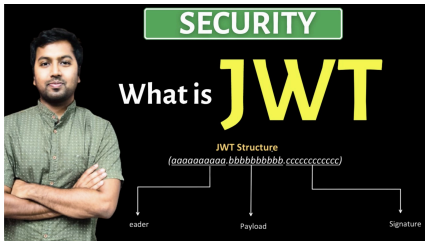
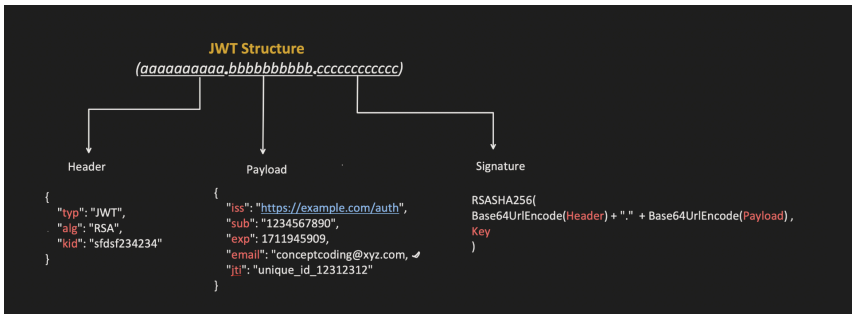
JWT(Json Web Token) Authentication

- **It's a Stateless Authentication method.**
  - **Stateless authentication means, server do not maintain the user authentication state (aka Session).**
- **As mentioned in previous video, JWT has 3 parts**
  - **HEADER.PAYLOAD.SIGNATURE**





- **Header**: Metadata about the token, including the algorithm used (HMAC, RSA etc.)
- **Payload**: Contains **claims** (user details like userId, role, expiry time)
- **Signature**: Ensures token integrity (prevents tampering).
  Any change in payload like role from "user" to "admin" recalculated signature will not match the original.

Sample token:

eyJhbGciOiJSUzI1NiIsInR5cCI6Ik.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9l.SfIKxwRJSMeKKF2QT4fwpMeJ(truncated)
_a …
)

Steps we will follow:

1. User Creation

"/api/user-register" API

We have to create User (with username, password and Role)

2. Token Generation

"/generate-token" API

In this, User will pass its Username and password. If details matched, we will generate JWT token and return the same

3. Token Validation

When User try to access any resource, it will pass back the JWT, we will validate the token, If JWT token is valid, we will Authenticate the user and let the user access the resources.

4. Refresh Token

"/refresh-token" API

We need to provide the functionality to the user to Refresh its JWT token.

JWT Authentication implementation in Spring boot:

1st: User Creation (dynamically)

We have already seen its implementation

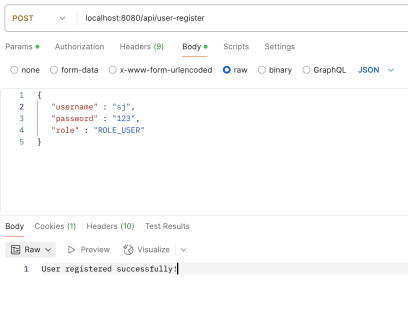
```java
@RestController
@RequestMapping("/api")
public class UserController {

    @Autowired
    UserRegisterEntityService userRegisterEntityService;

    @Autowired
    PasswordEncoder passwordEncoder;

    /*
    using this API to register the user into the system. username, password, role.
     */
    @PostMapping("/user-register")
    public ResponseEntity<String> register(@RequestBody UserRegisterEntity userRegisterDetails) {
        // Hash the password before saving
        userRegisterDetails.setPassword(passwordEncoder.encode(userRegisterDetails.getPassword()));

        userRegisterEntityService.save(userRegisterDetails);

        return ResponseEntity.ok( body: "User registered successfully!");
    }

    @GetMapping("/users")
    public String getUsersDetails(){
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        return "fetched user details successfully";
    }
}
```

```java
@Service
public class UserRegisterEntityService implements UserDetailsService {

    @Autowired
    private UserRegisterEntityRepository userAuthEntityRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return userAuthEntityRepository.findByUsername(username).orElseThrow(() -> new UsernameNotFoundException("user not found"));
    }

    public UserDetails save(UserRegisterEntity userRegisterEntity) {
        return userAuthEntityRepository.save(userRegisterEntity);
    }

}
```

```java
@Repository
public interface UserRegisterEntityRepository extends JpaRepository<UserRegisterEntity, Long> {

    Optional<UserRegisterEntity> findByUsername(String username);
}
```

```java
@Entity
@Table(name = "user_register")
public class UserRegisterEntity implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String password;

    private String role;

    public void setPassword(String password) { this.password = password; }

    public String getRole() { return role; }

    public void setRole(String role) { this.role = role; }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role));
    }

    @Override
    public String getPassword() { return password; }

    @Override
    public String getUsername() { return username; }
}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/user-register").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable());
        return http.build();
    }
}
```



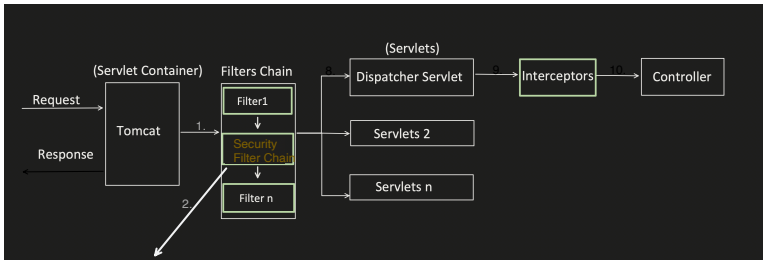
2nd: Token Generation

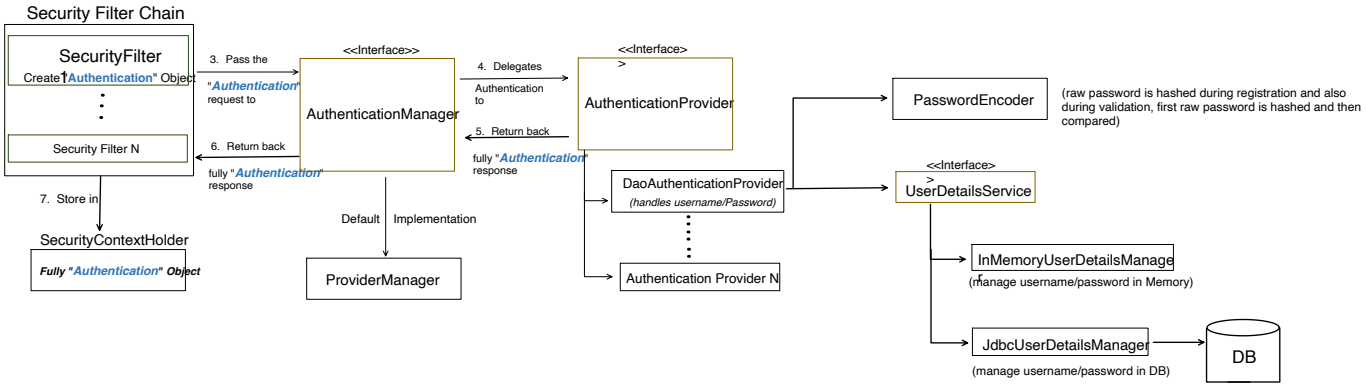
- **Spring boot do not provide any default implementation for JWT Authentication.**
- **Because different application can have different requirement regarding:**
  - **Payload (some need to put only username, some need to put Id etc.)**
  - **Signing Algorithm (some want RSA, some want HMAC etc.)**
  - **Token Refreshing Strategy (some might need it, some don't)**

Now, the onus comes to engineer to implement the JWT functionality, that's why there is no 1 solution for it. Different engineers, different ways to implement.

But we will try to stick to Security Framework only to implement the JWT functionality.

Quick recap of the Architecture:





Notice one behavior in above flow:

- **Security filter creates "*Authentication*" Object, with data coming from request like username/password or Session-ID or Token etc. But it does not know, which Authentication Provider can handle it.**
- **Authentication Manager has a list of Authentication Provider. It calls Support Method of each AuthenticationProvider and pass the "*Authentication*" object and checks, if they can handle the request.**

ProviderManager.java (framework code)

```
@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    Class<? extends Authentication> toTest = authentication.getClass();
    AuthenticationException lastException = null;
    AuthenticationException parentException = null;
    Authentication result = null;
    Authentication parentResult = null;
    int currentPosition = 0;
    int size = this.providers.size();
    for (AuthenticationProvider provider : getProviders()) {
        if (!provider.supports(toTest)) {
            continue;
        }
        if (logger.isTraceEnabled()) {
            logger.trace(LogMessage.format( format: "Authenticating request with %s (%d/%d)",
                    provider.getClass().getSimpleName(), ++currentPosition, size));
        }
        try {
            result = provider.authenticate(authentication);
            if (result != null) {
                copyDetails(authentication, result);
                break;
            }
        }
```

Iterating over the list of Authentication Providers

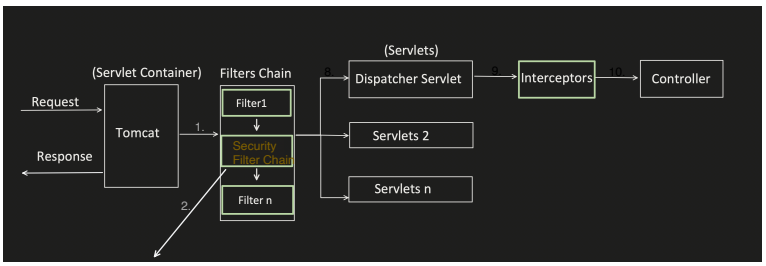
Calls "support" method and checks, if given Authentication Provider can handles the incoming Authentication request.

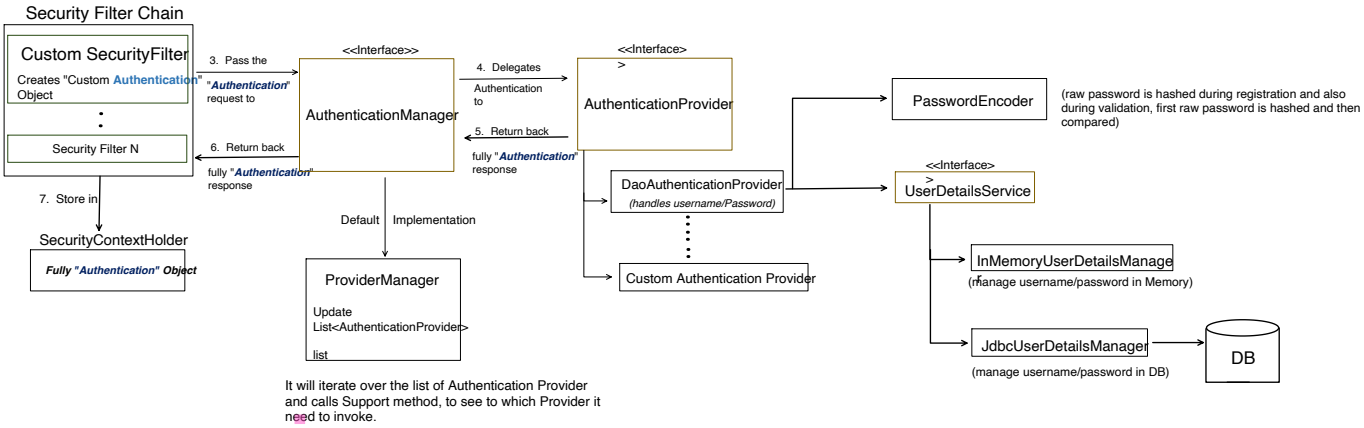
If yes, then only it calls its "authentication" method.

DaoAuthenticationProvider.java (framework code)

```
@Override
public boolean supports(Class<?> authentication) {
    return (UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication));
}
```

We just need to enhance this functionality for JWT implementation:





It will iterate over the list of Authentication Provider and calls Support method, to see to which Provider it need to invoke.

1. **Add our Custom Filter.**
2. **This Custom Filter creates an object of Custom Authentication Object.**
3. **Create new Custom Authentication Provider and also update Authentication Manager's "AuthenticationProvider" List and add our Custom Authentication Provider in it.**
4. **Now Authentication Manager get the custom Authentication Object, it will check all the providers to see, which can handle it and only our CustomAuthencationProvider will return true and thus Authentication Manager will pass the request to our provider.**

Lets, code for Token generation part.

- **First, we need to add JWT dependencies.**
- **In Token Generation part, user passes username/password in request and we have to match it against the stored username and password.**
- **This functionality is similar to DaoAuthenticationProvider, so we will try to use that.**
- **And if username/password is matched, we have to generate the token.**

Pom.xml

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.6</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.6</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId> <!-- Required for JSON processing -->
    <version>0.12.6</version>
</dependency>
```

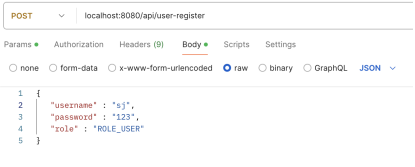
```java
public class JWTAuthenticationFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;
    private final JWTUtil jwtUtil;

    public JWTAuthenticationFilter(AuthenticationManager authenticationManager, JWTUtil jwtUtil) {
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;
    }

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

        if (!request.getServletPath().equals("/generate-token")) {
            filterChain.doFilter(request, response);
            return;
        }

        ObjectMapper objectMapper = new ObjectMapper();
        LoginRequest loginRequest = objectMapper.readValue(request.getInputStream(), LoginRequest.class);

        UsernamePasswordAuthenticationToken authToken =
                new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword());

        Authentication authResult = authenticationManager.authenticate(authToken);

        if (authResult.isAuthenticated()) {
            String token = jwtUtil.generateToken(authResult.getName(), expiryMinutes: 15); //15min
            response.setHeader( s: "Authorization", s1: "Bearer " + token);
        }
    }
}
```

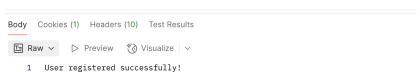
I am specifically using this Authentication object, so that DaoAuthenticationProvider can handle it.

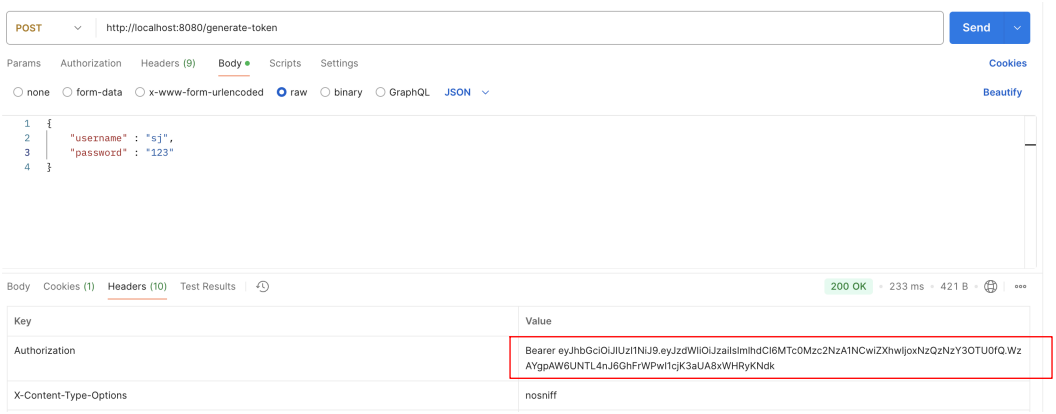
```java
public class LoginRequest {

    private String username;
    private String password;

    public String getUsername() { return username; }

    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }

    public void setPassword(String password) { this.password = password; }
}
```

```java
@Component
public class JWTUtil {

    private static final String SECRET_KEY = "your-secure-secret-key-min-32bytes";
    private static final Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes(StandardCharsets.UTF_8));


    // Generate JWT Token
    public String generateToken(String username, long expiryMinutes) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiryMinutes * 60 * 1000)) //in milliseconds
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }
}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    @Autowired
    public SecurityConfig(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder());
        return provider;
    }

    @Bean
    public PasswordEncoder passwordEncoder(){
       return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http, AuthenticationManager authenticationManager,
                                                   JWTUtil jwtUtil) throws Exception {

        // Authentication filter responsible for login
        JWTAuthenticationFilter jwtAuthFilter = new JWTAuthenticationFilter(authenticationManager, jwtUtil);

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/user-register").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) ; // add generate token filter
        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager() {
        return new ProviderManager(Arrays.asList(
                daoAuthenticationProvider()));
    }
}
```

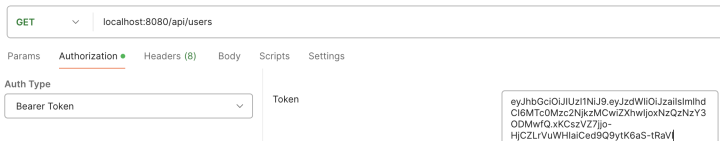






3rd: Token Validation

- **Now, if user try to access any restricted resource. They need to pass the Token in the Authorization Header like below:**




```java
public class JwtValidationFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;

    public JwtValidationFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }
    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        String token = extractJwtFromRequest(request);
        if (token != null) {

            JwtAuthenticationToken authenticationToken = new JwtAuthenticationToken(token);
            Authentication authResult = authenticationManager.authenticate(authenticationToken);
            if(authResult.isAuthenticated()) {
                SecurityContextHolder.getContext().setAuthentication(authResult);
            }
        }

        filterChain.doFilter(request, response);
    }

    private String extractJwtFromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader( s: "Authorization");
        if (bearerToken != null && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring( beginIndex: 7);
        }
        return null;
    }
}
```

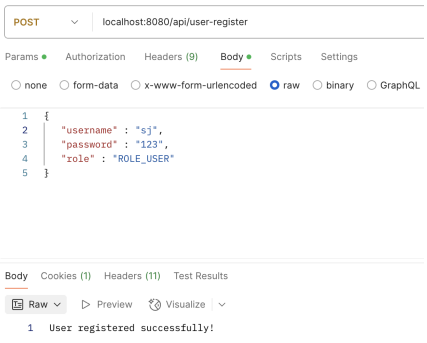
```java
public class JwtAuthenticationToken extends AbstractAuthenticationToken {

    private final String token;

    public JwtAuthenticationToken(String token) {
        super( authorities: null);
        this.token = token;
        setAuthenticated(false);
    }

    public String getToken() { return token; }

    @Override
    public Object getCredentials() { return token; }

    @Override
    public Object getPrincipal() { return null; }
}
```

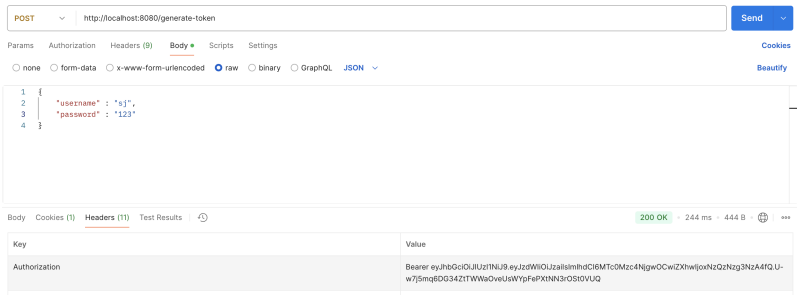
```java
public class JWTAuthenticationProvider implements AuthenticationProvider {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    public JWTAuthenticationProvider(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String token = ((JwtAuthenticationToken) authentication).getToken();

        String username = jwtUtil.validateAndExtractUsername(token);
        if (username == null) {
            throw new BadCredentialsException("Invalid JWT Token");
        }

        UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        return new UsernamePasswordAuthenticationToken(userDetails, credentials: null, userDetails.getAuthorities());
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return JwtAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

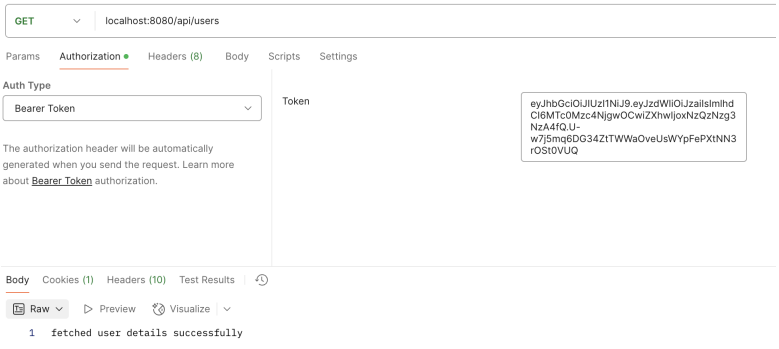
```java
@Component
public class JWTUtil {

    private static final String SECRET_KEY = "your-secure-secret-key-min-32bytes";
    private static final Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes(StandardCharsets.UTF_8));


    // Generate JWT Token
    public String generateToken(String username, long expiryMinutes) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiryMinutes * 60 * 1000)) //in milliseconds
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }

    public String validateAndExtractUsername(String token) {
        try {
            return Jwts.parser() JwtParserBuilder
                    .setSigningKey(key)
                    .build() JwtParser
                    .parseClaimsJws(token) Jws<Claims>
                    .getBody() Claims
                    .getSubject();
        } catch (JwtException e) {
            return null; // Invalid or expired JWT
        }
    }
}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    @Autowired
    public SecurityConfig(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public JWTAuthenticationProvider jwtAuthenticationProvider() {
        return new JWTAuthenticationProvider(jwtUtil, userDetailsService);
    }

    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder());
        return provider;
    }

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
```

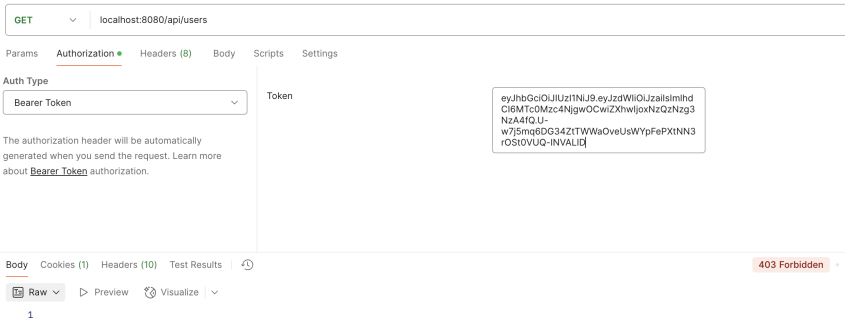
```java
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http, AuthenticationManager authenticationManager,
                                                   JWTUtil jwtUtil) throws Exception {

        // Authentication filter responsible for login
        JWTAuthenticationFilter jwtAuthFilter = new JWTAuthenticationFilter(authenticationManager, jwtUtil);

        // Validation filter for checking JWT in every request
        JwtValidationFilter jwtValidationFilter = new JwtValidationFilter(authenticationManager);

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/user-register").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) // generate token filter
                .addFilterAfter(jwtValidationFilter, JWTAuthenticationFilter.class); // validate token filter
        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager() {
        return new ProviderManager(Arrays.asList(
                daoAuthenticationProvider(),
                jwtAuthenticationProvider()
        ));
    }
}
```







When tried to add Invalid Token:



4th: Refresh Token

- **Generally access tokens are short lived.**
- **Once access token get expired, Refresh token (generally long lived) is used to obtain new access token, without requiring user to log in again.**

```java
public class JWTAuthenticationFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;
    private final JWTUtil jwtUtil;

    public JWTAuthenticationFilter(AuthenticationManager authenticationManager, JWTUtil jwtUtil) {
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;
    }

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

        if (!request.getServletPath().equals("/generate-token")) {
            filterChain.doFilter(request, response);
            return;
        }

        ObjectMapper objectMapper = new ObjectMapper();
        LoginRequest loginRequest = objectMapper.readValue(request.getInputStream(), LoginRequest.class);

        UsernamePasswordAuthenticationToken authToken =
                new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword());
        Authentication authResult = authenticationManager.authenticate(authToken);

        if (authResult.isAuthenticated()) {
            String token = jwtUtil.generateToken(authResult.getName(), expiryMinutes: 15); //15min
            response.setHeader( s: "Authorization", s1: "Bearer " + token);

            String refreshToken = jwtUtil.generateToken(authResult.getName(), expiryMinutes: 7 * 24 * 60); //7day
            // Set Refresh Token in HttpOnly Cookie
            //we can also send it in response body but then client has to store it in local storage or in-memory
            Cookie refreshCookie = new Cookie( name: "refreshToken", refreshToken);
            refreshCookie.setHttpOnly(true); //prevent javascript from accessing it
            refreshCookie.setSecure(true); // sent only over HTTPS
            refreshCookie.setPath("/refresh-token"); // Cookie available only for refresh endpoint
            refreshCookie.setMaxAge(7 * 24 * 60 * 60); // 7 days expiry
            response.addCookie(refreshCookie);
        }
    }
}
```

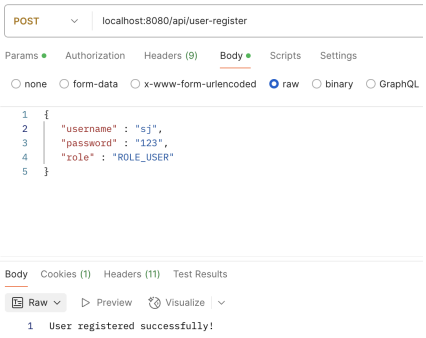
```java
public class JWTRefreshFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;
    private final JWTUtil jwtUtil;

    public JWTRefreshFilter(JWTUtil jwtUtil, AuthenticationManager authenticationManager) {
        this.jwtUtil = jwtUtil;
        this.authenticationManager = authenticationManager;
    }

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

        if (!request.getServletPath().equals("/refresh-token")) {
            filterChain.doFilter(request, response);
            return;
        }
        String refreshToken = extractJwtFromRequest(request);
        if (refreshToken == null) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            return;
        }
        JwtAuthenticationToken authenticationToken = new JwtAuthenticationToken(refreshToken);
        Authentication authResult = authenticationManager.authenticate(authenticationToken);
        if(authResult.isAuthenticated()) {
            String newToken = jwtUtil.generateToken(authResult.getName(), expiryMinutes: 15); //15min
            response.setHeader( s: "Authorization", s1: "Bearer " + newToken);
        }
    }

    private String extractJwtFromRequest(HttpServletRequest request) {
        Cookie[] cookies = request.getCookies();
        if (cookies == null) {
            return null;
        }
        String refreshToken = null;
        for (Cookie cookie : cookies) {
            if ("refreshToken".equals(cookie.getName())) {
                refreshToken = cookie.getValue();
            }
        }
        return refreshToken;
    }
}
```

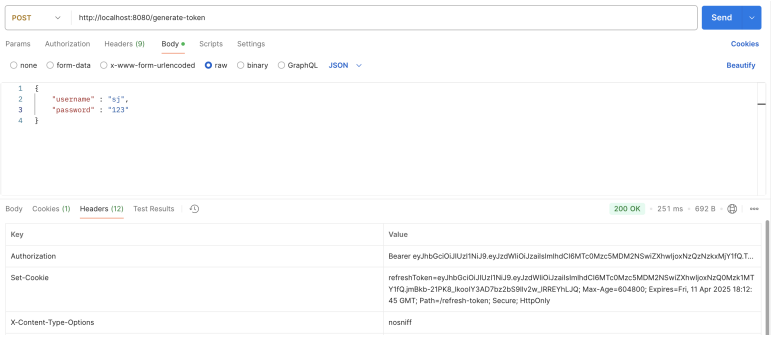
```java
public class JwtAuthenticationToken extends AbstractAuthenticationToken {

    private final String token;

    public JwtAuthenticationToken(String token) {
        super( authorities: null);
        this.token = token;
        setAuthenticated(false);
    }

    public String getToken() { return token; }

    @Override
    public Object getCredentials() { return token; }

    @Override
    public Object getPrincipal() { return null; }
}
```

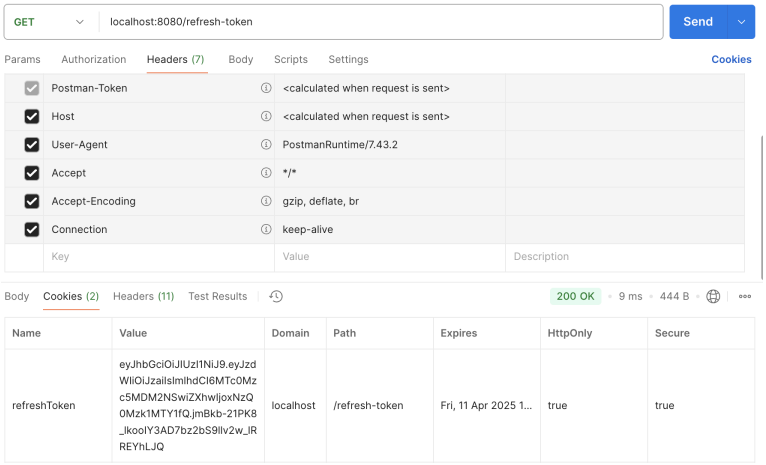
```java
@Component
public class JWTUtil {

    private static final String SECRET_KEY = "your-secure-secret-key-min-32bytes";
    private static final Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes(StandardCharsets.UTF_8));


    // Generate JWT Token
    public String generateToken(String username, long expiryMinutes) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiryMinutes * 60 * 1000)) //in milliseconds
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }

    public String validateAndExtractUsername(String token) {
        try {
            return Jwts.parser() JwtParserBuilder
                    .setSigningKey(key)
                    .build() JwtParser
                    .parseClaimsJws(token) Jws<Claims>
                    .getBody() Claims
                    .getSubject();
        } catch (JwtException e) {
            return null; // Invalid or expired JWT
        }
    }
}
```

```java
public class JWTAuthenticationProvider implements AuthenticationProvider {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    public JWTAuthenticationProvider(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String token = ((JwtAuthenticationToken) authentication).getToken();

        String username = jwtUtil.validateAndExtractUsername(token);
        if (username == null) {
            throw new BadCredentialsException("Invalid JWT Token");
        }

        UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        return new UsernamePasswordAuthenticationToken(userDetails, credentials: null, userDetails.getAuthorities());
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return JwtAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

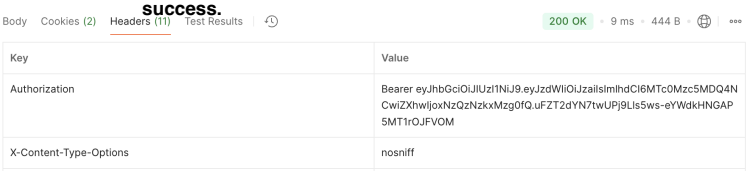
```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    @Autowired
    public SecurityConfig(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public JWTAuthenticationProvider jwtAuthenticationProvider() {
        return new JWTAuthenticationProvider(jwtUtil, userDetailsService);
    }

    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder());
        return provider;
    }

    @Bean
    public PasswordEncoder passwordEncoder(){
       return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http, AuthenticationManager authenticationManager,
                                                   JWTUtil jwtUtil) throws Exception {

        // Authentication filter responsible for login
        JWTAuthenticationFilter jwtAuthFilter = new JWTAuthenticationFilter(authenticationManager, jwtUtil);

        // Validation filter for checking JWT in every request
        JwtValidationFilter jwtValidationFilter = new JwtValidationFilter(authenticationManager);

        // refresh filter for checking JWT in every request
        JWTRefreshFilter jwtRefreshFilter = new JWTRefreshFilter(jwtUtil, authenticationManager);

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/user-register").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) // generate token filter
                .addFilterAfter(jwtValidationFilter, JWTAuthenticationFilter.class) // validate token filter
                .addFilterAfter(jwtRefreshFilter, JwtValidationFilter.class); // refresh token filter
        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager() {
        return new ProviderManager(Arrays.asList(
                daoAuthenticationProvider(),
                jwtAuthenticationProvider()
        ));
    }
}
```







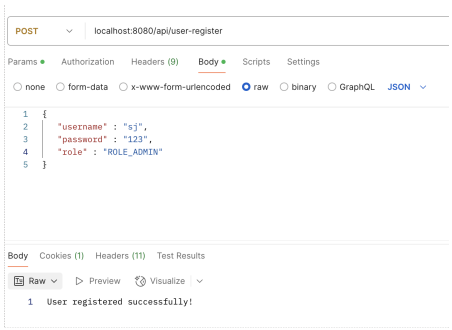
In Header, new access token is set, after api call is success.

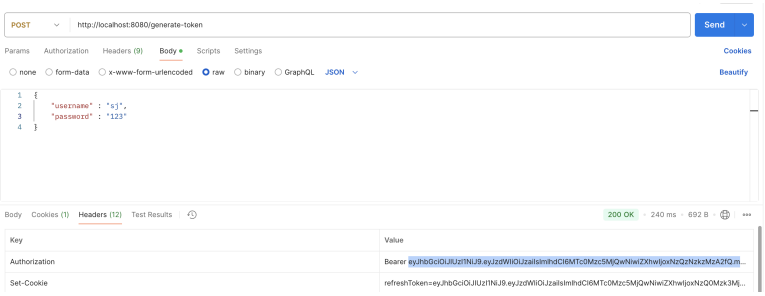


Authorization:

- **Works exactly the same as form and basic Authentication.**
- **AuthorizationFilter gets invokes, which matches the ROLE required for the API and role present for the user.**

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http, AuthenticationManager authenticationManager,
                                               JWTUtil jwtUtil) throws Exception {

    // Authentication filter responsible for login
    JWTAuthenticationFilter jwtAuthFilter = new JWTAuthenticationFilter(authenticationManager, jwtUtil);

    // Validation filter for checking JWT in every request
    JwtValidationFilter jwtValidationFilter = new JwtValidationFilter(authenticationManager);

    // refresh filter for checking JWT in every request
    JWTRefreshFilter jwtRefreshFilter = new JWTRefreshFilter(jwtUtil, authenticationManager);

    http.authorizeHttpRequests(auth -> auth
                .requestMatchers( ...patterns: "/api/user-register").permitAll()
                .requestMatchers( ...patterns: "/api/users").hasRole("USER")
                .anyRequest().authenticated())
            .sessionManagement(session -> session
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(csrf -> csrf.disable())
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) // generate token filter
            .addFilterAfter(jwtValidationFilter, JWTAuthenticationFilter.class) // validate token filter
            .addFilterAfter(jwtRefreshFilter, JwtValidationFilter.class); // refresh token filter
    return http.build();
}
```





As "/api/users" API needs role *ROLE_USER* but user has *ROLE_ADMIN.* So Mismatch happens