Security & Cross-cutting Concerns

Infinite Diversity Arising from Unity

Definition: Crosscutting Concerns

- Term comes from Aspect Oriented Programming [AOP]

It involves:

- "…the modularization of concerns such as transaction management that cut across multiple types and objects. (**Such concerns are often termed *crosscutting* concerns in AOP literature.)"**

Cross-cutting Technologies

- **Servlet Filter**
  - Generic Servlet/web based filter
- **Interceptor**
  - Spring MVC Handler specific Interceptor
- **Spring AOP**
  - Simplified AOP implementation- Method level granularity
  - Only Spring recognized Beans
  - Employs a runtime integration [AKA weaving] process
- **AspectJ**
  - Fine grained supports method & field level AOP
  - Employs a specialized compilation weaving process
  - Works with non-Spring components

Filter

- Based on Servlet Specification
- Coupled with the Servlet API
- Access to HttpServletRequest and HttpServletResponse objects
- Intended for operating on request and response object parameters like HTTP headers, URIs and/or HTTP methods
- Generically applied - regardless of how the servlet is implemented

- **EXAMPLES**: Authentication , Logging, auditing, UTF-8 encoding

Handler Interceptor

- Part of Spring MVC Handler mapping mechanism
- Fine grained access to the handler/controller
  - preHandle() - before controller execution
  - postHandle() - after controller execution
    - Can expose additional model objects to the view via the given ModelAndView
  - afterCompletion() - after rendering the view. Allows for proper resource cleanup
- Interceptors can be applied to a group of handlers

Volunteer Interceptor

```
public class VolunteerInterceptor implements HandlerInterceptor{
  //the other two methods are not listed here

  @Override
  public boolean preHandle(HttpServletRequest request,
    HttpServletResponse arg1, Object arg2) throws Exception {
    Principal principal = request.getUserPrincipal();
    String userMessage = "Welcome to web security demo!";
    if(request.isUserInRole("ROLE_ADMIN")) {
      userMessage += " ROLE_ADMIN has extra 20% off!";
    }
    request.setAttribute("userMessage", userMessage);
    return true;
  }
}
```

Interceptor Configuration

- **AntPathMatcher**
- The mapping matches URLs using the following rules:
  - ? matches one character
  - * matches zero or more characters
  - ** matches zero or more 'directories' in a path
- Executed in order of declaration

```
<mvc:interceptors>
   <mvc:interceptor>
      <mvc:mapping path="/**"/>
      <bean class="mum.edu.interceptor.VolunteerInterceptor" />
   </mvc:interceptor>
</mvc:interceptors>
```

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
   registry.addInterceptor(volunteerInterceptor()).addPathPatterns("/market/**");
}
```

@ControllerAdvice

- Cross-cutting controller exception handling for application, not just to an individual controller.
- Like an Annotation driven interceptor.
- Three types of methods are supported:
  - Exception handling methods annotated with @ExceptionHandler.
  - Model enhancement methods (for adding additional data to the model) annotated with @ModelAttribute.
  - Binder initialization methods (used for configuring form-handling) annotated with @InitBinder.

@ControllerAdvice example

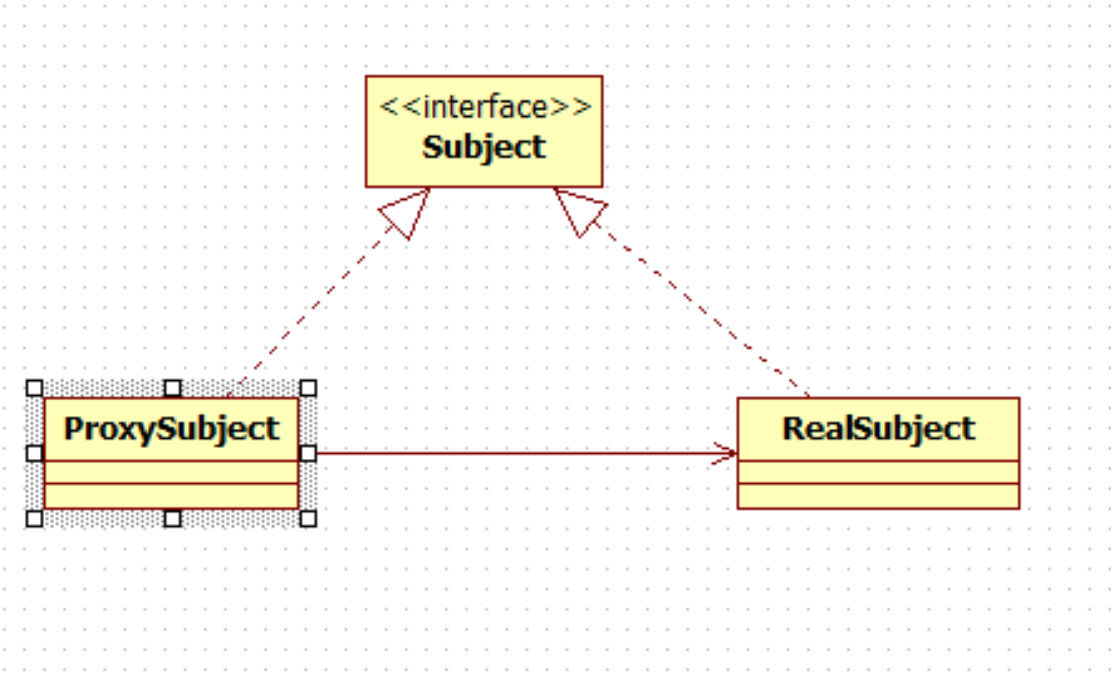
```
@ControllerAdvice
public class ControllerExceptionHandler {

  @ModelAttribute("testOrder")
  public String testOrder() {
    return "This is ADVICE ORDER!";
  }

  @ExceptionHandler(value = AccessDeniedException.class)
  public String accessDenied() {
    return "error-forbidden";
  }
}
```

AOP & ASPECTJ

- SpringAOP:
  1. Runtime weaving through proxy using the concept of a dynamic proxy
  2. Spring AOP supports only method level PointCut

- AspectJ:
  1. Compile time weaving if source available or post compilation weaving (using compiled files)
  2. AspectJ supports both method and field level Pointcuts

Spring AOP – Proxy Pattern



- **Subject** - Interface implemented by the RealSubject
- **Proxy** - Controls access to the RealSubject
- **RealSubject** - the real object that the proxy represents.

Main Point

- The different technologies [Filter, Interceptor, AOP] available in Spring, together provide a thorough solution to cross cutting concerns.
- *Creative intelligence enhances and strengthens uniquely differing values in life in a comprehensive way.*

What is Spring Security?

- Spring Security is a framework that focuses on providing both **authentication** and **authorization** (or "access-control") to Java web application and SOAP/RESTful web services
- Spring Security currently supports integration with all of the following technologies:
  - HTTP basic access authentication
  - LDAP system
  - SSO
  - …..
  - Your own authentication systems
- It is built on top of Spring Framework

Spring Security Fundamentals I

- Authentication
  - Confirming truth of credentials
  - Who are you?
- Authorization
  - Define access policy for principal
  - What can you do?
- Principal
  - User that performs the action
  - Currently logged in User
- GrantedAuthority
  - Application permission granted to a principal
- Roles
  - coarse-grained permission

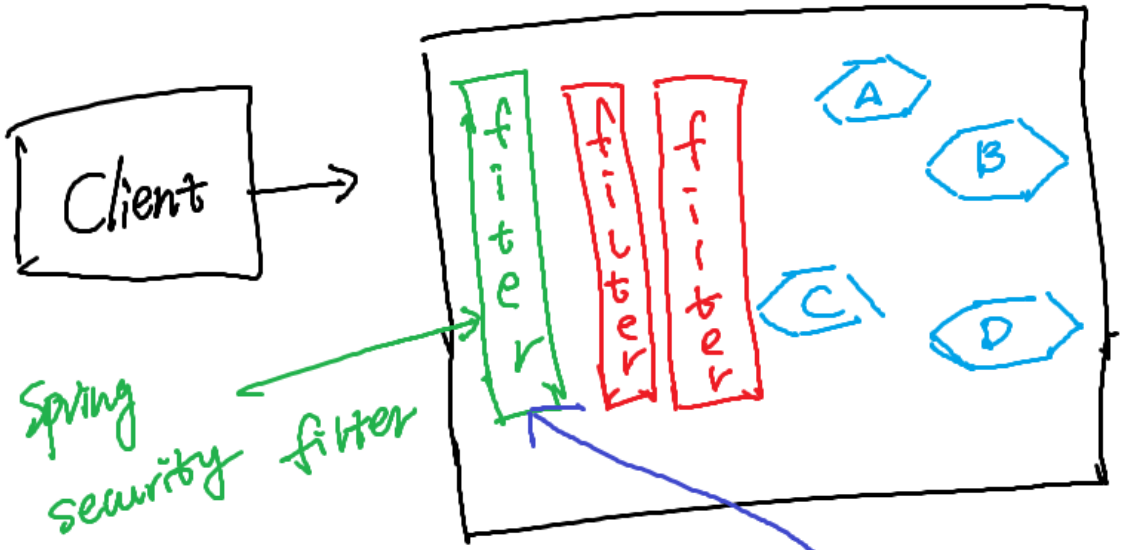
Spring Security Fundamentals I

- Authentication
  - Confirming truth of credentials
  - Who are you?
- Authorization
  - Define access policy for principal
  - What can you do?
- Principal
  - User that performs the action
  - Currently logged in User
- GrantedAuthority
  - Application permission granted to a principal
- Roles
  - coarse-grained permission

Spring Security Fundamentals II

- SecurityContext
  - Hold the authentication and other security information
- SecurityContextHolder
  - Provides access to SecurityContext
- AuthenticationManager
  - Controller in the authentication process
- AuthenticationProvider
  - Interface that maps to a data store which stores your user data.
- Authentication Object
  - Object is created upon authentication, which holds the login credentials.
- UserDetails
  - Data object which contains the user credentials, but also the Roles of the user.
- UserDetailsService
  - Collects the user credentials, authorities(roles) and build an UserDetails object.

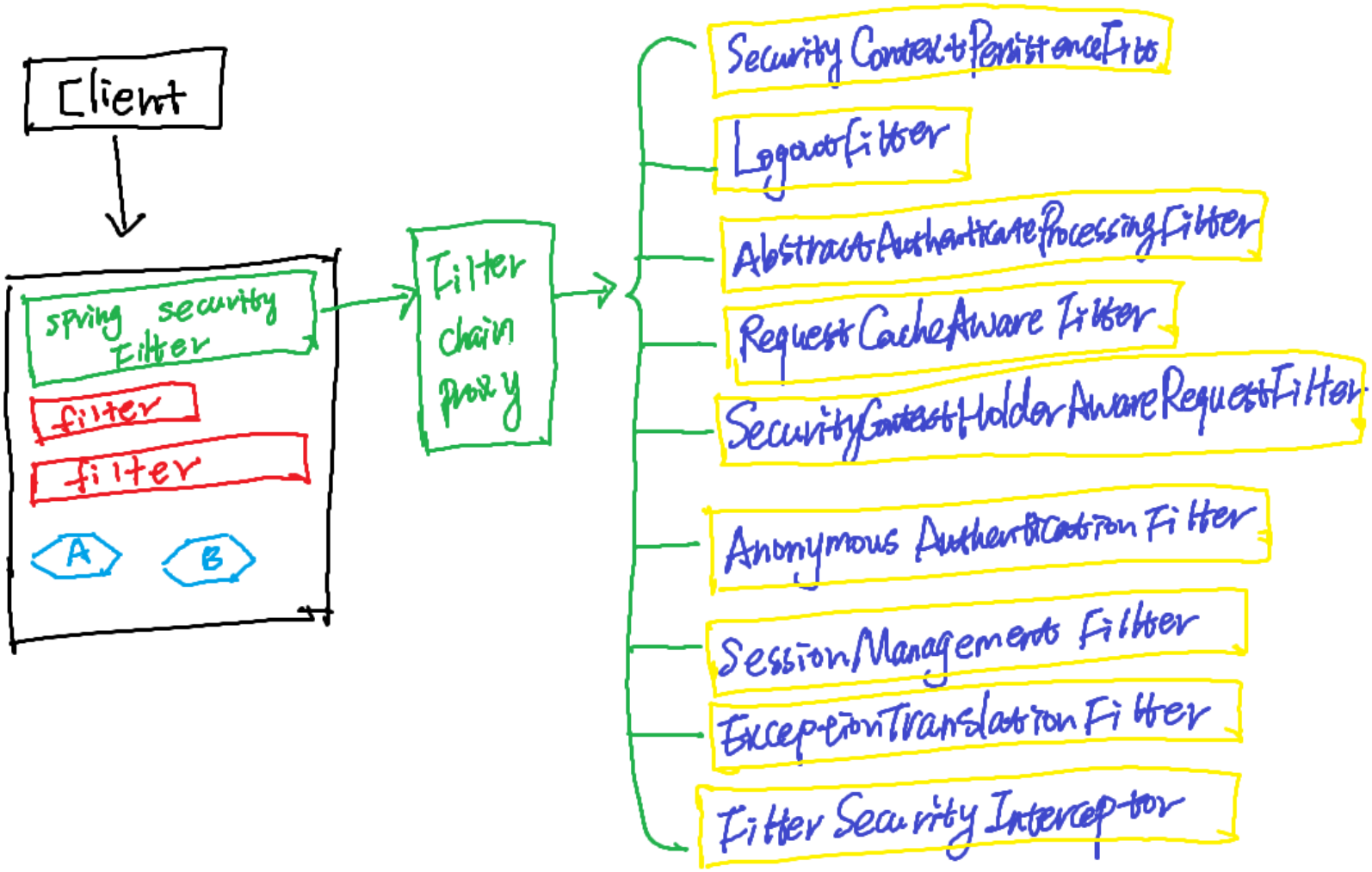
Spring Web Application Security Servlet **Filter** based




```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

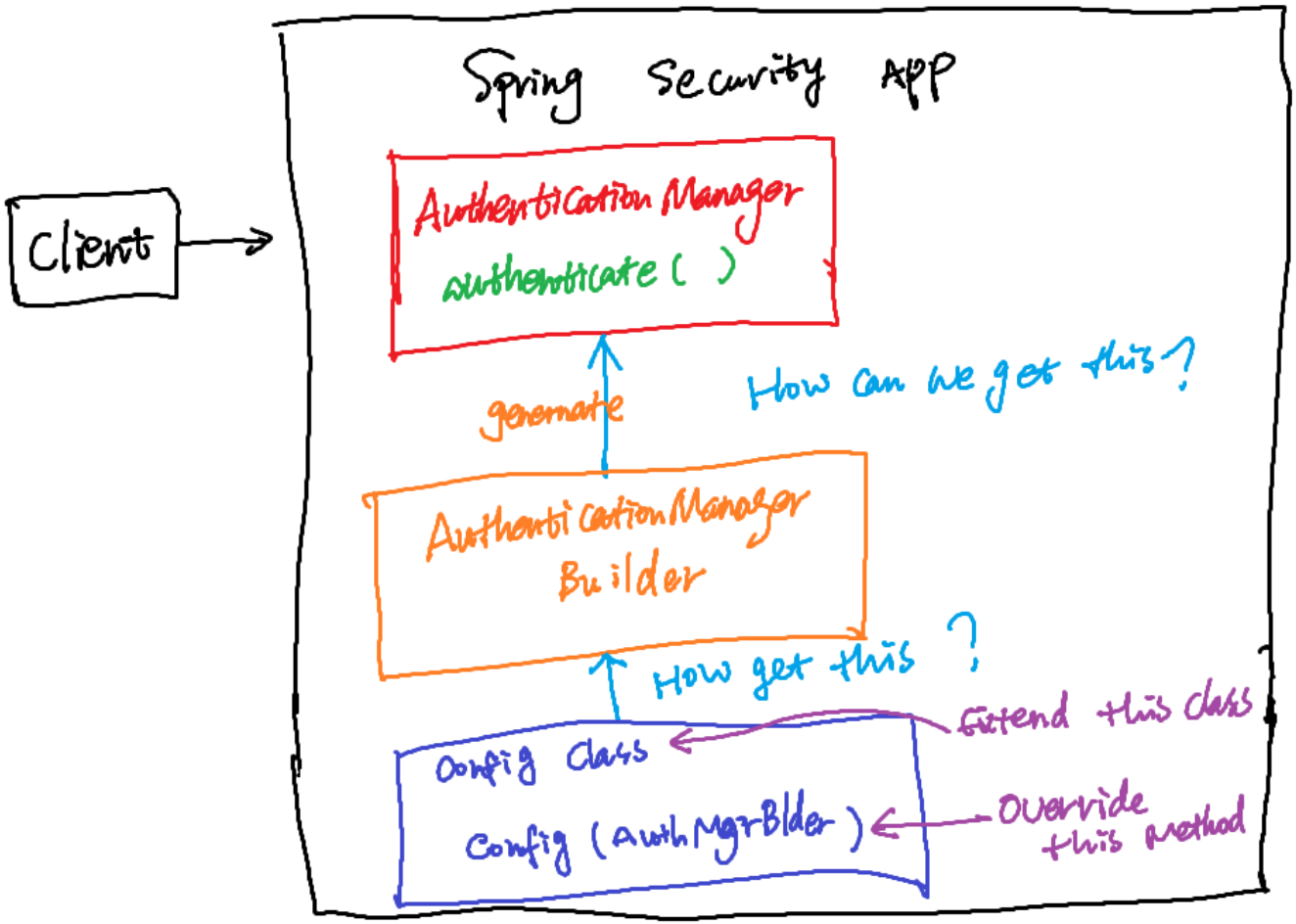
```
public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer {
}
```

Filter Chain



AuthenticationManager
AuthenticationManagerBuilder

```
public class SpringSecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

    }
}
```



Authorization Intro

```
public class SpringSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {

    }
}
```

Authentication

Authorization

Config based on chaining

http.authorizeRequests()

- <path> → <allowed-roles>
- ⋮
- <path> → <allowed-roles>

specify <path> with permission <allowed-roles>

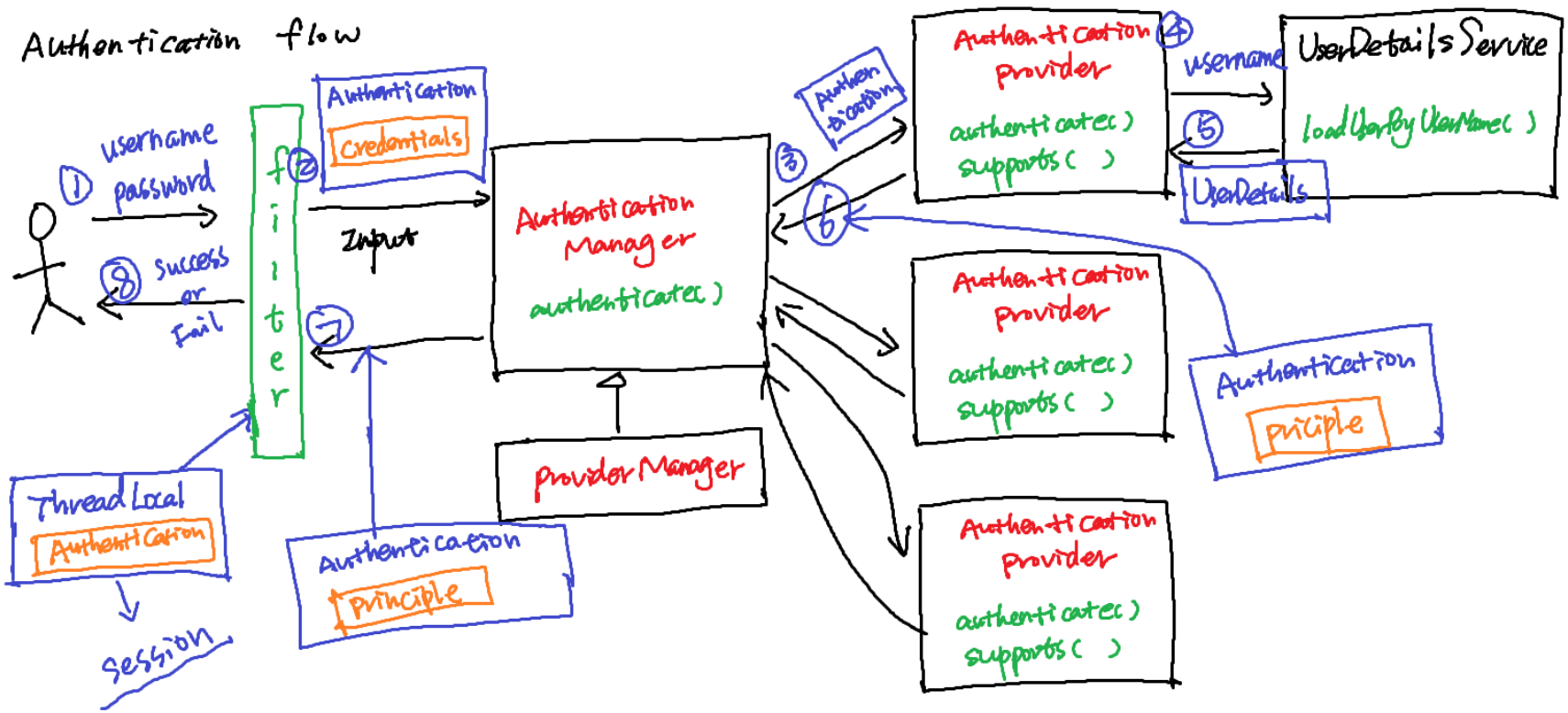
the most restrict

order of the config

the least restrict

Demo: spring-boot-security

Spring Security Architecture



Authentication Provider
InMemory authentication

```
@EnableWebSecurity
public class SpringSecurityConfiguration extends WebSecurityConfigurerAdapter {


    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("user")
                .password("user")
                .roles("USER")
                .and()
                .withUser("admin")
                .password("admin")
                .roles("ADMIN");
    }
}
```

Demo: spring-boot-security

Authentication Provider
JDBC authentication

```
public class SpringSecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Autowired
    DataSource dataSource;

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {

        auth.jdbcAuthentication()
                .dataSource(dataSource)
                .usersByUsernameQuery("select username,
password, enabled from users where username = ?")
                .authoritiesByUsernameQuery("select username,
authority from authorities where username = ?");

    }
}
```

Demo: spring-security-jdbc

Authentication Provider
JPA authentication I

```
@EnableWebSecurity
public class SpringSecurityConfiguration extends WebSecurityConfigurerAdapter {


    @Qualifier("JPAUserDetailService")
    @Autowired
    UserDetailsService userDetailService;

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

```
@Service
public class JPAUserDetailService implements UserDetailsService {

    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        Optional<User> user = userRepository.findByUsername(username);
        user.orElseThrow(() -> new UsernameNotFoundException("Not FOUND..."));
        return new JPAUserDetails(user.get());
    }
}
```

Authentication Provider
JPA authentication II

```
public class JPAUserDetails implements UserDetails {

    private String username;
    private String password;
    private boolean isActive;
    private Set<Role> roles;

    public JPAUserDetails(User user) {
        username = user.getUsername();
        password = user.getPassword();
        isActive = user.getActive() == 1 ? true : false;
        roles = user.getRoles();
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return roles.stream().map(role -> new SimpleGrantedAuthority(role.getRole()))
                .collect(Collectors.toList());

    }

    @Override
    public String getPassword() {
        return password;
    }

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

    @Override
    public boolean isEnabled() {
        return isActive;
    }
...
}
```

Demo: spring-security-jpa

PasswordEncoder

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

```
public void save(UserCredentials credentials) {

  String encodedPassword =
  passwordEncoder.encode(credentials.getPassword());
  credentials.setPassword(encodedPassword);
  userCredentialsRepository.save(credentials);
}
```

Custom Login, Logout & Access Denied Page

```
@Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .formLogin()
                    .loginPage("/login")
//                    .defaultSuccessUrl("/")
                    .failureUrl("/login?error=true")
                    .usernameParameter("username")
                    .passwordParameter("password")
                    .permitAll()
                .and()
                .logout()
                    .logoutUrl("/perform_logout") //change default /logout url to /perform_logout
                    .logoutSuccessUrl("/login?logout=true")
                    .invalidateHttpSession(true)
                    .clearAuthentication(true)
                    .permitAll()
                .and()
                .exceptionHandling()
                    .accessDeniedPage("/denied");
    }
```

Demo: spring-boot-login-logout

Authorization

- Web request authorization using interceptors.
- Method authorization using AspectJ or Spring AOP
- **Common usage pattern**
  - is to perform **some** web request authorization
  - coupled with Spring AOP method authorization on the services layer [**more secure**].

Authorization

- URL based Authorization
  - Patterns are always evaluated in the order they are defined

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers("/admin").hasRole("ADMIN")
            .antMatchers("/user").hasRole("USER")
            .antMatchers("/", "/h2-console/**").permitAll();
}
```

- Method Level Authorization

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
public class SpringSecurityConfiguration extends WebSecurityConfigurerAdapter {
}
```

➢ **MemberServiceImpl.java**

```
@Secured("ROLE_ADMIN")
@PreAuthorize("hasRole('ROLE_ADMIN')")
public void save(Member member) {
        memberRepository.save(member);
}
```

Demo: spring-boot-login-logout

@Secured({"ROLE_VIEWER","ROLE_EDITOR"}) == @PreAuthorize("hasRole('ROLE_VIEWER') or hasRole('ROLE_EDITOR')")

Spring Security Tag Library

- Basic support for security information and constraints in thymeleaf

```
<html xmlns:th="https://www.thymeleaf.org" xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity5">
```

- Authorize tag

```
<div sec:authorize="isAuthenticated()">
  <div>
    <form action="#" th:action="@{/logout}" method="post">
      <input type="submit" value="Logout" />
    </form>
  </div>
</div>
```

```
<div sec:authorize="hasRole('ROLE_ADMIN')">
    This content is only shown to administrators.
</div>
<div sec:authorize="hasRole('ROLE_USER')">
    This content is only shown to users.
</div>
```

- Authentication tag: renders the name of the current user

```
Logged in user: <span sec:authentication="name"></span> |
    Roles: <span sec:authentication="principal.authorities"></span>
```

Demo: spring-boot-login-logout

Cross Site Request Forgery (CSRF)

- Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.
- Malicious exploit of a website where unauthorized commands are transmitted from a user that the website trusts
- **"Classic" POST vulnerability**
  - visit a "bad" site while still logged into a "trusted" site…
  - Access to Trusted site can be "spoofed".
- **Recommendation:**
  - Use CSRF protection on any request that could be processed by a browser by normal users.

- **Automatically included when using or when you use thymeleaf:**

```
<form:form>
```

- If NOT using form:form, use security tag:

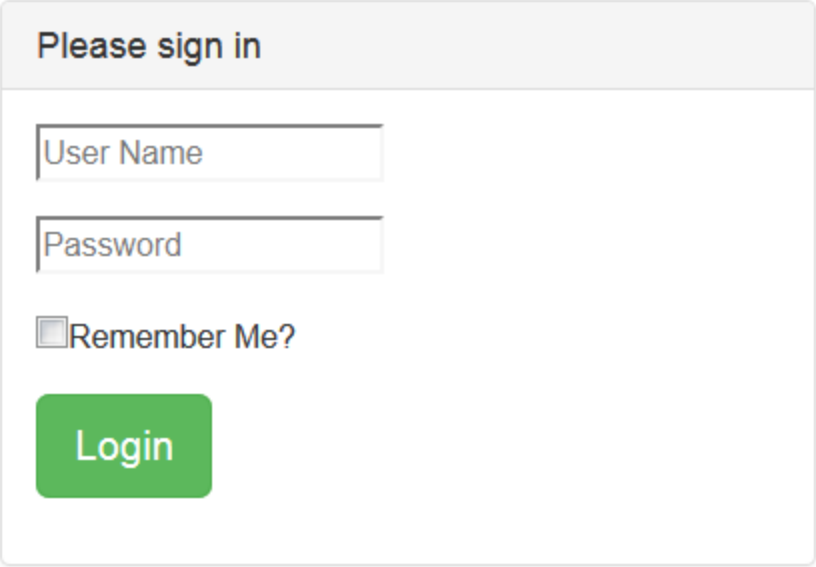
```
<input type="hidden" name="${_csrf.parameterName}"
   value="${_csrf.token}" />
```

```
<input
  type="hidden"
  th:name="${_csrf.parameterName}"
  th:value="${_csrf.token}" />
```

OWASP - CSRF Spring - CSRF

Remember Me

- **AKA persistent-login authentication**
- Able to remember the identity of a principal between sessions.
- Based on a permanent cookie [default expiration of 2 weeks.]




```
<input type="checkbox" name="keepMe"/>Remember Me?<br/>
```

Remember Me Configuration

- Simple Hash-Based Token Approach
  - It uses hashing to preserve the security of cookie-based tokens.
  - This approach has security issue and is commonly not recommended.
  - stores hashed user password in "remember me" cookie – easy to hack.

- Persistent Token Approach
  - Uses database to store the generated tokens
  - Uses combination of randomly generated series and token are persisted, making a brute force attack very unlikely.
  - Requires table persistent_logins in database

- token-validity defaults to 14 days

```
<security:remember-me data-source-ref="dataSource" token-
 validity-seconds="86400" remember-me-parameter="keepMe"/>
```

Demo: springsecurity

Main Point

- Authentication & Authorization underlie the entire web application. They provide a shield that makes the application invulnerable.
- *Transcendental consciousness is characterized by the quality of invincibility, which means one cannot be overcome or overpowered*