**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
* LDAP-based Authentication
* Session-based Authentication (e.g., Spring Security)

**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, and integration with token-based systems like OAuth2.

**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?**

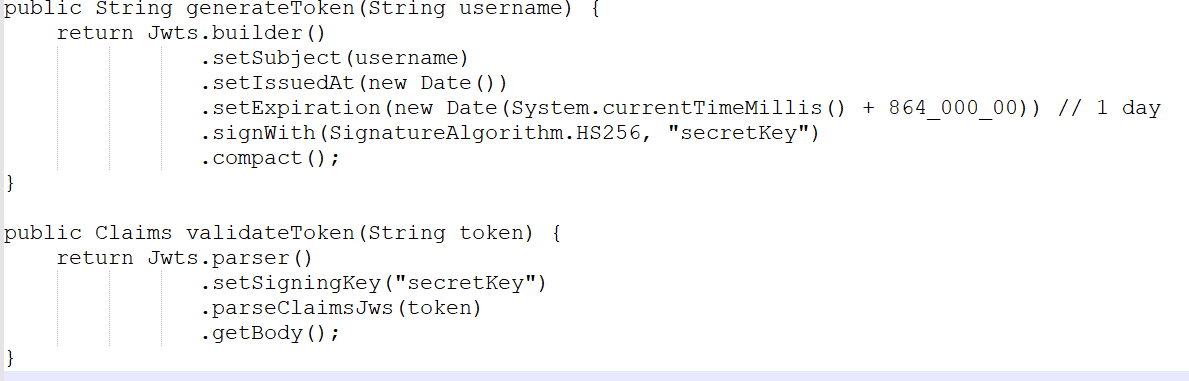
A JWT is a compact, URL-safe token format used to securely transmit information between parties. It is commonly used for authentication in stateless systems.

**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?**



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

* + 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 can JWT be invalidated after issuance?**

* + Store token IDs in a database or cache and check against them on each request.
  + Change the signing key.
  + Use short expiration times and rely on refresh tokens.

**What are the pros and cons of using JWT over session-based authentication?**

* + **Pros:**
    1. Stateless (no server-side storage).
    2. Easy to scale.
  + **Cons:**
    1. Larger token size.
    2. Cannot easily revoke 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:**

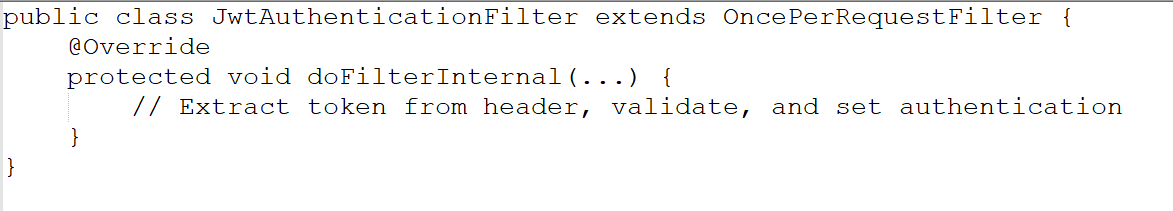
* **Session-Based:** Server stores user sessions in memory or a database, tied to a session ID stored in a browser cookie.
* **Token-Based:** Server issues a token (e.g., JWT) after authentication. The client sends this token in subsequent requests, and the server doesn't need to store session state.  
  **Scalability:** Token-based is more scalable since it doesn’t require server-side state management.

**Generate Token:**

**A close up of a text

Description automatically generated**

**Filter for JWT Validation:**

****

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

**RESTful API**

**What is a RESTful API?**

A RESTful API (Representational State Transfer) is an architectural style for designing networked applications. It uses standard HTTP methods and is stateless. Resources are represented via URLs.

**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

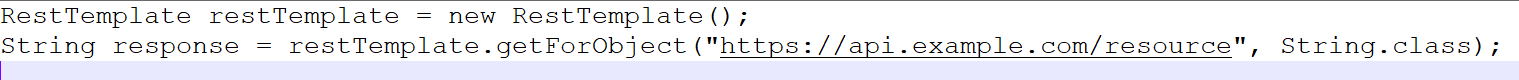
Description automatically generated**

Use JWT for token-based security.

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

Using libraries like RestTemplate, WebClient, or HttpURLConnection.

**Example with RestTemplate:**

****

**What is HATEOAS?**

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

**Example:**

**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).

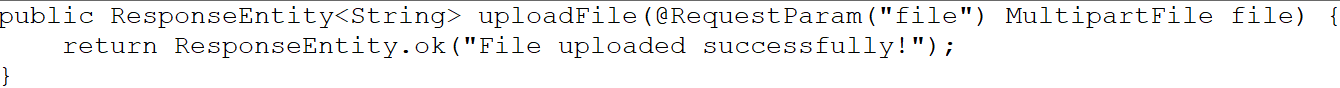
Optimize database queries.

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
* CompletableFuture
* Java DateTime API
* Method Reference
* Comparable and Comparator
* Optional Class
* Date/Time API

**What is Lambda Expression?**

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

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

Stream API is used to process collections of objects. Streams are designed to be efficient and can support improving your 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 is Functional Interface in Java 8?**

An interface with only one abstract method is known as a functional interface but there is no restriction, in a functional interface you can have n number of default methods and static methods.

**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.

**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. You may observe the thing by demonstrating a sample code in your local Java IDE.

**What are the advantages of Lambda Expression?**

* Avoid writing anonymous implementation
* Saves a lot of code
* Code is directly readable without interpretation

**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

Description automatically generated

**What are all functional interfaces introduced in Java 8?**

* Function
* Predicate
* Consumer
* Supplier

**Tell a few stream methods you used in your project?**

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

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

* Hard to use without an IDE
* Complex to debug

**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 contains the same value.
* And, it returns false if both does not contain the same value.

**Illustration**:

import java.util.Optional;  
  
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 Default Methods In Java 8?**

In Java 8, Default methods allows interfaces to have method implementations. This means that interfaces can contain concrete methods along with the abstract methods. The Default methods are defined using the **default** keyword.

**Illustration:**

interface Vehicle   
{  
 // Abstract method  
 void start();  
  
 // Default method  
 default void stop()   
{  
 System.out.println("Vehicle stopped");  
 }  
}  
  
class Car implements Vehicle   
{  
 @Override  
 public void start()   
{  
 System.out.println("Car started");  
 }  
}  
  
public class Main   
{  
 public static void main(String args[])   
{  
 Car car = new Car();  
 car.start(); // Output: Car started  
 car.stop(); // Output: Vehicle stopped  
 }  
}

**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.
* This allows us to write expressive and concise code.

**Illustration:**

// 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();  
 }  
}

**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**

**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.
* Provides transaction management and lazy loading.
* Simplifies schema generation and query writing (via HQL).

**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. **Configuration:** Configures Hibernate settings.
4. **Transaction:** Manages transactions.
5. **Query:** Executes HQL/SQL queries.
6. **Criteria:** Builds programmatic 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.
* Eliminates the need to write SQL.
* 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?**

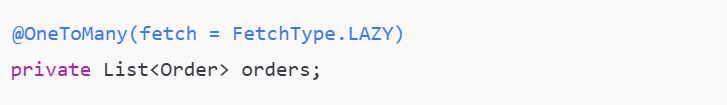
* Hibernate Query Language is an object-oriented query language.
* Works with entity attributes instead of table columns.
* Example: from Employee where department = 'HR'.

**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?**

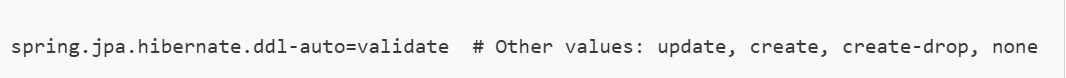
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

**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

**3. What is hibernate.hbm2ddl.auto and its options?**

* The hibernate.hbm2ddl.auto property controls how Hibernate handles database schema creation and updates.

**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.

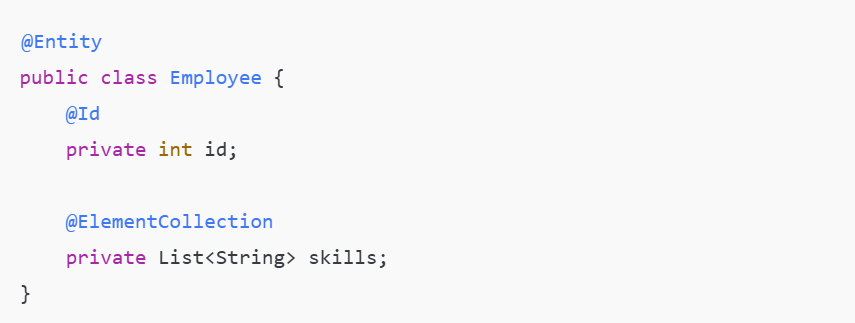
**Example:**

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

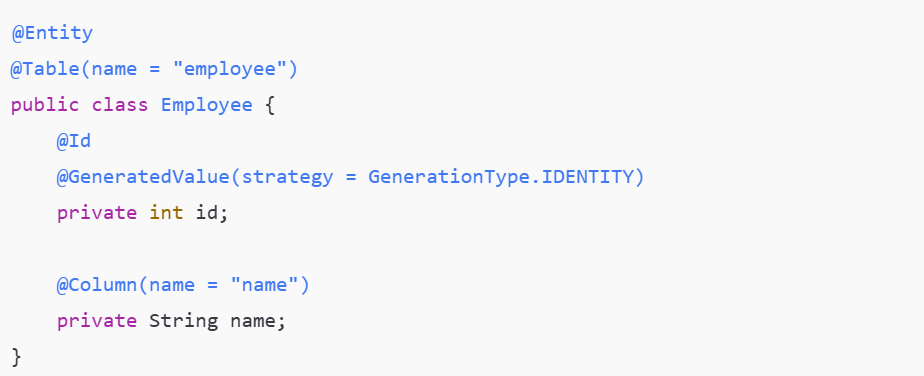
1. **SQL logging:**

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;

}

}

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

**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.