QA NOTEBOOK

Think. Test. Question. Write it down.

by Liudas Jankauskas
Automation Before Automation

THIS QA NOTEBOOK BELONGS TO

Name: _____

Team: _____

Role: _____

Good testing starts with questions, not test cases.

HOW TO USE THIS NOTEBOOK

This notebook is not a test case repository.

Use it to think.

Write down what changed.
What worries you.
What you don't understand.
What could break.
What users might do differently than expected.

Sketch flows. Capture weird behavior. Write hypotheses. Record questions before they disappear.

You don't need to fill every field.

The goal is not perfect documentation.

The goal is better testing.

Think first. Test second. Automate when it makes sense.

TESTING MINDSET / QUICK REMINDERS

Your job is not to prove that it works.
Your job is to learn where it might not.

Start with risk.
What matters most if this fails?

Question assumptions.
“What should happen?” is only the beginning.

Observe, don't just execute.
Look at responses, logs, timing, state changes and side effects.

A passing test proves very little.
It tells you what happened under one specific set of conditions.

A bug is not always an error message.
Wrong data, slow behavior, missing feedback or unexpected access can all be bugs.

Think beyond the happy path.
Real users do unexpected things.

Don't confuse coverage with confidence.
More tests do not automatically mean better testing.

QA NOTE
Testing is an investigation, not a checklist-completion exercise.

BEFORE YOU SAY “DONE”

Ask yourself:

☐ What changed?
☐ What can go wrong?
☐ Who could be affected?
☐ What assumptions did I make?
☐ What did I not test?
☐ What happens with invalid input?
☐ What happens at the boundaries?
☐ What happens when a dependency fails?
☐ What happens if the action is repeated?
☐ What happens with different permissions?
☐ What happens when it is slow?
☐ What happens over time?
☐ What happens outside my browser / device / environment?
☐ Did I look beyond the UI?
☐ What still makes me uncomfortable?

REMEMBER

You don't finish testing because there are no more tests.
You stop when the remaining risk is understood well enough.

SFDIPOT = San Francisco Depot

Based on James Bach's Heuristic Test Strategy Model and the Rapid Software Testing body of work.

A SIMPLE WAY TO FIND MORE TEST IDEAS

When you're stuck, look at the product from seven different angles.

S — STRUCTURE: What is it made of?

Components	Modules	Files	Database
APIs	Configuration	Dependencies	

F — FUNCTION: What does it do?

Features	Business rules	Calculations	Validation
Permissions	Error handling	User flows	

D — DATA: What does it process?

Valid / invalid	Empty / missing / null	Boundaries	Formats
Length	Encoding	Duplicates	Unexpected values

I — INTERFACES: What does it communicate with?

APIs	Databases	Third-party services	Authentication
Webhooks	Queues	External systems	

KEEP LOOKING

P — PLATFORM: Where does it run?

Browser	OS	Device	Screen size
Network	Locale	Timezone	Permissions

O — OPERATIONS: How will it be used?

Installation	Configuration	Deployment	Monitoring
Logging	Backup	Recovery	Maintenance

T — TIME: What changes with time?

Timeout	Expiration	Sessions	Scheduling
Timezones	Date boundaries	Long-running operations	Concurrency

QA NOTE

If you think you've run out of tests, run through SFDIPOT once more.

FUNCTIONAL IS NOT ENOUGH

A feature can be functionally correct and still be unusable, insecure or unreliable.

PERFORMANCE: Is it fast enough?

Response time	Load	Large data	Slow dependencies
Degradation over time			

SECURITY: Can someone do something they shouldn't?

Authentication	Authorization	Sensitive data	Input handling
Sessions	Information leakage		

RELIABILITY: Does it keep working?

Repeated use	Dependency failures	Unexpected state	Partial failures
Retries			

USABILITY: Can people actually use it?

Clarity	Feedback	Dangerous actions	Navigation
Errors			

ACCESSIBILITY: Can different users operate it?

Keyboard	Focus	Contrast	Labels
Screen readers	Zoom		

LOOK BEYOND FUNCTIONALITY

COMPATIBILITY: Where could behavior differ?

Browsers	Devices	OS	Screen sizes
Versions			

SCALABILITY: What happens when usage grows?

Users	Requests	Data	Connections
Background jobs			

RECOVERABILITY: What happens after failure?

Restart	Retry	Restore	Partial completion
Data consistency			

AVAILABILITY: What happens when something is unavailable?

API	Database	Network	Third party
Storage			

LOCALIZATION: What changes outside your locale?

Language	Dates	Time	Currency
Numbers	Text length		

OBSERVABILITY: If it fails in production, can we understand why?

Logs	Metrics	Traces	Errors
Alerts	Correlation IDs		

ASK ONE MORE QUESTION

What quality attribute am I ignoring because the feature appears to work?

WHAT AM I TESTING?

Feature / Endpoint / Build:

Date:

Environment:

Goal:

What changed?

What can go wrong?

Known risks:

What did I learn?

What is still uncertain?

Do I have enough confidence to move forward?
☐ Yes
☐ No
☐ With known risks

TESTING NOTES:

A successful response doesn't mean a correct response. Ask what should not have succeeded.

BUG INVESTIGATION

What happened?

What did I expect?

Can I reproduce it?

What changed?

What evidence do I have?

What am I assuming?

What is the smallest reproducible case?

INVESTIGATION NOTES

If it looks strange to you, imagine how it looks to the user.

TESTING IS NOT A DEMONSTRATION

Developers already know the feature works under the conditions they built and tested it for. Your job is not to demonstrate the happy path again. Your job is to discover when it stops working, under what conditions, and why that matters.

A useful testing question is not only: “Does it work?”
Ask instead: “When will it fail?”

Imagine the failure first, then work backwards.
What if the server is unavailable? What if the network is slow? What if the same action happens twice? Etc.

Someone may say: “Of course it fails if the server is down.”
Maybe. But if the application is critical, should it fail completely? Should another instance take over? Should a load balancer redirect traffic? Should the user see a meaningful degraded state instead of a crash? The obvious failure can still reveal an important requirement.

This way of thinking can be powerful: Imagine the bug already exists. Then try to prove it.

Instead of waiting to accidentally discover a problem, invent one: “I think this action can be executed twice and create duplicate data.” Now your testing has direction. You know what behavior you are trying to expose and can design conditions around it. This is backward thinking: start with a possible failure and work backwards toward the conditions that could produce it.

QA REMINDER
Do not spend all your time proving that the system works. Spend time discovering where its assumptions stop being true.

YOUR MOST IMPORTANT CUSTOMER MAY BE THE DEVELOPER

Testing creates information. That information is valuable only if it reaches the right people quickly, clearly, and with the right priority. Developers are among the most important consumers of your work. Help them understand the state of the product without forcing them to dig through noise. A tester who reports twenty cosmetic issues while a critical workflow is broken is not necessarily providing more value than someone who reports one important problem clearly.

Think about signal before volume. When you find a small issue, record it. Do not lose it. But you do not always need to interrupt the team immediately. Keep low-impact findings in your notes. Continue exploring. Look for the bigger risks first. Then, when you have stronger findings, bring the smaller issues with them.

Context changes how people perceive defects. A minor inconsistency reported alone can look like unnecessary noise. The same inconsistency presented together with evidence of a deeper quality problem may help explain a broader pattern.

Also respect developer attention. Before asking a developer: Check what you can investigate yourself. Reproduce the behavior several times. Look at the request and response. Check the console and logs you have access to. Record the environment and conditions. Separate facts from assumptions. Bring evidence, not only a question. The goal is to make communication useful.

QA REMINDER
Keep the small bugs. Prioritize the important ones. Do not hide information — organize it by value.