### 1. Architecture Design

#### (1) Basic

\*\*Question:\*\*

Describe the role of Model, View, and Controller in the MVC architecture, providing specific examples for each.

\*\*Answer:\*\*

- MVC(model-view-controller) is a design pattern that split an application to 3 components for easy maintainability and reusability.

1. Model

- to represent data and business logic like database interactions, rules, and validations.

- for example validations in business logic

- Example: User.prototype.isAdmin = function(){

return this.role === 'admin';

}

2. View

- render data output to User Interface can be display using any frontend like react,vue or just HTML

- create user interaction with the data had been display like button, form and many more.

- Example using react: function displayUsername({user}) {

return (

<div><h1>{user.name}</h1></div>

);

}

3. Controller

- a bridge between View and Model where the data will be process in order to update the Model and render View

- Example: const getUser = async(req, res) => {

try{

const user = await User.findByPk(req.param.id)

if(!user) return res.status(404).json({error: "User not found"})

res.render("displayName",{user})

}catch(error){

res.status(500).json({error: "Server Error"})

}

}

#### (2) Intermediate

\*\*Question:\*\*

Explain the basic concepts of Clean Architecture (or Onion Architecture), highlighting its differences from MVC. Provide a sample directory structure for implementing Clean Architecture.

\*\*Answer:\*\*

Clean Architecture or Onion Architecture is design pattern that stress on separation of concerns, dependency rule and testability. In dependency rule there layer which inner layers never depend on outer later.

The layer content several part of it, in the example the layer will be sort based on from innermost to outermost:

1. Entities(Domain Layer)

- contains enterprise-wide business rules for example like User, Order domain objects

- Example in core/entities/User.ts

export class User {

constructor(public id: string, public username: string, public email: string)

}

2.Use Case(Application Layer)

- contains application-specific business logic not UI related for example CreateUser

- use interface to communicate with infrastructre

- in core/usecases/CreateUser.ts

export class CreateUser {

constructor(private userRepository: UserRepository) {}

execute(name: string): User {

const user = new User(generateId(), name);

this.userRepository.save(user);

return user;

}

}

3.Interface Adapters(Controller, Presenter, Gateways)

- it process data between layers for example in controller and ORM

- in infrastructure/web/UserController.ts

export class UserController {

constructor(private createUser: CreateUser) {}

async handleRequest(req: Request, res: Response) {

const user = this.createUser.execute(req.body.name);

res.json(user);

}

}

4.Frameworks & Drivers(Outer Layers)

- external tools or library like db, UI, and HTTP framework

- for example using TypeORM which is an Object Relational Mapper for Ts and Js

- in infrastructure/db/TypeORMUserRepository.ts

export class TypeORMUserRepository implements UserRepository {

async save(user: User) {

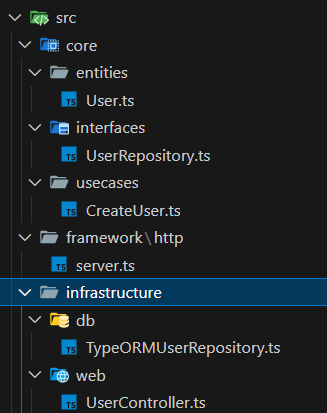
await getRepository(UserEntity).save(user);

}

}

MVC vs Clean Architecture

|  |  |  |
| --- | --- | --- |
| Aspect | MVC | Clean Architecture |
| Dependencies | Controller depend on Models and View | Inner layers never depend on outer layers |
| Flexibility | Dependant to framework | Easily swap |
| Business Logic | Often used in Controllers and Views | Strictly used in User Cases and Entities |
| Testability | Requires external mocking frameworks | Core logic and be test without mocks |



#### (3) Advanced

\*\*Question:\*\*

Compare Microservices Architecture and Monolithic Architecture, discussing the advantages and disadvantages of each. Describe strategies for migrating from a monolith to microservices.

\*\*Answer:\*\*

Microservice vs Monolithic

|  |  |  |
| --- | --- | --- |
| Aspect | Microservice | Monolith |
| Structure | Decoupled Service, independent | Single codebase as one unit |
| Db | Each service own its db | Shared db |
| Communication | HTTP, event streaming | Function call |
| Scalability | Horizontal scale | Vertical scale |
| Team Ownership | Cross team | Single team |

Microservices Pros & Cons

|  |  |
| --- | --- |
| Pros | Cons |
| Independent Scaling | Complexity |
| Fault Isolation | Network Latency |
| Tech Flexibility | Data Consistency |

Monolith Pros & Cons

|  |  |
| --- | --- |
| Pros | Cons |
| Simpler Development | Scalability Issues |
| Easier Testing | Tight coupling |
| Performance | Tech Lock-in |

Migration Strategies

1. Can gradually replace monolith features with microservices for example can redirect traffic to new feature to another proxy request like /api/v2/createUser in microservice.
2. Create separate database and use event sourcing to sync data like Kafka
3. Can create adapter to translate between monolith and microservices in order for both of them to communicate.

### 2. Object-Oriented Programming and Design Patterns

#### (4) Basic

\*\*Question:\*\*

List and explain the four fundamental principles of Object-Oriented Programming (OOP) with concrete examples.

\*\*Answer:\*\*

1. Encapsulation
   1. Attributes and functions put together into single unit in a class so we can restrict direct access to some of components by using private or protected to hide the internal state.
   2. Components can be only retrieved via setter or getter
   3. Easier to validate and avoid unintended state been modify.
   4. For example:

class User {

private username: String;

constructor(username: String){  
 this.username = username

}

public setUsername(username: String): void{

this.username = username

}

public getUsername():void{

return this.username

}

}

const user = new User(“John”);

console.log(user.getUsername())

console.log(user.username) //Error cause private property

1. Inheritence
   1. Reuse and extend code from an existing class(parent) to a new class
   2. Child can inherit methods and properties from parents
   3. Use extends for inheritance and super() to call parent constructors in order to avoid duplication and support hierarchical categorization.
   4. For example in code:

class User {

constructor(public name:String){}

greet():string{

return `Hello, ${this.name}`

}

}

class AdminUser extends User {

banUser(username:string): string {

return `${username} has been banned`

}

}

const user = new AdminUser(‘bob’)

user.greet();

user.banUser(‘john’);

1. Polymorphism
   1. Ability of an object to take different forms which allowing methods to behave differently
   2. Can override parent method in order to redefine it
   3. Interface or Abstract classes define a contract for multiple implementations
   4. Enable modularity without breaking code
   5. Support open/closed principle in SOLID
   6. For example reused class User from before

class PremiumUser extends User {

override greet():string{

return `Welcome to VIP user, ${this.name}`

}

}

Const user = new PremiumUser(‘Alice’)

user.greet()

1. Abstraction
   1. Used abstract or interface to define blueprints
   2. Reduce complexity and promote loose coupling
   3. For example

abstract class AuthService {

abstract login(username: string, password: string): boolean;

}

class DatabaseAuthService extends AuthService {

login(username: string, password: string): boolean {

console.log(`Authenticating ${username} via database...mock only`);

return true;

}

}

class OAuthService extends AuthService {

login(username: string, password: string): boolean {

console.log(`Authenticating ${username} via Oauth...mock only`);

return true;

}

}

// Usage (client code doesn't care about the implementation)

const auth: AuthService = new DatabaseAuthService();

auth.login("alice", "1234");

#### (5) Intermediate

\*\*Question:\*\* Choose three Gang of Four (GoF) design patterns and explain their purpose, structure, and provide implementation examples for each.

\*\*Answer:\*\*

1. Singleton Pattern
   1. Purpose: Ensure class has only one instance and can be access in global
   2. Structure
      1. Lazy initialization
      2. Static method to retrieve instance
      3. Private constructor
   3. Implementation
      1. Configuration management
      2. Logging
      3. Shared resource access

class AppConfig {

private static instance: AppConfig;

private constructor() {} // Block creation

static getInstance(): AppConfig {

If (!AppConfig.instance) {

AppConfig.instance = new AppConfig();

}

return AppConfig.instance;

}

}

const config1 = AppConfig.getInstance();

const config2 = AppConfig.getInstance();

console.log(config1 === config2); // true cause same instance

1. Observer pattern
   1. Define one-to-many dependency between objects so when one object changes all dependent will be notified.
   2. Structure
      1. Maintains list of observers and notify them
      2. Implement and update method
   3. Implementation
      1. Real time notification
      2. Event handling system

class Button {

private clicks: (() => void)[] = [];

onClick(handler: () => void) {

this.clicks.push(handler);

}

click() { // Notify all

this.clicks.forEach(handler => handler());

}

}

const button = new Button();

button.onClick(() => console.log("Clicked!"));

button.click();

1. Strategy Pattern
   1. Purpose: Family of algorithm encapsulate each one and interchangeable.
   2. Structure
      1. Context
      2. Implementation of interface(Strategies)
   3. Implementation
      1. Replace conditional logic
      2. Select algorithms in runtime

// Define strategies

const fastShipping = () => console.log("Fast shipping selected");

const cheapShipping = () => console.log("Cheap shipping selected");

//Define Context

class Checkout {

constructor(private shippingStrategy: () => void) {}

setStrategy(strategy: () => void) {

this.shippingStrategy = strategy;

}

checkout() {

this.shippingStrategy();

}

}

const checkout = new Checkout(fastShipping);

checkout.checkout(); // "Fast shipping selected"

checkout.setStrategy(cheapShipping);

checkout.checkout(); // "Cheap shipping selected"

#### (6) Advanced

\*\*Question:\*\*

Explain each principle of SOLID, highlighting the issues with code that violates these principles, and provide examples of how to correct such violations.

\*\*Answer:\*\*

1. Single Responsibility Principle
   1. Definition: A class should have one responsible only
   2. Violation example: Class User handle db operations and user properties

class User{

constructor(public name: string, public email: string){}

saveToDatabase(){

console.log(`Save ${this.name} to db`)

}

}

* 1. Correct example: It should be split into two different class

class User{

constructor(public name: string, public email: string) {}

}

class UserRepository{

save(user: User) {

console.log(`Saving ${user.name} to DB...`);

}

}

1. Open/closed Principle (OCP)
   1. Definition: Class should be open to extension and closed to modification
   2. Violation example: A Discount class hardcoded discount types

Which will cause a problem when add new type discount

class Discount {

apply(amount: number, type: “fixed” | “percentage”){

if (type === "fixed") return amount - 10;

if (type === "percentage") return amount \* 0.9;

}

}

* 1. Correct example: it should used abstraction

interface DiscountStrategy {

apply(amount: number): number;

}

class FixedDiscount implements DiscountStrategy {

apply(amount: number) { return amount - 10; }

}

class PercentageDiscount implements DiscountStrategy {

apply(amount: number) { return amount \* 0.9; }

}

class Discount {

constructor(private strategy: DiscountStrategy) {}

apply(amount: number) {

return this.strategy.apply(amount);

}

}

1. Liskov Substitution Principle
   1. Definition: Subclass should be replaceable for their parent class without breaking behaviour
   2. Violation example: A Square extending Rectangle but breaking the behaviour

class Rectangle {

constructor(public width: number, public height: number) {}

area() { return this.width \* this.height; }

}

class Square extends Rectangle {

constructor(size: number) {

super(size, size);

}

setWidth(width: number) {

this.width = this.height = width;

}

}

const rect: Rectangle = new Square(5);

rect.setWidth(10);

* 1. Correct example: Avoid inheritance if behaviour is changes

interface Shape {

area(): number;

}

class Rectangle implements Shape {

area() { return this.width \* this.height; }

}

class Square implements Shape {

area() { return this.size \*\* 2; }

}

1. Interface Segregation Principle
   1. Definition: Should not used interfaces that not consume
   2. Violation example: Robot class implement irrelevant method

Interface Human {

work();

eat();

sleep();

}

Class Robot implements Human {

work(console.log(“8 hours”));

eat(throw new Error(“Robot don’t eat”));

}

* 1. Correct example: Should split it smaller interface and implement only interface needed

Interface Workable(){

work: void;

}

Interface Eatable(){

eat(): void;

}

Class Robot implement Workable(){

work(){console.log(“8 hours”)}

}

1. Dependency Inversion Principle
   1. Definition: Depend on abstractions not concrete implementation
   2. Violation example which cant swith database without changing User class

class User{

private db = new MySqlDb();

save(){

this.db.saveUser(this)

}

}

* 1. Corrected Example: Depend on interface

interface UserRepository {

save(user: User): void;

}

class MySqlRepository implements UserRepository {

save(user:User){

console.log(“save to mysql”)

}

}

Class User {

constructor(private repository:UserRepository){}

save(){

this.repository.save(this)

}

}

### 3. Modern Web Development

#### (7) Basic

\*\*Question:\*\* Explain the differences between SPA (Single Page Application), MPA (Multi-Page Application), SSR (Server-Side Rendering), and SSG (Static Site Generation), with suitable use cases for each.

\*\*Answer:\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Description | Pros | Cons | Use Case |
| SPA(Single Page Appllication) | Loads a single HTML page and dynamically updates using JS without refresh whole page | 1. Fast client-side navigation  2. Good user experience with dynamic content update | 1.Slow load cause of JS bundle need to load  2.Reequire JS to be used | 1.Analytic tools  2.Admin dashboard |
| MPA(Multi page App) | 1.Full page reloads on navigation  2.Rendered HTML | 1.Simple architecture  2.Fast initial render | 1.Less interactive  2.Slower Navigation | 1.Ecommerce  2.News website |
| SSR(Server Side Rendering) | 1.Generate HTML on server request.  2.Hydrates to SPA after load | 1.Fast initial load  2.Great Seo | 1.Higher server cost  2.Slower time to interact | 1.Ecommerce  2.Marketing website |
| SSG(Static Site Gen) | 1.Server static file  2. Pre build HTML at deploy time | 1.Fast  2.Cheap | 1.No real data  2.Rebuild the website if update | 1. Blogs  2. Landing |

#### (8) Intermediate

\*\*Question:\*\* Describe the basic structure of a CI/CD pipeline and explain best practices in web application development. Provide concrete implementation examples using tools like GitHub Actions or Jenkins.

\*\*Answer:\*\*

1. Basic structure of CI/CD pipeline
   1. Trigger: Trigger by code changes and retrieve code from repository.
   2. Build: Compile code, install dependencies and produce artifacts
   3. Test: Run automation like Jest or Selenium
   4. Deploy: Delivers code to Staging or Production environment
2. Best Practice for Web Apps
   1. Small and frequent commit to minimize code conflict and easier debug
   2. Separate environment using Staging and Production
   3. Use environment variable
   4. Use rollback mechanism
   5. Monitor the application after deploy
3. Implementation Example

name: CI/CD Pipeline

on: push

jobs:

test:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- run: npm install

- run: npm test # Runs Jest

deploy:

needs: test

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- run: npm install && npm run build

- uses: actions/upload-artifact@v3

with:

name: build

path: build/

- run: aws s3 sync build/ s3://my-bucket # Deploy to S3

env:

AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_KEY }}

AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET }}

#### (9) Advanced

\*\*Question:\*\* Compare RESTful APIs and GraphQL from a technical perspective, including appropriate use cases and selection criteria for real-world projects.

\*\*Answer:\*\*

1. Technical comparision between Restful API and GraphQL
   1. API Structure & Endpoints
      1. REST : Each resource have it owns data
         1. For example:
            1. GET /users
      2. GraphQL: Fetch data via single endpoints
         1. For example:
            1. query {

user(id:1){

name

}

}

* 1. Efficiency of data fetching
     1. REST: Overfetching
        1. For example:
           1. {

“id”: 1,

“name”: ”John”,

“age”: 8

}

* + 1. GraphQL: Precise
       1. For example:
          1. {

user(id:1){

name

}

}

* 1. Nested Data
     1. Rest: often require multiple round trips
        1. For example: GET /user/1 -> GET /users/1/posts
     2. GraphQL: fetch nested objects in one go
        1. For example

{

user(id:1){

name

posts{

comments{

content

}

}

}

}

* 1. Versioning
     1. REST: /api/v1/users
     2. GraphQL: schema evolves by deprecating fields

1. Use Cases
   1. Choose REST when
      1. Simple data like CRUD operations
      2. Microservice communication
   2. Choose GraphQL
      1. Complex and nested data
      2. Mobile apps in order to reduce payload size
2. Example of real cases
   1. REST: Stripe and Twitter
   2. GraphQL: Github, Shopify
3. Selection Criteria:
   1. REST
      1. Need something quick and simple
      2. Rely on HTTP tools /middleware
   2. GraphQL
      1. Client need dynamic data
      2. Team familiar with GraphQL

### 4. Database and Persistence

#### (10) Basic

\*\*Question:\*\* What is database normalization? Explain the differences between 1NF, 2NF, and 3NF with concrete examples.

\*\*Answer:\*\*

1. What is database normalization?
   1. Process of organizing data in database to:
      1. Eliminate redundancy
      2. Ensure data integrity
      3. Improve data consistency
      4. Simplify maintenance
2. 1NF vs 2NF vs 3NF
   1. 1NF
      1. No repeating groups or arrays in single field
      2. Each column holds atomic values
      3. Violated example: Unnormalize Table Example

|  |  |  |
| --- | --- | --- |
| OrderID | CustomerName | Items |
| 101 | Alice | Pen,Notebook,Eraser |

* + 1. Corrected example: 1NF table split the multi valued Items column into individual rows to make atomic.

|  |  |  |
| --- | --- | --- |
| OrderID | CustomerName | items |
| 101 | Alice | Pen |
| 101 | Alice | Notebook |
| 101 | Alice | Eraser |

* 1. 2NF
     1. Must first be 1NF
     2. Non key attributes should depend in primary key not be part of it
        1. Violated example: Assume composite primary key: (OrderID, Item)

|  |  |  |
| --- | --- | --- |
| OrderID | CustomerName | Item |
| 101 | Alice | Pen |
| 101 | Alice | Noterebook |

* + - 1. Corrected example: 2NF Tables every non-key depend on full primary key

Orders Table

|  |  |
| --- | --- |
| OrderID | CustomerName |
| 101 | Alice |

OrderItem Table:

|  |  |
| --- | --- |
| OrderID | Item |
| 101 | Pen |
| 101 | Notebook |

* 1. 3NF
     1. Must first be 2NF
     2. Non key attributes should not depend on other non-key attributes
        1. Violated example: City depend on ZIP not with primary key

|  |  |  |  |
| --- | --- | --- | --- |
| CustomerID | CustomerName | ZIP | City |
| 1 | Alice | 10001 | New York |
| 2 | Bob | 20001 | Washington |

* + - 1. Corrected example: City linked through separate table

|  |  |  |
| --- | --- | --- |
| CustomerID | CustomerName | ZIP |
| 1 | Alice | 10001 |
| 2 | Bob | 20001 |

|  |  |
| --- | --- |
| ZIP | City |
| 10001 | New York |
| 20001 | Washington |

#### (11) Intermediate

\*\*Question:\*\* Compare NoSQL databases (MongoDB, DynamoDB, Redis, etc.) and relational databases (MySQL, PostgreSQL, etc.), discussing suitable use cases and selection criteria for each.

\*\*Answer:\*\*

1. NoSQL vs Relational Database

1. Data Modal

1.NoSQL Database

i. Flexible schema

ii.Four main types:

a. Document-based: MongoDB

b.Key-value: Redis

c. Column-family: Cassandra

d.Graph-based: Neo4j

iii.Best for:

a. Unstructured / semi-structured data

b. Dynamic data models

2. Relational Database

i.Structured in table

ii.Use schema to define strict data types

iii.Relationship through foreign keys.

iv. Best for:

a.Structured relational data with complex queries and transactions

b.Example: A banking system

2. Scability

1. NoSQL

i. horizontal scaling

ii. Easily handle big data and distributed systems

2.Relational

i.Vertical scaling

ii.Horizontal scaling is complex

3. ACID vs BASE

1.NoSQL

i. Follow BASE(Basically Availability, Soft state, Eventual consistency)

ii. Prioritizes availability and scalability over strong consistency

2. Relational DB

i.ACID properties(Atomicity, Consistency, Isolation, Durability)

ii.Ensure strong data consistency and integrity

4. Querying and Transactions

1.NoSQL

i. varies by type

ii. limited joins

2.Relational

i.powerful SQL querying language

ii.strong join support and relational integrity

2. Use cases

|  |  |  |
| --- | --- | --- |
| Use Case | Preferred DB Type | Example DB |
| E-commerce checkout & orders | Relational | PostgreSQL, MySQL |
| Product catalog with flexible schema | NoSQL | MongoDB |
| Real-time leaderboard or caching | NoSQL (key-value) | Redis |
| Banking or financial ledgers | Relational | PostgreSQL |

3. Selection criteria

|  |  |  |
| --- | --- | --- |
| Criteria | Relational DBs | NoSQL DBs |
| **Data Structure** | Structured, fixed schema | Semi/unstructured, flexible |
| **Query Language** | SQL | Varies (Mongo Query, etc.) |
| **Transactions** | Strong ACID | Limited or eventual consistency |
| **Scalability** | Vertical (harder to scale) | Horizontal (easy to scale) |
| **Use Case Fit** | Financial, ERP, CRM | CMS, Real-time apps, Analytics |

#### (12) Advanced

\*\*Question:\*\* Explain techniques for database performance tuning, addressing aspects such as index design, query optimization, sharding, and replication with specific examples.

\*\*Answer:\*\*

1. Index Design

i.Purpose: Speed up read queries by reducing full-table scans.

ii.Best Practices

a.Create indexes on frequently queried columns (WHERE, JOIN, ORDER BY).

b.Avoid over-indexing (slows down writes).

c.Use composite indexes for multi-column queries.

iii.Example

CREATE INDEX idx\_users\_email ON users(email);

iv. When to use

a.High-read, low-write tables like user profile or products.

2. Query Optimization

i.Purpose: Reduce query execution time.

ii. Best Practices

a.AvoidSELECT \* → Fetch only needed columns.

b.UseEXPLAIN ANALYZE to debug slow queries.

c.OptimizeJOIN**s** with proper indexes.

iii.Example

EXPLAIN ANALYZE SELECT order\_id, customer\_id FROM orders WHERE status = 'pending'

iv. When to use

a. list of products

3. Sharding (Horizontal Partitioning)

i.Purpose: Distribute data across multiple servers to handle large datasets.

ii.Best Practices

a.Range-based: Split by ID ranges (e.g., user\_id 1-1000 → Shard 1).

b.Hash-based: Distribute via hash function (e.g., user\_id % 4).

iii.Example

sh.enableSharding("mydb");

sh.shardCollection("mydb.users", { "user\_id": "hashed" });

iv. When to use

a.Tables with millions+ rows (e.g., social media posts).

4. Replication

i.Purpose: Improve read scalability and fault tolerance.

ii.Types

a.Master-Slave: Writes go to master, reads from slaves.

b.Multi-Master: Multiple nodes handle writes

iii. Example

ALTER SYSTEM SET wal\_level = 'replica'; //on master

pg\_basebackup -h master-host -D /var/lib/postgresql/replica -U repl\_user -P -v //om replicas

iv. When to use:

a. High-availability systems like ecommerce

### 5. Performance and Security

#### (13) Basic

\*\*Question:\*\* List five common security risks in web applications (such as those from the OWASP Top 10) and explain mitigation strategies for each.

\*\*Answer:\*\*

1. Injection (e.g., SQL Injection)

i. Description**:**

a.Occurs when untrusted data is sent to an interpreter as part of a command or query. Attackers can execute arbitrary SQL, OS, or NoSQL commands.

ii. Mitigation**:**

a.Use parameterized queries or ORMs.

b.Validate and sanitize input.

c.Employ web application firewalls (WAFs).

2. Broken Authentication

i. Description**:**

a. Improper implementation of authentication mechanisms can allow attackers to compromise passwords, keys, or session tokens.

b.Risks**:**

1.Session hijacking

2. Brute-force attacks

3. Predictable login tokens

ii. Mitigation**:**

a. Use strong password policies and 2FA (Two-Factor Authentication).

b. Invalidate sessions on logout.

c. Store passwords with bcrypt or Argon2.

d. Use secure, rotating session tokens.

3. Cross-Site Scripting (XSS)

i. Description**:**

a. XSS allows attackers to inject malicious JavaScript into web pages viewed by other users.

ii. Mitigation**:**

a. Use frameworks that auto-escape output like React

b. Sanitize HTML using libraries like DOMPurify.

c. Set appropriate Content Security Policy (CSP) headers.

4. Insecure Direct Object References (IDOR)

i. Description**:**

a. Occurs when an application exposes internal objects (like files, database records) without proper authorization checks.

ii. Mitigation**:**

a. Always enforce authorization on server side.

b. Use indirect references

c. Validate user access for every sensitive resource.

5. Security Misconfiguration

i. Description**:**

a. Insecure default configurations, open cloud buckets, overly verbose error messages, and unnecessary features can be exploited.

ii. Mitigation**:**

a. Harden and regularly audit configurations.

b. Disable directory listing and debug modes.

c. Apply least privilege principl**e** in servers and APIs.

#### (14) Intermediate

\*\*Question:\*\* List five methods to improve the performance of a web application, providing specific examples and measurement techniques for each.

\*\*Answer:\*\*

1. Optimize Images & Assets

i. Method:

a. Compress images (WebP format).

b. Lazy-load offscreen images.

c. Use CDN for static assets.

ii. Example:

<!-- Lazy-load images -->

<img src="placeholder.jpg" data-src="real-image.jpg" loading="lazy" alt="...">

iii.Measurement:

a. Lighthouse Audit: Check "Properly size images" score.

2. Minimize & Bundle JavaScript/CSS

i. Method:

a. Use Webpack or Vite to bundle and minify files.

b. Code-split with dynamic imports.

ii. Example:

// Dynamic import for lazy-loaded components

const HeavyComponent = React.lazy(() => import('./HeavyComponent'));

iii. Measurement:

a. Webpack Bundle Analyzer: Identify large dependencies.

b. Coverage Tab (Chrome DevTools): Find unused JS/CSS.

3. Enable Caching

i. Method:

a. Set Cache-Control headers for static assets.

b. Use Service Workers for PWA caching.

ii. Example

app.use(express.static('public', {

maxAge: '1y' // Cache for 1 year

}));

iii. Measurement:

a. Chrome DevTools → Network: Check memory/disk cache hits.

4. Database Query Optimization

i. Method:

a. Add indexes to frequent query columns.

b. Use EXPLAIN ANALYZE to debug slow queries.

ii. Example add index

CREATE INDEX idx\_users\_email ON users(email);

EXPLAIN ANALYZE SELECT \* FROM users WHERE email = 'test@example.com';

iii. Measurement:

a. Database logs: Track slow queries (>100ms).

b. PgHero (PostgreSQL): Identify missing indexes.

5. Reduce Server Response Time

i. Method:

a. Upgrade server hardware (CPU/RAM).

b. Use edge caching (Vercel, Cloudflare).

ii. Example

const cluster = require('cluster');

if (cluster.isMaster) {

for (let i = 0; i < 4; i++) cluster.fork(); // Use all CPU cores

}

iii. Measurement:

a. WebPageTest: Monitor Time-To-First-Byte (TTFB).

#### (15) Advanced

\*\*Question:\*\* Explain methods and best practices for securing containerized web applications, addressing aspects such as image scanning, principle of least privilege, and secrets management with specific examples.

\*\*Answer:\*\*

1. Image scanning

i. Purpose:

a. Ensure base images and dependencies are free from vulnerabilities

ii. Best Practices:

a. Use trusted base images

b. Regularly scan image for any vulnerabilities

c. Can automate scanning in CI/CD pipeline

iii. Tools:

a. Docker Scout

iv: Example command

docker scout cves <image>

2. Principal of Least Privilege

i. Non-root containers

a. Problem: Containers running as root can hijack the host

b. Best Practice:

1. Add user and run as non-root

c. Example:

FROM alpine

RUN adduser -D appuser

USER appuser

ii. Read only filesystems

a. Problem: Malicious file writes

b. Best practice:

1. Allow writes only to /tmp

c. Example:

securityContext:

readOnlyRootFilesystem: true

volumes:

- name: tmp

emptyDir: {}

3. Secrets Management

i. Purpose:

a. Protect sensitive data like API keys, database passwords, tokens, etc.

ii. Best Practices:

a. Never hardcode secrets in Dockerfiles or source code.

b. Use Docker secrets, Kubernetes secrets, or tools like HashiCorp Vault.

c. Load secrets via environment variables or external mount points.

iii. Example:

apiVersion: v1

kind: Secret

metadata:

name: db-secret

type: Opaque

data:

DB\_PASSWORD: password

### 6. Cloud and Infrastructure

#### (16) Basic

\*\*Question:\*\* Explain the concept and benefits of Infrastructure as Code (IaC), and discuss the differences between tools like Terraform, AWS CloudFormation, and Ansible.

\*\*Answer:\*\*

1. Concept

i. practice of provisioning and managing cloud infrastructure using machine readable config file, rather than manually configuring resource via UI or CLI. Using IaC infrastructure become:

a. Automated

b. Version controlled

c. Repeatable

2. Benefit

i. Consistency & Repeatability

a. Same script yield same environment every time

ii. Version control

a. Track changes using Git

b. Can rollback and code3 review

iii. Reduce Human Error

a. Automated configuration

iv. Better collaboration

a. Can collaborate on infrastructure itself

v. Disaster Recovery

a. Environment can be recreated easily in case of failure

3. Terraform vs AWS CloudFormation and Ansible

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Tool | Type | Language | Use Case Focus | Multi-Cloud | Example Use |
| Terraform | Declarative | HCL (HashiCorp) | Provisioning Infrastructure | Yes | Create VMs, databases, load balancers |
| CloudFormation | Declarative | JSON/YAML | AWS-specific provisioning | AWS only | Full AWS stack deployment (EC2, S3, etc.) |
| Ansible | Imperative | YAML (Playbooks) | Config Management + Provision | Yes | Install packages, manage files, users |

#### (17) Intermediate

\*\*Question:\*\* Describe the principles of cloud-native application design and how they differ from traditional on-premises applications, providing concrete examples.

\*\*Answer:\*\*

1. Principle

|  |  |  |
| --- | --- | --- |
| Principle | Description | Example |
| **Microservices** | Build applications as a suite of small, independently deployable services | A shopping app with services like Cart, Payment, Inventory |
| **API-First** | Services communicate via APIs, often REST or GraphQL | Each microservice exposes an HTTP API |
| **Containerization** | Package apps and dependencies into containers like Docker | Deploying a Node.js app in a Docker container |
| **DevOps & Automation** | Use CI/CD, automated testing, and monitoring | GitHub Actions pipelines to deploy on push |
| **Immutable Infrastructure** | Deploy new instances rather than updating in-place | Terraform provisions a fresh EC2 instance with each release |
| **Resiliency & Observability** | Handle failures gracefully and monitor via logs/metrics/traces | Circuit breakers, Prometheus + Grafana dashboards |
| **Scalability & Elasticity** | Automatically scale out/in based on demand | Auto-scaling groups or Kubernetes Horizontal Pod Autoscaler (HPA) |
| **12-Factor App Compliance** | Follow best practices around codebase, config, logging, and portability | Config via env vars, stateless processes, and external storage |

2. Cloud vs Traditional

|  |  |  |
| --- | --- | --- |
| Feature / Aspect | Traditional On-Premises | Cloud-Native |
| **Deployment Style** | Monolithic, often manual | Microservices, automated CI/CD |
| **Scalability** | Vertical scaling (manual upgrade) | Horizontal scaling (auto-scale) |
| **Infrastructure** | Static servers (bare metal/VMs) | Ephemeral (containers, serverless) |
| **Provisioning** | Manual or script-based | Automated (IaC: Terraform, CloudFormation) |
| **Monitoring** | Log files, basic alerts | Centralized monitoring, tracing (e.g., Prometheus) |
| **Failure Handling** | Single point of failure | Resilient by design (retry, fallback, replication) |
| **Release Cycles** | Long, risky deployments | Frequent, small releases with rollback capabilities |
| **Cost Structure** | Upfront CapEx | Pay-as-you-go (OpEx) |

3. Real World Example

i. Traditional App (On-Prem)

a. A bank hosts a monolithic Java EE app on on-prem servers.

b. Scaling is difficult: upgrading the server or deploying on new hardware.

c. Release requires downtime and multiple ops engineers.

ii. Cloud-Native App

a. A fintech startup builds a microservices-based Node.js app.

b. Services run in Docker containers on Kubernetes (EKS or GKE).

c. GitHub Actions triggers CI/CD → builds Docker images → deploys with Helm.

d. Logs and metrics are centralized with ELK stack and Prometheus.

#### (18) Advanced

\*\*Question:\*\* Explain the concepts, advantages, and challenges of serverless architecture, providing implementation examples using AWS Lambda, Azure Functions, or Google Cloud Functions, along with suitable use cases.

\*\*Answer:\*\*

1. Basic Principles

i. Event Driven: Automated functions trigger according to regularly scheduled events like HTTP, databases changes, etc.

ii. Zero Infrastructure: Resources for the architecture are not in the form of servers or virtual machines.

iii. Pay-per-Use: Billed each time the function is executed (e.g. increments of 100ms atleast).

1. Advantages

i. Auto-Scaling

a. Can effortlessly accommodate anything from one request to millions.

ii. Cost Efficient

a. Requires no payment when not in use.

iii. Faster deployment

a. Focus on code not infra.

iv. High Availability

a. Cloud redundancy in the system enhances with.

1. Challenges

i. Cold Starts

a. Use provisioned concurrency

ii. Vendor Lock-in

a. Embrace multi-cloud solutions (Serverless Framework).

iii. Debugging Complexity

a. Use distributed tracing to address.

iv. State Management

a. Off-load state to database.

1. Implementation Examples

i. AWS Lambda(Node.js)

a. Use Case: Advanced image thumbnail creation on S3 upload.

b. Example:

exports.handler = async (event) => {

const AWS = require('aws-sdk');

const s3 = new AWS.S3();

const image = await s3.getObject({

Bucket: event.Records[0].s3.bucket.name,

Key: event.Records[0].s3.object.key

}).promise();

const thumbnail = await sharp(image.Body).resize(200, 200).toBuffer();

await s3.putObject({

Bucket: 'thumbnails-bucket',

Key: `thumb-${event.Records[0].s3.object.key}`,

Body: thumbnail

}).promise();

};

ii. B. Azure Functions (Python)

a. Use Case: Process IoT sensor data from a queue.

b. Example

import json

import azure.functions as func

def main(msg: func.QueueMessage) -> None:

sensor\_data = json.loads(msg.get\_body())

if sensor\_data['temp'] > 30:

send\_alert(sensor\_data['device\_id'])

def send\_alert(device\_id: str):

print(f"ALERT: Device {device\_id} overheated!")

iii. Google Cloud Functions (Go)

a. Use Case: REST API for weather data.

b. Example

package p

import (

"net/http"

)

func WeatherAPI(w http.ResponseWriter, r \*http.Request) {

city := r.URL.Query().Get("city")

weather := fetchWeather(city) // Call external API

w.Write([]byte(weather))

}

5. When to use serverless

|  |  |  |
| --- | --- | --- |
| **Use Case** | **Service** | **Why?** |
| **File Processing** | AWS Lambda + S3 | Scale with uploads. |
| **APIs** | Azure Functions | Pay-per-call pricing. |
| **Scheduled Tasks** | GCP Cloud Scheduler | Run cron jobs cheaply. |
| **Real-Time Streams** | AWS Lambda + Kinesis | Process high-velocity data. |