

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$InitializeUserDetailsServiceConfigurer : Global AuthenticationManager configured with UserDetailsService bean with name inMemoryUserDetailsService
```



- 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

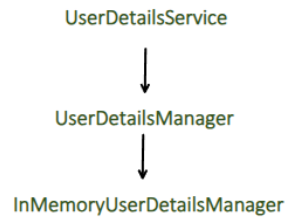
```
spring.security.user.name=my_username  
spring.security.user.password=my_password  
spring.security.user.roles=ADMIN
```

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$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering  
45512 --- [main] r$InitializeUserDetailsServiceConfigurer : Global AuthenticationManager configured with UserDetailsService bean with name inMemoryUserDetailsService  
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 `InMemoryUserDetailsService` Bean
(not recommended, only for development and testing)



```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

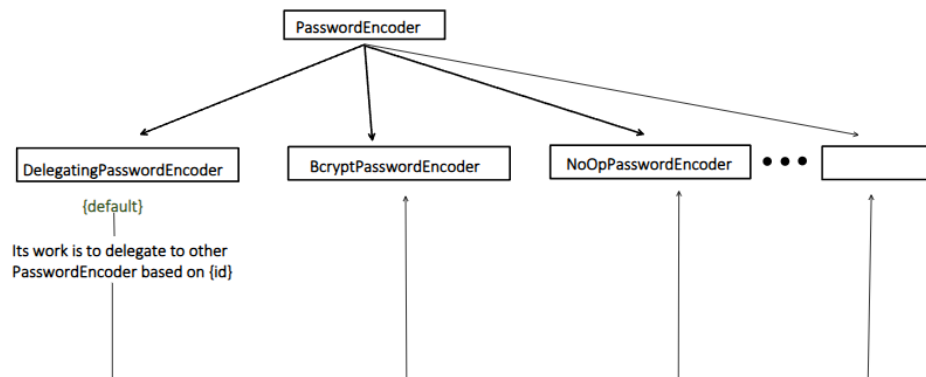
    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withUsername("my_username_1")
            .password("{noop}my_password_1") // {noop} means no encoding or hashing
            .roles("ADMIN")
            .build();

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

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

why we are appending {noop} here?







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


```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    // ...

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

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

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.

Name	X	Headers	<u>Payload</u>	Preview
 login				
 localhost		▼ Form Data	view source	
		username: my_username_1		
		password: my_password_1		
		_csrf: riTNYwhbq7nc26EwHywkGjgRz19ZXZgqeC		



A screenshot of a web browser window. The address bar shows 'localhost:8080'. Below the address bar, there is a light blue header area with the text 'Dimensions:'. The main content area is white and contains the text 'hello'.

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.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

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

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

UserAuthEntity.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;
    }

    @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;
    }
}
```

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).

UserAuthEntityRepository.java

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

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

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

Implements UserDetailsService because, for the same reason, during authentication, based on authentication method we are using Basic, jwt, form etc. It will try to load user first, but since we are using DB for storing the username and password, spring security don't know how to fetch it, so we have to implement UserDetailsService and override the method "loadUserByUsername"

UserAuthController.java


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

UserAuthController.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("User registered successfully!");  
    }  
}
```

Encode the password before storing it in DB.

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

```
@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()  
            );  
    }  
}
```

SecurityConfig.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();
    }
}
```

POST localhost:8080/auth/register

Params Authorization Headers (9) **Body** Scripts Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▾

```
1 {
2   "username": "my_username",
3   "password": "111abc",
4   "role": "ROLE_USER"
5 }
```

Body Cookies (2) Headers (11) Test Results

Pretty Raw Preview Visualize Text

```
1 User registered successfully!
```

Run Run Selected Auto complete Clear SQL statement:

```
SELECT * FROM USER_AUTH
```