how to improve scalability of application and performance optimization in java applications also...

For scalable applications and performance optimization in Java, focus on architecture, code efficiency, memory management, and concurrent processing. To resolve 100% CPU usage, you must first identify and then address the root cause, which can range from inefficient code to suboptimal garbage collection.

How to improve scalability

Scalability is the ability of an application to handle a growing amount of work. It is typically achieved through one of two methods:

- **Vertical scaling**: Strengthening an existing server with more resources like CPU and RAM.
- **Horizontal scaling**: Adding more servers to distribute the load.

To implement horizontal scaling, use the following strategies:

- **Adopt a microservices architecture**: Break down a monolithic application into smaller, independently deployable services. This allows you to scale individual components based on their specific demand.
- **Implement load balancing**: Use load balancers to distribute incoming network traffic across multiple servers to prevent any single server from becoming overwhelmed.
- **Design stateless services**: A stateless service does not retain client data between requests, meaning any instance can handle any request. This simplifies the process of adding new servers to meet demand.
- **Use asynchronous processing**: Offload heavy or time-consuming tasks to be processed in the background using message queues (e.g., RabbitMQ, Kafka). This allows the application to respond quickly to user requests while the work is being done.
- **Leverage cloud-native features**: Deploying on cloud platforms like AWS or Azure enables automatic scaling, which dynamically adjusts resources based on performance metrics.
- **Choose the right database strategy**: Options for scaling your database include read replicas, partitioning, and sharding to distribute data across multiple servers. Consider using NoSQL databases for workloads that require high throughput and flexible schemas.

Java performance optimization

Optimizing application performance in Java requires attention to JVM tuning, efficient coding, and effective resource management.

JVM and memory optimization

- **Tune the garbage collector (GC)**: The GC can cause performance issues if not configured properly. Modern JVMs offer various garbage collectors, such as G1 (the default since Java 9) and ZGC, which are designed to minimize pause times. Use JVM flags like `-Xms` and `-Xmx` to set appropriate heap sizes.
- **Monitor GC activity**: Use tools like JConsole, VisualVM, or GCViewer to analyze garbage collection logs. This helps identify inefficiencies and adjust GC parameters for optimal performance.

Code optimization

- **Use efficient data structures**: Choose the right data structure for your task. For

`int` and `double`. They are faster and consume less memory than their object-oriented counterparts (`Integer`, `Double`), which are stored on the heap.

- **Optimize loops and iterations**: Cache collection sizes before loops to avoid repeated lookups. Consider whether the Streams API or a traditional `for` loop is more performant for your specific use case, especially with smaller data sets.
- **Be cautious with recursion**: In Java, recursive calls can be costly and lead to `StackOverflowError` if not handled carefully. Use iterative solutions with loops where possible.
- **Improve database interactions**: Use connection pooling, optimize SQL queries with proper indexing, and use `PreparedStatement` instead of `Statement`.

Concurrency and parallelism

- **Use a thread pool**: Instead of creating a new thread for every task, use `ExecutorService` to manage a pool of worker threads. This reduces the overhead of thread creation and management.
- **Avoid unnecessary synchronization**: The `synchronized` keyword enforces a lock that can block other threads, creating performance bottlenecks. Use concurrent collections like `ConcurrentHashMap` for better performance.
- **Embrace reactive programming**: For I/O-bound applications, use libraries like Project Reactor or RxJava to handle asynchronous data streams more efficiently.

How to solve 100% CPU usage

When your Java application hits 100% CPU, it's a symptom of a deeper issue. Simply restarting the application or adding more resources will not fix the underlying problem.

Identify the cause

- **Use system monitoring tools**: Start by using system tools like `top` or `htop` (on Linux) to confirm the specific process ID (PID) of the JVM consuming high CPU.
- **Generate thread dumps**: Use the `jstack <PID>` command to generate a thread dump. This shows the state of every thread and its stack trace.
- **Analyze the thread dump**:
  - Look for threads in the `RUNNABLE` state that are consuming a high amount of CPU. You can often convert the native thread ID from the `top` output to hexadecimal and match it in the `jstack` output to find the exact thread causing the problem.
  - Examine the stack trace of these high-CPU threads to identify the specific methods and lines of code executing the heavy work.
- **Use a profiler**: For a more visual and in-depth analysis, use a profiler like JProfiler, VisualVM, or async-profiler. These tools can show you real-time CPU usage, thread activity, and even generate flame graphs to visualize the call stack.

Common causes and fixes

- **Garbage collection overhead**: If frequent GC cycles are consuming CPU, it can indicate a memory leak or an inadequately sized heap.
  - **Fix**: Profile memory usage to find memory leaks and tune GC settings by adjusting heap size and choosing a more suitable GC algorithm.
- **Infinite loops or inefficient algorithms**: A thread stuck in an infinite loop or a poorly optimized, CPU-intensive algorithm will max out a CPU core.
  - **Fix**: The thread dump and profiler will point to the problematic method. Analyze and

I/O. Optimize database queries or use asynchronous I/O frameworks.

Hibernate implementatuion in SpringBoot project all adnavce feature and realtime problems occur...

Prepare Top 40 Advanced Hibernate annotation based real time scenrio question while modeling...

Please provide the details answers to the above problems topic wise in sections Data modeling an...

Data modeling and mapping

1. Composite primary keys

- **Scenario**: For a `FlightTicket` entity, a simple ID is not enough. You need a composite key combining `flightId` and `seatNumber`.
- **Solution**: Use either `@IdClass` or `@EmbeddedId`.
  - `@IdClass`: Define a separate `FlightTicketPK` class with `flightId` and `seatNumber` and use `@IdClass(FlightTicketPK.class)` on the entity. The entity's primary key properties are also annotated with `@Id`.
  - `@EmbeddedId`: Create an embeddable class (`FlightTicketId`) containing the key fields and annotate it with `@Embeddable`. In the `FlightTicket` entity, use `@EmbeddedId` on a field of this type.
- **Implementation**: `@EmbeddedId` is often preferred as it cleanly encapsulates the composite key logic in a single object. The `Id` class for both must implement `Serializable` and have proper `equals()` and `hashCode()` methods.

2. Embedded entities

- **Scenario**: A `User` entity has a complex `Address` composed of `street`, `city`, and `zipcode`.
- **Solution**: Use `@Embeddable` on the `Address` class and `@Embedded` on the `Address` field in the `User` entity. Hibernate maps the `Address` properties to columns in the `User` table.
- **Customization**: To map the same embeddable object multiple times (e.g., `shippingAddress` and `billingAddress`), use `@AttributeOverrides` with `@AttributeOverride` to specify unique column names for each embedded instance.

3. Inheritance mapping (`JOINED`)

- **Scenario**: A `Vehicle` base class has subclasses `Car` and `Truck`.
- **Solution**: Annotate the `Vehicle` base class with `@Inheritance(strategy = InheritanceType.JOINED)`. Each class gets its own table, and a `JOIN` is used to assemble the full object.
- **Pros/Cons vs.** `SINGLE_TABLE`:
  - **Pros**: Less data redundancy, better normalization, no wasted space with null

- **Solution**: Implement Hibernate's `UserType` interface. The `nullSafeSet()` and `nullSafeGet()` methods handle the conversion between the Java object and the database column value. You can then register this custom type for use with `@Type`.

5. Polymorphic queries

- **Scenario**: Query all `Vehicle` entities and get the specific subclass instances.
- **Solution**: A standard `HQL` or `JPQL` query on the base class handles this. Hibernate's persistence context automatically materializes the correct subclass object.
- **Example**: `entityManager.createQuery("from Vehicle").getResultList()`.

6. Read-only entities (`@Immutable`)

- **Scenario**: Reference static data like a product catalog without updates.
- **Solution**: Annotate the entity class with `@Immutable`. Hibernate will not perform dirty checks on this entity, saving CPU cycles. It will also skip any update operations.
- **Benefits**: Increased performance for static data by bypassing the overhead of state management.

7. Dynamic naming (`NamingStrategy`)

- **Scenario**: Automatically convert `camelCase` Java fields to `snake_case` database column names.
- **Solution**: Configure a `PhysicalNamingStrategy` in `application.properties`. Spring Boot's default `SpringPhysicalNamingStrategy` does this out of the box, but you can create a custom one by extending `PhysicalNamingStrategy` and setting `spring.jpa.properties.hibernate.physical_naming_strategy`.

8. Auditing (`@CreationTimestamp`, `@UpdateTimestamp`)

- **Scenario**: Automatically manage `createdAt` and `updatedAt` columns.
- **Solution**: Use the `@CreationTimestamp` and `@UpdateTimestamp` annotations from the `org.hibernate.annotations` package on `LocalDateTime` or `Instant` fields.
- **Benefits vs. Manual**: These annotations remove the boilerplate code of manually setting timestamps in service methods or listeners, reducing errors and ensuring consistency.

9. Mapping enums

- **Scenario**: Map a Java `Enum` to a database column.
- **Solution**:
  - **Ordinal**: The default behavior, mapping the enum to its numeric index. Use `@Enumerated(EnumType.ORDINAL)`. This is space-efficient but fragile if the enum order changes.
  - **String**: Use `@Enumerated(EnumType.STRING)`. This maps the enum to its name string. It's more robust to changes in enum order but uses more storage.

10. Mapping immutable collections

- **Scenario**: Map an immutable `Set<String>` to a join table.
- **Solution**: Use `@ElementCollection`. This creates a separate table to store the collection elements. Combine this with the `@Immutable` annotation to prevent changes to the collection. For immutability, ensure no setter methods are provided for the collection field.

12. Cascade operations (`@Cascade`)

- **Scenario**: Delete all `Child` entities when the `Parent` is deleted.
- **Solution**: In the `@OneToMany` annotation on the `Parent` entity, set `cascade = CascadeType.REMOVE` . This ensures that when the parent entity is removed, the operation cascades to the associated child entities.

13. Bi-directional association maintenance

- **Scenario**: Adding a `Comment` to a `Post` .
- **Mistake**: Adding the `Comment` to the `post.getComments()` list but not setting the `comment.setPost()` reference.
- **Fix**: Always synchronize both sides of a bidirectional relationship within a helper method. The `mappedBy` attribute designates the non-owning side, but both sides must be updated for consistency.

14. Orphan removal

- **Scenario**: An `Order` entity is removed from a `User` 's order list.
- **Solution**: Use `@OneToMany(orphanRemoval = true)` . This marks child entities for deletion if they are no longer associated with a parent in the collection.
- **Difference from `CascadeType.REMOVE` **: `orphanRemoval` deletes a child when its reference is removed from the parent's collection, even if the parent entity still exists. `CascadeType.REMOVE` only deletes children when the parent is explicitly removed.

15. Fetch type strategy

- **Scenario**: A `User` has a `List<Order>` .
- **Solution**: Use `FetchType.LAZY` for the `List<Order>` unless the orders are *always* needed with the user. `EAGER` fetching should be used sparingly, primarily for `OneToOne` or small, frequently accessed associations.

Transaction management and concurrency

16. Optimistic locking

- **Scenario**: High-concurrency update of a single entity.
- **Solution**: Add a `@Version` annotated field (e.g., `Long` or `Integer`) to the entity. Hibernate increments this version on each update. If a concurrent transaction tries to save an outdated version, an `OptimisticLockException` is thrown.

17. Transaction isolation

- **Scenario**: Manage data consistency in concurrent transactions.
- **Solution**: The default isolation level in Spring is `DEFAULT` , which depends on the database. To override, use `@Transactional(isolation = Isolation.REPEATABLE_READ)` . Use `REPEATABLE_READ` to prevent dirty and non-repeatable reads.

18. Deadlock resolution

- **Debugging**:
  - Enable SQL logging to see the exact queries.
  - Monitor database logs for deadlock information.
  - Analyze thread dumps to identify which locks are held.

19. Programmatic transactions

- **Scenario**: Non-standard transaction management is needed.
- **Solution**: Use Spring's `TransactionTemplate` injected into your service. Call the `execute()` method with a `TransactionCallback` to define the transactional operations.

20. Transaction propagation

- `REQUIRED` vs. `REQUIRES_NEW`:
  - `REQUIRED`: Joins an existing transaction or starts a new one if none exists. This is the default and most common behavior.
  - `REQUIRES_NEW`: Always starts a new, independent transaction. The old transaction is suspended until the new one completes. Use this for operations that must succeed or fail independently.

Fetching strategies and query optimization

21. N+1 select problem

- **Scenario**: Fetching a list of `Store` entities and their `Product` collections, which are lazy-loaded.
- **Solution**: Use `@EntityGraph` to define a named graph that eagerly fetches related entities. Alternatively, use `JOIN FETCH` in a JPQL query.

22. Batch fetching

- **Scenario**: Fetching a large number of lazy-loaded child entities.
- **Solution**: Use `@BatchSize(size = 25)` on the `Store`'s `products` collection. Hibernate will load the products for multiple stores in a single query, significantly reducing the number of round trips.

23. Fetching collections for DTOs

- **Scenario**: Read-only DTO projection.
- **Solution**: Use `SELECT NEW` in a JPQL query. This constructs DTO objects directly from the query results, bypassing the persistence context and avoiding the overhead of managing entities.

24. Multi-bag fetch exception

- **Problem**: Joining multiple lazy-loaded `List` collections in a single query.
- **Cause**: Hibernate cannot handle multiple "bags" (`List`) efficiently and gets confused about which to populate.
- **Solution**:
  1. Change the collection type from `List` to `Set` if the order is not important.
  2. Use multiple queries with different `JOIN FETCH` clauses to retrieve the collections separately.

25. Named queries

- **Scenario**: Frequently used or complex queries.
- **Solution**: Use `@NamedQuery` or `@NamedNativeQuery` at the class level.
- **Benefits**: Query centralization, compile-time checking, and clearer separation of

Caching

27. Second-level caching

- **Scenario**: Frequently read, infrequently updated entities.
- **Solution**:
  1. Enable second-level cache in `application.properties`.
  2. Annotate the `Category` entity with `@Cacheable` and configure the cache region.

28. Query cache

- **Scenario**: Queries that are executed frequently with the same parameters.
- **Solution**:
  1. Enable the query cache in properties.
  2. Use `@QueryHint` in your repository method with the hint `org.hibernate.cacheable` set to `true`.
- **Trade-offs**: Increased memory usage to store query results, which may not be worthwhile for queries with highly variable parameters.

29. Detached entity and caching

- **Scenario**: A detached entity is retrieved from the cache and modified in a new transaction.
- **How it works**: The entity is detached, so it is not being tracked. To merge changes, `Session.merge()` is required. Hibernate compares the detached entity's state to the current state in the database and cache and updates accordingly.

30. Entity vs. collection cache

- **Entity Cache**: Caches a single entity instance. Uses `@Cacheable` on the entity class.
- **Collection Cache**: Caches the association between an entity and its collection of related entities. Uses `@Cache` on the collection field.

Advanced features and problems

31. Stateless session

- **Scenario**: High-volume, non-interactive batch processing.
- **Solution**: Use `StatelessSession` obtained from the `SessionFactory`. It does not have a first-level cache, perform dirty checks, or trigger lifecycle callbacks, making it very performant for bulk operations.

32. Custom interceptor

- **Scenario**: Automatically set an `auditorId` field on every entity before persistence.
- **Solution**: Implement `org.hibernate.Interceptor`. Override `onSave()` or `onFlushDirty()` to inject the auditor ID. Register the interceptor when building the `SessionFactory`.

33. Multi-tenancy

- **Scenario**: Sharing a single application instance for multiple tenants with separate schemas.
- **Solution**: Use the `SCHEMA` strategy. Implement a `CurrentTenantIdentifierResolver` to

35. Hibernate search

- **Scenario**: Full-text search capabilities on a `Product` entity.
- **Solution**: Integrate Hibernate Search with Lucene or Elasticsearch. Annotate the `Product` entity with `@Indexed` and the fields to be indexed with `@Field`. This automatically manages the search index.

36. Dirty checking

- **How it works**: Hibernate creates a snapshot of the entity's state when it is loaded into the session. At flush time, it compares the current state to the snapshot to detect changes.
- **Detached Entity**: An entity that is not managed by the current session.
- **Reattaching**: Use `Session.merge()` to copy the state of the detached entity onto a new or existing managed entity.

37. `LazyInitializationException`

- **Scenario**: Web controller returning a lazy-loaded collection for JSON serialization.
- **Solution**:
  1. Map the entity to a DTO within the transactional service method.
  2. Use a `JOIN FETCH` or `@EntityGraph` to eagerly fetch the required associations.
  3. Avoid `OSIV` in production and load data explicitly.

38. Entity listener

- **Scenario**: Handling entity lifecycle events.
- **Solution**: Implement a listener class and annotate methods with `@PrePersist`, `@PreUpdate`, etc. Register the listener using `@EntityListeners`.
- **Interceptor vs. Listener**: Listeners are more specific to entity events, while interceptors are for more general session-level interception.

39. Schema validation (`hbm2ddl.auto`)

- **Scenario**: Ensuring entity mappings match the database schema.
- **Solution**: Set `spring.jpa.hibernate.ddl-auto=validate`.
- **Risk in Production**: Using `create`, `create-drop`, or `update` can lead to unintended data loss or schema changes.

40. Bulk operations (`JPQL`)

- **Scenario**: Update a large number of entities.
- **Solution**: Use a bulk update query (`UPDATE` statement in `JPQL` or native SQL). These queries bypass the persistence context, making them very efficient for mass updates.

Explain Deep Dive with Long answers what all types of security we can implement in micro services ,...

Implementation in Spring Boot

The standard approach involves setting up a central **OAuth 2.0 Authorization Server** and configuring your microservices as **OAuth 2.0 Resource Servers** that validate the tokens issued by the Authorization Server.

1. **Authorization Server**: A separate Spring Boot application is configured to issue JWTs upon successful user authentication. Spring offers `spring-boot-starter-oauth2-authorization-server` to simplify this.
2. **Resource Servers (Microservices)**: Each microservice adds the `spring-boot-starter-oauth2-resource-server` dependency. Spring Security automatically validates the incoming JWT based on the configured issuer and public key.
3. **Authentication Flow**:
    1. A client (e.g., a web application) authenticates with the Authorization Server using credentials.
    2. The Authorization Server returns a JWT to the client.
    3. The client sends this JWT in the `Authorization` header of subsequent requests to any of the microservices.
    4. The microservice's Resource Server validates the token's signature and expiration and processes the request.
4. **Role-Based Access Control (RBAC)**: Claims within the JWT can contain user roles and permissions. The Resource Server can use these claims with `@PreAuthorize` or standard URL-based security to enforce fine-grained access control.

Example (`application.yml` for a Resource Server)

yaml

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://your-auth-server.com
```

Use code with caution.

2. Mutual TLS (mTLS)

mTLS is a robust security protocol for encrypting and authenticating communication between services (east-west traffic). Unlike standard TLS, where only the client verifies the server, mTLS requires both parties to present and validate certificates, establishing trust in both directions.

Implementation in Spring Boot

1. **Certificate Generation**: Use tools like OpenSSL or `keytool` to create a Certificate Authority (CA), a server certificate, and client certificates. The server needs a keystore containing its key and certificate, and a truststore containing the CA's certificate to trust clients. Each client needs a keystore containing its key and certificate, plus the CA's certificate in its truststore.
2. **Server-side Configuration**: In `application.yml` or `application.properties`, configure the server to require client authentication.

yaml

Use code with caution.

3. **Client-side Configuration**: For clients making requests to other microservices, you must configure the `RestTemplate` or `WebClient` to use the client's certificate. This can be done by creating a custom SSL context and an `HttpClient` that loads the client's keystore.

3. Securing external-to-internal API calls

An API Gateway is the central entry point for all external client requests. It acts as the first line of defense, providing a single point for authentication, authorization, rate limiting, and other security policies.

How to handle it

1. **Centralized Authentication**: The API Gateway intercepts all inbound requests and performs the initial authentication, typically using OAuth 2.0 tokens. It then validates the JWT and can propagate the user's identity (e.g., via a header) to the downstream internal services.
2. **Request Routing**: After authentication, the gateway routes the request to the appropriate internal microservice. It can enforce access control based on the token's claims, preventing unauthorized access to specific microservice endpoints.
3. **API Key Management**: For non-user-based external services, the API Gateway can enforce API keys. This simplifies security for third-party integrations and provides a centralized way to manage and revoke access.
4. **Token Exchange**: The API Gateway can convert external opaque tokens into internal JWTs, reducing the risk of token exposure and simplifying the internal security model.

4. Handling DDoS attacks

Distributed Denial-of-Service (DDoS) attacks attempt to overwhelm a service with excessive traffic, making it unavailable to legitimate users. Protection involves a multi-layered strategy using external services and application-level controls.

How to handle it

1. **External DDoS Mitigation Services**: A cloud-based service like Cloudflare or AWS Shield is the first line of defense. These services filter malicious traffic before it reaches your infrastructure by inspecting traffic patterns and using a global network to absorb volumetric attacks.
2. **API Gateway Rate Limiting**: Implement rate limiting on your API Gateway to prevent a single client or IP address from making an overwhelming number of requests within a short time. Spring Cloud Gateway offers built-in rate-limiting filters.
3. **Auto-Scaling**: Deploy your microservices in a cloud environment with autoscaling enabled. During a traffic surge, new instances of your microservices are automatically launched to handle the increased load, potentially absorbing the attack without a service outage.
4. **Web Application Firewall (WAF)**: A WAF inspects HTTP traffic and can filter requests that exhibit patterns of common attacks, such as SQL injection or cross-site scripting (XSS), which are often precursors to or part of a DDoS attack.
5. **Service Mesh**: In a Kubernetes environment, a service mesh like Istio or Linkerd can provide additional east-west DDoS protection by enforcing rate limits and access controls between services.

5. Other security best practices

- **Principle of Least Privilege**: Ensure that each microservice has only the minimum necessary permissions to perform its designated function. This limits the potential damage if a service is compromised.
- **CI/CD Pipeline Security**: Integrate security scanning tools into your CI/CD pipeline. These tools can automatically scan for vulnerabilities in code and dependencies during development.

To implement security in springBoot app based on MSA architecture What all steps and dependency...

For microservices security in a Spring Boot application, using JWT with a refresh token mechanism is a standard approach that balances statelessness, security, and user experience. This implementation can exist on its own or, for more complex scenarios, within an OAuth 2.0 framework.

JWT with refresh token implementation

This section details the steps, code, and dependencies needed to implement JWT authentication with a refresh token in a microservices environment. For this example, we assume one microservice acts as an authentication server to issue tokens, and other microservices act as resource servers to validate them.

Dependencies

Common (Authentication Server & Resource Server):

- `spring-boot-starter-web`: Provides web capabilities for REST endpoints.
- `spring-boot-starter-security`: The core of Spring Security for authentication and authorization.
- `jjwt-api`, `jjwt-impl`, and `jjwt-jackson`: For creating and validating JSON Web Tokens.

Authentication Server only:

- `spring-boot-starter-data-jpa` or a database starter (e.g., `mysql-connector-j`): To store user and refresh token data.

Project structure (Authentication Server)

```
└── src
    └── main
        └── java
            └── com.example.auth
                ├── config
                │   └── SecurityConfig.java
                ├── controller
                │   └── AuthController.java
                ├── entity
                │   ├── User.java
                │   └── RefreshToken.java
                ├── filter
                │   └── JwtAuthFilter.java
                ├── repository
                │   ├── UserRepository.java
                │   └── RefreshTokenRepository.java
                ├── service
```

library.

2. Entity classes (`User` & `RefreshToken`)

- `User.java` : Represents the user, storing credentials and roles.
- `RefreshToken.java` : Stores a unique refresh token, its expiration date, and links it to a user. It is persisted in the database.

3. Repository interfaces

- `UserRepository.java` : A JPA repository to find users by username.
- `RefreshTokenRepository.java` : A JPA repository to manage refresh tokens, including methods to find and verify tokens.

4. `JwtService` utility class
This class handles all JWT-related operations.

- `generateToken(String username)` : Creates and signs a new, short-lived JWT (e.g., 15 minutes).
- `validateToken(String token, UserDetails userDetails)` : Verifies the token's signature and expiration.
- `extractUsername(String token)` : Parses the token to get the username.

5. `RefreshTokenService`
This service manages refresh tokens.

- `createRefreshToken(String username)` : Creates and saves a new refresh token in the database with a long expiration time (e.g., 30 days).
- `findByToken(String token)` : Retrieves a refresh token from the database.
- `verifyExpiration(RefreshToken token)` : Checks if the refresh token has expired and deletes it if so.

6. `UserDetailsServiceImpl`
This service is a standard Spring Security component that loads user data for the authentication provider.

- It fetches user data from the database using the `UserRepository` .

7. `AuthController`
This REST controller exposes authentication endpoints.

- `POST /auth/login` : Authenticates a user. On success, it calls `JwtService` to generate an access token and `RefreshTokenService` to generate a refresh token, returning both to the client.
- `POST /auth/refreshtoken` : Accepts a refresh token. If valid, it returns a *new* access token.

8. `SecurityConfig`
This class configures the security filter chain.

- It disables CSRF protection and configures a `STATELESS` session management policy.
- It permits access to authentication endpoints (`/auth/**`).
- It adds `JwtAuthFilter` to the filter chain to intercept and validate JWTs on protected routes.

9. `JwtAuthFilter`
This custom filter intercepts requests to protected resources.

- It extracts the JWT from the `Authorization` header.
- It uses `JwtService` to validate the token.

and the JJWT dependencies.

2. `SecurityConfig`: Configure the filter chain to be stateless and to require authentication for all endpoints. Add the `JwtAuthFilter` to the chain, similar to the authentication server.

3. `JwtService` & `JwtAuthFilter`: These classes are identical to those on the authentication server, but they only perform validation—they do not generate new tokens.

OAuth 2.0 and its distinction from JWT authentication

JWT is a standard for creating tokens, while OAuth 2.0 is a framework for delegated authorization. The two are not mutually exclusive; in fact, OAuth 2.0 frequently uses JWT as the format for its access and ID tokens.

Feature	JWT Authentication (Standalone)	OAuth 2.0
Purpose	Authentication, ensuring the user is who they claim to be.	Authorization, granting a third-party application limited access to a user's resources.
Scope	Manages authentication and basic claims within a single application or set of trusted microservices.	Manages authorization across multiple applications and third-party services, often involving user consent.
Complexity	Relatively simple to implement for a small number of services.	More complex, involving multiple components like an Authorization Server, Resource Server, and Clients.
Revocation	Difficult. Requires implementing a token blacklist or relying on short-lived tokens and refresh tokens.	Built-in support for immediate token revocation via the Authorization Server.
Use Case	Internal microservices communication where trust is pre-established. Scalable and efficient due to statelessness.	When users need to grant external applications access to their data (e.g., "Login with Google").

How JWT fits into OAuth 2.0

In an OAuth 2.0 flow, an Authorization Server issues an access token. For a JWT-based OAuth 2.0 implementation, this access token is a JWT.

1. **Authorization Server**: Authenticates the user and issues a JWT as the access token, containing claims about the user and the granted scopes.

2. **Client Application**: Receives the JWT and sends it with each request to a resource server.

3. **Resource Server**: Validates the JWT locally without contacting the Authorization Server, using the public key provided by the Authorization Server.

This combination leverages OAuth's robust, standardized framework for managing authorization and consent, with JWT's performance benefits for stateless, scalable APIs.

Please provide step by step guide and code to implemet OAuth 2.0 in SpringBoot app and also...

For Spring Boot microservices, implementing OAuth 2.0 typically involves three distinct applications: an **Authorization Server** to handle user authentication and issue tokens, a

Use the Spring Initializr to create three separate projects:

1. `auth-server`: The authorization server.
2. `resource-server`: The resource server (your protected microservice).
3. `client-app`: The client application.

Add the necessary dependencies to each project:

- `auth-server`
  - `spring-boot-starter-web`
  - `spring-boot-starter-security`
  - `spring-boot-starter-oauth2-authorization-server`
  - `spring-security-oauth2-jose` (for JWT support)
  - `spring-boot-starter-data-jpa` and a database driver (e.g., H2)
- `resource-server`
  - `spring-boot-starter-web`
  - `spring-boot-starter-security`
  - `spring-boot-starter-oauth2-resource-server`
- `client-app`
  - `spring-boot-starter-web`
  - `spring-boot-starter-security`
  - `spring-boot-starter-oauth2-client`

Step 2: Configure the Authorization Server (`auth-server`)

1. Configure application properties (`auth-server/src/main/resources/application.yml`)
This sets up the database and a simple user for demonstration purposes.

yaml

```yaml
spring:
  datasource:
    url: jdbc:h2:mem:authdb
    driver-class-name: org.h2.Driver
  jpa:
    hibernate:
      ddl-auto: create-drop
  security:
    user:
      name: user
      password: password
```

Use code with caution.

2. Configure the authorization server (`auth-server/src/main/java/.../AuthorizationServerConfig.java`)
This is the central configuration class for the OAuth 2.0 server.

java

```java
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
```

```
import org.springframework.security.oauth2.core.AuthorizationGrantType;
import org.springframework.security.oauth2.core.ClientAuthenticationMethod;
import org.springframework.security.oauth2.core.oidc.OidcScopes;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.server.authorization.client.RegisteredCl
import org.springframework.security.oauth2.server.authorization.client.RegisteredCl
import org.springframework.security.oauth2.server.authorization.config.annotation.w
import org.springframework.security.oauth2.server.authorization.config.annotation.w
import org.springframework.security.oauth2.server.authorization.settings.ClientSett
import org.springframework.security.oauth2.server.authorization.settings.TokenSetti
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.oauth2.jwt.NimbusJwtEncoder;
import org.springframework.security.oauth2.jwt.NimbusJwtDecoder;
import com.nimbusds.jose.jwk.JWKSet;
import com.nimbusds.jose.jwk.source.ImmutableJWKSet;
import com.nimbusds.jose.jwk.source.JWKSource;
import com.nimbusds.jose.proc.SecurityContext;
import com.nimbusds.jose.jwk.RSAKey;

@Configuration
@EnableWebSecurity
public class AuthorizationServerConfig {

    private final PasswordEncoder passwordEncoder;

    public AuthorizationServerConfig(PasswordEncoder passwordEncoder) {
        this.passwordEncoder = passwordEncoder;
    }

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    public SecurityFilterChain authorizationServerSecurityFilterChain(HttpSecurity
        OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);
        http.getConfigurer(OAuth2AuthorizationServerConfigurer.class).oidc(Customiz
        return http.build();
    }

    @Bean
    public RegisteredClientRepository registeredClientRepository() {
        RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID
            .clientId("client")
            .clientSecret(passwordEncoder.encode("secret"))
            .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BA
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
            .redirectUri("http://127.0.0.1:8080/login/oauth2/code/client")
            .scope(OidcScopes.OPENID)
            .scope("api.read")
            .clientSettings(ClientSettings.builder().requireAuthorizationConsent(tr
            .build();
        return new InMemoryRegisteredClientRepository(registeredClient);
    }
}
```

Use code with caution.

3. Create JWK Source (`auth-server/src/main/java/.../Jwks.java`)

This class provides the private and public keys used for signing and verifying the JWT.

java

```
import com.nimbusds.jose.jwk.JWKSet;
import com.nimbusds.jose.jwk.RSAKey;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
```

```
                .privateKey(privateKey)
                .keyID(UUID.randomUUID().toString())
                .build();
    }
    private static KeyPair generateRsaKey() {
        try {
            KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA")
            keyPairGenerator.initialize(2048);
            return keyPairGenerator.generateKeyPair();
        } catch (NoSuchAlgorithmException ex) {
            throw new IllegalStateException(ex);
        }
    }
}
```

Use code with caution.

Step 3: Configure the Resource Server (`resource-server`)

1. Configure application properties (`resource-server/src/main/resources/application.yml`)

Tell the resource server where to find the authorization server's public keys (`jwk-set-uri`).

yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://127.0.0.1:9000
```

Use code with caution.

2. Configure the security filter chain (`resource-server/src/main/java/.../ResourceServerConfig.java`)

This configures Spring Security to validate the incoming JWT from the client.

java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSe
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class ResourceServerConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Except
        http
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt -> jwt.jwkSetUri("http:/
        return http.build();
    }
}
```

```
@RestController
public class ProtectedController {
    @GetMapping("/protected-data")
    public String getProtectedData() {
        return "This is protected data!";
    }
}
```

Use code with caution.

Step 4: Configure the Client App (`client-app`)

1. Configure application properties (`client-app/src/main/resources/application.yml`)
Define the OAuth 2.0 client registration details.

yaml

```
spring:
  security:
    oauth2:
      client:
        registration:
          client:
            client-id: client
            client-secret: secret
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            scope: openid, api.read
          provider:
            client:
              issuer-uri: http://127.0.0.1:9000
```

Use code with caution.

2. Create a controller to trigger the login (`client-app/src/main/java/.../ClientController.java`)
This controller handles the user login flow.

java

```
import org.springframework.security.oauth2.client.authentication.OAuth2Authenticati
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ClientController {

    @GetMapping("/")
    public String home(OAuth2AuthenticationToken auth) {
        if (auth != null) {
            return "Hello, " + auth.getPrincipal().getAttribute("name") + "!";
        }
        return "Hello, anonymous user!";
    }

    @GetMapping("/secure-page")
    public String securePage() {
        return "Welcome to the secure page!";
    }
}
```

		a resource server.
Protocol vs. Token	JWT is a **token format** (a standard for structuring information).	OAuth 2.0 is an **authorization protocol** (a framework for the interaction between parties).
Actors	Client and server.	Resource Owner, Client, Authorization Server, and Resource Server.
Complexity	Simpler to implement for direct authentication scenarios.	More complex, involving multiple components and a multi-step flow.
Token Management	Tokens are self-contained and difficult to revoke before expiration without a custom blacklist.	Provides a formal framework for refresh tokens and token revocation managed by the Authorization Server.
Delegation	Does not inherently support delegated access.	Specifically designed to allow clients to access resources on behalf of a user.
Use Case	Internal, highly trusted services where the client and resource server share trust. Ideal for stateless API authentication.	Third-party integrations (e.g., "Log in with Google"), federated identity management, and fine-grained access control with scopes.
Revocation	Requires a custom implementation (like a blacklist) to revoke an access token prematurely.	Offers built-in support for immediate token revocation via the Authorization Server, enhancing security.

What is the same/similar?

- **Security enhancement:** Both are designed to enhance the security of web applications by providing secure, token-based mechanisms.
- **Token-based:** Both rely on tokens to prove identity and access rights.
- **Support for RESTful services:** Both are well-suited for securing RESTful APIs, especially in microservices architectures.
- **JSON:** Both technologies are based on JSON. JWT uses it to represent claims, while OAuth uses it for token responses.
- **Can be used together:** In modern applications, JWT is often used *within* the OAuth 2.0 framework. OAuth provides the authorization framework and issues a JWT as the access token. The Resource Server then uses the JWT's self-contained nature for efficient, local validation. This hybrid approach combines the strengths of both.