How to Implement Custom Authentication in Spring Security

Implementing custom authentication allows you to bypass standard username/password flows and use unique mechanisms like API keys, OTPs, or legacy tokens.

Here is the step-by-step architecture to implement it cleanly:

1. The Core Components

You need to extend/implement three key components:

1. `AuthenticationFilter`: Intercepts the request and extracts the credentials (e.g., a header).
2. `AuthenticationToken`: A DTO that holds the credentials (unauthenticated) and later the user details (authenticated).
3. `AuthenticationProvider`: The logic that validates the credentials.

Step 1: Create a Custom Authentication Token

Extend `AbstractAuthenticationToken`. This object acts as the container for your data.

```
public class ApiKeyAuthenticationToken extends AbstractAuthenticationToken {
    private final String apiKey;

    // Constructor for Pre-Authentication (Unverified)
    public ApiKeyAuthenticationToken(String apiKey) {
        super(null);
        this.apiKey = apiKey;
        setAuthenticated(false);
    }

    // Constructor for Post-Authentication (Verified)
    public ApiKeyAuthenticationToken(String apiKey, Collection<? extends GrantedAuthority
        super(authorities);
        this.apiKey = apiKey;
        setAuthenticated(true);
    }

    @Override
    public Object getCredentials() { return null; } // Hide secret after auth if needed

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

Step 2: Create the Filter

This filter extracts the token from the request and hands it to the `AuthenticationManager`.

```
public class ApiKeyAuthFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;

    public ApiKeyAuthFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse respo
            throws ServletException, IOException {

        // 1. Extract the credential
        String requestApiKey = request.getHeader("X-API-KEY");

        if (requestApiKey == null) {
            filterChain.doFilter(request, response); // Allow other filters to try
            return;
        }

        // 2. Wrap it in your token
        ApiKeyAuthenticationToken token = new ApiKeyAuthenticationToken(requestApiKey);

        try {
            // 3. Delegate to AuthenticationManager
            Authentication authResult = authenticationManager.authenticate(token);

            // 4. Save to SecurityContext
            SecurityContextHolder.getContext().setAuthentication(authResult);
            filterChain.doFilter(request, response);

        } catch (AuthenticationException e) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            response.getWriter().write("Invalid API Key");
        }
    }
}
```

Step 3: Create the Provider (The Logic)

This is where you actually verify the token (e.g., check DB or validate hash).

```
@Component
public class ApiKeyAuthenticationProvider implements AuthenticationProvider {

    @Override
    public Authentication authenticate(Authentication authentication) throws Authenticati
        String apiKey = (String) authentication.getPrincipal();

        // 1. Your Custom Logic (e.g., DB lookup)
        if ("secret-key-123".equals(apiKey)) {
            // 2. Return fully populated token with Authorities
            return new ApiKeyAuthenticationToken(apiKey, AuthorityUtils.createAuthorityLi
        } else {
            throw new BadCredentialsException("Invalid API Key");
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        // Tells the Manager this provider handles ApiKeyAuthenticationToken
        return ApiKeyAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

Step 4: Register Everything in SecurityConfig

Finally, wire them together in your configuration chain.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http, AuthenticationManager authM
        http
            .addFilterBefore(new ApiKeyAuthFilter(authManager), UsernamePasswordAuthentic
            .authorizeHttpRequests(auth -> auth.anyRequest().authenticated());

        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http, ApiKeyAuthentic
        AuthenticationManagerBuilder builder = http.getSharedObject(AuthenticationManager
        builder.authenticationProvider(provider);
        return builder.build();
    }
}
```

Summary of Flow

1. **Filter** sees `X-API-KEY`, creates an `Unauthenticated Token`.
2. **Manager** finds the `Provider` that supports this token class.
3. **Provider** checks the DB, returns an `Authenticated Token` (with Roles).
4. **Filter** puts that token into the `SecurityContext`.
5. **Spring** now treats the user as logged in for the rest of the request.

⁂

1. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/429036f4-3bc3-4a7a-8c6f-90246e73b9b9/image.jpg
2. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/24f18c0a-b56e-4862-a619-e2959036a5c2/image.jpg
3. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/97e9ed86-ed11-4035-a18b-1eb05c1bab4e/image.jpg
4. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/87f62423-96f2-4071-9802-8f6699e0ecd8/image.jpg