Authenticating the login password

Authenticating the login password is crucial in securing user access to an application. In Spring Boot, you can implement password authentication using Spring Security, which provides robust and customisable authentication mechanisms. The following steps outline the process of authenticating the login password:

1. **User Entity and UserRepository:**
   Start by creating a User entity class representing your application's user. The User class typically includes properties such as username, password, and role. Also, create a UserRepository interface extending JpaRepository<User, Long>. This repository interface will provide methods for CRUD (Create, Read, Update, Delete) operations on the User entity.

2. **Password Encryption:**
   Encrypting user passwords before storing them in the database ensures user password security. Spring Security provides a PasswordEncoder interface that you can use to encode passwords. You can choose an implementation, such as BCryptPasswordEncoder, which applies a vital one-way hash function to passwords.

   Create a bean for the password encoder in your application's configuration or security configuration class. For example:

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class SecurityConfig {

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

3. Authentication Service:

Create an authentication service that handles the authentication logic. This service typically implements the UserDetailsService interface provided by Spring Security. Implement the loadUserByUsername method to load user details based on the username provided during login.

In the implementation, retrieve the user details from the UserRepository based on the provided username. If the user does not exist, throw an exception. If the user is found, create an instance of org.springframework.security.core.userdetails.User with the user details and return it.

```java
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    private final UserRepository userRepository;

    public UserDetailsServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("User not found");
        }

        // Create UserDetails instance with user details
        return org.springframework.security.core.userdetails.User
                .withUsername(user.getUsername())
                .password(user.getPassword())
                .roles(user.getRole())
                .build();
    }
}
```

4. Authentication Configuration:

Configure authentication in your application by creating a configuration class that extends WebSecurityConfigurerAdapter. Override the configure method to define authentication settings. Specify the user details service implementation, password encoder, and any additional authentication rules.

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

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;

    public SecurityConfig(UserDetailsService userDetailsService, PasswordEncoder passwordEncoder)
{
        this.userDetailsService = userDetailsService;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder);
    }
}
```

In this example, we inject the UserDetailsService and PasswordEncoder beans into the configuration class and use them to configure the authentication manager.

5. Login Endpoint and Authentication Flow:

Implement a login endpoint in your application to handle the login request. The endpoint should accept the username and password parameters. Spring Security will automatically handle the authentication process based on the configuration during the authentication flow.

For example, you can create a controller method to handle the login request:

```
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
@RequestMapping(“/api/auth”)
public class AuthController {

    @PostMapping("/signin")
    public void login(@RequestBody LoginForm loginForm) {
        // Perform authentication and handle success or failure
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        // ...
    }
}
```

AuthService for the “/*signin*” API

```
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@Service
public class AuthService {
    private AuthenticationManager authenticationManager;
    private PasswordEncoder passwordEncoder;

    public LoginResponseDto AuthenticateUser(LoginRequestDto requestDto) {
        // Create an auth token using username/email and password and authenticate it
using the Authentication manager
       // Generate the JWT token and pass it in the LoginResponseDto
    }
}
```

In this example, the login method receives a LoginForm object containing the username and password. Spring Security will automatically perform authentication based on the provided credentials.

Following these steps, you can use Spring Security to implement password authentication in a Spring Boot application. The login password will be securely authenticated by retrieving user details, encrypting passwords, and leveraging Spring Security's authentication mechanisms.