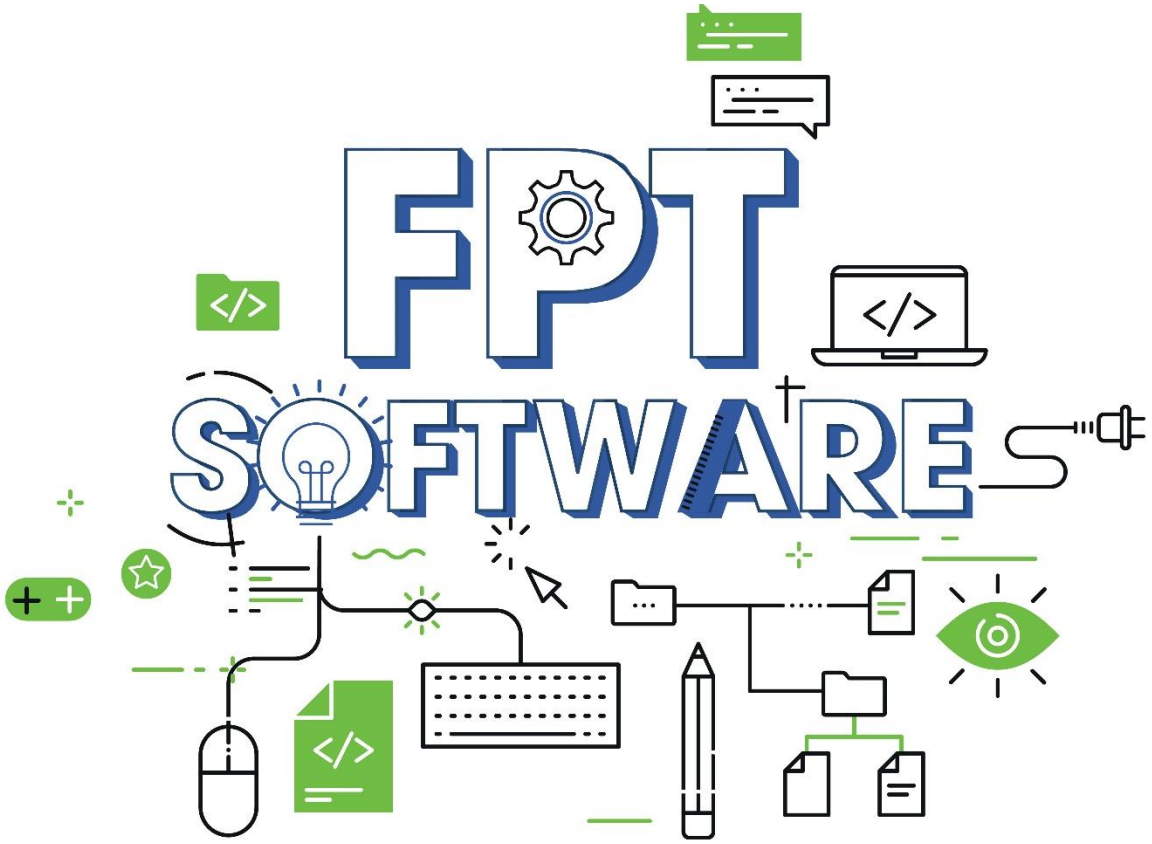
Spring Security for MVC Application

Fsoft Academy



Agenda

1. Introduction & Architecture
2. Username/Password Authentication
3. Authorization
4. Remember-Me Authentication
5. Handling Logouts

Lesson Objectives

1. Understand the fundamentals of Spring Security and its architecture in web applications.
2. Able to configure and utilize username/password authentication.
3. Able to use of password encoders to secure user passwords in Spring Security.
4. Know how to enable and use the Remember-Me feature in Spring Security to maintain user logins for an extended period.
5. Able to handle logouts in Spring Security and configure the logout functionality in a web application.

Section 1

Introduction

What is Security?

What is security?	Non function requirement	Types of security
Security is for **protecting** your **data** and **business logic** inside your web applications.	Security is very important similar to *scalability, performance* and *availability*. No client will specifically asks that I need security	Firewalls, HTTPS, SSL, Authentication, Authorization...

What is Spring Security?

- **Spring Security** is a framework that focuses on providing both authentication and authorization to Java applications.

- Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements

***Note**: The Spring Security version mentioned in the slide is 6.1.5 version*

Spring Security features

- Authentication
- Authorization
- Protection Against Common attacks
- Session Management
- Remember-Me Authentication

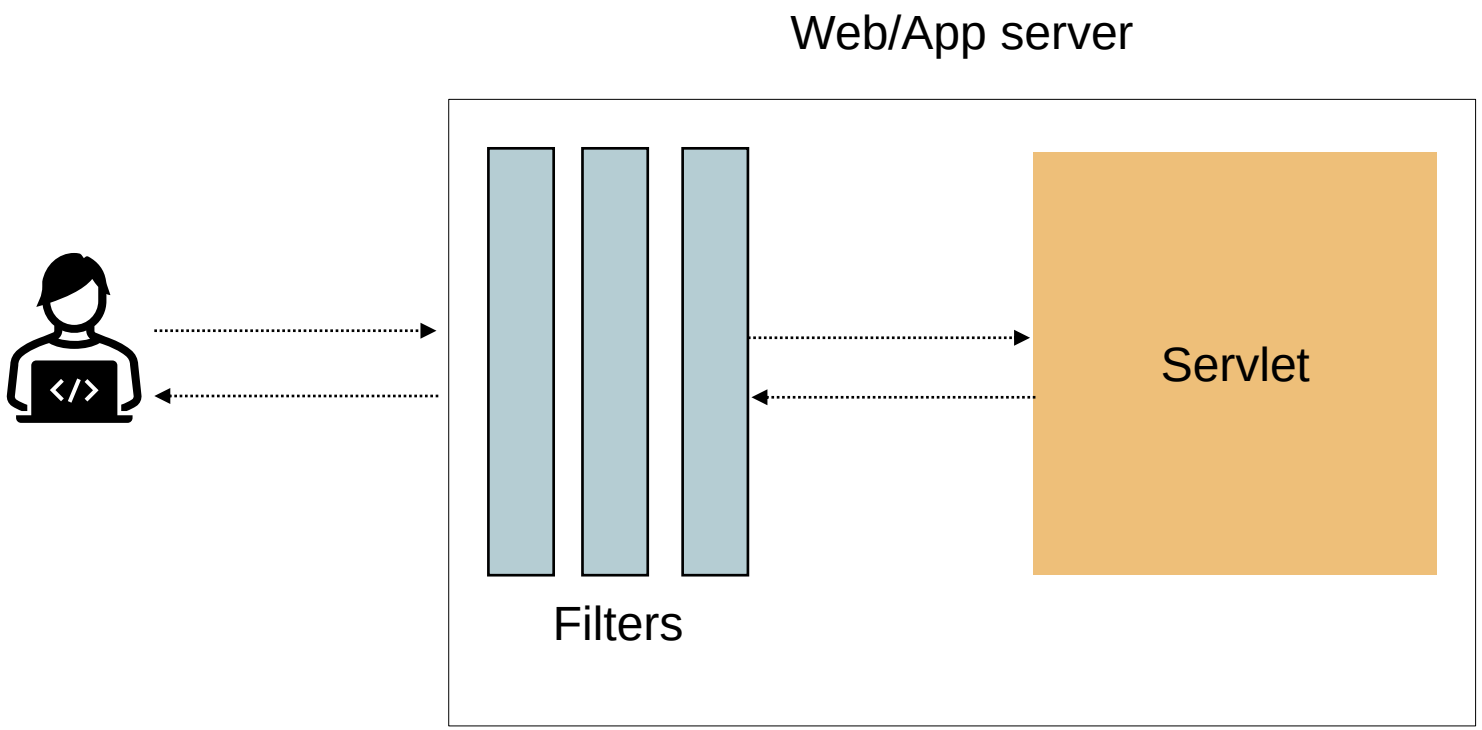
Why should we use Spring Security?

- **Spring Security** built by a team at Spring who are good at security by considering all the security scenarios.
- **Using Spring Security, we can secure web apps with minimum configurations.**
- **Spring Security handles the common security vulnerabilities like CSRF, CORS...**
- **Spring Security supports various standards of security to implement authentication, like using username/password, JWT tokens, OAuth2, OpenID...**

Section 2

Architecture

Recall the servlet filters

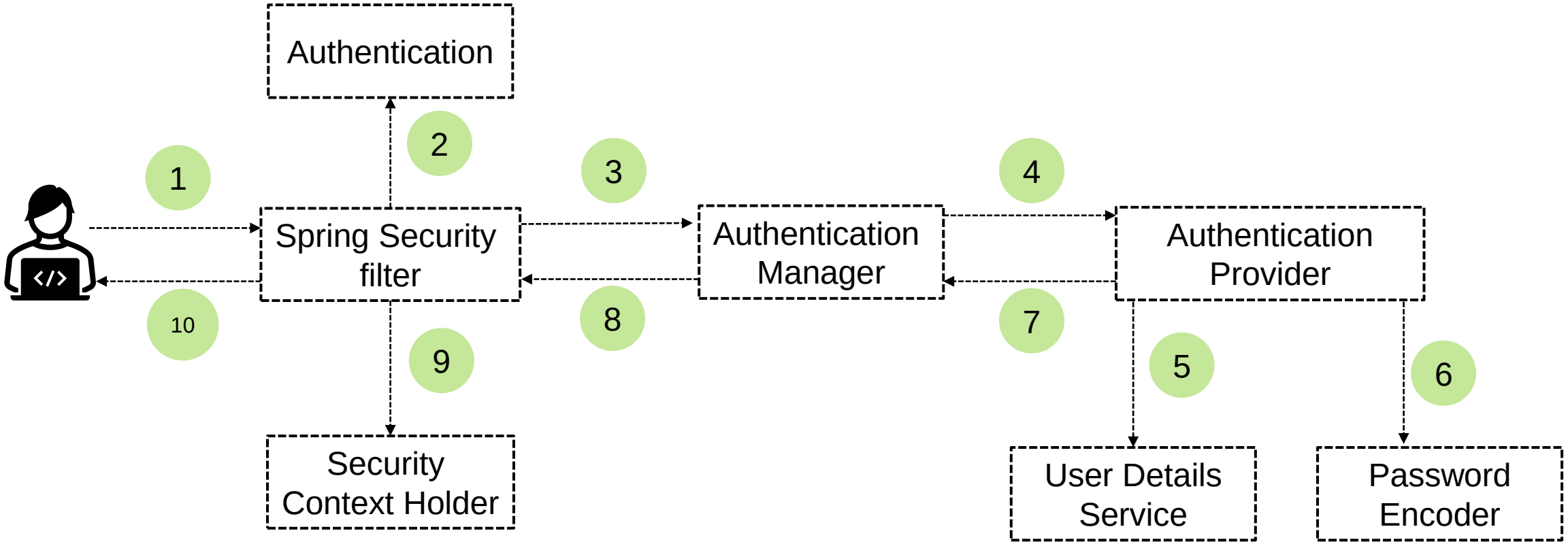


Role of Filters:

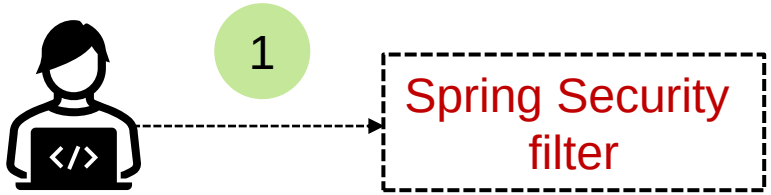
Filters inside Java web application can be used to **intercept** each request/response and do some pre work **before** our business logic.

So using the **same filters**, Spring Security enforce security based on own configuration inside a web application.

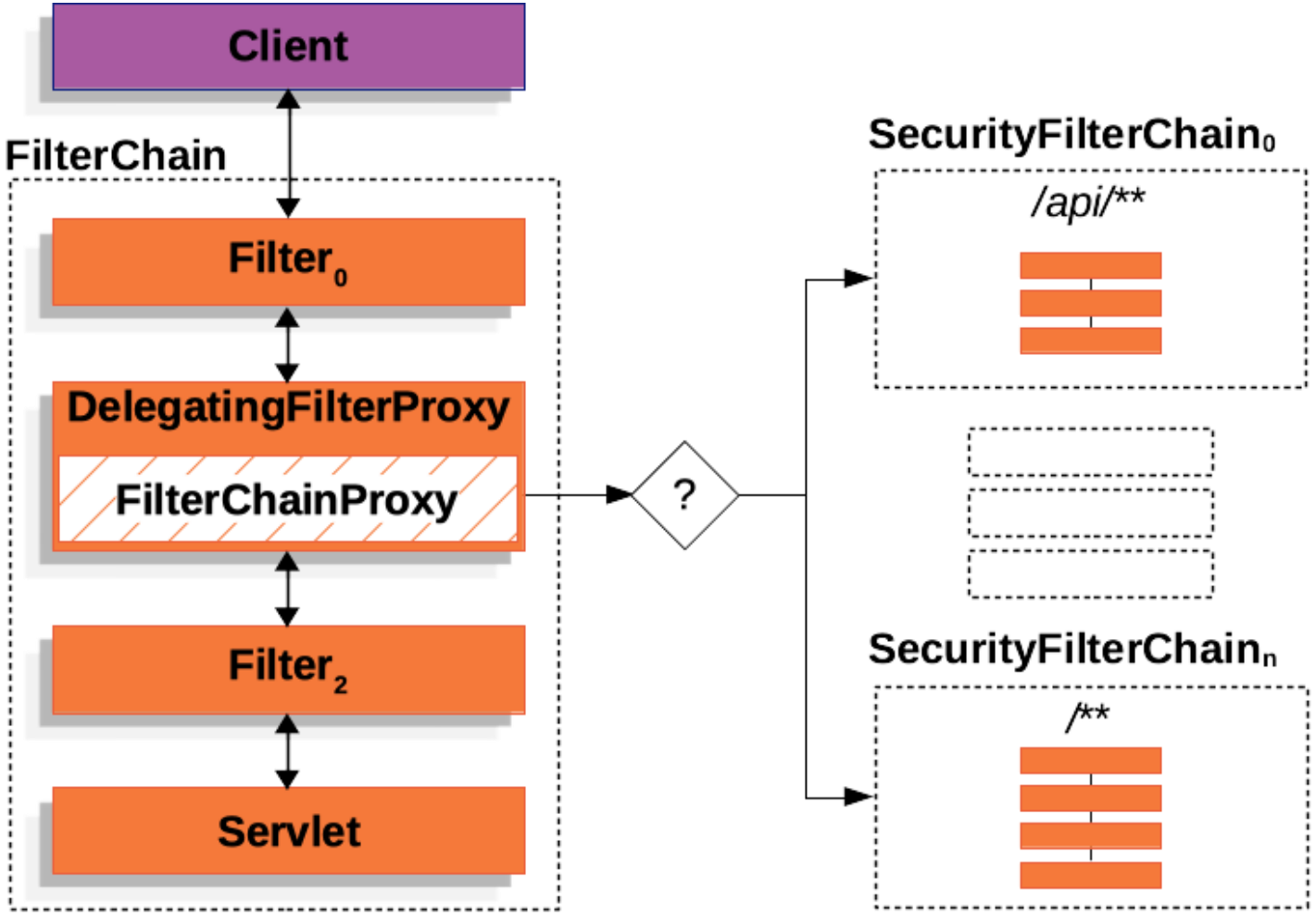
Spring Security internal flow



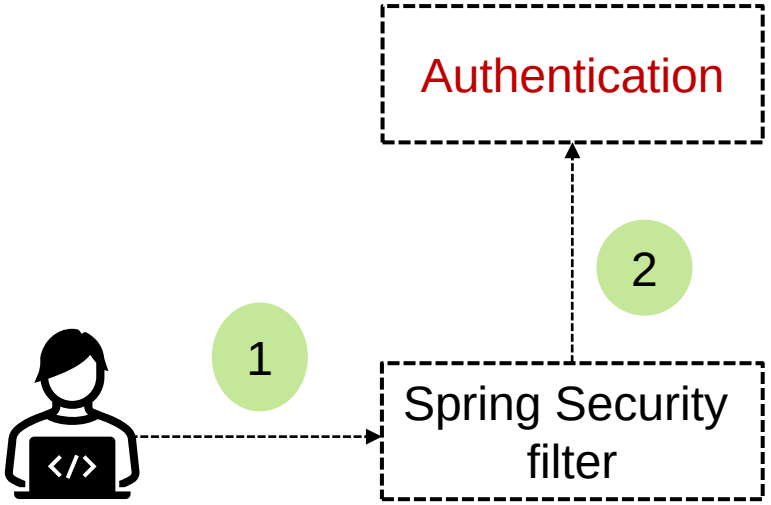
Spring Security filter



- **Spring Security** filter is a series of filters intercept each request & work together to identify if Authentication is required or not



Authentication interface



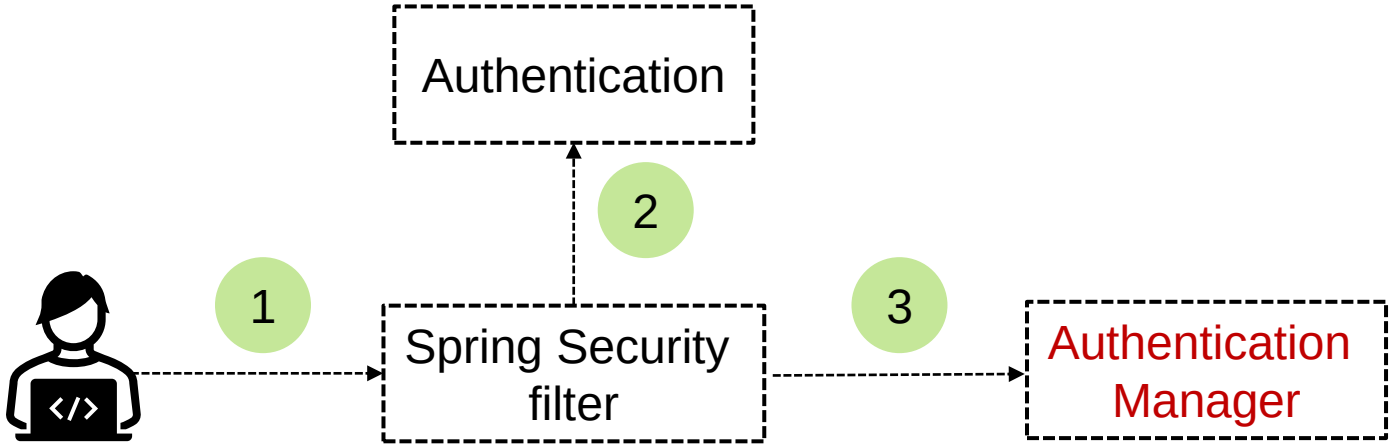
- The **Authentication** interface serves two main purposes within Spring Security:
  - ✓ An input to `AuthenticationManager` to provide the credentials a user has provided to authenticate.
  - ✓ Represent the currently authenticated user after login success

Authentication interface

- The **Authentication** contains:
  - ✓ *`principal`*: Identifies the user. When authenticating with a username/password this is often an instance of `UserDetails`.
  - ✓ *`credentials`*: Often a password. In many cases, this is cleared after the user is authenticated, to ensure that it is not leaked.
  - ✓ *`authorities`*: The `GrantedAuthority` instances are high-level permissions the user is granted. Two examples are roles and scopes.

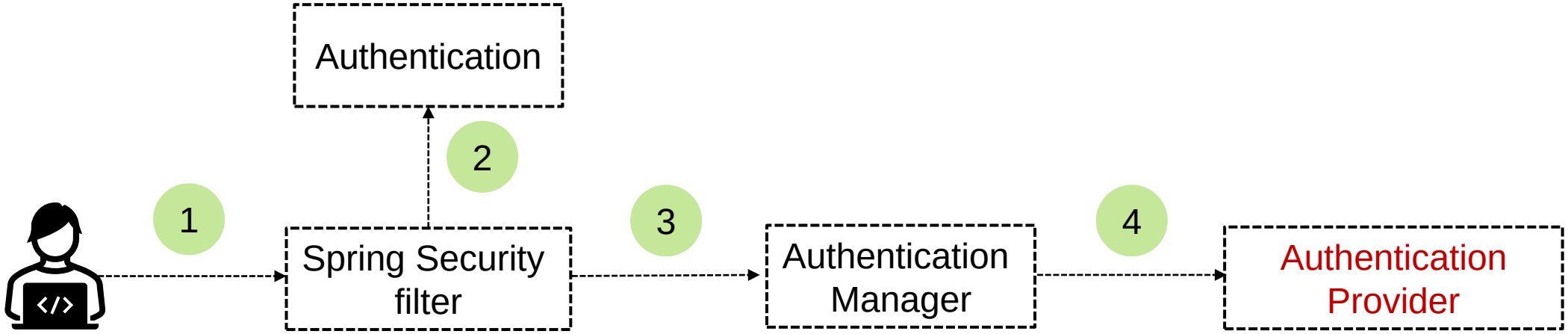
Authentication Manager

- `AuthenticationManager` is the API that defines how Spring Security's Filters perform authentication.



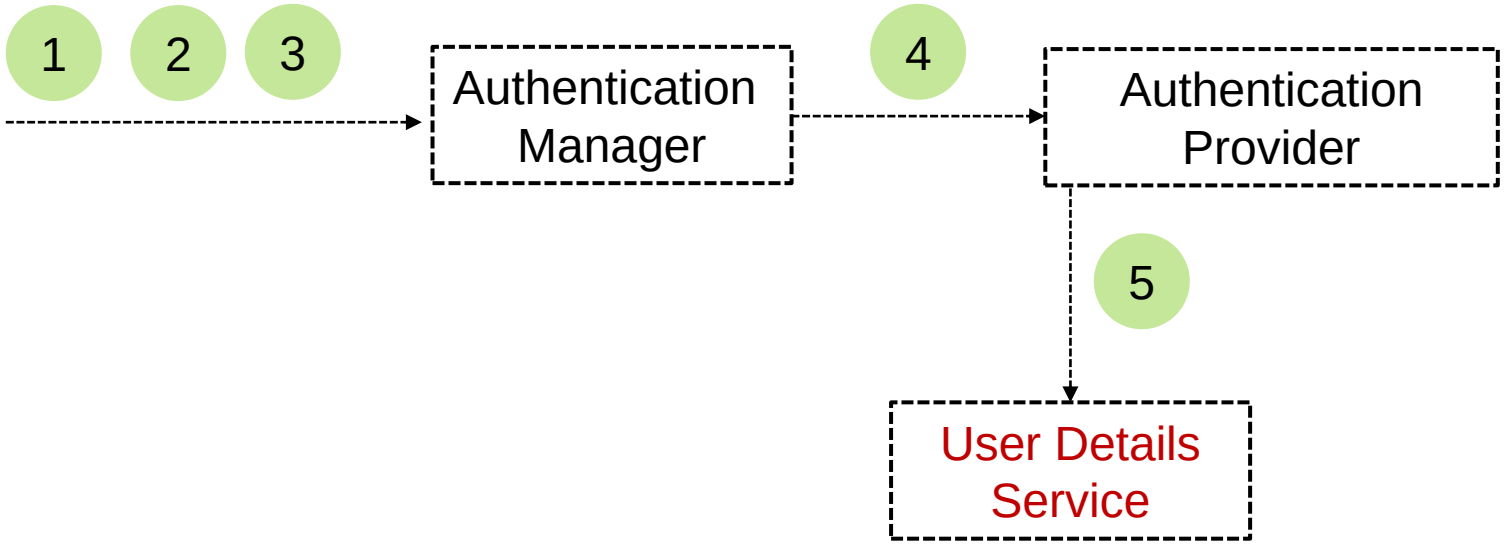
Authentication Provider

- Each `AuthenticationProvider` performs a *specific type of authentication*.
- For example,
  - ✓ `DaoAuthenticationProvider` supports username/password-based authentication
  - ✓ `JwtAuthenticationProvider` supports authenticating a JWT token.



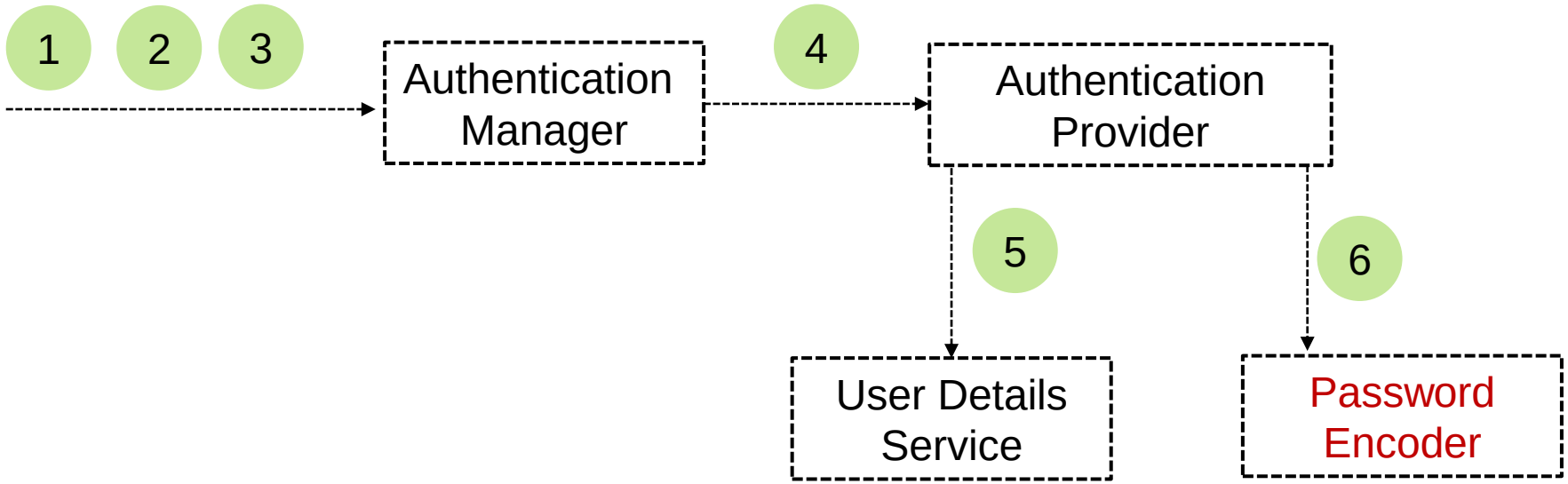
User Details Service

- An interface used to load user details during authentication.
- Implement it to validate user information.
- **Spring Security** relies on it to authenticate users.



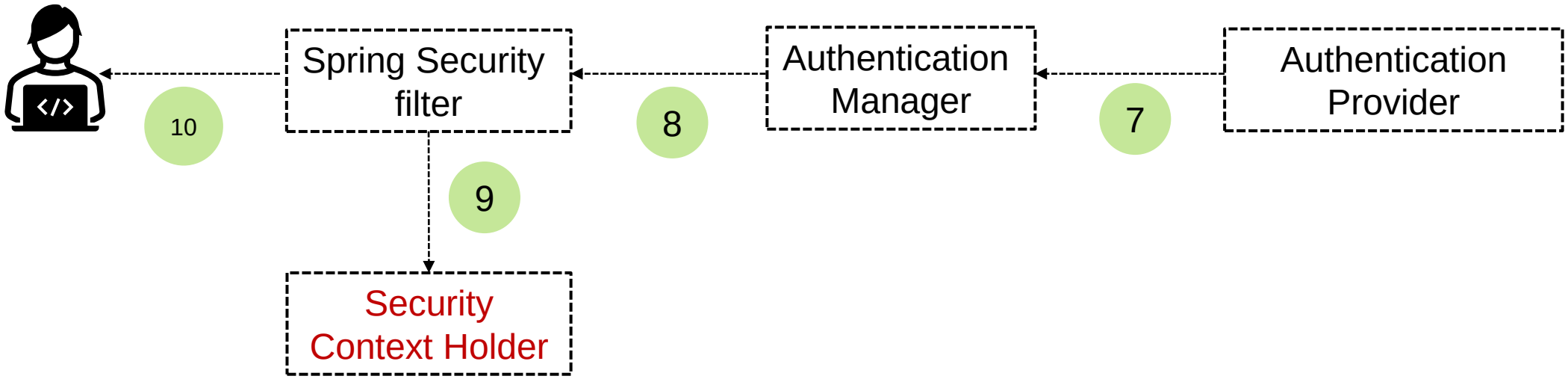
Password encoder

- An interface for securely encoding and verifying passwords.
- Prevents **storing plain-text passwords** in databases.
- Enhances security by protecting user credentials.



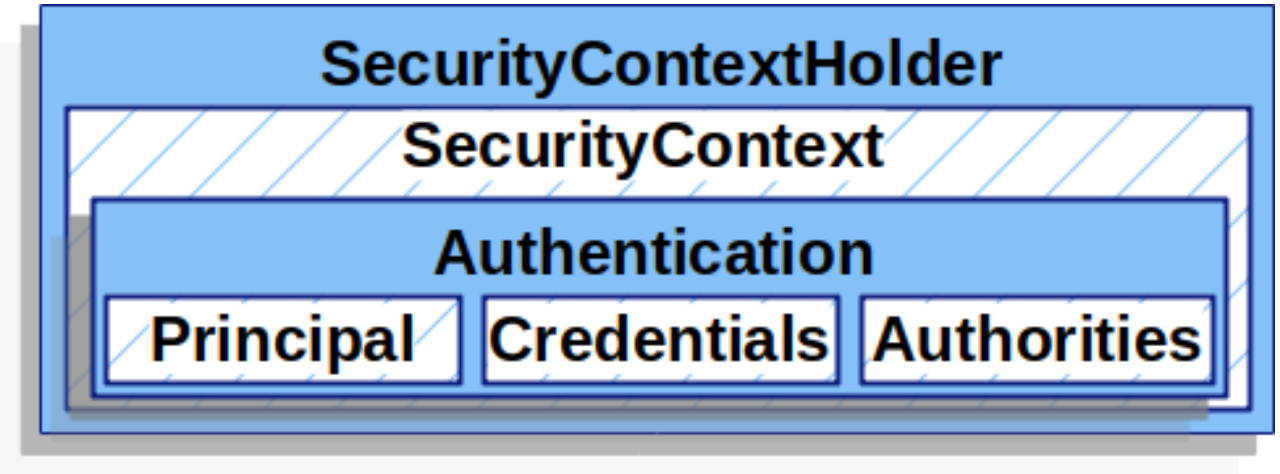
SecurityContextHolder

- The `SecurityContextHolder` is where Spring Security **stores the details of who is authenticated**.
- If it contains a value, it is used as the **currently authenticated user**.



SecurityContext

- The `SecurityContext` is obtained from the `SecurityContextHolder`.
- It contains an `Authentication` object.



Section 3

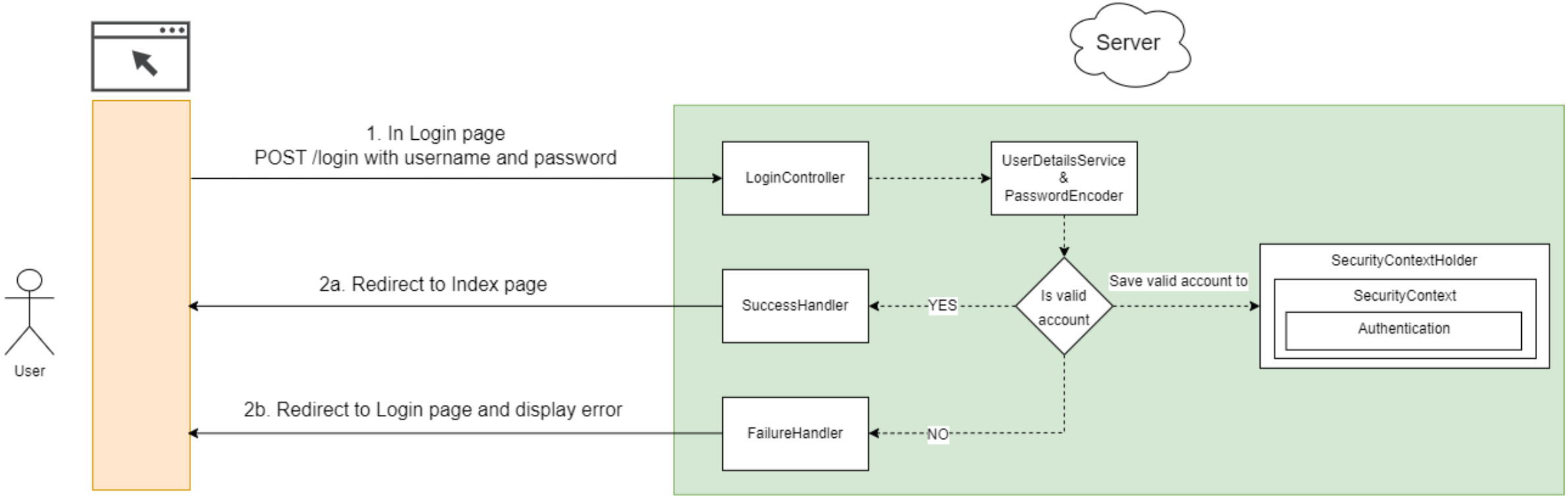
Username/Password Authentication

Installation - Spring Boot with Maven

- Spring Boot provides a **spring-boot-starter-security** starter that aggregates Spring Security-related dependencies

```
<dependencies>
  <!-- ... other dependency elements ... -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
</dependencies>
```

Authentication Flow



Database structure

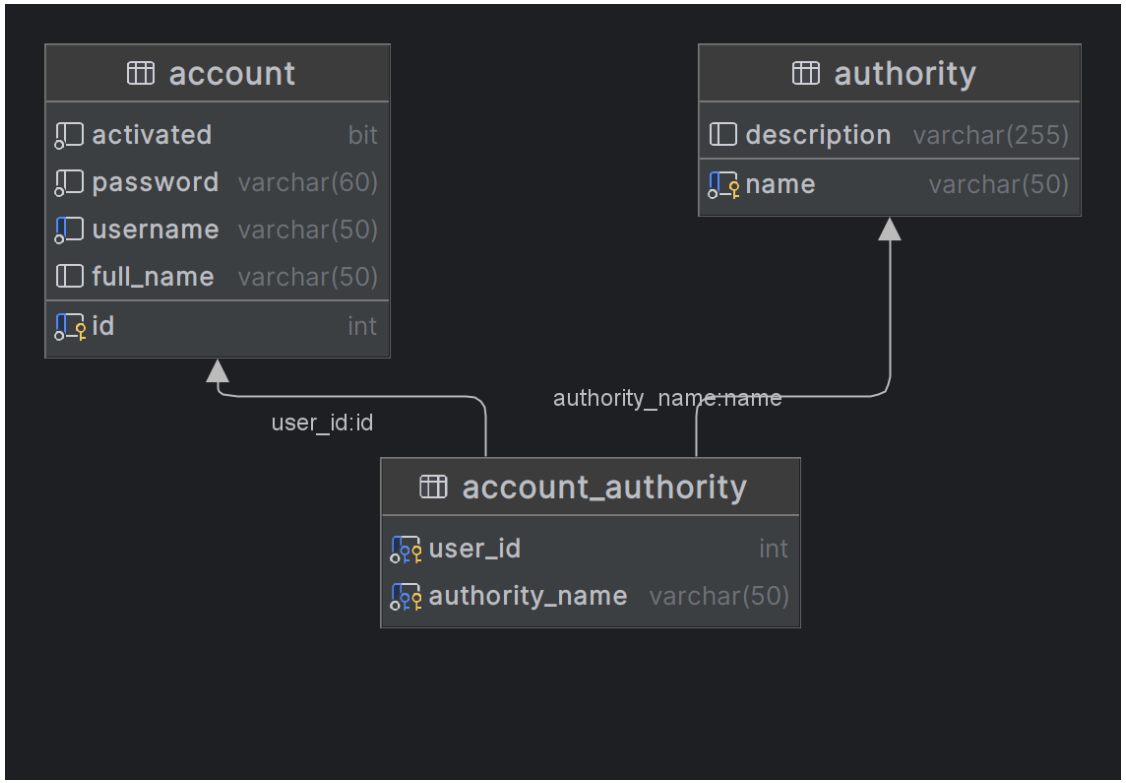


Table	Description
account	Table storing user information
authority	Table storing permissions information
account_authority	Table storing user permissions information

Spring Security filter

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable())
          .authorizeHttpRequests((requests) -> requests.anyRequest().authenticated())
                    .formLogin((form) -> form
                    .loginPage("/login")
                     .defaultSuccessUrl("/index").permitAll())
                  .logout(LogoutConfigurer::permitAll);

        return http.build();
    }
}
```

Explain:

- ✓ *All HTTP requests require authentication except for the login page, which is set to "/login."*
- ✓ *After successful login, users are redirected to "/index."*

PasswordEncoder

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

Explain:

- ✓ *This method defines a `BCryptPasswordEncoder` as a `PasswordEncoder` bean.*
- ✓ *The `BCryptPasswordEncoder` implementation uses the widely supported bcrypt algorithm to hash the passwords*

UserDetailsService

```
@Override
@Transactional(readOnly = true)
public UserDetails loadUserByUsername(final String username) {

    return accountRepository
            .findByUsernameIgnoreCase(username)
            .map(account -> createSpringSecurityUser(username, account))
            .orElseThrow(() ->
        new UsernameNotFoundException("Account: " + username
                                                + " was not found in the database"));
}
```

Explain :

- ✓ *It takes an `AccountRepository` to fetch user information from a database*
- ✓ *The `loadUserByUsername` method finds a user by username, maps it to a `UserDetails` object*

UserDetailsService

```
private User createSpringSecurityUser(String username, Account account) {
    if (!account.isActivated()) {
        throw new UserNotActivatedException("Account: " + username + " was not activated");
    }
    List<GrantedAuthority> grantedAuthorities = account
        .getAuthorities()
        .stream()
        .map(authority -> "ROLE_" + authority.getName()) // ROLE_ADMIN or ROLE_USER
        .map(SimpleGrantedAuthority::new)
        .collect(Collectors.toList());
    return new User(account.getUsername(), account.getPassword(), grantedAuthorities);
}
```

Explain:

- ✓ *If the account is not activated, it raises a `UserNotActivatedException`.*
- ✓ *It creates a Spring Security user with granted authorities and returns it for authentication*

SecurityContext

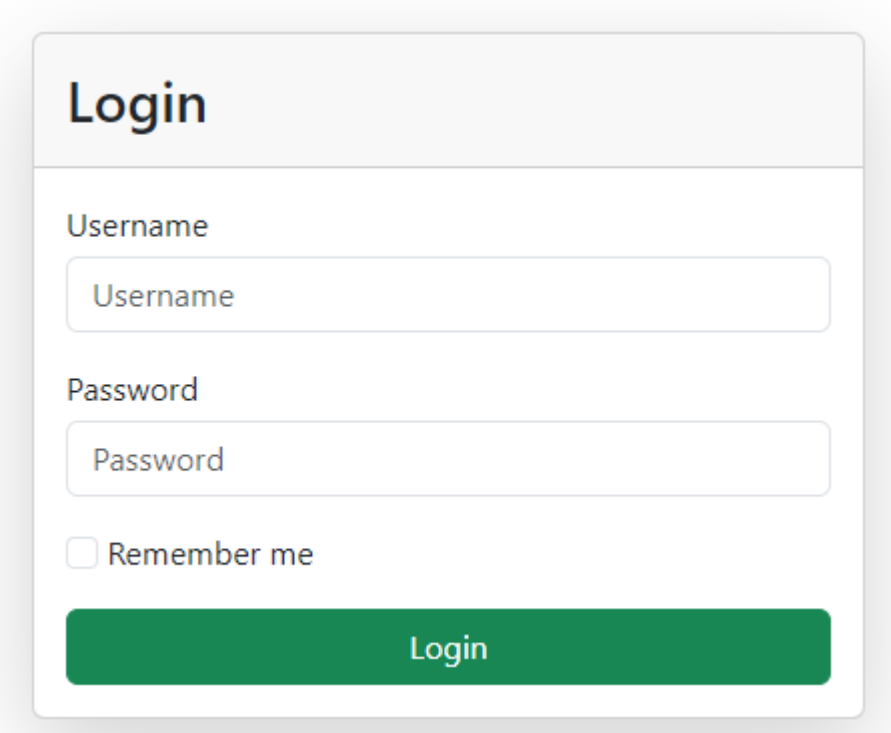
```
public static Optional<String> getCurrentUserLogin() {
    SecurityContext securityContext = SecurityContextHolder.getContext();
    Authentication authentication = securityContext.getAuthentication();

    if (authentication != null && authentication.getPrincipal() instanceof UserDetails springSecurityUser) {
        return Optional.of(springSecurityUser.getUsername());
    }
    return Optional.empty();
}
```

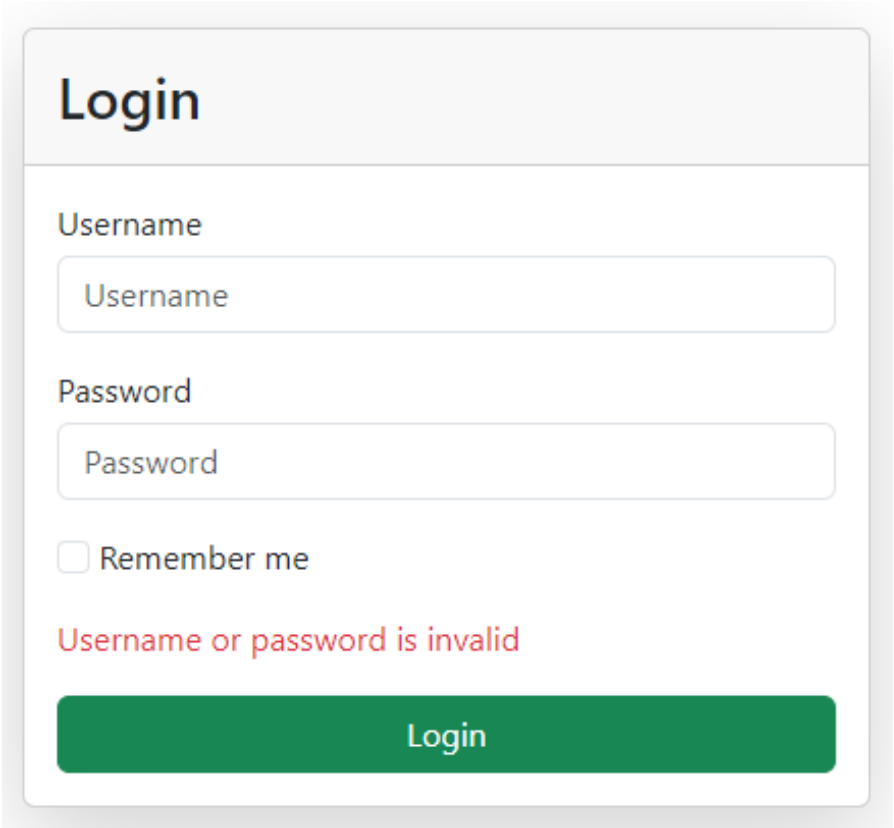
Explain:

✓ *Retrieve the login username of the currently authenticated user from* `SecurityContext`

Authentication Demo

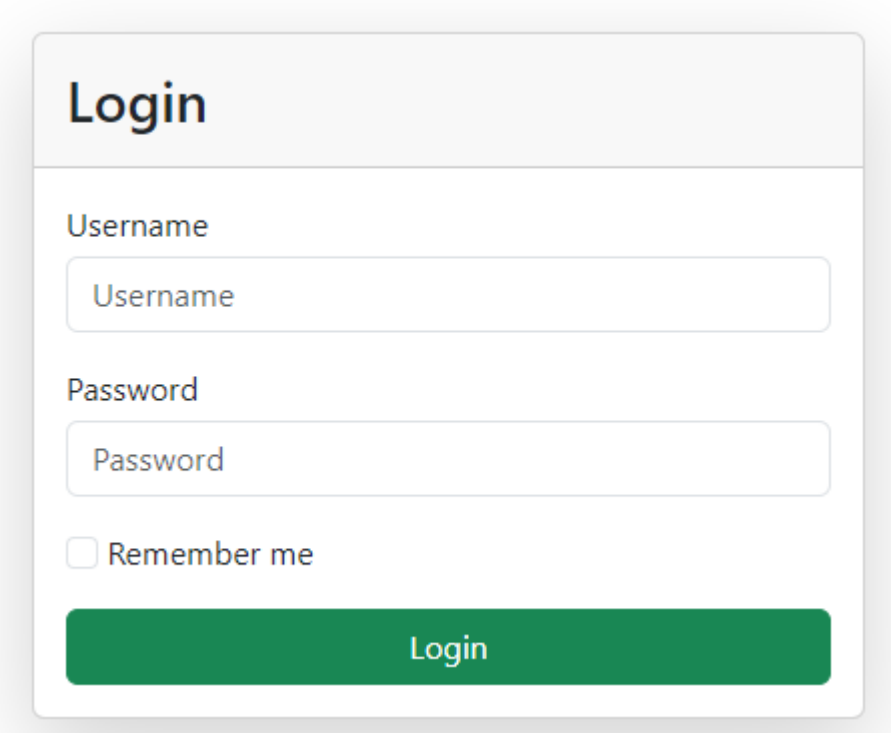
- Case of authentication with an invalid account.

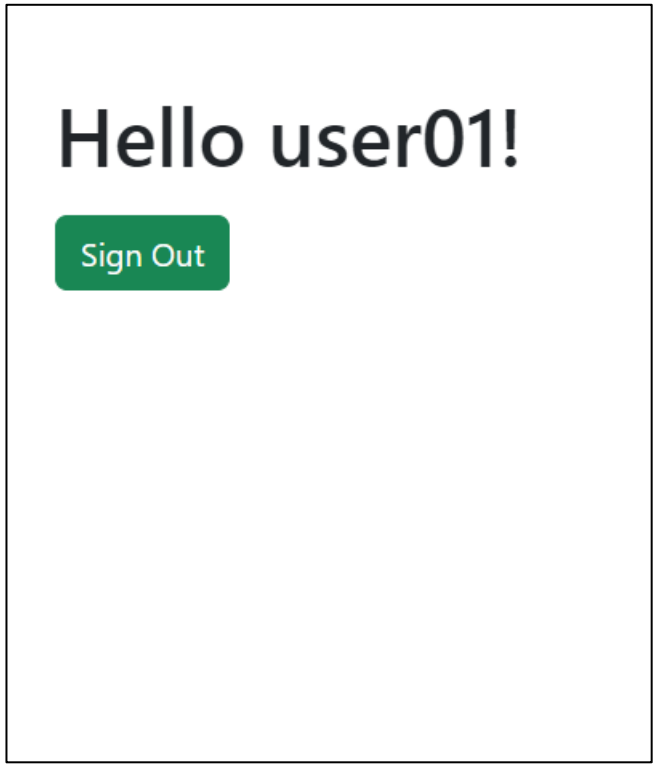




Authentication Demo

- Case of authentication with a valid account.

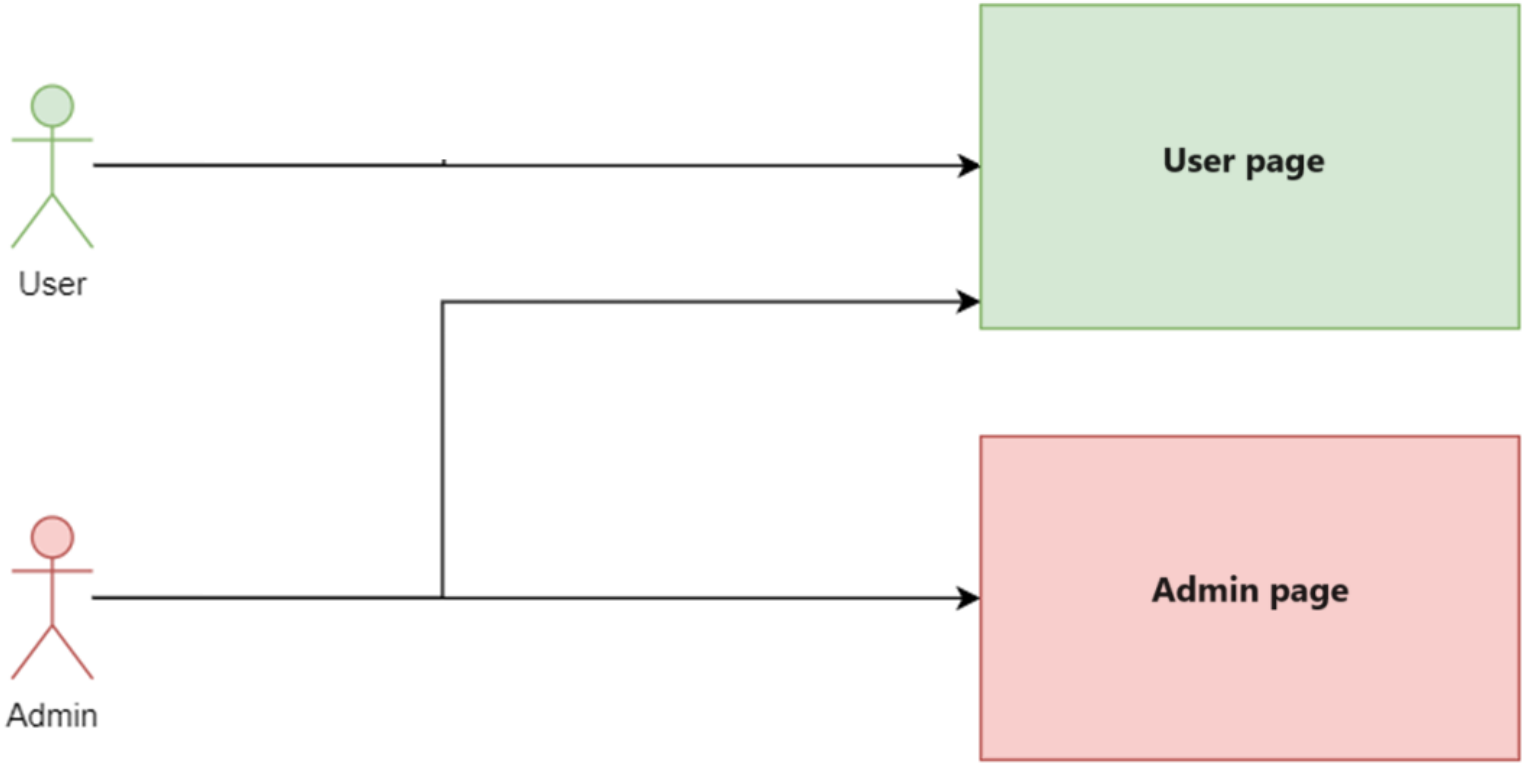




Section 4

Authorization

Authorization overview



How authorities stored in Spring Security

- Authorities/Roles information is stored inside `GrantedAuthority`.
- `SimpleGrantedAuthority` is the default implementation.

```
public interface GrantedAuthority
                    extends Serializable {
   String getAuthority();
}
```

```
public final class SimpleGrantedAuthority implements
GrantedAuthority {

   private static final long serialVersionUID = 610L;
   private final String role;
   public String getAuthority() {
      return this.role;
   }
   // .... Others implementation

}
```

How authorities stored in Spring Security

- Roles are stored with `GrantedAuthority` as strings with the "ROLE_" prefix.
- A logged-in account can have more than one role.

```
@Component("userDetailsService")
public class SecurityUserDetailsService implements UserDetailsService {
   // .... Others implementation

   private User createSpringSecurityUser(String username, Account account) {
      if (!account.isActivated()) {
         throw new AccountNotActivatedException("Account: " + username + " was not activated");
      }
      List<GrantedAuthority> grantedAuthorities = account.getAuthorities().stream()
         .map(authority -> "ROLE_" + authority.getName()) // ROLE_ADMIN or ROLE_USER
         .map(SimpleGrantedAuthority::new).collect(Collectors.toList());
      return new User(account.getUsername(), account.getPassword(), grantedAuthorities);
   }
}
```

Authorize HTTP Requests

- Spring Security allows you to model your authorization at the request level.
- Controlling who can access specific resources or endpoints based on user roles, permissions.
- Using methods like `hasRole()`, `hasAnyRole()` to implement authorization HTTP Request

Authorization with hasRole()

- In Spring Security, the `hasRole()` method is a powerful tool for authorization.
- It allows you to restrict access to specific parts of your application based on user roles.
- By using `hasRole()`, you can specify which roles are allowed to access certain endpoints or resources.

Authorization with hasRole()

- For example, protected page with the `"/admin"` prefix, only users with the "ADMIN" role can access them.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http
        // .... Others implementation
        .authorizeHttpRequests(request -> request
                .requestMatchers("/login").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
         )
        // .... Others implementation
   return http.build();
}
```

Authorization with hasAnyRole()

- In Spring Security, the `hasAnyRole()` method is a powerful tool for authorization.
- It allows you to restrict access to specific parts of your application based on user roles.
- By using `hasAnyRole()`, you can specify **a list** of roles that are allowed to access certain endpoints or resources.

Authorization with hasAnyRole()

- For example, protected page with the `"/user"` prefix, only users with the "ADMIN" or "USER" role can access them.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http
        // .... Others implementation
        .authorizeHttpRequests(request -> request
                .requestMatchers("/login").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/user/**").hasAnyRole("ADMIN", "USER")
                .anyRequest().authenticated()
         )
        // .... Others implementation
   return http.build();
}
```

Authorize - Method Security

- Spring Security supports modeling at the method level.
- Activate it in application by annotating any `@Configuration` class with `@EnableMethodSecurity`
- Using methods like `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, and `@PostFilter` to implement authorization method security

Authorize - Method Security

- `@PreAuthorize`: Specifies a pre-authorization expression that checks permissions before a method is invoked.
- `@PostAuthorize`: Defines a post-authorization expression to check permissions after a method has been executed.
- `@PreFilter`: Filters method input parameters based on a given expression before the method is called.
- `@PostFilter`: Filters the results of a method using a given expression after the method execution.

Authorization Demo

- We will perform testing with the matrix as follows:

Role	Page url start withs: /user/**	Page url start withs: /admin/**
USER	ALLOWED ACCESS	DENIED ACCESS
ADMIN	ALLOWED ACCESS	ALLOWED ACCESS

Section 5

Remember-Me Authentication

Remember-Me Authentication

- Remember-me or persistent-login authentication refers to web sites being able to remember the identity of a principal between sessions.
- Spring Security provides two concrete remember-me implementations:
  - ✓Uses hashing to preserve the security of cookie-based tokens
  - ✓Persistent storage mechanism to store the generated tokens.

***Note**: Note that both implementations require a* `UserDetailsService`

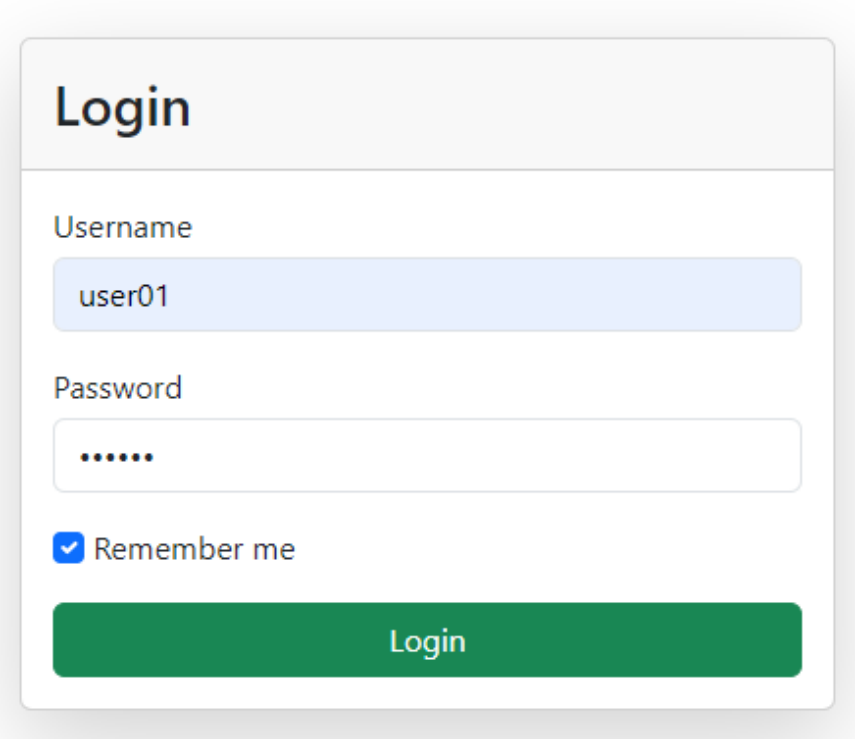
Remember-Me Authentication - Cookie-based

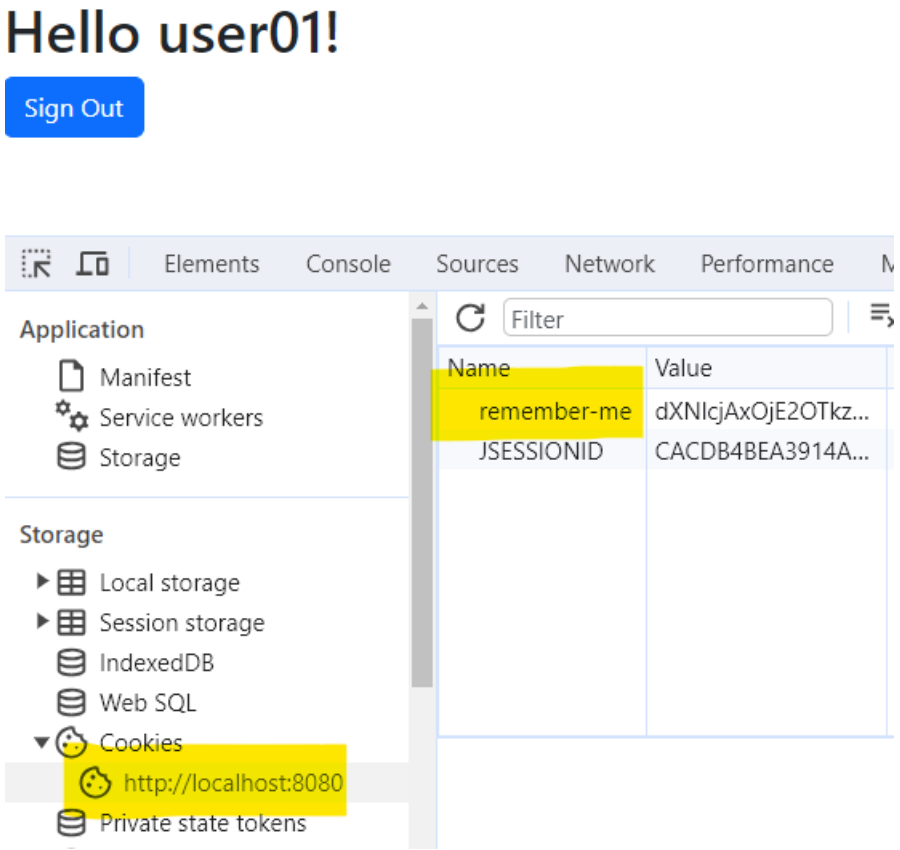
```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        ....
        .rememberMe(remember -> remember
                .tokenValiditySeconds(60 * 60 * 24) // 24h
                .key("your-remember-me-key")
                .rememberMeParameter("rememberMe")
        )
        ...
}
```

Explain :

- ✓ *This code begins the configuration of the "Remember Me" feature.*
- ✓ *Sets the token's validity time for 24 hours (60 seconds * 60 minutes * 24 hours).*
- ✓ *Defines a specific key for securing user information and tokens.*
- ✓ *Specifies the parameter name through which the user signals their intent to use the "Remember Me" feature. In this case, "rememberMe" is the parameter name.*

Remember-Me Authentication - Demo





Section 6

Handling Logouts

Handling Logouts

- In an application where end users can login, they should also be able to logout.
- By default, Spring Security stands up a `/logout` endpoint, so no additional code is necessary.
- If you request POST `/logout`, then it will perform the following:
  - ✓ Invalidate the HTTP session
  - ✓ Clear the `SecurityContextHolderStrategy` & `SecurityContextRepository`
  - ✓ Clean up any `RememberMe` authentication
  - ✓ Clear out any saved CSRF token
  - ✓ Fire a `LogoutSuccessEvent`

Customizing Logout URIs

- If you want to simply change the URI that Spring Security is matching, you can do so in the logout DSL in following way:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        ....
        .logout((logout) -> logout.logoutUrl("/my/logout/uri"))
       ...
}
```

Summary

- Introduction
- Architecture
- Username/Password Authentication
- Authorization
- Remember-Me Authentication
- Handling Logouts

THANK YOU!