Jwt Authentication and Authorisation

/**
Setting session management to stateless is essential when using JWT (JSON Web Tokens) for authentication in a web application. Here's why:

1. **JWT Token-based Authentication**:
   - JWT is a stateless authentication mechanism where authentication credentials (claims) are encoded into a secure token. Unlike traditional session-based authentication, where session state is maintained on the server, JWT allows the server to verify the authenticity of each request based solely on the information contained within the token.

2. **No Server-side Session Storage**:
   - With stateless JWT authentication, the server does not need to store session state or user sessions in memory or a persistent data store. This eliminates the need for server-side session management, such as session identifiers or session storage mechanisms like in-memory session storage, Redis, or databases.

3. **Scalability and Performance**:
   - Stateless authentication improves scalability and performance by removing the overhead associated with session management. Since each request contains all necessary authentication information within the JWT token itself, there is no need for the server to query a session store or perform session-related operations. This reduces the load on the server and allows it to handle a larger number of concurrent requests efficiently.

4. **Distributed Systems and Microservices**:
   - In distributed systems or microservices architectures, stateless authentication simplifies integration and communication between services. Each service can independently validate JWT tokens without relying on shared session state. This enables a more decoupled and resilient architecture, where services can operate autonomously and scale independently.

5. **Token-based Authorization**:
   - JWT tokens can contain authorization claims (e.g., roles, permissions) alongside authentication claims. By using stateless JWT authentication, authorization decisions can be made solely based on the information contained within the token, without the need for additional server-side lookups or database queries.

6. **Client-side Token Management**:
   - Since JWT tokens are generated and managed client-side, they can be securely stored in client applications (e.g., browser local storage, mobile app storage) and included in subsequent requests as part of the HTTP headers. This simplifies client-side authentication logic and reduces the reliance on

server-side session management.

In summary, setting session management to stateless is necessary for JWT authentication to leverage its benefits of scalability, performance, simplicity, and compatibility with modern distributed architectures and client-side applications. It aligns with the stateless nature of JWT tokens and eliminates the need for server-side session management in favor of token-based authentication.
*/

This allows us to access the page if one has logged in at first instance. He has to manually not enter each time.

Jwt token is made up of 3 components red, purple and blue coloured components

Go to jwt.io

Components -> header (algorithm for encryption and token type), payload (our data), and verify signature (contain how we are generating the token)

Actually the third component is like:

```
HMACSHA256(
      base64UrlEncode(header) + "." +  base64UrlEncode(payload), your-256-bit-secret-key
)
```

Now we need 3 dependencies:
Jjwt-api, jjwt-impl, jjwt-jackson

Make a component JwtService

```
@Component
class JwtService {
      public String generateToken(String username) {
            Map<String, Object> claims = new HashMap<>(); // all the components are called claims in jwt.
            return createToken(claims, username);
      }

      public String createToken(Map<String, Object> claims, String username) {
            return Jwts.builder().setClaims(claims).setSubject(username).setIssuedAt(new Date(System.currentTimeMillis()))
                        .setExpiration(new Date(System.currentTimeMillis() +
```

```
1000*60*30)) // 30 mins
                        .signKey(generateSignKey(),
SignatureAlgorithm.HS256).compact();
        }
        //we can see we want 256 bit decoded form of secret key
        public Key generateSignKey() {
                byte[] keyBytes = Decoders.BASE64.decode(SECRET);
                return Keys.hmacShaKeyFor(keyBytes);
        }
}
```

Now we can use this service in controller.

```
/**
```
The `generateSignKey` method in the `JwtService` class is used to generate a signing key for creating the digital signature of the JWT token. In JWT (JSON Web Token) authentication, digital signatures are used to ensure the integrity and authenticity of the token.

Here's how the `generateSignKey` method works:

1. **Decode Secret Key**:
   The method decodes a Base64-encoded secret key stored in the `SECRET` constant. The secret key is typically a random string of bytes that serves as the shared secret between the server (which issues the tokens) and the clients (which use the tokens for authentication).

   ```java
   byte[] keyBytes = Decoders.BASE64.decode(SECRET);
   ```

   This line decodes the Base64-encoded secret key (`SECRET`) into a byte array.

2. **Create Signing Key**:
   Once the secret key is decoded, it is used to create a signing key using the `Keys.hmacShaKeyFor` method. This method creates a cryptographic HMAC (Hash-based Message Authentication Code) key for use with the specified algorithm (`SignatureAlgorithm.HS256`).

   ```java
   return Keys.hmacShaKeyFor(keyBytes);
   ```

   The method returns the signing key, which is then used by the `Jwts.builder()` to sign the JWT token.

Overall, the `generateSignKey` method ensures that the server and clients share the same secret key for generating and verifying the digital signatures of JWT tokens, thereby enhancing the security of the authentication process.
**/

So now first we need to authenticate the user and then give the token for him

```
@PostMapping("/authenticate")
public String authenticateAndGetToken(@RequstBody AuthRequest ar) {
	Authentication authentication = authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(authRequest.getUsername(),
authRequest.getPassword()));
	if(authenticate.isAuthenticated()) {
		return jwtService.generateToken(ar.getUsername());
	}
	else {
		throw new UserNameNotFoundException("User with name " +
ar.getUsername()+ " not found");
	}
}
```

To get AuthenticateManager we have to define the bean for that

```
@Bean
public AuthenticationManager
authenticationManager(AuthenticationConfiguration config) {
	return config.getAuthenticationManager();
}
```

Now we need to create a filter class to intercept the request and check for token.

First we need to add the logic in JwtService class to extract the userName from the token.

```
public String extractUsername(String token) {
	return extractClaim(token, Claims::getSubject);
}

 public Date extractExpiration(String token) {
	return extractClaim(token, Claims::getExpiration);
   }

 public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
	final Claims claims = extractAllClaims(token);
```

```
        return claimsResolver.apply(claims);
    }

    private Claims extractAllClaims(String token) {
        return Jwts
                .parserBuilder()
                .setSigningKey(getSignKey())
                .build()
                .parseClaimsJws(token)
                .getBody();
    }

    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername()) && !
isTokenExpired(token));
    }


The filter which will be called for all the endpoints
@Component
public class JwtAuthFilter extends OncePerRequestFilter {

    @Autowired
    private JwtService jwtService;

    @Autowired
    private UserInfoUserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        String authHeader = request.getHeader("Authorization");
        String token = null;
        String username = null;
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            token = authHeader.substring(7);
            username = jwtService.extractUsername(token);
        }

        if (username != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {
```

```
            UserDetails userDetails =
userDetailsService.loadUserByUsername(username);
            if (jwtService.validateToken(token, userDetails)) {
                UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());
                authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

So we have to mention to use it before the
UserNamePasswordAuthentication.class

```
@Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        return http.csrf().disable()
                .authorizeHttpRequests()
                .requestMatchers("/products/new","/products/
authenticate").permitAll()
                .and()
                .authorizeHttpRequests().requestMatchers("/products/**")
                .authenticated().and()
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS) // we do
not have to put anything in cookies
                .and()
                .authenticationProvider(authenticationProvider())
                .addFilterBefore(authFilter,
UsernamePasswordAuthenticationFilter.class)
                .build();
    }
```

/**
The lines you've provided are responsible for enriching the authentication token (`authToken`) with additional details and setting it in the security context.

1. **`authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));`**:
   - The `WebAuthenticationDetailsSource` is a class provided by Spring Security that creates `WebAuthenticationDetails` objects.

  - `WebAuthenticationDetails` contains details about the current HTTP request, such as the remote address, session ID, and the user agent.
  - The `buildDetails` method creates a new `WebAuthenticationDetails` object based on the provided `HttpServletRequest`.
  - The `authToken.setDetails(...)` line sets these `WebAuthenticationDetails` as details of the authentication token. This enriches the token with additional context about the current HTTP request.

2. **`SecurityContextHolder.getContext().setAuthentication(authToken);`**:
  - `SecurityContextHolder` is a central class in Spring Security that holds the details of the current security context for the thread.
  - `getContext()` retrieves the current security context from the thread-local storage.
  - `setAuthentication(authToken)` sets the authentication object (`authToken`) in the security context. This effectively authenticates the user for the current request.
  - Once the authentication object is set in the security context, Spring Security will recognize the user as authenticated and allow access to secured resources or methods based on the user's roles and permissions.

In summary, these lines ensure that the authentication token is enriched with details about the HTTP request and then set in the security context, effectively authenticating the user for the current request in a Spring Security-enabled application.
*/