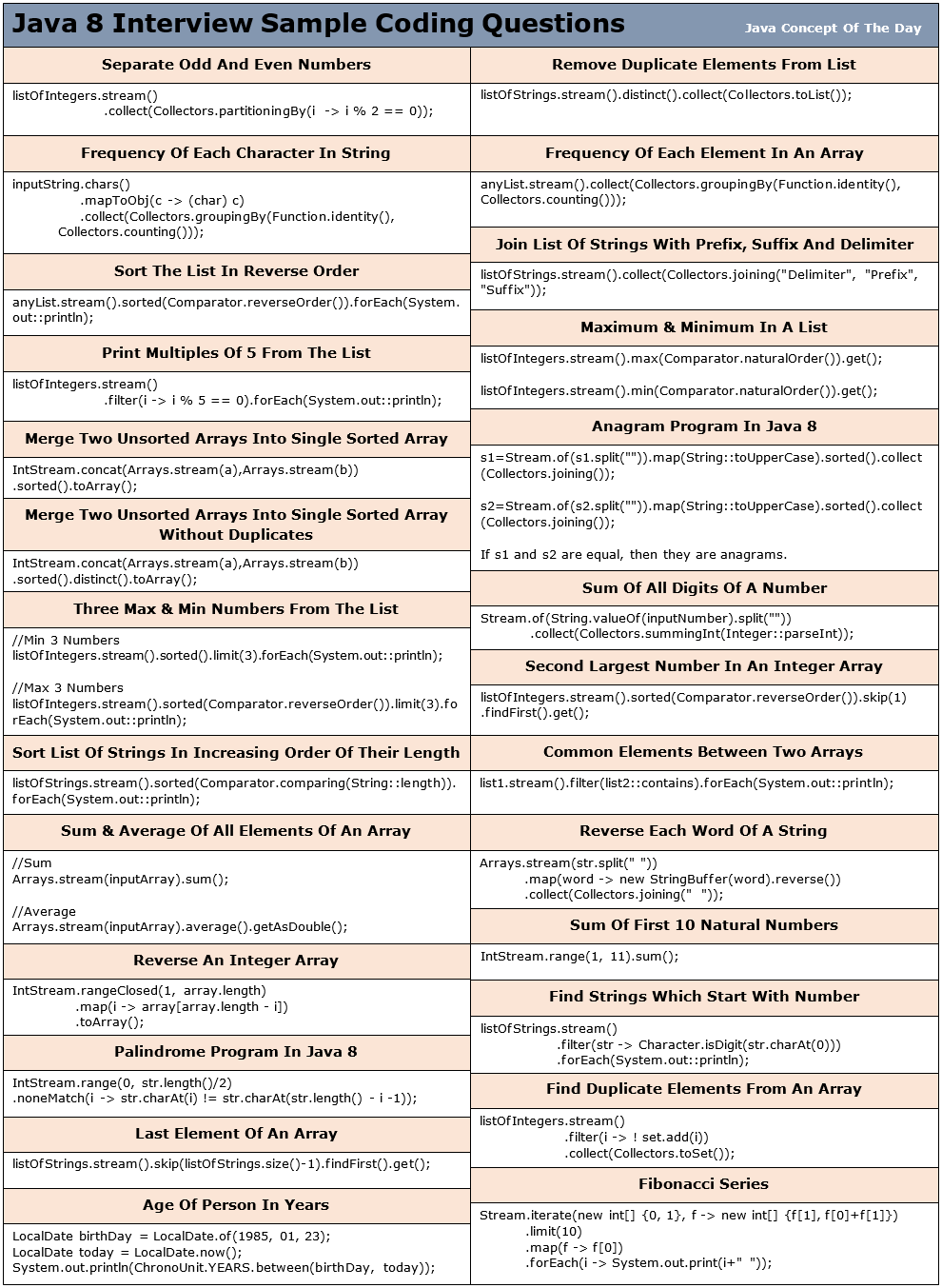
|  |  |  |
| --- | --- | --- |
| **Prime Number** if (number < 2 && number >= 0) return isPrime;  return !IntStream.*rangeClosed*(2, number / 2).anyMatch(i -> number % i == 0);  **SortBySecondName** Arrays.asList(sArray).stream().sorted((x,y)->{  String[] fs = x.split(",");  String[] ss = y.split(",");  return fs[1].compareTo(ss[1]);  }).forEach(p->System.out.print("["+p+"], "));  **StringAnagram**  char[] a1 = string1.toCharArray();  char[] a2 = string2.toCharArray();  Arrays.sort(a1); Arrays.sort(a2);  return Arrays.equals(a1, a2);  **isPallindrome**  return IntStream.range(0, checkString.length() / 2)  .noneMatch(i -> str.charAt(i) != str.charAt(checkString.length() - i - 1));  public class **OddEvenExample** implements Runnable {  @Override  public void run() {  for (int i = 1; i <= 10; i++) {  *synchronized (this)* {  if **(i % 2 == 0 && Thread.currentThread().getName().equals("t2"))** {  ***try { notifyAll(); SOP(i); wait();}***  catch (InterruptedException e) {e.printStackTrace(); }  }**else if(i % 2 != 0 && Thread.currentThread().getName().equals("t1"))**{  ***try { notify(); SOP(i); wait(); }***  catch (InterruptedException e) {e.printStackTrace();}  }}}}  public static void main(String[] args) {  OddEvenExample obj = new OddEvenExample();  Thread t1 = new Thread(obj, "t1");  Thread t2 = new Thread(obj, "t2");  t1.start(); t2.start(); } }   |  | | --- | | **MergeTwoArrays** Object[] streamConcat = **Stream.concat**(**Arrays.stream(a)**, Arrays.stream(b)).toArray()  Object[] systemArraycopy =  **Stream.of(a, b).flatMap(Stream::of)** .forEach(n::add);  List<Object> n = new ArrayList<>();  **Collections.addAll(n, a);**  Collections.addAll(n, b); | | public class ***Singleton*** **implements Serializable, Cloneable**{  **private static Singleton instance=null;**  **private Singleton**(){   if (instance != null) {  throw new InstantiationError("Already there");   } }  **protected Object readResolve**() { return instance; }  @Override  **protected Object clone() throws CloneNotSupportedException**{  return super.clone();}  public static Singleton getInstance(){  if(instance!=null)  return instance;  **synchronized (Singleton.class) {**  if(instance==null) { instance = new Singleton(); }  } return instance;  }}  **ReentrantLock** l = new ReentrantLock();  l.lock();  SOP("isLocked(): " + l.isLocked());  SOP("getHoldCount(): " + l.getHoldCount());  SOP("isHeldByCurrentThread(): " + l.isHeldByCurrentThread()); SOP("getQueueLength(): " + l.getQueueLength());  SOP("getHoldCount(): " + l.getHoldCount());  SOP("isLocked(): " + l.isLocked());  l.unlock();  SOP("isLocked(): " + l.isLocked());  SOP("isFair(): " + l.isFair());  **Highest Salary 🡪**  employeeList.stream() .collect(Collectors.maxBy(Comparator.comparing(Employee::geteSal)))  .get();  **FirstNonRepeatedChar** str.chars()  .mapToObj(i -> Character.toLowerCase(Character.valueOf((char) i))) .collect(Collectors.groupingBy(Function.identity(),  LinkedHashMap::new, Collectors.counting())) .entrySet().stream()  .filter(entry -> entry.getValue() == 1L)  .map(entry -> entry.getKey())  .findFirst().get();  ***Fibonacci Series***: List<Integer> list =  Stream.*iterate***(new int[]{1, 1}, n -> new int[]{n[1], n[0] + n[1]})**  .limit(series) .map(n -> n[0]).collect(Collectors.*toList*());  **ReverseString**  String revStr1 = Stream.of(str.split("")).reduce("", (r,c)->c+r);  **Convert List to HashMap**  Map<Integer, Employee> hMap = emp.stream().distinct() .collect(Collectors.toMap(  ***Employee::geteNumber, Function.identity(),***  ***(x, y) -> x, HashMap::new*)**  ); |
| **Book My Show: APIs Needed – *System Design***   * GetListOfCities() * GetListOfEventsByCity(CityId) * GetLocationsByCity(CityId) * GetLocationsByEventandCity(cityid, eventid) * GetEventsByLocationandCity(CityId, LocationId) * GetShowTiming(eventid, locationid) * GetAvailableSeats(eventid, locationid, showtimeid) * VarifyUserSelectedSeatsAvailable(eventid, locationid, showtimeid, seats) * BlockUserSelectedSeats() * BookUserSelectedSeat() * GetTimeoutForUserSelectedSeats()   **RDBMS Tables**   * Place (To save the hierarchal data for any given theatre-like country, state, city, and street) * Theatre, Screen, Tier (tier of seats), Seats, Movie * Offers, Ticket, User   **Java Memory Model - JMM**  1. ***Method area***: **All class level information** like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is **only one method area per JVM**, and it is a shared resource.  2. ***Heap area***: **Information of all objects** is stored in the heap area. There is also **one Heap Area per JVM**. It is also a shared resource.  3. ***Stack area***: **For every thread, JVM creates one run-time stack** which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. **It is not a shared resource**.  4. ***PC Registers***: **Store address of current execution instruction of a thread**. Obviously, each thread has separate PC Registers.  5. ***Native method stacks***: **For every thread, a separate native stack is created**. It stores native method information.  **System Design Concepts**  1. Domain Name System (DNS)  2. Load Balancer **3**. API Gateway  **4**. CDN **5**. Forward Proxy vs. Reverse Proxy  6. Caching **7**. Data Partitioning  8. Database Replication  9. Distributed Messaging Systems  10. Microservices **11**. NoSQL Databases  12. Database Index **13**. Distributed File Systems  14. Notification System **15**. Full-text Search  16. Distributed Coordination Services | ***Functional Interfaces***  **Consumer**<String> consumer = str -> SOP(str.toUpperCase());  consumer.**accept**("krsna");  **Function**<Double, Double> square = d -> d \* d;  SOP("Function "+square.**apply**(2.0));  **Predicate**<String> predicate = str -> str.length()>0;  SOP(predicate.**test**("krsna"));  **Supplier**<List<Integer>> supplier = () -> IntStream.range(0, 5)  .boxed().collect(Collectors.toList());  SOP(supplier.**get**());  ***SQL Query for Nth Highest Salary***  SELECT \* FROM (  SELECT emp\_name, emp\_salary,   ***DENSE\_RANK() OVER (ORDER BY emp\_salary DESC) AS rank***  FROM employee  ) AS subquery WHERE rank = 3;  ***Delete Duplicate rows from the table***  DELETE FROM customers WHERE ID NOT IN  (SELECT MIN(ID) FROM customers GROUP BY Name);  class Employee {  int id; String name;  **// constructor**  @Override  ***public int hashCode() {*** return Objects.hash(id, name); }  @Override  ***public boolean equals(Object obj) {***  if (this == obj) return true;  if (obj == null || getClass() != obj.getClass()) return false;  Employee employee = (Employee) obj;  return id == employee.id && name.equals(employee.name);  }  } Memory Management in Java |
|  |  |
| ***The Twelve Factors Applied to Microservices***  1. **Codebase** : One codebase per service, tracked in revision control; many deploys  2. **Dependencies** : Explicitly declare and isolate dependencies  3. **Config** : Store configuration in the environment  4. **Backing Services** : Treat backing services as attached resources  5. **Build, Release, Run** : Strictly separate build and run stages  6. **Processes** : Execute the app in one or more stateless processes  7. **Data Isolation** : Each service manages its own data  8. **Concurrency** : Scale out via the process model  9. **Disposability** : Maximize robustness with fast startup and graceful shutdown  10. **Dev/Prod Parity** : Keep development, staging, and production as similar as possible  11. **Logs** : Treat logs as event streams  12. **Admin Processes** : Run admin and management tasks as one-off processes  There are three class loaders: ***Bootstrap, Extension and System***  **Bootstrap** class loader: Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. ***It loads core java API classes present in the “JAVA\_HOME/jre/lib” directory***. This path is popularly known as the bootstrap path. It is implemented in native languages like C, C++.  **Extension** class loader: It is a child of the bootstrap class loader. ***It loads the classes present in the extensions directories “JAVA\_HOME/jre/lib/ext”***(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.  **System/Application** class loader: It is a child of the extension class loader. **It is responsible to load classes from the application classpath**. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the sun.misc.Launcher$AppClassLoader class.  **Note**: ***JVM follows the Delegation-Hierarchy principle to load classes***. | Key ***Differences Between @Autowired and @Bean***  ***@Autowired*** *is used* ***for automatically injecting Spring-managed beans*** *into other components*. It tells Spring to **find an existing bean** and inject it automatically. ✔ When you want **Spring to find and inject** a bean automatically ✔ Used in **Component-Scanned Beans** (e.g., @Component, @Service, @Repository, @Controller) ✔ Works with **constructor, setter, and field injection**  **@Bean** is used in a **@Configuration class** to **manually define  and create beans**. It gives more control over bean creation. ✔ When you **want full control** over bean creation ✔ When working with **third-party classes** that cannot use @Component ✔ When you need **custom bean logic** before returning the bean    jvm |



|  |  |
| --- | --- |
| ***SOLID Principles***  **Single Responsibility Principle** (SRP) 🡪 *A class should have only one reason to change*  **Open/Closed Principle** **🡪 *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification***  **Liskov’s Substitution Principle (LSP)** 🡪 ***Derived or child classes must be substitutable for their base or parent classes***  **Interface Segregation Principle (ISP)** 🡪 ***Do not force any client to implement an interface which is irrelevant to them***  **Dependency Inversion Principle (DIP)** 🡪 ***High-level modules should not depend on low-level modules. Both should depend on abstractions* ------------------------------Maven----------------------------------- mvn package** build, put it into **target folder**. **mvn install** to create a JAR/WAR, put it into **local .m2** **mvn deploy** to push artifacts to a **remote repository**. | running mvn install will execute:  validate → compile → test → package → verify → install |
| ***Predicate Join in Java (Criteria API with JPA)***  In Spring Boot with JPA, Predicate is used for dynamic query construction in the Criteria API. A predicate join is used when filtering data based on a condition applied to a related entity (i.e., performing a JOIN operation with conditions). @Repository @Transactional public class EmployeeRepositoryCustom { public List<Employee> findEmployeesByDepartmentName (String departmentName) {  @PersistenceContext  private EntityManager entityManager;  ***CriteriaBuilder cb = entityManager.getCriteriaBuilder();***  ***CriteriaQuery<Employee> query = cb.createQuery(Employee.class); Root<Employee> employee = query.from(Employee.class);***  // Joining Employee with Department  ***Join<Employee, Department> departmentJoin = employee.join("department");***  // Creating Predicate for department name  ***Predicate departmentPredicate = cb.equal(departmentJoin.get("name"), departmentName);***  // Applying Predicate in WHERE clause  ***query.select(employee).where(departmentPredicate);***  ***return entityManager.createQuery(query).getResultList();***  }} | |
| **Disadvantages of Using Spring Boot** 1️. ***Increased Memory Consumption:*** Leading to larger JAR files and higher memory usage.  2️. ***Slower Startup Time:*** Auto-configurations scan and initialize many unused components, making it slower.  3️. ***Less Control Over Auto-Configuration***: Automatically configures many dependencies, but developers may not have full control over how things are set up.  4️. ***Overhead Due to Unused Dependencies***: Spring Boot starters pull unnecessary dependencies, increasing application size.  5️. ***Debugging and Performance Tuning Can Be Complex***: Debugging issues related to auto-configuration, proxy beans, or circular dependencies can be challenging.  6️. ***Harder Migration to Spring Framework***: If an application grows too large, moving from Spring Boot to Spring Framework (for better control) is not easy.  7️. ***High Resource Usage in Microservices***: Running multiple microservices means each instance includes an embedded server, consuming extra CPU and RAM. ***🔹 When to Avoid Spring Boot🔹***  🚫 If your application requires high-performance, lightweight frameworks like Quarkus or Micronaut.  🚫 If you don’t need embedded servers and prefer traditional WAR deployments.  🚫 If you need full control over configurations (better with standard Spring Framework).  🚫 If you are working with low-memory environments (IoT, Edge Computing, etc.) | |

***One-to-One Association***

|  |  |
| --- | --- |
| @Entity  public class **User** {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String name;  ***@OneToOne(cascade = CascadeType.ALL)***  ***@JoinColumn(name="profile\_id", referencedColumnName = "id")***  private UserProfile profile;  // Getters & Setters  } | @Entity  public class **UserProfile** {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String bio;  ***@OneToOne(mappedBy = "profile")***  private User user;  // Getters & Setters  } |

***One-to-Many & Many-to-One Association***

|  |  |
| --- | --- |
| @Entity  public class Department {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String name;  ***@OneToMany(mappedBy = "department",   cascade = CascadeType.ALL)***  private List<Employee> employees = new ArrayList<>();  // Getters & Setters  } | @Entity  public class Employee {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String name;  ***@ManyToOne***  ***@JoinColumn(name = "department\_id")***  **private Department department;**  // Getters & Setters  } |

***Many-to-Many Association***

|  |  |
| --- | --- |
| @Entity  public class Student {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String name;  ***@ManyToMany***  ***@JoinTable(***  ***name = "student\_course",***  ***joinColumns = @JoinColumn(name = "student\_id"),***  ***inverseJoinColumns = @JoinColumn(name = "course\_id")***  ***)***  ***private List<Course> courses = new ArrayList<>();***  // Getters & Setters  } | @Entity  public class Course {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String title;  ***@ManyToMany(mappedBy = "courses")***  ***private List<Student> students = new ArrayList<>();***  // Getters & Setters  } |

***Kafka Annotations***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Annotation** | **Purpose** | **Usage** | | @EnableKafka | Enables Kafka in Spring Boot | Config class | | ***@KafkaTemplate*** | ***Sends messages to Kafka*** | Producer service | | ***@KafkaListener*** | ***Listens to Kafka topic messages*** | Consumer class | | @KafkaHandler | Handles different message types | Consumer class | | @DltHandler | Handles messages in Dead Letter Topic | Consumer class | | @KafkaListenerContainerFactory | Custom Kafka listener factory | Config class | | @KafkaStreamsDefaultConfiguration | Configures Kafka Streams | Config class | | @SendTo | Forwards messages to another topic | Consumer class | |

***High-Level System Design Approach***

1. **Clarify Requirements**: Functional & non-functional (latency, scalability, availability, consistency).
2. **Define High-Level Components**: API Gateway, Load Balancer, Microservices, Database, Caching, Messaging, etc.
3. **Data Model & Storage**: SQL vs NoSQL, Sharding, Replication.
4. **Scaling Strategy**: Vertical vs Horizontal Scaling, Caching, CDN.
5. **Fault Tolerance & Availability**: Retry Mechanism, Circuit Breaker, and Failover.
6. **Security & Observability**: Authentication, Authorization, Logging, Monitoring.
7. **Trade-offs & Optimizations**: CAP Theorem, Eventual Consistency, Performance Tuning.

***What Are Functional Requirements?***

Functional requirements describe the features, behavior, and expected outputs of a system. They focus on:

* User interactions (What actions can users perform?)
* System responses (What should happen after an action?)
* Business rules (Validation, calculations, workflow processing)
* APIs and Integrations (How does the system communicate with other services?)

**Functional Requirements vs Non-Functional Requirements**

| **Feature** | **Functional Requirements** | **Non-Functional Requirements** |
| --- | --- | --- |
| Focus | What the system should do | How the system should perform |
| Example | "Users can register and log in" | "System should handle 10,000 concurrent users" |
| Validation | Can be tested with unit and functional tests | Measured with performance tests |
| User Interaction | Directly affects users' actions | Affects system experience |
| Failure Impact | Breaks core functionality | Causes performance degradation |

|  |  |
| --- | --- |
| **Functional Requirements in System Design**  1. User Authentication & Authorization  2. API Functionalities  3. Data Processing & Storage  4. Caching & Performance Optimization  5. Payment & Transactions  6. Notification System  7. Logging & Monitoring  8. Search & Filtering | **Functional Requirements in Microservices**  ✅ Service-to-service communication (REST, gRPC, Kafka).  ✅ Event-driven processing (e.g., Kafka for asynchronous message handling).  ✅ Database per microservice (Each service owns its data).  ✅ Data consistency (Saga Pattern, 2PC for distributed transactions).  ✅ Failover and retry mechanisms (Circuit Breaker, Retry Pattern). |

***Key Takeaways for Interviews***

🔹 Always start with user flows: Describe the user journey.

🔹 Break down functionalities by modules: Authentication, API endpoints, caching, etc.

🔹 Mention real-world examples: "Amazon uses Elastic search for product search."

🔹 Discuss trade-offs: E.g., "We chose Kafka for event-driven processing instead of RabbitMQ for scalability."

|  |
| --- |
| **Interceptor:** Interceptors in Spring MVC are filters that **sit between the client and the controller**. They are used to intercept incoming HTTP requests before and/or after they reach the controller layer.🡪 write a component class which implementsHandlerInterceptor interface **@Component**  public class LoggingInterceptor implements HandlerInterceptor  **@Override** public boolean preHandle (HttpServletRequest request, HttpServletResponse response, Object handler) {  **@Override** public void afterCompletion (HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) 🡪 write a configuration class to register the custom interceptor  **@Configuration**  public class WebConfig implements **WebMvcConfigurer** {  @Autowired  private LoggingInterceptor loggingInterceptor;  @Override  *public void* ***addInterceptors****(InterceptorRegistry registry)* {  registry.**addInterceptor**(loggingInterceptor)  .addPathPatterns("/api/\*\*"); // Intercept only /api/\*  }} |

***Difference between REST API and Microservices***  
*REST API is about communication style (HTTP-based interaction), while Microservices is about application design (breaking a system into small, independent services).*

|  |  |  |
| --- | --- | --- |
| **Concept** | **REST API** | **Microservices** |
| What it is | Interface for data exchange (HTTP) | Architectural style for modular systems |
| Focus | *How* to communicate | *How* to structure/deploy the system |
| Usage | Used in Monoliths & Microservices | Commonly exposes REST APIs |

|  |  |  |
| --- | --- | --- |
| **Aspect** | **REST API** | **Microservices** |
| **Definition** | An architectural style that defines how systems communicate over HTTP using standard methods. | An architectural approach where an application is structured as a collection of loosely coupled services. |
| **Purpose** | Interface to expose or consume functionality over HTTP. | Entire application broken into independently deployable services. |
| **Scope** | Communication protocol/interface. | System design and structure (includes APIs, databases, services, infra). |
| **Communication** | Usually via REST (HTTP) using JSON/XML payloads. | Can use REST, messaging queues (e.g., Kafka), gRPC, etc. |
| **Granularity** | Endpoint-level. | Application/service-level. |
| **Focus** | Defines *how* components interact. | Defines *how* the system is architected and deployed. |
| **State** | Stateless by nature. | Microservices can be stateless or manage state via databases. |
| **Examples** | GET /users, POST /orders | User Service, Order Service, Payment Service (each with its own REST APIs). |

┌────────────────────────────┐

│ Web or Mobile App │

└────────────────────────────┘

|

▼

┌────────────────────────────┐

│ API Gateway │ ◄── RESTful communication (HTTP)

└────────────────────────────┘

/ | \

▼ ▼ ▼

┌────────────────┐ ┌────────────────┐ ┌────────────────┐

│ User Service │ │ Order Service │ │ Payment Service│ ◄── Microservices

└────────────────┘ └────────────────┘ └────────────────┘

| | |

▼ ▼ ▼

┌────────────┐ ┌────────────┐ ┌────────────┐

│User DB │ │Order DB │ │Payment DB │ ◄── Each service has its own DB

└────────────┘ └────────────┘ └────────────┘

***Multi-Data Source Config***  
@EnableJpaRepositories(

basePackages = "com.example.repo.primary",

entityManagerFactoryRef = "primaryEntityManagerFactory",

transactionManagerRef = "primaryTransactionManager"

)