**Common software development terms**

**Key Software Development Terms**

**Processes & Practices**

**Agile** - A flexible, iterative approach to software development (e.g., Scrum, Kanban).

**Scrum** - A popular Agile framework with sprints, daily stand-ups, and roles like Scrum Master.

**Kanban** - Visual workflow management method (e.g., boards with cards to track tasks).

**Waterfall** - A traditional linear software development process with sequential phases.

**SDLC (Software Development Life Cycle)** - The complete process of developing software, from planning to maintenance.

**Continuous Integration (CI)** - Automatically merging code changes into a shared repository several times a day.

**Continuous Deployment (CD)** - Automatically releasing every validated change to production.

**Continuous Delivery** - Similar to CD, but deployment requires a manual approval step.

**DevOps** - A culture and set of practices that bring together development and operations teams for faster delivery.

**Pair Programming** - Two programmers work together at one workstation: one writes code, the other reviews in real-time.

**Code Review** - Peers review code for quality, bugs, and style before merging.

**Refactoring** - Improving code structure without changing its external behavior.

**Version Control** - Systems (like Git) to track and manage changes to code.

**Tools & Technologies**

**Git** - A version control system.

**Repository (Repo)** - A storage location for code (e.g., on GitHub).

**Branch** - A parallel version of the codebase for testing new features or fixes.

**Merge** - Combining code from one branch into another.

**Pull Request (PR)** - A request to merge code changes, often with a review.

**Pipeline** - An automated series of steps to build, test, and deploy code.

**Docker** - Tool for packaging software into containers so it runs the same anywhere.

**Container** - A lightweight, standalone executable package with everything needed to run an application.

**Virtual Machine (VM)** - A software-based simulation of a computer.

**Cloud Computing** - Using remote servers (e.g., AWS, Azure, Google Cloud) instead of local machines.

**API (Application Programming Interface)** - A set of rules that lets software talk to other software.

**REST API** - A common style for designing web APIs.

**GraphQL** - An alternative API query language to REST.

**Framework** - Pre-written code structure to speed up development (e.g., React, Django).

**Library** - A collection of pre-written code used to solve common tasks.

**SDK (Software Development Kit)** - A set of tools to build applications for specific platforms.

**IDE (Integrated Development Environment)** - A software app for coding (e.g., VS Code, IntelliJ).

**Testing & Quality**

**Unit Testing** - Tests for individual pieces of code (functions, methods).

**Integration Testing** - Tests how multiple components work together.

**End-to-End (E2E) Testing** - Tests the whole application flow from start to finish.

**Mocking** - Faking parts of a system for testing.

**Linting** - Checking code for style errors and potential bugs.

**Static Analysis** - Examining code without running it to find bugs.

**Debugging** - Finding and fixing bugs in code.

**Security & Performance**

**Authentication** - Verifying user identity (login).

**Authorization** - Controlling what an authenticated user can do.

**Encryption** - Converting data to a secure format.

**SSL/TLS** - Protocols for secure web communication.

**Rate Limiting** - Restricting how often a user can make requests.

**Load Balancer** - Distributes traffic across servers to handle high loads.

**Latency** - Delay in processing or data transfer.

**Other Common Buzzwords**

**Microservices** - An architecture where an app is made of small, independent services.

**Monolithic** - A single, large codebase for the whole app.

**Scalability** - The ability to handle growing loads smoothly.

**Middleware** - Software that connects different parts of an app or system.

**Schema** - Structure of a database.

**ORM (Object Relational Mapper)** - A tool to interact with a database using programming language objects.

**CRUD (Create, Read, Update, Delete)** - Basic operations for managing data.

**Backend** - Server-side logic and database.

**Frontend** - What the user interacts with (UI, web pages).

**Full Stack** - Working on both backend and frontend.

**Responsive Design** - Making a website look good on all devices.

**Legacy Code** - Old code that is still in use.

**Tech Debt (Technical Debt)** - Extra work from quick fixes or poor design.

**Hotfix** - A quick bug fix in production.

**Patch** - Update to fix bugs or vulnerabilities.

**Release** - A version of software delivered to users.

**More Development Concepts**

**Staging Environment** - A replica of the production environment used to test new changes before releasing them live.

**Production Environment (Prod)** - The live system your users interact with.

**Local Environment** - Your personal computer setup where you write and test code.

**Rollback** - Reverting software to a previous version when something goes wrong.

**Hot Reload** - Instantly updating the app in the browser or simulator while coding, without restarting the server (e.g., React Native, Flutter).

**Build** - The compiled version of your application ready to run or deploy.

**Artifact** - The output of a build process (e.g., .jar, .exe, .apk).

**Dependency** - External code or libraries your project relies on.

**Dependency Management** - Tools and processes to handle libraries and versions (e.g., npm, pip).

**Package Manager** - A tool that automates installing and updating libraries (e.g., npm for JavaScript, pip for Python).

**Semantic Versioning (SemVer)** - A versioning system using MAJOR.MINOR.PATCH (e.g., 2.4.1).

**Legacy System** - An outdated computer system still in use.

**CI/CD Pipeline** - Automated steps that handle integration, testing, and deployment.

**Regression** - When a new change accidentally breaks something that worked before.

**Test Coverage** - The percentage of your code that’s covered by tests.

**Smoke Testing** - Basic tests to check if the main parts of an app work.

**Sanity Testing** - Quick checks after minor changes to verify functionality.

**Stress Testing** - Pushing the system beyond normal loads to find breaking points.

**Penetration Testing (Pen Test)** - Simulated cyber attack to find security weaknesses.

**Data & Database Terms**

**SQL (Structured Query Language)** - A language for managing relational databases.

**NoSQL** - Non-relational databases (e.g., MongoDB).

**Indexing** - Improving database query speed.

**Query** - A request for information from a database.

**Transaction** - A unit of work that must succeed completely or not at all.

**Normalization** - Organizing data to reduce redundancy.

**Denormalization** - Adding some redundancy for performance reasons.

**Data Migration** - Moving data from one system to another.

**Popular Architecture & Patterns**

**Client-Server Model** - The client requests services, the server provides them.

**Serverless** - Running code in the cloud without managing servers directly.

**Event-Driven Architecture** - Code that reacts to events (e.g., button clicks, sensor data).

**Singleton** - A design pattern that restricts a class to one instance.

**MVC (Model-View-Controller)** - A design pattern to separate logic, UI, and data.

**MVVM (Model-View-ViewModel)** - A variation of MVC popular in frontend frameworks.

**Observer Pattern** - When objects watch other objects for changes.

**Networking & APIs**

**Webhooks** - Automated messages sent from one system to another when an event happens.

**OAuth** - An open standard for secure authorization (logging in with Google/Facebook).

**JWT (JSON Web Token)** - A compact way to securely transmit user information.

**CORS (Cross-Origin Resource Sharing)** - A security feature controlling which domains can access your API.

**Latency** - Delay in data transfer.

**Bandwidth** - The amount of data transferred in a given time.

**Bonus: Misc Terms**

**Bug** - An error or flaw in the software.

**Issue Tracker** - A tool to manage bugs and feature requests (e.g., Jira).

**Epic** - A large body of work that can be broken down into smaller tasks (stories).

**User Story** - A simple description of a feature from the user’s perspective.

**Backlog** - A list of tasks or features to work on.

**Sprint** - A short period (1-4 weeks) to complete a set amount of work.

**Standup** - A quick daily team meeting to sync up.

**Retrospective** - A meeting to discuss what went well and what can improve after a sprint.

**PO (Product Owner)** - The person responsible for defining features and priorities.

**Stakeholder** - Anyone with an interest in the project (customers, managers, investors).

**Verification** answers the question, "Are we building the product right?" It is a static process that evaluates the software's artifacts (like requirements, design documents, and code) to ensure they conform to specified requirements and standards. Verification activities, such as reviews, walkthroughs, inspections, and static code analysis, typically happen throughout the software development lifecycle, aiming to catch errors as early as possible before they propagate to later stages and become more costly to fix.

**Validation** answers the question, "Are we building the right product?" It is a dynamic process that evaluates the finished software product to ensure it meets the user's actual needs, expectations, and the original business requirements. Validation typically involves executing the software (through various forms of testing like system testing, user acceptance testing, and beta testing) to confirm its functionality, performance, usability, and overall suitability for its intended use in real-world scenarios. This phase usually occurs towards the end of the development cycle or after specific modules are fully built.

**Extended Software Development Terms & Definitions**

**Architectural & Design Concepts**

**Microservice**: A small, independent service that does one thing well and communicates with other services over a network. Popular in scalable backend systems.

**Monolith**: Opposite of microservices - an entire application built as one large unit.

**Serverless**: Cloud computing where you run functions without managing servers yourself - e.g. AWS Lambda.

**API Gateway**: A server that acts as an entry point for APIs, handling authentication, routing, and rate limiting.

**Load Balancer**: A tool that distributes incoming network traffic across multiple servers to keep things fast and available.

**Container**: A lightweight, portable software package that includes everything needed to run an application (e.g., Docker).

**Orchestration**: Managing multiple containers/services automatically (e.g., Kubernetes).

**Web & Browser Basics**

**Cookie**: Small data stored by a website in the user’s browser to remember things (e.g., login).

**Session**: A temporary state that keeps track of a user’s activity on a website until they log out or close the browser.

**Local Storage**: Browser-based storage that can keep data even after the session ends.

**Cache**: Temporary storage to make sites load faster by saving copies of files locally.

**CDN (Content Delivery Network)**: A global network of servers that deliver content faster by serving it from a location close to the user.

**Programming Practices**

**Refactoring**: Improving code structure without changing its behavior.

**DRY (Don’t Repeat Yourself)**: A principle to avoid code duplication.

**YAGNI (You Aren’t Gonna Need It)**: Don’t add functionality until it’s necessary.

**KISS (Keep It Simple, Stupid)**: Keep code simple and clear.

**SOLID**: Five principles for clean, maintainable object-oriented design:

* S: Single Responsibility
* O: Open/Closed
* L: Liskov Substitution
* I: Interface Segregation
* D: Dependency Inversion

**Version Control & Deployment**

**Branch**: A separate line of development in Git.

**Merge**: Combining code changes from one branch to another.

**Pull Request (PR)**: A request to merge your branch into the main codebase.

**Fork**: A copy of a repository you can freely modify.

**Pipeline**: Automated process for building, testing, and deploying code.

**Artifact**: A file or bundle produced during the build process (e.g., .jar, .zip).

**Security**

**Token**: A small piece of data used to prove identity (e.g., JWT).

**OAuth**: A standard for secure delegated access (e.g., “Log in with Google”).

**CSRF (Cross-Site Request Forgery)**: An attack tricking a user into performing actions they didn’t intend.

**XSS (Cross-Site Scripting)**: An attack where malicious scripts run in the user’s browser.

**SSL/TLS**: Encryption protocols that secure data between your browser and the server (HTTPS).

**Data & Backend**

**ORM (Object-Relational Mapping)**: A tool that lets you interact with databases using objects (e.g., SQLAlchemy, Hibernate).

**NoSQL**: A type of database that doesn’t use traditional tables - e.g., MongoDB.

**Schema**: The structure/blueprint of a database.

**Query**: A request for data from a database.

**Endpoint**: A specific URL where an API receives requests.

**Cloud & DevOps**

**Infrastructure as Code (IaC)**: Managing servers and cloud resources using code files (e.g., Terraform).

**CI/CD**: Continuous Integration/Continuous Deployment - automating testing and delivery.

**Provisioning**: Setting up cloud resources automatically.

**Rolling Update**: Deploying changes gradually to reduce downtime.

**Frontend Development**

**DOM (Document Object Model)**: The structure of an HTML page as seen by JavaScript.

**SPA (Single Page Application)**: A web app that loads one page and updates content dynamically.

**Component**: A reusable piece of UI (React, Vue, Angular).

**State Management**: Keeping track of app data (Redux, Vuex).

**Webpack**: A bundler that compiles JavaScript, CSS, and assets.

**Bonus: Misc**

**Webhook**: A way for one app to send real-time data to another (e.g., payment notifications).

**CLI (Command Line Interface)**: A text-based way to interact with your computer or tools.

**SDK (Software Development Kit)**: A bundle of tools to build apps for a platform.

**IDE (Integrated Development Environment)**: A complete coding workspace (VSCode, PyCharm).

**Linting**: Checking code for errors or style issues.

**Errors**

An **error** is a technical problem in the code or system that prevents it from working correctly. It’s something the computer or compiler detects (syntax error, runtime error, type error, etc). Example:

1. A missing semicolon causes a syntax error.
2. Dividing by zero causes a runtime error.

A **mistake** is a human action that introduces an error. It’s the cause of the error, not the error itself. The computer does not know it’s a “mistake”, only that it sees an “error”. Example:

* A programmer writes = instead of ==.
  + The mistake is using = (assignment) when == (comparison) was intended.
  + The error is the logic or syntax problem it causes in the code.

**Forgetting a semicolon in code**

**Mistake** = cause (human action) → You forgot the semicolon.

**Error** = result (computer’s reaction) → The compiler reports a syntax error.

**Common types of programming errors**

**a) Syntax Error**

Mistakes that break the rules of the programming language. Example: missing semicolon, unmatched brackets, wrong keywords. Won’t run at all until fixed.

**b) Runtime Error**

The program compiles (or starts) but fails during execution. E.g., divide by zero, accessing a missing file. Example: int x = 5/0; → crash at runtime.

**c) Logic Error**

The code runs without crashing, but does the wrong thing. Example: Using = instead of == in a condition. Hardest to detect - it does not throw an error but gives wrong results.

**d) Type Error**

Mixing data types incorrectly. Example: Adding a number to a string without conversion.

**e) Semantic Error**

Similar to logic error - the code means something different than the programmer intended. Often overlaps with logic error. It’s a logic mistake that changes the intended meaning. The compiler/interpreter won’t detect it - you have to find it yourself!

int a = 10;

int b = 20;

int average = a + b / 2;

printf("%d", average);

**Maintenance**

Maintenance in ICT (Information and Communication Technology) means all the activities done regularly to keep hardware, software, or systems working properly, fix problems, improve performance, and prevent failures.

Maintenance ensures that computers, networks, programs, and other ICT tools stay reliable, secure, and efficient throughout their lifetime.

**Types of Maintenance**

In ICT and software/hardware management, maintenance is commonly grouped into 4 main types:

**1. Corrective Maintenance**

Involves fixing problems after they occur. Example: Repairing a broken printer, fixing a software bug, or replacing a faulty hard drive.

**2. Preventive Maintenance**

Involves doing regular checks and care to prevent failures in the first place. Example: Cleaning computers to prevent dust damage, updating antivirus, backing up data, or checking hardware connections.

**3. Adaptive Maintenance**

Involves making a system work in a new environment or with new requirements. Example: Updating a program so it works with a new operating system or adapting an app to support a new device.

**4. Perfective Maintenance**

Involves improving a system’s performance or adding small new features - not fixing errors, but making it better. Example: Optimizing code to run faster, improving user interfaces, or adding a simple feature to help users.

**Other Useful Terms**

* **Emergency Maintenance**: Urgent fixes needed immediately to keep things running (usually part of corrective maintenance).
* **Predictive Maintenance**: Using data and monitoring tools to predict when something will fail and fix it before it does (common in modern large systems).

In short: Maintenance = Keeping ICT systems reliable and useful through planned checks, problem-fixing, updates, and improvements.

**Testing**

**Testing** is the process of checking whether a system, program, or component works correctly.

It helps find errors (bugs), ensures the system meets user requirements, and confirms it’s reliable, safe, and ready to use. In ICT or software, testing is done before, during, and after development - and sometimes during maintenance.

**Why do we test?**

1. To find and fix bugs or mistakes.
2. To check if all parts work together properly.
3. To make sure the system does what users expect.
4. To reduce failures when the system goes live.

**Types of testing**

**1. Unit Testing**

Involves testing individual pieces of code (like functions or modules) separately to make sure each works correctly. Example: Checking if a login function accepts the correct password.

**2. Integration Testing**

Involves testing how different parts (modules) work together. Example: After unit testing the login and database parts, check if they connect properly.

**3. System Testing**

Involves testing the complete system as a whole to see if it meets the requirements. Example: Running the entire application to check all features - login, search, print, etc.

**4. Acceptance Testing**

Done by the end user or client to confirm the system meets their needs. Example: The user checks if a new HR system does all tasks they asked for.

**5. Regression Testing**

Re-testing the system after changes or bug fixes to ensure nothing else broke. Example: After fixing a search bug, test the whole search and related features again.

**6. Performance Testing**

Checks how the system performs under load - speed, response time, stability. Example: See if an app can handle 1000 users at once without crashing.

**7. Stress Testing**

Pushes the system beyond normal limits to see how it behaves under extreme conditions. Example: What happens if 10,000 users log in at once?

**8. Usability Testing**

Checks how easy and user-friendly the system is. Example: See if new users can easily navigate a website.

**9. Security Testing**

Checks for weaknesses that hackers or viruses could exploit. Example: Test if passwords are stored securely.

**10. Alpha & Beta Testing**

**Alpha Testing**: Done by developers/testers inside the company before releasing to users.

**Beta Testing**: Done by a small group of real users to find final issues before the public release.

**Examples**

**1. Unit Testing**

**Example**:

You wrote a function addNumbers(a, b) that returns a + b.

You write unit tests like:

* addNumbers(2, 3) should return 5
* addNumbers(-1, 1) should return 0

**2. Integration Testing**

**Example**:

You have:

* A login module
* A user database

You test:

* Does the login page correctly send the username/password to the database?
* Does it return the right user if the credentials are valid?

**3. System Testing**

**Example**:

You built an entire e-commerce website. System testing checks:

* Can users register accounts?
* Can they search for products?
* Can they add to cart, checkout, and get email receipts?

**4. Acceptance Testing**

**Example**:

Your client wants a payroll system that generates monthly pay slips. The client checks:

* Are salaries calculated correctly?
* Are tax deductions applied properly?
* Does the payslip show the correct employee details?

**5. Regression Testing**

**Example**:

You fixed a bug in the shopping cart’s total price calculation. Regression test:

* Add 5 items to the cart → check total
* Remove an item → check total updates correctly
* Make sure other features like “save cart” still work.

**6. Performance Testing**

**Example**:

* A social media app must load profiles fast. You check:
* How long does the feed take to load with 1,000 posts?
* Do messages send instantly with 200 people online?

**7. Stress Testing**

**Example**: A ticket booking site expects normal traffic of 100 users per minute. Stress test:

* Simulate 5,000 users booking at once - does it crash? Does it recover?

**8. Usability Testing**

**Example**: You built an online bank app. Ask real users:

* Can they find the “Transfer Money” button easily?
* Do they understand error messages?
* Do they complete tasks without confusion?

**9. Security Testing**

**Example**: Your system stores user passwords. Security test:

* Try SQL injection to see if database can be hacked.
* Check if passwords are hashed/encrypted.
* Test if login locks after repeated wrong tries.

**10. Alpha & Beta Testing**

**Example**:

* **Alpha**: Developers inside your company test the app in a test environment - they find obvious bugs.
* **Beta**: Release the app to a small group of customers - they use it in real life and report real-world issues you didn’t see.

**Tips for Preparing Test Plans**

1. Write clear goals: What feature are you testing?
2. Write expected outcomes: What should happen if it works?
3. Include edge cases: Wrong inputs, empty inputs, big numbers, symbols, etc.
4. Record results: Pass or fail - note bugs.
5. Repeat often: Test again after any code change.

**Summary of Types of Testing**

|  |  |  |
| --- | --- | --- |
| **SN** | **Type** | **What it checks** |
| 1. | Unit | Smallest code parts |
| 2. | Integration | Connections between parts |
| 3. | System | Whole system together |
| 4. | Acceptance | Meets client/user needs |
| 5. | Regression | Changes didn’t break things |
| 6. | Performance | Speed and capacity |
| 7. | Stress | Behavior under extreme load |
| 8. | Usability | User-friendliness |
| 9. | Security | Safety from attacks |
| 10. | Alpha/Beta | Early real-world feedback |

**ChatGPT Q&A based on ICT Officer (Programmer) - MOI**

1. Q: What are the key steps you follow when designing a computer system chart?

A: I start by understanding user requirements, then I break down processes into modules, map input/output, define data flows, and represent the workflow visually using flowcharts or UML diagrams for clarity.

2. Q: How do you ensure that your systems documentation is clear and complete?

A: I write documentation in simple language, include diagrams, use consistent naming conventions, maintain version control, and review with users to confirm accuracy.

3. Q: What is the purpose of a systems flowchart?

A: A systems flowchart shows how data flows through a system - from input, processing, to output - helping developers and users understand the steps and dependencies.

4. Q: How would you help a user during system analysis?

A: I’d conduct interviews, gather requirements, observe workflows, clarify needs, identify pain points, and translate these into technical specifications.

5. Q: Describe a time you executed a test plan. What did you do?

A: I prepared test cases from user requirements, ran the tests systematically, logged defects, corrected bugs, re-tested fixes, and documented all outcomes for quality assurance.

6. Q: How do you make sure your test plans meet quality standards?

A: I align test plans with company quality guidelines, use checklists, review with peers, and get user approval to ensure all functions are covered.

7. Q: Give an example of routine ICT maintenance you can perform.

A: I can clean and check hardware, update software, check backups, ensure anti-virus is working, and replace faulty peripherals.

8. Q: How do you ensure compliance with standard operating procedures (SOPs)?

A: I always review the SOPs before starting tasks, keep documentation updated, and follow checklists to confirm all steps are done as required.

9. Q: Why is it important to have systems documentation?

A: It helps new developers understand the system, supports troubleshooting, ensures continuity when staff change, and is required for audits or compliance.

10. Q: How do you handle an urgent user request while working on another task?

A: I assess priority, communicate timelines clearly, manage expectations, and, if critical, switch tasks or delegate as appropriate.

11. Q: How would you explain a technical problem to a non-technical user?

A: I’d avoid jargon, use simple analogies, provide visuals if needed, and check for understanding by asking questions.

12. Q: What tools do you use to design systems flow charts?

A: Tools like Microsoft Visio, Lucidchart, Draw.io, or basic diagramming features in Word or PowerPoint.

13. Q: Why is documentation of test plans important?

A: It ensures that tests are repeatable, provides evidence of quality checks, helps track issues, and supports future system upgrades.

14. Q: What would you do if you discover a user is not following standard procedures?

A: I’d talk to the user, explain the risk, offer training if needed, and report to the supervisor if it continues.

15. Q: What programming languages or tools are you most comfortable with?

A: (Model: Candidate should answer honestly, e.g.,) I’m comfortable with Python, JavaScript, SQL, and have used frameworks like Django and Laravel.

16. Q: How do you handle a situation where system requirements keep changing?

A: I’d communicate the impact of changes on timelines and cost, document all changes, and ensure stakeholder sign-off before implementing.

17. Q: What measures do you take to prevent data loss during maintenance?

A: I ensure backups are up-to-date, verify recovery procedures, and test restores regularly.

18. Q: How would you improve an existing system?

A: By gathering user feedback, identifying inefficiencies, analyzing logs for bottlenecks, and proposing updates or automation where practical.

19. Q: Why is it important to perform low-level maintenance regularly?

A: It prevents bigger problems, extends equipment life, ensures performance, and reduces unexpected downtime.

20. Q: What qualities make you a good fit for this ICT Officer (Programmer) role?

A: I have strong analytical skills, I write clear documentation, I support users patiently, I test thoroughly, and I follow standard procedures reliably.

21. Q: How would you gather user requirements for a new system?

A: I’d organize interviews, surveys, observe workflows, hold workshops, and document requirements clearly for user sign-off.

22. Q: What is the difference between a data flow diagram and a systems flowchart?

A: A data flow diagram shows how data moves between processes. A systems flowchart maps the physical flow of information and the sequence of operations in a system.

23. Q: Explain why version control is important for system documentation.

A: It tracks changes, prevents data loss, allows rollback if needed, and keeps everyone working on the latest version.

24. Q: If a user struggles to adapt to a new system, how would you help?

A: I’d provide one-on-one support, create step-by-step guides, answer questions patiently, and collect feedback for further improvements.

25. Q: What are test cases, and how do you write them?

A: Test cases are step-by-step instructions to verify that a system feature works as expected. They include inputs, actions, expected results, and actual results.

26. Q: What would you check first if a computer won’t power on?

A: Check power cables, power source, power button, and if necessary, test with another power supply or outlet.

27. Q: Why is proper documentation helpful during maintenance?

A: It provides guidance for troubleshooting, records hardware/software configurations, and makes handover easier.

28. Q: What is your approach to training new users on a system?

A: I’d do a simple demo, provide written instructions, let users practice, and answer questions on the spot.

29. Q: Why is routine preventive maintenance important?

A: It detects problems early, reduces breakdowns, improves performance, and extends equipment life.

30. Q: How do you secure user data in a system you design?

A: By using access controls, encryption, secure login procedures, regular backups, and compliance with data protection policies.

31. Q: Explain the difference between black box and white box testing.

A: Black box tests functionality without looking at the internal code. White box testing checks internal logic and code structure.

32. Q: How do you handle multiple tasks with tight deadlines?

A: I prioritize by urgency and impact, break tasks into smaller steps, communicate clearly, and stay organized.

33. Q: Give an example of a standard operating procedure you follow.

A: For software updates: check compatibility, back up data, notify users, perform the update during low-usage hours, and test afterward.

34. Q: What does routine maintenance of ICT equipment include?

A: Cleaning hardware, checking connections, updating software, scanning for viruses, and testing backups.

35. Q: What programming best practices do you follow?

A: Use clear variable names, comment code, follow consistent style, test frequently, and keep backups.

36. Q: If you find a security vulnerability while working, what would you do?

A: Document it, inform the supervisor immediately, recommend fixes, and assist in patching it.

37. Q: What role does user feedback play in system improvement?

A: It identifies bugs, usability issues, and new needs, helping refine the system for better performance.

38. Q: How do you make sure your work aligns with business processes?

A: By studying business workflows, consulting stakeholders, and designing solutions that support existing processes.

39. Q: Why is testing important before deploying a system?

A: It finds errors early, saves time and cost later, ensures reliability, and builds user trust.

40. Q: What would you do if a supervisor assigns you a task outside your job description?

A: I’d do my best to handle it, ask for guidance if needed, and ensure it doesn’t affect my core duties.

41. Q: What steps do you take to keep your system designs up to date?

A: I review user feedback, monitor system performance, check for technology updates, and update documentation regularly.

42. Q: How do you handle bugs that appear after a system goes live?

A: I replicate the bug, identify the cause, fix it in a test environment, get user confirmation, then deploy the fix with proper documentation.

43. Q: What is the difference between debugging and testing?

A: Testing checks for errors in a system; debugging finds the cause of an error and fixes it.

44. Q: How do you prioritize tasks when multiple users request help?

A: I assess urgency and impact, handle critical issues first, communicate timelines, and keep users updated.

45. Q: Why is backup important before system maintenance?

A: It ensures that if anything goes wrong, data can be recovered without loss or corruption.

46. Q: How would you handle confidential information you come across while working?

A: I’d follow data privacy policies, avoid sharing it, secure the files, and report breaches immediately.

47. Q: What’s your approach to troubleshooting hardware faults?

A: I inspect for physical damage, check power and connections, run diagnostics, swap parts to isolate faults, and replace or repair as needed.

48. Q: What makes a good test plan?

A: Clear objectives, detailed steps, expected results, test data, and a way to log and track defects.

49. Q: How do you make sure your system is user-friendly?

A: I use simple interfaces, clear instructions, consistent design, gather user feedback, and test usability with actual users.

50. Q: Explain how you’d assist a user who is struggling with a new feature.

A: I’d demonstrate the feature step-by-step, watch them try it, correct mistakes gently, and provide a quick reference guide.

51. Q: What is your strategy for documenting system changes?

A: I record what was changed, why it was done, who approved it, the date, and update the main documentation repository.

52. Q: Why is it important to separate development, testing, and production environments?

A: To prevent unfinished or faulty code from affecting live systems, and to test changes safely before deployment.

53. Q: How would you handle a disagreement with a user about system requirements?

A: I’d listen carefully, explain technical constraints, find a compromise that meets user needs while staying feasible, and document decisions.

54. Q: What should you do before running a routine update?

A: Back up data, inform users, check update notes for risks, schedule downtime if needed, and test after the update.

55. Q: How do you make sure the systems you work on stay secure?

A: Use secure coding practices, regular updates, strong passwords, limit user permissions, and monitor logs for suspicious activity.

56. Q: What is your biggest challenge when working with non-technical users?

A: Translating technical terms into simple language - I overcome this with patience, examples, and clear visuals.

57. Q: What do you do if your test plan fails repeatedly?

A: Re-examine requirements, check for errors in test cases, fix any code issues, and run new tests until results are correct.

58. Q: Give an example of a low-level ICT maintenance task.

A: Cleaning dust from hardware, updating antivirus definitions, replacing a faulty keyboard, or defragmenting a hard disk.

59. Q: What programming principle do you follow to make your code maintainable?

A: I write modular, well-commented code, follow naming conventions, avoid duplication, and keep functions small and clear.

60. Q: Why is teamwork important in system development?

A: Different roles bring different expertise - teamwork ensures better design, faster troubleshooting, and higher user satisfaction.

61. Q: What would you do if you suspect a user is misusing a system?

A: I’d gather evidence carefully, follow the organization’s policy for reporting misuse, and alert my supervisor discreetly.

62. Q: Why is user training important during system implementation?

A: It helps users adapt quickly, reduces mistakes, increases productivity, and ensures the system is used as designed.

63. Q: What is your approach to managing system changes to avoid downtime?

A: Plan changes during low-use hours, inform users in advance, back up data, test changes first, and monitor afterward.

64. Q: If a user loses an important file, how would you help?

A: I’d check the Recycle Bin, search backups, try file recovery tools if needed, and advise on better file-saving habits.

65. Q: How can you reduce hardware failure?

A: By doing preventive maintenance: keeping equipment clean, checking cables, monitoring performance, and replacing worn-out parts early.

66. Q: What’s the purpose of a system flowchart?

A: To show how data and processes flow in a system, making it easier to design, troubleshoot, and document.

67. Q: What do you do if your test plan passes but users still find bugs?

A: I’d review the test plan for gaps, expand test cases, fix the bugs, and update the plan for next time.

68. Q: How do you protect sensitive documents on a shared computer?

A: Use strong passwords, limit access rights, encrypt files if needed, and log out after use.

69. Q: Give an example of a simple quality management method you follow.

A: I peer-review code with colleagues, run consistent test scripts, and document issues and solutions.

70. Q: How do you balance speed and quality in programming?

A: I write clear, modular code and test as I build - this avoids rushing fixes later and keeps quality high.

71. Q: What do you include in a systems design document?

A: System overview, objectives, data flow diagrams, hardware/software requirements, user roles, and testing plans.

72. Q: Why is regular communication with users important during system design?

A: It helps gather clear requirements, prevents misunderstandings, and builds trust for a smoother implementation.

73. Q: What would you do if a piece of ICT equipment repeatedly fails?

A: Check for root cause (power, usage, age), repair or replace it, and document the issue for future reference.

74. Q: How do you handle confidential test data?

A: Keep it secure, limit who sees it, delete it safely when done, and never share it without permission.

75. Q: What is your role when working with a systems analyst?

A: I support by providing technical details, helping design charts, documenting flows, and building/test coding as required.

76. Q: Why are standard operating procedures (SOPs) important?

A: They keep tasks consistent, reduce errors, maintain quality, and help new staff learn faster.

77. Q: What do you do if a user demands a feature you know is risky?

A: I explain the risks, offer safer alternatives, and escalate to my supervisor if needed.

78. Q: How do you check that your systems meet business process requirements?

A: By mapping workflows, getting user feedback, testing with real data, and adjusting as needed.

79. Q: What are your steps for a simple hardware upgrade?

A: Back up data, power off, replace the part, test the device, and update documentation.

80. Q: Why is it important to document even small system changes?

A: So future troubleshooting is easier, other team members know what’s changed, and audits run smoothly.

81. Q: Why is version control important when working on system code?

A: It keeps track of changes, allows you to revert to earlier versions if needed, and makes teamwork easier by managing multiple updates.

82. Q: How do you handle repetitive user mistakes in a system?

A: I look for root causes, improve the system’s design if possible, and provide simple user training or guides.

83. Q: What would you do if you accidentally caused a system failure?

A: I’d inform my supervisor immediately, follow backup and recovery procedures, fix the issue, and document what happened to prevent repeat mistakes.

84. Q: Why is testing with real data important?

A: It shows how the system handles actual scenarios and helps uncover hidden bugs that test data might not reveal.

85. Q: What is the benefit of user feedback during system testing?

A: Users often spot practical issues developers may overlook - their feedback helps improve usability and reliability.

86. Q: How do you make sure your flowcharts are clear?

A: I use standard symbols, label steps clearly, keep lines neat, and test the chart with a colleague for understanding.

87. Q: How would you handle unauthorized software installed on a work computer?

A: Report it to the supervisor, explain the security risks, and help remove it safely.

88. Q: Why should you update your documentation regularly?

A: So that future maintenance, upgrades, or troubleshooting can be done quickly and accurately by anyone.

89. Q: How do you protect your code from unauthorized edits?

A: I use secure passwords, version control with permissions, and store backups in a safe location.

90. Q: How do you verify if low-level maintenance tasks are done properly?

A: I double-check the work, test the equipment, and keep a log of what was done and when.

91. Q: What would you do if a user keeps ignoring standard procedures?

A: I’d remind them of the importance, explain risks, and if it continues, report to my supervisor to handle it formally.

92. Q: Why is it important to document test plan results?

A: To prove the system works as intended, show compliance with quality standards, and help troubleshoot later issues.

93. Q: What is a key sign that your test plan may need improvement?

A: If bugs keep slipping through or users report repeated issues despite testing.

94. Q: How do you know if your system charts are complete?

A: They clearly show all inputs, outputs, processes, and data flow with no missing connections.

95. Q: What does routine hardware maintenance include?

A: Cleaning, checking connections, updating drivers, testing performance, and replacing failing parts.

96. Q: How do you keep your programming skills updated?

A: I read documentation, join online courses, test new techniques, and practice coding regularly.

97. Q: How do you handle tight deadlines with complex tasks?

A: I break tasks into parts, prioritize essentials, communicate realistic timelines, and ask for help if needed.

98. Q: Why should you follow business processes when designing systems?

A: To make sure the system fits how the organization works, avoids unnecessary steps, and improves productivity.

99. Q: What is the biggest risk if you skip user documentation?

A: Users may misuse the system, make errors, or fail to benefit fully - which leads to more support issues.

100. Q: Why is it important to keep your supervisor informed of your progress?

A: It builds trust, helps them coordinate tasks, and ensures you get support or guidance when needed.

**Possible past Questions**

**Tools for CI/CD**

CI/CD (Continuous Integration/Continuous Delivery or Continuous Deployment) is a set of practices that automates the stages of software delivery.

**CI/CD Tools Overview**

**CI (Continuous Integration) Tools**

|  |  |
| --- | --- |
| **Tool** | **Description** |
| Jenkins | A widely used open-source automation server. Supports many plugins for building, testing, and deploying code. |
| GitLab CI/CD | Built directly into GitLab. Provides end-to-end automation from coding to deployment. |
| GitHub Actions | Native CI/CD tool integrated into GitHub. Allows workflows triggered by repository events. |
| CircleCI | Cloud-based CI/CD tool known for speed and easy setup with GitHub/GitLab. |
| Travis CI | A simple CI tool, especially good for open-source projects hosted on GitHub. |
| Bitbucket Pipelines | Integrated CI/CD tool in Bitbucket, useful for teams using Atlassian tools. |

**CD (Continuous Deployment/Delivery) Tools**

|  |  |
| --- | --- |
| **Tool** | **Description** |
| Argo CD | A GitOps-based continuous delivery tool for Kubernetes deployments. |
| Spinnaker | Multi-cloud delivery tool used for safe and repeatable releases. |
| Flux | Lightweight GitOps tool for automating Kubernetes deployments. |
| Octopus Deploy | Used for complex deployment automation, especially in .NET environments. |
| AWS CodeDeploy | Part of AWS CI/CD services, allows automated deployment to EC2, Lambda, etc. |

**Containerization & Orchestration**

|  |  |
| --- | --- |
| **Tool** | **Description** |
| Docker | Packages apps and dependencies into containers. Essential for microservices and CI/CD. |
| Kubernetes | Manages and orchestrates containers. Works with tools like Argo CD or Flux for CD. |
| Helm | Kubernetes package manager that simplifies deployment and configuration of apps. |

**Artifact Repositories**

|  |  |
| --- | --- |
| **Tool** | **Description** |
| JFrog Artifactory | Stores build artifacts like JARs, Docker images, etc. |
| Sonatype Nexus | Repository manager similar to Artifactory; supports Maven, npm, Docker, and more. |

**Monitoring & Feedback Tools**

|  |  |
| --- | --- |
| **Tool** | **Description** |
| Prometheus | Monitoring system and time-series database, often used with Kubernetes. |
| Grafana | Visualizes monitoring data from Prometheus and other sources. |
| ELK Stack (Elasticsearch, Logstash, Kibana) | Log collection and visualization stack for analyzing CI/CD logs and metrics |

If you're building CI/CD pipelines, you typically:

* Use GitHub/GitLab to host code
* Set up Jenkins/GitHub Actions/GitLab CI for building and testing
* Deploy using Docker + Kubernetes + ArgoCD/Flux
* Store artifacts in Artifactory/Nexus
* Monitor deployments with Prometheus + Grafana

**Kubernetes for microservice architecture**

Kubernetes is a container orchestration platform that manages and deploys microservices across a cluster. It handles scaling, load balancing, fault tolerance, and service discovery, making it ideal for running multiple independent microservices.

**Kubernetes** (often abbreviated as K8s) is an open-source platform designed to automate the deployment, scaling, and management of containerized applications, especially across clusters of servers. It focuses on managing many containers, often from different applications, across multiple machines.

**Key capabilities of Kubernetes:**

1. **Orchestration**: Automates tasks like scheduling containers, managing their lifecycle, scaling them up or down based on demand.
2. **Self-healing**: If a container fails, Kubernetes can automatically restart it, replace it, or reschedule it to another healthy server.
3. **Load Balancing**: Distributes network traffic across multiple instances of your application to ensure high availability and performance.
4. **Service Discovery**: Allows containers to find and communicate with each other easily.
5. **Automated Rollouts and Rollbacks**: Manages updates to your application without downtime and allows easy reversion to previous versions if issues arise.
6. **Resource Management**: Efficiently allocates CPU, memory, and other resources to containers across the cluster.

**Docker** is a set of tools and a platform for building, packaging, and running individual applications in isolated units called containers.

**Format for documentation**

Common formats include:

* Markdown (.md) – Simple, widely used (e.g., GitHub README).
* HTML / PDF – For formal, distributable documents.
* Swagger/OpenAPI – For API documentation.
* RST (reStructuredText) – Used in Python/Sphinx projects.

**Angular features**

1. Two-way data binding
2. Component-based architecture
3. Dependency Injection (DI)
4. Routing
5. Directives and Pipes
6. RxJS for reactive programming
7. CLI for scaffolding projects

**TDD - Test Driven Development**

A software development approach where:

1. Write tests first.
2. Write minimal code to pass the test.
3. Refactor - Ensures clean, tested, and bug-resistant code from the start.

**Single Page Application (SPA)**

A web app that loads a single HTML page and updates content dynamically via JavaScript without reloading the page. Examples: Gmail, Google Docs, Facebook. Frameworks to use include are React, Angular, Vue.

**Indexes in database**

Indexes improve query speed by allowing the database to find data faster (like a book index).

* Types: B-Tree, Hash, Composite, Full-Text.
* Downsides: Slightly slower writes and more storage.

**View in database**

A view is a virtual table based on a SQL query.

* It does not store data, only the result of a query.
* Useful for abstraction, simplification, and security (hiding columns).

**Tools for restful API**

1. Postman - Testing and documenting REST APIs.
2. Swagger (OpenAPI) - API design, documentation, and mocking.
3. Insomnia - REST/GraphQL client for debugging APIs.
4. Express.js (Node.js) - Lightweight web framework to build RESTful services.
5. Flask/Django (Python) - Web frameworks with REST API support.
6. Spring Boot (Java) - Common for building enterprise-level REST APIs.

**Rest vs GraphQL**

**REST (Representational State Transfer)**

REST (Representational State Transfer) and GraphQL (Graph Query Language) are both architectural styles for designing APIs (Application Programming Interfaces) that allow client applications to communicate with servers to exchange data. While they both achieve this goal, they do so with fundamentally different approaches, each with its own strengths and weaknesses.

**Key Characteristics of REST:**

1. **Resources**: Everything is a resource (e.g., /users, /products/123).
2. **Unique URLs (Endpoints)**: Each resource or collection of resources has a distinct URL (e.g., api.example.com/users, api.example.com/products/1).
3. **HTTP Methods**: Uses standard HTTP verbs to perform actions on resources:

* **GET**: Retrieve a resource or a collection of resources.
* **POST**: Create a new resource.
* **PUT**: Update an existing resource (typically replaces the entire resource).
* **PATCH**: Apply partial modifications to a resource.
* **DELETE**: Remove a resource.

1. **Statelessness**: Each request from a client to the server must contain all the information needed to understand the request. The server does not store any client context between requests.
2. **Client-Server Decoupling**: The client and server are independent, allowing them to evolve separately.
3. **Cacheability**: Responses can be cached to improve performance.
4. **Layered System**: Can have intermediaries (e.g., load balancers, proxies) between client and server without affecting their interaction.
5. **Response Formats**: Commonly uses JSON (JavaScript Object Notation), but can also return XML, HTML, or plain text.

**When to use REST:**

1. **Simple APIs with well-defined resources**: When your data model is straightforward, and clients typically need all or most of the data related to a specific resource (e.g., fetching a user profile, a single product).
2. **Caching is a priority**: REST's statelessness and use of HTTP methods make it inherently more cache-friendly at the HTTP level.
3. **Public APIs with diverse consumers**: Because REST uses standard HTTP, it's widely understood and easy for many different clients (web, mobile, third-party integrations) to consume without specialized tooling.
4. **When you need standard HTTP error codes**: REST leverages HTTP status codes (200 OK, 404 Not Found, 500 Internal Server Error, etc.) for indicating the outcome of a request, which is familiar to web developers.
5. **Resource-oriented architectures**: If your system naturally fits into a clear structure of distinct resources that clients interact with.
6. **Microservices architectures**: REST is a common choice for communication between microservices.

**GraphQL**

**GraphQL** is a query language for APIs and a runtime for fulfilling those queries with your existing data. It's not a strict architectural style like REST, but rather a specification that allows clients to precisely define the data they need from the server. It typically operates over a single HTTP endpoint, usually using a POST request.

**Key Characteristics of GraphQL:**

1. **Single Endpoint**: Unlike REST, which has many endpoints for different resources, a GraphQL API usually exposes a single URL endpoint (e.g., /graphql).
2. **Client-Specified Data**: Clients send queries that define the exact data structure they need. The server responds with only that data, eliminating "over-fetching" (getting more data than you need) and "under-fetching" (needing multiple requests to get all the data for a single view).
3. **Strongly-Typed Schema**: A GraphQL API defines a strict schema that describes all possible data types, fields, and relationships available. This schema acts as a contract between client and server, enabling powerful tooling (like auto-completion and validation).
4. **Queries, Mutations, and Subscriptions**:

* **Queries**: For fetching data (read operations).
* **Mutations**: For modifying data (create, update, delete operations).
* **Subscriptions**: For real-time, event-based data updates (server pushes data to client).

1. **Hierarchical**: Queries mirror the structure of the data, making it intuitive to request nested related objects in a single request.
2. **Versionless by Design**: The flexibility of client-specified queries often reduces the need for explicit API versioning (e.g., v1, v2), as new fields can be added to the schema without breaking existing clients. Deprecation can be handled gracefully.

**When to use GraphQL:**

1. **Complex and Interrelated Data Models**: When your data has many relationships, and clients often need to fetch data from multiple "resources" in a single view (e.g., a user, their posts, and comments on those posts).
2. **Mobile Applications**: To minimize network requests and data transfer, which is crucial for mobile devices with limited bandwidth or unstable connections.
3. **Frontend-driven Development**: When frontend teams need more control over data fetching and want to rapidly iterate on UI without constant backend changes. It empowers frontends to ask for exactly what they need.
4. **Aggregating Data from Multiple Sources**: When your backend needs to pull data from various microservices, legacy APIs, or databases, GraphQL can act as a single "API gateway" layer to unify these disparate sources into one coherent graph for the client.
5. **Preventing Over-fetching and Under-fetching**: When optimizing network payloads is a key concern.
6. **Real-time Data Needs**: When your application requires live updates and continuous data streams (e.g., chat applications, live dashboards).
7. **Evolving APIs**: When your API is frequently changing, and you want to add new fields or deprecate old ones without forcing clients to upgrade API versions.

**Summary of Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **REST** | **GraphQL** |
| Data Fetching | Multiple endpoints, over-fetching/under-fetching | Single endpoint, client requests exact data needed |
| Endpoints | Multiple URLs per resource | Typically a single URL (/graphql) |
| Request Type | Uses HTTP verbs (GET, POST, PUT, DELETE) | Primarily POST requests with query/mutation/subscription |
| Schema | Optional, often implicit | Mandatory, strongly-typed (GraphQL Schema Definition Language) |
| Versioning | Often handled by URL versions (e.g., /v1/) | Less need for versioning, handled by deprecation/addition of fields |
| Error Handling | HTTP status codes (4xx, 5xx) | Always 200 OK for valid requests; errors in response body |
| Real-time | Not built-in (requires SSE, WebSockets) | Built-in Subscriptions for real-time updates |

**Object-oriented programming features**

1. **Encapsulation** - Bundling data and methods together.
2. **Abstraction** - Hiding complex details from the user.
3. **Inheritance** - Deriving new classes from existing ones.
4. **Polymorphism** - Same method can have different meanings in different classes.

**Encapsulation**

Encapsulation involves wrapping data (variables) and methods into a single unit (class) and restricting direct access to some of the object's components.

class BankAccount:

def \_\_init\_\_(self):

self.\_\_balance = 0 # private variable

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

**Configuration of microservice**

Use external config files (YAML, JSON, .env). Tools include:

* Spring Cloud Config, Consul, etcd for dynamic configs.
* Docker Secrets or Kubernetes ConfigMaps for secure config handling.

Best practice: Keep configs versioned and separate from code.

**To check performance of monitoring tools**

**Tools**:

* Prometheus + Grafana – Real-time performance and resource monitoring.
* New Relic, Datadog, ELK Stack – Full-stack application performance tracking.

**Metrics to check:**

1. CPU, Memory usage
2. Response time
3. Throughput
4. Error rates
5. Latency

**Code reviews consideration**

Code reviews are a critical process for improving code quality, sharing knowledge, and catching bugs early. When conducting a code review, it's important to have a systematic approach. Considerations include:

**I. Correctness & Functionality**

* Does the code work as intended? This is the most fundamental question. Does it meet the requirements of the feature or bug fix?
* Edge Cases and Error Handling: Does the code gracefully handle unexpected inputs, error conditions, invalid data, or boundary conditions? (e.g., empty lists, null values, division by zero, network failures).
* Concurrency/Race Conditions: If applicable, are there potential issues with multiple threads or processes accessing shared resources? Are locks, mutexes, or atomic operations used correctly?
* Data Integrity: Does the code correctly handle data transformations, storage, and retrieval without corruption or loss?
* Logic Errors: Are there any subtle logical flaws that might lead to incorrect behavior under specific circumstances?

**II. Readability & Maintainability**

* Clarity and Simplicity: Is the code easy to understand at first glance? Is it overly complex or convoluted? Can it be simplified?
* Naming Conventions: Are variable, function, class, and file names clear, descriptive, and consistent with project standards? Do they convey their purpose?
* Code Structure and Organization: Is the code logically grouped? Are functions/methods and classes of reasonable size? Is there unnecessary nesting?
* Comments: Are comments present where necessary to explain why certain decisions were made, complex logic, or non-obvious parts? Are they up-to-date and not redundant?
* Duplication (DRY Principle): Is there repeated code that could be refactored into reusable functions, classes, or modules?
* Modularity and Cohesion: Does each unit (function, class) have a single, well-defined responsibility?
* Coupling: Is the code loosely coupled, minimizing dependencies between different parts of the system?

**III. Performance & Efficiency**

* Algorithm Efficiency: Are efficient algorithms and data structures used for the task? (e.g., avoiding N-squared operations on large datasets).
* Resource Usage: Does the code efficiently use memory, CPU, and I/O resources? Are there potential memory leaks or excessive object creation?
* Database Queries: Are database queries optimized? Are N+1 queries avoided? Are indexes used effectively?
* Network Calls: Are unnecessary network calls avoided? Is data transmission optimized (e.g., payload size)?

**IV. Security**

* Input Validation: Are all external inputs properly validated and sanitized to prevent injection attacks (SQL, XSS, command injection), buffer overflows, or other vulnerabilities?
* Authentication and Authorization: Are access controls correctly implemented? Are sensitive operations protected?
* Sensitive Data Handling: Is sensitive data (passwords, API keys, PII) stored, transmitted, and handled securely (encryption, proper logging)?
* Dependency Vulnerabilities: Are new dependencies introduced free of known vulnerabilities?
* Error Messages: Do error messages reveal too much sensitive information to potential attackers?

**V. Testing**

* Test Coverage: Are new features adequately covered by unit, integration, or end-to-end tests?
* Test Quality: Are the tests meaningful, reliable, and readable? Do they truly test the intended functionality and edge cases?
* Broken Tests: Do the existing tests still pass after the changes? (This should be caught by CI, but worth noting for the review context).

**VI. Design & Architecture**

* Architectural Fit: Does the change fit well within the existing system architecture? Does it introduce new patterns that deviate from the established design without good reason?
* Scalability: Does the change consider future growth or increased load?
* Extensibility: Is the code designed to be easily extended or modified in the future without major refactoring?
* Abstractions: Are appropriate levels of abstraction used? Are they leaky?

**VII. Documentation**

* Code Documentation (Docstrings/XML Comments): Are functions, classes, and complex modules adequately documented with their purpose, parameters, return values, and any side effects?
* READMEs/External Docs: If the change introduces new components or significant architectural shifts, is there updated external documentation (e.g., README files, design documents)?

**VIII. Best Practices & Standards**

* Coding Style (Linters/Formatters): Does the code adhere to the project's established coding style guidelines (e.g., PEP 8 for Python, ESLint for JavaScript)? (Automated tools should catch most of this, but it's a good mental checklist).
* Language Idioms: Does the code use idiomatic constructs for the language, making it more familiar to other developers?
* Dependency Management: Are new dependencies introduced properly managed (e.g., added to package.json, requirements.txt)?

**IX. Change Impact**

* Backward Compatibility: Does the change introduce any breaking changes for existing clients or modules?
* Deployment Considerations: Are there any special steps required for deploying this code? Does it require database migrations, environment variable changes, or infrastructure updates?
* Performance Impact: What is the expected performance impact on the system?
* Logging & Monitoring: Is appropriate logging implemented for debugging and monitoring in production? Are new metrics needed?

**X. Reviewer's Role & Etiquette**

* Be Constructive and Respectful: Focus on the code, not the person. Offer suggestions rather than demands.
* Explain "Why": When suggesting changes, explain the reasoning behind them (e.g., "This might cause a memory leak because...", "A dict would be more efficient here for X reason.").
* Prioritize Comments: Distinguish between critical issues (must fix) and suggestions (nice to have).
* Don't Be Overly Nitpicky: Focus on important issues. Minor style preferences that don't affect correctness or readability significantly can be overlooked or addressed by automated formatters.
* Balance Speed and Thoroughness: Don't let reviews become a bottleneck, but don't rush through them either.
* Learn from the Code: Use the review as an opportunity to understand new parts of the codebase and learn from the author.

**Framework for Server-side rendering**

When it comes to Server-Side Rendering (SSR), the frameworks you choose are heavily dependent on the programming language and ecosystem you're working with. SSR essentially means that your web server processes the initial HTML, CSS, and potentially some JavaScript, and sends a fully rendered page to the client's browser, rather than just an empty HTML file that JavaScript then populates.

**I. JavaScript / Node.js Ecosystem (Most Prominent for Modern SSR)**

This ecosystem has seen the most innovation and adoption for SSR, especially with the rise of single-page applications (SPAs) that wanted the benefits of SEO and faster initial load times.

**i. Next.js (React Framework):** A popular open-source React framework that enables server-side rendering, static site generation, and client-side rendering with ease. It's often referred to as a "full-stack framework" for React. When to Use:

* Ideal for complex React applications that need excellent SEO, fast initial page loads, and dynamic content without compromising on the modern development experience of React. It's a leading choice for blogs, e-commerce, and dashboards.

**ii. Nuxt.js (Vue.js Framework)**: The equivalent of Next.js but for the Vue.js ecosystem. It's an intuitive framework for building performant Vue.js applications. When to Use:

* Best for Vue.js developers looking for similar benefits to Next.js: improved SEO, faster time-to-content, and a structured approach to large-scale Vue applications.

**iii. SvelteKit (Svelte Framework)**: The official application framework for Svelte, a reactive JavaScript compiler. Svelte is known for compiling components into tiny, highly efficient JavaScript. When to Use:

* For developers who prefer Svelte's "no runtime" approach and want a framework that handles SSR and routing for them, leading to very fast and lightweight applications.

**II. Traditional Server-Side Frameworks (SSR is their Native Mode)**

These frameworks have been doing server-side rendering long before JavaScript frameworks popularized it. They primarily render HTML on the server using templating engines.

**i. Django (Python):** A high-level Python web framework that encourages rapid development and clean, pragmatic design.When to Use:

* Full-stack Python development, especially for database-driven applications, CMSs, or anything requiring a robust backend coupled with server-rendered pages.

**ii. Ruby on Rails (Ruby)**: A full-stack web application framework written in Ruby, known for its "convention over configuration" philosophy. When to Use:

* Rapid prototyping and building full-stack web applications where productivity and convention are highly valued.

**iii. Laravel (PHP)**: A popular PHP web application framework with expressive, elegant syntax. When to Use:

* Building robust PHP-based web applications, APIs, and CMSs where server-side rendering is the primary rendering strategy.

**iv. Spring Boot (Java)**: A framework that simplifies the development of production-ready, stand-alone, Spring-based applications. When to Use:

* Enterprise-level Java applications, particularly those requiring strong backend capabilities, robust security, and traditional server-rendered views.

**v. ASP.NET Core (C# / .NET)**: A cross-platform, high-performance, open-source framework for building modern, cloud-enabled, internet-connected applications. When to Use:

* Building robust web applications within the .NET ecosystem, especially for Windows environments or cloud-native applications on Azure.

**CSS box shadow**

The box-shadow CSS property allows you to add shadow effects around an element's frame. It's a powerful tool for creating visual depth, emphasis, and a sense of three-dimensionality on web pages. You can apply multiple shadows to a single element, separated by commas.

box-shadow: [inset] h-offset v-offset [blur-radius] [spread-radius] [color];

**Increase space between border and its content in element**

Use the CSS padding property.

selector {

padding: 20px; /\* 20 pixels of padding on all sides \*/

}

**Tool for testing Restful API**

API Testing clients (Manual & Exploratory testing) using

**i. Postman**: Used for manual API testing, API development and debugging, team collaboration on API specifications, basic automated tests.

**ii. Insomnia**: Used for API development, debugging, and testing, especially for developers who prefer a clean, focused interface and integration with their code editor.

**iii. Thunder Client (VS Code Extension)**: Used for Quick API testing and debugging without leaving your code editor, ideal for developers working with VS Code.

**iv. cURL**: used for Quick command-line testing, scripting automated tests in shell scripts, fundamental debugging. Excellent for understanding the raw HTTP request/response.

**v. HTTPie**: Similar to cURL but with a focus on readability and ease of use for command-line interactions.

**SQL and NoSQL**

SQL (Structured Query Language) and NoSQL (Not Only SQL) refer to two fundamentally different approaches to database management. They each have distinct characteristics, strengths, and weaknesses, making them suitable for different types of applications and data.

**SQL (Relational Databases)**

SQL databases are relational databases that organize data into predefined, structured tables. Each table consists of rows and columns, similar to a spreadsheet. Relationships between tables are established using foreign keys. SQL is the standard language used to define, manipulate, and query data in these databases. Examples of SQL Databases:

* MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, SQLite, MariaDB

**Key Characteristics of SQL Databases**

1. **Schema-based**: Requires a predefined schema (structure) for the data. All data must conform to this schema, meaning you define columns, their data types, and relationships upfront. Changes to the schema can be complex and may require downtime.
2. **Table-based**: Data is stored in tables with rows and columns.
3. **ACID Compliance**: Most SQL databases strictly adhere to ACID properties (Atomicity, Consistency, Isolation, Durability). This guarantees that database transactions are processed reliably, maintaining data integrity even during system failures.
   * Atomicity: All operations in a transaction either succeed or fail as a whole.
   * Consistency: A transaction brings the database from one valid state to another.
   * Isolation: Concurrent transactions execute independently without interfering with each other.
   * Durability: Once a transaction is committed, it remains committed even in the event of power loss or system failures.
4. **Vertical Scalability**: Traditionally scales by increasing the resources (CPU, RAM, SSD) of a single server. While horizontal scaling (sharding) is possible, it's often more complex to implement compared to NoSQL.
5. **Powerful Query Language**: SQL is a mature and widely adopted query language, excellent for complex queries, joins across multiple tables, and aggregations.
6. **Mature Ecosystem**: Has been around for decades, boasting a vast ecosystem of tools, support, and experienced developers.

**NoSQL (Not Only SQL / Non-Relational Databases)**

NoSQL databases are non-relational databases that provide a mechanism for storing and retrieving data that is modeled in means other than the tabular relations used in relational databases. They are designed to handle large volumes of unstructured, semi-structured, and rapidly changing data. The term "NoSQL" often means "Not only SQL" because some NoSQL databases do offer SQL-like query capabilities. Examples of NoSQL Databases:

* MongoDB (Document), Apache Cassandra (Wide-Column), Redis (Key-Value/In-memory data structure store), Amazon DynamoDB (Key-Value, Document), Neo4j (Graph), Apache CouchDB (Document), Elasticsearch (Document, Search Engine)

**Key Characteristics**:

1. **Flexible Schema**: Do not require a predefined schema. Data can be stored in various formats (JSON documents, key-value pairs, graphs, wide-columns) and documents within the same collection can have different fields. This makes it easier to iterate quickly on evolving data models.
2. **Diverse Data Models**: Instead of tables, NoSQL databases come in different types:

* **Document Databases**: Store data in flexible, semi-structured "documents" (often JSON, BSON, or XML). Ideal for content management, user profiles, catalogs. (e.g., MongoDB, CouchDB)
* **Key-Value Stores**: The simplest form, storing data as a collection of key-value pairs. Very fast for read/write operations. Ideal for caching, session management, user preferences. (e.g., Redis, Amazon DynamoDB, Memcached)
* **Wide-Column Stores (Column-Family)**: Store data in columns rather than rows, allowing for flexible column definitions per row. Optimized for large datasets and analytical queries over specific columns. (e.g., Apache Cassandra, HBase)
* **Graph Databases**: Store data in nodes and edges, focusing on relationships between data entities. Ideal for social networks, recommendation engines, fraud detection. (e.g., Neo4j, Amazon Neptune)

1. **Horizontal Scalability (Scale-out)**: Designed to distribute data across multiple servers (sharding) to handle massive amounts of data and high traffic loads. This is generally more cost-effective and easier to achieve than vertical scaling for very large datasets.
2. **BASE Properties (Often)**: Many NoSQL databases follow the BASE consistency model (Basically Available, Soft state, Eventually consistent) rather than strict ACID. This prioritizes availability and partition tolerance over immediate consistency.
3. **Basically Available**: The system is always available for queries.
4. **Soft stat**e: The state of the system may change over time, even without input.
5. **Eventual consistency**: Data will eventually be consistent across all nodes, but there might be a delay. (Some NoSQL databases now offer strong consistency options).
6. **Less Mature Ecosystem**: Newer technology, so the ecosystem of tools and established practices is still evolving compared to SQL.
7. **Variable Query Languages**: Query languages vary widely by database type (e.g., MongoDB Query Language, Cassandra Query Language, Gremlin for graphs), lacking a single universal standard like SQL.

**Unit testing**

Unit Testing focuses on verifying the smallest testable parts of an application, known as "units," in isolation from the rest of the codebase. A unit is typically a function, method, or class, and unit tests ensure that each of these individual components works correctly according to its specifications, without relying on external dependencies like databases, networks, or file systems, which are usually mocked or stubbed out.

**Integration testing**

Integration Testing focuses on verifying that different units or components of an application work correctly when combined and interact with each other, as well as with external systems like databases, APIs, or third-party services. Unlike unit tests, integration tests assess the interfaces and interactions between these components, ensuring that data flows correctly and that the combined parts achieve their intended functionality within a larger system.

**Recursion**

Recursion is a programming technique where a function calls itself, directly or indirectly, to solve a problem. It works by breaking down a complex problem into smaller, identical sub-problems until it reaches a simple "base case" that can be solved directly. Once the base case is solved, the results from each sub-problem are combined as the function calls return, eventually leading to the solution of the original, larger problem. This approach is often used for tasks that can be naturally divided into self-similar parts, such as traversing tree structures, calculating factorials, or generating sequences.

**Truth about function**

The truth about a function in programming is that it's a fundamental building block designed to encapsulate a specific, reusable block of code that performs a single, well-defined task. Functions promote modularity by allowing developers to break down complex programs into smaller, manageable, and independent units. They accept zero or more inputs (arguments), perform operations, and often produce an output (return value), effectively acting as a mapping from input to output. This encapsulation simplifies debugging, improves code readability, and enables code reuse across different parts of an application or even in entirely different projects, thereby increasing efficiency and reducing redundancy.

**State management in js**

State management in JavaScript refers to the practice of handling and organizing the dynamic data (the "state") that determines what is displayed in a user interface, especially in complex client-side applications. As applications grow, data can become scattered across many components, making it difficult to track changes, ensure consistency, and debug issues. State management solutions, such as Redux, Vuex, Zustand, or React's Context API, provide a centralized and predictable way to store, update, and access this shared application data, ensuring that all parts of the UI reflect the current state consistently and reactively.

**Models.py in django**

models.py in Django is a crucial file within a Django application where you define your database schema using Python classes. Django's Object-Relational Mapper (ORM) translates these Python classes into database tables, with each class representing a table and each attribute within the class representing a column. This allows developers to interact with the database using familiar Python objects and methods (e.g., User.objects.create(...), Product.objects.filter(...)) instead of writing raw SQL, greatly simplifying database interactions and making the code more readable and maintainable.

**Function of query parameter**

Query parameters are key-value pairs appended to a URL after a question mark (?), separated by ampersands (&), used to send small pieces of data from the client to the server when making a request. Their primary function is to filter, sort, paginate, or otherwise modify the data returned by the server for a specific resource, or to pass specific instructions to the server without changing the resource's fundamental path. For instance, in www.example.com/products?category=electronics&sort=price\_asc, category and sort are query parameters instructing the server to return products filtered by "electronics" and sorted by "price ascending."

**Single responsibility principle (SRP)**

The Single Responsibility Principle, often applied in software design, states that a module, class, or even a function should have one, and only one, reason to change. This means that each unit of code should be responsible for just one part of the application's functionality, encapsulating that responsibility entirely. Adhering to SRP leads to more robust, maintainable, and testable code because changes to one responsibility won't inadvertently affect other unrelated functionalities, making the system easier to understand, modify, and extend.

**Output of console.log(this)**

The output of console.log(this) in JavaScript is highly dependent on the execution context in which it is called.

* In the global scope (e.g., directly in a script in a browser), this refers to the global object, which is window in browsers or global in Node.js.
* Inside a regular function in non-strict mode, this also refers to the global object. In strict mode, this will be undefined.
* Inside an object method, this refers to the object itself that the method belongs to.
* Inside a constructor function when used with new, this refers to the newly created instance of the object.
* Inside an event handler, this typically refers to the DOM element that triggered the event.
* Arrow functions behave differently; they do not have their own this context and instead inherit this from their lexical (enclosing) scope.

**Does normalization deal with combining or splitting tables?**

Database normalization primarily deals with splitting tables to reduce data redundancy and improve data integrity. It's a systematic process of organizing the columns and tables in a relational database to ensure that data dependencies are properly enforced and that information is stored logically. By breaking down large, complex tables into smaller, more manageable ones, and defining relationships between them using keys, normalization minimizes duplicate data, prevents inconsistencies during updates, insertions, or deletions, and makes the database more flexible and efficient for future modifications.

**Embedded systems**

An embedded system is a specialized computer system - a combination of a computer processor, computer memory, and input/output peripheral devices - that is integrated into a larger mechanical or electronic system to perform a dedicated, specific function. Unlike general-purpose computers, embedded systems are typically designed for real-time computing constraints, often have no direct user interface (or a very simple one), and run firmware specific to their task. Examples range from simple devices like a microwave oven controller or a digital watch, to complex systems such as vehicle engine control units, industrial robots, or medical equipment.

**Real-time OS (RTOS)**

A Real-time Operating System (RTOS) is a specialized operating system designed for real-time computing applications that process data and events with critically defined time constraints, ensuring predictability and stability. Unlike general-purpose operating systems (like Windows or Linux) that prioritize high throughput, an RTOS prioritizes deterministic behavior, guaranteeing that tasks are executed within precise, fixed time frames, often within microseconds or milliseconds. This predictability is crucial for systems where delays could lead to failure, safety hazards, or significant performance degradation, such as in industrial control systems, avionics, medical devices, or automotive systems.

**Decrease load in microservice**

To decrease load in a microservices architecture, several strategies can be employed. This often involves a combination of **scaling** (horizontally scaling out instances of services that are under heavy load), **caching** (storing frequently accessed data closer to the consumer or within the service itself to reduce database or downstream service calls), **optimizing code and database queries** within individual services, implementing **load balancing** to distribute incoming requests evenly across service instances, and utilizing **asynchronous communication patterns** (like message queues) to decouple services and prevent cascading failures or bottlenecks during peak times. These approaches aim to reduce the processing burden on individual service instances and distribute the workload efficiently across the system.

**Write once, run anywhere or WORA (choose either React Native, Flutter, Ionic, Xamarin)**

“Write Once, Run Anywhere" (WORA) is a key philosophy and characteristic primarily associated with the Java programming language. It means that Java code, once compiled into bytecode (a platform-independent intermediate language), can be executed on any device or operating system that has a compatible Java Virtual Machine (JVM) installed, without needing to be recompiled for each specific platform. This portability significantly simplifies software deployment and maintenance, as developers can write their applications on one system and be confident they will run consistently across diverse computing environments.

**API versioning**

API versioning is the practice of managing changes to an Application Programming Interface (API) by introducing distinct versions, ensuring that existing client applications continue to function correctly even as the API evolves. This is crucial because altering an API (e.g., changing endpoints, removing fields, modifying data types) without proper versioning can break applications that rely on the older structure. Common versioning strategies include embedding the version number in the URL (e.g., /api/v1/users), using custom HTTP headers, or employing query parameters, allowing developers to introduce new features or changes while providing a stable interface for consumers still using previous versions.

**Function of API gateway in microservice**

In a microservices architecture, an API Gateway acts as a single, intelligent entry point for all client requests, abstracting the complexity of the underlying microservices. Its primary functions include routing requests to the appropriate backend services, handling cross-cutting concerns such as authentication, authorization, rate limiting, and caching, and performing request/response transformation. By centralizing these concerns, an API Gateway simplifies client applications, enhances security, improves performance, and enables independent development and deployment of individual microservices, preventing clients from needing to directly interact with multiple backend services.

**Decrease while/for loop workload**

To decrease the workload of while or for loops, which directly translates to improving performance, focus on reducing the number of iterations and minimizing the operations performed inside the loop body. Strategies include:

1. **Reducing iterations**: Ensure the loop iterates only as many times as strictly necessary.
2. **Optimizing operations within the loop**: Move any computations or expensive function calls that produce the same result in every iteration outside the loop (hoisting).
3. **Using efficient data structures and algorithms**: Choose data structures that provide faster access or manipulation for your specific use case (e.g., hash maps over arrays for lookups).
4. **Avoiding unnecessary work**: Don't perform calculations or allocate resources inside the loop if they aren't directly contributing to the current iteration's result.
5. **Breaking early**: If the desired condition is met, use break to exit the loop.

**Output of 10 + 20 + “30”**

The output of 10 + 20 + "30" in JavaScript is "3030". This occurs due to JavaScript's type coercion rules for the + operator. When the + operator encounters a string operand, it performs string concatenation rather than numerical addition. In this expression, 10 + 20 is evaluated first because it involves only numbers, resulting in 30. Then, 30 + "30" is evaluated; since "30" is a string, the numerical 30 is coerced into a string "30", and the two strings are concatenated, yielding "3030".

**Try catch and final**

try-catch-finally is a fundamental error handling construct used in many programming languages to manage exceptions (runtime errors) gracefully. The try block encloses the code that might potentially throw an error. If an error occurs within the try block, execution immediately jumps to the catch block, which contains code to handle or recover from the specific error. Regardless of whether an error occurred or was caught, the finally block always executes after the try and catch blocks (or after a return statement within them), making it ideal for cleanup operations like closing files, releasing resources, or terminating connections, ensuring these actions happen reliably.

**Modularity**

Modularity in programming is a design principle that involves breaking down a software system into smaller, independent, and interchangeable components or "modules," each responsible for a distinct piece of functionality. Each module is designed to perform a specific task and expose a clear interface for interacting with other modules, while hiding its internal implementation details. This approach enhances maintainability, readability, reusability, and testability of the code, as changes within one module are less likely to impact others, and individual components can be developed and tested in isolation before being integrated into the larger system.

**Can a table have multiple primary keys?**

No, a database table in a relational database can only have one primary key (PK). The **primary key** is a column or a set of columns that uniquely identifies each row in the table, ensuring data integrity and uniqueness. While a primary key can be composed of multiple columns (known as a composite primary key), the table as a whole still has only one designated primary key constraint. A table can, however, have multiple unique keys or unique constraints, which also enforce uniqueness for their respective columns but are not the table's primary identifier.

**Upgrade vs Update**

he terms "upgrade" and "update" in software refer to different types of changes. An **update** is typically a smaller, incremental release designed to fix bugs, patch security vulnerabilities, improve performance, or add minor enhancements to the current version of the software. It usually doesn't involve significant changes to the core functionality or user interface. An **upgrade**, on the other hand, represents a more substantial change, often involving a new major version of the software, introducing significant new features, a redesigned interface, or fundamental architectural changes. Upgrades may require a more involved installation process and sometimes come with a cost.

**Tools for real-time collaboration in designing**

For real-time collaboration in web and UI/UX design, tools that allow multiple users to work simultaneously on the same design file and see changes instantly are essential. The most prominent example is **Figma**, which is a cloud-based design tool enabling designers to create, prototype, and collaborate in real-time within a single browser-based environment, complete with commenting and version history. Other tools like Miro (for collaborative whiteboarding, brainstorming, and flowcharts), **Adobe XD** (with co-editing features), and **Sketch** (though historically desktop-first, it offers cloud-based collaboration for shared libraries and prototypes) also facilitate real-time or near real-time collaborative workflows for various stages of the design process.

**Function of JS in web development**

JavaScript (JS) plays a multifaceted and indispensable role in modern web development, primarily functioning to bring interactivity, dynamism, and responsiveness to web pages that HTML and CSS alone cannot provide. On the client-side, JS enables dynamic content updates, manipulates the Document Object Model (DOM) to change page structure and styles, handles user interactions (like clicks, form submissions, and keyboard inputs), fetches data asynchronously from servers (AJAX), creates animations, and enables complex client-side application logic. With Node.js, JavaScript's role extends to the server-side, allowing developers to build full-stack applications using a single language for both client and server, facilitating real-time communication, API creation, and database interactions.

**Programming Qns**

**Answer: False (C)**

int x = 10;

if (x = 0)

printf("True");

else

printf("False");

In C, 0 is treated as False in a boolean context, while any non-zero value is True. Therefore, the if (0) condition is false.

**Answer: True (Python)**

print(bool("False"))

In Python, the bool() function returns True for any non-empty string and False only for an empty string (""). The string "False" is a non-empty string.

**Answer: 10 (PHP)**

echo 5 + "5 apples";

PHP has loose typing and attempts to convert strings to numbers in arithmetic operations. When "5 apples" is used in an addition, PHP tries to convert it to a number. It successfully converts the leading "5" into the integer 5 and then stops when it encounters non-numeric characters ( " apples"). So, 5 + 5 is performed.

**Answer: 2 (C++ and C)**

int x = 2, y = 3;

cout << (x & y);

The & operator in C++ (and C) is the bitwise AND operator. It performs a logical AND operation on each corresponding bit of its operands.

x = 2 in binary is 0010 and y = 3 in binary is 0011

Bitwise AND (0010 & 0011):

* 0 & 0 = 0
* 0 & 0 = 0
* 1 & 1 = 1
* 0 & 1 = 0

The result is 0010 in binary, which is 2 in decimal.

**Microservice**

Microservices is an architectural style that structures an application as a collection of small, independent, and loosely coupled services, each designed to perform a single business capability. Unlike traditional monolithic applications where all components are tightly integrated into a single deployable unit, microservices communicate with each other through lightweight mechanisms, often using APIs (like REST or gRPC). This approach allows teams to develop, deploy, and scale each service independently, using different technologies if needed, leading to increased agility, resilience, and flexibility in managing complex systems.

**Types of Microservices**

While there isn't a universally agreed-upon formal categorization of "types" of microservices, they are often **classified by the patterns** they implement or the business capabilities they represent. Common patterns include **API Gateway** (a single entry point for clients), Service Discovery (how services find each other), **Database per Service** (each microservice owns its data), **Event-Driven Architecture** (services communicate via asynchronous events), and **Aggregator** (combining responses from multiple services). Functionally, they are typically divided by business domains, such as an "Order Service," "User Service," or "Payment Service," each responsible for a specific slice of the overall business logic.

**Tools for microservices**

Developing, deploying, and managing microservices requires a robust set of tools across various categories. For **containerization**, **Docker** is essential for packaging services, while **Kubernetes** is the leading platform for orchestrating and managing these containers at scale. **API Gateways** like Kong or AWS API Gateway handle routing and cross-cutting concerns. **Service Mesh** tools such as Istio provide advanced traffic management and observability. For **monitoring and logging**, Prometheus, Grafana, and ELK stack (Elasticsearch, Logstash, Kibana) are widely used, alongside distributed tracing tools like Jaeger or Zipkin. Finally, **CI/CD pipelines** (e.g., Jenkins, GitLab CI, GitHub Actions) automate the build, test, and deployment processes for independent services.

**Microservice architecture**

Microservice architecture is an approach to designing software where an application is composed of small, independent services, each running in its own process and communicating via lightweight mechanisms. This architectural style emphasizes decentralization, with each service owning its data and being responsible for a specific, focused business function. It contrasts with monolithic architectures by promoting autonomous teams, independent deployment, technological diversity (polyglot persistence and programming), and enhanced fault isolation. While offering benefits in scalability, resilience, and agility, it also introduces complexities in terms of distributed data management, inter-service communication, monitoring, and overall operational overhead.

**Web services**

Web services are standardized ways for different applications to communicate and exchange data over a network, typically the internet. They enable interoperability between diverse systems, regardless of the programming language or platform they are built on, by using common web protocols like HTTP and data formats like XML or JSON. Essentially, web services allow machines to interact with other machines, providing a programmatic interface to expose business logic or data, facilitating seamless integration between disparate software components. This allows for functionalities like a mobile app fetching real-time weather data or an e-commerce site integrating with a payment gateway.

**Types of Web services**

While the term "web services" can broadly refer to any web-based communication between applications, the two most prominent and widely recognized types are **SOAP (Simple Object Access Protocol) services** and **RESTful services** (based on Representational State Transfer). SOAP services are protocol-based, relying on XML for message formatting and often utilizing WSDL (Web Services Description Language) for service description, providing strict contracts and robust features suitable for enterprise-level integrations. RESTful services, on the other hand, are an architectural style that leverages standard HTTP methods (GET, POST, PUT, DELETE) to interact with resources identified by URLs, typically using lighter data formats like JSON, making them more flexible, scalable, and widely adopted for modern web APIs.

**REST (Representational State Transfer)** is an architectural style for designing networked applications, leveraging standard HTTP methods (like GET, POST, PUT, DELETE) to interact with resources identified by URLs. It is stateless, meaning each request from a client to the server contains all the necessary information, and the server doesn't store any client context between requests. REST typically uses lightweight data formats like JSON, making it flexible, highly scalable, and widely adopted for modern web APIs due to its simplicity and efficiency.

**SOAP (Simple Object Access Protocol)**, on the other hand, is a protocol specification for exchanging structured information in the implementation of web services. Unlike REST, SOAP is highly standardized and protocol-agnostic, meaning it can run over various protocols (though most commonly HTTP), and relies on XML for all message formatting. SOAP provides a strict contract between client and server, offering built-in features for security (WS-Security), reliability, and transaction management, making it suitable for enterprise-level applications requiring high levels of security, formal contracts, and guaranteed delivery.

**Software running slow, give 3 possible causes and 3 solutions**

When software is running slow, it can be a frustrating experience with several potential culprits. Here are three common causes and their corresponding solutions:

**Possible Causes:**

1. **Insufficient System Resources**: The software might be performing slowly because the computer or server it's running on simply doesn't have enough CPU power, RAM, or fast enough storage (like an old HDD versus an SSD). If the application is constantly waiting for resources, it will feel sluggish, especially under heavy workloads or when multiple applications are running simultaneously.
2. **Inefficient Code or Algorithms**: The slowness might stem from the software's internal design, meaning the code itself is not optimized for the tasks it performs. This could involve using inefficient algorithms that take too long to process large datasets, making excessive or redundant calculations, or poorly managing memory, leading to unnecessary processing cycles and delays.
3. **Network Latency or Bandwidth Issues**: For applications that communicate over a network (e.g., cloud-based software, web applications, or microservices), slow performance can often be attributed to high network latency (delays in data transmission) or insufficient bandwidth. If data takes too long to travel between the client and server, or between different parts of a distributed system, the entire application will feel unresponsive regardless of individual component speed.

**Solutions**:

1. **Upgrade Hardware or Allocate More Resources**: To address resource limitations, consider upgrading the physical hardware (e.g., installing more RAM, switching to a Solid State Drive, or adding a faster CPU). If the software is running on a virtual machine or cloud instance, allocate more virtual CPUs, memory, or provision higher-performance storage to provide the necessary computing power.
2. **Perform Code Profiling and Optimization**: Use specialized profiling tools to identify exact bottlenecks within the software's code. Once identified, refactor inefficient algorithms, optimize database queries, implement caching mechanisms for frequently accessed data, or redesign problematic sections to reduce computational load and improve execution speed.
3. **Analyze and Optimize Network Communication**: Troubleshoot network connectivity to rule out physical issues or congestion. For application-level optimization, reduce the number of network requests (e.g., batching calls), minimize the size of data payloads (e.g., using compression or sending only necessary fields), or implement robust caching strategies at the client or API Gateway level to reduce reliance on constant server communication.

**Steps to develop a system**

Developing a software system typically follows a structured process known as the System Development Life Cycle (SDLC), which can be adapted to various methodologies like Waterfall, Agile, or DevOps. Here are the common steps involved:

1. **Requirements Gathering and Analysis**: This initial phase involves understanding the problem or opportunity the system aims to address. It requires thorough communication with stakeholders to identify and document their needs, functionalities, performance expectations, security requirements, and user experience goals. The output is typically a detailed requirements specification document, outlining what the system should do.
2. **System Design**: In this phase, the detailed architecture and components of the system are planned based on the gathered requirements. This includes designing the system's overall structure, user interface (UI/UX), database schema, network architecture, modules, and algorithms. High-level design focuses on the big picture, while detailed design specifies the low-level components and their interactions.
3. **Implementation/Development**: This is where the actual coding takes place. Developers write the software code according to the design specifications, using chosen programming languages, frameworks, and tools. This phase often involves breaking down the system into smaller, manageable units or features that are developed iteratively, especially in agile methodologies.
4. **Testing**: Once the code is written, it undergoes rigorous testing to identify and fix defects, ensure it meets the specified requirements, and verify its stability and performance. This includes various types of testing such as unit testing (individual components), integration testing (interactions between components), system testing (entire system functionality), user acceptance testing (UAT - by end-users), and performance testing.
5. **Deployment**: After successful testing, the system is deployed into its production environment, making it available to end-users. This involves setting up the necessary infrastructure, installing the software, configuring databases, and ensuring all dependencies are in place. The deployment process can range from simple installations to complex, automated continuous deployment pipelines.
6. **Maintenance and Monitoring**: The final, ongoing phase involves supporting the system in production. This includes monitoring its performance, addressing bugs or issues that arise, applying security patches, and implementing enhancements or new features based on evolving user needs or business requirements. This continuous feedback loop ensures the system remains relevant, stable, and effective over its lifespan.

**Tell us a project you work on your role, challenges you faced and how you overcome it**

The best way to answer it is to use the STAR method (Situation, Task, Action, Result). Here's how to structure your answer effectively:

**1. Choose the Right Project (The "Situation" & "Task")**

* Select a Relevant Project: Pick a project that is pertinent to the job you're interviewing for and highlights skills the employer is looking for. It doesn't have to be a massive, company-wide initiative; a smaller, impactful project where you played a significant role is often better than a large one where your contribution was minor.
* Ensure it has a Clear Challenge & Resolution: The project must have a distinct challenge you genuinely faced and successfully overcame. Avoid projects that went perfectly smoothly.
* Briefly Set the Stage: Start by concisely describing the project itself. What was its goal? What was the context? (e.g., "During my time at [Previous Company/University], I worked on a project to develop a new [type of system/feature] aimed at [achieving a specific business objective, e.g., improving customer engagement].")

**2. Define Your Role (Your Specific Contribution)**

* Focus on "I," Not "We": While teamwork is great, the interviewer wants to understand your individual contribution. Clearly state your responsibilities.
* Be Specific: (e.g., "My role was the lead backend developer, responsible for designing and implementing the API for data processing," or "I was the sole QA engineer tasked with establishing the testing framework and test cases for this module.")
* Connect to Impact: Briefly explain how your role was critical to the project's success.

**3. Detail the Challenges Faced (The "Action" Trigger)**

* Be Specific about the Problem: Don't just say "we had problems." Describe the exact challenge. Was it a technical hurdle (e.g., "We faced significant performance bottlenecks when scaling our database queries to handle large user traffic")? A communication breakdown? A tight deadline? An unexpected scope change?
* Explain the Impact: Briefly describe why this was a challenge and what its potential negative consequences were if not addressed (e.g., "This threatened to delay our launch by two months and risked a poor user experience").
* Focus on Your Discovery/Identification: (e.g., "I identified that our existing data indexing strategy was insufficient for the projected user load...")

**4. Explain How You Overcame It (The Core "Action" Part)**

* Your Actions: This is the most crucial part. Detail the concrete steps you took to address the challenge.
* Problem-Solving Process: Did you research? Experiment? Consult others? (e.g., "I researched alternative database indexing strategies, prototyping three different approaches.")
* Tools/Techniques Used: Mention specific technologies, methodologies, or analytical skills you applied. (e.g., "Using a profiling tool, I pinpointed the exact inefficient queries, then implemented a new caching layer with Redis.")
* Collaboration: If it was a team effort, acknowledge it but still focus on your contribution. (e.g., "I then collaborated with the frontend team to integrate the new API endpoints seamlessly...")
* Decision-Making: If you had options, explain why you chose a particular path.
* Show Initiative & Persistence: Demonstrate that you didn't give up and actively sought solutions.

**5. State the Outcome/Result (The "Result" & Learning)**

* Quantify the Impact: Whenever possible, use numbers to describe the positive outcome. (e.g., "As a result, we reduced the API response time by 60% under load," or "We successfully launched the feature on schedule and saw a 15% increase in user engagement.")
* Learning Points: Briefly reflect on what you learned from the experience, whether it's a technical skill, a teamwork lesson, or a new approach to problem-solving. (e.g., "This experience taught me the critical importance of early performance testing in a microservices environment," or "I learned how effective cross-functional communication can prevent major roadblocks.")

**Cope with completing a project and meeting deadline (mention 5)**

Coping with the pressure of completing a project and meeting deadlines is a common challenge. Here are five effective strategies to manage the workload and stress:

**1. Prioritize and Break Down Tasks:**

* **Strategy**: Don't look at the entire project as one daunting task. Break it down into smaller, manageable sub-tasks. Then, prioritize these tasks based on urgency and importance. Use techniques like the Eisenhower Matrix (Urgent/Important) or simply label tasks as high, medium, or low priority.
* **Benefit**: This approach makes the project seem less overwhelming, provides clear next steps, and ensures that the most critical components are addressed first. Checking off smaller tasks also provides a sense of progress and motivation.

**2. Effective Time Management and Scheduling:**

* **Strategy**: Create a realistic schedule or timeline for yourself, allocating specific blocks of time for different tasks. Use tools like calendars, to-do lists, or project management software. Include buffer time for unexpected issues. Practice techniques like the Pomodoro Technique (focused work sprints with breaks) to maintain concentration.
* **Benefit**: A well-structured schedule helps you stay organized, avoids procrastination, and ensures that all aspects of the project receive adequate attention, reducing last-minute rush and panic.

**3. Communicate Proactively and Manage Expectations:**

* **Strategy**: Maintain open and honest communication with your team, stakeholders, and clients. If you anticipate potential delays or encounter unforeseen obstacles, communicate them early. Don't wait until the last minute. Propose solutions or adjusted timelines if necessary.
* **Benefit**: Proactive communication builds trust, allows for adjustments to expectations, and gives others a chance to assist or adapt, preventing misunderstandings and negative surprises at the deadline.

**4. Manage Stress and Take Regular Breaks:**

* **Strategy**: The pressure of deadlines can lead to burnout. Incorporate short, regular breaks into your workday to step away from the screen, stretch, or do something relaxing. Ensure you get enough sleep, eat healthy, and engage in physical activity. Avoid excessive caffeine or energy drinks, which can lead to crashes.
* **Benefit**: Taking breaks and managing stress are crucial for maintaining focus, creativity, and overall well-being. A rested mind is more efficient and less prone to errors, ultimately helping you work smarter, not just harder.

**5. Learn to Delegate or Ask for Help:**

* **Strategy**: If you're feeling overwhelmed and genuinely unable to meet the deadline alone, don't hesitate to delegate tasks if you're in a leadership position, or openly ask colleagues, mentors, or supervisors for help or guidance. Be specific about what assistance you need.
* **Benefit**: Recognizing when you need support is a sign of strength, not weakness. Delegation or seeking help can redistribute the workload, provide fresh perspectives, and prevent individual burnout, ensuring the project's success without sacrificing your well-being

**Challenges during integration**

Integration, particularly in software development, is the phase where independently developed components, modules, or services are combined and made to work together as a cohesive system. This process often uncovers a unique set of challenges:

1. **Interface Mismatches and Communication Protocol Issues**: Different components, especially if developed by separate teams or using diverse technologies, may have conflicting expectations regarding data formats, API endpoints, communication protocols (e.g., REST vs. gRPC), or even the interpretation of specific fields. This leads to errors where one component sends data in a format the other doesn't understand, or attempts to call an endpoint that doesn't exist.
2. **Data Inconsistencies and Synchronization**: When multiple services or modules manage their own data stores, ensuring data consistency and synchronization across the entire integrated system becomes complex. Issues like stale data, partial updates, or conflicting records can arise, especially in distributed transactions, leading to integrity problems that are difficult to debug.
3. **Dependency Hell and Version Conflicts**: As components are integrated, managing their various internal and external dependencies can become a nightmare. Different components might rely on different versions of the same library, framework, or even operating system components, leading to "dependency hell" where conflicts prevent successful compilation or runtime execution.
4. **Performance Bottlenecks and Scalability Degradation**: While individual components might perform well in isolation, their combined interactions can introduce unforeseen performance bottlenecks. This could be due to excessive inter-service communication overhead, inefficient data transfer, or one component becoming a choke point for the entire integrated system, hindering overall scalability.
5. **Complex Debugging and Error Tracing**: Identifying the root cause of an error in an integrated system is significantly harder than in a monolithic application. An error might originate in one component, propagate through several others, and manifest in a completely different part of the system. Tracing the flow of execution and data across multiple services or modules without robust logging, monitoring, and distributed tracing tools becomes a major challenge.

**Docker hub and benefits (mention 5)**

**Docker Hub** is a cloud-based registry service provided by Docker that serves as a central repository for finding, sharing, and managing Docker images. It acts like a GitHub for Docker images, allowing developers to store their custom images, explore official images from vendors, and access images created by the community. Its five key benefits include:

1. **Public and Private Repositories**: Allowing both public sharing and secure private storage of images;
2. **Official Images**: Providing trusted, pre-built images for popular software and operating systems;
3. **Automated Builds**: Integrating with source code repositories (like GitHub) to automatically build images upon code changes;
4. **Webhooks**: Triggering actions (like deployments) upon image pushes;
5. **Team Collaboration**: Facilitating sharing and management of images within development teams.

**Software design patterns (for highly complex software)**

Software design patterns are generalized, reusable solutions to common problems encountered during software design. They are not direct code but rather templates for how to solve recurring design challenges in **object-oriented programming**, providing a shared vocabulary and best practices to improve code maintainability, readability, and scalability.

For highly complex software, architectural patterns and enterprise integration patterns are particularly crucial, such as the Microservices pattern (for breaking down large systems into small, independent services), Event-Driven Architecture (for loosely coupled communication), Service Mesh (for managing inter-service communication), and Domain-Driven Design (DDD) principles (for aligning software design with complex business domains), all of which help manage complexity by promoting modularity, decoupling, and clear separation of concerns.

**Security in API**

Security in APIs (Application Programming Interfaces) is paramount to protect sensitive data and prevent unauthorized access or malicious attacks, as APIs serve as direct entry points to an application's backend. Key measures include **authentication** (verifying the identity of the client, typically using API keys, OAuth 2.0, or JWTs), **authorization** (determining what an authenticated client is allowed to do, often using role-based access control), **data encryption** (using HTTPS/SSL/TLS to secure data in transit), **input validation and sanitization** (preventing injection attacks like SQL injection or XSS), **rate limiting** (to prevent abuse or Denial-of-Service attacks), and comprehensive **logging and monitoring** (to detect and respond to suspicious activity). Implementing these layers of defense ensures that only legitimate users and applications can interact with the API in intended ways, safeguarding the underlying systems and data.

**Optimize a program that is slow in the API and Database**

Optimizing a program that is slow at the API and database levels often requires a multi-faceted approach, as performance issues in these areas are frequently intertwined. Here are key strategies for each:

**API Optimization Strategies:**

**1. Reduce Network Overhead and Payload Size:**

* **Minimize Data Returned**: Instead of sending entire objects, filter responses to include only the necessary fields the client requires (/?fields=id,name,email).
* **Data Compression**: Implement Gzip or Brotli compression on API responses to reduce the amount of data transferred over the network.
* **Efficient Protocols/Formats**: Consider using more efficient serialization formats like Protocol Buffers or MessagePack, or a protocol like gRPC, which can be faster than traditional REST with JSON for certain use cases, especially in microservices communication.
* **Batching Requests**: Allow clients to combine multiple small requests into a single, larger request to reduce round trips.

**2. Implement Caching at Various Levels:**

* **Client-side Caching**: Utilize HTTP caching headers (e.g., Cache-Control, ETag) to allow browsers or mobile apps to cache API responses, reducing the need for repeated requests.
* **API Gateway/Edge Caching**: Implement caching at the API Gateway or CDN level for frequently accessed, non-volatile public data.
* **Service-level Caching**: Cache results of expensive computations or database queries within the API service itself (e.g., using Redis or Memcached), so subsequent requests can be served from memory instead of hitting the database or other downstream services.

**3. Optimize Asynchronous Processing for Long-Running Tasks:**

* **Decouple Operations**: For tasks that take a long time to complete (e.g., generating reports, processing large files, sending emails), avoid making the API request block until the task is finished. Instead, have the API quickly return an acknowledgment, put the task onto a message queue (e.g., RabbitMQ, Kafka), and process it asynchronously in a separate worker service.
* **Webhooks/Polling**: Provide a mechanism (like webhooks or a polling endpoint) for the client to be notified when the asynchronous task is complete, or to check its status.

**Database Optimization Strategies:**

**1. Indexing and Query Tuning:**

* **Proper Indexing**: Create appropriate indexes on columns frequently used in WHERE clauses, JOIN conditions, ORDER BY clauses, or GROUP BY clauses. Over-indexing can also hurt performance on writes, so index strategically.
* **Analyze Slow Queries**: Use database profiling tools (e.g., EXPLAIN in SQL, MongoDB's explain()) to understand the execution plan of slow queries, identify bottlenecks, and rewrite them for efficiency.
* **Avoid N+1 Queries**: Ensure your application isn't making N extra database queries for every one main query (e.g., loading a list of posts, then making a separate query for each post's author). Use JOINs or ORM features for eager loading.

**2. Efficient Connection Management and Pooling:**

* **Connection Pooling**: Implement connection pooling on your application servers. Reusing existing database connections instead of opening and closing a new one for every request significantly reduces overhead and improves throughput.
* **Tune Pool Size**: Configure the connection pool size appropriately. Too few connections can lead to waiting, too many can overwhelm the database.

**3. Schema Design and Scaling Strategies:**

* **Denormalization (Strategic)**: While normalization is good for data integrity, carefully chosen denormalization (duplicating some data) can significantly speed up read-heavy queries by avoiding complex joins, especially when joins are slow.
* **Correct Data Types**: Use the most appropriate and smallest possible data types for your columns (e.g., SMALLINT instead of INT if values are small) to reduce storage and improve performance.
* **Database Scaling**: For very high loads, consider scaling strategies like:
  + **Read Replicas**: Direct read traffic to read-only replicas to offload the primary database.
  + **Sharding/Partitioning**: Distribute data across multiple database servers to horizontally scale both storage and query processing.
  + **Vertical Scaling**: Upgrade the database server's hardware (more CPU, RAM, faster SSDs).

**Security in Spring boot applications**

Security in Spring Boot applications is primarily managed through **Spring Security**, a powerful and highly customizable authentication and access-control framework. It provides comprehensive solutions for common security requirements, including user authentication (e.g., username/password, OAuth 2.0, JWT), authorization (defining access rules based on roles or permissions for URLs, methods, or specific data), protection against common vulnerabilities like CSRF and XSS, and integration with various authentication providers. Developers can easily configure these features in their Spring Boot applications, leveraging Spring Security's defaults and extensibility to build robust and secure web services and applications.

**C# Asp.net**

ASP.NET, coupled with C#, is a widely used, open-source web framework developed by Microsoft for building modern, cloud-based, internet-connected applications, including web apps, APIs, and microservices. It provides a comprehensive set of tools, libraries, and design patterns (like MVC or Razor Pages) that streamline the development process, enabling C# developers to create high-performance, cross-platform web solutions. ASP.NET handles common web development concerns such as request handling, routing, security, and data access, allowing developers to focus on business logic while leveraging the power and type-safety of the C# language and the extensive .NET ecosystem.

**DevOps and how it helps developers**

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) to shorten the systems development life cycle and provide continuous delivery with high software quality. For developers

1. DevOps significantly helps by streamlining workflows, automating repetitive tasks, and providing faster feedback loops.
2. It enables continuous integration and continuous delivery (CI/CD), meaning developers can commit code more frequently, have automated tests run immediately, and see their changes deployed rapidly and reliably.
3. This automation reduces manual errors, speeds up the release cycle, allows developers to spend less time on operational overhead and more time on writing code, and fosters a culture of shared responsibility and rapid iteration.

**Monolithic vs Microservices**

A **monolithic architecture** designs an application as a single, indivisible unit where all its components (UI, business logic, data access) are tightly coupled and run within a single process. While simpler to develop initially and deploy, this approach can become complex to manage, scale, and update as the application grows, as any change requires redeploying the entire system. In contrast, **microservices architecture** structures an application as a collection of small, independent, and loosely coupled services, each performing a specific business capability and communicating via lightweight mechanisms. This allows for independent development, deployment, and scaling of individual services, offering greater agility, resilience, and flexibility for large and complex systems.

**MVC (Model-View-Controller)**

MVC (Model-View-Controller) is a popular architectural pattern that separates an application into three interconnected components, aiming to divide responsibilities and improve maintainability. The Model manages the application's data, logic, and rules. The View is responsible for presenting the data to the user, typically the user interface. The Controller acts as an intermediary, receiving input from the user (via the View), processing it by interacting with the Model, and then updating the View based on the Model's state. This separation of concerns allows for parallel development, easier testing, and more flexible UI changes without affecting the underlying data or business logic.

**Synchronous task**

A synchronous task in programming refers to an operation that must complete entirely before the program can proceed to the next instruction. When a synchronous task is executed, the program "waits" for that specific task to finish its work, blocking any subsequent operations until it returns a result or completes its process. This linear, blocking execution flow is straightforward to reason about but can lead to unresponsive applications, especially in environments like web browsers where long-running synchronous tasks can freeze the user interface, or on servers where they can hold up resources and reduce throughput

**Constructors in programming**

In object-oriented programming, a **constructor** is a special type of method that is automatically called when an object of a class is created (instantiated). Its primary purpose is to initialize the newly created object's state, setting up its initial values for instance variables and performing any necessary setup operations before the object can be used. Constructors typically share the same name as their class and do not have a return type, ensuring that objects are in a valid and usable state immediately after their creation, facilitating proper encapsulation and reliable object behavior

**HTTP Methods**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **HTTP Method** | **Description / Purpose** | **Safety (Read-only)** | **Idempotence (Repeatable Result)** | **Common Use Case** |
| GET | Retrieves data or a resource from the server. Should not have side effects. | Yes | Yes | Fetching a resource (e.g., GET /users/123), querying a collection (e.g., GET /products?category=electronics) |
| POST | Submits data to the server to create a new resource. Can also be used to submit data for processing that is not necessarily resource creation. | No | No | Creating a new user (e.g., POST /users), sending form data, uploading a file. |
| PUT | Updates an existing resource with the provided data, or creates it if it doesn't exist. It replaces the entire resource. | No | Yes | Updating an entire user profile (e.g., PUT /users/123), replacing a document. |
| DELETE | Removes a specified resource from the server. | No | Yes | Deleting a specific user (e.g., DELETE /users/123), removing a blog post. |
| PATCH | Applies partial modifications to a resource. It's used to update only specific fields of a resource without replacing the entire entity. | No | Yes | Updating only a user's email address (e.g., PATCH /users/123 with just the email field). |
| HEAD | Retrieves only the headers of a resource, without the actual body. Identical to GET but without the response body. | Yes | Yes | Checking if a resource exists, verifying content type or last modified date before a GET request. |
| OPTIONS | Describes the communication options for the target resource. Often used by clients to understand the capabilities of an endpoint (e.g., what methods are supported). | Yes | Yes | Pre-flight requests in CORS (Cross-Origin Resource Sharing), API discovery. |