Tools - Git/Github/Postman/Testing/debugging

Here is **a complete interview-focused question bank for Git, GitHub, Postman, Testing & Debugging**.

These cover **all levels — fundamentals → deep concepts → real-time scenario questions** expected for 1–3 YOE Full Stack developers.

🔥 GIT / GITHUB INTERVIEW QUESTIONS

📌 Git Basics

✅ 1) What is Git? Why is it used?

■ Explanation (simple, conceptual)

Git is a **distributed version control system** that tracks changes in your source code over time. It lets multiple developers work on the same project without overwriting each other's work.

Every developer has a **local copy** of the repository, making Git fast, safe, and reliable.

■ When/Why is it used?

We use Git because:

- It helps maintain **version history** of code.
- Enables **collaboration** between developers.
- Allows **branching & merging** for parallel feature development.
- Supports **rollback** to any previous stable version.
- Works offline due to its distributed nature.

■ Example / Code snippet

```
git init
git add .
git commit -m "Initial commit"
```

■ Short summary line

👉 *Git is a distributed version control system that tracks code changes and supports team collaboration.*

✅ 2) What is the difference between Git and GitHub?

■ Explanation (simple, conceptual)

Git is the **tool** used for version control.

GitHub is a **cloud-based hosting platform** used to store Git repositories and collaborate.

■ When/Why is it used?

Use Git for:

- Local development
- Branching/committing
- Tracking code changes

Use GitHub for:

- Centralized team collaboration
- Pull requests and code reviews
- CI/CD pipelines
- Backup of repositories

■ Example / Code snippet

```
git remote add origin https://github.com/user/project.git
git push -u origin main
```

■ Short summary line

👉 *Git is a version control system; GitHub is a platform to store and collaborate on Git repositories.*

✅ 3) What is a repository?

■ Explanation (simple, conceptual)

A repository (repo) is a **storage location** where all project files and their version history are kept.

It contains:

- Source code
- Branches
- Commits history
- Configuration files

■ When/Why is it used?

Repos are used to:

- Maintain structured versions of a project
- Enable collaboration across teams
- Organize project modules and history

■ Example / Code snippet

Creating a local repo:

```
git init
```

Cloning a remote repo:

```
git clone https://github.com/user/repo.git
```

■ Short summary line

👉 *A repository is a project storage that contains code along with its entire version history.*

✅ 4) Local repo vs Remote repo

■ Explanation (simple, conceptual)

- A **local repository** exists on your local system. It stores your branches, commits, and changes.
- A **remote repository** exists on a cloud platform like GitHub, GitLab, Bitbucket, Azure DevOps.

■ When/Why is it used?

Local repo:

✔️ Code changes, experiments, commits

✔️ Work offline

Remote repo:

✔️ Team collaboration

✔️ Pull/Pull Requests

✔️ Backup & CI/CD integrations

■ Example / Code snippet

```
Local repo:     git commit -m "Update UI"
Remote repo:    git push origin feature/login
```

■ Short summary line

👉 *Local repo is your private workspace; remote repo is the shared version used for team collaboration.*

✅ 5) What is staging area / index?

■ Explanation (simple, conceptual)

The staging area (index) is an intermediate space where Git stores files *before* committing them.

It allows you to review or choose what exactly will go into the next commit.

■ When/Why is it used?

Use staging when:

- You want to commit **selected files**, not everything
- You want to organize commits logically
- You want to review changes before finalizing

■ Example / Code snippet

```
git add index.html
git add src/app/home.component.ts
```

■ Short summary line

👉 *The staging area is a preparation zone where files are placed before making a commit.*

✅ 6) What is a commit? What is a hash?

■ Explanation (simple, conceptual)

A **commit** is a snapshot of your project at a specific point in time.

Each commit represents a meaningful change or feature update.

A **hash** is a unique 40-character SHA-1 identifier that Git generates for every commit.

It guarantees uniqueness and helps locate any version instantly.

■ When/Why is it used?

We use commits to:

- Save progress of work
- Organize meaningful units of change
- Track who changed what and when

We use hash to:

- Identify specific commits
- Roll back or cherry-pick changes
- Compare versions

■ Example / Code snippet

```
git commit -m "Added login validation"
git log
```

Output shows hash:

```
commit a3f5c9d7b12340ea89c63f23bc...
```

■ Short summary line

👉 *A commit is a saved snapshot of code; a hash uniquely identifies that snapshot.*

📌 Core Commands (Very important)

✅ 1) `git init` vs `git clone`

■ Explanation (simple, conceptual)

- `git init` → creates a **new empty Git repository** in your local machine.
- `git clone` → copies an **existing remote repository** into your local machine.

■ When/Why is it used?

Use **git init** when:

- Starting a brand-new project from scratch.
- Converting an existing folder into a repository.

Use **git clone** when:

- You want to work on an already existing project from GitHub / GitLab.
- Collaborating with a team.

■ Example / Code snippet

```
git init        // Start new repo
git clone https://github.com/user/project.git   // Download remote repo
```

■ Short summary line

👉 `git init` *creates a new repo;* `git clone` *copies an existing remote repo.*

✅ 2) `git add .` vs `git add filename`

■ Explanation (simple, conceptual)

- `git add .` → stages **all modified and new files** in the folder.
- `git add filename` → stages **only one specific file**.

■ When/Why is it used?

Use `git add .` when:

- You want to quickly stage all changes (common in small commits).

Use `git add filename` when:

- You want **clean, meaningful commits** with only selected files.
- You want to avoid adding unwanted files.

■ Example / Code snippet

```
git add .                  // Add everything
git add src/app/login.ts   // Add specific file
```

■ Short summary line

👉 `git add .` *stages all files;* `git add filename` *stages specific files.*

✅ 3) `git commit -m ""`

■ Explanation (simple, conceptual)

A commit records the staged changes with a **message** describing what you changed.

- `m ""` lets you write that message directly.

■ When/Why is it used?

Use it when:

- You want to save your work with a meaningful description.
- You want traceability in team environments.

Good commit messages help teammates understand changes without reading the entire diff.

■ Example / Code snippet

```
git commit -m "Added login validation"
git commit -m "Fixed API response mapping for dashboard"
```

■ Short summary line

👉 `git commit -m` *saves staged changes with a descriptive, searchable message.*

✅ 4) `git status` vs `git log`

■ Explanation (simple, conceptual)

- `git status` → shows **current working directory state** (untracked, modified, staged files).
- `git log` → shows the **history of commits** with author, date, and hash.

■ When/Why is it used?

Use `git status` when:

- You want to check which files changed before committing.
- You want to confirm what's staged vs unstaged.

Use `git log` when:

- You want to review commit history.

- You want to find a commit hash for revert, reset, or debugging.

■ Example / Code snippet

```
git status
git log --oneline
```

■ Short summary line

👉 `git status` *checks current changes;* `git log` *checks commit history.*

✅ 5) `git push` vs `git pull` vs `git fetch`

■ Explanation (simple, conceptual)

- `git push` → sends your local commits to the remote repo.
- `git pull` → fetches remote changes **and merges them** into your branch.
- `git fetch` → downloads remote changes **without merging** (safe preview).

■ When/Why is it used?

Use **git push**:

- When you want to publish your changes for the team.

Use **git pull**:

- When you want the latest code + auto-merge.

Use **git fetch**:

- When you want to inspect remote changes before merging.
- Prevent accidental merge conflicts.

■ Example / Code snippet

```
git push origin main
git pull origin main
git fetch origin
```

■ Short summary line

👉 *Push uploads changes, pull downloads + merges, fetch only downloads.*

✅ 6) What does `git diff` show?

■ Explanation (simple, conceptual)

`git diff` shows **line-by-line differences** between:

- Your working directory and staging area

- Or between commits/branches when specified

It highlights additions, deletions, and modifications.

■ When/Why is it used?

Use it to:

- Review changes before staging
- Understand code differences during merge conflicts
- Validate what exactly changed in a commit

■ Example / Code snippet

```
git diff            // Compare working directory vs staging
git diff --staged   // Compare staging vs last commit
git diff main dev   // Compare branches
```

■ Short summary line

👉 `git diff` *shows detailed differences between files or commits.*

✅ 7) How to undo commit?

■ Explanation (simple, conceptual)

There are multiple ways depending on the situation.

■ When/Why is it used?

Case 1: Undo last commit but keep changes

Use when you committed something by mistake but still want the code.

```
git reset --soft HEAD~1
```

Case 2: Undo commit AND remove changes

Use when commit + code both are wrong.

```
git reset --hard HEAD~1
```

Case 3: Undo commit after pushing (safe way)

Use *revert* when others may have pulled your changes.

```
git revert <commit-hash>
```

This creates a **new commit** that reverses the bad one.

■ Example / Code snippet

Undo local commit (keep files):

```
git reset --soft HEAD~1
```

Undo commit on shared branch:

```
git revert a3c9f...
```

■ Short summary line

👉 *Use reset for local undo; use revert for safe undo after pushing.*

📌 Branching & Merging

✅ 1) What is a branch? Why use it?

■ Explanation (simple, conceptual)

A branch is an independent line of development that allows you to work on features, bug fixes, or experiments **without touching the main codebase**.

Each branch starts from a commit and contains its own changes until merged.

■ When/Why is it used?

We use branches for:

- **Parallel development** (multiple developers working independently)
- **Feature isolation** (each feature in its own branch)
- **Safe experimentation** without breaking production
- **Organized workflows** like Git Flow or trunk-based development

In a real project, teams use branches like:

- `feature/login-page`
- `bugfix/report-not-loading`
- `hotfix/authentication-issue`

■ Example / Code snippet

```
git branch feature/payment
git checkout feature/payment
```

■ Short summary line

👉 *A branch is an isolated workspace that helps develop features safely without affecting the main code.*

✅ 2) `git branch` vs `git checkout -b`

■ Explanation (simple, conceptual)

- `git branch` → **creates a new branch**, but does NOT switch to it.
- `git checkout -b` → **creates a branch AND switches** to it in one command.

■ When/Why is it used?

Use `git branch` when:

- You want to create multiple branches without switching.

Use `git checkout -b` when:

- You want to start working on the new branch immediately.
- It saves time and avoids two separate commands.

■ Example / Code snippet

```
git branch dev          // Only creates branch
git checkout dev         // Switch to dev

git checkout -b feature/ui  // Create + switch in one step
```

■ Short summary line

👉 *`git branch` only creates a branch; `git checkout -b` creates and switches to it.*

✅ 3) What is merge conflict? How do you resolve it?

■ Explanation (simple, conceptual)

A merge conflict happens when Git cannot automatically merge changes because **two developers modified the same line or same file section**.

Git stops merging and asks you to manually fix the conflict.

■ When/Why is it used?

Occurs during:

- `git merge`
- `git pull`
- `git rebase`

Conflicts typically happen when:

- Two changes overlap
- A file is edited and deleted in different branches
- Same function is modified by two developers

■ Example / Code snippet

Conflict example inside file:

```
<<<<<<< HEAD
  title = "Dashboard";
=======
  title = "User Dashboard Page";
>>>>>>> feature/new-ui
```

Resolution steps:

1. Open the conflicted files
2. Decide correct or combined code
3. Remove conflict markers (`<<<<`, `====`, `>>>>`)
4. Stage resolved file

   ```
   git add filename
   ```

5. Complete merge

   ```
   git commit
   ```

■ Short summary line

👉 *A merge conflict occurs when two branches modify the same code; you resolve it by manually choosing the correct content.*

✅ 4) Fast-forward merge vs three-way merge

■ Explanation (simple, conceptual)

Fast-forward merge:

- Happens when the target branch has no new commits.
- Git simply moves the branch pointer forward.
- No merge commit is created.

Three-way merge:

- Happens when both branches have diverged.
- Git needs to compare **two branch tips + their common ancestor**.
- Creates a merge commit.

■ When/Why is it used?

Use **fast-forward** when:

- Feature branch is strictly ahead of main
- You want a clean, linear history

Use **three-way merge** when:

- Both branches have commits
- You want to preserve the merge context
- Typical in team workflows

■ Example / Code snippet

Fast-forward:

```
git checkout main
git merge feature/payment
```

Three-way merge:

```
git merge feature/api   // Creates a merge commit because branches diverged
```

■ Short summary line

👉 *Fast-forward moves branch pointer; three-way merge creates a merge commit when branches diverge.*

✅ 5) Rebase vs Merge — difference & use case

■ Explanation (simple, conceptual)

Merge:

- Combines two branches by creating a merge commit.
- Maintains full history of how development happened.
- History becomes **non-linear**.

Rebase:

- Moves your branch commits on top of another branch.
- Rewrites commit history.
- Produces a **clean, linear history**.

■ When/Why is it used?

Use **Merge** when:

- You want to preserve original history
- Multiple developers are working together
- Working on shared branches (safe)

Use **Rebase** when:

- You want clean, linear history
- You want to avoid unnecessary merge commits

- Working on private/local branches

Important rule:

🚫 *Never rebase a branch that is already pushed and used by the team.*

It rewrites history and causes conflicts for others.

■ Example / Code snippet

Merge:

```
git checkout main
git merge feature/logs
```

Rebase:

```
git checkout feature/logs
git rebase main
```

■ Short summary line

👉 *Merge preserves history with merge commits; Rebase creates a clean linear history by rewriting commits.*

📌 Hard / Soft / Mixed Reset (Most asked)

✅ 1) `git reset --soft`

■ Explanation (simple, conceptual)

`git reset --soft` moves **only the HEAD pointer** to a previous commit but **keeps all your changes in the staging area**.

Nothing is deleted. Nothing is lost.

It simply **undoes a commit**, but the code stays staged.

■ When/Why is it used?

Use `--soft` when:

- You want to rewrite or improve the previous commit message.
- You mistakenly committed but still want the same changes in staging.
- You want to combine multiple commits into one clean commit (squash strategy).

Example scenario:

You committed “Added validation” but forgot to add one more file.

Soft reset lets you redo the commit cleanly.

■ Example / Code snippet

```
git reset --soft HEAD~1
```

Effect:

✔️ Commit undone

✔️ All changes still staged

✔️ Ready to recommit

■ Short summary line

👉 *Soft reset removes commit but keeps changes staged—best for fixing or rewriting commits.*

✅ 2) `git reset --mixed` *(default reset)*

■ Explanation (simple, conceptual)

`git reset --mixed` moves the HEAD back and **unstages the files**, but **keeps your changes in the working directory**.

This means:

- Commit is undone
- Changes are NOT staged
- But code changes are still present

It is the **default behavior** when you run `git reset HEAD~1`.

■ When/Why is it used?

Use `--mixed` when:

- You want to undo commit **and re-select** what to stage
- You committed too many files unintentionally
- You want to clean up your staging area

Real-time scenario example:

You accidentally staged and committed 10 files but only 3 were needed.

■ Example / Code snippet

```
git reset --mixed HEAD~1
```

Effect:

✔️ Commit undone

✔️ Files become unstaged

✔️ Code still exists on disk

■ Short summary line

👉 *Mixed reset removes commit and unstages files but keeps the actual code changes.*

✅ 3) `git reset --hard` *(dangerous — deletes changes)*

■ Explanation (simple, conceptual)

`git reset --hard` resets:

- **HEAD pointer**
- **Index (staging area)**
- **Working directory**

to the selected commit.

This means **all changes after that commit are permanently deleted**.

■ When/Why is it used?

Use **only when**:

- You want to discard all local changes completely
- Your branch is in a broken state and you want a fresh start
- You want to exactly match a stable commit

🚨 **Warning:**

- Changes are lost permanently
- Cannot recover unless already pushed or in reflog
- Never use this if you have uncommitted work you need later

■ Example / Code snippet

```
git reset --hard HEAD~1
```

Effect:

❌ Commit removed

❌ Staging cleared

❌ Local code changes deleted

✔️ Branch exactly matches previous commit

■ Short summary line

👉 *Hard reset deletes commits AND file changes—use carefully because it permanently removes work.*

📌 GitHub Specific

✅ 1) Fork vs Clone vs Branch

■ Explanation (simple, conceptual)

Fork:

- A fork is a **complete copy** of someone else's repository into **your own GitHub account**.
- Used mainly in open-source contributions.

Clone:

- A clone is a **local copy** of a remote repository on your computer.
- It allows you to work offline and push changes later.

Branch:

- A branch is a **separate line of development** inside the same repository.
- Used for features, bug fixes, experiments.

■ When/Why is it used?

Use **Fork** when:

- You don't have write access to original repo.
- You want to contribute to public/open-source projects.

Use **Clone** when:

- You want to download an existing remote repo locally.
- You want to work on your own project with full access.

Use **Branch** when:

- Working with a team on feature-based workflow.
- Wanting isolated development without affecting main branch.

■ Example / Code snippet

```
git clone https://github.com/user/project.git
git checkout -b feature/login
```

■ Short summary line

👉 *Fork copies repo to your GitHub account, Clone copies repo to your local system, Branch creates isolated development inside same repo.*

✅ 2) Pull Request vs Merge Request

■ Explanation (simple, conceptual)

Both PR (Pull Request) and MR (Merge Request) mean the same concept:

→ A request to merge one branch's changes into another branch.

Pull Request (PR)

- Popular term used in **GitHub**.

Merge Request (MR)

- Term used in **GitLab, Bitbucket, Azure DevOps**.

■ When/Why is it used?

A PR/MR is used for:

- Reviewing code before merging
- Discussing changes with reviewers
- Ensuring quality, testing & standards
- Maintaining clean Git workflow

Teams use PR/MR to avoid directly pushing into main branch.

■ Example / Code snippet

In GitHub:

```
Click "New Pull Request"
Source: feature/login
Target: main
```

■ Short summary line

👉 *Pull Request (GitHub) and Merge Request (GitLab) are same—they request to merge one branch into another with review.*

✅ 3) Code Review Best Practices

■ Explanation (simple, conceptual)

Code review ensures clean, efficient, consistent, and bug-free code before merging.

It is a quality gate for every merge.

■ When/Why is it used?

Use code reviews to:

- Catch bugs early
- Maintain coding standards
- Improve readability & maintainability
- Avoid performance and security issues
- Ensure consistent patterns across team

■ Example / Best practices list

As a reviewer:

- Check for readable, clean, maintainable code
- Ensure meaningful variable/function names
- Review API validations & error handling
- Verify SQL queries for performance

- Check unit tests, logs, and exceptions
- Avoid duplication (DRY principle)
- Ensure code follows team conventions
- Verify no secrets (passwords/API keys) are committed

As a developer receiving review:

- Write small and focused commits
- Attach screenshots for UI changes
- Add proper comments and documentation
- Be open to feedback
- Avoid committing unnecessary files

■ Short summary line

👉 *Code reviews ensure quality, consistency, readability, and prevent bugs before merging.*

✅ 4) What are GitHub Actions (CI/CD basics)?

■ Explanation (simple, conceptual)

GitHub Actions is a **CI/CD automation platform** built into GitHub.

It allows you to automate workflows such as:

- Running tests
- Building your application
- Deploying to servers or cloud
- Checking coding standards

It uses YAML files placed in `.github/workflows/`.

■ When/Why is it used?

Use GitHub Actions for:

- Automated testing on every push
- Build & deploy pipelines
- Automatic code checks (linting/security)
- Continuous Integration (team commits don't break code)
- Continuous Deployment (auto deploy after merge)

Real-time example:

When you push code to `main`, CI runs unit tests.

If everything passes, CD deploys the app to production.

■ Example / Code snippet

Simple CI pipeline example:

```
name: CI Pipeline

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Install dependencies
      run: npm install
    - name: Run tests
      run: npm test
```

■ Short summary line

👉 *GitHub Actions automates CI/CD pipelines like testing, building, and deployment using YAML workflows.*

📌 Real-time Scenario Questions

✅ 1) You committed the wrong file — how will you revert?

■ Explanation (simple, conceptual)

Reverting a wrong file depends on whether the commit was pushed and whether you want to remove the file from history or just undo the commit locally. Options: **amend**, **reset**, or **revert**.

- If the commit is **local and not pushed**, you can amend or reset.
- If the commit is **already pushed/shared**, use `git revert` or create a new commit that removes the file to avoid rewriting shared history.

■ When/Why is it used?

Use these approaches when you accidentally committed sensitive files, build artifacts, or simply wrong code and you want to:

- Keep history clean (local only)
- Avoid breaking teammates' copies (shared)
- Remove sensitive data (rotate keys + remove from history with caution)

■ Example / Code snippet

A. Local commit, not pushed — amend (remove file from last commit but keep changes unstaged):

```
# Unstage the file from index and remove it from last commit
git reset HEAD^ -- path/to/wrong-file.txt   # move commit pointer back but keep changes
```

```
git restore --staged path/to/wrong-file.txt # unstage the file if needed
# Recommit only intended files
git commit -m "Fix: commit without wrong file"
```

Or simpler:

```
git rm --cached path/to/wrong-file.txt      # remove file from repo but keep locally
git commit --amend --no-edit                # amend last commit to exclude it
```

B. Commit already pushed — safe approach using revert:

```
# Create a new commit that undoes the changes from the bad commit
git revert <bad-commit-hash>
git push origin feature/branch
```

C. If you accidentally pushed sensitive data (advanced & risky):

- Rotate secrets immediately (external step).
- Remove from history using `git filter-repo` or `git filter-branch` (dangerous; coordinate with team).
- Force-push only after team agreement:

```
# after rewriting history (example)
git push --force-with-lease origin branch
```

■ Short summary line

👉 *If not pushed use reset/amend; if pushed create a safe undo with `git revert`; never rewrite shared history without team coordination.*

✅ 2) You want to update your branch with latest main — how?

■ Explanation (simple, conceptual)

To bring your feature branch up-to-date with `main`, you can either **merge main into your branch** or **rebase your branch onto main**. Merge preserves history and creates a merge commit; rebase rewrites commits to create a linear history.

■ When/Why is it used?

- **Merge** when you want to preserve the true history and avoid rewriting commits (safe for shared branches).
- **Rebase** when you want a clean linear history and you are working on a private/local branch (do not rebase public/shared commits).

■ Example / Code snippet

A. Merge (safe for shared work):

```
git checkout feature/my-feature
git fetch origin
git merge origin/main        # merges latest main into your branch
# resolve conflicts if any, then
git push origin feature/my-feature
```

B. Rebase (clean history — for local/private branches):

```
git checkout feature/my-feature
git fetch origin
git rebase origin/main       # replay your commits on top of main
# resolve conflicts during rebase, then
git push --force-with-lease origin feature/my-feature   # only if branch is private
```

C. Safer pattern (fetch + rebase interactively then push):

```
git fetch origin
git switch feature/my-feature
git rebase --onto origin/main
# run tests locally, then
git push --force-with-lease origin feature/my-feature
```

Practical tip:

- Always run tests and lint after merging/rebasing and before pushing. Use `-force-with-lease` instead of `-force` to avoid clobbering others' changes.

■ Short summary line

👉 *Merge to safely integrate main into your branch; rebase to produce a linear history—only rebase private branches and coordinate before force-pushing.*

✅ 3) Two developers edited same file — conflict handling steps?

■ Explanation (simple, conceptual)

A merge conflict happens when Git cannot automatically reconcile overlapping changes. You must **manually resolve** the conflicting parts, test, and complete the merge or rebase.

■ When/Why is it used?

Conflicts occur during `git merge`, `git pull`, or `git rebase` when edits overlap. Proper conflict resolution avoids regressions and ensures the intended behavior remains intact.

■ Example / Resolution steps

1. **Identify conflict** — Git reports conflicted files:

```
git status
```

2. **Open conflicted files** — you'll see markers:

```
<<<<<<< HEAD
// your branch changes
=======
// incoming branch changes
>>>>>>> feature/other
```

3. **Decide resolution** — pick one side, merge code, or create a combined solution. Prefer minimal, well-tested logic. In .NET/Angular projects, ensure imports, routing, and DI registrations remain consistent.
4. **Edit & remove markers** — produce the final correct code.
5. **Run tests / build locally** — e.g., `dotnet build`, `ng serve` or unit tests to verify.
6. **Stage resolved files** and complete merge:

```
git add src/app/xyz.component.ts
git commit   # or `git rebase --continue` if rebasing
git push
```

7. **Communicate** — inform the other developer and, if needed, add a short note in PR about the resolution reasoning.

Tools to help:

- Use IDE merge tools (VS Code, Visual Studio) or GUI (Sourcetree, GitKraken).
- Use `git mergetool` to invoke configured merge tool.

Practical tips:

- Resolve logically, not just by choosing one side. Combine the best of both when possible.
- If conflict resolution is complex, create a small unit test to capture expected behavior before finalizing.

■ Short summary line

👉 *Resolve conflicts by editing conflict markers, testing the merged code, staging, and committing; always validate via build/tests and communicate changes.*

✅ 4) How do you maintain version control in a team?

■ Explanation (simple, conceptual)

Maintaining version control in a team is about **process, tools, and discipline**. It includes branching strategy, code review workflows, CI/CD, protected branches, consistent commit messages, and automated checks to ensure code quality and stability.

■ When/Why is it used?

To avoid merge chaos, maintain a stable main branch, enable reliable releases, and make collaboration scalable and auditable. Good practices reduce bugs, speed up reviews, and make rollbacks and audits straightforward.

■ Example / Practices & Tooling

Processes:

- Adopt a branching model (e.g., *Git Flow* or *Trunk-based*). Example for feature flow:
  - `main` (production)
  - `develop` or directly `main` with short-lived feature branches: `feature/*`, `bugfix/*`, `hotfix/*`
- Use **Pull Requests** for every merge into `main` or `develop`.
- Enforce **small, focused PRs** with descriptive titles and messages.

Quality Gates:

- Protect `main` with branch protection rules (require PR reviews, passing CI, no force pushes).
- Require at least 1–2 reviewers and successful CI checks before merge.

Automation:

- CI pipelines run unit tests, lint, static analysis, and integration tests (GitHub Actions, Azure DevOps pipelines).
- Automated deployment pipelines (CD) for staging and production with gated approvals.

Standards & Communication:

- Agree on **commit message conventions** (e.g., Conventional Commits). Example: `feat(auth): add JWT refresh endpoint`.
- Keep `.gitignore` accurate to avoid committing build artifacts or secrets.
- Use feature flags for incremental rollout.
- Use tags/releases for versioning (semantic versioning `v1.2.3`).

Backup & Recovery:

- Regular backups of remote repos (hosted services usually handle this).
- Use `git reflog` and backups for local recoveries when needed.

Example Commands / Configuration:

- Protect branch on GitHub: require pull request reviews & status checks.
- CI sample (very short):

```
# .github/workflows/ci.yml (concept)
on: [pull_request]
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
```

```
    - name: Setup .NET
      uses: actions/setup-dotnet@v2
    - run: dotnet restore && dotnet build && dotnet test
```

■ Short summary line

👉 *Maintain team version control with a clear branching model, protected branches, PR reviews, automated CI/CD, consistent commit conventions, and good communication.*

🔥 POSTMAN INTERVIEW QUESTIONS

📌 Basics

✅ 1) What is Postman? Why do we use it?

■ Explanation (simple, conceptual)

Postman is an **API development and testing tool** used to send HTTP requests and inspect responses.

It provides an easy UI to test REST APIs without writing code.

Developers use it to validate API behavior, debug issues, automate tests, and collaborate across teams.

■ When/Why is it used?

We use Postman when we want to:

- Test **GET/POST/PUT/DELETE** APIs quickly
- Validate backend logic independently of UI
- Debug server errors (400, 401, 500, etc.)
- Automate test cases for APIs
- Share API collections with the team
- Mock APIs when backend or frontend is not ready
- Visualize responses like JSON, headers, cookies

In real projects:

Developers test API endpoints during feature development, QA verifies business logic, and DevOps uses it for monitoring and regression.

■ Example / Use-case

Testing a login API manually:

```
POST /api/auth/login
Body:
{
  "username": "mohini",
```

```
  "password": "12345"
}
```

Postman shows:

✔️ Status code

✔️ Response time

✔️ Response body

✔️ Headers

■ Short summary line

👉 *Postman is a tool to send and test API requests, debug responses, and automate API testing.*

✅ 2) How do you send GET/POST/PUT/DELETE requests?

■ Explanation (simple, conceptual)

Postman allows you to select an HTTP request method and enter the API URL.

Depending on the method, you can send data through query params, headers, or body.

■ When/Why is it used?

Each method is used for different API operations:

Method	Purpose
GET	Fetch data
POST	Create new data
PUT	Update existing data (idempotent)
DELETE	Remove data

This helps developers test CRUD operations without needing a UI.

■ Example / Code snippet

GET request

```
GET https://api.myapp.com/users
```

POST request

```
POST https://api.myapp.com/users
Body (JSON):
{
  "name": "Mohini",
  "email": "mohini@test.com"
}
```

PUT request

```
PUT https://api.myapp.com/users/101
Body:
{
  "name": "Updated Name"
}
```

DELETE request

```
DELETE https://api.myapp.com/users/101
```

Steps in Postman:

1. Select GET/POST/PUT/DELETE from dropdown
2. Enter API URL
3. Add Headers (like `Content-Type: application/json`)
4. Add Body (for POST/PUT)
5. Click "Send"

■ Short summary line

👉 *In Postman, choose HTTP method → enter URL → add headers/body → hit Send to test API behavior.*

✅ 3) What are Headers, Body, Params?

■ Explanation (simple, conceptual)

These three define how data is sent to the server in an HTTP request.

Headers

- Key–value pairs that send metadata.
- Example: authentication token, content-type, caching info.

Body

- The main data sent in POST/PUT requests.
- Usually JSON, form-data, or raw text.

Params

- Query parameters appended in URL to filter or search.
- Example: `?page=1&limit=10`.

■ When/Why is it used?

Use **Headers** when sending:

- `Content-Type: application/json`
- `Authorization: Bearer <token>`

- Custom headers for APIs

Use **Body** when sending:

- User input (login data, form data, payload)
- Large JSON objects
- File uploads (form-data)

Use **Params** when:

- Filtering lists (`?status=active`)
- Pagination
- Sorting

■ Example / Code snippet

Headers example

```
Authorization: Bearer eyJhbGci...
Content-Type: application/json
```

Body example (POST)

```
{
  "name": "Mohini",
  "age": 23
}
```

Params example

```
GET /users?page=1&limit=20
```

■ Short summary line

👉 *Headers carry metadata, Body carries data in POST/PUT, and Params add filters or query values in the URL.*

📌 API Testing Concepts

✅ 1) Different content types in Postman (`JSON` , `XML` , `form-data`)

■ Explanation (simple, conceptual)

Content types tell the server **what kind of data** is being sent in a request body.

Postman allows sending different data formats based on the API requirement.

The most common formats are:

- **JSON (application/json)** – Used in REST APIs, easy for both client/server.

- **XML (application/xml)** – Older systems, SOAP-based APIs.
- **form-data (multipart/form-data)** – Used for file uploads or sending key–value pairs.
- **x-www-form-urlencoded** – Form submissions; data sent as URL-encoded strings.

■ When/Why is it used?

Use **JSON** when:

✔ Sending structured data (backend expects JSON)

✔ Modern REST APIs (.NET, Node, Java, Spring)

Use **XML** when:

✔ Working with legacy systems, banking APIs, SOAP services

Use **form-data** when:

✔ Uploading files (images, documents)

✔ Sending large multi-part content

✔ Submitting fields like text + file together

Use **x-www-form-urlencoded** when:

✔ Standard HTML form submissions

✔ OAuth token requests (common in authentication flows)

■ Example / Code snippet

JSON example

```
{
  "username": "mohini",
  "password": "1234"
}
```

XML example

```
<User>
   <Name>Mohini</Name>
   <Role>Admin</Role>
</User>
```

form-data example

```
key: profilePic     value: <choose file>
key: name           value: Mohini
```

■ Short summary line

👉 *JSON for structured REST data, XML for legacy/SOAP APIs, and form-data for file uploads or mixed inputs.*

✅ 2) How do you pass query parameters?

■ Explanation (simple, conceptual)

Query parameters are **key–value pairs** added at the end of the API URL to filter, search, or paginate results.

Format:

```
?key=value&key2=value2
```

■ When/Why is it used?

Use query parameters when you need:

- Pagination (`page=1&limit=10`)
- Filtering (`status=active`)
- Searching (`search=invoice`)
- Sorting (`sortBy=date`)

APIs commonly use them for GET requests.

■ Example / Code snippet

Manual URL example:

```
GET /users?page=1&limit=20&sort=name
```

Postman UI:

- Go to **Params** tab
- Add:

  ```
  Key: page      Value: 1
  Key: limit     Value: 20
  ```

Postman automatically generates:

```
/users?page=1&limit=20
```

■ Short summary line

👉 *Query params are added after* `?` *to filter, search, or paginate API data.*

✅ 3) What is the Authorization tab used for?

■ Explanation (simple, conceptual)

The Authorization tab is where you specify **how the client will authenticate** with the server.

Many APIs require credentials, tokens, or keys before allowing access.

Postman supports multiple auth types:

- API Key
- Basic Auth (username/password)
- Bearer Token
- OAuth 2.0
- JWT
- AWS Signature
- No Auth

■ When/Why is it used?

Use the Authorization tab when:

- Testing secure APIs
- Sending tokens from login services
- Calling APIs that need authentication
- Passing API keys or credentials safely

This saves time compared to manually adding Authorization headers each time.

■ Example / Code snippet

Basic Auth example

```
username: admin
password: admin123
```

Bearer token example

```
Authorization: Bearer <token>
```

Postman auto-fills headers for you.

■ Short summary line

👉 *Authorization tab is used to add authentication (API keys, tokens, OAuth) required for secure APIs.*

✅ 4) What is Bearer Token / JWT token usage?

■ Explanation (simple, conceptual)

A **Bearer Token** is a security token used to authenticate API requests.

If you "bear" the token, you are allowed access.

A **JWT (JSON Web Token)** is a specific type of bearer token that contains:

- Header (algorithm)

- Payload (user info, roles, expiry)
- Signature (ensures token integrity)

JWT is commonly used in modern authentication systems (.NET, Node.js, Spring, Angular apps).

■ When/Why is it used?

Use Bearer/JWT tokens when:

- Calling protected APIs
- Maintaining secure sessions between frontend (Angular) and backend (.NET)
- Avoiding sending username/password for every request
- Implementing stateless authentication

Use-case in your Angular + .NET project:

✔ User logs in → backend generates JWT → Angular stores token → sends token in every request

■ Example / Code snippet

Example JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Request using Bearer token in Postman

Headers:

```
Authorization: Bearer eyJhbGc...
```

Angular example:

```
http.get('/api/users', {
  headers: {
    Authorization: `Bearer ${token}`
  }
});
```

■ Short summary line

👉 *Bearer/JWT tokens authenticate secure API requests without sending credentials each time.*

📌 Collections & Scripts

✅ 1) Postman Collections — why useful?

■ Explanation (simple, conceptual)

A Postman Collection is a **group of saved API requests** organized in folders.

Instead of manually retyping URLs or headers every time, collections let you save requests once and reuse them whenever needed.

They act like a "project" where all related APIs are neatly stored.

■ When/Why is it used?

Collections are useful when you need to:

- Test multiple APIs for the same project
- Maintain organized CRUD operations
- Share the whole API set with your team
- Run API sequences (login → fetch → update → delete)
- Perform regression testing
- Automate API tests using Postman Runner or Newman

In real-time projects, developers and QA teams store:

- Auth APIs
- User APIs
- Admin APIs
- File upload APIs

...all under one collection.

■ Example / Use-case

Creating a collection for “Expense Tracker APIs”

```
Expense Tracker Collection
 ├── Auth APIs
 │    ├── POST /login
 │    └── POST /register
 ├── Expense APIs
 │    ├── GET /expenses
 │    ├── POST /expenses
 │    └── DELETE /expenses/{id}
```

■ Short summary line

👉 *Collections help save, organize, share, and automate all your project APIs in one place.*

✅ 2) Environments & Variables — how used?

■ Explanation (simple, conceptual)

Environments in Postman allow you to create **different sets of variables** for different stages (Local, Dev, QA, Prod).

Variables replace hard-coded values like URLs, tokens, user IDs, etc.

Example variables:

- `{{baseUrl}}`

- `{{authToken}}`
- `{{userId}}`

■ When/Why is it used?

Use Environments & Variables when:

- Switching between dev → QA → production servers
- Reusing same API with different test data
- Automatically injecting tokens after login
- Avoiding manual edits in every request

This reduces repeated work and makes API testing cleaner.

■ Example / Code snippet

Create Environment "DEV" with variables:

```
baseUrl : https://dev.myapp.com/api
token   : {{generated-token}}
```

Use in API:

```
GET {{baseUrl}}/users
Headers:
Authorization: Bearer {{token}}
```

■ Short summary line

👉 *Environments store variables like URL/token so you can reuse the same API for Dev, QA, or Prod without changes.*

✅ 3) Pre-request script vs Tests tab

■ Explanation (simple, conceptual)

Postman supports JavaScript-based scripts in two places:

Pre-request Script

- Runs *before* the API request is sent.
- Used to prepare data or set variables.

Tests Tab

- Runs *after* the response is received.
- Used to validate the response or write assertions.

■ When/Why is it used?

Use **Pre-request Script** for:

- Generating timestamps

- Creating dynamic tokens
- Setting environment variables
- Chaining requests (e.g., login first)

Use **Tests Tab** for:

- Validating status code
- Checking response values
- Saving response data
- Automated testing

■ Example / Code snippet

Pre-request script example

```
pm.environment.set("currentTime", new Date().toISOString());
```

Tests Tab example

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.environment.set("userId", pm.response.json().id);
```

■ Short summary line

👉 *Pre-request scripts prepare data before calling the API, while Tests tab validates the API response.*

✅ 4) How do you automate test cases in Postman?

■ Explanation (simple, conceptual)

Postman supports automation using **scripts + collections + runners**.

You can write tests using JavaScript in the Tests tab and run the entire collection automatically through Postman Runner or Newman.

Automation helps verify APIs quickly during regression cycles or CI/CD.

■ When/Why is it used?

Use Postman test automation when:

- You want to run all APIs at once
- You need repeatable regression tests
- You want to verify that backend APIs work after code changes
- Integrating API tests into CI/CD pipeline

This reduces manual testing effort.

■ Example / Code snippet

Test script in Postman

```
pm.test("Validate response structure", function () {
    pm.expect(pm.response.code).to.equal(200);
    pm.expect(pm.response.json()).to.have.property("email");
});
```

Automate using Collection Runner

1. Open Collection → Run
2. Select Environment
3. Set iterations
4. Click **Start Run**

Automation using Newman (CLI)

```
newman run ExpenseTracker.postman_collection.json -e DevEnv.json
```

This can be integrated with:

- GitHub Actions
- Jenkins
- Azure DevOps

■ Short summary line

👉 *Automation is done using Test scripts + Collection Runner + Newman to run multiple APIs with assertions automatically.*

📌 Real-time Scenarios

✅ 1) API returns 500 error — how will you debug?

■ Explanation (simple, conceptual)

A `500 Internal Server Error` means something on the server threw an unhandled exception or failed (runtime error, dependency failure, resource exhaustion, etc.). Debugging is about reproducing the request, locating the failing code/path, reading the exception/stack trace, and fixing the root cause (null refs, DB timeouts, serialization issues, etc.).

■ When/Why is it used

You debug 500s when an endpoint unexpectedly fails in production or staging. Quick diagnosis reduces downtime and prevents user-impacting regressions. Use structured steps so you don't ship a temporary fix that masks the real problem.

■ Example / Code snippet (step-by-step actionable)

1. Reproduce and collect context

- Reproduce with Postman/curl using same headers, body, user, and environment.

```
curl -v -X POST https://api.example.com/orders \
  -H "Authorization: Bearer <token>" \
  -H "Content-Type: application/json" \
  -d '{"amount": 100, "userId": 42}'
```

2. Check server logs & correlation id

- Look at structured logs (App Insights / Serilog). Correlate request ID or timestamp to the error.
- Example Serilog middleware pattern (ASP.NET Core) to ensure meaningful logs:

```
app.UseExceptionHandler(errApp ⇒
{
    errApp.Run(async context ⇒
    {
        var ex = context.Features.Get<IExceptionHandlerFeature>()?.Error;
        var id = Activity.Current?.Id ?? context.TraceIdentifier;
        logger.LogError(ex, "RequestId:{RequestId} Unexpected error", id);
        context.Response.StatusCode = 500;
        await context.Response.WriteAsJsonAsync(new { error = "Internal server error", id });
    });
});
```

3. Read exception + stack trace

- Identify the failing assembly and line (e.g., NullReference in Service.X or SqlTimeout during query).

4. Check dependencies

- Database: slow/blocked queries, missing migrations, deadlocks.
- External services: timeouts, 5xxs, auth failures.
- Serialization issues: circular refs, large payloads.

5. Reproduce locally with same data

- Use same request body and environment variables locally, add breakpoints in VS/VSCode and run debugger.

6. Inspect DB/resource

- Run the SQL query in SSMS; check indexes, long-running transactions. Example:

```
SET STATISTICS IO, TIME ON;
-- suspect query
SELECT * FROM Orders WHERE UserId = 42;
```

7. Short-term mitigation + fix

- Add validation to avoid nulls, increase timeout if justified, add retry/backoff to external calls, or fix query (add index).

- Add unit/integration tests to prevent regression.

8. Post-mortem

- Add monitoring/alerts, write a regression test, and document root cause. If secrets or data caused it, rotate credentials.

■ Short summary line

👉 *Reproduce the request, check structured logs & stack trace, debug locally with same inputs, inspect dependencies (DB/external), fix root cause, and add tests/monitoring to prevent recurrence.*

✅ 2) Input validation testing example

■ Explanation (simple, conceptual)

Input validation testing verifies that the API accepts valid input, rejects invalid input, and returns appropriate errors (400 Bad Request) with useful messages. It covers type checks, required fields, length, ranges, format, SQL injection, and business rules.

■ When/Why is it used

We test validation to prevent bad data reaching business logic/DB, avoid runtime exceptions, guard against security issues, and provide clear feedback to clients (frontend or other services).

■ Example / Code snippet (examples of tests & server-side validation)

Server-side: ASP.NET Core model validation

```
public class CreateUserDto
{
   [Required] public string FirstName { get; set; }
   [Required] [EmailAddress] public string Email { get; set; }
   [Range(18, 100)] public int Age { get; set; }
}

[HttpPost("users")]
public IActionResult CreateUser([FromBody] CreateUserDto dto)
{
   if (!ModelState.IsValid) return BadRequest(ModelState);
   // create user...
   return Created(...);
}
```

Test cases (Postman / unit tests):

- Positive: valid body → expect `201 Created`.

```
{ "firstName": "Mohini", "email": "mohini@test.com", "age": 25 }
```

- Missing required field → expect `400 Bad Request` & message:

```
{ "email": "mohini@test.com", "age": 25 }  // missing FirstName
```

- Invalid format → expect `400` : invalid email

```
{ "firstName":"M", "email":"not-an-email", "age": 25 }
```

- Boundary tests → age = 17 → expect `400` ; age = 100 → ok.
- SQL injection / malicious input → expect sanitized rejection or safe handling

```
{ "firstName": "'; DROP TABLE Users; --", "email": "x@x.com", "age": 25 }
```

Automated unit test example (xUnit)

```
[Fact]
public async Task CreateUser_InvalidEmail_ReturnsBadRequest()
{
    var dto = new CreateUserDto { FirstName="M", Email="bad", Age=20 };
    var resp = await _client.PostAsJsonAsync("/users", dto);
    Assert.Equal(HttpStatusCode.BadRequest, resp.StatusCode);
}
```

■ Short summary line

👉 *Input validation tests ensure APIs accept valid data and reject invalid/malicious input via model validation, boundary checks, format rules, and automated tests.*

✅ 3) Pagination & filter test case examples

■ Explanation (simple, conceptual)

Pagination controls how many records are returned and which slice of results, while filtering refines results by criteria. Testing ensures correctness, consistency, performance, and correct metadata (total counts, next/prev links).

■ When/Why is it used

We test pagination & filters to ensure UI lists behave correctly, APIs scale for large datasets, and clients can reliably page through results without missing or duplicating records.

■ Example / Code snippet (test cases + API patterns)

Common API patterns

- Offset/Limit: `GET /expenses?page=2&pageSize=20`
- Limit/Offset alternative: `GET /expenses?offset=20&limit=20`
- Keyset pagination (cursor): `GET /expenses?cursor=abc123&limit=50`

Test cases

1. **Basic pagination**

- Request `page=1&pageSize=10` → expect 10 items and `totalCount >= 10`.

2. **Last page smaller**
   - If total=23, `page=3&pageSize=10` → expect 3 items.
3. **Invalid values**
   - `page=0` or `pageSize=-1` → expect `400 Bad Request`.
4. **Boundary**
   - `pageSize`=max allowed (e.g., 100) → ensure server caps or rejects oversized sizes.
5. **Filtering combined with pagination**
   - `GET /expenses?status=approved&page=1&pageSize=20` → ensure results all have `status=approved`.
6. **Sorting consistency**
   - Request `sort=createdAt_desc` and ensure order is correct and stable across pages.
7. **No results**
   - Filter that yields no data → expect `200` with empty array and `totalCount=0`.
8. **Performance**
   - Large page size should still respond within SLA; check DB explain plan and indexes.
9. **Cursor/keyset tests**
   - Verify `cursor` returns next set and no duplication/skip across sequential requests.

Sample response structure

```
{
  "items": [{ ... }, { ... }],
  "page": 2,
  "pageSize": 10,
  "totalCount": 85,
  "hasNext": true
}
```

Integration test snippet (pseudo)

```
var resp = await client.GetAsync("/expenses?page=3&pageSize=10");
var dto = await resp.Content.ReadFromJsonAsync<PageResult<ExpenseDto>>();
Assert.Equal(3, dto.Page);
Assert.True(dto.Items.Count <= 10);
```

■ Short summary line

👉 *Test pagination for correct slicing, last-page behavior, invalid parameters, combined filters, sorting stability, and performance under large datasets.*

✅ 4) Testing authentication required APIs

■ Explanation (simple, conceptual)

Testing authenticated endpoints verifies that valid credentials/tokens allow access, invalid/missing/expired tokens are rejected, and role/permission checks enforce access control. Tests cover token generation, propagation, expiry, and scope-based authorization.

■ When/Why is it used

Use these tests to ensure security boundaries are enforced—frontend cannot access protected endpoints without login, and the backend returns correct status codes (401 Unauthorized, 403 Forbidden).

■ Example / Code snippet (Postman flow + code)

1. Get token (login)

- Postman: `POST /auth/login` → save token from response to environment variable

```
// Tests tab after login
pm.environment.set("authToken", pm.response.json().token);
```

2. Use token for protected call

- Add header: `Authorization: Bearer {{authToken}}`
- Expect `200 OK` for valid token.

3. Negative tests

- Missing token → expect `401 Unauthorized`.
- Expired token (simulate by issuing token with short TTL) → expect `401`.
- Token with insufficient role/claim → expect `403 Forbidden`.

4. Sample C# attribute + policy (ASP.NET Core)

```
[Authorize(Policy = "MustBeAdmin")]
[HttpDelete("users/{id}")]
public IActionResult DeleteUser(int id) { ... }

// Configure policy
services.AddAuthorization(options ⇒
    options.AddPolicy("MustBeAdmin", policy ⇒ policy.RequireClaim("role", "Admin")));
```

5. Automated test examples

- **Happy path**

```
var login = await client.PostAsJsonAsync("/auth/login", creds);
var token = await login.Content.ReadFromJsonAsync<TokenResponse>();
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token.A
ccessToken);
var resp = await client.GetAsync("/orders");
Assert.Equal(HttpStatusCode.OK, resp.StatusCode);
```

- **Missing token**

```
var resp2 = await client.GetAsync("/orders");
Assert.Equal(HttpStatusCode.Unauthorized, resp2.StatusCode);
```

- **Role-based**

```
// create token with non-admin role and call admin endpoint → expect Forbidden
```

6. Edge cases

- Replay attacks: ensure nonce/timestamps if required.
- Scope checks: ensure token with right scopes (read/write).
- Token revocation: test that revoked tokens are rejected.

■ Short summary line

👉 *Test auth endpoints by automating token retrieval, verifying success with valid tokens, and asserting 401/403 for missing, expired, or insufficient-scope tokens; also validate role/policy-based access.*

🔥 TESTING INTERVIEW QUESTIONS

📌 Basic Testing Fundamentals

✅ 1) What is unit testing? Why is it needed?

■ Explanation (simple, conceptual)

Unit testing is a testing method where **individual components or functions** of an application are tested in isolation. Each test verifies one small "unit" of logic—like a service method, utility function, or business rule.

In .NET or Angular, this often means testing:

- Service methods
- Business logic
- Validation logic
- Utility/helper functions

Unit tests do **not** test databases, APIs, or UI—only isolated logic.

■ When/Why is it used?

We use unit testing to:

- Catch bugs early (before integration)
- Improve code reliability
- Ensure new changes don't break existing functionality

- Document expected behavior
- Enable safe refactoring
- Speed up development cycle

Teams using CI/CD rely on unit tests to prevent regressions during every push.

■ Example / Code snippet

C# example using xUnit

```
public int Add(int a, int b) ⇒ a + b;

[Fact]
public void Add_ReturnsCorrectSum()
{
    var result = Add(3, 5);
    Assert.Equal(8, result);
}
```

■ Short summary line

👉 *Unit testing checks individual functions in isolation to ensure reliability and prevent early-stage bugs.*

✅ 2) Manual testing vs automation testing

■ Explanation (simple, conceptual)

Manual Testing:

Human testers execute test cases without using scripts. Used for UI flows, usability checks, exploratory testing.

Automation Testing:

Tests are executed using automated tools or scripts (Postman, Selenium, NUnit, Cypress, etc.). Used when tests are repetitive and large in number.

■ When/Why is it used?

Use Manual Testing when:

- UI validation
- User experience testing
- Small, one-time tests
- Exploratory scenarios

Use Automation Testing when:

- Large test suites
- Repetitive test cases
- Regression testing

- API testing with dynamic data
- Performance/load testing

Automation brings speed, accuracy, reusability, and CI/CD integration.

■ Example / Code snippet

Automation example using Postman Tests:

```
pm.test("Status is 200", () => {
    pm.response.to.have.status(200);
});
```

■ Short summary line

👉 *Manual testing relies on human execution; automation runs tests via scripts for speed, consistency & scalability.*

✅ 3) Functional vs Non-functional testing

■ Explanation (simple, conceptual)

Functional Testing checks **what the system does**.

It validates business logic, features, inputs/outputs.

Non-functional Testing checks **how the system behaves**.

It validates performance, security, scalability, reliability, etc.

■ When/Why is it used?

Functional Testing

Used to verify:

- API responses
- Input validation
- Login flow
- CRUD operations
- Business rules

Non-functional Testing

Used to verify:

- Load time
- Stress capacity
- Security vulnerabilities
- Scalability under traffic
- Reliability and uptime

Both combined ensure a complete quality check.

■ Example / Code snippet

Functional test:

```
POST /login → returns 200 + token
```

Non-functional test:

```
Load test API with 1000 requests/sec → ensure API stays under 200ms response time
```

■ Short summary line

👉 *Functional testing checks correctness of features; non-functional testing checks performance, security, and system behavior.*

✅ 4) What is test case? Test scenario?

■ Explanation (simple, conceptual)

Test Case

A detailed set of steps, inputs, expected results written for a specific condition. Highly granular.

Test Scenario

A high-level testing concept that covers an entire user flow or module. Broader and more abstract.

■ When/Why is it used?

Use **Test Scenarios** when:

- Planning overall coverage
- Explaining flows to stakeholders
- High-level testing strategy

Use **Test Cases** when:

- Executing detailed tests
- Validating corner cases
- Ensuring repeatable results

■ Example / Code snippet

Test Scenario:

"Verify user can log in successfully."

Test Cases:

1. Login with valid credentials → success
2. Invalid username → 401 error
3. Missing password → 400 error
4. Locked user → 403 error

■ Short summary line

👉 *A test scenario is high-level; a test case is a detailed step-by-step validation for a scenario.*

✅ 5) What is regression testing?

■ Explanation (simple, conceptual)

Regression testing ensures that **new code changes have not broken existing features**. Even a small modification can affect other modules—regression ensures stability.

It retests the already-tested functionality after:

- Code changes
- Bug fixes
- Refactoring
- New feature additions

■ When/Why is it used?

Use regression testing when:

- Deploying new builds
- Adding new functionalities
- Fixing issues in existing modules
- Performing refactoring or optimization

It ensures:

- No side effects
- Stability before release
- Confidence in deployments

■ Example / Code snippet

If you add a new “Update Profile” feature:

- Re-test login
- Re-test dashboard
- Re-test user list
- Re-test profile retrieval

Automation helps run regression quickly:

```
newman run MyCollection.json
```

■ Short summary line

👉 *Regression testing rechecks old features to ensure new changes haven't broken existing functionality.*

📌 API Testing (Back-end heavy)

✅ 1) Validation of success & error responses

■ Explanation (simple, conceptual)

Validating success and error responses ensures an API behaves correctly under both **normal** and **unusual** conditions.

A good API should return the right **status codes**, **response body**, and **error messages** consistently.

■ When/Why is it used?

We validate responses to:

- Ensure API follows REST standards
- Provide meaningful feedback to frontend/clients
- Detect broken business logic early
- Verify error-handling paths (null inputs, invalid params, server failure)

Both success and failure paths must be tested to ensure reliability.

■ Example / Code snippet

Success response validation (200 OK / 201 Created)

```
POST /users
{
  "name": "Mohini",
  "email": "m@test.com"
}

Expect:
Status: 201 Created
Body:
{
  "id": 101,
  "name": "Mohini",
  "email": "m@test.com"
}
```

Error response validation

```
POST /users
{
  "email": "invalid-email"
}

Expect:
Status: 400 Bad Request
Body:
```

```
{
  "error": "Invalid email format"
}
```

Server error

```
Status: 500 Internal Server Error
```

■ Short summary line

👉 *Success/error validation ensures APIs return correct codes, correct data, and clear error messages for all situations.*

✅ 2) Negative test cases for POST API

■ Explanation (simple, conceptual)

Negative testing verifies how the API reacts to **invalid, incomplete, or unexpected input**.

It ensures the API gracefully rejects bad data rather than crashing or returning incorrect responses.

■ When/Why is it used?

Used to:

- Improve API stability
- Prevent invalid data entering DB
- Ensure validations are enforced
- Check security corner cases
- Avoid runtime exceptions (500 errors)

■ Example / Code snippet

For a POST `/register` API:

Negative test cases

1. **Missing required field**

```
{ "email": "test@mail.com" }
→ Expect 400: “Name is required”
```

1. **Invalid email format**

```
{ "name": "Mohini", "email": "invalid" }
→ Expect 400: “Invalid email format”
```

1. **Field length too long**

```
{ "name": "aaaaaaaaaaaaaaaaa..." }
```

→ Expect 400: “Name exceeds max length”

1. **Invalid data type**

```
{ "age": "twenty" }
→ Expect 400: “Age must be a number”
```

1. **SQL injection attempt**

```
{ "name": "'; DROP TABLE Users;--" }
→ Expect 400 / sanitized input
```

1. **Duplicate value**

```
{ "email": "existing@mail.com" }
→ Expect 409 Conflict
```

1. **Unauthorized POST**

- No token → `401 Unauthorized`
- Wrong role → `403 Forbidden`

■ Short summary line

👉 *Negative tests check how APIs handle invalid, missing, or malicious inputs to ensure robust validation.*

✅ 3) Boundary testing example

■ Explanation (simple, conceptual)

Boundary testing checks system behavior at **extreme limits** — minimum, maximum, and just outside valid ranges.

Most bugs occur exactly at the edges (0, 1, max value), not in the middle.

■ When/Why is it used?

- Validates constraints (length, range, limits)
- Prevents memory, performance, or validation issues
- Ensures API behaves correctly under edge conditions
- Often used with numeric, date, pagination, age, price, or string fields

■ Example / Code snippet

For a field: **Age must be between 18 and 60**

Valid boundaries

- Age = 18
- Age = 60

Invalid boundaries

- Age = 17 → expect 400
- Age = 61 → expect 400

Another example — pageSize between 1 and 100:

- `pageSize=1` → valid
- `pageSize=100` → valid
- `pageSize=0` → invalid
- `pageSize=101` → invalid

For string length (max 50 chars):

- 50 chars → ok
- 51 chars → reject

■ Short summary line

👉 *Boundary testing checks API behavior at min/max limits to ensure validations work correctly.*

✅ 4) Idempotency — why important for PUT/DELETE?

■ Explanation (simple, conceptual)

Idempotency means calling the same API multiple times results in the **same final state**.

No duplicate side effects should occur.

REST requires:

- **PUT** → idempotent
- **DELETE** → idempotent
- **GET** → idempotent
- **POST** → not idempotent (creates new resources)

■ When/Why is it used?

Idempotency is important because:

- Network issues may trigger **retries**
- Clients may resend requests unintentionally
- Prevents duplicate updates or accidental data deletion
- Ensures safe, predictable API behavior in distributed systems

■ Example / Code snippet

PUT example (should be idempotent):

```
1st call:
PUT /users/100
{ "name": "Mohini" }

2nd call:
PUT /users/100
{ "name": "Mohini" }

Final result → same user, no duplicates, no side effects.
```

DELETE example (should be idempotent):

```
DELETE /users/100
DELETE /users/100

Both calls → user 100 removed, second call returns 204 No Content.
```

POST (NOT idempotent):

```
POST /payments
→ Creates new payment
→ Calling twice creates 2 transactions
```

In Microservices / Banking / Payments systems — idempotency is critical for reliability.

■ Short summary line

👉 *Idempotency ensures PUT/DELETE produce the same result even if called multiple times—critical for safe, retryable API behavior.*

📌 Tools & Framework Concepts

✅ 1) NUnit / xUnit basics (if applicable)

■ Explanation (simple, conceptual)

NUnit and xUnit are **unit testing frameworks** for .NET applications.

They provide annotations, assertions, and utilities to write structured, repeatable automated tests.

- **NUnit** → older, attribute-heavy style (`[Test]` , `[SetUp]` , `[TearDown]`)
- **xUnit** → modern, lightweight, widely used; replaces setup/teardown with constructors & `IDisposable`

Both help verify code correctness without running the full application.

■ When/Why is it used?

We use testing frameworks when we want to:

- Validate business logic

- Prevent bugs during refactoring
- Run automated tests in CI/CD pipelines
- Ensure reliability of services, validation rules, utilities
- Mock dependencies cleanly (with Moq/AutoFixture)

In real .NET projects:

- Backend services are tested using xUnit (most popular today).
- Tests run automatically during `dotnet test` or GitHub Actions pipeline.

■ Example / Code snippet

xUnit basic example

```
public class Calculator
{
    public int Add(int a, int b) => a + b;
}

public class CalculatorTests
{
    [Fact]
    public void Add_ShouldReturnCorrectSum()
    {
        var calc = new Calculator();
        var result = calc.Add(2, 3);
        Assert.Equal(5, result);
    }
}
```

NUnit example

```
[Test]
public void Add_ShouldReturnCorrectSum()
{
    var calc = new Calculator();
    Assert.AreEqual(5, calc.Add(2, 3));
}
```

■ Short summary line

👉 *NUnit and xUnit are .NET frameworks used to write automated unit tests with assertions for reliable backend logic.*

✅ 2) Mocking & Stubbing concepts

■ Explanation (simple, conceptual)

Mocking substitutes real dependencies (like DB, API calls, email services) with **fake objects** that simulate behavior.

Stubbing provides predefined responses for methods.

Both isolate the unit being tested so only your logic is tested—NOT the external dependencies.

■ When/Why is it used?

We use mocks/stubs when:

- A service depends on external systems
- Calling real APIs/DB is not feasible in unit tests
- We want predictable, controlled responses
- We want to avoid slow or costly operations
- Testing error paths without breaking production APIs

In .NET, **Moq** is the most common mocking library.

■ Example / Code snippet

Mocking example (using Moq)

```
public interface IUserRepo
{
    User GetUser(int id);
}

[Fact]
public void GetUser_ReturnsMockedUser()
{
    var mockRepo = new Mock<IUserRepo>();
    mockRepo.Setup(r ⇒ r.GetUser(1)).Returns(new User { Id = 1, Name = "Mohini" });

    var result = mockRepo.Object.GetUser(1);
    Assert.Equal("Mohini", result.Name);
}
```

Stubbing example

```
mockRepo.Setup(r ⇒ r.GetUser(It.IsAny<int>()))
        .Returns(new User { Id = 99, Name = "StubUser" });
```

■ Short summary line

👉 *Mocking simulates behavior of real dependencies; stubbing returns predefined outputs to isolate unit logic.*

✅ 3) What is code coverage?

■ Explanation (simple, conceptual)

Code coverage measures **how much of your source code is executed** by automated tests.

It's represented as a percentage of lines, branches, or statements covered during test runs.

Higher coverage means more of your logic is tested.

Coverage types:

- **Line coverage** → lines executed
- **Branch coverage** → all if/else paths
- **Method coverage** → functions executed

■ When/Why is it used?

We use code coverage to:

- Ensure critical paths are tested
- Improve confidence before deploying
- Identify untested or risky parts of code
- Maintain quality in CI/CD pipelines
- Avoid regressions after refactoring

But coverage does **not** guarantee correctness—it only shows what was executed, not whether logic is correct.

Ideal range for real projects:

- **70–85%** is realistic and maintainable
- 100% is rare and usually unnecessary

■ Example / Code snippet

Running coverage in .NET:

```
dotnet test /p:CollectCoverage=true /p:CoverletOutputFormat=lcov
```

Sample output:

```
Total Lines: 120
Covered Lines: 90
Coverage: 75%
```

In CI/CD (GitHub Actions):

```
- name: Run tests with coverage
  run: dotnet test /p:CollectCoverage=true
```

■ Short summary line

👉 *Code coverage shows how much of your code is executed by tests, helping measure test completeness and risk.*

📌 Real Scenario Questions

✅ 1) How do you test login module?

■ Explanation (simple, conceptual)

Testing a login module covers **functional**, **security**, and **integration** aspects: verify correct credentials produce a token/session, invalid credentials are rejected with appropriate status codes, token lifecycle (expiry/refresh) works, and protected endpoints reject missing/invalid tokens. Tests include unit tests for authentication logic, integration tests for the full API flow, Postman/manual checks, and end-to-end tests from the Angular UI.

■ When/Why is it used?

We test login to ensure:

- Users can authenticate reliably (happy path)
- Unauthenticated/unauthorized access is prevented (401/403)
- Tokens are correctly issued, stored, refreshed, and revoked
- Security aspects (rate limits, lockouts, brute-force protection) are enforced

  This prevents major security and UX regressions, especially in banking/automation products.

■ Example / Code snippet

1) Unit test (C# — xUnit) for auth logic

```
[Fact]
public void Authenticate_ValidCredentials_ReturnsJwt()
{
   var mockUserRepo = new Mock<IUserRepository>();
   mockUserRepo.Setup(r ⇒ r.ValidateCredentials("user","pass"))
            .Returns(new User{ Id=1, UserName="user" });

   var auth = new AuthService(mockUserRepo.Object, jwtOptions);
   var token = auth.Authenticate("user","pass");

   Assert.False(string.IsNullOrEmpty(token));
}
```

2) Integration test (calls actual controller)

```
var resp = await client.PostAsJsonAsync("/api/auth/login", new { username="m", password="p"
});
Assert.Equal(HttpStatusCode.OK, resp.StatusCode);
var body = await resp.Content.ReadFromJsonAsync<LoginResponse>();
Assert.NotNull(body.AccessToken);
```

3) Postman tests (automated)

- Request: `POST /api/auth/login` with valid creds

- Tests tab:

```
pm.test("Login returns 200", () => pm.response.to.have.status(200));
pm.test("Token present", () => pm.expect(pm.response.json().accessToken).to.be.a('string'));
pm.environment.set("authToken", pm.response.json().accessToken);
```

- Follow-up protected call:
  - Add header `Authorization: Bearer {{authToken}}` and assert 200.

4) Frontend (Angular) E2E / Cypress

- Automate login flow: fill form, submit, assert redirect and token stored in `localStorage` or cookie.

```
cy.get('#username').type('mohini');
cy.get('#password').type('pass123');
cy.get('#loginBtn').click();
cy.url().should('include', '/dashboard');
cy.window().its('localStorage.authToken').should('exist');
```

5) Security & edge tests

- Wrong password → expect `401`
- Missing fields → expect `400` with validation messages
- Expired token → protected call returns `401`
- Brute-force: confirm rate-limit or account lock after N attempts (integration test or manual).

■ Short summary line

👉 *Test login by unit-testing auth logic, integration tests for token issuance, Postman + automated collection for flow, and E2E tests for the UI while validating security and edge cases.*

✅ 2) How do you test a file upload API?

■ Explanation (simple, conceptual)

File upload testing verifies correct handling of files (type, size, streaming), security (virus scanning, path traversal), performance (multipart streaming, large files), and correct responses (201/400/413). Tests include unit tests for validation logic, integration tests against the real API, Postman/manual uploads, and automated stress tests for large files.

■ When/Why is it used?

We test file upload APIs to:

- Prevent malicious files or oversized payloads from breaking the server
- Ensure proper storage and metadata persistence (DB entry, path, checksum)
- Confirm streaming works and memory is not exhausted
- Validate user feedback (progress, success/failure codes)

This is critical for user-facing apps and automation workflows that accept attachments.

■ Example / Code snippet

1) ASP.NET Core controller (robust pattern)

```
[HttpPost("upload")]
public async Task<IActionResult> Upload(IFormFile file)
{
    if (file == null) return BadRequest("No file provided");
    if (file.Length == 0) return BadRequest("Empty file");
    if (file.Length > 10 * 1024 * 1024) return StatusCode(413, "File too large"); // 10MB limit

    var ext = Path.GetExtension(file.FileName).ToLowerInvariant();
    var allowed = new[] { ".png", ".jpg", ".pdf" };
    if (!allowed.Contains(ext)) return BadRequest("Invalid file type");

    // stream to disk or cloud storage
    var filePath = Path.Combine(_storagePath, Guid.NewGuid() + ext);
    await using var stream = File.Create(filePath);
    await file.CopyToAsync(stream);

    // persist metadata in DB
    // return 201 with metadata
    return CreatedAtAction(nameof(GetFile), new { id = id }, new { id, fileName = file.FileName });
}
```

2) Postman test cases

- Valid small file (png) → expect `201 Created` and returned metadata
- Large file > 10MB → expect `413 Payload Too Large`
- Unsupported extension (.exe) → expect `400 Bad Request`
- Missing file field → expect `400 Bad Request`
- Malicious filename with traversal `../../` → ensure server sanitizes and stores safe path

3) Automated integration test (C#)

```
var content = new MultipartFormDataContent();
content.Add(new StreamContent(File.OpenRead("test.png")), "file", "test.png");
var resp = await client.PostAsync("/api/files/upload", content);
Assert.Equal(HttpStatusCode.Created, resp.StatusCode);
```

4) Performance & security checks

- Use load tool (k6/JMeter/Newman scripts) to upload multiple files concurrently to ensure streaming and CPU/memory limits hold.
- Integrate antivirus/malware scan step (e.g., ClamAV) as part of upload pipeline.
- Validate checksum/ETag stored for integrity.

■ Short summary line

👉 *Test file uploads by validating size/type/streaming behavior with unit/integration tests, Postman/manual checks, performance stress tests, and security scans to ensure safe, scalable uploads.*

✅ 3) One bug you found during development & how you fixed it?

■ Explanation (simple, conceptual)

Describe a real debugging story focusing on reproduction, root cause analysis, a targeted fix, tests added, and outcome (metrics). Keep it concise and outcome-focused.

■ When/Why is it used?

Interviewers look for problem-solving, debugging process, collaboration, and ownership—show you can find root causes, communicate, and prevent recurrence.

■ Example / Story (structured, speakable)

Situation: While working on the Intelligent Automation transaction service at Bank of America, a production intermittent failure caused an API to return `500 Internal Server Error` for a subset of transactions. This increased downtime and impacted automation jobs.

Reproduce & Investigate:

- Reproduced locally using the same request payload captured from logs.
- Correlated logs using request correlation id (middleware added earlier) and found a `NullReferenceException` in the transaction enrichment service.
- The failure occurred only when optional metadata (`transaction.metadata`) was missing from certain legacy clients.

Root Cause:

- The enrichment service assumed `metadata` was always present and directly accessed `metadata.currency` without null checks. Later refactoring introduced the assumption but did not add backward compatibility for legacy callers.

Fix Implemented:

1. **Immediate hotfix (safe and small):** added defensive checks and defaulting before the enrichment step to stop the 500 errors.

```
var currency = transaction.Metadata?.Currency ?? "USD"; // default
if (transaction.Metadata == null) transaction.Metadata = new Metadata();
```

1. **Permanent fix:** updated DTO validation and introduced model binding checks so requests missing critical fields returned `400` with clear messages instead of 500.

```
if(transaction.Metadata == null)
    return BadRequest(new { error = "Missing transaction.metadata" });
```

1. **Tests & Monitoring:**

- Added unit tests for enrichment with null metadata.
- Added integration test replicating legacy payloads.
- Added an App Insight alert for repeated 500 rates and a metric for missing metadata occurrences.

Outcome:

- Production errors stopped for that flow immediately after hotfix.
- Downtime reduced; monitoring showed a 20% reduction in overall incidents for that service (matching the improvement metric on my resume).
- Longer-term, the fix prevented similar regressions and improved API contract clarity with consumers.

■ Short summary line

👉 *I diagnosed an intermittent 500 by reproducing logs locally, fixed a null-reference with defensive coding + validation, added tests and alerts — cutting recurrence and improving reliability.*

🔥 DEBUGGING INTERVIEW QUESTIONS

📌 Basics

✅ 1) What is debugging?

■ Explanation (simple, conceptual)

Debugging is the process of **identifying, analyzing, and fixing defects** in an application.

It involves observing the program's step-by-step execution, checking variable values, understanding control flow, and isolating the line of code causing the issue.

Debugging ensures the system behaves as expected in both normal and edge conditions.

■ When/Why is it used?

Debugging is essential when:

- The application throws runtime errors (NullReferenceException, HTTP 500, UI crash).
- Logic doesn't produce expected output.
- API responses are incorrect or incomplete.
- Database queries return wrong results.
- UI is not updating due to state or binding issues.
- You need to verify behavior after refactoring or integrating third-party APIs.

It improves reliability, reduces production issues, and accelerates development.

■ Example / Code snippet

Example of catching a null reference:

```
var result = service.GetUser(id); // result is null
var name = result.Name;          // debug reveals exception
```

Debugger helps you inspect `result` before accessing it.

■ Short summary line

👉 *Debugging is the process of isolating and fixing errors by analyzing code execution step-by-step.*

✅ 2) Breakpoints — how do you use them?

■ Explanation (simple, conceptual)

A breakpoint is a marker placed in code that **pauses execution** at a specific line so you can inspect program state — variables, conditions, memory, call stack, and flow.

It is the most fundamental debugging tool.

■ When/Why is it used?

Breakpoints are used when you want to:

- Inspect variable values at a certain line
- Verify logic execution order
- Stop before an exception occurs
- Check if your function is being called
- Validate conditional logic
- Debug API logic (controllers/services) or Angular components

■ Example / Code snippet

In Visual Studio:

- Click left gutter next to a line → breakpoint appears
- Run **F5 (Start Debugging)**
- Debugger pauses at the breakpoint
- Inspect variables using hover or Watch window

Conditional breakpoint example

Stops only when the condition is met:

```
id == 5
```

■ Short summary line

👉 *Breakpoints pause execution at specific lines to let you inspect variables and logic flow in detail.*

✅ 3) Step Over vs Step Into vs Step Out

■ Explanation (simple, conceptual)

These are debugger navigation controls that help you move through code execution.

Step Into (F11)

- Goes *inside* the method being called.
- Best for investigating detailed logic.

Step Over (F10)

- Executes the current line **without entering** into called methods.
- Best when the method is already tested or not relevant to the bug.

Step Out (Shift + F11)

- Finishes the current method and returns to the caller.
- Useful when you stepped into a method by mistake or logic is correct.

■ When/Why is it used?

Use **Step Into** when:

- Debugging nested method calls
- You want to analyze logic inside a function
- Investigating exceptions thrown by internal methods

Use **Step Over** when:

- You want to skip known/working code
- The method is irrelevant to the bug
- Faster debugging without diving deep

Use **Step Out** when:

- You're done checking inside a method
- You want to return to higher-level logic
- You want to avoid repetitive stepping inside unimportant lines

■ Example / Code snippet

```
var user = service.GetUser(id);  // F11 → goes inside GetUser
var validated = Validate(user);  // F10 → skip method internals
return validated;
```

■ Short summary line

👉 *Step Into goes inside a method, Step Over skips it, and Step Out exits the current method back to the caller.*

✅ 4) Watch window, Call stack usage

■ Explanation (simple, conceptual)

Watch Window:

A debugging tool where you can **add variables or expressions** to monitor their values while stepping through code.

It shows real-time changes and helps track complex logic.

Call Stack:

Shows the **sequence of function calls** that led to the current line.

Helps identify the execution path and where errors originated.

■ When/Why is it used?

Use **Watch Window** when:

- Tracking value changes for variables
- Monitoring expressions or conditions
- Debugging loops, conditions, state changes
- Investigating unexpected calculations

  Example: watching `totalAmount`, `user.Name`, `response.StatusCode`.

Use **Call Stack** when:

- Debugging deep or nested function calls
- Finding which controller/service triggered a bug
- Understanding recursion or async flow
- Locating the exact origin of exceptions

  Especially useful in .NET when exceptions bubble up.

■ Example / Code snippet

Watch Window example

Add:

```
user
user.Address.City
totalAmount * 0.18
```

Call Stack example (Visual Studio view):

```
Controller → Service → Repository → DBContext → Exception
```

■ Short summary line

👉 *Watch Window monitors variable values live, while Call Stack shows the full execution path to help trace the origin of bugs.*

📌 Backend Debugging

✅ 1) Debugging NullReferenceException

■ Explanation (simple, conceptual)

A `NullReferenceException` occurs when code attempts to access a member (property/method/indexer) on a variable that is `null`. It's a symptomatic error showing that an expected object instance was not created or returned. Root causes include missing initialization, failed dependency injection, unexpected `null` from external services, or race conditions in async code.

■ When/Why is it used

You debug this when an API, service, or UI throws `NullReferenceException` (often a 500 on backend). Fixing it prevents crashes and ensures robust null-safety—especially important in banking/automation systems where missing data must be handled gracefully.

■ Example / Code snippet (step-by-step debugging approach)

Reproduce & capture:

- Reproduce the failing request in Postman with same payload and headers.
- Note stack trace and request correlation id from logs.

Inspect stack trace:

- Stack trace shows exact file and line. Example:

```
System.NullReferenceException: Object reference not set to an instance of an object.
   at MyApp.Services.TransactionService.Enrich(Transaction tx) in TransactionService.cs:line 87
```

Use debugger & breakpoints:

- Place a breakpoint on the reported line and run locally with identical input. Inspect variables (`tx`, `tx.Metadata`, etc.).

Common fixes:

1. **Null checks / defensive coding**

```
if (transaction?.Metadata == null) {
    // default or return BadRequest
    transaction.Metadata = new Metadata();
}
var currency = transaction.Metadata.Currency ?? "USD";
```

1. **Validate inputs early (model binding)**

```
if (!ModelState.IsValid) return BadRequest(ModelState);
```

1. **Fix DI configuration or initialization**

- Ensure services registered in `Startup.cs` / `Program.cs`:

```
services.AddScoped<ITransactionService, TransactionService>();
```

1. **Handle async race conditions**

- Await correctly; avoid accessing results before completion.

Add tests & logging:

- Add unit tests for null payloads and add structured logs when critical objects are null:

```
_logger.LogWarning("Transaction missing metadata. RequestId:{ReqId}", reqId);
```

■ Short summary line

👉 *NullReferenceException indicates a missing object — reproduce via logs, breakpoint the failing line, add null checks, validate inputs, fix DI/init issues, and add tests to prevent recurrence.*

✅ 2) Common HTTP errors — 400, 401, 403, 404, 500

■ Explanation (simple, conceptual)

These are standard status codes used by APIs to communicate result types:

- **400 Bad Request** — client sent invalid or malformed data (validation failure).
- **401 Unauthorized** — authentication required or token missing/invalid.
- **403 Forbidden** — authenticated but not authorized (insufficient permissions).
- **404 Not Found** — resource or endpoint does not exist.
- **500 Internal Server Error** — unexpected server-side error or unhandled exception.

■ When/Why is it used?

Using correct status codes helps clients (Angular apps, other services) respond appropriately: show validation messages, redirect to login, show permission errors, fallback logic for missing data, and trigger alerts for server issues. Clear codes improve debuggability and UX.

■ Example / Code snippet

Validation (400)

```
if (!ModelState.IsValid) return BadRequest(ModelState);
```

Authentication (401)

```
[Authorize]
public IActionResult Get() => Ok();
```

If token missing → framework returns 401.

Authorization (403)

```
[Authorize(Policy = "AdminOnly")]
public IActionResult Delete() => Forbid();
```

Not found (404)

```
var user = repo.Get(id);
if (user == null) return NotFound();
```

Server error (500) — global handling

```
app.UseExceptionHandler("/error"); // centralize logging & friendly message
```

Interview speakable mapping:

- 400 → fix request payload/validation
- 401 → ask user to login / refresh token
- 403 → adjust roles/claims or request elevated access
- 404 → verify endpoint/ID and client routing
- 500 → debug server logs, stack trace, dependencies

■ Short summary line

👉 *400 = bad input, 401 = unauthenticated, 403 = forbidden, 404 = missing resource, 500 = server error — each guides specific fixes and client behavior.*

✅ 3) How do you debug API failure?

■ Explanation (simple, conceptual)

Debugging an API failure is a structured investigation: reproduce the failing request, gather logs/trace, inspect stack traces and telemetry, validate environment/config, isolate dependencies (DB, caches, external services), reproduce locally, fix root cause, and add tests/monitoring to prevent recurrence.

■ When/Why is it used?

Used whenever endpoints return unexpected errors (5xx), wrong data, timeouts, or inconsistent behavior across environments. A repeatable, systematic approach reduces mean time to resolution for production incidents.

■ Example / Step-by-step actionable checklist

1. **Reproduce the issue**
   - Use Postman or curl with exact headers, body, and user context.
2. **Collect request context**
   - Use correlation/request-id from client logs or response headers.
3. **Examine server logs & APM**

- Check structured logs (Serilog/App Insights) and traces to find exception and failing stack.

4. **Check health & dependencies**
   - DB connectivity, connection pool exhaustion, cache errors, auth provider availability.
   - Example DB check:

     ```
     SELECT top 5 * FROM sys.dm_exec_requests WHERE blocking_session_id <> 0;
     ```

5. **Reproduce locally & debug**
   - Load same payload locally and step through with breakpoints. Validate configuration (connection strings, feature flags).
6. **Isolate causes**
   - Temporarily mock external services to see if failure persists. Use `git bisect` if regressions suspected.
7. **Fix & test**
   - Apply a minimal safe fix (null-check, timeout adjustments, retry/backoff), add unit/integration tests and QA validation.
8. **Deploy & monitor**
   - Deploy to staging then production; monitor metrics (error rate, latency) and alerts.
9. **Post-mortem**
   - Document root cause, preventive measures (input validation, circuit breaker, retries, alerts).

Tools & commands you'll mention in interview

- Postman/curl for reproducing requests.
- Application Insights / ELK for logs.
- `dotnet run` + Visual Studio debugger for local reproduction.
- DB tools (SSMS) and `EXPLAIN` / `SET STATISTICS` queries.
- Health endpoints and metrics dashboards.

■ Short summary line

👉 *Debug API failures by reproducing the request, checking logs/traces, isolating dependencies, debugging locally, applying a tested fix, and adding monitoring to prevent recurrence.*

✅ 4) How do you debug SQL performance issues?

■ Explanation (simple, conceptual)

SQL performance issues occur due to slow queries, missing indexes, inefficient joins, blocking/locks, parameter sniffing, or non-optimal schema/access patterns. Debugging requires measuring query cost, examining execution plans, checking indexes/stats, monitoring blocking, and profiling to determine root cause.

■ When/Why is it used?

Use this when APIs are slow, timeouts occur, or high CPU/IO is observed on the DB server. Fixing SQL performance directly improves response times and reduces backend errors—critical for transaction-heavy automation systems.

■ Example / Step-by-step actionable checklist

1. **Identify slow queries**
   - Use DB monitoring (SQL Server Profiler, Query Store, `sys.dm_exec_query_stats`) to find high-duration or high-frequency queries.

```
SELECT top 20
  qs.total_elapsed_time/qs.execution_count AS avg_time_ms,
  qt.text
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
ORDER BY avg_time_ms DESC;
```

2. **Check execution plan**
   - Get the actual execution plan (SSMS). Look for table scans, missing index warnings, expensive sorts, or hash joins on large datasets.
3. **Check indexes and statistics**
   - Ensure proper indexes exist for WHERE/JOIN columns. Update statistics if stale:

```
UPDATE STATISTICS dbo.Orders;
```

4. **Parameter sniffing & parameterization**
   - Test with representative parameter values. Use `OPTION (RECOMPILE)` or parameter hints carefully where necessary.
5. **Look for blocking / deadlocks**
   - Use `sys.dm_tran_locks` or Query Store to find locks causing long waits:

```
SELECT * FROM sys.dm_tran_locks WHERE resource_database_id = DB_ID();
```

6. **Optimize query & schema**
   - Replace `SELECT *` with required columns, avoid subqueries when join is better, add covering indexes, or denormalize where read performance is critical.
7. **Pagination & large result sets**
   - Use keyset pagination for large tables instead of OFFSET/LIMIT. Apply LIMIT to reduce data transferred.
8. **Test indexes impact**
   - Create index in staging and measure before and after using execution plans and `SET STATISTICS IO, TIME ON`.
9. **Caching & batching**

- Cache frequent reads (Redis), and batch writes to reduce DB load.

10. **Monitor resource utilization**
    - Check CPU, memory, disk IO on DB server—sometimes hardware limits cause slow queries.

Example fix

Problem: Query doing table scan on Orders by CustomerId.

Fix: Add index and rewrite:

```
CREATE NONCLUSTERED INDEX IX_Orders_CustomerId_CreatedAt ON Orders(CustomerId, Crea
tedAt);
```

Measure improvement via `Actual Execution Plan` and `avg_time_ms`.

■ Short summary line

👉 *Debug SQL performance by identifying slow queries, analyzing execution plans, adding/adjusting indexes, checking blocking/stats, and validating fixes in staging before production.*

📌 Frontend Debugging

✅ 1) Chrome DevTools — Console, Network, Elements panels

■ Explanation (simple, conceptual)

Chrome DevTools is the browser's built-in toolbox for debugging frontend issues.

- **Console**: shows runtime logs, errors, warnings, and lets you run JavaScript interactively.
- **Network**: shows all HTTP requests, status codes, timing, request/response headers and bodies. Vital for API debugging.
- **Elements**: displays DOM and CSS; lets you inspect, edit, and test style/layout changes live.

■ When/Why is it used?

Use DevTools to:

- See JS exceptions and stack traces (Console).
- Verify API calls, response payloads, CORS/preflight, latency and caching (Network).
- Inspect broken layouts, missing classes, or binding issues (Elements).
- Reproduce front-end bugs quickly without rebuilding the app.

For a full-stack interview role, you'll often reproduce an API-vs-UI mismatch by checking Network and Console first, then inspecting Elements for DOM/binding problems.

■ Example / Commands / Quick workflow

1. Open DevTools: `F12` or `Ctrl+Shift+I`.
2. **Console**:

- Look for red errors.
- Evaluate expressions: `ng.getComponent($0)` (with Angular DevTools) or simple `document.querySelector(...)`.

3. **Network**:
   - Filter by XHR / Fetch to see API calls.
   - Click a request → inspect Headers / Preview / Response / Timing.
   - Replay a failed request with "Copy as fetch/cURL" to test in Postman.
4. **Elements**:
   - Inspect a failing component DOM: right-click → Inspect.
   - Edit styles live to check fixes.
5. **Useful extras**:
   - Use Sources panel to set breakpoints in compiled JS (or map to TS with source maps).
   - Use Performance to record and identify rendering or scripting bottlenecks.

Example quick-check: UI not showing data:

- Console: any JS error?
- Network: did the GET `/api/items` return 200 with expected JSON?
- Elements: is the DOM node present but empty or missing binding?

■ Short summary line

👉 *Console finds runtime errors, Network inspects API traffic, and Elements inspects DOM/CSS — together they quickly locate frontend issues.*

✅ 2) CORS error — how to solve?

■ Explanation (simple, conceptual)

CORS (Cross-Origin Resource Sharing) is a browser security mechanism that blocks web pages from making requests to a different origin unless the server explicitly allows it. A CORS error happens when the server does not include the correct `Access-Control-*` headers in response or preflight (OPTIONS) fails.

■ When/Why is it used?

Browsers enforce CORS for security (prevent cross-site attacks). You handle CORS when your Angular SPA (served from e.g., `http://localhost:4200`) calls a backend API at a different origin (e.g., `https://api.mybank.com`).

■ Example / Fixes (Angular dev & .NET production-ready)

Server-side fix (ASP.NET Core) — recommended

```
// Program.cs / Startup.cs
builder.Services.AddCors(options =>
{
```

```
    options.AddPolicy("AllowAngularDev", policy =>
    {
        policy.WithOrigins("http://localhost:4200")  // explicit origin
              .AllowAnyHeader()
              .AllowAnyMethod()
              .AllowCredentials(); // only if you need cookies/auth
    });
});

app.UseCors("AllowAngularDev");
```

- Ensure OPTIONS preflight is allowed and returns 200 with the CORS headers.
- **Do not** use `AllowAnyOrigin()` with `AllowCredentials()` — browsers will reject it; prefer explicit origins.

Dev-time workaround (Angular CLI proxy) — avoids CORS by proxying

`proxy.conf.json`

```
{
  "/api": {
    "target": "https://localhost:5001",
    "secure": false,
    "changeOrigin": true
  }
}
```

Run:

```
ng serve --proxy-config proxy.conf.json
```

This makes the browser think the API is same-origin.

If using credentials (cookies/JWT in cookies):

- Server must send `Access-Control-Allow-Credentials: true` and origin must be explicit (not).
- Client must set `withCredentials: true` on fetch/XHR.

■ Short summary line

👉 *CORS errors are resolved by configuring server-side `Access-Control-*` headers (explicit origins) or using an Angular dev proxy; prefer server fixes for production and avoid `*` with credentials.*

✅ 3) UI not updating — state debugging steps

■ Explanation (simple, conceptual)

When UI doesn't update, the root cause is usually: data not received, binding not wired, change detection not triggered, or the component state mutated incorrectly (OnPush issues). Debugging is about verifying data flow from network → component → template and ensuring Angular detects changes.

■ When/Why is it used?

Use these steps whenever the UI shows stale data after API success, buttons not reflecting state changes, or lists not updating after CRUD operations—common in Angular apps integrated with REST APIs.

■ Example / Debugging checklist & commands

1. **Check Console & Network**
   - Console for errors; Network to confirm API returned expected payload.
2. **Confirm data arrives in component**
   - Add a `console.log(data)` in subscription or use debugger breakpoints in `ngOnInit` / service subscription.

```
this.userService.getUser().subscribe(u => {
  console.log('user', u); // verify data
  this.user = u;
});
```

3. **Check template bindings**
   - Confirm template uses correct property: `{{ user.name }}` vs `{{ userName }}`.
4. **Inspect OnPush / immutability**
   - If component uses `ChangeDetectionStrategy.OnPush`, ensure you **replace** reference or call `markForCheck()` when mutating:

```
// preferred: replace array reference
this.items = [...this.items, newItem];

// or
this.cd.markForCheck();
```

5. **Verify change detection triggers**
   - If update happens inside non-Angular callback (e.g., WebSocket, setTimeout without zone), use `NgZone.run()` or `ChangeDetectorRef.detectChanges()`:

```
this.zone.run(() => { this.value = newVal; });
// or
this.cd.detectChanges();
```

6. **Check async pipe vs manual subscription**
   - Using `| async` automatically triggers change detection; manual subscription must assign to component property.
7. **Use Chrome DevTools Elements**
   - Inspect DOM to see if nodes exist but invisible (CSS) or missing content.
8. **Unit / E2E tests**

- Add a small test to reproduce expected DOM after action if intermittent.

■ Short summary line

👉 *When UI won't update, confirm API returned data, verify component receives it, check template bindings and change detection (OnPush/immutability), and use* `markForCheck` / `detectChanges` *or* `NgZone.run()` *as needed.*

✅ 4) Angular change detection debugging (if stuck)

■ Explanation (simple, conceptual)

Angular change detection compares component state to the DOM and updates views. Default strategy checks all components; `OnPush` checks only on input reference changes or explicit triggers. Knowing how it works helps debug why a view didn't refresh.

■ When/Why is it used?

Use change-detection debugging when UI is stale, bindings don't reflect model changes, or performance issues arise (too many cycles). It's crucial for reactive, high-performance UIs and for diagnosing async/update timing bugs.

■ Example / Practical techniques & snippets

1. **Identify strategy**

```
@Component({ changeDetection: ChangeDetectionStrategy.OnPush })
```

   - If `OnPush`, you must change object references or call `markForCheck()`.

2. **Log lifecycle hooks**

   - Add `ngDoCheck()` to see how and when detection runs:

```
ngDoCheck() { console.log('DoCheck', this.someProp); }
```

3. **Use** `ChangeDetectorRef`

```
constructor(private cd: ChangeDetectorRef) {}
// after mutating in async callback:
this.cd.markForCheck(); // schedules view update in OnPush
// or
this.cd.detectChanges(); // runs sync detection immediately
```

4. **Use** `NgZone` **for external callbacks**

```
constructor(private zone: NgZone) {}
someExternalCallback(data) {
```

```
  this.zone.run(() => { this.data = data; }); // re-enters Angular zone
}
```

5. **Prefer immutability for OnPush**

```
this.items = [...this.items, newItem]; // changes reference → triggers OnPush
```

6. **Debug tools**
   - Use **Augury** or Angular DevTools to inspect change detection cycles and component trees.
   - Add temporary `console.log` in component setters for `@Input()` to see when inputs change.
7. **Performance tip**
   - For expensive components, use `OnPush` + immutable data and memoize derived data with pure pipes.

■ Short summary line

👉 *Angular change detection updates views automatically for Default strategy; with OnPush you must change object references or call `markForCheck` / `detectChanges` (or run code inside `NgZone`) to trigger updates — use logs and DevTools to trace cycles.*

📌 Real-Time Scenarios

✅ 1) Application slow — where do you start debugging?

■ Explanation (simple, conceptual)

Performance issues can come from frontend rendering, backend/API latency, database slowness, network bottlenecks, or infrastructure (CPU/memory/IO). Start by narrowing the problem domain — is it client-side (browser) or server-side — then drill into the slowest component using observability (logs, traces, metrics) and profiling.

■ When/Why is it used?

You use this approach when users report slow pages, high response times, or SLA breaches. A methodical root-cause hunt avoids guessing and reduces mean time to resolution: identify whether to optimize Angular change detection, CDN/caching, SQL queries, or back-end service code.

■ Example / Actionable steps & commands

1. **Reproduce & measure**
   - Note exact URL, user, time, and steps. Use Lighthouse/Chrome DevTools `Performance` to get frontend load metrics (TTFB, FCP, TTI).
   - Use Postman or curl to measure API latencies.
2. **Top-level split: Frontend vs Backend**
   - If Network tab shows API responses slow → backend.

- If APIs are fast but page still slow → frontend rendering/asset issues.

3. **Frontend checks**
   - DevTools Network: check large assets, long JS parse time, blocking scripts.
   - Lighthouse: audit for render-blocking resources, large images, caching.
   - Angular: check change detection cycles, heavy template bindings or large lists; consider `OnPush`, virtual scroll, lazy loading.
4. **Backend checks**
   - Look at APM traces (Application Insights, NewRelic) to find slow endpoints and database calls.
   - Run `dotnet-counters` / `dotnet-trace` to inspect CPU and GC behavior.

```
dotnet-counters ps
dotnet-counters monitor --process-id <pid> System.Runtime
```

5. **Database and queries**
   - Capture slow queries via Query Store / `sys.dm_exec_query_stats` (SQL Server) and examine execution plans. Add missing indexes or rewrite queries.
6. **Infrastructure**
   - Check CPU, memory, disk I/O, network latency on servers/containers. Autoscaling or resource limits may be misconfigured.
7. **Quick fixes & caching**
   - Add HTTP caching, server-side caching (Redis), CDNs for static assets, database caching for frequent reads, or background batching for heavy writes.
8. **Validate & measure improvement**
   - After a fix, rerun benchmarks, Lighthouse, or load tests (k6/JMeter) to verify impact.

■ Short summary line

👉 *Start by splitting frontend vs backend, use DevTools/APM/DB tools to find the slowest layer, then optimize the specific hotspot (e.g., lazy load, indexing, caching) and re-measure.*

✅ 2) API returning data but UI not showing — steps?

■ Explanation (simple, conceptual)

When the API returns data but UI remains empty, the problem lies in the data pipeline between network → component → template. You must verify the API response, the HTTP handling in the client, component state updates, binding/template correctness, and change detection behavior.

■ When/Why is it used?

Use these steps during frontend debugging when network calls are successful but the user does not see expected data. This is common in Angular apps due to incorrect property bindings, OnPush

change detection issues, incorrect data mapping, or errors swallowed silently.

■ Example / Debugging checklist & code checks

1. **Network confirmation**
   - Open DevTools → Network → find the API call → confirm **status 200** and correct JSON body in Response.
2. **Console errors**
   - Check Console for runtime errors that might abort rendering (bindings, undefined properties).
3. **Verify the subscription / fetch code**

```
this.api.getItems().subscribe(
  data => {
    console.log('API data', data);
    this.items = data; // ensure assignment
  },
  err => console.error(err)
);
```

   - Add `console.log` to confirm `data` is assigned.
4. **Template binding correctness**
   - Confirm template uses the exact property (`items` vs `itemList`) and structural directives:

```
<div *ngFor="let it of items">{{it.name}}</div>
```

5. **Change detection / OnPush**
   - If component uses `OnPush`, ensure you change object references or call `markForCheck()`:

```
this.items = [...this.items, newItem];
// or
this.cd.markForCheck();
```

6. **Async pipe vs manual subscribe**
   - If using `| async`, make sure the Observable is assigned to a template variable. If using manual subscribe, ensure the component property is updated.
7. **Edge cases: empty arrays vs null**
   - UI may show nothing for `null` but should for `[]`. Check API returns `[]` not `null`.
8. **Invisible DOM / CSS**
   - Inspect Elements panel to check whether DOM nodes exist but are hidden by CSS (`display:none`), or overlay covers them.
9. **Permission/feature flags**
   - Ensure UI code doesn't hide data based on roles/flags; check the evaluated conditions.

10. **Unit/E2E test**

- Add a test that mocks the API and asserts DOM content rendering (Cypress / Karma).

■ Short summary line

👉 *Confirm the API response, verify client subscription and property assignment, check template bindings and change-detection (OnPush/async), inspect DOM/CSS, then add tests to prevent regressions.*

✅ 3) Memory leak — how will you detect?

■ Explanation (simple, conceptual)

A memory leak is when an application continuously consumes more memory over time because objects are retained (references not released), causing degraded performance or crashes. Detection requires profiling, heap snapshots, and looking for growing object counts or retained closures that should be collected.

■ When/Why is it used?

Use leak detection when memory usage steadily grows under normal load, GC runs but memory not reclaimed, or OOM crashes occur. In production systems (banking/automation) leaks can cause service outages—quick detection and remediation are critical.

■ Example / Tools & actionable steps

Frontend (Angular / Browser)

1. **Chrome DevTools Heap Snapshot**
   - Open DevTools → Performance / Memory → take heap snapshots over time while reproducing the scenario.
   - Compare snapshots to find objects that keep growing (DOM nodes, closures, event listeners).
2. **Find detached DOM nodes**
   - Search for "Detached DOM tree" objects in snapshots — indicates elements removed from DOM but still referenced in JS.
3. **Common causes & fixes**
   - Unsubscribed Observables (`subscribe` without `unsubscribe`) — use `takeUntil` or `async` pipe.
   - Long-lived timers (`setInterval`) not cleared — call `clearInterval`.
   - Global caches or static arrays growing without bounds — implement size limits or pruning.
   - Event listeners not removed on destroy — remove in `ngOnDestroy`.

```
ngOnDestroy() {
  this.subscription.unsubscribe();
  window.removeEventListener('resize', this.onResize);
}
```

Backend (.NET)

1. **Collect metrics & traces**
   - Monitor memory usage and GC metrics via App Insights / Prometheus / dotnet-counters.
2. **Heap dump & analysis**
   - Use `dotnet-gcdump` / `dotnet-dump` to capture a memory dump and analyze with Visual Studio or `dotnet-trace`.

     ```
     dotnet-dump collect -p <pid> -o dump.dmp
     ```

   - Load dump in Visual Studio or PerfView to inspect retained objects and dominators.
3. **Identify leaks**
   - Look for large object retention (e.g., static collections, caching without eviction, event handlers attached to long-lived objects).
4. **Fix patterns**
   - Dispose of `IDisposable` objects, detach event handlers, use weak references where appropriate, implement cache eviction policies.
5. **Validate**
   - Reproduce load test and ensure memory usage stabilizes after GC cycles.

■ Short summary line

👉 *Detect leaks with heap snapshots (Chrome) or memory dumps (dotnet-dump), look for growing retained objects (detached DOM, static collections, unsubscribed events), fix by unsubscribing/disposing and retest under load.*

✅ 4) Production issue debugging approach

■ Explanation (simple, conceptual)

Production debugging is a disciplined process: **observe → contain → diagnose → fix → validate → learn**. The priority is to minimize user impact while preserving evidence and applying a safe, tested fix. Communication and rollback strategies are also part of the playbook.

■ When/Why is it used?

Use this approach when customers report outages, error spikes, or critical regressions. It balances speed (mitigation) with correctness (root cause fix), while ensuring team coordination and post-incident prevention.

■ Example / Runbook-style steps

1. **Triage & communicate**
   - Acknowledge incident, inform stakeholders, open an incident channel, assign owner, and capture timeline.
2. **Gather evidence**

- Collect timestamps, request IDs, user IDs, logs, metrics, dashboards, and recent deployments. Correlate via trace IDs.

3. **Contain / Mitigate**
   - Apply quick mitigations: scale up, revert recent deploy, disable a feature flag, or route traffic to a healthy region. Prioritize non-invasive fixes first.
4. **Diagnose**
   - Use logs, APM traces, DB metrics, and health checks to find root cause. Reproduce safely in staging if possible.
5. **Implement fix**
   - Apply the smallest safe change (hotfix or rollback). Prefer config fixes, toggling flags, or releasing a tested patch.
6. **Validate**
   - Confirm metrics return to normal, run smoke tests, and verify affected user journeys.
7. **Post-mortem**
   - Document timeline, root cause, impact, actions taken, and remediation (tests, alerts, code changes). Share learnings.
8. **Preventive measures**
   - Add monitoring/alerts, increase test coverage, add rate-limits, add circuit breakers for external calls, implement schema/contract validations.
9. **Restore trust**
   - Communicate resolution to stakeholders and affected users, provide RCA and next steps.

Tools & patterns to mention

- Use structured logs with correlation IDs, APM (Application Insights/NewRelic), metric dashboards (Grafana/Prometheus), and feature flags (LaunchDarkly/rollout).
- Use `-force-with-lease` and rollback carefully if you revert a bad deploy.
- Automate runbooks for common incidents (DB connection, out-of-memory).

■ Short summary line

👉 *In production, triage and mitigate first (contain user impact), gather logs/traces to diagnose, apply a safe fix or rollback, validate thoroughly, then run a post-mortem and add preventive monitoring/tests.*

If you prepare these questions thoroughly → **Git + Postman + Debugging round is FULL COVERED.** 🔥
