Session maintenance

In Spring Boot, different mechanisms are available to achieve session maintenance in web applications. Here are some of the commonly used means:

1. **HttpSession:**
    - Spring Boot supports HttpSession, a standard Java EE interface for session management. You can use the HttpSession object to store and retrieve session attributes.
    - This mechanism is suitable for small to medium-sized applications that don't require complex session management.
2. **Spring Session:**
    - Spring Session extends the HttpSession mechanism, providing additional features and flexibility. It allows you to store session data in different storage backends, such as Redis, MongoDB, JDBC, etc., instead of relying on the default server-side session storage.
    - Spring Session also offers advanced features like session clustering and distributed session management.
    - To use Spring Session in your Spring Boot application, include the necessary dependencies and configure the session storage backend. You can refer to the official Spring Session documentation for detailed instructions.
3. **JWT (JSON Web Tokens):**
    - JWT is a stateless session management mechanism that doesn't require server-side storage. In this approach, the server generates a token containing user information and signs it with a secret key.
    - The token is then sent to the client and included in subsequent requests. The server can validate and extract user information from the token without storing any session state.
    - To implement JWT-based session management in Spring Boot, you can use libraries like Spring Security and JSON Web Token (JWT) support.
    - Spring Security provides mechanisms to generate and validate JWT tokens, and you can configure it to handle authentication and authorisation based on the tokens. You can find more information about JWT support in Spring Security in the official documentation.

4. **Custom Session Management:**
    - Spring Boot allows you to implement custom session management mechanisms tailored to your requirements.
    - You can create session management logic by implementing the HttpSession interface or extending Spring's session-related classes.
    - Custom session management can be helpful when you need fine-grained control over session handling, integration with external systems, or specific session storage solutions.
    - It requires more advanced knowledge of Spring Boot and the underlying technologies involved.

These are some of the mechanisms you can use to achieve session maintenance in Spring Boot applications. The choice of the mechanism depends on factors such as application size, scalability needs, security requirements, and integration with other components. Consider your specific use case and requirements to determine the most suitable session management approach.

Session maintenance with the help of JSON Web Tokens (JWT) in Spring Boot:

Introduction to Session Maintenance with JWT

Session maintenance is crucial for web applications to manage user authentication and authorisation. JSON Web Tokens (JWT) provide a stateless and secure method for session management. JWTs are self-contained tokens that store user information and can be used to authenticate and authorise subsequent requests.

In a Spring Boot application, session maintenance using JWT involves the following steps:

1. **Generating a JWT:** Upon successful authentication, a JWT is generated and returned to the client. The JWT contains user information and a digital signature.

2. **Validating a JWT:** The server validates the JWT's signature for each incoming request and checks its expiration. If the JWT is valid, the server allows access to the requested resource.

3. **Extracting User Information:** After validating the JWT, the server extracts the user information stored within the token, such as the username or user roles. This information is used to authorise the user and provide personalised responses.

Now, let's explore these steps in more detail.

Generating a JWT

To generate a JWT in Spring Boot, you must perform the following tasks:

1. Configure the JWT secret and expiration time in your application's properties file.

```
# application.properties

jwt.secret=your-secret-key
jwt.expiration=3600000
```

2. Create an authentication controller that handles the authentication request and generates the JWT upon successful authentication. Use the Jwts class to build the JWT and sign it with the secret key.

```
@RestController
public class AuthenticationController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    JwtAuthenticationHelper jwtHelper;

    @PostMapping("/api/authenticate")
    public ResponseEntity<JwtResponse> authenticate(@RequestBody
AuthenticationRequest request) {
        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(request.getUsername(),
request.getPassword())
        );
        // Create UserDetails instance based on your user management approach
        UserDetails userDetails =
userDetailsService.loadUserByUsername(request.getUsername());

        // Generate JWT
        String token = jwtHelper.generateToken(userDetails);

        return JwtResponse.builder().jwtToken(token).build();
    }

    // Other methods and classes
}
```

Validating a JWT

To validate a JWT in Spring Boot, you'll need to perform the following tasks:

1. Create a filter or interceptor that intercepts incoming requests and validates the JWT. You can implement this by extending the OncePerRequestFilter class and overriding the doFilterInternal method.

Method 1

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {
        String token = extractTokenFromRequest(request);

        if (token != null && validateToken(token)) {
            Authentication auth = getAuthenticationFromToken(token);
            SecurityContextHolder.getContext().setAuthentication(auth);
        }

        filterChain.doFilter(request, response);
    }

    private String extractTokenFromRequest(HttpServletRequest request) {
        // Extract token from request header, query parameter, or cookie
        // Return null if no token is foundĀ
    }

    private boolean validateToken(String token) {
        try {

Jwts.parserBuilder().setSigningKey(jwtSecret).build().parseClaimsJws(token);
            return true;
        } catch (JwtException e) {
            return false;
        }
    }

    private Authentication getAuthenticationFromToken(String token) {
        Claims claims =
Jwts.parserBuilder().setSigningKey(jwtSecret).build().parseClaimsJws(token).getBo
dy();
        String username = claims.getSubject();

        // Fetch user details from the database or other source
        // Create and return an instance of UsernamePasswordAuthenticationToken
    }
}
```

Method 2

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Override
      protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain)
                  throws ServletException, IOException {


            String requestHeader = request.getHeader("Authorization");

            //Bearer yeybaggiwjsbshdhddhf
            String username =null;
            String token =null;
            if(requestHeader!=null && requestHeader.startsWith("Bearer"))
            {
                  token = requestHeader.substring(7);

                  username= jwtHelper.getUsernameFromToken(token);

                  if(username!=null &&
SecurityContextHolder.getContext().getAuthentication()==null)
                  {
                        UserDetails userDetails =
userDetailsService.loadUserByUsername(username);

                        if(!jwtHelper.isTokenExpired(token))
                        {
                              UsernamePasswordAuthenticationToken
usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken
(token, null,userDetails.getAuthorities());
            usernamePasswordAuthenticationToken.setDetails
           (new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext()
            .setAuthentication(usernamePasswordAuthenticationToken);
                  }

            }
      }
            filterChain.doFilter(request, response);
      }
}
```

2. Configure the JWT filter in your application's security configuration to intercept incoming requests and perform JWT validation.

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

    // Other configuration code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
            .authorizeRequests()
                .antMatchers("/api/authenticate").permitAll() // Allow access to
authentication endpoint
                .anyRequest().authenticated()
            .and()
            .addFilterBefore(new JwtAuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class);
    }

    // Other methods and classes
}
```

Extracting User Information

After validating the JWT, you can extract the user information from the token. This information is typically used for authorisation purposes or to provide personalised responses.

To extract the user information from the JWT, you'll need to perform the following tasks:

1. Extract the claims (payload) from the validated JWT.

2. Retrieve the desired user information from the claims.

```
String username = claims.getSubject();
String role = (String) claims.get("role");
// Other user information
```

3. Use the extracted user information to authorise the user or customise the response.

```
// Perform authorization checks based on user roles or other attributes
if (role.equals("admin")) {
    // Perform admin-specific actions
} else {
    // Perform actions for other user roles
}
```

By following these steps, you can implement session maintenance with JWT in your Spring Boot application, allowing secure authentication and authorisation for your users.

Base 64 Password Encoding

Base64 encoding represents binary or raw data (images, documents, or binary files) using only printable ASCII characters. It works by encoding groups of 3 bytes (24 bits) into four ASCII characters from a set of 64 characters (hence the name Base64).

The standard Base64 character set includes A-Z, a-z, 0-9, and the symbols '+' and '/'. Additionally, the '=' symbol may be used as padding at the end of the encoded text to ensure its length is a multiple of 4.

Here's a high-level overview of how Base64 encoding works:

1. **Grouping Data**: The data is divided into chunks of 3 bytes (24 bits).
2. **Conversion to 4 Characters**: Each 24-bit chunk is split into four 6-bit chunks.
3. **Mapping to Printable Characters**: Each 6-bit chunk is represented by a character from the Base64 character set.
4. **Padding**: If the length of the input data is not divisible by 3, padding '=' characters are added at the end to make the length a multiple of 4.

Let's see how we can implement Base 64 password encoding in Spring Boot:

```
public LoginResponseDTO authenticateUser(LoginRequestDTO requestDTO) {
        Authentication authToken = new UsernamePasswordAuthenticationToken(
                requestDTO.getUsernameOrEmail(), requestDTO.getPassword());

  Authentication authentication = authenticationManager.authenticate(authToken);

        SecurityContextHolder.getContext().setAuthentication(authentication);

        LoginResponseDTO responseDTO = new LoginResponseDTO();
        responseDTO.setEmail(authentication.getName());
        Optional<User> userOptional = userRepository.findByEmail(authentication.getName());
        if (userOptional.isPresent()) {
            User user = userOptional.get();
            responseDTO.setName(user.getName());
            responseDTO.setEmail(user.getEmail());
            responseDTO.setUsername(user.getUsername());

  String usernamePassword = user.getUsername() + ":" + requestDTO.getPassword();
            String bytesEncoded = Base64.getEncoder()
            .encodeToString(usernamePassword.getBytes());
            String token = "Basic " + bytesEncoded.toString();
            responseDTO.setToken(token);
            responseDTO.setId(user.getId());
        }

        return responseDTO;
    }
```