1.1 What is JavaScript?

JavaScript is a ***lightweight, interpreted, object-based scripting language*** used to make web pages interactive.
It runs in the browser and can also run on the server (Node.js).

1.2 Is JavaScript the same as Java?

✕ No.

- **Java: Compiled, strongly typed, class-based OOP language.**
- **JavaScript: Interpreted, dynamically typed, prototype-based scripting language.**

1.3 How to include JavaScript in an HTML page?

```
<!-- Inline JS -->
<button onclick="alert('Hello!')">Click Me</button>

<!-- Internal JS -->
<script>
  console.log("Hello from script tag");
</script>

<!-- External JS -->
<script src="script.js"></script>
```

1.4 What are variables in JavaScript?

Variables store data.
In modern JS (ES6), we use:

- `var (function-scoped, old style)`
- `let (block-scoped)`
- `const (block-scoped, cannot be reassigned)`

```
let name = "John";
const PI = 3.14;
var age = 25;
```

1.5 What are JavaScript data types?

Primitive: string, number, boolean, null, undefined, symbol, bigint

Non-primitive: objects, arrays, functions

```
let str = "Hello";        // string
let num = 42;             // number
let flag = true;          // boolean
let x;                    // undefined
let obj = {a:1, b:2};     // object
let arr = [1,2,3];        // array
```

1.6 What is typeof operator?

```
console.log(typeof 42);        // "number"
console.log(typeof "hello");   // "string"
console.log(typeof null);      // "object" (quirk in JS)
```

---

2. Operators & Comparisons

2.1 Difference between == and ===

- == → checks only **value** (type conversion is done)
- === → checks **value and type**

```
console.log(5 == "5");   // true
console.log(5 === "5");  // false
```

2.2 What are logical operators?

- && → AND
- || → OR
- ! → NOT

```
let a = true, b = false;
console.log(a && b); // false
console.log(a || b); // true
console.log(!a);     // false
```

🔥 3. Functions

3.1 Declaring a function

```
function add(a, b) {
  return a + b;
}
console.log(add(2, 3)); // 5
```

3.2 Arrow functions (ES6)

```
const multiply = (a, b) => a * b;
console.log(multiply(4, 5)); // 20
```

3.3 Default parameters

```
function greet(name = "Guest") {
  console.log("Hello, " + name);
}
greet();            // Hello, Guest
```

```
greet("Nisha");     // Hello, Nisha
```

嘺 4. Control Statements

```
let n = 10;
if (n > 5) console.log("Greater than 5");

for (let i = 1; i <= 3; i++) {
  console.log(i);
}

let i = 1;
while (i <= 3) {
  console.log(i);
  i++;
}
```

廳 5. Objects & Arrays

5.1 Creating and accessing objects

```
let emp = { name: "John", salary: 50000 };
console.log(emp.name);        // John
console.log(emp["salary"]);   // 50000
```

5.2 Array operations

```
let arr = [1,2,3];
arr.push(4);        // [1,2,3,4]
arr.pop();          // [1,2,3]
arr.unshift(0);     // [0,1,2,3]
console.log(arr.length); // 4
```

5.3 Iterating over arrays

```
arr.forEach(num => console.log(num));

let squares = arr.map(x => x*x);
console.log(squares);
```

瀘 6. Important Concepts

6.1 What is this keyword?

- this refers to the **context** in which a function is called.

```
console.log(this);  // In browser, refers to window

let obj = {
  name: "John",
```

```
  greet: function() {
    console.log("Hello " + this.name);
  }
};
obj.greet();        // Hello John
```

6.2 What is Hoisting?

JavaScript moves variable & function declarations to the top during compilation.

```
console.log(x);  // undefined (not error)
var x = 10;
```

6.3 Difference between var, let, and const

Keyword	Scope	Re-declare	Re-assign
var	function	✔	✔
let	block	✖	✔
const	block	✖	✖

6.4 What is an IIFE (Immediately Invoked Function Expression)?

```
(function(){
  console.log("Runs immediately!");
})();
```

🧠 7. DOM Manipulation (Common in Interviews)

```
<div id="msg">Hello</div>
<button onclick="changeText()">Click Me</button>

<script>
function changeText() {
  document.getElementById("msg").innerText = "Hello JavaScript!";
}
</script>
```

⚡ 8. Events

```
document.getElementById("btn").addEventListener("click", function(){
  alert("Button Clicked!");
});
```

🔍 9. ES6 Features Often Asked

- Arrow functions

- Template literals
- Destructuring
- Spread/rest operators
- Modules (import/export)

Example:

```
const person = {name:"Alice", age:25};
const {name, age} = person;
console.log(name, age);   // Alice 25
```

✍ 10. Annotations / Comments in JavaScript

Annotations are usually comments or JSDoc used to explain code.

Single-line comment

```
// This is a single-line comment
```

Multi-line comment

```
/*
 This is a
 multi-line comment
*/
```

JSDoc Annotation

```
/**
 * Adds two numbers
 * @param {number} a
 * @param {number} b
 * @returns {number} sum
 */
function add(a,b){
  return a+b;
}
```

🔥 11. Common Interview Questions

Question	Short Answer
What is NaN in JavaScript?	"Not-a-Number" — result of invalid math operation (e.g., 0/0)
Is JavaScript single-threaded?	Yes, it’s single-threaded but uses an event loop for async tasks
Difference between null and undefined	null = empty value assigned by programmer; undefined = variable declared but not assigned
What is callback function?	A function passed as an argument to another function to be executed later
What is Promise?	An object that represents the result of an async operation
What is async/await?	Syntactic sugar to handle Promises in a cleaner, synchronous-like way

Question	Short Answer
What are template literals?	Strings with backticks and${}` interpolation

---

責 Pro Tips

- Focus on syntax, variables, functions, DOM, and ES6 basics — most common in interviews for Java devs.
- Be ready to compare Java vs JavaScript.
- You don’t need to learn frameworks (React, Angular) unless job description mentions them.
- JSDoc comments and basic annotations show good coding practice.

JavaScript Mock Interview Q&A

透 Section 1: Basics (Q1–Q5)

Q1: What is JavaScript?

> JavaScript is a lightweight, interpreted, single-threaded, prototype-based scripting language used to make web pages interactive.
> It can also run on servers via Node.js.

Q2: Is JavaScript the same as Java?

> ✕ No.

- **Java: Compiled, class-based, strongly typed.**
- **JavaScript: Interpreted, dynamically typed, prototype-based.**

Q3: How do you include JavaScript in a web page?

```
<!-- Internal -->
<script>
  console.log("Hello");
</script>

<!-- External -->
<script src="script.js"></script>
```

Q4: What are the data types in JavaScript?

- **Primitive: string, number, boolean, null, undefined, symbol, bigint**
- **Reference: objects, arrays, functions**

```
let str = "Hello";
let num = 42;
```

```
let obj = {a:1};
```

Q5: What is the difference between var, let, and const?

Keyword	Scope	Re-declare	Re-assign
var	function-scoped	✔	✔
let	block-scoped	✖	✔
const	block-scoped	✖	✖

Section 2: Operators & Control (Q6–Q9)

Q6: What's the difference between == and ===?

```
5 == "5"   // true  (compares value only)
5 === "5"  // false (compares value and type)
```

Q7: What are null and undefined?

- **undefined: variable declared but not assigned a value.**
- **null: intentional empty value.**

Q8: Write a simple for-loop to print numbers 1 to 5.

```
for(let i=1; i<=5; i++) {
  console.log(i);
}
```

Q9: Write an if statement to check if a number is positive.

```
let n = 10;
if(n > 0) console.log("Positive");
```

灀 Section 3: Functions (Q10–Q13)

Q10: How do you declare a function in JavaScript?

```
function add(a, b) {
  return a + b;
}
```

Q11: What is an arrow function?

```
const multiply = (a, b) => a * b;
```

Q12: What is a callback function?

A function passed as an argument to another function to be executed later.

```
function greet(name, callback) {
  console.log("Hello " + name);
  callback();
}
greet("Nisha", () => console.log("Welcome!"));
```

Q13: Explain default parameters.

```
function greet(name = "Guest") {
  console.log("Hello " + name);
}
greet();         // Hello Guest
greet("John");   // Hello John
```

🏛 Section 4: Arrays & Objects (Q14–Q17)

Q14: How do you create and access an object?

```
let emp = {name: "Alice", age: 25};
console.log(emp.name);
console.log(emp["age"]);
```

Q15: How do you add and remove elements in an array?

```
let arr = [1,2,3];
arr.push(4);   // [1,2,3,4]
arr.pop();     // [1,2,3]
```

Q16: How to iterate over an array?

```
let nums = [1,2,3];
nums.forEach(n => console.log(n));
```

Q17: How to transform an array using map()?

```
let nums = [1,2,3];
let squares = nums.map(x => x*x); // [1,4,9]
```

🖱 Section 5: DOM & Events (Q18–Q20)

Q18: How to select an element in DOM?

```
let heading = document.getElementById("title");
```

Q19: How to change the text of an element?

```
document.getElementById("title").innerText = "New Title";
```

Q20: How to add a click event listener?

```
document.getElementById("btn").addEventListener("click", () => {
  alert("Button clicked!");
});
```

⚡ Section 6: ES6+ Features & Concepts (Q21–Q23)

Q21: What is destructuring?

```
const person = {name: "Alice", age: 25};
const {name, age} = person;
console.log(name, age);
```

Q22: What is a Promise?

An object that represents the eventual result of an asynchronous operation.

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Done!"), 1000);
});
promise.then(res => console.log(res));
```

Q23: What are template literals?

```
let name = "Nisha";
console.log(`Hello, ${name}!`);
```

✍ Section 7: Comments / Annotations & Advanced (Q24–Q25)

Q24: How do you add comments or annotations in JS?

```
// Single-line comment

/*
 Multi-line
 comment
*/

/**
 * Adds two numbers
 * @param {number} a
 * @param {number} b
 */
function add(a,b){ return a+b; }
```

Q25: What is hoisting?

JavaScript moves declarations (not initializations) to the top during compilation.

```
console.log(x); // undefined
var x = 10;
```

MULTITHREADING IN JAVA

Core Concepts

Q1: What is a thread in Java?

A thread is a lightweight subprocess — the smallest unit of execution within a program.
A Java process can have multiple threads running concurrently.

Q2: Difference between process and thread

Process	Thread
Independent execution	Part of a process
Has its own memory space	Shares memory of the process
Heavyweight	Lightweight
Created by OS	Created by the process

Q3: How do you create a thread in Java?

✔ Two ways:

1. **Extending `Thread` class**

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " +
Thread.currentThread().getName());
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // Starts a new thread
    }
}
```

2. **Implementing `Runnable` interface**

```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Task executed by " +
Thread.currentThread().getName());
    }
}

public class Test {
    public static void main(String[] args) {
```

```
        Thread t = new Thread(new MyTask());
        t.start();
    }
}
```

顯 Best practice: Use Runnable because Java supports multiple interfaces but only single inheritance.

Q4: What is the difference between start() and run()?

start()	run()
Creates a new thread and executes run() in it	Executes code in the current thread (no new thread)

Q5: What are the different thread states?

- **NEW → Thread object created but not started.**
- **RUNNABLE → Ready to run, waiting for CPU time.**
- **RUNNING → Actively executing.**
- **WAITING / TIMED_WAITING / BLOCKED → Temporarily inactive.**
- **TERMINATED / DEAD → Finished execution.**

Q6: What is thread priority?

A hint to the scheduler about which thread should run first (range 1–10).
Default = **`Thread.NORM_PRIORITY`** (5).

```
t1.setPriority(Thread.MAX_PRIORITY);
```

Q7: What is synchronization in Java?

Synchronization ensures that only one thread accesses a shared resource at a time, preventing data inconsistency.

Q8: How do you synchronize a method?

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }
}
```

Q9: How do you use a synchronized block?

```
synchronized(this) {
```

```
   // critical section
}
```

Q10: Difference between synchronized method and synchronized block

Synchronized Method	Synchronized Block
Locks the whole method	Locks only the critical section
Less efficient	More efficient

Q11: What is volatile keyword?

Ensures that the value of a variable is always read from main memory, not from a thread’s cache.
Used for variables accessed by multiple threads.

Q12: What is deadlock?

Deadlock occurs when two or more threads are waiting for each other’s resources and none can proceed.

Q13: Example of deadlock

```
class Test {
    public static void main(String[] args) {
        final String r1 = "Resource1";
        final String r2 = "Resource2";

        Thread t1 = new Thread(() -> {
            synchronized(r1) {
                System.out.println("Thread 1 locked r1");
                try { Thread.sleep(100);} catch(Exception e){}
                synchronized(r2) {
                    System.out.println("Thread 1 locked r2");
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized(r2) {
                System.out.println("Thread 2 locked r2");
                try { Thread.sleep(100);} catch(Exception e){}
                synchronized(r1) {
                    System.out.println("Thread 2 locked r1");
                }
            }
        });

        t1.start();
        t2.start();
    }
}
```

Q14: How to prevent deadlock?

- Avoid nested locks
- Use tryLock() from java.util.concurrent.locks
- Use proper lock ordering

Q15: What is the difference between sleep(), wait(), and join()?

Method	Belongs To	Releases Lock	Purpose
sleep()	Thread class	✗	Pause thread for given time
wait()	Object class	✓	Thread waits until notified
join()	Thread class	✗	Waits for another thread to finish

🧾 EXCEPTION HANDLING IN JAVA

Core Concepts

Q1: What is an exception in Java?

An exception is an event that disrupts the normal flow of the program.

Q2: Types of exceptions

Type	Description	Examples
Checked Exception	Checked at compile time	IOException, SQLException
Unchecked Exception	Occurs at runtime	NullPointerException, ArithmeticException
Error	Serious problems (not recoverable)	OutOfMemoryError

Q3: How do you handle exceptions?

Using try-catch:

```
try {
    int a = 5/0;
} catch(ArithmeticException e) {
    System.out.println("Cannot divide by zero");
}
```

Q4: What is the finally block?

A block that executes regardless of whether an exception occurs or not.
Usually used to release resources like database connections or file handles.

```
try {
   int a = 5/0;
} catch(Exception e) {
   System.out.println("Exception: " + e);
} finally {
   System.out.println("Always executes");
}
```

Q5: What is the difference between throw and throws?

throw	throws
Used to explicitly throw an exception	Declares exceptions a method can throw
Used inside a method	Declared in method signature

```
void checkAge(int age) {
    if(age < 18) throw new ArithmeticException("Not eligible");
}

void myMethod() throws IOException {
    // code that may throw IOException
}
```

Q6: What is the difference between checked and unchecked exceptions?

- **Checked: Must be handled at compile time (try/catch or declare with `throws`).**
- **Unchecked: Detected at runtime (no compile-time checking).**

Q7: Can we have multiple catch blocks?

Yes, to handle different types of exceptions.

```
try {
    int arr[] = new int[5];
    arr[6] = 10;
} catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Index error");
} catch(Exception e) {
    System.out.println("Other error");
}
```

Q8: What is a custom exception?

A user-defined exception that extends Exception or RuntimeException.

```
class AgeException extends Exception {
    AgeException(String msg) {
        super(msg);
    }
}

class Test {
    public static void main(String[] args) {
        try {
```

```
            int age = 15;
            if(age < 18) throw new AgeException("Underage!");
        } catch(AgeException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Q9: Can a finally block override a return statement in a try block?

No. The return value is determined by the last executed statement in finally.

Q10: What is the difference between final, finally, and finalize()?

Keyword	Purpose
final	Constant or prevent inheritance
finally	Executes cleanup code after try/catch
finalize()	Called by GC before object is removed

) Spring Security

Spring Security is a powerful authentication, authorization, and security framework for Java applications, mainly built on top of Spring.

Key Features:

- **Authentication → Verifies *who* the user is.**
- **Authorization → Checks *what* the user can access (roles, permissions).**
- **Password Security → Supports hashing (e.g., BCrypt).**
- **Protection against:**
    - **CSRF (Cross-Site Request Forgery)**
    - **Session fixation attacks**
    - **Clickjacking**
- **Integration with OAuth2, JWT, LDAP, SAML, etc.**

Example (Basic Authentication in Spring Boot):

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/public/**").permitAll()  // no auth required
            .antMatchers("/admin/**").hasRole("ADMIN") // only admin
```

```
            .anyRequest().authenticated()  // all others need login
            .and()
            .httpBasic();  // Basic Authentication
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.inMemoryAuthentication()
            .withUser("nisha")
            .password("{noop}password") // {noop} means no encoding
            .roles("USER")
            .and()
            .withUser("admin")
            .password("{noop}admin123")
            .roles("ADMIN");
    }
}
```

顯 This config:

- `/public/** → accessible by everyone`
- `/admin/** → only for ADMIN role`
- `all other APIs → need authentication`

---

2) `Encryption & Decryption of API Data`

Even with authentication, sometimes sensitive API data (like passwords, financial info, personal data) needs **`extra protection`** beyond HTTPS/TLS.

Options:

1. **HTTPS (Transport Layer Security)**
    - **By default, APIs should always use HTTPS (encrypts all requests & responses).**
    - **This protects data in *transit*.**
2. **Custom Encryption (for extra layer)**
    - **Encrypt request payload at the client side → send → decrypt at server side.**
    - **Useful when storing or transmitting very sensitive info.**

Example using AES Encryption:

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.util.Base64;

public class AESUtil {
    private static SecretKey secretKey;

    static {
        try {
            KeyGenerator keyGen = KeyGenerator.getInstance("AES");
            keyGen.init(128); // 128-bit key
```

```
            secretKey = keyGen.generateKey();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    // Encrypt
    public static String encrypt(String data) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        byte[] encrypted = cipher.doFinal(data.getBytes());
        return Base64.getEncoder().encodeToString(encrypted);
    }

    // Decrypt
    public static String decrypt(String encryptedData) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        byte[] decoded = Base64.getDecoder().decode(encryptedData);
        byte[] decrypted = cipher.doFinal(decoded);
        return new String(decrypted);
    }
}
```

Usage inside API:

```
@RestController
@RequestMapping("/secure")
public class SecureController {

    @GetMapping("/encrypt")
    public String encryptMessage(@RequestParam String message) throws Exception
{
        return AESUtil.encrypt(message);
    }

    @GetMapping("/decrypt")
    public String decryptMessage(@RequestParam String encrypted) throws
Exception {
        return AESUtil.decrypt(encrypted);
    }
}
```

顯 Client can send encrypted payload, server decrypts it.

✅ Best Practice:

- Always use HTTPS for transport.
- Use Spring Security for authentication/authorization.
- Use AES/RSA encryption if you want to encrypt payload fields (e.g., PAN, SSN).
- Use JWT (JSON Web Token) for secure stateless authentication.

Do you want me to show you a JWT + Spring Security example (most common in real APIs)?

You said:

global exception

ChatGPT said:

Perfect 👍 You're asking about Global Exception Handling in Spring Boot.

Instead of writing try-catch in every controller, Spring provides a clean way to handle exceptions globally using @ControllerAdvice + @ExceptionHandler.

Global Exception Handling in Spring Boot

Step 1: Create Custom Exception

```
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

Step 2: Create Global Exception Handler

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;

import java.util.Date;

@ControllerAdvice
public class GlobalExceptionHandler {

    // Handle specific exception
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<?> resourceNotFoundException(ResourceNotFoundException
ex, WebRequest request) {
        ErrorDetails errorDetails = new ErrorDetails(new Date(),
ex.getMessage(), request.getDescription(false));
        return new ResponseEntity<>(errorDetails, HttpStatus.NOT_FOUND);
    }

    // Handle global exceptions
    @ExceptionHandler(Exception.class)
    public ResponseEntity<?> globalExceptionHandler(Exception ex, WebRequest
request) {
        ErrorDetails errorDetails = new ErrorDetails(new Date(),
ex.getMessage(), request.getDescription(false));
        return new ResponseEntity<>(errorDetails,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Step 3: Create Error Response Model

```
import java.util.Date;
```

```
public class ErrorDetails {
    private Date timestamp;
    private String message;
    private String details;

    public ErrorDetails(Date timestamp, String message, String details) {
        super();
        this.timestamp = timestamp;
        this.message = message;
        this.details = details;
    }

    // getters
    public Date getTimestamp() { return timestamp; }
    public String getMessage() { return message; }
    public String getDetails() { return details; }
}
```

Step 4: Usage in Controller

```
@RestController
@RequestMapping("/api")
public class DemoController {

    @GetMapping("/user/{id}")
    public String getUser(@PathVariable int id) {
        if (id != 1) {
            throw new ResourceNotFoundException("User not found with id: " +
id);
        }
        return "User Found!";
    }
}
```

Example Output

Request → GET /api/user/5

Response →

```
{
  "timestamp": "2025-09-24T11:20:15.123+00:00",
  "message": "User not found with id: 5",
  "details": "uri=/api/user/5"
}
```

✔ Advantages:

- Centralized error handling
- No need to write try-catch in every controller
- Consistent error response format

Would you like me to also show you a global exception handler with @RestControllerAdvice and ResponseEntityExceptionHandler (the cleaner way used in real projects)?

You said:

interview question can be asked from the company like c-edge or tcs or infosys --topic on -->1)Spring security 2)encryption decryption of API and gobal exception. write all the possible questions with there proper answer with detail explaintion and with examples

ChatGPT said:

Great question 👍

If you're preparing for interviews with companies like C-Edge, TCS, Infosys, they usually test concepts + practical knowledge on topics like Spring Security, Encryption/Decryption in APIs, and Global Exception Handling.

I'll compile a list of possible interview questions with detailed answers + examples for each topic.

---

1) Spring Security

Q1. What is Spring Security and why is it used?

Answer:
Spring Security is a powerful framework that provides authentication, authorization, and protection against common attacks (CSRF, session fixation, clickjacking). It integrates seamlessly with Spring Boot and supports multiple authentication mechanisms like Basic Auth, JWT, OAuth2, LDAP, SAML.

🔹 Example:

- `/admin/** → only accessible to ADMIN`
- `/user/** → accessible to USER role`

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/user/**").hasAnyRole("USER", "ADMIN")
        .antMatchers("/public/**").permitAll()
        .anyRequest().authenticated()
        .and().httpBasic();
}
```

Q2. Explain Authentication vs Authorization in Spring Security.

Answer:

- **Authentication =** ***Who are you?*** **→ verifying identity (username/password, token, etc.)**
- **Authorization =** ***What can you access?*** **→ checking user's role/privileges**

🔹 Example:

- A user logs in with credentials → authentication.
- The same user tries to access /admin/dashboard but has role USER → denied → authorization.

Q3. What is CSRF protection in Spring Security?

Answer:
CSRF (Cross-Site Request Forgery) is when an attacker tricks a logged-in user into performing unwanted actions.
Spring Security enables CSRF tokens by default for state-changing requests (POST, PUT, DELETE).

🔹 Example:

When you send a form request, Spring Security expects a hidden token like:

```
<input type="hidden" name="_csrf" value="abc123xyz">
```

This ensures requests are genuine.

Q4. How does password encryption work in Spring Security?

Answer:
Spring Security uses PasswordEncoders like BCryptPasswordEncoder to hash passwords instead of storing plain text.

🔹 Example:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

- Password "12345" → stored as $2a$10$QvO3qf... (hashed).
- Even if DB leaks, passwords remain safe.

Q5. How do you implement JWT authentication in Spring Security?

Answer (high-level steps):

1. User logs in → server validates → generates JWT token.
2. Token is sent in response.
3. For every request → client sends token in header.
4. Server validates token → allows or denies access.

🔹 Example header:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR...
```

2) Encryption & Decryption in API

Q1. Why do we need encryption in API communication?

Answer:

- To protect sensitive data (passwords, credit card numbers, personal info).
- Even with HTTPS, some companies use payload-level encryption for extra security.
- Prevents MITM (Man-in-the-Middle) attacks.

Q2. What are common algorithms used for encryption in Java APIs?

Answer:

- **AES (Advanced Encryption Standard) → symmetric (same key for encrypt & decrypt).**
- **RSA (Rivest–Shamir–Adleman) → asymmetric (public/private keys).**
- **SHA-256/512 → hashing (one-way, for password storage).**

Q3. Example of AES Encryption & Decryption in Java API.

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.util.Base64;

public class AESUtil {
    private static SecretKey secretKey;

    static {
        try {
            KeyGenerator keyGen = KeyGenerator.getInstance("AES");
            keyGen.init(128); // 128-bit AES
            secretKey = keyGen.generateKey();
        } catch (Exception e) { throw new RuntimeException(e); }
    }

    public static String encrypt(String data) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        return
Base64.getEncoder().encodeToString(cipher.doFinal(data.getBytes()));
    }

    public static String decrypt(String encryptedData) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        return new
String(cipher.doFinal(Base64.getDecoder().decode(encryptedData)));
    }
}
```

Q4. What’s the difference between Hashing vs Encryption?

Answer:

- **Encryption = two-way (you can encrypt & decrypt).**
- **Hashing = one-way (cannot be decrypted, only compared).**
  顎 Example:
- **Password storage → use hashing (BCrypt, SHA).**
- **Secure message transfer → use encryption (AES, RSA).**

Q5. How does HTTPS secure APIs?

Answer:

- Uses TLS/SSL protocol for encryption.
- Data between client & server is encrypted with public/private keys.
- Prevents sniffing & tampering.

---

3) Global Exception Handling

Q1. What is Global Exception Handling in Spring Boot?

Answer:
Instead of writing try-catch inside every controller, we use @ControllerAdvice or @RestControllerAdvice to handle exceptions in a centralized way.

Q2. How do you implement a global exception handler?

Answer:

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGlobal(Exception ex) {
        return new ResponseEntity<>("Something went wrong: " + ex.getMessage(),
                                    HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Q3. Difference between @ControllerAdvice and @RestControllerAdvice?

Answer:

- `@ControllerAdvice → used with @ResponseBody for JSON responses.`
- `@RestControllerAdvice = @ControllerAdvice + @ResponseBody (shortcut).`

Q4. How do you customize error responses?

`Answer:`

By creating an error response model.

```
public class ErrorDetails {
    private Date timestamp;
    private String message;
    private String details;
    // constructor + getters
}
```

Then return ErrorDetails from @ExceptionHandler.

Q5. Can you extend Spring's ResponseEntityExceptionHandler?

Answer:

Yes. You can extend ResponseEntityExceptionHandler and override methods like handleMethodArgumentNotValid() to handle validation errors globally.

顯 Example:

```
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(
        MethodArgumentNotValidException ex,
        HttpHeaders headers, HttpStatus status, WebRequest request) {
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getFieldErrors().forEach(error ->
        errors.put(error.getField(), error.getDefaultMessage()));
    return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
}
```