Authentication and Session Management

How can we restrict single user if he try to login to apllication from diffrent browser.
Demerit of token
how many types of authrntication can we use and each of their use and which is better. (like oauth, jwt, etc)
design database from scratch using entity class
write query to fetch only those user data where department have some user.
write a program to sort username based on length using streams api.
synchronous and asynchronous mode of communication.

I am java spring and sql developer. explain in detail respect to it.

Here's a detailed breakdown of each question in relation to Java, Spring Boot, and SQL, tailored for your expertise:

---

1. Restrict a Single User from Logging in via Different Browsers

Implementation:

- **Solution**: Use **Spring Security** with session control.
- **Approach**:
  - Maintain a session registry (e.g., `SessionRegistryImpl` in Spring Security).
  - Invalidate existing sessions when a user logs in from a new browser.

Code Example:

```java
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private SessionRegistry sessionRegistry;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .anyRequest().authenticated()
                .and()
            .sessionManagement()
                .maximumSessions(1)
                .sessionRegistry(sessionRegistry);
    }

    @Bean
    public SessionRegistry sessionRegistry() {
        return new SessionRegistryImpl();
        }
    }
```

Drawbacks:

- Relies heavily on cookies and session tracking, which might not work seamlessly in stateless architectures like microservices.
- Requires additional storage for session information in distributed systems.

---

2. Demerits of Token-Based Authentication

Issues:

1. **Stateless Nature**:
    - Tokens like JWT are stateless, so you cannot revoke them easily after issuance without maintaining a blacklist.
2. **Token Size**:
    - JWTs are larger than session IDs due to embedded claims, impacting performance in headers.
3. **Security Risks**:
    - Stolen tokens can grant unauthorized access until expiration.
    - No out-of-the-box support for fine-grained invalidation.

---

3. Types of Authentication and Their Use Cases

(a) OAuth 2.0:

- Use: Delegated access to resources (e.g., login via Google, Facebook).
- Better for: Third-party integrations.
- Security: Uses access tokens, refresh tokens.

(b) JWT (JSON Web Token):

- Use: Stateless, distributed authentication in microservices.
- Better for: APIs needing scalability and stateless operations.

(c) Session-Based Authentication:

- Use: Tracks session on the server using cookies.
- Better for: Web apps with single-server backends.

(d) Basic Authentication:

- Use: Simple username-password over HTTPS.
- Better for: Internal tools with minimal traffic.

(e) SAML:

- Use: Single sign-on in enterprise systems.
- Better for: Enterprise identity federation.

Recommendation:

- **Microservices**: Use JWT.
- **Web Apps**: Use session-based or OAuth 2.0.

---

4. Design Database Using Entity Classes

Example: User and Department

```java
@Entity
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    private List<User> users;
}

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String username;

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

---

5. Query to Fetch Only Departments with Users

SQL Query:

```sql
SELECT d.*
FROM department d
WHERE EXISTS (
    SELECT 1
    FROM user u
    WHERE u.department_id = d.id
);
```

Using JPQL:

```java
@Query("SELECT d FROM Department d WHERE d.users IS NOT EMPTY")
List<Department> findDepartmentsWithUsers();
```

---

6. Sort Usernames by Length Using Streams API

```java
List<String> usernames = Arrays.asList("Alice", "Bob", "Christopher");

List<String> sorted = usernames.stream()
    .sorted(Comparator.comparingInt(String::length))
    .collect(Collectors.toList());

sorted.forEach(System.out::println);
```

Output:

```
Bob
Alice
Christopher
```

---

7. Synchronous and Asynchronous Communication

Synchronous:

- Definition: Real-time request-response model.
- Example in Java: REST APIs using `` `RestTemplate` ``.

  ```java
  RestTemplate restTemplate = new RestTemplate();
  String response = restTemplate.getForObject("http://example.com/api", String.class);
  ```

- Use Case: When immediate response is needed (e.g., user login).

Asynchronous:

- Definition: Non-blocking, response arrives later.
- Example in Java: Message Queues like RabbitMQ with `` `@Async` ``.

  ```java
  @Async
  public CompletableFuture<String> processRequest() {
  ```

```
        return CompletableFuture.completedFuture("Processed");
    }
```

- Use Case: Background tasks, microservice communication.

Would you like additional examples or deeper explanations on any topic?

Yeah explain everthing about question 3 in detail. how to implement each one. and also in jwt explain about structutre of jwt token and expalin what is stateful and stateless and diffrence and why use which one.

Question 3: Types of Authentication in Detail

(a) OAuth 2.0

What is it? OAuth 2.0 is an open standard for access delegation, often used for third-party application access. It allows secure resource sharing between applications without sharing user credentials.

Implementation in Spring Boot:

1. Add dependencies for Spring Security OAuth2:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

2. Configure application properties:

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: YOUR_CLIENT_ID
            client-secret: YOUR_CLIENT_SECRET
            scope: profile, email
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            authorization-grant-type: authorization_code
```

3. Set up security:

```java
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .anyRequest().authenticated()
                .and()
            .oauth2Login();
    }
}
```

4. When the user logs in via Google, a token is issued to access Google services.

Use Case: Delegated authentication like login via Google, Facebook, etc.

(b) JWT (JSON Web Token)

What is it? JWT is a compact, URL-safe token format used for stateless authentication. It consists of three parts:

- **Header**: Metadata about the token (e.g., type and algorithm).
- **Payload**: Claims, such as user data and expiration.
- **Signature**: Ensures the token hasn’t been tampered with.

Structure of a JWT:

```css
HEADER.PAYLOAD.SIGNATURE
```

Example:

```plaintext
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJKb2huIERvZSIsImlhdCI6MTUxNjIzOTAyMiwiZXhwIjo
xNTE2MjM5MDk4fQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

1. **Header**:

   ```json
   {
       "alg": "HS256",
       "typ": "JWT"
   }
   ```

   Encoded to: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`

2. **Payload**:

```json
{
    "sub": "John Doe",
    "iat": 1516239022,
    "exp": 1516239098
}
```

Encoded to: `eyJzdWIiOiJKb2huIERvZSIsImlhdCI6MTUxNjIzOTAyMiwiZXhwIjoxNTE2MjM5MDk4fQ`

3. **Signature**:
   Hash of the header and payload with a secret key.

Implementation in Spring Boot:

1. Add dependencies:

```xml
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
```

2. Generate JWT Token:

```java
public String generateToken(String username) {
    return Jwts.builder()
        .setSubject(username)
        .setIssuedAt(new Date())
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60)) // 1 hour
        .signWith(SignatureAlgorithm.HS256, "my-secret-key")
        .compact();
}
```

3. Validate JWT Token:

```java
public Claims validateToken(String token) {
    return Jwts.parser()
        .setSigningKey("my-secret-key")
        .parseClaimsJws(token)
        .getBody();
}
```

Use Case: Microservices requiring stateless communication.

(c) Session-Based Authentication

What is it? Session-based authentication uses server-side storage to maintain user login state. A session ID is stored in a cookie.

Implementation in Spring Boot:

1. Configure session management:

```
java


  @Override
  protected void configure(HttpSecurity http) throws Exception {
      http
          .authorizeRequests()
              .anyRequest().authenticated()
              .and()
          .formLogin()
              .and()
          .sessionManagement()
              .maximumSessions(1);
  }
```

2. Session data is maintained on the server.

Use Case: Traditional monolithic web applications.

(d) Basic Authentication

What is it? Sends a username and password encoded in Base64 with each request.

Implementation in Spring Boot:

1. Enable Basic Auth:

```
java


  @Override
  protected void configure(HttpSecurity http) throws Exception {
      http
          .authorizeRequests()
              .anyRequest().authenticated()
              .and()
          .httpBasic();
  }
```

Use Case: Simple internal APIs.

(e) SAML (Security Assertion Markup Language)

What is it? An XML-based standard for exchanging authentication and authorization data.

Implementation in Spring Boot:

- Use libraries like `` `spring-security-saml2-service-provider` ``.

Use Case: Single sign-on in enterprise applications.

Stateful vs Stateless Authentication

Feature	Stateful Authentication	Stateless Authentication
State	Server stores session state.	No state stored on server.
Scalability	Limited scalability (needs centralized DB).	Highly scalable (no server dependency).
Performance	Overhead due to state synchronization.	Better performance with reduced server load.
Security	Sessions can be hijacked.	Tokens must be secured to prevent misuse.

Why Stateless:

- Better for distributed systems and microservices.
- Reduces server dependency.

Why Stateful:

- Easier to implement for small-scale applications.
- Revoking sessions is simple.

Which is Better?

- **Microservices**: Use **JWT** or **OAuth 2.0** for scalability.
- **Web Applications**: Use **Session-based Authentication** for simplicity.

Would you like any further details or additional code examples?