bug Report Title

The title is the most important line. In triage meetings, the team sees only titles.

Formula: **[What happens] when [condition/trigger]**

Good Titles	Bad Titles
"Login returns 500 when email contains '+' character"	"Bug in login"
"Cart total shows $0.00 after applying 100% discount"	"URGENT: production issue!!!"

continued on next page

Good Titles	Bad Titles
“Profile image upload silently fails for files > 5MB”	“Test case TC-AUTH-007 failed”
“Search results infinite spinner on empty query”	“Search broken”

5.3 Attachments That Accelerate Fixes

Attachment	When to Use	Impact
Annotated screenshot	UI bugs, layout issues	Developer sees exactly what’s wrong
Video (< 60 seconds)	Complex multi-step flows	Developer sees the full reproduction
HAR file	API/network issues	Developer sees every request/response
Console log	JavaScript errors	Developer sees the exact error + stack trace
cURL command	API bugs	Developer reproduces without touching UI

cURL Example

```
curl -X POST https://staging.example.com/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email": "user+tag@test.com", "password": "ValidPass123!"}'
# Returns: HTTP 500 {"error": "invalid_parameter"}
```

5.4 Isolation: Narrowing Down Before Filing

Before filing, try to narrow the scope:

1. **Browser-specific?** Try Chrome, Firefox, Safari
2. **Environment-specific?** Try staging and production

3. **Data-specific?** Try different inputs
4. **Timing-specific?** Every time, or intermittent?
5. **Permission-specific?** All users, or certain roles?

Reporting Intermittent Bugs

- State the reproduction rate: "Reproduces ~3 out of 10 attempts"
- Describe what did NOT trigger it
- Note patterns: "More frequent right after deployment"
- Include timestamps so the team can check server logs

> **Pro Tip**: A bug report that says "fails with user+tag@test.com but works with user@test.com" saves the developer hours of investigation. The narrower you can make it, the faster the fix.

5.5 The Rapid Bug Report (Under Pressure)

For production incidents, use this minimal template:

```
Title: [What breaks] when [trigger]
Environment: Production, [timestamp]
Steps: [minimal path]
Actual: [what happened, error codes]
Impact: [who is affected, how many users]
```

Add details later. Priority is getting info to the right people fast.

5.6 Bug Report Examples

Example 1: API Bug (Good)

```
Title: Login returns HTTP 500 when email contains '+' character
Severity: High | Priority: P1

Environment: Chrome 122, macOS 14.3, staging build #4521

Steps to Reproduce:
1. Navigate to https://staging.example.com/login
2. Enter email: user+tag@test.com
3. Enter password: ValidPass123!
4. Click "Sign In"

Expected: User is authenticated and redirected to /dashboard.

Actual: Server returns HTTP 500. Response body: {"error": "invalid_parameter"}.
No client-side errors. Network tab shows POST /auth/login returns 500.

Notes: Works correctly with user@test.com (no + character).
Also fails with user+anything@test.com. Suspect the + is not URL-encoded
before being sent to the backend.

Attachment: har-login-plus-character.har
```

Example 2: UI Bug (Good)

```
Title: Cart total shows $0.00 after applying 100% discount code
Severity: High | Priority: P2

Environment: Firefox 121, Windows 11, staging build #4518

Steps to Reproduce:
1. Add any item to cart (used "Blue Widget" at $29.99)
2. Navigate to cart
3. Enter discount code: FULLOFF (100% discount)
4. Click "Apply"

Expected: Cart total shows $0.00. "Proceed to Checkout" button is active.
Order summary shows item at $29.99 with -$29.99 discount line.

Actual: Cart total shows $0.00 correctly, but "Proceed to Checkout" button
is grayed out. Tooltip says "Cart is empty." The system interprets $0.00
total as an empty cart.

Attachment: screenshot-cart-zero-grayed-out.png (annotated)
```

Example 3: Intermittent Bug (Good)

```
Title: Dashboard fails to load (~30% of page refreshes) with blank white screen
Severity: High | Priority: P1

Environment: Chrome 122, macOS 14.3, production
Reproduction rate: ~3 out of 10 page refreshes

Steps to Reproduce:
1. Log in to https://app.example.com
2. Navigate to /dashboard
3. Refresh the page (Cmd+R)
4. Repeat 10 times

Expected: Dashboard loads with charts and metrics every time.

Actual: Approximately 3 out of 10 times, the page shows a blank white screen.
Console error: "TypeError: Cannot read properties of undefined (reading 'map')"
at DashboardWidget.jsx:142

Notes:
- Occurs on both Chrome and Firefox
- Does NOT occur on Safari (tested 20 refreshes, 0 failures)
- More frequent with many browser tabs open (memory pressure?)
- First observed after build #4520 deployed on 2025-03-14

Timestamps of occurrences:
- 2025-03-15 09:12:34 UTC
- 2025-03-15 09:12:51 UTC
- 2025-03-15 09:13:22 UTC

Attachment: console-dashboard-error.png
```

Key Takeaways — Chapter 5

- A bug report must enable reproduction in under two minutes
- Include all five fields: environment, steps, expected, actual, severity/priority
- Title formula: “[What happens] when [condition]”
- Attach evidence: screenshots, HAR files, console logs, cURL commands
- Isolate before filing — narrow browser, environment, data, permissions
- Respond quickly to developer questions

Self-Assessment Quiz — Chapter 5

1. What are the five essential fields of a bug report?
2. Why is "Login broken" a bad bug report title? Write a better one for the same issue.
3. What is a HAR file and when would you attach one to a bug report?
4. You found a bug that happens only intermittently. What extra information should your bug report include?
5. A developer comments "Cannot reproduce" on your bug. What do you do?

Exercises — Chapter 5

Beginner: Find a real bug on any public website (accessibility issues count — check keyboard navigation). Write a complete bug report with all five fields.

Intermediate: Visit a public website and intentionally try to break it (empty form submissions, special characters, extreme input lengths, rapid clicks). Write bug reports for any three issues you find.

Advanced: Have a colleague test a public website and write a bug report. Then try to reproduce from their report alone without asking questions. Score the report: could you reproduce in under 2 minutes? If not, rewrite it and explain what was missing.

Career Translation

Resume phrasing

- Authored detailed bug reports with isolation analysis and evidence attachments, achieving a 90% first-attempt reproduction rate by developers and reducing average fix turnaround from 5 days to 1.5 days
- Established bug report quality standards and templates adopted across a 15-person engineering team, reducing "cannot reproduce" responses by 60%
- Isolated root causes before filing — narrowing browser, environment, data, and permission scope — saving an estimated 4+ developer hours per critical bug

- Contributed HAR captures, annotated screenshots, and cURL reproduction commands with every API-related defect report

Cover letter framing

I write bug reports as persuasive documents designed for speed. Developers reproduce my bugs in under two minutes because every report includes minimal reproduction steps, isolated scope, evidence attachments, and clear expected-versus-actual comparison. This approach accelerates fix cycles and builds trust between QA and engineering — developers prioritize my reports because they know the information is complete and actionable.

Interview framing

"I approach bug reporting as a communication challenge, not just a documentation task. My goal is for the developer to reproduce the issue in under two minutes without asking me a single question. I optimize for minimal reproduction steps — stripping away anything not required to trigger the bug — and always include environment details, evidence, and isolation notes. Trade-offs include the time spent narrowing down before filing versus filing immediately. I invest 10-15 minutes in isolation because it saves hours of developer investigation and avoids the dreaded 'cannot reproduce' loop."

What not to say

- "I just describe what happened and let the developer figure it out" — shifts the burden of investigation to the wrong person
- "I mark everything as Critical to get attention" — destroys credibility and clogs triage
- "Screenshots are usually enough" — misses HAR files, console logs, and cURL commands that accelerate API bug fixes
- "If it's intermittent, I just note that it sometimes fails" — no reproduction rate, no timestamps, no patterns
- "The developer should be able to reproduce any bug from the title alone" — titles are summaries, not reproduction instructions

Interview Depth Check

Question 1

Prompt: You file a bug report and the developer responds "Cannot reproduce." Walk me through exactly what you do next.

What a strong answer should cover:

- Verify you can still reproduce it yourself
- Compare environments: your browser/OS/version vs. the developer's
- Check for differences in test data, feature flags, and user permissions
- Offer to pair with the developer for live reproduction
- Add more detail to the report: exact timestamps, network conditions, reproduction rate

Example answer:

- First, I re-test to confirm I can still reproduce. If I cannot, I note it as intermittent and add reproduction rate data
- If I can reproduce, I compare environments — I check my browser version, OS, feature flags, and test data against what the developer used
- I add a cURL command or HAR file so the developer can reproduce the exact request without relying on UI interaction
- If the gap persists, I offer a 15-minute screen share where I reproduce live while the developer watches and checks server logs simultaneously
- I update the bug report with all new findings from this investigation, so the next person has a complete picture

Question 2

Prompt: You discover a production bug at 11 PM on a Friday that affects checkout for approximately 5% of users. Walk me through your immediate actions and the bug report you write.

What a strong answer should cover:

- Use the rapid bug report template — speed over perfection
- Notify the on-call engineer immediately (Slack, PagerDuty, phone)
- Include: what breaks, who is affected, approximate scope, timestamps
- Attach evidence: error codes, screenshots, reproduction steps
- Follow up with a detailed report once the immediate crisis is handled

Example answer:

- I immediately file a rapid bug report: "Checkout returns 500 for users with gift card + credit card split payment. ~5% of checkout attempts. Production. First observed 2025-03-14 23:05 UTC."
- I alert the on-call engineer via the team's incident channel with a link to the bug report
- I include the HTTP 500 response body, the specific payment combination that triggers it, and 3 timestamps of confirmed occurrences
- I note what works: single payment method checkouts succeed, so the issue is specific to split payments
- Once the incident is handled, I update the report with full isolation details, root cause if known, and affected user count

Question 3

Prompt: A junior QA engineer writes this bug title: "URGENT!!! Something wrong with the search page." How do you coach them to write a better title, and what would a good title look like?

What a strong answer should cover:

- Explain the title formula: "[What happens] when [condition/trigger]"
- Remove emotional language (URGENT, exclamation marks) — severity/priority fields handle urgency
- Be specific about the observable behavior
- Include the trigger or condition that causes the issue

Example answer:

- I would explain that triage meetings show only titles — the team needs to understand the bug from the title alone
- I would walk through the formula: "[Observable behavior] when [trigger condition]"
- I would ask: "What specifically happens on the search page? Does it crash? Show wrong results? Take too long?"
- If the answer is "search returns zero results when query contains an apostrophe," the title becomes: "Search returns zero results when query contains apostrophe character"
- I would point out that "URGENT" belongs in the priority field (P1), not in the title, and that exclamation marks reduce rather than increase credibility

Question 4

Prompt: You found a bug where the user profile page displays another user's email address for 0.5 seconds before loading the correct one. It is a minor visual flash, but it exposes personal data. How do you write this bug report, and what severity/priority do you assign?

What a strong answer should cover:

- Severity: High or Critical (PII exposure, even briefly)
- Priority: P1 (privacy/security issue, regulatory risk)
- The report must emphasize the security dimension, not just the visual glitch
- Include evidence: video showing the flash, network tab showing the wrong data being loaded
- Note GDPR/privacy implications in the report

Example answer:

- Title: "Profile page briefly displays another user's email address before loading correct data"
- Severity: Critical — personal data exposure (even momentary) is a privacy violation

- Priority: P1 — this is a data leak with potential GDPR and regulatory implications
- Steps would be minimal: log in as User A, navigate to /profile, observe the 0.5-second flash of User B's email
- I would attach a screen recording showing the flash, a screenshot of the network tab showing the initial API response containing the wrong user's data
- Notes section: "This appears to be a caching issue — the previous user's profile data is rendered before the correct data loads. Even though the flash is brief, automated scrapers or screen recording tools could capture the exposed email. Recommend treating as a security incident."

Question 5

Prompt: You are testing an API endpoint and discover that sending a malformed JSON payload returns a 500 Internal Server Error instead of a 400 Bad Request. Is this worth filing? How would you write the report?

What a strong answer should cover:

- Yes, it is worth filing — 500 errors indicate unhandled exceptions on the server
- Unhandled exceptions can leak stack traces, expose internal architecture, and indicate missing input validation
- Severity: Medium (no data loss, but poor error handling and potential information disclosure)
- The report should include the exact cURL command for instant reproduction

Example answer:

- Absolutely worth filing — a 500 response means the server crashed on unexpected input instead of validating and rejecting it gracefully
- Title: "POST /api/v1/users returns 500 instead of 400 when request body contains malformed JSON"
- Severity: Medium — no data loss, but the 500 response may include stack traces that reveal internal architecture to attackers

- I would include the cURL command: `curl -X POST /api/v1/users -H "Content-Type: application/json" -d '{"name": "test",'` (note the trailing comma making it invalid JSON)
- Expected: HTTP 400 with error "Invalid JSON in request body." Actual: HTTP 500 with stack trace including file paths and database connection strings
- Notes: "This pattern suggests the API lacks a global input validation middleware. Recommend checking all endpoints for similar behavior."

Chapter 6: Severity vs Priority

These are independent axes. A bug can be high severity but low priority, or low severity but high priority. Confusing them is one of the most common mistakes in QA.

6.1 Definitions

Severity = Technical Impact (QA assesses)

Level	Definition	Example
Critical	Crash, data loss, security breach, complete failure	App crashes on checkout for all users
High	Major feature broken, no workaround	Payment fails for credit cards (PayPal works)
Medium	Feature partially broken, workaround exists	Search returns results but sorting doesn't work
Low	Cosmetic, minor inconvenience	Button alignment off by 2px on one page

Priority = Business Urgency (Product/management decides)

Level	Definition	Example
P0	Fix immediately, drop everything	Production is down
P1	Fix within 24 hours	Critical flow broken for subset of users
P2	Fix this sprint	Important but not blocking
P3	Fix eventually	Nice to have, low user impact

6.2 The Two-Axis Matrix

	High Priority	Low Priority
High Severity	App crashes on checkout — fix now	App crashes on 10,000-char name — rare, fix next sprint
Low Severity	CEO's name misspelled on homepage — cosmetic but urgent	Footer link color slightly off-brand — fix eventually

Why This Matters

A QA engineer who marks every bug as "Critical / P0" will be ignored. When everything is critical, nothing is critical. Accurate severity/priority classification builds trust with the development team.

6.3 Who Decides What

- **Severity**: QA assesses (factual observation about technical impact)
- **Priority**: Product owner sets (business decision considering users affected, deadlines, workarounds, opportunity cost)
- **When they disagree**: QA advocates for the fix but respects the prioritization — unless it involves data loss, security, or compliance (then escalate)

6.4 Common Mistakes

> **Common Mistake #1: Inflating Severity**
> Marking every bug "Critical" to get attention. Result: the team stops trusting severity ratings.
> **Fix**: Be honest. A cosmetic bug is Low severity even if it annoys you personally.

> **Common Mistake #2: Using a Single Scale**
> Some teams use only "High/Medium/Low" for both dimensions. This forces trade-offs that obscure information.
> **Fix**: Insist on separate fields. If your tool doesn't support it, add a custom field.

> **Common Mistake #3: Never Updating Priority**
> A P3 bug from three months ago may now be P1 because a major customer complained.
> **Fix**: Reassess priority during triage meetings.

6.5 Triage Meetings

Bug triage is where QA presents findings and the team assigns priority:

1. QA presents the bug with severity
2. Product owner assesses business impact → sets priority
3. Engineering estimates fix effort
4. Team decides: fix now, this sprint, or backlog

Triage Questions

- Who is affected? All users, a subset, internal only?
- Is there a workaround?
- What is the blast radius? One feature or the entire app?
- Is this a regression? (Regressions get higher priority — they broke something that worked)
- Security or compliance implications? (Automatic priority escalation)

Key Takeaways — Chapter 6

- Severity is technical impact (QA). Priority is business urgency (product/management).
- Keep them as separate fields — never collapse into one scale
- High severity ≠ high priority, and vice versa
- Accurate classification builds trust; inflation destroys it
- Triage meetings are where severity meets priority

Exercises — Chapter 6

Beginner: For each bug below, assign severity AND priority, and justify: 1. The "Forgot Password" email link points to the wrong URL 2. Admin dashboard shows wrong time zone for log timestamps 3. SQL injection vulnerability in the search bar 4. Mobile app crashes when rotating screen during video playback 5. Company logo on login page is the old version from a 2023 rebrand

Intermediate: Create a severity/priority matrix for your team with definitions, examples, and response time expectations for each combination. Make it something you could present at a team meeting.

Advanced: Review 10 bugs in any open-source project's issue tracker (GitHub Issues). For each, assess whether the severity and priority labels seem appropriate. Write a brief analysis of any you would re-classify and why.

Career Translation

Resume phrasing

- Established severity/priority classification framework adopted across 3 product teams, reducing triage meeting time by 35% through consistent, calibrated assessments
- Accurately classified 200+ defects using independent severity and priority axes, earning a reputation for trustworthy assessments that accelerated fix prioritization
- Introduced two-axis severity/priority matrix that eliminated the "everything is Critical" pattern, enabling the team to focus developer time on genuinely urgent issues

- Participated in weekly bug triage meetings, presenting defect severity assessments with business impact context that informed sprint planning decisions

Cover letter framing

I treat defect classification as a trust-building exercise. By consistently applying accurate severity and priority ratings — resisting the temptation to inflate — I have built credibility with development teams who know my "Critical" truly means critical. This discipline ensures triage meetings are productive: the team spends time deciding what to fix, not debating whether the classification is accurate.

Interview framing

"I approach severity and priority as two independent questions: 'How bad is this technically?' and 'How urgent is the business need to fix it?' I assess severity factually — a crash is Critical, a cosmetic misalignment is Low — and then work with the product owner to determine priority based on user impact, workarounds, and deadlines. I optimize for calibration: if I rate 50% of bugs as Critical, I have lost credibility. Trade-offs include occasionally under-rating severity when I am uncertain about blast radius, so I document my reasoning and invite the team to challenge my assessment at triage."

What not to say

- "I mark everything as Critical so developers pay attention" — this destroys the credibility of the entire classification system
- "Severity and priority are basically the same thing" — they are independent axes and conflating them loses information
- "I let the developer decide the priority" — priority is a business decision, not a technical one; QA provides severity, product sets priority
- "Low severity bugs aren't worth filing" — low severity bugs still need tracking; some become high priority due to business context
- "I don't attend triage meetings because they're a developer thing" — QA's presence at triage is essential for severity defense and context

Interview Depth Check

Question 1

Prompt: Give me an example of a bug that is high severity but low priority, and one that is low severity but high priority. Explain the reasoning for each classification.

What a strong answer should cover:

- High severity / low priority: technically severe but rare or affects few users
- Low severity / high priority: minor defect but business-critical context
- Clear separation of technical impact from business urgency
- Demonstrate understanding that these are independent axes

Example answer:

- High severity, low priority: The application crashes when a user enters a 10,000-character name in the registration field. Technically it is a crash (Critical severity), but in practice almost no real user will ever enter a 10,000-character name. Priority is P3 — fix eventually, because the realistic user impact is near zero
- Low severity, high priority: The company CEO's name is misspelled on the public homepage. Technically it is a cosmetic text error (Low severity), but the CEO noticed it before a press event and wants it fixed immediately. Priority is P0 — fix now, because of the reputational context
- The key is that severity is objective and technical, while priority is subjective and contextual

Question 2

Prompt: A product manager sets a security vulnerability (SQL injection in search) to P3 because "only 2% of users use the search feature." How do you respond?

What a strong answer should cover:

- SQL injection is Critical severity regardless of usage frequency
- Security vulnerabilities warrant priority escalation — compliance and legal risk
- Usage frequency is irrelevant for security; an attacker only needs one endpoint
- Escalate respectfully but firmly, citing security policy and regulatory requirements

Example answer:

- I would respectfully push back by explaining that security vulnerabilities are not assessed by usage frequency — an attacker does not need 2% of users to exploit SQL injection; they need one request
- I would cite the company's security policy (or industry best practices if no policy exists) that security vulnerabilities are automatically P1 or higher
- I would explain the blast radius: SQL injection could expose the entire database, not just the search feature — customer data, passwords, payment information
- If the PM still disagrees, I would escalate to the security team or engineering leadership, documenting my concern in writing
- I would frame it as risk management: "The cost of fixing this now is hours of developer time; the cost of a breach is legal liability, reputation damage, and regulatory fines"

Question 3

Prompt: During triage, a developer argues that a bug you rated as High severity is actually Medium because "the user can just refresh the page as a workaround." How do you evaluate their argument?

What a strong answer should cover:

- Workarounds affect priority (business urgency), not severity (technical impact)
- The presence of a workaround may lower priority but does not change the technical severity

- Evaluate whether the workaround is discoverable by the user without assistance
- The bug still exists; the workaround just reduces the user's frustration

Example answer:

- I would clarify the distinction: the workaround affects priority (since users can get unblocked), but the technical severity remains the same — the feature is still broken
- I would ask: "Will the user know to refresh? Is there an error message guiding them, or do they see a blank screen with no indication of what to do?"
- If the bug causes data loss, incorrect calculations, or security issues, a refresh workaround does not reduce severity at all
- I would be open to lowering priority from P1 to P2 if the workaround is reliable and discoverable, but I would not change the severity classification
- The bug report should document the workaround so that support teams can assist users while the fix is developed

Question 4

Prompt: Your bug backlog has 150 open bugs. The engineering lead asks you to help clean it up. How do you approach re-triaging 150 bugs efficiently?

What a strong answer should cover:

- Sort by age first — old bugs may no longer be relevant
- Batch by feature area to re-evaluate in context
- Close bugs for features that have been redesigned or removed
- Re-assess priority based on current business context (not original filing date)
- Timebox the effort and prioritize the re-triage itself

Example answer:

- I would sort all 150 bugs by last-modified date and quickly scan for stale bugs (6+ months with no activity) — these are candidates for closure or re-verification
- I would batch the remaining bugs by feature area and check each against the current product: has the feature been redesigned, removed, or already fixed?
- For surviving bugs, I would re-assess priority in today's context — a P2 from 6 months ago may now be P3 because the feature was deprecated, or P1 because a major customer complained
- I would propose a timebox: spend 2 hours on the first pass (close obvious stale bugs), then a 1-hour triage meeting for the remaining ambiguous ones
- After cleanup, I would recommend a monthly hygiene pass of 30 minutes to prevent the backlog from growing stale again

PART IV

TEST PLANNING

Chapter 7: Risk-Based Testing

You will never have time to test everything. Risk-based testing ensures your limited effort is concentrated where it prevents the most damage.

7.1 The Risk Equation

Risk = Probability of failure x Impact of failure

Risk Assessment Matrix

Feature Area	Likelihood	Impact	Risk Level	Test Depth
Payment processing	Medium	Critical	**High**	Full regression, edge cases, security
User profile editing	Low	Low	**Low**	Happy path, one negative case

continued on next page

Feature Area	Likelihood	Impact	Risk Level	Test Depth
New third-party integration	High	High	**Critical**	Exhaustive testing
Static FAQ page	Very low	Very low	**Minimal**	Smoke test only
Search	Medium	Medium	**Medium**	Core scenarios, performance spot check
Admin user management	Low	High	**High**	CRUD, permissions, audit logging

Factors Increasing Likelihood

- Recently modified code
- Complex business logic
- Third-party dependencies
- First-time implementation
- Technical debt
- Concurrency handling

Factors Increasing Impact

- Revenue-generating flows (checkout, payments)
- User data handling (registration, profiles)
- Security boundaries (auth, authorization)
- Compliance features (GDPR, audit logs)
- Public-facing features
- Downstream dependencies

7.2 Test Depth by Risk Level

Risk Level	Test Depth
Critical	Exhaustive: all positive, all negative, edge cases, security, performance, cross-browser
High	Thorough: all positive, key negative, boundary values, one cross-browser check
Medium	Standard: happy path, key negative cases, one boundary check
Low	Light: happy path only
Minimal	Smoke: verify it loads and basic function works

7.3 Estimation Techniques

Work Breakdown

Test Type	Estimated Time
Functional positive (12 cases)	3 hours
Functional negative (8 cases)	2 hours
Edge cases and BVA	2 hours
Cross-browser (3 browsers)	1.5 hours
Exploratory session	1 hour
Bug verification	1 hour
Total	**10.5 hours**

Three-Point Estimation

(Optimistic + 4 x Most Likely + Pessimistic) / 6
Example: (2 + 4×3 + 6) / 6 = **3.3 days**

7.4 Entry and Exit Criteria

Entry Criteria (before testing begins)

- Build deployed successfully

- Smoke tests passing
- Test data loaded
- Requirements available and reviewed
- Test environment stable

Exit Criteria (testing is "done enough")

- All critical/high bugs resolved and verified
- 95% of planned test cases executed
- No open blockers
- Regression suite green
- Exploratory sessions completed for high-risk areas

When Exit Criteria Are Not Met

Document: which criteria are not met, the associated risks, and your recommendation (release / don't release / release with mitigations). This is a risk acceptance decision for management, not QA. QA provides data; management accepts risk.

7.5 The Lightweight Test Plan

```
Feature:  Checkout Flow Redesign
Release:  v2.5.0
QA Lead:  Jane D.
Date:     2025-03-15

SCOPE:
- In scope: New checkout UI, payment integration, order confirmation
- Out of scope: User registration, product catalog, admin panel

RISK ASSESSMENT:
- Payment processing: CRITICAL — new Stripe v3 integration
- Address validation: HIGH — new autocomplete API
- Order confirmation: MEDIUM — mostly UI changes
- Email notifications: LOW — existing system, no changes

TEST APPROACH:
- Functional: 25 test cases covering all risk levels
- Exploratory: 2 sessions focused on payment edge cases
- Cross-browser: Chrome, Firefox, Safari (latest)
```

```
- Performance: Load test checkout with 100 concurrent users

ENTRY CRITERIA:
- Build deployed to staging
- Stripe sandbox configured
- Test accounts created (admin, customer, guest)

EXIT CRITERIA:
- All critical/high bugs resolved
- 90%+ test case pass rate
- Payment verified for all card types

SCHEDULE:
- Day 1-2: Functional testing
- Day 3: Exploratory sessions
- Day 4: Cross-browser and regression
- Day 5: Bug verification and sign-off
```

Key Takeaways — Chapter 7

- Risk = Probability x Impact — test high-risk areas first and deepest
- Use risk assessment to allocate limited time where it matters most
- Entry criteria prevent wasted testing on unstable builds
- Exit criteria define "done enough" and what to do when they're not met
- Document strategy in a lightweight test plan

Exercises — Chapter 7

Beginner: Create a risk assessment matrix for a simple blog application (posts, comments, user accounts, admin panel, RSS feed). Assign risk levels and test depth.

Intermediate: You are QA lead for a release with: (1) new Apple Pay integration, (2) redesigned settings page, (3) search indexing bug fix, (4) updated Terms of Service page. Create a complete risk assessment, estimate testing time using three-point estimation, and define entry/exit criteria.

Advanced: Create a full test plan for a mobile banking app release that includes: new biometric login, updated transaction history, and a bug fix

for notification delays. Include risk assessment, test approach, schedule, entry/exit criteria, and resource allocation.

Career Translation

Resume phrasing

- Created risk-based test strategies that allocated 70% of testing effort to Critical and High-risk features, reducing post-release defect escape rate by 40%
- Built risk assessment matrices for every release cycle, enabling data-driven go/no-go decisions and aligning QA priorities with business impact
- Defined entry and exit criteria for 20+ release cycles, preventing 8 premature deployments on unstable builds and ensuring measurable quality gates
- Authored lightweight test plans that reduced planning overhead from 3 days to 4 hours while maintaining full coverage of scope, risk, approach, and schedule

Cover letter framing

I plan testing the way an investor allocates capital — based on risk. High-risk areas like payment processing and authentication receive exhaustive coverage, while low-risk static pages get smoke tests. This approach ensures teams ship on schedule without gambling on untested critical paths. My lightweight test plans communicate strategy clearly to stakeholders without drowning in documentation that nobody reads.

Interview framing

"I approach test planning by asking 'Where would a failure hurt the most?' first. I build a risk matrix — probability times impact — and allocate test depth accordingly. Payment flows get exhaustive coverage with edge cases and cross-browser testing; FAQ pages get a smoke test. I optimize for risk reduction per hour of testing effort. Trade-offs include the possibility of under-testing a 'low-risk' area that turns out to have bugs, which is why I include at least one exploratory session for every feature area, even low-risk ones."

What not to say

- "I test everything equally" — this wastes time on low-risk areas while under-testing critical ones
- "I don't write test plans because they become outdated" — lightweight plans take hours, not days, and prevent chaos
- "Entry criteria slow us down" — entry criteria prevent wasting days of testing on a broken build
- "We release when we feel confident" — exit criteria replace feelings with measurable thresholds
- "Risk assessment is a project manager's job" — QA is uniquely qualified to assess defect likelihood based on code complexity and testing history

Interview Depth Check

Question 1

Prompt: You are the sole QA engineer on a team releasing a feature in 5 days. The feature includes a new Stripe payment integration, a redesigned navigation bar, and an updated help center. You estimate 40 hours of testing needed but only have 20 hours available. How do you plan?

What a strong answer should cover:

- Risk assessment: Stripe integration is Critical risk, navigation is Medium, help center is Low
- Allocate time proportionally: ~14 hours on payments, ~4 hours on navigation, ~2 hours on help center
- Define what "testing" means at each depth: exhaustive for payments, core scenarios for navigation, smoke for help center
- Communicate the trade-off and risk acceptance to stakeholders
- Define clear exit criteria tied to risk levels

Example answer:

- Risk assessment: Stripe integration is Critical (new third-party, revenue flow, possible data issues), navigation is Medium (UI change, could affect usability but no data risk), help center is Low (static content, minimal logic)
- Allocation: 14 hours on Stripe — covering all card types, error handling, refunds, webhooks, edge cases; 4 hours on navigation — core flows on 3 browsers, mobile responsiveness; 2 hours on help center — links work, content renders, search functions
- Exit criteria: Stripe must pass 100% of test cases with zero open Critical/High bugs; navigation must pass core flows; help center must load without errors
- I would document the plan and explicitly state: "Help center receives smoke testing only. Risk accepted: cosmetic or content issues may escape to production. Mitigation: content can be hotfixed without code deployment."

Question 2

Prompt: The development team pushes back on your entry criteria, saying "We can't wait for the staging environment to be stable — we need to start testing on the feature branch." How do you respond?

What a strong answer should cover:

- Acknowledge the pressure and the desire to move fast
- Explain the cost of testing on an unstable environment: wasted time, false failures, missed real bugs
- Propose a compromise: smoke test on the feature branch, full testing on staging
- Quantify the risk: "Last time we tested on an unstable build, we wasted 2 days re-executing failed tests"

Example answer:

- I would acknowledge the time pressure and share the goal of getting feedback early

- I would explain from experience: testing on unstable environments creates noise — tests fail due to environment issues, not product bugs, and the team wastes time investigating false failures
- My compromise: I will run a targeted smoke test on the feature branch to catch obvious blockers, then execute the full test plan on staging once the build is deployed and stable
- I would quantify: "In the last release, we skipped entry criteria and lost 12 hours re-running tests that failed due to a deployment issue, not a product bug. The 2 hours we would have spent waiting for stability would have saved 12."
- If the team still insists, I would test on the feature branch but document in the test plan: "Testing executed on unstable environment. Results may include false failures. Re-execution needed on staging before sign-off."

Question 3

Prompt: It is release day. Your exit criteria say "All Critical and High bugs resolved," but there is one High bug open: a performance issue where the dashboard takes 8 seconds to load instead of the target 3 seconds. The team wants to release. What do you do?

What a strong answer should cover:

- Present the data: the exit criterion is not met, and here is the specific gap
- Assess the real-world impact: is 8 seconds acceptable for now? How many users are affected?
- Propose options: delay release, release with a known issue and a mitigation plan, or reclassify the bug
- QA provides the data and recommendation; management decides on risk acceptance
- Document the decision regardless of outcome

Example answer:

- I would present the situation clearly: "Exit criteria state all High bugs must be resolved. BUG-4521 (dashboard load time 8s vs. target 3s) is still open."
- I would provide context: "This affects all users who visit the dashboard. While it is not broken — it still loads — the 8-second delay is 2.5x slower than our target and may impact user satisfaction."
- I would propose options: (A) Delay release by 1 day for the fix, (B) Release with a known issue and schedule a hotfix for this week, (C) Re-classify as Medium if the team agrees 8 seconds is acceptable temporarily
- I would give my recommendation: "Option B — release with a documented known issue and a committed hotfix date — balances shipping on time with acknowledging the quality gap."
- Regardless of the decision, I would document: who decided, what the accepted risk is, and the timeline for resolution

Question 4

Prompt: How do you use data from previous releases to improve your risk assessment for the next release?

What a strong answer should cover:

- Track defect density by feature area across releases
- Areas with recurring bugs get higher risk ratings in future assessments
- Analyze escaped defects (production bugs) to identify where test coverage was insufficient
- Use velocity of fixes as an indicator of code stability
- Feed exploratory testing findings into risk models

Example answer:

- I maintain a simple spreadsheet tracking defects per feature area per release. If the payment module had 12 bugs in the last 3 releases but user profiles had 2, payment gets a higher likelihood rating next time

- I specifically track escaped defects — bugs that reached production — because these reveal gaps in my risk model. If I rated search as Low risk but it had 3 production escapes, I need to re-evaluate
- I look at code churn: features with frequent code changes in the last sprint get a likelihood bump because new code introduces new bugs
- Developer feedback is another input: if a developer says "this refactor touched a lot of shared code," I increase risk even if the feature itself seems simple
- Over time, this data-driven approach makes each risk assessment more accurate than the previous one, moving from guesswork to evidence-based planning

Chapter 8: Verification vs Validation

Two terms frequently confused, even by experienced engineers. Understanding the difference shapes how you approach testing.

8.1 The Core Distinction

Aspect	Verification	Validation
Question	"Are we building the product **right**?"	"Are we building the **right** product?"
Focus	Conformance to specification	Conformance to user needs
Timing	During development	After build / with real users
Who	Developers, QA, reviewers	End users, product owners, beta testers

8.2 The Classic Example

A team builds a login system to spec: email field, 8-character password, redirect to /dashboard. **Verification** confirms the code works as specified. **Validation** reveals users expected phone number login — the spec never mentioned it. The product passes verification but fails validation.

8.3 Why Both Matter

A product can pass all verification and fail validation:

- **Feature nobody uses**: 50-chart dashboard renders perfectly, but users need only 3 charts and can't find them
- **Correct but confusing**: Error "INVALID_CREDENTIALS_401" is accurate but users don't know what to do
- **Spec was wrong**: Shows USD prices but launching in Europe where users expect EUR

8.4 QA's Role in Both

Verification: Write test cases against specs, report deviations, maintain regression suites, participate in reviews

Validation: Participate in UAT, provide usability feedback, question confusing requirements, report "works correctly but seems wrong" observations

> **Pro Tip (Shift-Left Validation)**: Don't wait until UAT. During requirements refinement, ask: "This spec says X, but would users expect Y?" During design review: "This flow has 7 steps — can we reduce to 3?"

Key Takeaways — Chapter 8

- Verification: "Building the product right" — does it match the spec?
- Validation: "Building the right product" — does it meet user needs?
- A product can pass verification and fail validation
- QA must do both: check specs AND advocate for users
- Shift validation left: question requirements early

Career Translation

Resume phrasing

- Integrated both verification (spec conformance) and validation (user need fulfillment) into test strategies, catching 10+ usability issues per quarter that would have shipped as technically correct but user-hostile features

- Championed shift-left validation by questioning requirements during sprint refinement, identifying 15 spec gaps before development began and saving an estimated 60 developer hours in rework
- Provided validation feedback during UAT cycles by correlating user behavior analytics with test results, surfacing features that passed all test cases but had 30% user abandonment rates
- Bridged QA and product by raising "works correctly but seems wrong" observations that led to 8 UX redesigns improving user task completion rates

Cover letter framing

I test beyond the specification. While verification — confirming the product matches the spec — is essential, I also actively validate whether the spec itself serves the user. This dual perspective means I catch not only bugs in implementation but also flaws in the requirements themselves. Teams I work with ship products that are both technically correct and genuinely useful.

Interview framing

"I approach testing as two distinct questions: 'Did we build it right?' and 'Did we build the right thing?' Verification is the first question — I check that the code matches the specification. Validation is the second — I check that the specification matches the user's actual need. I optimize for catching both types of failures because shipping a perfectly implemented feature that nobody uses is just as costly as shipping a broken one. Trade-offs include the political sensitivity of questioning requirements — telling a PM 'the spec is wrong' requires data and diplomacy — which is why I frame validation feedback as user advocacy, not criticism."

What not to say

- "My job is to test against the spec, not question it" — misses half of QA's value
- "Validation is the product manager's problem" — QA is closest to the actual user experience during testing

- "If it matches the spec, it passes" — a feature can pass every test case and still fail the user
- "I don't get involved in requirements discussions" — shift-left validation is one of the highest-leverage QA activities
- "Verification and validation are basically the same thing" — confusing these terms is a red flag in interviews

Interview Depth Check

Question 1

Prompt: A feature passes all 50 test cases, but during UAT, users say the flow is confusing and they cannot complete the task. What happened, and what should QA have done differently?

What a strong answer should cover:

- The feature passed verification (matches spec) but failed validation (does not meet user needs)
- The spec itself was flawed — it defined a technically correct but user-hostile flow
- QA should have participated in requirements review and questioned the flow early
- Usability feedback should be part of the QA process, not only UAT
- Exploratory testing with a usability focus could have caught this earlier

Example answer:

- This is a textbook case of verification passing while validation fails — the code does exactly what the spec says, but the spec did not account for how real users think
- What QA should have done differently: during requirements review, ask "Has this flow been user-tested?" or "Can we reduce these 7 steps to 3?" — questioning the design before code is written
- During testing, I would run at least one exploratory session with a usability charter: "Explore the checkout flow as a first-time user to discover confusion points and abandonment risks"

- Going forward, I would advocate for lightweight usability validation — even having 3 internal colleagues attempt the flow without instructions reveals confusion points
- The lesson: 100% test case pass rate does not mean the product is ready. Verification without validation is incomplete quality assurance

Question 2

Prompt: Your team is building a notification system. The spec says "Send email notification when order ships." You notice the spec does not mention push notifications, SMS, or in-app notifications. How do you approach this?

What a strong answer should cover:

- This is a validation opportunity — questioning whether the spec fully addresses user needs
- Raise it as a requirements gap, not a bug
- Research user expectations: do competitors send push notifications? Do users have notification preferences?
- Propose the question to the product owner before development begins (shift-left validation)
- If the spec stays as-is, verify email works correctly while documenting the limitation

Example answer:

- I would raise this during refinement: "The spec specifies email only. Have we considered that users may expect push notifications or SMS, especially since our competitors offer multi-channel notifications?"
- I would frame it as a user expectation question, not a demand: "Users who have push notifications enabled on our mobile app may expect a shipping notification there, not just in their email inbox"
- If the product owner decides email is sufficient for v1, I accept that decision but document it: "Scope: email notification only. Future consideration: push/SMS/in-app notifications"

- I would then verify email works correctly against the spec (verification) while noting in my test report: "Validation note: user expectations for multi-channel notification not addressed in v1"

Question 3

Prompt: A developer says "I don't see the point of validation — if the product owner wrote the spec, they already validated the requirements with users." How do you respond?

What a strong answer should cover:

- Specs often reflect business goals, not verified user behavior
- Requirements are assumptions until validated with real users
- Even well-researched specs miss edge cases and interaction patterns that QA discovers during testing
- QA sees the product from a unique angle — they use it more intensively than anyone
- Validation is continuous, not a one-time event at spec writing

Example answer:

- I would acknowledge that product owners do significant research, then explain: "Specs are hypotheses about what users need. Even the best-researched spec makes assumptions that can only be validated by observing actual usage."
- I would give a concrete example: "A spec might say 'display 50 charts on the analytics dashboard' because a customer requested it. But during testing, I notice it takes 12 seconds to load and most charts require scrolling past the fold. The spec is technically correct, but the user experience is poor."
- I would explain QA's unique position: "We interact with the product more intensively than any other role. We fill out every form, click every button, and try every path. That intensity reveals usability issues that product owners — who think in features — may not see."
- Validation is not questioning the PM's competence; it is adding a second perspective that strengthens the product

Question 4

Prompt: How would you incorporate validation into a sprint workflow without adding significant overhead?

What a strong answer should cover:

- Ask validation questions during refinement (no extra meeting needed)
- Include a usability-focused exploratory charter in every testing cycle
- Report "works but seems wrong" observations alongside bug reports
- Invite one non-team-member to attempt key flows without instructions (lightweight usability testing)
- Review user analytics after release to check actual vs. expected usage

Example answer:

- During refinement (already happening): I add 2-3 validation questions per story: "Who is the user? What are they trying to accomplish? Have we confirmed they need this flow?"
- During testing (minimal overhead): I add one exploratory charter per sprint with a usability focus — "Explore [new feature] as a first-time user to discover confusion points." This takes 30-60 minutes
- During bug reporting (no overhead): When I observe something technically correct but confusing, I file it as an "Observation" or "UX Feedback" alongside my bug reports. The product owner can triage these separately
- Post-release (5 minutes per feature): I check analytics — did users complete the flow? Where did they drop off? This data validates whether the feature meets its purpose
- Total overhead: approximately 1-2 hours per sprint, which consistently prevents multi-sprint rework on features that fail validation at UAT

PART V

PRACTICE TEST CASES

This section contains 200 practice test cases across 10 application types. Study them. Adapt them. Use them as templates for your own work.

Application 1: E-Commerce Store (20 test cases)

Registration & Login

```
TC-ECOM-001: Verify successful registration with valid email and strong password
GIVEN the registration page is loaded
WHEN  user enters email "new@example.com", password "Str0ng!Pass", confirms password
THEN  account is created, confirmation email sent, user redirected to welcome page

TC-ECOM-002: Verify registration fails with already-registered email
GIVEN user "existing@example.com" already has an account
WHEN  new registration attempted with "existing@example.com"
THEN  error "An account with this email already exists" displayed

TC-ECOM-003: Verify login fails after 5 incorrect password attempts (account lockout)
GIVEN user "locked@example.com" has a valid account
WHEN  5 consecutive login attempts with wrong password
THEN  account is locked, message "Account locked. Try again in 30 minutes." displayed

TC-ECOM-004: Verify password reset email contains a valid, time-limited link
GIVEN user requests password reset for "user@example.com"
WHEN  user clicks the reset link within 24 hours
THEN  user is directed to set a new password; link expires after use
```

Product Catalog

```
TC-ECOM-005: Verify product search returns relevant results
GIVEN products "Blue Widget" and "Red Widget" exist in catalog
WHEN  user searches for "widget"
THEN  both products appear in search results, ordered by relevance

TC-ECOM-006: Verify search returns empty state for non-existent product
GIVEN no product named "xyznonexistent" exists
WHEN  user searches for "xyznonexistent"
THEN  message "No products found. Try a different search." displayed

TC-ECOM-007: Verify product filtering by price range
GIVEN products exist at $10, $25, $50, $100, $200
WHEN  user filters by price range $20-$100
THEN  only products at $25, $50, and $100 are displayed

TC-ECOM-008: Verify product detail page displays all required information
GIVEN product "Blue Widget" exists with images, description, price, reviews
WHEN  user navigates to the product detail page
THEN  all product images load, description is complete, price shows $29.99, reviews section visible
```

Shopping Cart

```
TC-ECOM-009: Verify adding a product to cart updates cart count
GIVEN cart is empty
WHEN  user adds "Blue Widget" to cart
THEN  cart icon shows count "1", cart page shows Blue Widget at $29.99

TC-ECOM-010: Verify updating item quantity recalculates total
GIVEN cart contains 1x Blue Widget at $29.99
WHEN  user changes quantity to 3
THEN  line item shows 3x $29.99 = $89.97, cart total updates to $89.97

TC-ECOM-011: Verify removing the last item from cart shows empty cart state
GIVEN cart contains 1 item
WHEN  user clicks "Remove"
THEN  cart shows "Your cart is empty" with a "Continue Shopping" link

TC-ECOM-012: Verify cart persists after logout and login
GIVEN user has 2 items in cart and logs out
WHEN  user logs back in with same account
THEN  cart still contains the same 2 items with correct quantities
```

Checkout

```
TC-ECOM-013: Verify successful checkout with credit card
GIVEN cart has items totaling $89.97 and user is logged in
WHEN  user enters valid shipping address, selects standard shipping, enters valid Visa card
THEN  order is placed, confirmation page shows order number, confirmation email sent

TC-ECOM-014: Verify checkout fails with expired credit card
GIVEN user enters card with expiration date in the past
WHEN  user clicks "Place Order"
THEN  error "Your card has expired. Please use a different payment method."

TC-ECOM-015: Verify discount code application
GIVEN cart total is $100.00 and discount code "SAVE20" gives 20% off
WHEN  user enters "SAVE20" and clicks "Apply"
THEN  discount line shows -$20.00, new total is $80.00

TC-ECOM-016: Verify shipping cost calculation for different methods
GIVEN cart total is $50.00
WHEN  user selects "Express Shipping" ($12.99)
THEN  order total shows $50.00 + $12.99 = $62.99

TC-ECOM-017: Verify order cannot be placed with empty required fields
GIVEN checkout form is loaded
WHEN  user clicks "Place Order" with shipping address fields empty
THEN  validation errors appear on all required fields, order is not placed
```

Order Management

```
TC-ECOM-018: Verify order appears in order history after purchase
GIVEN user just completed a purchase (order #12345)
WHEN  user navigates to "My Orders"
THEN  order #12345 appears with correct date, items, total, and status "Processing"

TC-ECOM-019: Verify order cancellation within allowed window
GIVEN order #12345 was placed 1 hour ago (cancellation window is 24 hours)
WHEN  user clicks "Cancel Order"
THEN  order status changes to "Cancelled", refund initiated, confirmation email sent

TC-ECOM-020: Verify order cancellation blocked after shipping
GIVEN order #12345 has status "Shipped"
WHEN  user attempts to cancel
THEN  message "This order has already shipped and cannot be cancelled. Please initiate a return."
```

Application 2: Banking App (20 test cases)

```
TC-BANK-001: Verify login with valid credentials shows account dashboard
GIVEN user has active banking account
WHEN  user enters correct username and password
THEN  dashboard shows account summary with balance, recent transactions

TC-BANK-002: Verify two-factor authentication is required on new device
GIVEN user logs in from unrecognized device
WHEN  correct credentials entered
THEN  2FA code requested via SMS/email before granting access

TC-BANK-003: Verify account balance matches sum of transactions
GIVEN account has opening balance of $5,000 and 3 transactions (-$100, +$250, -$75)
```

```
WHEN  user views account balance
THEN  balance shows $5,075.00

TC-BANK-004: Verify internal transfer between own accounts
GIVEN user has checking ($2,000) and savings ($3,000) accounts
WHEN  user transfers $500 from checking to savings
THEN  checking shows $1,500, savings shows $3,500, transfer appears in both histories

TC-BANK-005: Verify transfer fails when amount exceeds balance
GIVEN checking account has $2,000
WHEN  user attempts to transfer $2,500
THEN  error "Insufficient funds" displayed, no transfer executed

TC-BANK-006: Verify external transfer requires additional verification
GIVEN user initiates transfer to external bank account
WHEN  transfer amount is $1,000
THEN  additional security verification required (2FA or security questions)

TC-BANK-007: Verify daily transfer limit enforcement
GIVEN daily transfer limit is $5,000 and user has transferred $4,500 today
WHEN  user attempts additional $1,000 transfer
THEN  error "Daily transfer limit exceeded. Remaining today: $500."

TC-BANK-008: Verify transaction history shows correct date, amount, description
GIVEN user made purchase of $45.99 at "Coffee Shop" on March 15
WHEN  user views transaction history
THEN  entry shows: Mar 15, "Coffee Shop", -$45.99, Running balance correct

TC-BANK-009: Verify transaction search by date range
GIVEN user has transactions from Jan-March
WHEN  user filters by date range Feb 1 - Feb 28
THEN  only February transactions displayed

TC-BANK-010: Verify bill payment scheduling
GIVEN user schedules $150 electric bill payment for March 20
WHEN  March 20 arrives
THEN  payment processed, balance debited, status shows "Paid"

TC-BANK-011: Verify overdraft protection notification
GIVEN checking balance is $50 and overdraft protection is enabled
WHEN  user makes $75 purchase
THEN  purchase approved, overdraft notification sent, fees displayed

TC-BANK-012: Verify session timeout after inactivity
GIVEN user is logged into banking app
WHEN  15 minutes pass with no activity
THEN  session expires, user redirected to login with "Session expired" message

TC-BANK-013: Verify statement download in PDF format
GIVEN user has February 2025 statement available
WHEN  user clicks "Download PDF"
THEN  PDF downloads with correct transactions, totals, and bank branding

TC-BANK-014: Verify card freeze/unfreeze functionality
GIVEN user has active debit card
WHEN  user clicks "Freeze Card"
THEN  card status changes to "Frozen", transactions declined, user can unfreeze

TC-BANK-015: Verify mobile check deposit
GIVEN user takes photos of front and back of check for $500
WHEN  user submits deposit
THEN  deposit appears as "Pending" with expected clearance date

TC-BANK-016: Verify login attempt from foreign IP triggers security alert
GIVEN user normally logs in from US
WHEN  login attempted from IP geolocated to another country
THEN  security alert email sent, additional verification required
```

```
TC-BANK-017: Verify account nickname can be customized
GIVEN user has checking account "Account ending 4521"
WHEN  user renames to "Main Checking"
THEN  name updates across all views (dashboard, transfers, statements)

TC-BANK-018: Verify joint account shows both account holders
GIVEN account is joint between User A and User B
WHEN  either user logs in
THEN  account shows both names, both can view and transact

TC-BANK-019: Verify interest calculation on savings account
GIVEN savings account has $10,000 at 4.5% APY
WHEN  monthly interest is applied
THEN  approximately $36.99 interest credited (10000 * 0.045 / 12)

TC-BANK-020: Verify accessibility — screen reader reads account balance
GIVEN user is using VoiceOver (iOS) or TalkBack (Android)
WHEN  user navigates to account balance
THEN  screen reader announces "Account balance: five thousand and seventy-five dollars"
```

Application 3: Social Media Platform (20 test cases)

```
TC-SOCIAL-001: Verify posting a text-only status update
GIVEN user is logged in on the home feed
WHEN  user types "Hello world!" and clicks "Post"
THEN  post appears at top of feed with correct timestamp and user avatar

TC-SOCIAL-002: Verify posting with an image attachment
GIVEN user creates a new post
WHEN  user attaches a 3MB .jpg image and clicks "Post"
THEN  post appears with image thumbnail; clicking opens full-size image

TC-SOCIAL-003: Verify character limit enforcement on posts
GIVEN post character limit is 500
WHEN  user types 501 characters
THEN  counter shows "-1", post button is disabled

TC-SOCIAL-004: Verify liking a post increments the like count
GIVEN a post has 42 likes
WHEN  user clicks "Like"
THEN  count shows 43, heart icon fills/changes color, user's name appears in like list

TC-SOCIAL-005: Verify unliking a post decrements the count
GIVEN user has liked a post (count: 43)
WHEN  user clicks "Unlike"
THEN  count returns to 42, heart icon returns to default

TC-SOCIAL-006: Verify commenting on a post
GIVEN user views a post
WHEN  user types "Great post!" in comment box and clicks "Comment"
THEN  comment appears under the post with user's name and timestamp

TC-SOCIAL-007: Verify deleting own post
GIVEN user has a published post
WHEN  user clicks "Delete" and confirms
THEN  post is removed from feed, likes and comments are also removed

TC-SOCIAL-008: Verify user cannot edit another user's post
GIVEN User A views User B's post
```

```
WHEN  User A looks for edit option
THEN  no edit option is available for other users' posts

TC-SOCIAL-009: Verify following another user
GIVEN User A is not following User B
WHEN  User A clicks "Follow" on User B's profile
THEN  User B appears in User A's following list; User A appears in User B's followers

TC-SOCIAL-010: Verify follower count updates in real-time
GIVEN User B has 100 followers
WHEN  User A follows User B
THEN  User B's follower count shows 101 without page refresh

TC-SOCIAL-011: Verify blocking a user hides their content
GIVEN User A blocks User B
WHEN  User A views their feed
THEN  all of User B's posts, comments, and interactions are hidden

TC-SOCIAL-012: Verify direct message delivery
GIVEN User A sends "Hey!" to User B via DM
WHEN  User B opens their message inbox
THEN  message "Hey!" appears from User A with correct timestamp

TC-SOCIAL-013: Verify notification for new follower
GIVEN User A follows User B
WHEN  User B checks notifications
THEN  notification "User A started following you" appears

TC-SOCIAL-014: Verify profile picture upload and display
GIVEN user has default avatar
WHEN  user uploads a 2MB .png profile picture
THEN  new picture appears on profile, in posts, in comments, and in search results

TC-SOCIAL-015: Verify hashtag creates a clickable link
GIVEN user posts "Love this #sunset"
WHEN  post is published
THEN  "#sunset" is a clickable link that shows all posts with that hashtag

TC-SOCIAL-016: Verify @mention sends notification to mentioned user
GIVEN User A posts "Thanks @UserB for the help!"
WHEN  User B checks notifications
THEN  notification "@UserA mentioned you in a post" appears

TC-SOCIAL-017: Verify feed loads additional posts on scroll (infinite scroll)
GIVEN user's feed has 100+ posts
WHEN  user scrolls to bottom of initially loaded posts
THEN  additional posts load automatically without page refresh

TC-SOCIAL-018: Verify post with link shows URL preview card
GIVEN user includes "https://example.com/article" in their post
WHEN  post is published
THEN  a preview card shows the article's title, image, and description

TC-SOCIAL-019: Verify reporting a post for inappropriate content
GIVEN user sees an offensive post
WHEN  user clicks "Report" and selects "Harassment"
THEN  confirmation "Report submitted" shown; post is flagged for moderation

TC-SOCIAL-020: Verify privacy setting — "Only followers can comment"
GIVEN user sets comments to "Followers only"
WHEN  a non-follower tries to comment on user's post
THEN  comment box is disabled with message "Only followers can comment"
```

Application 4: Healthcare Patient Portal (20 test cases)

```
TC-HEALTH-001: Verify patient can view upcoming appointments
GIVEN patient has appointment on March 20 at 2:00 PM with Dr. Smith
WHEN  patient logs in and navigates to "Appointments"
THEN  appointment shows: date, time, provider name, location, appointment type

TC-HEALTH-002: Verify appointment booking with available slot
GIVEN Dr. Smith has availability on March 25 at 10:00 AM
WHEN  patient selects the slot and confirms
THEN  appointment booked, confirmation shown, reminder email/SMS sent

TC-HEALTH-003: Verify appointment cancellation with 24-hour notice
GIVEN appointment is in 48 hours
WHEN  patient cancels
THEN  appointment removed, slot freed, cancellation confirmation sent

TC-HEALTH-004: Verify late cancellation shows warning
GIVEN appointment is in 12 hours
WHEN  patient attempts to cancel
THEN  warning "Late cancellation fee of $25 may apply. Continue?"

TC-HEALTH-005: Verify prescription list shows current medications
GIVEN patient has active prescriptions for Medication A and Medication B
WHEN  patient navigates to "Prescriptions"
THEN  both medications shown with dosage, frequency, prescribing doctor, refill date

TC-HEALTH-006: Verify prescription refill request
GIVEN Medication A is eligible for refill (refill date passed)
WHEN  patient clicks "Request Refill"
THEN  request submitted, pharmacy notified, status shows "Refill Requested"

TC-HEALTH-007: Verify lab results display
GIVEN lab results for blood panel are available
WHEN  patient views lab results
THEN  results show test name, value, reference range, and flagged abnormal values

TC-HEALTH-008: Verify secure messaging to provider
GIVEN patient needs to ask Dr. Smith a question
WHEN  patient sends message "Question about my medication dosage"
THEN  message appears in sent items, provider receives notification

TC-HEALTH-009: Verify access to visit summaries
GIVEN patient had a visit on March 10
WHEN  patient navigates to "Visit History"
THEN  summary shows date, provider, diagnosis codes, treatment notes, follow-up instructions

TC-HEALTH-010: Verify insurance information display
GIVEN patient has BlueCross insurance on file
WHEN  patient views "Insurance"
THEN  plan name, member ID, group number, copay information displayed

TC-HEALTH-011: Verify billing statement access
GIVEN patient has outstanding balance of $150
WHEN  patient navigates to "Billing"
THEN  itemized charges, insurance adjustments, patient responsibility, and payment options shown

TC-HEALTH-012: Verify online bill payment
GIVEN patient owes $150
WHEN  patient enters credit card and clicks "Pay $150"
THEN  payment processed, receipt generated, balance updated to $0

TC-HEALTH-013: Verify HIPAA-compliant session timeout
GIVEN patient portal session timeout is 10 minutes
WHEN  10 minutes of inactivity
THEN  session expires, user must re-authenticate, no PHI visible on screen

TC-HEALTH-014: Verify proxy access (parent viewing child's records)
GIVEN parent has proxy access to child's account (minor)
WHEN  parent logs in and switches to child's profile
THEN  child's appointments, prescriptions, and records are visible

TC-HEALTH-015: Verify immunization record display
GIVEN patient has COVID-19 and flu vaccination records
WHEN  patient navigates to "Immunizations"
THEN  each vaccination shows date, type, lot number, administering provider

TC-HEALTH-016: Verify allergies are prominently displayed
GIVEN patient has documented allergy to Penicillin
WHEN  patient views their health summary
THEN  "Allergy: Penicillin — Reaction: Anaphylaxis" displayed prominently in red/warning style

TC-HEALTH-017: Verify appointment reminder notification
GIVEN patient has appointment in 24 hours
WHEN  24-hour mark is reached
THEN  SMS and/or email reminder sent with appointment details
```

```
TC-HEALTH-018: Verify patient cannot access another patient's records
GIVEN Patient A is logged in
WHEN  Patient A attempts to access Patient B's record URL
THEN  access denied, "Unauthorized" error, security event logged

TC-HEALTH-019: Verify telehealth video visit launch
GIVEN patient has telehealth appointment in 5 minutes
WHEN  patient clicks "Join Visit"
THEN  video call interface opens with camera/microphone test, waiting room for provider

TC-HEALTH-020: Verify data export (patient's right to their data)
GIVEN patient requests data export
WHEN  export is processed
THEN  downloadable file includes demographics, visits, labs, medications in standard format
```

Application 5: SaaS Dashboard (20 test cases)

```
TC-SAAS-001: Verify dashboard loads with correct widgets for user role
GIVEN admin user logs in
WHEN  dashboard page loads
THEN  shows: revenue chart, active users, support tickets, system status widgets

TC-SAAS-002: Verify date range filter updates all widgets
GIVEN dashboard shows "Last 30 days" by default
WHEN  user changes to "Last 7 days"
THEN  all charts and metrics update to reflect the 7-day window

TC-SAAS-003: Verify data export to CSV
GIVEN revenue chart shows monthly data
WHEN  user clicks "Export CSV"
THEN  CSV file downloads with headers and data matching the chart

TC-SAAS-004: Verify team member invitation
GIVEN admin navigates to "Team" settings
WHEN  admin enters "newuser@company.com" and selects "Editor" role
THEN  invitation email sent, pending invite shown in team list

TC-SAAS-005: Verify role-based access control
GIVEN "Viewer" role user logs in
WHEN  user tries to access "Settings" page
THEN  "Settings" link not visible in navigation; direct URL shows "Access Denied"

TC-SAAS-006: Verify API key generation
GIVEN user navigates to "API Keys"
WHEN  user clicks "Generate New Key"
THEN  key displayed once with copy button, masked after page reload, creation date logged

TC-SAAS-007: Verify webhook configuration
GIVEN user adds webhook URL "https://example.com/webhook"
WHEN  user selects events "user.created" and "payment.completed" and saves
THEN  webhook appears in list with selected events and "Active" status

TC-SAAS-008: Verify usage meter shows current billing period consumption
GIVEN plan allows 10,000 API calls/month, 7,500 used
WHEN  user views usage dashboard
THEN  meter shows 75% used, "2,500 remaining", projected overage warning if on pace

TC-SAAS-009: Verify plan upgrade flow
GIVEN user is on "Starter" plan
WHEN  user selects "Pro" plan and enters payment
THEN  plan upgrades immediately, prorated charge applied, new features accessible

TC-SAAS-010: Verify plan downgrade at end of billing period
```

```
GIVEN user is on "Pro" plan (paid through March 31)
WHEN  user downgrades to "Starter"
THEN  Pro features available until March 31, then Starter features only

TC-SAAS-011: Verify notification preferences
GIVEN user navigates to notification settings
WHEN  user disables email notifications for "weekly summary"
THEN  weekly summary emails stop; other notification types unaffected

TC-SAAS-012: Verify audit log records admin actions
GIVEN admin changes a user's role from Editor to Viewer
WHEN  admin views audit log
THEN  entry shows: timestamp, admin name, action "Role changed", target user, old/new values

TC-SAAS-013: Verify search across all data entities
GIVEN user searches for "John"
WHEN  results load
THEN  shows matching customers, tickets, and invoices with entity type labels

TC-SAAS-014: Verify chart renders correctly with zero data
GIVEN new account with no data
WHEN  dashboard loads
THEN  charts show "No data yet" placeholder, no broken visualizations or errors

TC-SAAS-015: Verify SSO login via SAML
GIVEN organization has configured SAML SSO
WHEN  user enters company email on login page
THEN  redirected to company's identity provider, authenticated, returned to dashboard

TC-SAAS-016: Verify two-factor authentication setup
GIVEN user enables 2FA
WHEN  user scans QR code with authenticator app and enters code
THEN  2FA enabled, recovery codes displayed, subsequent logins require 2FA

TC-SAAS-017: Verify data retention policy enforcement
GIVEN organization sets 90-day data retention
WHEN  data older than 90 days exists
THEN  data is archived/deleted per policy, audit log records the purge

TC-SAAS-018: Verify custom report builder
GIVEN user creates report with: filter by date range, group by product, metric: revenue
WHEN  user runs the report
THEN  report shows revenue grouped by product for the selected date range

TC-SAAS-019: Verify white-labeling (custom branding)
GIVEN admin uploads company logo and sets brand color to #FF5722
WHEN  any team member logs in
THEN  custom logo and brand color appear throughout the interface

TC-SAAS-020: Verify account deletion and data removal
GIVEN admin requests account deletion
WHEN  confirmation period (30 days) passes
THEN  all data permanently deleted, login disabled, confirmation email sent
```

Application 6: Mobile App (20 test cases)

```
TC-MOBILE-001: Verify app launches within 3 seconds on supported devices
GIVEN app is installed on iPhone 14 / Pixel 7
WHEN  user taps app icon
THEN  splash screen appears immediately, main screen loads within 3 seconds
```

```
TC-MOBILE-002: Verify offline mode shows cached content
GIVEN user loaded the home feed while online
WHEN  device loses network connection
THEN  cached feed is displayed, "You're offline" banner appears

TC-MOBILE-003: Verify push notification delivery and tap behavior
GIVEN user receives push notification "New message from John"
WHEN  user taps the notification
THEN  app opens directly to the message conversation with John

TC-MOBILE-004: Verify app behavior on incoming phone call
GIVEN user is in the middle of composing a post
WHEN  incoming phone call interrupts
THEN  app state is preserved; after call ends, draft is still intact

TC-MOBILE-005: Verify portrait and landscape orientation
GIVEN user is viewing a photo gallery
WHEN  device is rotated from portrait to landscape
THEN  UI adapts correctly, no content is cut off, no crash occurs

TC-MOBILE-006: Verify pull-to-refresh updates content
GIVEN feed was loaded 5 minutes ago
WHEN  user pulls down on the feed
THEN  loading indicator appears, new content loads, feed updates

TC-MOBILE-007: Verify biometric authentication (Face ID / fingerprint)
GIVEN user has enabled Face ID for login
WHEN  user opens the app
THEN  Face ID prompt appears; successful scan logs user in

TC-MOBILE-008: Verify deep link opens correct screen
GIVEN user taps link "myapp://product/12345" in a text message
WHEN  app opens
THEN  product detail page for item #12345 is displayed

TC-MOBILE-009: Verify image upload from camera and gallery
GIVEN user wants to post a photo
WHEN  user selects "Camera" option
THEN  camera opens, user takes photo, photo appears in the post composer

TC-MOBILE-010: Verify app handles low storage gracefully
GIVEN device has < 50MB free storage
WHEN  user tries to download a large file
THEN  message "Not enough storage. Free up space and try again."

TC-MOBILE-011: Verify background location updates (if applicable)
GIVEN user has granted location permission
WHEN  app is in background and user moves to a new area
THEN  location-based content updates when app is foregrounded

TC-MOBILE-012: Verify dark mode displays correctly
GIVEN device is set to dark mode
WHEN  app is opened
THEN  all screens use dark color scheme, text is readable, no white flash on screen transitions

TC-MOBILE-013: Verify app update prompt
GIVEN a new mandatory app version is available
WHEN  user opens the outdated app
THEN  modal prompt "Please update to continue" with App Store / Play Store link

TC-MOBILE-014: Verify gesture navigation (swipe back)
GIVEN user is on a detail screen
WHEN  user swipes from left edge
THEN  navigates back to previous screen with smooth animation

TC-MOBILE-015: Verify accessibility — dynamic type support
GIVEN user has set system font size to "Extra Large"
WHEN  app is opened
THEN  all text in the app respects the system font size setting

TC-MOBILE-016: Verify VoiceOver / TalkBack navigation
GIVEN screen reader is enabled
WHEN  user navigates through the app
THEN  all interactive elements have accessible labels, correct reading order

TC-MOBILE-017: Verify app behavior on slow network (3G equivalent)
GIVEN network is throttled to 3G speed
WHEN  user loads the feed
THEN  loading indicator shown, content loads (possibly images deferred), no timeout error

TC-MOBILE-018: Verify multi-window / split-screen on tablets
GIVEN app is running on iPad in split-screen mode
WHEN  user resizes the app window
THEN  UI adapts responsively, no content overlap or missing elements

TC-MOBILE-019: Verify form data persistence on app backgrounding
GIVEN user is filling out a long form
WHEN  user switches to another app and returns
THEN  all entered data is preserved in the form fields

TC-MOBILE-020: Verify logout clears all local data
```

```
GIVEN user logs out
WHEN  another user logs into the same device
THEN  no data from previous user is visible (cache, images, messages cleared)
```

Application 7: Content Management System (20 test cases)

```
TC-CMS-001: Verify creating a new blog post with title and body
GIVEN admin is logged into CMS
WHEN  admin enters title "My First Post", writes body content, clicks "Save Draft"
THEN  post saved as Draft with correct title, body, creation timestamp

TC-CMS-002: Verify rich text editor formatting
GIVEN admin is editing a post
WHEN  admin applies bold, italic, heading, bullet list, and hyperlink
THEN  all formatting renders correctly in editor and in published view

TC-CMS-003: Verify image embedding in post body
GIVEN admin is editing a post
WHEN  admin inserts an image via upload (2MB .jpg)
THEN  image appears inline in editor, renders on published page, alt text is editable

TC-CMS-004: Verify post scheduling for future publication
GIVEN admin writes a post
WHEN  admin sets publish date to March 25 at 9:00 AM and clicks "Schedule"
THEN  post status is "Scheduled", not visible to public until March 25 9:00 AM

TC-CMS-005: Verify post revision history
GIVEN admin has edited a post 3 times
WHEN  admin views "Revisions"
THEN  all 3 versions visible with timestamps, author, and ability to restore any version

TC-CMS-006: Verify SEO metadata editing
GIVEN admin is editing a post
WHEN  admin sets meta title, meta description, and URL slug
THEN  page source shows correct meta tags; URL matches the custom slug

TC-CMS-007: Verify category and tag assignment
GIVEN categories "Tech" and "Business" exist
WHEN  admin assigns both categories and tags "startup, saas"
THEN  post appears on both category pages and both tag pages

TC-CMS-008: Verify media library upload and management
GIVEN admin navigates to Media Library
WHEN  admin uploads 5 images
THEN  all images appear with thumbnails, file size, dimensions, upload date

TC-CMS-009: Verify user role permissions (Editor vs Author)
GIVEN Author role can create and edit own posts only
WHEN  Author tries to edit another author's post
THEN  access denied, "You don't have permission to edit this post"

TC-CMS-010: Verify post URL generates correct SEO-friendly slug
GIVEN post title is "10 Tips for Better Testing in 2025!"
WHEN  post is saved
THEN  URL slug auto-generates as "10-tips-for-better-testing-in-2025"

TC-CMS-011: Verify commenting system on published posts
GIVEN a published post with comments enabled
WHEN  visitor submits comment "Great article!"
THEN  comment appears (or enters moderation queue) with visitor's name and timestamp
```

```
TC-CMS-012: Verify comment moderation workflow
GIVEN moderation is enabled and a new comment arrives
WHEN  admin views "Pending Comments"
THEN  comment shown with Approve / Reject / Spam options

TC-CMS-013: Verify 404 page for deleted or non-existent posts
GIVEN post at /blog/old-post was deleted
WHEN  visitor navigates to that URL
THEN  custom 404 page displayed with search bar and popular posts

TC-CMS-014: Verify RSS feed generation
GIVEN blog has 10 published posts
WHEN  visitor accesses /feed or /rss
THEN  valid RSS XML with latest posts, titles, descriptions, and links

TC-CMS-015: Verify bulk post operations (delete, change status)
GIVEN admin selects 5 posts via checkboxes
WHEN  admin chooses "Move to Trash" from bulk actions
THEN  all 5 posts moved to trash, confirmation shown

TC-CMS-016: Verify post preview before publishing
GIVEN admin has a draft post
WHEN  admin clicks "Preview"
THEN  post renders exactly as it will appear to visitors, including images and formatting

TC-CMS-017: Verify multi-language content (if applicable)
GIVEN post exists in English
WHEN  admin creates a French translation
THEN  French version accessible at /fr/post-slug, language switcher links both versions

TC-CMS-018: Verify site search indexes new content
GIVEN admin publishes a new post with title "Selenium Grid Setup"
WHEN  visitor searches for "Selenium Grid"
THEN  new post appears in search results

TC-CMS-019: Verify page builder / template selection
GIVEN templates "Default" and "Landing Page" exist
WHEN  admin selects "Landing Page" template for a new page
THEN  page uses landing page layout (full-width, no sidebar)

TC-CMSS-020: Verify backup and restore functionality
GIVEN admin initiates a full site backup
WHEN  backup completes
THEN  downloadable backup file created; restoring it recreates all content and settings
```

Application 8: Booking System (20 test cases)

```
TC-BOOK-001: Verify available time slots display correctly
GIVEN service "Haircut" has slots at 9:00, 10:00, 11:00 on March 20
WHEN  customer views March 20 availability
THEN  three available slots shown; booked slots are not shown

TC-BOOK-002: Verify successful booking creates confirmation
GIVEN customer selects March 20 at 10:00 AM for "Haircut"
WHEN  customer confirms booking
THEN  confirmation page shows details, confirmation email sent, slot no longer available

TC-BOOK-003: Verify double-booking prevention
GIVEN March 20 at 10:00 AM is already booked
WHEN  another customer tries to book the same slot
```

```
THEN  slot shows as unavailable, error "This time slot is no longer available"

TC-BOOK-004: Verify booking cancellation
GIVEN customer has booking on March 20 at 10:00 AM
WHEN  customer cancels at least 24 hours before
THEN  booking removed, slot freed, cancellation email sent, no fee charged

TC-BOOK-005: Verify late cancellation fee
GIVEN booking is in 6 hours, cancellation policy requires 24-hour notice
WHEN  customer cancels
THEN  warning "Late cancellation fee of $25 applies" shown, fee charged on confirmation

TC-BOOK-006: Verify booking reschedule
GIVEN customer has booking on March 20 at 10:00 AM
WHEN  customer reschedules to March 21 at 2:00 PM
THEN  old slot freed, new slot booked, updated confirmation sent

TC-BOOK-007: Verify multi-service booking
GIVEN customer selects "Haircut" (30 min) and "Color" (60 min)
WHEN  customer books
THEN  90-minute block reserved, total price shows combined amount

TC-BOOK-008: Verify provider calendar view
GIVEN provider has 5 bookings on March 20
WHEN  provider views their calendar
THEN  all 5 bookings shown with customer name, service, time, and status

TC-BOOK-009: Verify email reminder before appointment
GIVEN booking is tomorrow at 10:00 AM
WHEN  24 hours before the appointment
THEN  reminder email/SMS sent to customer with booking details

TC-BOOK-010: Verify timezone handling for remote bookings
GIVEN provider is in EST, customer is in PST
WHEN  customer books a 2:00 PM slot
THEN  customer sees 2:00 PM PST, provider sees 5:00 PM EST

TC-BOOK-011: Verify payment collection at time of booking
GIVEN service "Consultation" costs $75
WHEN  customer completes booking with credit card
THEN  $75 charged, receipt emailed, booking confirmed

TC-BOOK-012: Verify deposit vs full payment option
GIVEN service costs $200, deposit required is $50
WHEN  customer books
THEN  $50 charged now, remaining $150 due at appointment

TC-BOOK-013: Verify recurring booking (weekly appointment)
GIVEN customer sets up weekly "Therapy" appointment every Tuesday at 3 PM
WHEN  recurring booking is created
THEN  appointments created for next 4 weeks, each visible in customer and provider calendars

TC-BOOK-014: Verify waitlist functionality
GIVEN all slots on March 20 are booked
WHEN  customer joins waitlist
THEN  customer notified if a slot opens, can book within a time window

TC-BOOK-015: Verify service provider unavailability (vacation/blocked time)
GIVEN provider blocks March 25-27 as vacation
WHEN  customer views availability for those dates
THEN  no slots shown for March 25-27

TC-BOOK-016: Verify customer review after appointment
GIVEN appointment on March 20 is completed
WHEN  customer receives review request email and submits 4-star review
THEN  review posted on provider's profile, average rating updated
```

```
TC-BOOK-017: Verify group booking (multiple participants)
GIVEN "Yoga Class" accepts up to 15 participants, 13 currently booked
WHEN  customer books for 2 participants
THEN  15/15 capacity reached, class shows as "Full"

TC-BOOK-018: Verify booking form custom fields
GIVEN service "Consultation" has custom field "Reason for visit"
WHEN  customer fills in "Annual checkup" and books
THEN  provider sees "Reason for visit: Annual checkup" in booking details

TC-BOOK-019: Verify no-show marking and policy enforcement
GIVEN customer didn't show up for appointment
WHEN  provider marks booking as "No Show"
THEN  no-show fee applied if policy exists, customer notified, recorded in history

TC-BOOK-020: Verify calendar integration (Google Calendar / iCal)
GIVEN customer books appointment
WHEN  customer clicks "Add to Calendar"
THEN  .ics file downloads or Google Calendar event created with correct details
```

Applications 9-10: Marketplace and API-Driven App (40 test cases)

Application 9: Marketplace (20 test cases)

```
TC-MKTPL-001: Verify seller can create a product listing
GIVEN seller is logged in
WHEN  seller fills in title, description, price, photos, category and clicks "List"
THEN  listing appears on marketplace, status "Active"

TC-MKTPL-002: Verify buyer can search and filter listings
GIVEN 100 listings exist across categories
WHEN  buyer searches "laptop" and filters by price $500-$1000
THEN  only laptop listings in that price range displayed

TC-MKTPL-003: Verify buyer can place an order
GIVEN buyer selects a $599 laptop listing
WHEN  buyer clicks "Buy Now" and enters payment
THEN  order created, seller notified, payment held in escrow

TC-MKTPL-004: Verify escrow release after delivery confirmation
GIVEN order is delivered and buyer confirms receipt
WHEN  confirmation period (3 days) passes
THEN  payment released to seller minus marketplace fee

TC-MKTPL-005: Verify dispute opening by buyer
GIVEN buyer received item that doesn't match description
WHEN  buyer opens dispute with reason and photos
THEN  dispute created, seller notified, escrow hold extended

TC-MKTPL-006: Verify seller rating after transaction
GIVEN transaction is complete
WHEN  buyer leaves 4-star rating with comment
THEN  rating appears on seller profile, seller's average updated

TC-MKTPL-007: Verify listing expiration
GIVEN listing has 30-day active period
WHEN  30 days pass without sale
THEN  listing moves to "Expired", seller notified, option to renew
```

```
TC-MKTPL-008: Verify seller dashboard shows sales metrics
GIVEN seller has 10 completed sales this month
WHEN  seller views dashboard
THEN  shows: total revenue, number of orders, average rating, pending shipments

TC-MKTPL-009: Verify prohibited item detection
GIVEN platform prohibits weapon sales
WHEN  seller lists item with keywords flagged as weapons
THEN  listing rejected, seller notified of policy violation

TC-MKTPL-010: Verify buyer messaging to seller
GIVEN buyer has a question about a listing
WHEN  buyer sends message "Is this still available?"
THEN  message delivered to seller, seller can respond

TC-MKTPL-011: Verify shipping label generation
GIVEN seller confirms order for shipment
WHEN  seller clicks "Generate Label"
THEN  shipping label PDF with tracking number created

TC-MKTPL-012: Verify order tracking
GIVEN seller has shipped order with tracking number
WHEN  buyer views order status
THEN  tracking information shows carrier, status, estimated delivery

TC-MKTPL-013: Verify return/refund process
GIVEN buyer requests return within 14-day window
WHEN  seller approves return
THEN  return shipping label generated, refund processed after item received

TC-MKTPL-014: Verify seller verification/identity check
GIVEN new seller registers
WHEN  seller submits ID verification documents
THEN  documents reviewed, seller badge updated to "Verified" upon approval

TC-MKTPL-015: Verify category navigation
GIVEN categories Electronics > Laptops > Gaming Laptops exist
WHEN  buyer navigates through category tree
THEN  each level shows correct subcategories and listing counts

TC-MKTPL-016: Verify saved/favorite listings
GIVEN buyer views a listing
WHEN  buyer clicks "Save"
THEN  listing appears in buyer's "Saved Items", heart icon filled

TC-MKTPL-017: Verify price negotiation / make offer
GIVEN listing supports offers
WHEN  buyer offers $500 on a $599 listing
THEN  seller receives offer notification, can accept/reject/counter

TC-MKTPL-018: Verify seller payout schedule
GIVEN seller has $1,000 in cleared earnings
WHEN  weekly payout runs
THEN  $1,000 transferred to seller's bank account, payout receipt generated

TC-MKTPL-019: Verify multi-quantity listing
GIVEN seller lists 50 units of same item at $19.99 each
WHEN  buyer purchases 3 units
THEN  47 units remaining, buyer charged $59.97

TC-MKTPL-020: Verify promotional listing (featured placement)
GIVEN seller pays for featured listing
WHEN  buyers view the category page
THEN  featured listing appears in prominent position with "Featured" badge
```

Application 10: API-Driven App (20 test cases)

```
TC-API-001: Verify GET /users returns paginated user list
GIVEN 150 users exist in the system
WHEN  GET /api/v1/users?page=1&limit=20
THEN  response 200 with 20 users, total_count: 150, next_page: 2

TC-API-002: Verify POST /users creates a new user
GIVEN valid user payload with name, email, role
WHEN  POST /api/v1/users with JSON body
THEN  response 201, user created with generated ID, timestamps set

TC-API-003: Verify POST /users rejects duplicate email
GIVEN user with email "existing@test.com" exists
WHEN  POST /api/v1/users with same email
THEN  response 409, error "User with this email already exists"

TC-API-004: Verify PUT /users/:id updates user fields
GIVEN user #42 has name "John"
WHEN  PUT /api/v1/users/42 with {"name": "Jonathan"}
THEN  response 200, user name updated, updated_at timestamp changed

TC-API-005: Verify DELETE /users/:id removes user
GIVEN user #42 exists
WHEN  DELETE /api/v1/users/42
THEN  response 204, subsequent GET /users/42 returns 404

TC-API-006: Verify authentication required for protected endpoints
GIVEN no auth token provided
WHEN  GET /api/v1/users
THEN  response 401, error "Authentication required"

TC-API-007: Verify expired token is rejected
GIVEN auth token expired 1 hour ago
WHEN  GET /api/v1/users with expired token
THEN  response 401, error "Token expired"

TC-API-008: Verify rate limiting
GIVEN rate limit is 100 requests/minute
WHEN  101st request sent within 1 minute
THEN  response 429, "Rate limit exceeded", Retry-After header set

TC-API-009: Verify input validation — missing required field
GIVEN POST /api/v1/users requires "email" field
WHEN  request sent without email
THEN  response 400, error "email is required"

TC-API-010: Verify input validation — invalid email format
GIVEN POST /api/v1/users with email "not-an-email"
WHEN  request processed
THEN  response 400, error "email must be a valid email address"

TC-API-011: Verify filtering with query parameters
GIVEN users with roles "admin", "editor", "viewer" exist
WHEN  GET /api/v1/users?role=admin
```

```
THEN  response 200, only admin users returned

TC-API-012: Verify sorting with query parameters
GIVEN users created at different timestamps
WHEN  GET /api/v1/users?sort=created_at&order=desc
THEN  response 200, users ordered newest first

TC-API-013: Verify CORS headers for allowed origins
GIVEN allowed origin is "https://app.example.com"
WHEN  OPTIONS request from that origin
THEN  Access-Control-Allow-Origin header present

TC-API-014: Verify response format consistency
GIVEN any successful response
WHEN  checking response structure
THEN  consistent envelope: {"data": [...], "meta": {"total": N, "page": N}}

TC-API-015: Verify error response format consistency
GIVEN any error response
WHEN  checking response structure
THEN  consistent format: {"error": {"code": "...", "message": "..."}}

TC-API-016: Verify GET /users/:id for non-existent user
GIVEN user #99999 does not exist
WHEN  GET /api/v1/users/99999
THEN  response 404, error "User not found"

TC-API-017: Verify SQL injection attempt is handled
GIVEN malicious input in query parameter
WHEN  GET /api/v1/users?name='; DROP TABLE users; --
THEN  response 400 or empty results, no database damage

TC-API-018: Verify large payload rejection
GIVEN request body exceeds 1MB limit
WHEN  POST /api/v1/users with oversized payload
THEN  response 413, error "Payload too large"

TC-API-019: Verify webhook delivery on user creation
GIVEN webhook configured for "user.created" event
WHEN  new user is created via API
THEN  webhook POST sent to configured URL with user data

TC-API-020: Verify API versioning
GIVEN v1 returns field "name", v2 returns "first_name" and "last_name"
WHEN  GET /api/v1/users/1 and GET /api/v2/users/1
THEN  each version returns its respective field structure
```

PART VI

CAPSTONE PROJECT

Capstone: Complete Test Plan for a Real Application

The Assignment

Choose a publicly accessible web application (an e-commerce site, a productivity tool, a social platform — anything you use regularly). Create a complete test package that demonstrates every skill from this book.

Deliverables

Deliverable 1: Risk Assessment (Chapter 7)

Create a risk matrix covering at least 8 feature areas. For each, assess:

- Likelihood of defects (Very Low / Low / Medium / High)
- Business impact (Very Low / Low / Medium / High / Critical)
- Overall risk level
- Assigned test depth

Deliverable 2: Test Cases (Chapters 1-2)

Write 30 test cases using the techniques from this book:

- At least 5 using BVA
- At least 5 using EP
- At least 3 using a decision table
- At least 3 using state transitions
- At least 5 negative test cases
- At least 5 covering the happy path
- Use Given/When/Then format for all

Deliverable 3: Exploratory Testing (Chapters 3-4)

- Write 5 charters using the formula
- Execute 2 sessions (30 minutes each)
- Write complete session reports for both
- Create an aggregation table

Deliverable 4: Bug Reports (Chapters 5-6)

- File at least 3 bug reports for real issues you find
- Include all five essential fields
- Assign both severity and priority with justification
- Include at least one attachment (screenshot, HAR file, or console log)

Deliverable 5: Test Plan (Chapter 7)

Write a lightweight test plan covering:

- Scope (in scope / out of scope)
- Risk assessment summary
- Test approach
- Entry and exit criteria
- Schedule

Grading Yourself

Criteria	Excellent	Good	Needs Work
Risk assessment is realistic and justified	Risks clearly explained with evidence	Risks assigned but reasoning brief	Risks seem arbitrary
Test cases are clear and executable	A stranger could run them	Most are clear, a few are vague	Steps are ambiguous or missing data
BVA/EP/Decision Table applied correctly	Techniques used where appropriate	Mostly correct with minor errors	Techniques misapplied or missing
Exploratory sessions are documented	Rich notes, clear findings, good charters	Adequate notes, some gaps	Sparse notes, vague charters
Bug reports would enable reproduction	Developer could reproduce in 2 min	Developer could reproduce with 1 question	Developer would say “cannot reproduce”
Severity/priority correctly assessed	Both present and clearly justified	Present but justification weak	Missing or confused
Test plan is realistic and complete	All sections present, realistic schedule	Most sections present	Major gaps

APPENDICES

Appendix A: Glossary

Term	Definition
Actual Result	What the software actually did when a test was executed
Boundary Value Analysis (BVA)	Test design technique focusing on values at the edges of input ranges
Charter	A mission statement for an exploratory testing session
Decision Table	A matrix mapping condition combinations to expected outcomes
Debrief	A post-session discussion reviewing exploratory testing findings
Defect	A flaw in the software that causes incorrect or unexpected behavior (synonym: bug)
Entry Criteria	Conditions that must be met before testing begins
Equivalence Partitioning (EP)	Dividing inputs into groups expected to behave identically
Exit Criteria	Conditions that define when testing is complete

continued on next page

Term	Definition
Expected Result	What the software should do according to the specification
Exploratory Testing	Simultaneous test design, execution, and learning
Happy Path	The primary success scenario (everything goes right)
Negative Test	A test verifying the system handles incorrect input gracefully
Pairwise Testing	Testing all pairs of parameter combinations
Positive Test	A test verifying the system works with correct input
Precondition	The required system state before a test can be executed
Priority	Business urgency of fixing a defect (P0-P3)
Regression Testing	Re-testing after changes to ensure nothing previously working is broken
Risk-Based Testing	Allocating test effort based on probability and impact of failure
SBTM	Session-Based Test Management — structured approach to exploratory testing
Severity	Technical impact of a defect (Critical / High / Medium / Low)
Smoke Test	A quick test to verify basic functionality works
State Transition	A change from one state to another triggered by an event
Test Case	A set of conditions and steps to verify a specific behavior
Test Plan	A document describing the scope, approach, and schedule for testing
Validation	Confirming the product meets user needs (“right product”)

continued on next page

Term	Definition
Verification	Confirming the product meets specifications (“product right”)

Appendix B: Templates

Bug Report Template

```
Title: [What happens] when [condition/trigger]
Severity: [Critical / High / Medium / Low]
Priority: [P0 / P1 / P2 / P3]

Environment:
- Browser/Device:
- OS:
- Build/Version:
- Feature Flags:

Steps to Reproduce:
1.
2.
3.

Expected Result:


Actual Result:


Attachments:
- [ ] Screenshot
- [ ] Video
- [ ] HAR file
- [ ] Console log

Notes:
(Additional context, isolation attempts, workaround if known)
```

Session Report Template

```
Charter:
Tester:
Duration:
Start:
Environment:

AREAS COVERED:
-

BUGS FOUND:
-

QUESTIONS / RISKS:
-

FOLLOW-UP CHARTERS:
-
```

Lightweight Test Plan Template

```
Feature:
Release:
QA Lead:
Date:

SCOPE:
- In scope:
- Out of scope:

RISK ASSESSMENT:
| Feature Area | Risk Level | Test Depth |
|--------------|------------|------------|
|              |            |            |

TEST APPROACH:
- Functional:
- Exploratory:
- Cross-browser:
- Performance:

ENTRY CRITERIA:
-
```

```
EXIT CRITERIA:
-

SCHEDULE:
- Day 1:
- Day 2:
- Day 3:
```

Appendix C: Further Reading

- **"Lessons Learned in Software Testing"** by Cem Kaner, James Bach, Bret Pettichord — the classic, practical wisdom book
- **"Explore It!"** by Elisabeth Hendrickson — the definitive guide to exploratory testing
- **"A Practitioner's Guide to Software Test Design"** by Lee Copeland — deep dive into all test design techniques
- **"Perfect Software: And Other Illusions About Testing"** by Gerald Weinberg — understanding what testing can and cannot do
- **"Agile Testing"** by Lisa Crispin and Janet Gregory — how testing fits in Agile teams
- **ISTQB Foundation Syllabus** — the international standard body of knowledge (free PDF)
- **Ministry of Testing** (ministryoftesting.com) — community, articles, and courses
- **James Bach's blog** (satisfice.com) — exploratory testing thought leadership
- **Michael Bolton's blog** (developsense.com) — critical thinking about testing

Appendix D: Self-Assessment Quiz Answers

Chapter 1 Answers

1. ID, Title, Preconditions, Steps, Expected Result, Actual Result, Test Data

2. "Login test" is vague — which aspect? Better: "Verify login succeeds with valid email and password" or "Verify login fails with incorrect password"
3. Expected Result is what *should* happen (from the spec). Actual Result is what *did* happen (filled during execution).
4. GIVEN a premium user with active subscription / WHEN user logs in and navigates to analytics / THEN analytics dashboard loads with user's data
5. It's a compound step. Split into: "1. Set up test data [specify how]. 2. Log in with [specific credentials]. 3. Navigate to Settings page."

Chapter 2 Answers

1. 0 chars (empty), 1 char, 49 chars, 50 chars, 51 chars, and a "typical" mid-range value
2. Invalid low (0 or negative), Valid (1-99), Invalid high (100+)
3. 2^3 = 8 rules (Free trial Y/N × Paid Y/N × Admin Y/N). However, "Free trial" and "Paid" are likely mutually exclusive, reducing to 6 meaningful rules.
4. New → Processing → Shipped → Delivered. Cancelled connects from New and Processing. Invalid transitions: Shipped → Cancelled, Delivered → Cancelled.
5. Pairwise is for many parameters with few values each, where full combinations are impractical. Decision tables are for business rules with defined condition/action relationships. Use pairwise for configuration testing (browser × OS × language); use decision tables for business logic (discount rules, approval workflows).

Chapter 3 Answers

1. Charters, Sessions, and Debriefs
2. "Explore the social share button with different content types (text, image, link, video) to discover formatting or preview issues across platforms."
3. Without a time box, exploration becomes aimless. The constraint forces focus and creates a measurable unit of work for tracking and reporting.

4. Exploratory testing is structured (charter + time box + notes + debrief). Ad-hoc testing has no structure, no documentation, and no accountability.
5. Example for a calendar app: **S**tructure (monthly/weekly/daily views, event types), **D**ata (recurring events, timezones, all-day events), **T**ime (how do events behave at midnight, across DST changes, for multi-day events?)

This is Book 11 in the Modern QA Course Library. For the free overview course, visit modernQAcourse.com. For the complete 25-book library, visit [purchase link].

About This Sample

You have just read **Chapter 1 of Manual Testing Fundamentals: The Bedrock** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-11-manual-testing-fundamentals/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.