AuthenticationProvider

When implementing Spring Security, the framework automatically provides a default Authentication Provider behind the scenes. However, for database authentication, we need to create our own custom Authentication Provider.

👉 Creating a Custom Authentication Provider

To connect our Spring Security with a database, we need to:

- Create a bean inside SecurityConfig class that returns an AuthenticationProvider
- Use DaoAuthenticationProvider which implements the AuthenticationProvider interface
- Configure this provider to work with our database
- Understanding the Class Hierarchy
  - AuthenticationProvider is an interface
  - DaoAuthenticationProvider is an implementation class for AuthenticationProvider interface
  - DaoAuthenticationProvider extends AbstractUserDetailsAuthenticationProvider
  - AbstractUserDetailsAuthenticationProvider implements AuthenticationProvider

👉 DaoAuthenticationProvider

Creating a DaoAuthenticationProvider object alone is not enough. It needs to know:

- Which database we're working with
- How to represent user data
- What the user table name is

👉 UserDetailsService

To provide this information, we need a UserDetailsService:

- The default UserDetailsService works with static values
- We need a custom implementation to work with database values
- We connect it to our provider using setUserDetailsService()

👉 PasswordEncoder

We also need to specify a password encoder:

- Use setPasswordEncoder() method

- For simple testing, we can use NoOpPasswordEncoder.getInstance()
- In production, you should use a secure encoder like BCrypt

Example:

```java
package com.telusko.springsecdemo.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.NoOpPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public AuthenticationProvider authProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance());
        return provider;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(customizer -> customizer.disable())
            .authorizeHttpRequests(request -> request.anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        return http.build();
    }

    // The commented code below shows the old in-memory authentication approach
    // that we're replacing with database authentication
    /*
     * @Bean public UserDetailsService userDetailsService() {
     *
     * UserDetails user=User
     *     .withDefaultPasswordEncoder()
     *     .username("navin")
     *     .password("n@123")
     *     .roles("USER")
     *     .build();
     *
     * UserDetails admin=User
     *     .withDefaultPasswordEncoder()
     *     .username("admin")
     *     .password("admin@789")
     *     .roles("ADMIN")
     *     .build();
     *
     * return new InMemoryUserDetailsManager(user,admin);
     * }
     */
}
```

- The @Autowired private UserDetailsService userDetailsService; needs to be implemented
- We need to create a custom class that implements UserDetailsService to connect to our database
- The Authentication Provider depends on UserDetailsService to know how to authenticate users
- For production use, always use a secure password encoder (not NoOpPasswordEncoder)

➢ **After setting up this configuration, you'll need to**

- Create a class that implements UserDetailsService
- Define how to fetch user details from your database
- Map database user records to Spring Security's UserDetails objects