1. Core Java

SECTION 1

1.What are the main features of Java?

- **Simple** – Easy syntax, similar to C/C++
- **Object-Oriented** – Everything is an object
- **Platform Independent** – WORA (Write Once, Run Anywhere)
- **Secure** – Bytecode verification, no pointer manipulation
- **Robust** – GC + strong memory management
- **Multithreaded** – Built-in thread support
- **Portable** – Code runs across systems
- **High Performance** – JIT compiler
- **Dynamic** – Supports dynamic loading of classes

2. Prove that Java is Platform Independent

- Java code is compiled to **bytecode** (`.class`)
- Bytecode is executed by JVM
- Every OS has its own JVM implementation
- Therefore same `.class` file runs on any OS → **Platform Independent**

3. Difference between JDK, JRE, JVM

JDK (Java Development Kit)

- Contains JRE + development tools
- Required for compiling programs
- Tools: `javac`, `javadoc`, `jdb`, etc.

JRE (Java Runtime Environment)

- Contains JVM + libraries
- Required to run Java applications

JVM (Java Virtual Machine)

- Executes bytecode
- Provides platform independence
- Performs JIT compilation, GC, security checks

4. Difference between == and `equals()`

- `==` → compares **reference** (addresses)
- `equals()` → compares **content/value**

5. Garbage Collection in Java

- Automatic memory cleanup by JVM
- Reclaims memory of objects not reachable
- Uses algorithms:
  - Mark and Sweep
  - Generational GC
  - G1 GC
- Cannot be forced, only suggested with `System.gc(`

6. Wrapper Classes

- Primitive → Object wrappers
- `byte → Byte`
- `short → Short`
- `int → Integer`
- `long → Long`
- `float → Float`
- `double → Double`
- `char → Character`
- `boolean → Boolean`

7. final vs finally vs finalize

final

- Variable → constant
- Method → cannot override
- Class → cannot inherit

finally

- Used in try-catch
- Always executes (except System.exit)

finalize()

- Called by GC before object destruction
- Deprecated in recent Java versions

8. Java String Pool

- Located in **Heap → String Constant Pool**
- Stores and reuses string literals
- Improves memory usage
- `"abc"` goes into pool
- `new String("abc")` creates new object in heap

9. Can we override static methods?

 No

- Static methods belong to **class**, not objects
- They can be **hidden**, not overridden

10. String vs StringBuffer

Feature	String	StringBuffer
Mutability	Immutable	Mutable
Performance	Slow	Faster
Thread-safety	Not Thread-safe	Thread-safe (synchronized)
Use Case	Read-only strings	Many modifications

SECTION 2

2. OOPs (Object-Oriented Programming)

1. What are the four pillars of OOPs?

1. Encapsulation

- Binding data + methods into a single unit (class)
- Achieved using private variables + public getters/setters

2. Inheritance

- One class acquires properties of another
- Types: Single, Multilevel, Hierarchical
- Promotes code reusability

3. Polymorphism

- Many forms → same method behaves differently
- Types: Compile-time (overloading), Runtime (overriding)

4. Abstraction

- Hiding internal implementation
- Achieved with **abstract classes** & **interfaces**

2. Explain method overloading and method overriding.

Method Overloading

- Same method name, different parameters
- occurs in same class
- Compile-time polymorphism

Method Overriding

- Same method name + parameters
- Child class overrides parent class method
- Runtime polymorphism
- Requires inheritance

3. Difference between abstraction and encapsulation

Abstraction	Encapsulation
Hides details	Hides data
Achieved via abstract class/interface	Achieved via classes & access modifiers
Focus on *what*	Focus on *how*

4. What is a constructor in Java? Types of constructors?

Constructor

- Special method called when an object is created
- Same name as class
- No return type

Types

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor** (not built-in, but can be created manually)

5. What is polymorphism in Java? Provide examples.

Types:

1. Compile-time Polymorphism (Overloading)

```
void sum(int a, int b)
void sum(double a, double b)
```

2. Runtime Polymorphism (Overriding)

```
class A { void show(){} }
class B extends A { void show(){} }
```

6. Difference between interface and abstract class

Feature	Interface	Abstract Class
Methods	abstract + default + static	abstract + concrete
Variables	public static final	any type
Multiple inheritance	Yes	No
Constructor	No	Yes
Use Case	Full abstraction	Partial abstraction

7. What is an inner class in Java?

A class defined **inside another class**.
Types:

1. **Member Inner Class**
2. **Static Nested Class**
3. **Local Inner Class**
4. **Anonymous Inner Class**

Used for event handling, grouping helper classes.

8. Access Modifiers (ALL)

Modifier	Access
Public	Everywhere
protected	Same package + subclasses
default (no modifier)	Same package
private	Within class only

9. protected vs Default

Feature	protected	default
Same class	YES	YES
Same package	YES	YES
Subclass (different package)	YES	NO
Outside package	NO	NO

SECTION 3

3. Exception Handling

1. What is an exception in Java?

- An **exception** is an unwanted event that disrupts program execution.
- It occurs at **runtime**.
- Everything is represented as an object of `Throwable`.

Types:

1. **Checked Exceptions** → checked at compile time
2. **Unchecked Exceptions** → occur at runtime

2. Difference between checked and unchecked exceptions

Checked	Unchecked
Checked at **compile time**	Checked at **runtime**
Must handle using try/catch or throws	Not mandatory to handle
Examples: IOException, SQLException	Examples: NullPointerException, ArithmeticException

3. What are try, catch, finally, throw, and throws used for?

try

- Code that may throw an exception.

catch

- Handles the exception.

finally

- Executes always (close resources).

throw

- Used to explicitly throw an exception.

throws

- Tells the calling method that this method might throw an exception.

4. Can we have multiple catch blocks for a single try block?

✔ **Yes**

- Used to handle different exception types.
- Must be ordered from **specific → general**.

5. What is a custom exception, and how do you create one?

User-defined exception for business rules.

Steps to create a Custom Exception:

1. Create a class extending `Exception` (checked) or `RuntimeException` (unchecked)
2. Create a constructor with message
3. Throw it using `throw`

Example:

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

```
}
```

6. Explain the concept of try-with-resources

- Introduced in Java 7
- Automatically closes resources
- Works only with classes implementing **AutoCloseable**

Example:

```
try (FileReader fr = new FileReader("a.txt")) {
    // code
}
```

7. throw vs throws

throw	throws
Used to explicitly throw exception	Used in method signature
Throws **single exception**	Can declare **multiple exceptions**
Within method	With method definition

8. How to Create a Custom Exception (Steps)

1. Create a new class
2. Extend `Exception` / `RuntimeException`
3. Add constructor
4. Use `throw` keyword
5. Handle using try/catch

SECTION 4

4. Multithreading

1. What is multithreading in Java and How to achieve it?

Multithreading

- Executing multiple threads (lightweight processes) **simultaneously**.
- Used for multitasking, performance, background jobs.

How to achieve multithreading?

Two ways:

1. **Extending Thread class**
2. Implementing Runnable interface

Difference between Runnable and Thread

Runnable	Thread
Interface	Class
Use when class already extends another	Use when not extending any class
run() only	start(), sleep(), interrupt() etc.
Better for shared resources	Less preferred

3. How do you create a thread in Java?

Method 1: Extending Thread

```
class A extends Thread {
    public void run() {
        System.out.println("Thread running");
    }
}
new A().start();
```

Method 2: Implementing Runnable

```
class A implements Runnable {
    public void run() {
        System.out.println("Thread running");
    }
}
Thread t = new Thread(new A());
t.start();
```

4. What are thread states in Java?

Thread lifecycle states:

1. **New**
2. **Runnable**
3. **Running**
4. **Blocked**
5. **Waiting**
6. **Timed Waiting**
7. **Terminated**

5. Explain synchronization and how to achieve it in Java.

Synchronization

- Prevents multiple threads from accessing shared resources simultaneously.
- Avoids race conditions.

Ways to achieve:

1. **Synchronized method**
2. **Synchronized block**
3. **Static synchronization**
4. **Lock interface (ReentrantLock)**

Example:

```
synchronized void display() { }
```

6. Difference between wait() and sleep()

wait()	sleep()
From Object class	From Thread class
Releases lock	Does not release lock
Used for inter-thread communication	Used to pause thread
Must be inside synchronized block	Can be anywhere

7. Thread Life Cycle

1. **New** → Thread created
2. **Runnable** → Ready to run
3. **Running** → Executing
4. **Blocked/Waiting** → Waiting for resource or signal
5. **Terminated** → Finished

8. List all the methods in Multithreading

Thread class methods:

- `start()`
- `run()`
- `sleep()`
- `yield()`
- `join()`
- `interrupt()`

- `isAlive()`
- `getName()`
- `setName()`
- `getPriority()`
- `setPriority()`
- `currentThread()`

Object class (thread-related)

- `wait()`
- `wait(long)`
- `notify()`
- `notifyAll()`

SECTION 5

5. Collections Framework

1. What is the difference between ArrayList and LinkedList?

Feature	ArrayList	LinkedList
Storage	Dynamic array	Doubly linked list
Access Speed	Faster (O(1))	Slower (O(n))
Insert/Delete	Slow (shifting)	Fast (node replacing)
Memory	Less	More
Best Use	Frequent access	Frequent insert/delete

2. How is HashMap different from Hashtable?

HashMap	Hashtable
Not synchronized	Synchronized
Faster	Slower
Allows null key & values	No null key/value
Introduced in Java 1.2	Legacy class

3. What is the difference between HashSet and TreeSet?

HashSet	TreeSet
No ordering	Sorted order
Backed by HashMap	Backed by TreeMap
Faster	Slower
Null allowed	Null NOT allowed

4. Explain the internal working of HashMap.

Working Steps:

1. Key's `hashCode()` is generated
2. HashMap calculates bucket index → `hash(key) % capacity`
3. Stores Entry as a **node** in bucket
4. If collision happens → uses **LinkedList** / **TreeNode**
5. If bucket size > 8 → converts to **Tree (Red-Black Tree)**

Retrieval:

- Key → hashCode → bucket → search by equals()

5. What is the difference between List, Set, and Map?

List	Set	Map
Stores duplicates	No duplicates	Key-value pairs
Ordered	Unordered (HashSet) / Sorted (TreeSet)	Key unique
Examples: ArrayList	Examples: HashSet	Examples: HashMap

6. How does a ConcurrentHashMap work?

- Thread-safe version of HashMap
- Uses **segment locking** (Java 7)
- Uses **CAS + bucket-level locking** (Java 8)
- No full map lock
- Faster than Hashtable

7. Explain Comparator and Comparable interfaces.

Comparable

- Natural sorting
- Implemented in class
- Method: `compareTo()`
- Only one sorting sequence

Comparator

- External sorting
- Implemented outside class
- Method: `compare()`
- Multiple sorting sequences possible

8. What is a legacy class? List all legacy classes in Collections.

Legacy classes existed before Java Collections Framework.

Legacy classes include:

- `Vector`
- `Stack`
- `Hashtable`
- `Enumeration`
- `Dictionary`
- `Properties`

SECTION 6

6. JDBC (Java Database Connectivity)

1. What is JDBC, and why is it used?

JDBC (Java Database Connectivity)

- A Java API to connect and interact with databases.
- Allows Java programs to execute SQL queries.

Used for:

- Connecting to DB
- Executing queries
- Retrieval and modification of data

2. Steps to connect to MySQL using JDBC

STEP 1: Load the Driver

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

STEP 2: Establish Connection

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/dbname", "root", "password");
```

STEP 3: Create Statement

```
Statement st = con.createStatement();
```

STEP 4: Execute Query

```
ResultSet rs = st.executeQuery("SELECT * FROM employee");
```

STEP 5: Process Results

```
while(rs.next()) {
    System.out.println(rs.getInt(1));
}
```

STEP 6: Close Connection

```
con.close();
```

3. Types of JDBC Drivers

1. **Type-1:** JDBC-ODBC Bridge
2. **Type-2:** Native API Driver
3. **Type-3:** Network Protocol Driver
4. **Type-4:** Thin Driver (Pure Java → most used for MySQL)

4. How do you handle transactions in JDBC?

Steps:

```
con.setAutoCommit(false);

PreparedStatement ps = con.prepareStatement("insert ...");
ps.executeUpdate();

con.commit(); // commit transaction
con.rollback(); // rollback on error
```

5. Why use PreparedStatement over Statement?

PreparedStatement advantages:

- Prevents **SQL Injection**
- Pre-compiled → faster
- Supports dynamic parameters (?)
- Better performance for repeated queries

6. How to implement batch processing in JDBC?

Used to execute multiple queries at once.

Example:

```
PreparedStatement ps = con.prepareStatement("insert into emp values(?,
?)");

ps.setInt(1, 1);
ps.setString(2, "John");
ps.addBatch();

ps.setInt(1, 2);
ps.setString(2, "Mike");
ps.addBatch();

ps.executeBatch();
```

7. Explain ResultSet and its types.

ResultSet:

- Represents table of data returned from SQL query.

Types:

1. **TYPE_FORWARD_ONLY** → forward only
2. **TYPE_SCROLL_INSENSITIVE** → scroll but no DB change reflection
3. **TYPE_SCROLL_SENSITIVE** → scroll & reflect DB changes

Concurrency:

- **CONCUR_READ_ONLY**
- **CONCUR_UPDATABLE**

SECTION 7

7. Spring Framework

1. What is the Spring Framework?

Spring is a **lightweight, open-source, enterprise-level framework** that provides infrastructure support for building Java applications.

Key Features:

- Lightweight
- IoC (Inversion of Control)
- Dependency Injection
- AOP (Aspect Oriented Programming)
- Transaction management

- Spring MVC, Spring Boot, Security, etc.

2. Explain Dependency Injection (DI) and Inversion of Control (IoC).

IoC (Inversion of Control)

- Instead of the programmer creating objects,
  the **Spring Container** creates and manages object life cycle.

DI (Dependency Injection)

- Dependencies are **injected** rather than created manually using `new`.

Types of DI

1. **Constructor Injection**
2. **Setter Injection**
3. **Field Injection (with @Autowired)**

3. Annotation-based and XML-based Configuration

Annotation-based

- Uses annotations like `@Component`, `@Service`, `@Autowired`
- Configured using `@ComponentScan`

XML-based

- Uses `applicationContext.xml`
- Beans defined in `<bean>` tags

Setter Injuction (PRPOERTY NAME=)

COnstructor (CONSTRUCTOR-ARGS)

Autowired (AUTOMATICALLY INJECTS DEPENDENCIES CAN BE CONSTUROS, GETTERSERRER FIELD)

7. What are Spring Beans and Bean Scopes?

Spring Bean

- Object managed by Spring IoC container.

Bean Scopes

Scope	Meaning
singleton	One instance per container
prototype	New instance every time
request	Per HTTP request
session	Per HTTP session
application	Servlet context scope
websocket	Per WebSocket session

8. What is Spring ApplicationContext?

- Central interface of the Spring IoC container
- Responsible for:
  - Creating beans
  - Managing beans
  - Dependency injection

Example:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("config.xml");
```

9. Difference between @Component, @Service, and @Repository

Annotation	Use
@Component	Generic bean
@Service	Service layer
@Repository	DAO layer (provides DB exception translation)

10. How does Spring manage transactions?

Using `@Transactional` annotation.

Features:

- Rollback on exception
- Commit on success
- Declarative transaction management

Example:

```
@Transactional
public void updateData() {
    // operations
}
```

8. Hibernate /JPA

1. What is Hibernate?

Hibernate is an **ORM (Object Relational Mapping) framework** that maps Java objects to database tables.

Features

- Eliminates JDBC boilerplate
- Auto table creation
- Caching support
- HQL (Hibernate Query Language)
- Lazy Loading
- Supports different databases

Explain the Hibernate architecture.

Create configuration

- **SessionFactory** → Creates Sessions
- **Session** → Talks to DB
- **Transaction** → Manages commit/rollback
- **Query** → Fetch/update data
- **Persistent Objects** → POJOs mapped to tables

2. What is ORM (Object Relational Mapping)?

ORM is a technique that maps **Java classes → Database tables**
and **Java object fields → Table columns**.

Example:
Class → Employee
Table → employee
Field → name
Column → name

3. What is a Hibernate Configuration File?

`hibernate.cfg.xml`

Contains →

- DB connection setup
- Dialect
- Mapping resources

Example:

```
<property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
<property name="hibernate.connection.username">root</property>
<property
name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
```

4. What is SessionFactory and Session?

SessionFactory

- Heavy-weight object
- Created once per application
- Thread-safe
- Used to create sessions

Session

- One DB connection
- Lightweight
- ‘Not thread-safe

6. Hibernate Object States

State	Meaning
Transient	Object not associated with DB
Persistent	Associated with Hibernate session
Detached	Session closed, object exists
Removed	Scheduled for deletion

What is the role of Hibernate caching?

Explain HQL (Hibernate Query Language).

- HQL looks similar to SQL but works with **Java classes and objects**, not tables.
- Instead of table names, you use **entity (class) names**.
- It helps fetch, insert, update, and delete data in an object-oriented way.

Example:

```
Query q = session.createQuery("from Employee");
```

What are the advantages of using Hibernate over JDBC?

Automatic sql, built in caching, less code

SECTION 9

9. Spring MVC

1. What is Spring MVC, and how does it work?

Short Notes

Spring MVC is a **web framework** used to build Java-based web applications using the **Model–View–Controller** architecture.

Flow

1. Client sends request
2. **DispatcherServlet** receives request
3. It finds the **Controller**
4. Controller returns **Model + View name**

5. ViewResolver finds actual view (JSP/HTML/Thymeleaf)
6. Response sent back

2. What is DispatcherServlet in Spring MVC?

Short Notes

DispatcherServlet is the **front controller**.
It handles every incoming HTTP request.

Responsibilities

- Mapping requests to controllers
- Calling view resolver
- Handling exceptions
- Managing web application flow

3. How does request handling work in Spring MVC?

Flow Diagram

```
Request → DispatcherServlet → HandlerMapping → Controller Method
→ ModelAndView → ViewResolver → Response
```

4. What is the role of @Controller annotation?

Short Answer

- Marks a class as a Spring MVC controller
- Used to handle **web page-based requests** (returns views)

Example:

```
@Controller
public class HomeController {
    @GetMapping("/home")
    public String home() {
        return "home.jsp";
    }
}
```

5. What is ModelAndView in Spring MVC?

Short Notes

Used to return:

- View name
- Model data

Example:

```
ModelAndView mv = new ModelAndView("home");
mv.addObject("name", "John");
return mv;
```

6. Explain @RequestMapping annotation

Short Notes

Maps a URL to a controller method.

Example:

```
@RequestMapping("/login")
public String login() {
    return "login.jsp";
}
```

7. How can you validate form inputs in Spring MVC?

Steps

1. Add validation dependency
2. Use Bean Validation annotations
3. Use `@Valid` in controller
4. Use `BindingResult` to check errors

Example:

```
public class User {
    @NotNull
    @Size(min = 3)
    private String name;
}
@PostMapping("/save")
public String save(@Valid User user, BindingResult result) {
    if (result.hasErrors()) {
        return "form.jsp";
    }
    return "success.jsp";
}
```

8. Difference between @Controller and @RestController

Feature	@Controller	@RestController
Purpose	Web pages (views)	REST APIs
Returns	JSP/HTML	JSON/XML
Contains	Requires @ResponseBody	Auto JSON
Use case	MVC web apps	REST services

9. JPA vs Hibernate

JPA	Hibernate
A specification	Implementation of JPA
Provides rules	Provides actual code
Interfaces only	Has classes, API
Example: EntityManager	Example: Session

10. Difference between Controller and Service Layer

Layer	Purpose
Controller	Handles HTTP requests
Service	Business logic, processing
DAO/Repository	Database operations

11. What does @SpringBootApplication do?

It is a combination of 3 annotations:

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
```

Meaning:

- Enables auto configuration
- Scans components
- Creates Spring Boot application

12. What all annotations do I have in Spring (important)?

Common Spring Boot annotations

- @Component
- @Service
- @Repository
- @Controller
- @RestController
- @Autowired
- @Qualifier
- @Bean
- @Configuration
- @ComponentScan
- @SpringBootApplication
- @GetMapping / @PostMapping / @PutMapping / @DeleteMapping
- @RequestParam
- @PathVariable
- @ResponseBody
- @RequestBody

- @Entity
- @Table
- @Id
- @GeneratedValue
- @ManyToOne
- @OneToMany
- @OneToOne
- @ManyToMany
- @JoinColumn

13. @Id is for Primary Key. What annotation is for Foreign Key?

Answer

`@JoinColumn` — used to define **foreign key column**

Example:

```
@ManyToOne
@JoinColumn(name = "dept_id")
private Department department;
```

10. Spring Boot

1. What is Spring Boot, and why is it used?

Short Points

- Spring Boot is used to **build stand-alone, production-ready Spring applications**
- Removes the need for XML configuration
- Provides **embedded servers** (Tomcat/Jetty/Netty)
- Provides **auto-configuration**
-

2. Advantages of Spring Boot

- No XML configuration
- Embedded server support
- Auto-configuration
- Starter dependencies
- Actuator monitoring
- Faster development
- Easy integration with microservices
- Production-ready features

3. Explain the Spring Boot architecture

Layers

1. **Presentation Layer**
   - Controllers (REST endpoints)
2. **Business Layer**
   - Services (business logic)
3. **Persistence Layer**
   - Repositories (JPA/Hibernate)
4. **Models**
   - Entity classes
5. **Spring Boot Layer**
   - Autoconfiguration
   - Embedded server
   - Starters

Flow

Client → Controller → Service → Repository → DB → Response

4. What is the role of application.properties or application.yml?

Short Notes

Used for configuring application settings such as:

- Server port
- DB connection
- Logging
- JPA/Hibernate settings
- Custom application variables

Example (properties)

```
server.port=8081
spring.datasource.username=root
```

5. How do you create a Spring Boot application using Spring Initializr?

Steps

1. Go to start.spring.io
2. Select:

    - Maven/Gradle
    - Java
    - Spring Boot version
3. Add dependencies:
    - Spring Web
    - Spring JPA
    - MySQL Driver
    - Lombok
4. Generate project
5. Import into IDE
6. Run the `main` class

6. Difference between @SpringBootApplication and @ComponentScan

@SpringBootApplication

Combination of:

- @Configuration
- @EnableAutoConfiguration
- @ComponentScan

@ComponentScan

- Tells Spring where to scan for components
- Used when packages are outside default scope

Example:

```
@ComponentScan("com.test.controllers")
```

7. What are the various Spring Boot starters?

Common Starters

Starter	Purpose
spring-boot-starter-web	Web / REST APIs
spring-boot-starter-data-jpa	Hibernate + JPA
spring-boot-starter-security	Spring Security
spring-boot-starter-test	JUnit + Mockito
spring-boot-starter-thymeleaf	Thymeleaf UI

Starter	Purpose
spring-boot-starter-webflux	Reactive APIs
spring-boot-starter-validation	Bean validation

8. How do you create a custom banner in Spring Boot?

Steps

1. Create file:
   `src/main/resources/banner.txt`
2. Add ASCII text

Example:

```
  Welcome to My App!
```

Property (optional):

```
spring.main.banner-mode=off
```

11. Spring Web / Spring REST

1. How do you create a RESTful API using Spring Boot?

Steps

1. Create Spring Boot project
2. Add dependencies:
   - spring-boot-starter-web
3. Create `@RestController`
4. Create API methods using
   - @GetMapping
   - @PostMapping
   - @PutMapping
   - @DeleteMapping

Example

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

    @GetMapping("/{id}")
    public String getUser(@PathVariable int id) {
        return "User = " + id;
    }
}
```

2. Explain the @RestController annotation

Short Notes

- Combines **@Controller + @ResponseBody**
- Returns **JSON/XML** instead of views
- Used for REST APIs

Example:

```
@RestController
public class ProductController {}
```

3. Difference between @RequestBody and @ResponseBody

Annotation	Purpose
@RequestBody	Converts JSON → Java object (input)
@ResponseBody	Converts Java object → JSON (output)
Included in	@RestController (automatically)

Example:

```
@PostMapping("/save")
public String saveUser(@RequestBody User user) {
    return "Saved " + user.getName();
}
```

4. What is the use of @PathVariable and @RequestParam?

@PathVariable

Values from URL path
Example URL: `/user/10`

```
@GetMapping("/user/{id}")
public String getUser(@PathVariable int id) { }
```

@RequestParam

Query parameters
Example URL: `/search?name=John`

```
@GetMapping("/search")
public String search(@RequestParam String name) { }
```

5. How do you handle CORS in Spring Boot?(cross origin resource sharing)

Method 1: Controller level

```
@CrossOrigin(origins = "http://localhost:3000")
@GetMapping("/data")
public String getData() { return "OK"; }
```

Method 2: Global Level

Create a WebConfig class:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**").allowedMethods("*");
    }
}
```

6. How do you manage pagination & sorting in Spring REST APIs?

Using Spring Data Pageable

Example

```
@GetMapping("/users")
public Page<User> getUsers(Pageable pageable) {
    return userRepository.findAll(pageable);
}
```

URL Example

```
/users?page=0&size=5&sort=name,asc
```

7. HTTP Methods (@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping)

@GetMapping

Retrieve data

```
@GetMapping("/user")
```

@PostMapping

Create new resource

```
@PostMapping("/user")
```

@PutMapping

Update existing resource (full update)

```
@PutMapping("/user/{id}")
```

@PatchMapping

Partial update

```
@PatchMapping("/user/{id}")
```

@DeleteMapping

Delete resource

```
@DeleteMapping("/user/{id}")
```

12. Spring Exception Handling

1. What are the different ways to handle exceptions in Spring Boot?

Ways to handle exceptions

1. **Try–Catch inside method**
2. **@ExceptionHandler** (Method level)
3. **@ControllerAdvice** (Global exception handling)
4. **ResponseStatusException**
5. **Custom Exceptions**
6. **ErrorController (optional)**

2. How do you use @ControllerAdvice for global exception handling?

Short Notes

- Used to handle exceptions globally
- Applies to all controllers
- Works with `@ExceptionHandler` inside it

Example

```
@ControllerAdvice
public class GlobalExceptionHandler {
```

```
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                             .body(ex.getMessage());
    }
}
```

3. Explain the @ExceptionHandler annotation

Short Notes

- Used to handle specific exceptions
- Placed inside `@Controller` or `@ControllerAdvice`

Example

```
@ExceptionHandler(NullPointerException.class)
public String handleNull(NullPointerException ex) {
    return "Null pointer occurred!";
}
```

4. How do you return a custom response for an exception in Spring?

Using custom error response class

```
public class ErrorResponse {
    private String message;
    private Date timestamp;
}
```

ControllerAdvice

```
@ControllerAdvice
public class GlobalHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleException(UserNotFoundException ex) {

        ErrorResponse error = new ErrorResponse();
        error.setMessage(ex.getMessage());
        error.setTimestamp(new Date());

        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }
}
```

5. Creating a Custom Exception — Steps

Step 1: Create custom exception class

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

Step 2: Throw the exception in service

```
if(user == null) {
    throw new UserNotFoundException("User not found with ID " + id);
}
```

Step 3: Handle it in @ControllerAdvice

(Shown above)

13. Spring Boot Mockito

Junit v/s Mockito

1. What is Mockito?

Short Points

- Mockito is a **mocking framework** for unit testing in Java.
- Used to create **dummy objects** (mocks) for testing service, DAO, controller.
- Helps in **isolating layers** → test only one layer at a time.

2. Why do we use Mockito?

Short Points

1. To avoid calling actual DB/API.
2. To test only the business logic.
3. Faster + more reliable testing.
4. Helps verify method calls.
5. Allows stubbing (predefined return values).

3. Important Mockito Annotations

Annotation	Use
`@Mock`	Creates mock object
`@InjectMocks`	Injects mock into target class
`@ExtendWith(MockitoExtension.class)`	Enables Mockito in JUnit 5
`when()`	Stubbing methods
`verify()`	Check if method was called

4. Basic Mockito Test Example

Service Class

```
@Service
public class UserService {

    @Autowired
    private UserRepository repo;

    public User getUser(int id) {
        return repo.findById(id).orElse(null);
    }
}
```

Mockito Test (JUnit 5)

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserRepository repo;

    @InjectMocks
    private UserService service;

    @Test
    void testGetUser() {
        User user = new User(1, "John");

        when(repo.findById(1)).thenReturn(Optional.of(user));

        User result = service.getUser(1);

        assertEquals("John", result.getName());
        verify(repo, times(1)).findById(1);
    }
}
```

Stubbing is a concept used in **software testing**, especially in **unit testing** and **test-driven development**.

Simple definition

A **stub** is a *dummy piece of code* that **replaces a real function, class, or module** during testing.
It returns **predefined responses** so that you can test your code **without depending on external systems**.

Why use stubs?

- The real component is **not yet implemented**
- The real component is **slow**, e.g., database calls, network calls
- The real component has **side effects**, e.g., sending emails
- You want a **controlled, predictable** test environment

❓ 1. Difference between @Mock and @InjectMocks

@Mock

- Creates a **mock object** of a class or interface.
- This mock does **not execute real code** — it returns default or stubbed values.
- Used for **dependencies** in unit tests.

@InjectMocks

- Creates an **instance of the class being tested**.
- Automatically **injects the mocks** (created using `@Mock`) into it via:
  - constructor injection
  - setter injection
  - field injection

Example

```
@Mock
private UserRepository userRepository;  // Dependency is mocked

@InjectMocks
private UserService userService;        // Real object with mocks injected
```

`userService` is real.
`userRepository` is fake (mocked).
Mockito injects the mock into the service.

2. How to write unit tests for Spring Boot services using Mockito

Steps:

1. Use `@ExtendWith(MockitoExtension.class)`
2. Mock dependencies with `@Mock`
3. Inject them into the service with `@InjectMocks`
4. Stub behavior using `when(...)`
5. Verify behavior using `verify(...)`

✔ Example

Service to test:

```
@Service
public class UserService {
    @Autowired
    private UserRepository repo;
```

```
    public User getUser(int id) {
        return repo.findById(id).orElse(null);
    }
}
```

Unit test using Mockito:

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {

    @Mock
    private UserRepository repo;

    @InjectMocks
    private UserService userService;

    @Test
    void testGetUser() {
        User user = new User(1, "Alice");

        when(repo.findById(1)).thenReturn(Optional.of(user));

        User result = userService.getUser(1);

        assertEquals("Alice", result.getName());
        verify(repo, times(1)).findById(1);
    }
}
```

3. What are Mockito.when() and Mockito.verify() used for?

Mockito.when()

Used to **stub/mock behavior** — define what the mock should return.

Example:

```
when(repo.findById(1)).thenReturn(Optional.of(user));
```

Meaning:
"If `repo.findById(1)` is called, return the given user."

Mockito.verify()

Used to **check whether a method was called**, how many times, and with what arguments.

Example:

```
verify(repo, times(1)).findById(1);
```

Meaning:
"Ensure this method is called exactly once."

4. How to mock dependencies using @MockBean in Spring Boot

@MockBean

- Used in **Spring Boot test classes** with `@SpringBootTest` or `@WebMvcTest`.
- Replaces a **Spring bean** in the ApplicationContext with a mock.
- Useful when the test loads the Spring context.

✔ Example

```
@SpringBootTest
class UserServiceIntegrationTest {

    @Autowired
    private UserService userService;  // real bean

    @MockBean
    private UserRepository repo;      // replaced in Spring context

    @Test
    void testGetUser() {
        User user = new User(1, "John");

        when(repo.findById(1)).thenReturn(Optional.of(user));

        User result = userService.getUser(1);

        assertEquals("John", result.getName());
    }
}
```

Difference from `@Mock`:

- `@Mock` → Mockito mock, does NOT load Spring context
- `@MockBean` → Spring Boot creates a mock and injects it into Spring context

14. Microservices

1. What are microservices, and how do they differ from monolithic architecture?

Microservices

A microservices architecture breaks an application into **small, independent services**, each responsible for a **single business capability**.

Characteristics:

- Independently deployable
- Loosely coupled
- Owns its **own database**
- Can be built using different languages/tech stacks
- Scales horizontally

Monolithic Architecture

A monolith is a **single large application** where:

- All modules are tightly coupled
- Shares the same database
- Deployments affect the entire application
- Scaling requires scaling the whole application

Key Differences

Feature	Monolithic	Microservices
Deployment	Single unit	Each service independently
Scalability	Whole app	Per service
Tech Stack	Same for all modules	Different per service
Fault Isolation	Low	High
Database	Shared	Separate per service

2. Role of Service Discovery and Registration in Microservices

In microservices, services often run on dynamic hosts/ports (containers, scaling, etc.).
To communicate, services need to **discover each other automatically**.

Service Registration

- Each service **registers its IP and port** with a *service registry*.
- Example: **Eureka Server (Spring Cloud Netflix)**

Service Discovery

Services **query the registry** to get the location of other services.

Two Types:

1. **Client-side discovery** → e.g., Eureka
2. **Server-side discovery** → e.g., Kubernetes, AWS ELB

3. How do you implement inter-service communication?

There are two main approaches:

A) Synchronous (HTTP)

1. **RestTemplate** (deprecated, older)
2. **WebClient** (reactive, recommended)
3. **Feign Client** (declarative, common in microservices)

Example using **Feign**:

```
@FeignClient(name = "order-service")
public interface OrderClient {
    @GetMapping("/orders/{id}")
    OrderDto getOrder(@PathVariable int id);
}
```

B) Asynchronous (Messaging)

- Kafka
- RabbitMQ
- JMS
- Event-driven communication

4. Role of API Gateway in Microservices

An API Gateway is the **entry point** for all clients.

Responsibilities:

- Routing requests to services
- Load balancing
- Authentication & authorization
- Rate limiting
- Caching
- Request/response transformation
- Logging & monitoring

Examples:

- Spring Cloud Gateway
- Zuul
- Kong
- NGINX

Why needed?

Without a gateway, clients must call multiple microservices directly → complex, insecure.

5. Explain the Circuit Breaker Pattern

Used to **prevent cascading failures** when a service is down or slow.

How it works:

- If a service fails repeatedly, the circuit *opens*.
- Further calls return immediately with fallback logic.
- After some time, it goes to **half-open** → test requests.
- If recovered → back to **closed**.

Tools:

- Resilience4j (recommended)
- Hystrix (deprecated)

Example using Resilience4j:

```
@CircuitBreaker(name = "orderService", fallbackMethod = "fallbackMethod")
public Order getOrder(int id) {
    return orderClient.getOrder(id);
}

public Order fallbackMethod(int id, Exception e) {
    return new Order(id, "Default Order");
}
```

6. How do you manage configuration in microservices using Spring Cloud Config?

Spring Cloud Config provides:

- **Centralized configuration server**
- Externalized properties (Git, GitHub, file system)
- Runtime refresh using **/actuator/refresh**
- Version control

Steps:

1. Create Config Server

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {}
```

application.yml

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/myrepo/config-repo
```

2. Client service uses Config Server

```
bootstrap.properties:

spring.application.name=payment-service
spring.cloud.config.uri=http://localhost:8888
```

3. Access configuration

Files in Git:

```
payment-service.yml
payment-service-dev.yml
payment-service-prod.yml
```

4. Refresh configuration

Enable:

```
@RefreshScope
public class PaymentServiceConfig { ... }
```

Call endpoint:

```
POST /actuator/refresh
```

15. RestTemplate

1. What is RestTemplate in Spring?

RestTemplate is a Spring-provided **synchronous HTTP client** used to make REST API calls.

Key features:

- Performs HTTP operations: GET, POST, PUT, DELETE, etc.
- Converts JSON ↔ Java objects using HttpMessageConverters.
- Part of **Spring Web** module.

□□ **Important Note**
RestTemplate is **deprecated for future development**.
Spring recommends using **WebClient** (from Spring WebFlux) for new applications.

2. How do you make HTTP GET and POST calls using RestTemplate?

✔ A) HTTP GET Example

```
RestTemplate restTemplate = new RestTemplate();
```

```
String url = "http://localhost:8080/api/users/1";

User user = restTemplate.getForObject(url, User.class);
```

Or using `getForEntity`:

```
ResponseEntity<User> response = restTemplate.getForEntity(url, User.class);
User user = response.getBody();
```

✔ B) HTTP POST Example

```
RestTemplate restTemplate = new RestTemplate();

String url = "http://localhost:8080/api/users";

User newUser = new User("John", "john@gmail.com");

User createdUser = restTemplate.postForObject(url, newUser, User.class);
```

Or using `postForEntity`:

```
ResponseEntity<User> response = restTemplate.postForEntity(url, newUser,
User.class);
User createdUser = response.getBody();
```

? Common RestTemplate Methods

Method	Description
`getForObject`	GET, returns body
`getForEntity`	GET, returns full response (headers + body + status)
`postForObject`	POST, returns body
`postForEntity`	POST, returns response
`put`	PUT request
`delete`	DELETE request
`exchange`	Fully customized HTTP request

3. How do you handle errors with RestTemplate?

You can use:

✔ A) Custom `ResponseErrorHandler`

```
public class CustomErrorHandler implements ResponseErrorHandler {

    @Override
    public boolean hasError(ClientHttpResponse response) throws IOException
{
        return response.getStatusCode().isError();
    }

    @Override
    public void handleError(ClientHttpResponse response) throws IOException
{
        throw new RuntimeException("Error: " + response.getStatusCode());
    }
}
```

Register it:

```
RestTemplate restTemplate = new RestTemplate();
restTemplate.setErrorHandler(new CustomErrorHandler());
```

✔ B) Handle exceptions using try–catch

```
try {
    User user = restTemplate.getForObject(url, User.class);
} catch (HttpClientErrorException e) {
    System.out.println("Client Error: " + e.getStatusCode());
} catch (HttpServerErrorException e) {
    System.out.println("Server Error: " + e.getStatusCode());
}
```

✔ Common Exceptions RestTemplate throws

- `HttpClientErrorException` (4xx errors)
- `HttpServerErrorException` (5xx errors)
- `ResourceAccessException` (connection issues)

4. What are the alternatives to RestTemplate?

1. WebClient (Recommended)

- Part of Spring WebFlux
- Supports **asynchronous + reactive** programming
- Future replacement for RestTemplate

Example:

```
WebClient client = WebClient.create();

User user = client.get()
        .uri("http://localhost:8080/api/users/1")
        .retrieve()
        .bodyToMono(User.class)
        .block();
```

---

2. Feign Client (Declarative REST Client)

Used in **Microservices (Spring Cloud)**.

```
@FeignClient(name = "user-service")
public interface UserClient {
    @GetMapping("/users/{id}")
    User getUser(@PathVariable int id);
}
```

---

3. OkHttp Client

Fast and lightweight HTTP client by Square.

---

4. Apache HttpClient

Common alternative but requires more boilerplate compared to RestTemplate/WebClient.

16. WebClient

1. What is WebClient in Spring?

WebClient is the **non-blocking, asynchronous** HTTP client introduced in **Spring WebFlux**.

Key features:

- Supports **asynchronous** and **reactive** data streams
- Non-blocking I/O (better performance under high load)
- Can also make blocking calls if needed (`.block()`)
- Replacement for **RestTemplate**

Package:

`org.springframework.web.reactive.function.client.WebClient`

3. How does WebClient differ from RestTemplate?

Feature	RestTemplate	WebClient
Architecture	Blocking	Non-blocking (reactive)
Performance	Thread-per-request (less scalable)	Event-loop model (high concurrency)
API Style	Imperative	Reactive (Mono/Flux)
Future Support	Deprecated for future	Recommended
Suitable For	Simple synchronous apps	High-performance, real-time, microservices

Summary:

- **RestTemplate = synchronous**
- WebClient = asynchronous + reactive

3. How do you make asynchronous calls using WebClient?

✔ Example: Asynchronous GET Request

```
WebClient client = WebClient.create();

Mono<User> userMono = client.get()
        .uri("http://localhost:8080/api/users/1")
        .retrieve()
        .bodyToMono(User.class);

// async: register callback
userMono.subscribe(user -> {
    System.out.println("Received: " + user.getName());
});
```

✔ No blocking → returns immediately
✔ `subscribe()` executes asynchronously

✔ Example: Asynchronous POST Request

```
Mono<User> response = client.post()
        .uri("http://localhost:8080/api/users")
        .bodyValue(new User("John", 25))
        .retrieve()
        .bodyToMono(User.class);

response.subscribe(result -> System.out.println("Saved: " + result));
```

Asynchronous streaming with Flux

```
Flux<User> userStream = client.get()
        .uri("http://localhost:8080/api/users/stream")
        .retrieve()
        .bodyToFlux(User.class);
```

```
userStream.subscribe(System.out::println);
```

4. Explain how to use WebClient with Spring WebFlux

A) Add dependency

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

B) Create a WebClient Bean

```
@Configuration
public class WebClientConfig {
    @Bean
    public WebClient webClient() {
        return WebClient.builder().build();
    }
}
```

C) Inject and use WebClient in a Service

```
@Service
public class UserService {

    @Autowired
    private WebClient webClient;

    public Mono<User> getUser(int id) {
        return webClient.get()
                .uri("http://localhost:8080/api/users/{id}", id)
                .retrieve()
                .bodyToMono(User.class);
    }
}
```

✔ Controller (Reactive)

```
@RestController
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/users/{id}")
    public Mono<User> getUser(@PathVariable int id) {
        return userService.getUser(id);
    }
}
```

Here:

- WebClient returns a **Mono**
- Controller returns the Mono directly
- WebFlux handles request asynchronously

17. Feign Client

1. What is Feign Client, and why is it used in microservices?

Feign Client

Feign is a **declarative REST client** provided by Spring Cloud.

Instead of writing RestTemplate or WebClient code, you only write an **interface**, and Feign automatically generates the implementation.

✔ Why Feign is used in microservices?

- Makes **inter-service communication easier**
- Reduces boilerplate HTTP code (no more RestTemplate)
- Integrates with:
  - **Service Discovery** (Eureka)
  - **Load Balancing** (Ribbon / Spring Cloud LoadBalancer)
  - **Circuit Breakers** (Hystrix or Resilience4j)
- Supports **fallback logic**

Feign simplifies code dramatically:

Without Feign → write full HTTP code
With Feign → just write an interface

2. How do you create a Feign Client in Spring Boot?

Step 1 — Add dependency (Spring Cloud OpenFeign)

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Step 2 — Enable Feign in main class

```
@SpringBootApplication
@EnableFeignClients
public class OrderServiceApplication {}
```

Step 3 — Create Feign Client Interface

```
@FeignClient(name = "user-service")
public interface UserClient {

    @GetMapping("/users/{id}")
    UserDto getUser(@PathVariable("id") int id);
}
```

Step 4 — Use Feign Client in a service

```
@Service
public class OrderService {
```

```
    @Autowired
    private UserClient userClient;

    public OrderResponse createOrder(int userId) {
        UserDto user = userClient.getUser(userId);
        return new OrderResponse(user.getName(), "Order Created");
    }
}
```

3. How does Feign Client handle load balancing?

Feign integrates with **Spring Cloud LoadBalancer** (Ribbon was used earlier but removed).

How it works:

1. You specify service name in `@FeignClient(name="user-service")`.
2. Feign asks the **Service Registry** (Eureka) for all instances:
3. `user-service → [instance1, instance2, instance3]`
4. Load Balancer picks an instance using a strategy:
    - Round Robin (default)
    - Random
    - Weighted

Example:

If `user-service` has 3 instances, Feign distributes calls like:

```
Call 1 → instance1
Call 2 → instance2
Call 3 → instance3
Call 4 → instance1
```

 No need to configure load balancing manually—Feign does it automatically.

4. Explain how to use Feign Client with Hystrix for fault tolerance

 Note: **Hystrix is deprecated**, but still commonly asked in interviews.
(Spring recommends **Resilience4j** now.)

✔ Step 1 — Enable Hystrix

Add dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

Enable Hystrix:

```
@SpringBootApplication
@EnableFeignClients
@EnableHystrix
public class OrderServiceApplication {}
```

✔ Step 2 — Add fallback class

```
@Component
public class UserClientFallback implements UserClient {
    @Override
    public UserDto getUser(int id) {
        return new UserDto(id, "Default User", "N/A");
    }
}
```

✔ Step 3 — Configure Feign Client with fallback

```
@FeignClient(
    name = "user-service",
    fallback = UserClientFallback.class
)
public interface UserClient {
    @GetMapping("/users/{id}")
    UserDto getUser(@PathVariable("id") int id);
}
```

✔ How Hystrix works here

If `user-service` is:

- Down
- Slow
- Throws exception

Then:

- Circuit breaker opens
- Calls `UserClientFallback.getUser()` instead

This prevents cascading failures in microservices.

18. API Gateway

1. What is an API Gateway, and why is it important in microservices architecture?

API Gateway

An API Gateway is a **single entry point** for all external clients in a microservices system.

✔ Why is it important?

In microservices:

- Clients would otherwise call many services individually
- Each service has different URLs, ports, protocols
- Security, logging, rate-limiting, etc., would need to be duplicated

⍰ API Gateway solves this by providing:

- **Request routing** → sends request to the right microservice
- **Authentication & authorization**
- **Rate-limiting**
- **Load balancing**
- **Logging & monitoring**
- **Centralized cross-cutting concerns**
- **Response aggregation**

Common gateways:

- Spring Cloud Gateway (modern)
- Zuul (old)
- Kong
- NGINX
- AWS API Gateway

2. Explain how to set up an API Gateway using Spring Cloud Gateway

✔ Step 1 — Add dependency

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

✔ Step 2 — Enable Eureka Client

```
@SpringBootApplication
@EnableEurekaClient
public class ApiGatewayApplication {}
```

✔ Step 3 — Configure routes in `application.yml`

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://USER-SERVICE
          predicates:
            - Path=/users/**
        - id: order-service
          uri: lb://ORDER-SERVICE
          predicates:
            - Path=/orders/**
```

Here:

- `lb://` → load-balanced URI (via Eureka + LoadBalancer)
- API Gateway maps:
  - `/users/**` → user-service
  - `/orders/**` → order-service

✔ Step 4 — Run gateway → clients call it like:

```
GET http://localhost:8080/users/1
```

Gateway forwards it to:

```
http://USER-SERVICE/users/1
```

3. What are filters in Spring Cloud Gateway?

Filters are components that allow you to intercept requests and responses.

They are used for:

- Authentication
- Logging
- Adding/modifying headers
- Rate limiting
- Request validation
- IP blocking

Types of Filters:

Filter Type	Runs When	Example

Filter Type	Runs When	Example
Pre-filter	Before forwarding to service	Logging, authentication
Post-filter	After service response	Response transformation

✔ Example: Adding a filter in `application.yml`

```
filters:
  - AddRequestHeader=token, abc123
```

✔ Example: Custom Filter

```
@Component
public class RequestLoggingFilter implements GatewayFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        System.out.println("Request URI: " +
exchange.getRequest().getURI());
        return chain.filter(exchange);
    }
}
```

4. How do you implement rate-limiting in an API Gateway?

Spring Cloud Gateway provides built-in rate-limiting using **Redis**.

✔ Step 1 — Add Redis dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
```

✔ Step 2 — Configure rate-limiter in `application.yml`

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://USER-SERVICE
          predicates:
            - Path=/users/**
          filters:
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 5   # 5 requests per
second
                redis-rate-limiter.burstCapacity: 10  # max burst
                key-resolver: "#{@ipKeyResolver}"
```

✔ Step 3 — Define Key Resolver (e.g., based on Client IP)

```
@Bean
public KeyResolver ipKeyResolver() {
    return exchange -> Mono.just(

exchange.getRequest().getRemoteAddress().getAddress().getHostAddress()
    );
}
```

✔ Each IP is restricted to **5 requests per second**
✔ Can use other resolvers:

- User ID
- API key
- Auth token

19. Spring Security

1. What is Spring Security, and how does it work?

Spring Security is a powerful framework used to secure Java applications. It provides:

- **Authentication** → Verifying *who* the user is
- **Authorization** → Controlling *what* the user is allowed to access
- Protection from common attacks → CSRF, session fixation, clickjacking, etc.

How it works:
Spring Security uses a **chain of filters** that intercept every HTTP request. These filters decide:

- Whether the request is authenticated
- Whether the user has permission to access the resource
-

2. How do you configure Basic Authentication in Spring Security?

Spring Boot 3.x+ (Spring Security 6+) example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated()
            )
```

```
            .httpBasic()  // Enables basic authentication
            .csrf().disable();

        return http.build();
    }

    @Bean
    public InMemoryUserDetailsManager userDetailsService() {
        UserDetails user = User.withUsername("admin")
                .password("{noop}password")
                .roles("USER")
                .build();
        return new InMemoryUserDetailsManager(user);
    }
}
```

Basic auth sends credentials as Base64 in the header:
`Authorization: Basic base64(username:password)`

3. What is JWT, and how do you implement it in Spring Boot?

JWT (JSON Web Token) is a stateless token-based authentication mechanism.

A JWT contains:

- **Header** (algorithm + token type)
- **Payload** (user info, roles, expiration)
- **Signature** (verify integrity)

Why JWT?

- No session storage required → *scalable*
- Each request contains the token → *stateless*
- Easy for microservices → *distributed authentication*

How JWT is implemented in Spring Boot:

✔ User logs in → Server validates credentials
✔ Server generates JWT
✔ Client stores token (usually in localStorage)
✔ Client sends token with each request → `Authorization: Bearer <token>`
✔ Server validates token via filter before allowing access

Example JWT filter snippet:

```
public class JwtAuthFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
```

```
            throws ServletException, IOException {

        String header = request.getHeader("Authorization");
        if (header != null && header.startsWith("Bearer ")) {
            String token = header.substring(7);
            // validate token, extract username, set authentication
        }

        filterChain.doFilter(request, response);
    }
}
```

4. How can you secure REST APIs using Spring Security?

1️⃣ Configure security rules

Example:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception
{
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/auth/**").permitAll()
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
        )
        .csrf().disable();

    return http.build();
}
```

2️⃣ Use authentication method:

- Basic Auth
- JWT
- OAuth2
- Sessions

3️⃣ Add filters (JWT, OAuth, custom filters)

4️⃣ Protect endpoints by roles/permissions using annotations:

```
@PreAuthorize("hasRole('ADMIN')")
public List<User> getAllUsers() { ... }
```

5. What are security filters, and how do they work in Spring?

Spring Security uses a **Security Filter Chain**, which is a list of filters executed in order for every request.

Examples of important security filters:

Filter	Purpose
`UsernamePasswordAuthenticationFilter`	Handles login with username/password
`JwtAuthenticationFilter`	(custom) Validates JWT tokens
`BasicAuthenticationFilter`	Handles HTTP Basic login
`CsrfFilter`	Protects from CSRF attacks
`AuthorizationFilter`	Checks if the user has permission

How they work:

- Filters process the request **before** it reaches the controller.
- They authenticate the request, authorize it, or block it.
- Custom filters can be added using:

```
http.addFilterBefore(jwtFilter,
UsernamePasswordAuthenticationFilter.class);
```

20 .Basic MySQL Questions

1. What is MySQL?

MySQL is an open-source **relational database management system (RDBMS)** that stores data in tables using SQL.
It is widely used because it is fast, reliable, scalable, and free.

2. Types of Joins in MySQL + Examples

Join Type	Description	Example
INNER JOIN	Returns only matching rows from both tables	`SELECT * FROM A INNER JOIN B ON A.id = B.id;`
LEFT JOIN	Returns all rows from left table + matching rows from right	`SELECT * FROM A LEFT JOIN B ON A.id = B.id;`
RIGHT JOIN	Returns all rows from right table + matching rows from left	`SELECT * FROM A RIGHT JOIN B ON A.id = B.id;`
FULL OUTER JOIN *(not supported directly)*	Returns all rows from both tables	Use: `LEFT JOIN UNION RIGHT JOIN`
CROSS JOIN	Returns Cartesian product	`SELECT * FROM A CROSS JOIN`

Join Type	Description	Example

```
B;
```

3. Difference Between CHAR and VARCHAR

CHAR	VARCHAR
Fixed-length	Variable-length
Always reserves fixed space	Stores only actual characters
Faster for fixed data (e.g., codes)	Saves memory
Example: `CHAR(10)` → stores 10 chars always	Example: `VARCHAR(10)` → stores 1–10 chars

4. Difference Between WHERE and HAVING

WHERE	HAVING
Filters **rows before** grouping	Filters **groups after** grouping
Cannot use aggregate functions	Can use aggregates (`SUM`, `COUNT`, etc.)
Used with SELECT, UPDATE, DELETE	Used only with SELECT + GROUP BY

Example:

```
SELECT dept, COUNT(*)
FROM employees
WHERE salary > 30000
GROUP BY dept
HAVING COUNT(*) > 5;
```

5. How to Create a Table in MySQL

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    salary DECIMAL(10,2),
    department VARCHAR(30)
);
```

6. What Is a Primary Key? Can a Table Have Multiple?

A **primary key** uniquely identifies each row.
Properties:

- Must be unique
- Cannot be NULL
- Only **one** primary key allowed per table
  (but can be **composite**, i.e., multiple columns combined)

Example:

```
PRIMARY KEY (emp_id, dept_id)
```

7. What Are Indexes and Why Are They Used?

Indexes speed up **searching and querying** by creating a special lookup structure.

Example:

```
CREATE INDEX idx_name ON employees(name);
```

They improve SELECT performance but slow down INSERT/UPDATE because index also needs updating.

8. GROUP BY Usage + Example

```
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department;
```

Groups records by department.

9. Difference Between DELETE, TRUNCATE, DROP

Command	Action	Use Case
DELETE	Deletes rows one by one, can use WHERE	Remove specific data
TRUNCATE	Removes all rows, faster, cannot use WHERE	Reset a table
DROP	Deletes whole table structure	Remove table completely

10. What Is a Foreign Key and Why Used?

A **foreign key** enforces relationships between tables.

Example:

```
FOREIGN KEY (dept_id) REFERENCES departments(id)
```

Used for referential integrity → prevents invalid data.

11. What Is Normalization + Normal Forms

Normalization reduces redundancy.

1NF

- No repeating groups
- Atomic (single value) columns

2NF

- Must be 1NF
- No partial dependency (non-key depends on part of composite key)

3NF

- Must be 2NF
- No transitive dependency (non-key depends on another non-key)

Higher forms: BCNF, 4NF, 5NF (advanced)

12. Stored Procedures + How to Create

Stored procedure = reusable block of SQL.

```
DELIMITER //
CREATE PROCEDURE getEmployees()
BEGIN
    SELECT * FROM employees;
END //
DELIMITER ;
```

Call it:

```
CALL getEmployees();
```

13. What Is a VIEW and Why Used?

A **view** is a virtual table created using a query.
Used for:

- Reusable queries
- Security (restrict sensitive columns)

Example:

```
CREATE VIEW v_emp AS
SELECT name, department FROM employees;
```

14. Explain ACID Properties

Property	Meaning
A – Atomicity	All or nothing transactions
C – Consistency	DB moves from one valid state to another
I – Isolation	Transactions don't interfere
D – Durability	Data is safe even after crash

15. What Is a Subquery? When Used?

A query inside another query.

Example:

```
SELECT name
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Used for comparisons, filtering based on another table, nested logic.

16. How Indexing Works + Types of Indexes

MySQL uses **B-Tree** or **Hash** structures internally.

Types:

- **Primary Index**
- **Unique Index**
- **Regular Index**
- **Composite Index** (multiple columns)
- **Full-Text Index** (for text search)
- **Spatial Index** (for GIS data)

17. Difference Between INNER and OUTER JOIN

JOIN	Explanation
INNER JOIN	Returns only matching rows
OUTER JOIN	Returns matching + non-matching rows (LEFT, RIGHT, FULL)

18. What Is a Trigger in MySQL? How to Create?

Trigger = code executed automatically on INSERT/UPDATE/DELETE.

Example:

```
CREATE TRIGGER before_insert_emp
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    SET NEW.created_at = NOW();
END;
```

19. Handling Errors in Stored Procedures

Use `DECLARE ... HANDLER`:

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
    SELECT 'Error occurred';
END;
```

20. Finding Duplicate Records

Example:

```
SELECT name, COUNT(*)
FROM employees
GROUP BY name
HAVING COUNT(*) > 1;
```

21. Purpose of LIMIT

Used to limit the number of returned rows.

```
SELECT * FROM employees LIMIT 5;
```

22. Difference Between UNION and UNION ALL

UNION	UNION ALL
Removes duplicates	Keeps duplicates
Slower	Faster
Uses sort operation	No sorting

Example:

```
SELECT name FROM A
UNION
SELECT name FROM B;
```