

Session 9

Spring Security

Real applications keep users in a database and show their own login page. This session replaces Spring Security's defaults with a User entity, a custom `UserDetailsService`, a custom login form, logout, and registration with automatic login.

Objectives

✕ Do not copy. Read for understanding and the viva

- Complete questions 37 to 40 of the manual: spring security
- Prepare the deliverable before the lab and finish it during the session
- Be ready to explain every step in the viva

Questions Covered

✕ Do not copy. Read for understanding and the viva

Question	Requirement	Status
Q37	Create a User table into the database and bind the user entity with Spring Security...	Complete
Q38	Create a Custom Login Page in HTML and authenticate using Spring Security in Spring Boot	Complete
Q39	Implement logout functionality in Spring Security	Complete
Q40	Create a User registration form and validate the form. Once information is validated...	Complete

Preparation

✗ Do not copy. Read for understanding and the viva

- User table: id, username, password (BCrypt hash), enabled. Never store plain passwords; use `BCryptPasswordEncoder`.
- The `SecurityFilterChain` bean configures `formLogin().loginPage("/login")` and `logout()`.
- Auto-login after registration means authenticating programmatically with `AuthenticationManager` and placing the result in the `SecurityContextHolder`.

The project is still `admission-api` with the security starter from Session 8. `SecurityConfig.java` is one file that grows across Questions 37 to 39; it is listed once, under Question 38, with the parts labelled. Nothing was executed here; every listing and expected page was checked by reading against Spring Security 6.3.

Question 37

Problem Statement

■ Write in lab record

Create a User table into the database and bind the user entity with Spring Security for Login.

Solution

■ Write in lab record

Steps

1. Add the `User` entity in `in.ignou.admission.entity` and `UserRepository` in `in.ignou.admission.repository`.
2. Create the package `in.ignou.admission.security` with `AppUserDetailsService` and `DataSeeder`.
3. Create `SecurityConfig` in the same package with the `PasswordEncoder` bean (full file under Question 38).
4. Restart. Hibernate creates `users`; the seeder inserts `admin` with a BCrypt hash; the console no longer prints a generated password.

5. Check in MySQL: `SELECT id, username, LEFT(password, 7), enabled FROM users;` .

6. Log in at `/login` (still Spring's default page at this point) as `admin` / `admin123` .

Program

- Lab record: every tab is one file of the answer. Write all of them.

User.java

User.java

```
1 // src/main/java/in/ignou/admission/entity/User.java (Session 9, Q37)
2 package in.ignou.admission.entity;
3
4 import jakarta.persistence.Column;
5 import jakarta.persistence.Entity;
6 import jakarta.persistence.GeneratedValue;
7 import jakarta.persistence.GenerationType;
8 import jakarta.persistence.Id;
9 import jakarta.persistence.Table;
10
11 /** Login account. @Table("users") because USER is a reserved word in several
12  * @Entity
13  * @Table(name = "users")
14  */
15 public class User {
16
17     @Id
18     @GeneratedValue(strategy = GenerationType.IDENTITY)
19     private Long id;
20
21     @Column(nullable = false, unique = true, length = 50)
22     private String username;
23
24     /** BCrypt hash, never the raw password. 60 characters. */
25     @Column(nullable = false, length = 100)
26     private String password;
27
28     private boolean enabled = true;
29
30     public Long getId() { return id; }
31     public void setId(Long id) { this.id = id; }
32     public String getUsername() { return username; }
33     public void setUsername(String username) { this.username = username; }
34     public String getPassword() { return password; }
35     public void setPassword(String password) { this.password = password; }
36     public boolean isEnabled() { return enabled; }
37     public void setEnabled(boolean enabled) { this.enabled = enabled; }
38 }
```

The table Hibernate produces, for the record and for anyone running with `ddl-auto=none` :

users.sql

```

1  -- Session 9, Q37: what Hibernate generates for the User entity (ddl-auto=up
2  -- Run by hand only if you keep ddl-auto=none.
3  CREATE TABLE users (
4      id          BIGINT          NOT NULL AUTO_INCREMENT,
5      username    VARCHAR(50)     NOT NULL,
6      password    VARCHAR(100)    NOT NULL,    -- BCrypt hash, always 60 chars
7      enabled     BIT              NOT NULL,
8      PRIMARY KEY (id),
9      UNIQUE KEY uk_users_username (username)
10 ) ENGINE=InnoDB;

```

Output

Checked by reading. Console on first start:

```

1  Hibernate: create table users (enabled bit not null, id bigint not null auto
2  Hibernate: alter table users add constraint UK_users_username unique (userna
3  Hibernate: insert into users (enabled,password,username) values (?,?,:)
4  Seeded user admin / admin123

```

MySQL:

```

1  +----+-----+-----+-----+
2  | id | username | LEFT(password,7) | enabled |
3  +----+-----+-----+-----+
4  |  1 | admin   | $2a$10$          |      1 |
5  +----+-----+-----+-----+

```

The `Using generated security password` line is gone: a `UserDetailsService` bean of our own switches off the in-memory user. Logging in as `admin / admin123` works; `admin / wrong` shows “Bad credentials”; a username that is not in the table shows the same message (Spring hides the difference on purpose).

Explanation

Spring Security never reads a table itself. It calls `UserDetailsService.loadUserByUsername`, receives a `UserDetails` (username, hashed password, enabled flags, authorities) and then asks

the `PasswordEncoder` to compare the submitted password with the stored hash. `AppUserDetailsService` is that adapter: it loads our entity through `UserRepository` and copies it into Spring's own `User` builder. The entity is named `User` too, so the Spring class is referenced by its full name to avoid an import clash. `BCryptPasswordEncoder.encode` produces a 60-character string beginning `$2a$10$`; the salt is inside the string, so the same password gives a different hash each time and `matches` still works. The seeder hashes in Java because a hash cannot be typed by hand into `data.sql`. `@Table(name = "users")` avoids the reserved word `USER` in several databases.

Question 38

Problem Statement

■ Write in lab record

Create a Custom Login Page in HTML and authenticate using Spring Security in Spring Boot.

Solution

■ Write in lab record

Steps

1. Add `PageController` in `in.ignou.admission.web` with `GET /login` and `GET /dashboard`.
2. Add `login.html` and `dashboard.html` under `src/main/resources/templates`.
3. Complete `SecurityConfig` with the `SecurityFilterChain` bean: `/login` permitted for all, `loginPage("/login")`, success URL `/dashboard`.
4. Restart. Open `http://localhost:8080/dashboard`; you are redirected to your own `/login` page.
5. Submit `admin` / `admin123`; the dashboard shows "Welcome, admin". Submit a wrong password; the page returns with the red error line and `?error` in the URL.

Program

■ Lab record: every tab is one file of the answer. Write all of them.

SecurityConfig.java

SecurityConfig.java

```
1 // src/main/java/in/ignou/admission/security/SecurityConfig.java (Session
2 package in.ignou.admission.security;
3
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6 import org.springframework.security.authentication.AuthenticationManager;
7 import org.springframework.security.config.Customizer;
8 import org.springframework.security.config.annotation.authentication.configu
9 import org.springframework.security.config.annotation.web.builders.HttpSecur
10 import org.springframework.security.config.annotation.web.configuration.Enab
11 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
12 import org.springframework.security.crypto.password.PasswordEncoder;
13 import org.springframework.security.web.SecurityFilterChain;
14
15 @Configuration
16 @EnableWebSecurity
17 public class SecurityConfig {
18
19     /** Q37: BCrypt with a random salt per password; strength 10 (default) =
20     @Bean
21     public PasswordEncoder passwordEncoder() {
22         return new BCryptPasswordEncoder();
23     }
24
25     /** Q40 uses this to log the new user in programmatically. */
26     @Bean
27     public AuthenticationManager authenticationManager(AuthenticationConfigu
28         return config.getAuthenticationManager();
29     }
30
31     @Bean
32     public SecurityFilterChain filterChain(HttpSecurity http) throws Excepti
33         http
34             .authorizeHttpRequests(auth -> auth
35                 .requestMatchers("/login", "/register", "/css/**", "/actuato
36                 .anyRequest().authenticated())
37             // Q38: our own page at GET /login; the POST /login handler is s
38             .formLogin(form -> form
39                 .loginPage("/login")
40                 .defaultSuccessUrl("/dashboard", true)
41                 .permitAll())
42             // Q39: POST /logout ends the session and returns to the login p
43             .logout(logout -> logout
```

```

44         .logoutUrl("/logout")
45         .logoutSuccessUrl("/login?logout")
46         .invalidateHttpSession(true)
47         .deleteCookies("JSESSIONID")
48         .permitAll())
49         // lets curl -u user:pass call /api/** (browser still uses the f
50         .httpBasic(Customizer.withDefaults())
51         // the JSON API is called by curl/Postman, not from a browser se
52         .csrf(csrf → csrf.ignoringRequestMatchers("/api/**", "/xml/**",
53         return http.build();
54     }
55 }

```

Output

Checked by reading. Browser flow:

1. GET /dashboard while logged out: 302 Location: http://localhost:8080/login .
2. /login renders the heading "Student Admission Portal", two inputs, a Sign in button and a Register link. View source shows a hidden field _csrf inside the form, added by th:action .
3. Wrong password: 302 Location: /login?error , and the page shows "Invalid username or password." in red.
4. Right password: 302 Location: /dashboard , page shows "Welcome, admin." with two links and a Logout button.

The same flow from curl, keeping cookies in a jar:

```
1 curl -s -c jar.txt http://localhost:8080/login | grep _csrf
```

BASH

```
1 <input type="hidden" name="_csrf" value="Zt3k9...">
```

```
1 curl -i -b jar.txt -c jar.txt -X POST http://localhost:8080/login \
2   -d "username=admin" -d "password=admin123" -d "_csrf=Zt3k9..."
```

BASH

```
1 HTTP/1.1 302
2 Location: http://localhost:8080/dashboard
```

```
1 curl -s -b jar.txt http://localhost:8080/dashboard | grep Welcome
```

BASH

```
1 <p>Welcome, <span>admin</span>.</p>
```

The JSON API also works with HTTP Basic now, including writes, because `/api/**` is excluded from CSRF:

```
1 curl -s -u admin:admin123 http://localhost:8080/api/students/1
```

BASH

```
1 {"id":1,"name":"Asha Verma","email":"asha@example.com","phone":"9876543210",
```

Explanation

Only the GET half of `/login` is ours. `PageController.login()` returns the Thymeleaf template; the POST to `/login` is still handled by `UsernamePasswordAuthenticationFilter`, which reads the `username` and `password` parameters (the input names must match), calls the `AuthenticationManager`, and on success saves the `SecurityContext` in the HTTP session and redirects. `defaultSuccessUrl("/dashboard", true)` forces the redirect target; without `true` Spring sends the user back to whatever protected URL they first asked for. `permitAll()` on the login block covers `/login`, `/login?error` and `/login?logout`. The `param.error` expression in the template reads the query string, which is how the page knows to show the message. `httpBasic` stays on so curl and Postman can call the API with `-u`; `csrf.ignoringRequestMatchers("/api/**", ...)` lets those token-less calls through while every browser form keeps its token.

Question 39

Problem Statement

■ Write in lab record

Implement logout functionality in Spring Security.

Solution

■ Write in lab record

Steps

1. The `.logout(...)` block in `SecurityConfig` (listed under Question 38) sets the URL, the redirect, session invalidation and cookie removal.
2. `dashboard.html` already carries the Logout form. It must be a form with `method="post"`; a plain link does not work.
3. Log in, click Logout. The browser lands on `/login?logout` with the green "You have been logged out." line.
4. Press the browser Back button: the dashboard does not reappear; `/dashboard` redirects to `/login` because the session is gone.

Program

The two parts, extracted from the files above:

```
1 // SecurityConfig.filterChain, Q39 part
2 .logout(logout → logout
3     .logoutUrl("/logout")           // POST /logout triggers LogoutFilter
4     .logoutSuccessUrl("/login?logout") // where to go afterwards
5     .invalidateHttpSession(true)    // drop the server-side session
6     .deleteCookies("JSESSIONID")   // and tell the browser to forget it
7     .permitAll())
```

JAVA

```
1 <!-- dashboard.html, Q39 part -->
2 <form th:action="@{/logout}" method="post">
3   <button type="submit">Logout</button>
4 </form>
```

HTML

Output

Checked by reading. Browser: after clicking Logout the address bar shows

`http://localhost:8080/login?logout` and the page shows "You have been logged out." Re-entering `/dashboard` by hand redirects to `/login`.

curl, continuing from the jar of Question 38 (a fresh CSRF token is read from the dashboard first):

```
1 TOKEN=$(curl -s -b jar.txt http://localhost:8080/dashboard | grep -o '_csrf' BASH)
2 curl -i -b jar.txt -c jar.txt -X POST http://localhost:8080/logout -d "_csrf"
```

```
1 HTTP/1.1 302
2 Location: http://localhost:8080/login?logout
3 Set-Cookie: JSESSIONID=; Max-Age=0; Expires=Thu, 01 Jan 1970 00:00:10 GMT; P
```

```
1 curl -i -b jar.txt http://localhost:8080/dashboard BASH
```

```
1 HTTP/1.1 302
2 Location: http://localhost:8080/login
```

A GET to `/logout` does nothing useful (it is not a mapping of ours and CSRF only protects state-changing methods): with the form login active, `GET /logout` is simply redirected to `/login` like any other protected URL. A POST without the token returns `403`.

Explanation

`LogoutFilter` sits early in the chain and only reacts to `POST /logout` (POST because CSRF is enabled; with CSRF off, GET would also work). It runs the configured handlers in order:

`SecurityContextLogoutHandler` clears the `SecurityContextHolder` and invalidates the `HttpSession`, `CookieClearingLogoutHandler` writes a `Set-Cookie` with `Max-Age=0` for `JSESSIONID`, and then `SimpleUrlLogoutSuccessHandler` redirects to `/login?logout`. Because the session object is destroyed on the server, the old `JSESSIONID` value is useless even if the browser still has it, which is why the Back button test fails cleanly.

Question 40

Problem Statement

■ Write in lab record

Create a User registration form and validate the form. Once information is validated and saved, write functionality to auto-login using Spring Security.

Solution

■ Write in lab record

Steps

1. Add `spring-boot-starter-validation` to `pom.xml` and run Maven, Update Project.
2. Add `RegistrationForm` and `RegistrationController` in `in.ignou.admission.web`, and `register.html` in `templates`.
3. `/register` is already in the `permitAll()` list of `SecurityConfig`; `authenticationManager` is already a bean there.
4. Restart. Open `/register`, submit an empty form: three red messages. Submit `ab` as the username: the length message. Submit mismatched passwords: "Passwords do not match". Submit `admin`: "Username is already taken".
5. Submit a valid form (`ravi_k`, `ravi123`, `ravi123`): the browser lands on `/dashboard` showing "Welcome, ravi_k." without visiting the login page.
6. Check MySQL: `SELECT username, LEFT(password,7) FROM users;` shows the second row with a `$2a$10$` hash.

Program

■ Lab record: every tab is one file of the answer. Write all of them.

pom.xml fragment

pom.xml fragment

```
1 <!-- Session 9, Q40: Bean Validation (@NotBlank, @Size, @Email). Add inside
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-validation</artifactId>
5 </dependency>
```

Output

Checked by reading. Validation messages as rendered next to each field:

Input	Message shown
empty username	Username is required
ab	Username must be 4 to 50 characters
ravi k (space)	Letters, digits and underscore only
admin	Username is already taken
password 123	Password must be 6 to 40 characters
confirm differs	Passwords do not match

Successful registration from curl (token from the register page, then the POST, then the dashboard with the same jar):

```
1 curl -s -c jar2.txt http://localhost:8080/register | grep -o '_csrf' value="BASH
```

```
1 Q7pM2...
```

```
1 curl -i -b jar2.txt -c jar2.txt -X POST http://localhost:8080/register BASH
2 -d "username=ravi_k" -d "password=ravi123" -d "confirmPassword=ravi123" -d
```

```
1 HTTP/1.1 302
2 Location: http://localhost:8080/dashboard
```

```
1 curl -s -b jar2.txt http://localhost:8080/dashboard | grep Welcome BASH
```

```
1 <p>Welcome, <span>ravi_k</span>.</p>
```

Console during the POST:

```
1 Hibernate: select u1_0.id,u1_0.enabled,u1_0.password,u1_0.username from user
2 Hibernate: insert into users (enabled,password,username) values (?, ?, ?)
3 Hibernate: select u1_0.id,u1_0.enabled,u1_0.password,u1_0.username from user
```

The first select is `existsByUsername`, the insert is `save`, and the last select is `loadUserByUsername` called by the `AuthenticationManager` during auto-login.

Explanation

Validation runs in two layers. The annotations on `RegistrationForm` (`@NotBlank`, `@Size`, `@Pattern`) are Bean Validation constraints; `@Valid` on the handler parameter triggers them, and the errors land in the `BindingResult` declared right after the parameter. Checks that need data or two fields, password match and username uniqueness, are added by hand with `rejectValue`, which puts them in the same `BindingResult` so the template shows them the same way through `th:errors`. Only when `hasErrors()` is false is the entity built; the raw password is hashed once and the form object is discarded.

Auto-login repeats what the login filter does.

`UsernamePasswordAuthenticationToken.unauthenticated(username, rawPassword)` is the request; `authenticationManager.authenticate` returns an authenticated token after `AppUserDetailsService` and BCrypt agree. The token goes into a fresh `SecurityContext`, which is set on the holder for the current thread and saved to the HTTP session by `HttpSessionSecurityContextRepository`. That save is the step students miss: Spring Security 6 no longer saves the context automatically, so without `saveContext` the redirect to `/dashboard` would arrive with an anonymous session and bounce to `/login`.

Viva Questions

✗ Do not copy. Read for understanding and the viva

- **Q:** What does Spring Security call to look up a user, and what does it get back? **A:** `UserDetailsService.loadUserByUsername`; it returns a `UserDetails` with username, hashed password, flags and authorities.
- **Q:** Why does BCrypt give a different hash for the same password each time? **A:** It generates a random salt and stores it inside the hash string; `matches` re-reads the salt.
- **Q:** Which part of `/login` is our code and which is Spring's? **A:** The GET page is ours; the POST is handled by `UsernamePasswordAuthenticationFilter`.

- **Q:** Why must the logout button be a POST form? **A:** CSRF protection is on, so `LogoutFilter` only accepts POST with a valid token.
- **Q:** Why must `BindingResult` follow the `@Valid` parameter directly? **A:** Spring binds errors to the immediately preceding model attribute; otherwise it throws `MethodArgumentNotValidException`.
- **Q:** Why is `confirmPassword` in the form class but not in the entity? **A:** It exists only to validate input; it is never stored.
- **Q:** Which line makes the auto-login survive the redirect? **A:** `contextRepository.saveContext(context, request, response)`, which stores the context in the HTTP session.
- **Q:** Why does the seeder hash the password in Java instead of `data.sql`? **A:** BCrypt output cannot be typed by hand; `encode` produces a salted hash at run time.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Storing the raw password or hashing with MD5/SHA; Spring refuses raw passwords (`There is no PasswordEncoder mapped for the id "null"`) and SHA has no salt.
- Naming the inputs `user` and `pass`; Spring reads `username` and `password` and every login fails with "Bad credentials".
- Writing the login form with `action="/login"` instead of `th:action`, which drops the CSRF token and produces a 403 on submit.
- Using a link `<a href="/logout">` for logout and reporting that logout does not work.
- Forgetting `spring-boot-starter-validation`; `@Valid` compiles but no constraint runs and empty forms are saved.
- Setting the `SecurityContextHolder` but not saving the context to the session, so the auto-login is lost on the very next request.

Session Summary

■ Write in lab record

- `User` entity, `UserRepository`, `AppUserDetailsService`, `DataSeeder` and the generated `users` table with one BCrypt row

- `SecurityConfig` with `PasswordEncoder`, `AuthenticationManager` and the `SecurityFilterChain` (custom login, logout, basic auth, CSRF exclusions for the API)
- `PageController`, `login.html` and `dashboard.html` with the browser flow and the curl transcript
- The logout block, the POST form and the transcript showing the 302 to `/login?logout` and the cleared cookie
- `RegistrationForm`, `RegistrationController`, `register.html`, the validation message table and the auto-login transcript

 [Edit this page](#)

[< Previous](#)
Session 8

[Next >](#)
Session 10