Username/Password Authentication

One of the most common ways to authenticate a user is by validating a username and password. Spring Security provides comprehensive support for authenticating with a username and password.

You can configure username and password authentication using the following:

Simple Username/Password Example

Java XML Kotlin

JAVA

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

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authorize) -> authorize
                .anyRequest().authenticated()
            )
            .httpBasic(Customizer.withDefaults())
            .formLogin(Customizer.withDefaults());

        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails userDetails = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();

        return new InMemoryUserDetailsManager(userDetails);
    }

}
```

The preceding configuration automatically registers an in-memory `UserDetailsService` with the `SecurityFilterChain`, registers the `DaoAuthenticationProvider` with the default `AuthenticationManager`, and enables Form Login and HTTP Basic authentication.

To learn more about username/password authentication, consider the following use cases:

- I want to learn how Form Login works
- I want to learn how HTTP Basic authentication works
- I want to learn how `DaoAuthenticationProvider` works
- I want to manage users in memory
- I want to manage users in a database
- I want to manage users in LDAP
- I want to publish an `AuthenticationManager` bean for custom authentication
- I want to customize the global `AuthenticationManager`

Publish an `AuthenticationManager` bean

A fairly common requirement is publishing an `AuthenticationManager` bean to allow for custom authentication, such as in a `@Service` or Spring MVC `@Controller`. For example, you may want to authenticate users via a REST API instead of using Form Login.

You can publish such an `AuthenticationManager` for custom authentication scenarios using the following configuration:

Publish AuthenticationManager bean for Custom Authentication

Java | XML | Kotlin

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

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authorize) -> authorize
                .requestMatchers("/login").permitAll()
                .anyRequest().authenticated()
            );

        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager(
            UserDetailsService userDetailsService,
            PasswordEncoder passwordEncoder) {
        DaoAuthenticationProvider authenticationProvider = new
DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userDetailsService);
        authenticationProvider.setPasswordEncoder(passwordEncoder);

        return new ProviderManager(authenticationProvider);
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails userDetails = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();

        return new InMemoryUserDetailsManager(userDetails);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

}
```

With the preceding configuration in place, you can create a `@RestController` that uses the `AuthenticationManager` as follows:

Create a `@RestController` for Authentication

Java Kotlin

JAVA

```
@RestController
public class LoginController {

    private final AuthenticationManager authenticationManager;
    public LoginController(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    @PostMapping("/login")
    public ResponseEntity<Void> login(@RequestBody LoginRequest loginRequest) {
        Authentication authenticationRequest =
            UsernamePasswordAuthenticationToken.unauthenticated(loginRequest.username(),
loginRequest.password());
        Authentication authenticationResponse =
            this.authenticationManager.authenticate(authenticationRequest);
        // ...
    }

    public record LoginRequest(String username, String password) {
    }

}
```

NOTE

In this example, it is your responsibility to save the authenticated user in the `SecurityContextRepository` if needed. For example, if using the `HttpSession` to persist the `SecurityContext` between requests, you can use `HttpSessionSecurityContextRepository`.

Customize the `AuthenticationManager`

Normally, Spring Security builds an `AuthenticationManager` internally composed of a `DaoAuthenticationProvider` for username/password authentication. In certain cases, it may still be desired to customize the instance of `AuthenticationManager` used by Spring Security. For example, you may need to simply disable credential erasure for cached users.

To do this, you can take advantage of the fact that the `AuthenticationManagerBuilder` used to build Spring Security's global `AuthenticationManager` is published as a bean. You can configure the builder as follows:

Configure global AuthenticationManagerBuilder

Java Kotlin

JAVA

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        // ...
        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        // Return a UserDetailsService that caches users
        // ...
    }

    @Autowired
    public void configure(AuthenticationManagerBuilder builder) {
        builder.eraseCredentials(false);
    }

}
```

Alternatively, you may configure a local `AuthenticationManager` to override the global one.

Configure local AuthenticationManager for Spring Security

Java XML Kotlin

JAVA

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authorize) -> authorize
                .anyRequest().authenticated()
            )
            .httpBasic(Customizer.withDefaults())
            .formLogin(Customizer.withDefaults())
            .authenticationManager(authenticationManager());

        return http.build();
    }

    private AuthenticationManager authenticationManager() {
        DaoAuthenticationProvider authenticationProvider = new
DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userDetailsService());
        authenticationProvider.setPasswordEncoder(passwordEncoder());
        ProviderManager providerManager = new ProviderManager(authenticationProvider);
        providerManager.setEraseCredentialsAfterAuthentication(false);

        return providerManager;
    }

    private UserDetailsService userDetailsService() {
        UserDetails userDetails = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();

        return new InMemoryUserDetailsManager(userDetails);
    }

    private PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

}
```

Section Summary

- Reading Username/Password
- Password Storage