**JWT TOKEN**

**What is authentication and authorization?**

Authentication is the process of verifying a user's identity (e.g., using a username and password).

Authorization is the process of determining whether an authenticated user has permission to access a specific resource or perform an action.

A close-up of a computer code

Description automatically generated

**What are some common approaches to implement authentication in Java applications?**

* Basic Authentication
* OAuth2
* JWT Tokens
* Session-based Authentication (e.g., Spring Security)

**Basic Authentication:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**Note:** Write controller class other wise you will get **404 not found** error.

**JWT Tokens**

**Pom.xml**

**JwtUtilityClass**

**JwtAuthenticationFilterClass**

**SecuriyConfigClass**

**AuthenticationControllerClass**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**Create JWT Utility Class:**

**A computer code with text

AI-generated content may be incorrect.**

**Create a JwtAuthenticationFilter**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**A computer code with text

AI-generated content may be incorrect.**

**Configure Spring Security**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**A computer code with text

AI-generated content may be incorrect.**

**Create AuthController to Generate Tokens**

**A close-up of a computer code

AI-generated content may be incorrect.What is the role of Spring Security in authentication and authorization?**

* Spring Security provides configurable tools for authentication and authorization, including form-based login, role-based access control.

**How do role-based access control (RBAC) and attribute-based access control (ABAC) differ?**

* + RBAC assigns permissions based on roles (e.g., ADMIN, USER).
  + ABAC considers attributes of the user, environment, and resource (e.g., age > 18).

**What is a JWT, and why is it used?** (**JSON Web Token) (JavaScript Object Notation)**

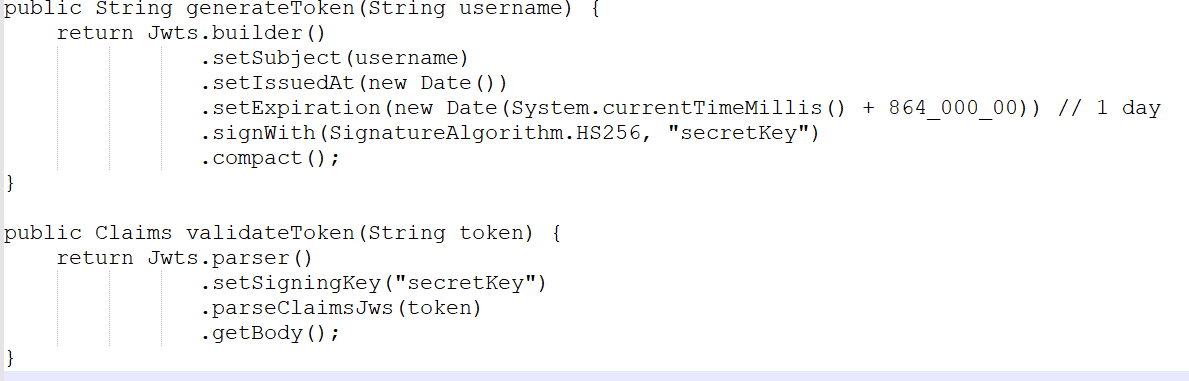
JSON Web Token (JWT) is a way to securely transmit information between parties using JSON. JWTs are commonly used for authentication and authorization. They are compact and can be sent in URLs, HTTP headers, or POST parameters.

**Explain the structure of a JWT.**

* + A JWT consists of three parts:
    1. **Header**: Metadata (e.g., type and algorithm).
    2. **Payload**: Claims (e.g., user data, roles).
    3. **Signature**: Validates token integrity.

**Example Token:** eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsImFkbWluIjp0cnVlfQ.Xu4PvWo74u6KYZJImoGfb8Uu6\_4PPbgZNrV17shWo1M

**How can you implement JWT authentication in a Java application?**



Generate Token:

1. **Jwts.builder()** → Creates a new JWT.
2. **.setSubject(username**) → Stores **username** in the token payload (claims).
3. **.setIssuedAt(new Date())** → Adds the **current timestamp** to track token issuance.
4. **.setExpiration(new Date(System.currentTimeMillis() + EXPIRATION\_TIME))** →
   * Sets an **expiration time** (current time + 1 day).
5. **.signWith(key, SignatureAlgorithm.HS256) →**
   * Signs the JWT using **HMAC SHA-256** with the secret key.
6. **.compact()** → Converts everything into a **JWT string**.

Validate Token:

 **Jwts.parser()**

* Creates a **JWT parser** instance to process the token.

 **.setSigningKey("secretKey")**

* Sets the **secret key** used to **validate the token's signature**.
* The same key used to **sign** the token must be used for **validation**.

 **.parseClaimsJws(token)**

* **Parses the JWT token**, verifying:  
  ✅ It is **well-formed**.  
  ✅ It **hasn't been tampered with**.  
  ✅ The **signature matches** using the given secret key.

 **.getBody()**

* Extracts the **claims (payload)** from the token.
* The payload contains **user-related information**, like username, roles, issued time, and expiration.

**What are claims in a JWT, and what are the differences between registered, public, and private claims?**

Claims are pieces of information (key-value pairs) stored in the token to convey details about a user or an entity.

* + 1. Registered claims are standardized (e.g., iss, sub, exp).
    2. Public claims are user-defined but must avoid collisions.
    3. Private claims are used for custom data shared between parties.

**How do you secure a JWT?**

* + Use strong secret keys.
  + Use HTTPS to protect tokens in transit.
  + Set short expiration times.
  + Implement refresh tokens.

**How would you implement role-based access control using JWT in a Spring Boot application?**

A close up of text

Description automatically generated

**What would you do if a JWT token expires?**

Implement a refresh token mechanism to issue a new token without requiring re-authentication.

**Can a JWT be modified or tampered with? How would you prevent this?**

A JWT can be tampered with, but tampering invalidates the signature. Always verify the token's signature using the secret key.

**How would you log out a user in a JWT-based system?**

* Add the JWT to a blacklist.
* Track invalidated tokens in a database or cache.

**What are the common errors you might encounter when using JWT?**

* Expired token error.
* Signature verification failure.
* Malformed token error.

**Explain the difference between symmetric and asymmetric encryption in JWT signing.**

* Symmetric uses the same key for signing and verification (e.g., HS256).
* Asymmetric uses a public-private key pair (e.g., RS256).

**How can you handle multi-factor authentication (MFA) in a JWT-based system?**

* Add an intermediate step after verifying the password to validate a second factor (e.g., OTP) and then issue a JWT.

**How can you implement token-based authentication for APIs in Spring Boot?**

* + Use a filter to intercept requests and validate JWTs:

A screenshot of a computer code

Description automatically generated

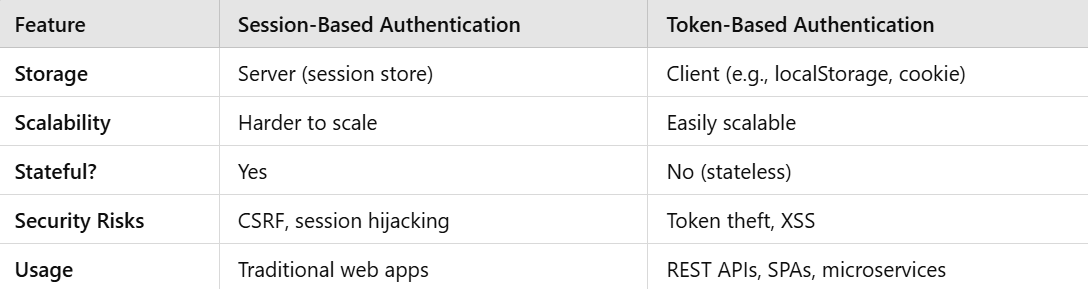
**Session-Based vs Token-Based Authentication:**

**Session-Based Authentication**

The server creates and stores a session for an authenticated user and uses cookies to track the session.

**Token-Based Authentication**

The server issues a token (typically JWT) after authentication, and the client sends the token in requests to authenticate itself.



**login**

**- authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(user.getUsername(), user.getPassword());**

**- spring security calls customerUserdetailService.loadByUsername();**

**- generate token**

**accessing admin:**

**- postman sends a request with providing JWT Token**

**- extract the username**

**- extract the role**

**- new authentication object created.**

**- spring security check whether it has required role or not.**

**==============================================================================**

**RESTful API**

**A close-up of a computer code

AI-generated content may be incorrect.**

**What is API?**

API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other

**What is a RESTful API?**

A **RESTful API** is an **API that follows REST (Representational State Transfer) principles** to enable communication between clients (e.g., web browsers, mobile apps) and servers over the **HTTP/HTTPS** protocol.

**What are HTTP methods in REST?**

**GET**: Retrieve a resource.

**POST**: Create a new resource.

**PUT**: Update an existing resource.

**DELETE**: Remove a resource.

**PATCH**: Partial update of a resource.

**What is the difference between PUT and POST?**

**POST:** Creates a new resource. It’s not idempotent (sending the same request multiple times can create multiple resources).

**PUT:** Updates or creates a resource at a specific URI. It’s idempotent (repeated requests have the same effect).

**What is idempotence in REST?**

An operation is idempotent if repeated execution has the same result.

Example: Repeated GET or DELETE requests don’t change the state of the server.

**How do you create a RESTful API in Java?**

By using frameworks like Spring Boot, JAX-RS (Jersey/RESTEasy), or SparkJava.

**Example in Spring Boot:**

**A computer code with black text

Description automatically generated**

**What is @RestController in Spring Boot?**

It’s a specialized version of @Controller that combines @Controller and @ResponseBody. It’s used for creating RESTful web services.

**How do you handle exceptions in a REST API?**

Use @ControllerAdvice and @ExceptionHandler for global exception handling.

**Example:**

**A close-up of a computer code

Description automatically generated**

**What is @RequestMapping vs @GetMapping in Spring Boot?**

@RequestMapping is a versatile annotation that maps HTTP methods and paths.

@GetMapping is a shorthand for @RequestMapping(method = RequestMethod.GET).

**How do you secure a REST API in Java?**

**Use Spring Security to secure endpoints. Example:**

**A close-up of a computer code

Description automatically generated**

Use JWT for token-based security.

**What is HATEOAS?**

Hypermedia as the Engine of Application State: It’s a principle of REST where responses include links to other actions/resources.

**Examples:**

**A computer code with black text

Description automatically generated**

**How do you version a REST API?**

Use URI versioning: /v1/resource.

Use header versioning: Accept: application/vnd.api.v1+json.

**What is the difference between @PathVariable and @RequestParam?**

@PathVariable: Extracts values from the URI path.

Example: /api/resource/{id}

@RequestParam: Extracts query parameters.

Example: /api/resource?id=1

**How do you handle pagination in a REST API?**

Pass page and size as query parameters: Example: /api/resources?page=0&size=10

Response includes metadata for total pages, current page, etc.

**How do you test REST APIs in Java?**

Use tools like Postman or Curl for manual testing.

Use MockMvc or RestAssured for automated testing. Example with MockMvc:

**java**

**A computer code with black text

Description automatically generated**

**What is the difference between synchronous and asynchronous REST APIs?**

**Synchronous**: The client waits for the server to respond before proceeding.

**Asynchronous**: The client can continue other tasks while the response is processed later.

**Example of asynchronous REST API in Spring Boot:**

**A computer code with black text

Description automatically generated with medium confidence**

**How do you ensure REST API performance?**

Use caching (e.g., @Cacheable in Spring).

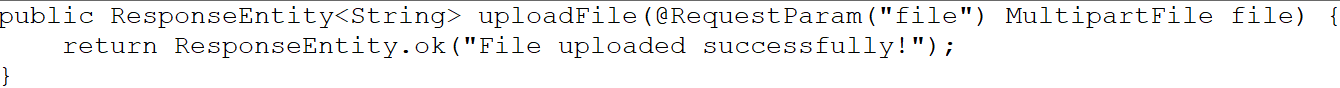
Optimize database queries.

Use pagination for large datasets.

Compress responses (e.g., GZIP).

**How do you handle large file uploads/downloads in REST?**

**Use MultipartFile for uploads:** **@PostMapping("/upload")**

****

Stream large files for downloads.

**What are REST API best practices?**

Use meaningful URIs (/users instead of /getAllUsers).

Use HTTP status codes correctly (200 OK, 404 Not Found, 401 Unauthorized).

Document APIs using Swagger/OpenAPI.

**What is OpenAPI/Swagger?**

A tool for designing, documenting, and consuming REST APIs. It automatically generates documentation from code using annotations like @ApiOperation.

**Status Codes:**

**200 OK –** Successful GET, PUT, PATCH, DELETE requests.

**201 Created –** Successful resource creation via POST.

**204 No Content –** Successful DELETE or update without a response body.

**400 Bad Request** – Malformed requests or invalid input.

**401 Unauthorized –** Missing or invalid authentication token.

**403 Forbidden** – Insufficient permissions**.**

**404 Not Found –** Resource does not exist.

**409 Conflict –** Resource state conflicts (e.g., duplicate data).

**500 Internal Server Error –** Unexpected backend error.

**Java 8 Features**

🡪 Java 8 was designed to improve developer productivity, code readability, and performance, making it a watershed moment in the Java language's evolution.

**What features do you know or use in Java 8?**

Here you can list down all the key features of Java 8 like,

* Functional Interface
* Lambda Expression
* Stream API
* Completable Future
* Java DateTime API
* Method Reference
* Comparable and Comparator
* Optional Class

**Functional Interface**

A screenshot of a computer

Description automatically generated

**What is Functional Interface in Java 8?**

A **functional interface** in Java is an interface that contains exactly **one abstract method** but can have multiple **default** and **static** methods. It is primarily used to enable **lambda expressions** and **method references**, allowing a more concise and functional programming approach.

**TYPES OF FUNCTIONAL INTERFACES:**

**1.Consumer**

**2. Supplier**

**3.Function**

**4.Predicate**

**All are present in java.util.function; package.**

**1.Consumer:**

**🡪** It represents an operation, that accept a single input parameter and returns no result.

**A computer screen shot of a computer code

AI-generated content may be incorrect.**

**A computer code with text

AI-generated content may be incorrect.**

**2.Supplier:**

🡪 It represents the supplier of the result. Accept no input parameter but produce a result.

A computer screen shot of a computer code

AI-generated content may be incorrect.

A computer code with black text

AI-generated content may be incorrect.

**3.Function**

**🡪** Represents function, that accept one argument process and produces it results.

A close-up of a computer screen

AI-generated content may be incorrect.

A computer code with text

AI-generated content may be incorrect.

**4.Predicate**

**🡪** It represents function, that accept one argument and return the Boolean.

**A white background with black text

AI-generated content may be incorrect.**

**Lambda Expression**

**What is Lambda Expression?**

A lambda expression in Java is a short block of code that acts like a method, but without need of a name.

It eliminates boilerplate code when implementing functional interfaces.

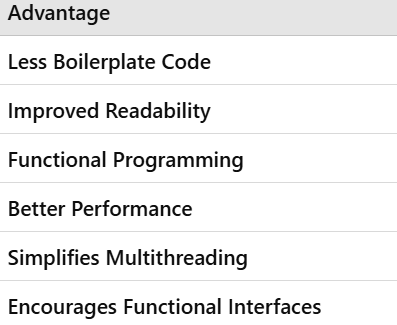
A screenshot of a computer

AI-generated content may be incorrect.

**Can we extend a functional interface from another functional interface?**

Yes, we can extend but if you extend that your functional interface will not act as a functional interface because it will find multiple abstract methods inside that.

**What are the advantages of Lambda Expressio n?**



**What are the disadvantages of Lambda expression?**

* Hard to use without an IDE
* Complex to debug

**How are functional interfaces and Lambda Expressions related?**

Functional interfaces in Java are interfaces that only contains one abstract method.

* Lambda expressions provide a simple way to implement functional interfaces.
* Lambda expressions can be used wherever functional interfaces are needed.

// Functional interface  
interface MyFunctionalInterface {  
 void myMethod();  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Lambda expression for implemention of the functional interface  
 MyFunctionalInterface myLambda = () -> System.out.println("Hello Lambda!");  
   
 // calling method, using lambda expression  
 myLambda.myMethod();  
 }  
}

**How do you define a custom functional interface and use it with a lambda expression?**

**A computer code with text

AI-generated content may be incorrect.**

**Can lambda expressions access local variables?**→ Yes, but only effectively final variables (cannot be modified after assignment).

**Can lambda expressions access instance and static variables?**→ Yes, unlike local variables, they can modify instance and static variables.

**Stream API**

**What is Stream API in Java 8?**

* Stream API is used to process collections of objects.
* It will improve program's performance by allowing you to avoid unnecessary loops and iterations.
* Streams can be used for filtering, collecting, printing, and converting from one data structure to another, etc.

**What are the stream methods you used in your project?**

* filter
* forEach
* sorted
* map
* flatMap
* reduce
* groupingBy
* collect

**How is Stream different from Collections?**  
→ Collections store data and allow modifications, while Streams process data and are **immutable**.

**How do you create a Stream in Java?**

**A close-up of a computer code

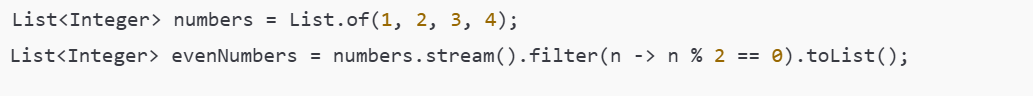
AI-generated content may be incorrect.**

**Can a Stream be reused?**  
→ No, a Stream cannot be reused after a terminal operation.

**What is map () in Stream API?**→ Transforms each element in a stream.



**What is filter () in Stream API?**→ Filters elements based on a condition.

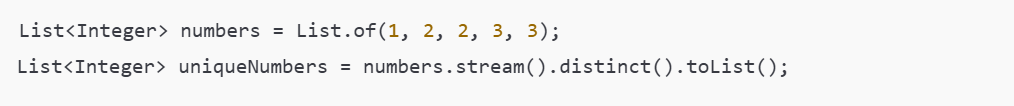


**What is flatMap () in Java Streams?**  
→ Flattens multiple streams into a single stream.

A close up of text

AI-generated content may be incorrect.

**What is distinct () in Java Streams?**  
→ Removes duplicate elements.

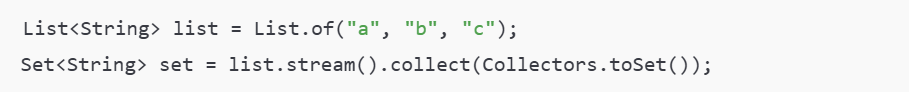


**What does sorted () do in Stream API?**  
→ Sorts elements naturally or using a comparator.

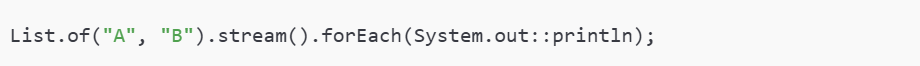




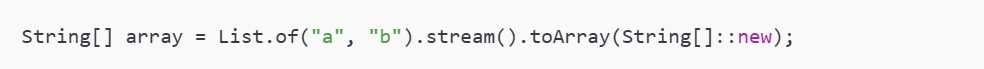
**What is collect () in Java Streams?**  
→ Collects stream elements into a collection.

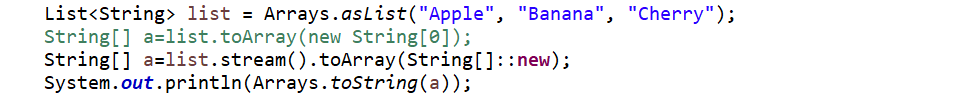


**What does forEach() do?**  
→ Iterates over each element.



**How do you convert a Stream to an array?**

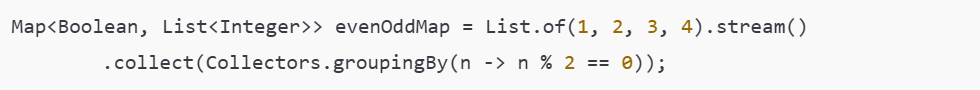
****

****

**How do you convert an IntStream to a List?**

****

**What is groupingBy() in Stream API**?  
→ Groups elements based on a classifier function.

****

**What is partitioningBy() in Java Streams?**→ Similar to groupingBy() but creates only two partitions (true and false).

**How do you find the max or min element using Stream?**

****

**How do you remove duplicates from a List using Stream API?**

****

**How do you limit the number of elements in a Stream?**→ Use limit(n).

****

**What is reduce () in Stream API?**→ Performs aggregation like sum, min, max.

****

**When should you not use Streams?**

* When modifying a collection (e.g., adding/removing elements).
* When the operation is stateful (e.g., keeping track of previous values).
* When debugging is required (since streams make debugging harder).

**Completable Future**

* Runs tasks asynchronously (non-blocking execution).
* Performs dependent tasks (chain multiple async operations).
* Handles errors (exception handling in async tasks).
* Combines multiple futures (run multiple tasks in parallel and combine results)

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Java DateTime API**

Java 8 introduced a new Date-Time API (java.time package) to solve the problems of the old java.util.Date and java.util.Calendar, which were mutable, error-prone, and not thread-safe.

A screenshot of a computer

AI-generated content may be incorrect.

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**A computer screen shot of a code

AI-generated content may be incorrect.**

**Method Reference**

Method Reference is a shorthand lambda expression that refers to a method without executing it. It improves code readability and reusability.

**Comparable and Comparator**

**Differentiate Between Comparable and Comparator in Java.**

Java provides two interfaces for configuring objects using class data members:

* Comparable
* Comparator

A screenshot of a computer

AI-generated content may be incorrect.

A computer code on a white background

AI-generated content may be incorrect.

**When to use map and flatMap?**

map (): It is used where we have to map the elements of a particular collection to a specific function, and then we need to return the stream that contains the updated results.

Example: Multiply all the elements of a list by 3 and return the updated list.

flatMap(): It is used where we have to transform or flatten the string, as we can't flatten our string using map().

Example: Get the first Character of all the String present in a List of Strings and return the result in form of a stream.

A computer code with black text

AI-generated content may be incorrect.

**What is Optional Class in Java 8?**

In Java 8, Optional Class is a container object.

* The Optional class used to represent a value that may be present or may not be.
* This class helps in avoiding null pointer exceptions by providing methods to check the presence of a value before accessing it.
* This helps null values handling more effectively.

**Example:**

Optional<String> optionalName = Optional.ofNullable("John");  
  
// Check if value is present  
if (optionalName.isPresent()) {  
 System.out.println("Name is present: " + optionalName.get());  
} else {  
 System.out.println("Name is not present");  
}

**Provide Some Optional Methods in Java 8.**

**of**: It creates an Optional with a non-null value.

**ofNullable**: It creates an Optional with a given nullable value.

**empty**: It creates an empty Optional.

**isPresent**: This checks whether the Optional contains a non-null value.

**get**: It gets the value if present, otherwise it throws an exception i.e. NoSuchElementException.

**orElse**: It returns the value if present, otherwise returns the specified default value.

**orElseGet**: It returns the value if present, otherwise it returns the result of invoking the supplier function.

**orElseThrow**: It returns the value if present, otherwise it throws an exception produced by the provided supplier.

**map**: It applies a function to the value if present and return a new Optional with the result or return an empty Optional if no value is present.

**filter**: It applies a predicate to the value if present and return an Optional with the value if it matches the predicate, otherwise return an empty Optional.

**What is Date-Time API in Java 8?**

The Date-Time API in Java 8 provides a set of classes for date-time conversions, including timelines and advanced programming.

* It imports the **java.time** package, and this package contains **LocalDate, LocalTime, LocalDateTime, ZonedDateTime,** and other classes.
* This API provides better robustness, consistency and thread safety compared to legacy Date and Calendar classes.

**What is Optional equals() method in Java?**

In Java, the **equals()** method of the Optional class is used to compare two Optional objects for equality.

* It returns true if both the Optional objects contain the same value.
* And it returns false if both does not contain the same value.

**Illustration**:  
public class Main   
{  
 public static void main(String args[])   
{  
 // Creating Optional objects  
 Optional<String> opt1 = Optional.of("Sweta");  
 Optional<String> opt2 = Optional.of("Sweta");  
 Optional<String> opt3 = Optional.of("Dash");  
  
 // Comparing Optional objects  
 System.out.println(opt1.equals(opt2)); // true  
 System.out.println(opt1.equals(opt3)); // false  
 }  
}

**What is ArrayList forEach() method in Java?**

In Java, the forEach() method is used to iterate over each ArrayList element.

* It performs specified operation for each element.
* It simplifies iteration and shortens the code.
* It takes a Consumer as a parameter, which represents the action to be performed on each element.

ArrayList<Integer> numbers = new ArrayList<>();  
numbers.add(1);  
numbers.add(2);  
numbers.add(3);  
  
numbers.forEach(num -> System.out.println(num));  
  
**Output:**  
1  
2  
3

**How to find duplicate elements in a Stream in Java?**

**Count occurrence of a given character in a string using Stream API in Java.**

**How to get Slice of a Stream in Java?**

**How to Reverse elements of a Parallel Stream in Java?**

**Write a Program to Iterate over a Stream with Indices in Java 8.**

**What is method reference in Java 8?**

Method reference is a concise way to use a lambda expression for calling a method directly. It simplifies the code by providing a shorthand notation. are four types of method references that are listed below:

* Static Method Reference
* Instance Method Reference of a particular object
* Referencing an instance method of an unspecified object belonging to a specific class.
* Constructor Reference.

**Example:**

numList.stream().filter(n -> n > 5).sorted().forEach(System.out::println);

**What is MetaSpace in Java 8?**

In Java 8, Metaspace stores class metadata in native memory, separate from the heap. It can dynamically expand, overcoming size limitations, and enhances garbage collection efficiency, auto-tuning, and metadata distribution.

* It is used by the JVM to store metadata about loaded classes and methods.
* It replaces the PermGen space, offering dynamic allocation, separate memory management from the heap, and improved garbage collection, thereby mitigating PermGen space errors.

**Hibernate**

Use spring-data-jpa dependency

Hibernate is an ORM (Object-Relational Mapping) framework for Java that simplifies database interactions

**1. Why is Hibernate better than JDBC?**

* Reduces boilerplate code by mapping Java objects to database tables.
* Database-independent; no need to write database-specific SQL.
* Supports caching for better performance.

**2. What is a Session in Hibernate?**

* A Session is a short-lived object used to interact with the database.
* It is not thread-safe and is used to perform CRUD operations.
* Operations are executed when the transaction is committed or flushed.

**3. List and describe the Hibernate framework’s essential interfaces.**

1. **Session:** Handles database operations.
2. **SessionFactory:** Creates Session objects; is thread safe.
3. **Transaction:** Manages transactions.
4. **Query:** Executes HQL/SQL queries.

**4. What is the Hibernate Configuration File?**

* It specifies database connection properties, dialect, entity mappings, and caching settings.
* Example: hibernate.cfg.xml or hibernate.properties.

**5. What is an Entity in Hibernate?**

* A class representing a database table.
* Annotated with @Entity and fields are mapped to table columns.
* Example:

java

@Entity

public class Employee {

@Id

private int id;

private String name;

}

**6. What is ORM (Object-Relational Mapping)?**

* Maps Java objects to database tables.
* Makes database operations more object-oriented and simpler.

**7. Difference Between Session and SessionFactory**

| **Session** | **SessionFactory** |
| --- | --- |
| Short-lived | Long-lived |
| Not thread-safe | Thread-safe |
| Performs CRUD | Creates Session objects |

**8. What is HQL?**

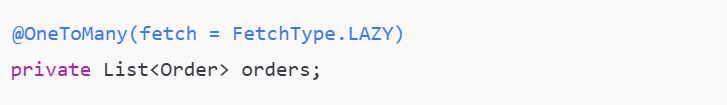
* Hibernate Query Language is an object-oriented query language.
* Works with entity attributes instead of table columns.
* Example: from Employee where department = 'HR'.

**9. Difference Between get() and load()**

| **get()** | **load()** |
| --- | --- |
| Fetches immediately | Returns a proxy, fetches lazily. |
| Returns null if not found | Throws exception if not found. |

**10. What is Lazy Loading?**

* Data is fetched only when accessed, not upfront.
* Saves memory and improves performance.
* Example: @OneToMany(fetch = FetchType.LAZY)



**What is the purpose of application.properties in a Spring Boot application?**

It is used to configure application settings, including Hibernate, database connections, logging, and more.

**How do you configure Hibernate dialect in application.properties?**

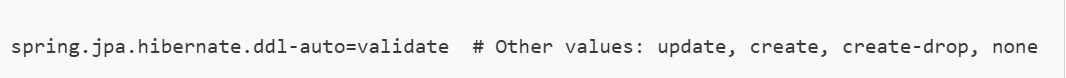
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

**How do you enable Hibernate’s SQL logging?**

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

**How do you configure Hibernate to validate or update the schema?**

****

**How do you configure the database connection pool in application.properties?**

**A screenshot of a computer code

Description automatically generated**

**How do you configure Hibernate batch size in application.properties?**

****

**How do you configure the connection timeout for the database?**

****

**What is the difference between spring.datasource and spring.jpa properties?**

* spring.datasource: Configures database connection details like URL, username, password, and driver.
* spring.jpa: Configures Hibernate-specific properties like SQL logging, dialect, and schema management.

**How do you configure multiple data sources in application.properties?**

****

**What are the advantages of Hibernate over other ORM tools?**

* **Database Independent:** Works with multiple databases using a dialect.
* **Reduces Boilerplate Code:** Simplifies persistence with object-oriented methods.
* **Caching Support:** Improves performance with first- and second-level caching.
* **HQL Support:** Object-oriented query language makes queries simpler.
* **Lazy Loading:** Loads data only when needed, saving resources.
* **Automatic Schema Generation:** Creates and manages tables automatically.

**Explain the Hibernate architecture and its key components.**

Hibernate has a layered architecture with the following components:

1. **SessionFactory:** A factory for creating Session objects; it is thread-safe and heavyweight.
2. **Session:** A lightweight, non-thread-safe object used to interact with the database.
3. **Transaction:** Handles atomic database operations.
4. **Query:** Used for HQL and SQL queries.
5. **Configuration:** Loads Hibernate settings and mappings.
6. **Cache:** Improves performance by reducing database access.

**What are the different states of an object in Hibernate?**

1. **Transient:** Object exists only in memory and is not associated with a database or Session.
2. **Persistent:** Object is associated with a Session and is synchronized with the database.
3. **Detached:** Object was once persistent but is now disconnected from the Session.

**What is the difference between save () and persist ()?**

| **save ()** |  | **persist ()** |
| --- | --- | --- |
| Returns the generated ID. |  | Does not return the ID. |
| Can be used outside a transaction. |  | Must be used within a transaction. |
| Inserts the object immediately. |  | Defers insert until flush/commit. |

**What is the difference between merge () and update ()?**

| **merge ()** | **update ()** |
| --- | --- |
| Merges changes from detached object into a persistent object. | Reassociates a detached object with the Session. |
| Does not throw an exception if the object is already in the Session. | Throws an exception if the object is already in the Session. |
| Returns a new persistent instance. | Does not return a new instance. |

**What is the difference between first-level cache and second-level cache in Hibernate?**

| **First-Level Cache** | **Second-Level Cache** |
| --- | --- |
| Enabled by default. | Must be explicitly enabled. |
| Specific to a Session. | Shared across Sessions. |
| Exists for the duration of a Session. | Exists for the duration of the SessionFactory. |

**What are the annotations used in Hibernate?**

1. **@Entity:** Marks a class as a database entity.
2. **@Table:** Specifies the table name.
3. **@Id:** Marks the primary key field.
4. **@GeneratedValue:** Configures how the primary key is generated.
5. **@Column:** Maps a field to a table column.
6. **@OneToOne, @OneToMany, @ManyToOne, @ManyToMany:** Define relationships.
7. **@JoinColumn:** Specifies the join column for relationships.
8. **@Transient:** Excludes a field from persistence.

**How does Hibernate handle database transactions?**

* Hibernate uses **ACID transactions** to ensure data consistency.
* Transactions are managed by the Transaction interface in Hibernate.
* Operations like save(), update(), and delete() are performed within a transaction.
* Commit (transaction.commit()) saves changes; rollback (transaction.rollback()) undoes changes.
* In Spring Boot, transactions are often managed with @Transactional.

**How would you handle versioning in Hibernate?**

* Versioning is used to manage concurrent updates to data.
* Add a @Version annotation to a field (e.g., int, long, or Timestamp) in your entity.
* Hibernate increments the version number with each update to detect conflicts.

A screen shot of a computer code

Description automatically generated

**How do you use composite keys in Hibernate?**

* Composite keys are created using multiple fields as a primary key.
* Use the @IdClass or @EmbeddedId annotations.

**A computer screen shot of a code

Description automatically generated**

**Example with @EmbeddedId:**

A screenshot of a computer code

Description automatically generated

**Options:**

1. **validate:** Validates schema but makes no changes.
2. **update:** Updates the schema without dropping existing data.
3. **create:** Drops and recreates the schema every time.
4. **create-drop:** Drops the schema at the end of the session.
5. **none:** Disables schema management.

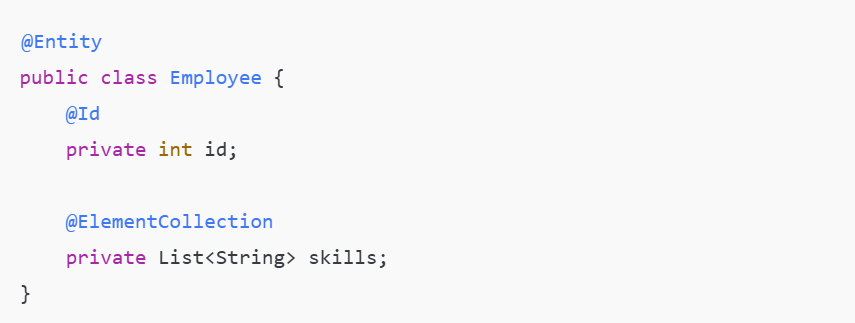
**Example:**

spring.jpa.hibernate.ddl-auto=update

**How do you map a collection of elements in Hibernate?**

* Use @ElementCollection for a collection of simple types or embeddable objects.
* Use @OneToMany or @ManyToMany for relationships with other entities.

**Example with @ElementCollection:**



**Example with @OneToMany:**

A computer code on a white background

Description automatically generated

**What are the common properties in the Hibernate configuration file?**

1. **Database connection settings:**

hibernate.connection.url=jdbc:mysql://localhost:3306/mydb

hibernate.connection.username=root

hibernate.connection.password=password

hibernate.connection.driver\_class=com.mysql.cj.jdbc.Driver

1. **Hibernate dialect:**

hibernate.dialect=org.hibernate.dialect.MySQLDialect

1. **Schema management:**

hibernate.hbm2ddl.auto=update

1. **Caching settings:**

hibernate.cache.use\_second\_level\_cache=true

hibernate.cache.provider\_class=org.hibernate.cache.EhCacheProvider

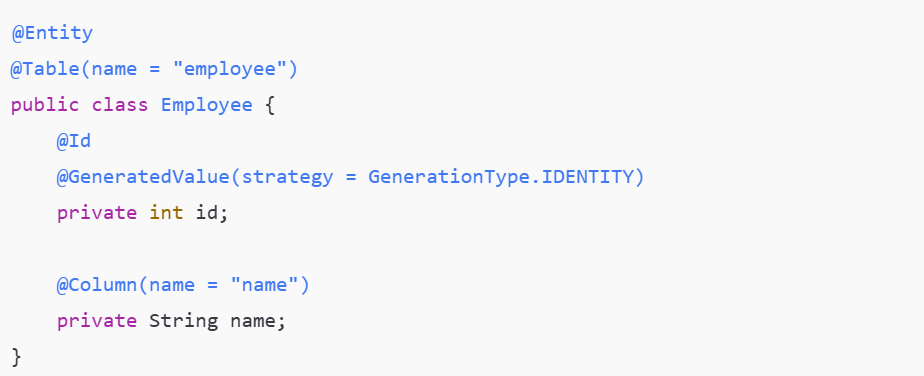
1. **SQL logging:**

hibernate.show\_sql=true

hibernate.format\_sql=true

**How do you configure Hibernate with annotations?**

* Add Hibernate annotations directly to your entity classes:



**What is the N+1 select problem in Hibernate and how do you solve it?**

* **Problem:** Occurs when a query retrieves a collection, and Hibernate executes additional queries for each related entity (e.g., one query for the main entity and N queries for its relationships).
* **Solution:** Use **eager fetching** or **batch fetching** with @OneToMany(fetch = FetchType.EAGER) or JOIN FETCH in HQL.

**What is the difference between native SQL and HQL in Hibernate?**

| **Native SQL** | **HQL** |
| --- | --- |
| Uses plain SQL queries. | Object-oriented query language. |
| Works with database tables. | Works with entity objects. |
| Database-specific. | Database-independent. |

**What are proxies in Hibernate?**

* A proxy is a placeholder object created by Hibernate to support **lazy loading**.
* The proxy initializes the actual object when needed, reducing memory and performance costs.

**What is the role of @Entity, @Table, and @Column annotations in Hibernate?**

* **@Entity:** Marks a class as a database entity.
* **@Table:** Maps the entity to a specific database table (optional).
* **@Column:** Maps a class field to a specific database column (optional).

**Example:**

A screenshot of a computer code

Description automatically generated

**9. Explain the concept of Interceptor and Event Listeners in Hibernate.**

* **Interceptor:** Allows custom logic to be executed during Hibernate operations like save, delete, or update. Implement Interceptor interface to define custom behavior.
* **Event Listener:** A more advanced feature that hooks into Hibernate's lifecycle events, like pre-insert or post-update.

**Example of Interceptor:**

public class MyInterceptor extends EmptyInterceptor {

@Override

public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types) {

System.out.println("Entity saved: " + entity);

return false;

}

}

**==============================================================================**

**ANNOTATIONS**

**@SpringBootApplication**

This annotation is used to mark the main class of a Spring Boot application. It encapsulates @SpringBootConfiguration, @EnableAutoConfiguration, and @ComponentScan annotations with their default attributes.

**@EnableAutoConfiguration**

It automatically creates, and registers beans based on both the included jar files in the classpath and the beans defined by us.

**@ComponentScan**

@ComponentScan is an annotation used in the Spring Framework for auto-detecting and registering Spring-managed components (e.g. beans, controllers, services, repositories, etc.) within a specified package or set of packages.

**@Configuration**

The @Configuration annotation in Spring marks a class as a configuration class that provides bean definitions.

**Stereotype Annotations**

1. **Component**

without having to write any explicit code, Spring will:

Scan our application for classes annotated with @Component

Instantiate them and inject any specified dependencies into them.

1. Service
2. RestController/Controller
3. Repository

**Spring Core Annotations**

1. Configuration
2. Bean
3. Autowired
4. Qualifier
5. primary
6. Lazy
7. Value
8. PropertySource
9. ConfigurationProperties
10. Profile
11. Scope

**@Configuration:**

The @Configuration annotation in Spring marks a class as a configuration class that provides bean definitions.

**@Bean**

It indicates that a method produces a bean to be managed by the Spring Container. It is usually declared in Configuration class to create Spring bean definitions.

**@Qualifier**

The @Qualifier annotation is used to resolve the autowiring conflict, when there are multiple beans of same type.

**@Primary**

It indicates that a bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency.

**@Lazy**

It indicates that a bean should be lazily initialized, meaning it will only be created when it's first requested, rather than during the application context initialization.

**@Value**

@Value is a core annotation in Spring that is primarily used for assigning default values to variables and method parameters.

**@PropertySource**

It is used to provide properties file to Spring Environment.

**@ConfigurationProperties**

to map or bind the . properties or yml configuration values to Java objects. (with prefix)

**@Profile**

to resolve the challenge of controlling which parts of our application should be active under a particular set of conditions or environments.

**@Scope**

It indicates the lifecycle of an instance, such as singleton or prototype.

**REST API ANNOTATIONS**

1. RestController
2. Controller
3. RequestMapping
4. GetMapping
5. PostMapping
6. PutMapping
7. DeleteMapping
8. RequestBody
9. PathVariable
10. RequestParam
11. Controller Advice & ExceptionHandler

**@Controller**

@Controller is used to define a Spring MVC controller that processes HTTP requests and typically returns a view (like a JSP, Thymeleaf, or HTML page).

Methods in a @Controller class usually return the name of a view or a ModelAndView object. The returned view name is resolved by a view resolver to render the HTML page or template.

**@RestController**

@RestController is a specialized version of @Controller that is used for creating RESTful web services. It combines @Controller and @ResponseBody, meaning that the return values of methods are directly written to the HTTP response body rather than being resolved as a view name.

Methods in a @RestController class typically return objects, which are automatically serialized into JSON or XML (based on the content type requested) and sent in the HTTP response body.

**@RequestMapping**

This is an annotation used to map web requests to specific handler methods or classes. It can be applied at the class level or method level in a Spring controller.

**@ControllerAdvice**

It is used to define global exception handlers, model attribute handlers, and binding handlers that apply to all controllers or a subset of controllers in your application.

**@ExceptionHandler**

It is used to handle exceptions thrown by a specific controller method or across a set of controllers. It provides a way to define methods that should be called when a certain type of exception is thrown.

**SPRING DATA JPA**

1. Entity
2. Table
3. Column
4. Transactional

**==============================================================================**

**JUnit**

**Unit testing** is a software testing technique where individual components or units of a software application are tested in isolation to ensure they work as expected.

**Junit:**

* Unit testing framework for java and it’s framework.
* Junit5 is the next generation of JUnit.
* The goal is to create an up-to-date foundation for developer-side testing on the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing.

**AssertJ:**

* AssertJ is a Java library that provides a rich set of assertions and truly helpful error messages.
* It improves test code readability and is designed to be super easy to use within your favorite IDE.

**Mockito**

* Mockito is mocking framework for Java
* It lets you write beautiful tests with a clean & simple API.
* The tests are very readable, and they produce clean verification errors.
* Mockito 3.x requires Java 8 or above.

**What is the difference between @Test and @Before annotations in JUnit?**

* @Test: Marks a method as a test method to be run by the JUnit framework.
* @Before: Marks a method to be executed before each test case. It's typically used to set up the test environment (e.g., initializing objects).

**What is the purpose of @After annotation in JUnit?**

* @After is used to define a method that runs after each test case. It is generally used for cleanup, like closing resources or resetting values.

**Explain the @BeforeClass and @AfterClass annotations.**

* @BeforeClass: This annotation is used to mark a method that should be executed once before any of the test methods in the class are run.
* @AfterClass: This annotation marks a method to be executed once after all test methods in the class have been run.

**What is a test suite in JUnit?**

* A test suite is a collection of test cases that can be run together. In JUnit, you can use @RunWith and @Suite.SuiteClasses to define a suite of tests.

**What are assertions in JUnit?**

Assertions are used to check the expected result against the actual result of the test case. Common assertions include:

* + assertEquals(expected, actual)
  + assertTrue(condition)
  + assertFalse(condition)
  + assertNotNull(object)
  + assertNull(object)

**How do you handle exceptions in JUnit tests?**

In JUnit, you can test for exceptions using the expected attribute of the @Test annotation or use try-catch blocks:

* + @Test(expected = Exception.class)
  + Alternatively, use assertThrows in JUnit 5 for more control.

**What is the difference between JUnit 4 and JUnit 5?**

* JUnit 4 uses annotations like @Test, @Before, and @After, while JUnit 5 introduces new annotations (@BeforeEach, @AfterEach, etc.) and supports Lambda expressions for more flexible tests.
* JUnit 5 also introduces the concept of extensions (similar to JUnit 4's Rules) and allows for better integration with modern IDEs and build tools.

**What is @ParameterizedTest in JUnit 5?**

* @ParameterizedTest allows you to run a test multiple times with different inputs. It is used when you want to test the same logic under various conditions or inputs.

**What is the use of @Disabled annotation?**

* The @Disabled annotation is used to disable a test method or class. The test will not run, and the framework will report it as skipped.

**How do you test private methods in JUnit?**

Private methods can be tested in two ways:

* + **Reflection**: You can use Java reflection to access and invoke private methods.
  + **Refactor to package-private or protected**: This makes it easier to test the method within the same package or subclass.

**What is mocking in JUnit?**

* Mocking is a technique used to simulate the behavior of real objects to test specific parts of a program in isolation. Frameworks like **Mockito** are often used in combination with JUnit for creating mock objects and verifying interactions with dependencies.

**What is a test double?**

* A test double is an object that takes the place of a real object in the context of a unit test. Types of test doubles include:
  + **Mock**: A mock object simulates the behavior of real objects and verifies interactions.
  + **Stub**: A stub is a simple implementation of an interface that returns predefined responses.
  + **Spy**: A spy records the calls made to it and can return values.

**What is the use of @Timeout in JUnit 5?**

* The @Timeout annotation is used to specify that a test should complete within a given time limit. If the test takes longer than the specified time, it will fail.

**Explain the difference between assertEquals() and assertSame() in JUnit.**

* + assertEquals(expected, actual): Checks if the expected value is equal to the actual value based on the equals() method.
  + assertSame(expected, actual): Checks if the expected and actual objects are the same (i.e., they refer to the same memory location).

**What is the role of @TestInstance in JUnit 5?**

* @TestInstance specifies the lifecycle of test instances. By default, JUnit 5 creates a new instance of the test class for each test method. Using @TestInstance(Lifecycle.PER\_CLASS) makes JUnit create only one instance of the test class for all test methods.

**How do you run JUnit tests from the command line?**

* You can run JUnit tests from the command line using build tools like **Maven** or **Gradle**:
  + **Maven**: mvn test
  + **Gradle**: gradle test

**What are assumptions in JUnit?**

* Assumptions are used to specify conditions that must be true for a test to execute. If an assumption fails, the test is ignored (skipped). For example, Assume.assumeTrue(condition) will skip the test if the condition is false.

**What is the @EnableJUnit4 annotation in JUnit 5?**

* @EnableJUnit4 is used to run JUnit 4 tests in a JUnit 5 environment, allowing backward compatibility with existing JUnit 4 tests.

**How can you test the performance of a method in JUnit?**

* You can test the performance of a method using @Test(timeout = <time\_in\_ms>) in JUnit 4, or by using a custom performance extension or tool in JUnit 5.

**==============================================================================**

**What is Serialization and Deserialization in Java with Example?**

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

**A computer code on a white background

AI-generated content may be incorrect.**

**Mutable vs Immutable - Key Differences**

Mutable Objects (Can Change)

* State can be modified after creation.
* Examples: ArrayList, HashMap, StringBuilder, StringBuffer
* Use Case: When you need to frequently update data

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Immutable Objects (Cannot Change)**

* State cannot be modified after creation.
* Examples: String, List.of(), Set.of(), Map.of(), toList() (Java 16+), Collectors.toUnmodifiableList()
* Use Case: When you need thread safety or data consistency.

**A screenshot of a computer code

AI-generated content may be incorrect.**

**What is spring framework and its advantages?**

It is an open-source framework for building Java-based applications. It provides tools and features to simplify enterprise development.

**Key Features:**

1. **Dependency Injection (DI)** – Manages object creation and dependencies.
2. **Aspect-Oriented Programming (AOP)** – Separates cross-cutting concerns like logging and security from the business logic, making the code more modular and easier to maintain.
3. **Spring MVC** – A framework for building web applications.
4. **Spring Boot** – Simplifies app setup with pre-configured templates.
5. **Spring Security** – Handles authentication and authorization.
6. **Data Access** – Simplifies working with databases and integrates with technologies like JDBC, Hibernate, and JPA.

**Advantages:**

* **Loose Coupling** – Makes components independent and easier to maintain.
* **Testability** – Encourages writing testable code.
* **Comprehensive** – Offers built-in support for many tasks (data access, security, etc.).
* **Flexibility** – Works with various platforms and technologies.

**Where does custom exception fits in?**

Custom exceptions in a Spring application are used to handle specific error conditions that aren’t covered by standard exceptions. They help you manage application-specific errors and provide meaningful feedback**.**

**A screen shot of a computer

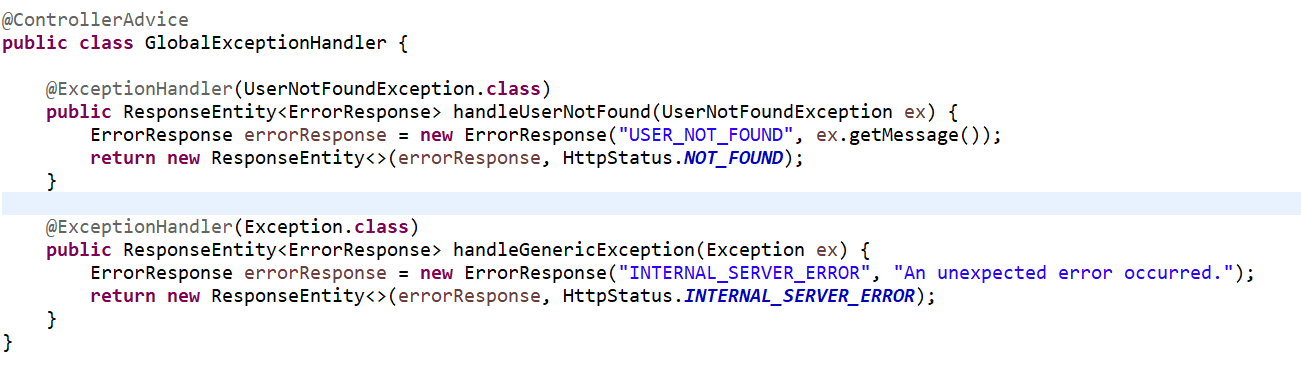
AI-generated content may be incorrect.**

**A close-up of a computer screen

AI-generated content may be incorrect.**

**Handling Custom Exceptions with @ControllerAdvice**

In a Spring application, you can use the @ControllerAdvice annotation to handle custom exceptions globally and return a meaningful response to the client.



**What is unit testing?**

**Unit testing** is a software testing technique where individual components or units of a software application are tested in isolation to ensure they work as expected.

**What is @SpringbootApplication purpose?**

It combines three main annotations:

1. **@Configuration**: Marks the class as a configuration class, meaning it can contain bean definitions.
2. **@EnableAutoConfiguration**: Tells Spring Boot to automatically configure the application based on the dependencies present in the classpath. For example, if spring-boot-starter-data-jpa is included, it automatically configures JPA support.
3. **@ComponentScan**: Tells Spring to scan the current package (and its sub-packages) for Spring components like @Component, @Service, @Repository, etc., and register them as beans in the application context.
4.  **@SpringBootApplication** is typically placed on the main class of a Spring Boot application (usually the one with the main() method).
5.  It simplifies configuration by combining common annotations and allows you to run a Spring Boot application without additional boilerplate code.
6.  The main job of the class marked with @SpringBootApplication is to launch the Spring Boot application.

**12)What is Purpose of RestController?**

The **@RestController** annotation in Spring is used to define a RESTful web service controller. It is a convenience annotation that combines two important annotations in Spring:

1. **@Controller**: Marks the class as a Spring MVC controller that can handle HTTP requests.
2. **@ResponseBody**: Indicates that the return values of methods in the class should be automatically converted to JSON or XML and sent as the HTTP response body, rather than being rendered as a view.

**How do you achieve Dependency Injection in Spring?**

**Dependency Injection (DI)** in Spring is a core concept that allows you to inject dependencies (such as services, repositories, or components) into a class, rather than creating them directly within the class.

Spring provides several ways to achieve dependency injection, but the most common methods are **Constructor Injection**, **Setter Injection**, and **Field Injection**.

1. **Constructor Injection**



2. **Setter Injection**:

A screenshot of a computer program

AI-generated content may be incorrect.

3. **Field Injection**:

A screenshot of a computer program

AI-generated content may be incorrect.

**What is the purpose of SpringBootApplication?**

The purpose of a **Spring Boot application** is to provide a **quick, easy, and efficient way to create stand-alone, production-ready applications** with minimal configuration. It is part of the **Spring Framework** and simplifies the process of setting up and running Spring applications.

 **Quick Setup**: You can get started with minimal configuration.

 **Embedded Server**: It includes a built-in web server (like Tomcat), so no need for external setup.

 **Auto-Configuration**: Automatically configures your application based on the dependencies you add.

**How do you customize the serialization?**

use transient keyword

**What is the best strategy of sorting elements in ArrayList?**

Collections.sort(list);

Collections.sort(list, Comparator.comparingInt(person -> person.age));

**what is RestControllerAdvice? And the purpose?**

@RestControllerAdvice is a specialized version of @ControllerAdvice in Spring, specifically used in RESTful web services. It is a convenience annotation that combines **@ControllerAdvice** and **@ResponseBody**. It allows you to handle exceptions globally and return appropriate responses, without needing to manually handle exceptions in every individual controller.