Spring boot Security - Part2

Friday, 7 March 2025 3:48 PM

User Creation:

Before, we proceed with User Authentication and Authorization methods, we first need to see, User creation process because that's the first step.

Authentication and Authorization of User will happen only after User is created.

Lets see, what will happen when we add below security dependency, as seen in previous ***Architecture*** video and starts the server:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
```

Provides core feature like:
- Authentication
- Authorization
- Security filters etc.

Logs when server is started:

```
2025-03-08T15:09:16.356+05:30  INFO 44103 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2025-03-08T15:09:16.466+05:30  WARN 44103 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database
2025-03-08T15:09:16.500+05:30  WARN 44103 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: b5341001-1e0d-4bad-9440-0f8b1c51cfa9

This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

```
 main] r$InitializeUserDetailsManagerConfigurer : Global AuthenticationManager configured with UserDetailsService bean with name inMemoryUserDetailsManager
 main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
 main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 1.751 seconds (process running for 1.889)
exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]      : Initializing Spring DispatcherServlet 'dispatcherServlet'
exec-1] o.s.web.servlet.DispatcherServlet       : Initializing Servlet 'dispatcherServlet'
exec-1] o.s.web.servlet.DispatcherServlet       : Completed initialization in 1 ms
```

So, what exactly happened here?

- **During server startup, user is created automatically with default username: "user"**
- **Random password is generated for testing.**
- **Each time, server is restarted, new random password will get generated.**

SecurityProperties.java

```
public static class User {

/** Default user name */
private String name = "user";

/** Password for the default username */
Private String password = UUID.randomUUID().toString();

/** Granted roles for the default username */
private List<String> roles = new ArrayList<>();

private boolean passwordGenerated = true;
                .
                .
                //getters and setters
                .
}
```

@AutoConfiguration

UserDetailsServiceAutoConfiguration.java

```
@Bean
public InMemoryUserDetailsManager inMemoryUserDetailsManager(SecurityProperties properties,
        ObjectProvider<PasswordEncoder> passwordEncoder) {
    SecurityProperties.User user = properties.getUser();
    List<String> roles = user.getRoles();
    return new InMemoryUserDetailsManager(User.withUsername(user.getName())
        .password(getOrDeducePassword(user, passwordEncoder.getIfAvailable()))
        .roles(StringUtils.toStringArray(roles))
        .build());
}
```

InMemoryUserDetailsManager.java

```
private final Map <String, Mutable UserDetails> users= new HashMap<>
();
public InMemoryUserDetailsManager(UserDetails... users) {
    for (UserDetails user : users) {
        createUser(user);
    }
}
```

```
@Override
public void createUser(UserDetails user) {
    Assert.isTrue(!userExists(user.getUsername()), message: "user should not exist");
    this.users.put(user.getUsername().toLowerCase(), new MutableUser(user));
```

```
}
```

How we can control the user creation logic?

1st: Using application.properties
(not recommended, only for development and testing)

application.properties

```
spring.security.user.name=my_username
spring.security.user.password=my_password
spring.security.user.roles=ADMIN
```

Internally, it uses reflection and calls setUserName() and setPassword() method of **SecurityProperties.java** and overrides the default values.

Now, during application startup, no default username and default password is created.

```
45512 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
45512 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view ren
45512 --- [           main] r$InitializeUserDetailsManagerConfigurer : Global AuthenticationManager configured with UserDetailsService bean with name inMemoryUserDetailsManager
45512 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
45512 --- [           main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 1.701 seconds (process running for 1.832)
45512 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
45512 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
45512 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
```

2nd: By creating custom InMemoryUserDetailsManager Bean

(not recommended, only for development and testing)

UserDetailsService

↓

UserDetailsManager

↓

InMemoryUserDetailsManager

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withUsername("my_username_1")
                .password("{noop}my_password_1") // {noop} means no encoding or hashing
                .roles("ADMIN")
                .build();

        UserDetails user2 = User.withUsername("my_username_2")
                .password("{noop}1234")  // {noop} means no encoding or hashing
                .roles("USER")
                .build();

        return new InMemoryUserDetailsManager(user1, user2);
    }
}
```

why we are appending {noop} here?

The default, format for storing the password is :
{id}encodedpassword

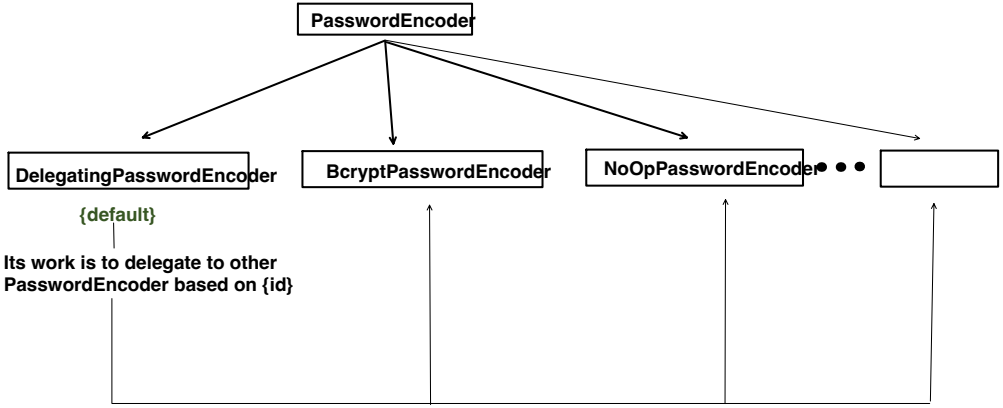
{id} can be either:

- *{noop}*
- *{bcrypt}*
- *{sha256}*
- *Etc..*

-- During User password storing step, if we want to store user password without any encoding or hashing, then we store "{noop}plain_password"

-- Now, during authentication process:

- **1st, it will fetch the user password from inMemory.**
- **2nd, it goes for comparing logic, inMemory password and password provided for authentication.**
- **3rd, it will take out the {noop} or {bcrypt} etc. from inMemory password.**
- **4th, Then if its {noop}, it will directly compare the remaining inMemory password and provided password for authentication.**
- **5th, if say its {bcrypt}, it first do hashing of provided password using BCryptPasswordEncoder and then match it with remaining inMemory Password.**

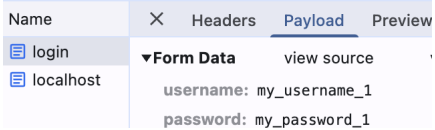


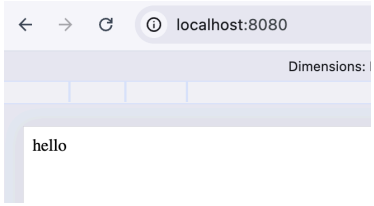
Lets say, if we want to store the hashed password (hashed using ***bcrypt*** algorithm)

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

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withUsername("my_username_1")
                .password("{bcrypt}" + new BCryptPasswordEncoder().encode( rawPassword: "my_password_1"))
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(user1);
    }
}
```

InMemory, password is stored as : {bcrypt}hased_password
and during authentication, I am providing "my_password_1"

But still I am able to successfully authenticate because of **DelegationPasswordEncoder,** it first checks the format of stored password {id} i.e. {bcrypt}, so it passes the incoming password to BcryptPasswordEncoder, and after hashing, it has done the matching.





If, we don't want to store {bcrypt} or any other hashing algo {id} in front of password, then we can define which PasswordEncoder to use.

Now, since we are always using 1 encoding/hashing algorithm, and control will not goes to "DelegationPasswordEncoder", and it will directly goes to specific Password Encoder, so now no need to put {id} in front of password.

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

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

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withUsername("my_username_1")
                .password(new BCryptPasswordEncoder().encode( rawPassword: "my_password_1"))
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(user1);
    }
}
```

3rd: Storing UserName and Password (after hashed) in DB
(recommended for production)

Implements UserDetails because, During Authentication (form, basic, jwt etc..), security framework tries to fetch the user and return the object of UserDetails only, if we don't implement it, then we have to do the mapping (from UserAuthEntity to UserDetails).

UserAuthEntity.java

```java
@Entity
@Table(name = "user_auth")
public class UserAuthEntity 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;

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

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }

    //getters and setters
    @Override
    public String getPassword() {
        return password;
    }
```

UserAuthEntityRepository.java

```java
@Repository
public interface UserAuthEntityRepository extends JpaRepository<UserAuthEntity, Long> {

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

Implements UserDetail because, for the same authentication, based method we are using B etc. it will try to load us we are using DB for sto username and passwo security don't know ho we have to implement UserDetailsService and method "loadUserByU

UserAuthEntityService.java

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

    @Autowired
    private UserAuthEntityRepository userAuthEntityRepository;

    public UserDetails save(UserAuthEntity userAuth) {
        return userAuthEntityRepository.save(userAuth);
    }

    @Override
    public UserAuthEntity loadUserByUsername(String username) throws UsernameNotFoundException

        return userAuthEntityRepository.findByUsername(username)
```

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

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

    public String getRole() {
        return role;
    }

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

```java
                .orElseThrow(() -> new UsernameNotFoundException("User not found"));
    }
}
```

UserAuthController.java

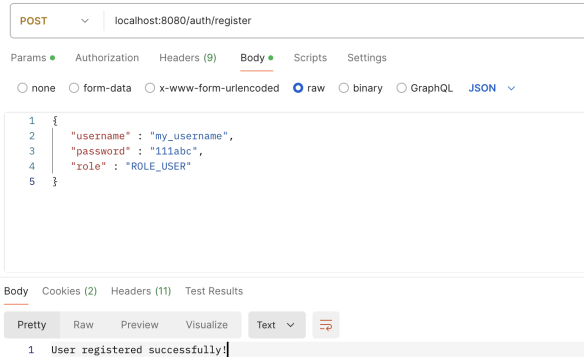
```java
@RestController
@RequestMapping("/auth")
public class UserAuthController {

    @Autowired
    private UserAuthEntityService userAuthEntityService;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @PostMapping("/register")
    public ResponseEntity<String> register(@RequestBody UserAuthEntity userAuthDetails) {
        // Hash the password before saving
        userAuthDetails.setPassword(passwordEncoder.encode(userAuthDetails.getPassword()));

        // Save user
        userAuthEntityService.save(userAuthDetails);
        return ResponseEntity.ok( body: "User registered successfully!");
    }
}
```

Now, by-default in spring boot security, all the endpoints are AUTHENTICATED, means we have to authenticate ourself by either username/password or JWT etc.. To access any API, so how we will access "***/auth/register" API,*** which is just a first step to create user.

Yes, we have to relax the authentication for this API and its industry standard.

SecurityConfig.java

```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: "/auth/register").permitAll()
                        .anyRequest().authenticated()
                )
                .csrf(csrf -> csrf.disable())
                .httpBasic(Customizer.withDefaults());
        return http.build();
    }
}
```



Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_AUTH
```

SELECT * FROM USER_AUTH;

ID	PASSWORD	ROLE	USERNAME
1	$2a$10$euzihUhyp4exMejDkyDb0eK2q49sqTcG8EShOnT2GmaL7lxvIeOfO	ROLE_USER	my_username

(1 row, 3 ms)

Edit