Lesson 13
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 run time integration [AKA weaving] process

AspectJ

Fine grained supports method & field level AOP

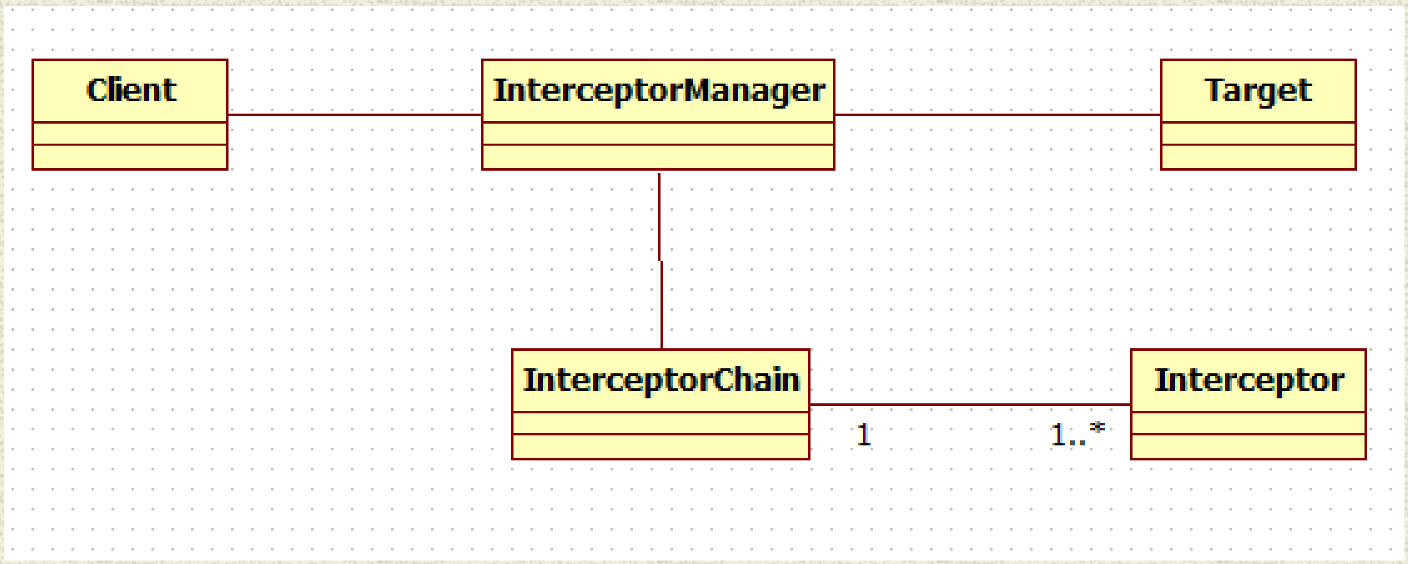
Employs a specialized compilation weaving process

Works with non-Spring components

Interceptor Chain

Core J2EE Patterns - Intercepting Filter

Preprocessing and post-processing of a client Web request and response are required



FILTER SERVLET

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 void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler,ModelAndView
modelAndView) throws Exception {
String userMessage= "Become a Community Member- Join the Team!";

Principal principal = request.getUserPrincipal();

if (principal != null) {
   if (request.isUserInRole("ROLE_ADMIN") )
userMessage= "There is ALWAYS Free cookies at www.freebies.com";
else
   userMessage = "We have Many NEW and exciting Volunteer
opportunities!!!";
 }
```

Interceptor Configuration

If there are multiple interceptors configured, *preHandle()* method is executed in the order of configuration whereas *postHandle()* and *afterCompletion()* methods are invoked in the reverse order.

- **AntPathMatcher**
- The mapping matches

  ? matches one cl

  * 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>
```

@ControllerAdvice

Cross-cutting concern for entire application, not just to an individual controller.

Annotation driven interceptors.

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.

AOP & ASPECTJ

SpringAOP:

1)Runtime weaving through proxy using the concept of a dynamic proxy

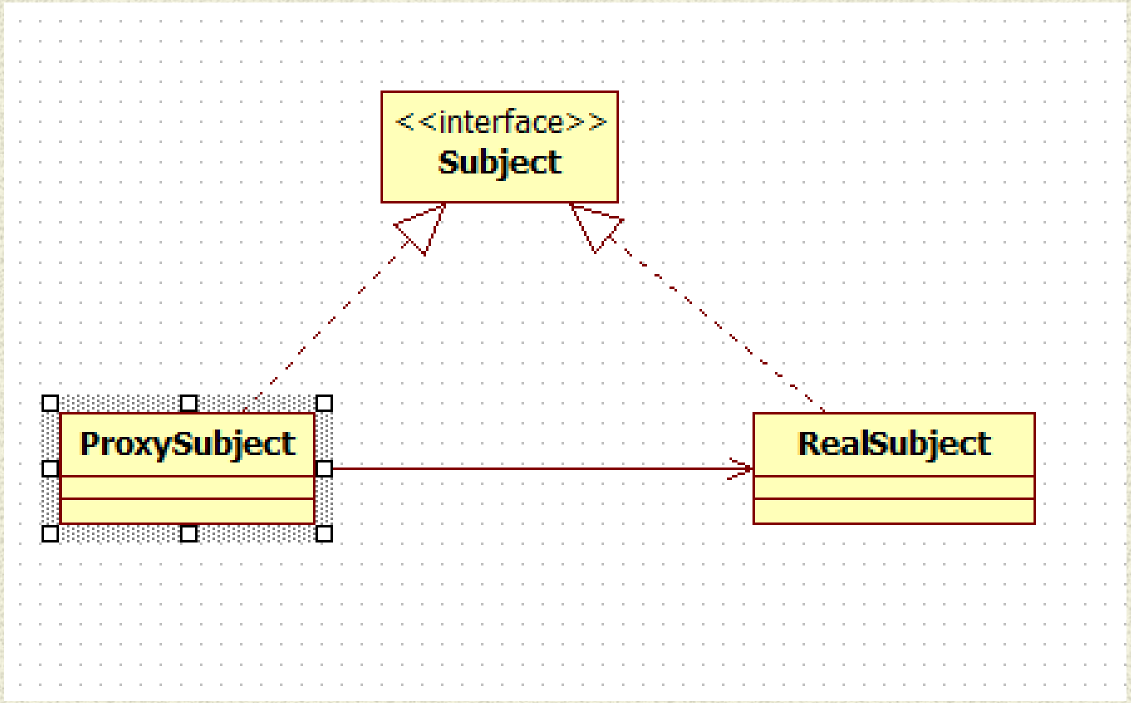
2)Spring AOP supports only method level PointCut

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.*
-

Authentication Authorization

Authentication refers to unique identifying information from each system user, generally in the form of a username and password.

Authorization refers to the process of allowing or denying individual user access to resources.

Basic and Digest Authentication

Basic authentication

- Handshake based on HTTP headers
- Transmits username/password as “plain text”
  - Base64 encoding
- Used in conjunction with SSL-HTTPS
- Used with form-based authentication
- Secure data at rest

Base64

Digest Authentication

- Transmits encrypted username/password
- “Double” handshake to get hash “seed”
- More complex – more vulnerable

Spring Security Tag Library

Basic support for security information and constraints in JSPs

Basically 3 tags

- **authorize tag**

```
<security:authorize access="isAuthenticated()">
```

- **authentication tag**

```
<security:authentication property="principal.username" />
```

renders the name of the current user.

- **accesscontrollist tag**

used with Spring Security's ACL module

```
<security:accesscontrollist hasPermission="admin,designer"
        domainObject="${someObject}">
    Display if user has either permission for someObject.
</sec:accesscontrollist>
```

Spring Security JSP Tag Library example

```
<%@ taglib prefix="security"
           uri="http://www.springframework.org/security/tags"%>
```

Welcome.jsp

```
 <security:authorize access="isAuthenticated()">
  Welcome  <security:authentication property="principal.username" />
</security:authorize>

<security:authorize access="isAnonymous()">
    <a href="<spring:url value='/login' />" > Login</a>
</security:authorize>
```

Spring Web Application Security Servlet Filter based

Spring Security's web infrastructure is based entirely on standard servlet filters.

Agnostic to specific web technology.

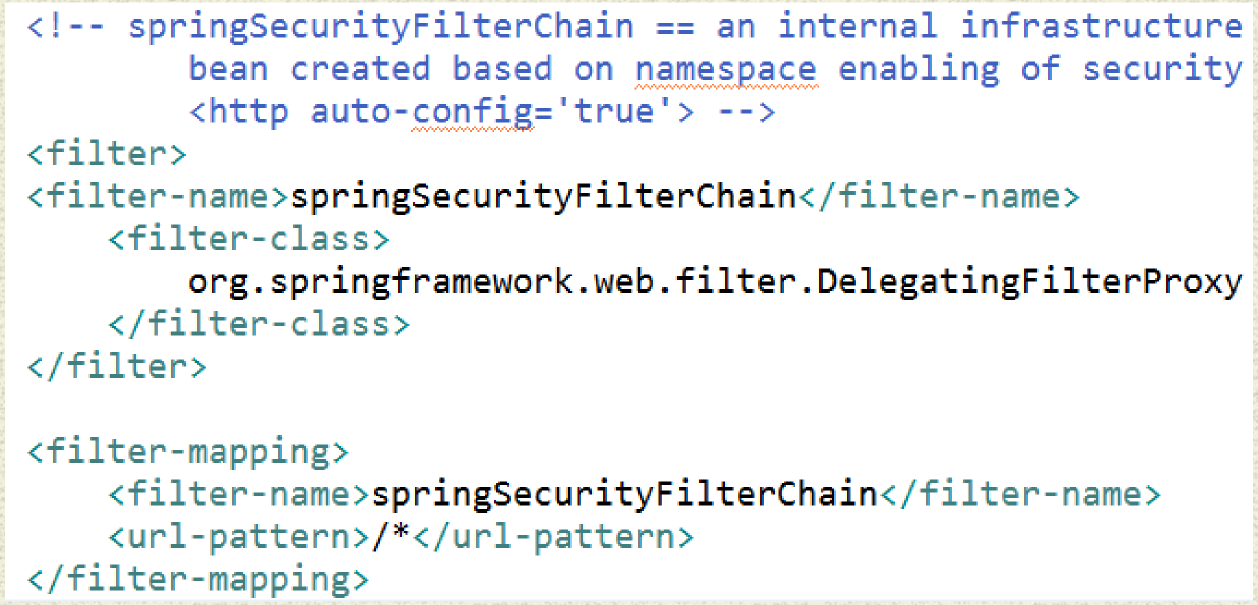
Based on HttpServletRequests and HttpServletResponses

Usage:

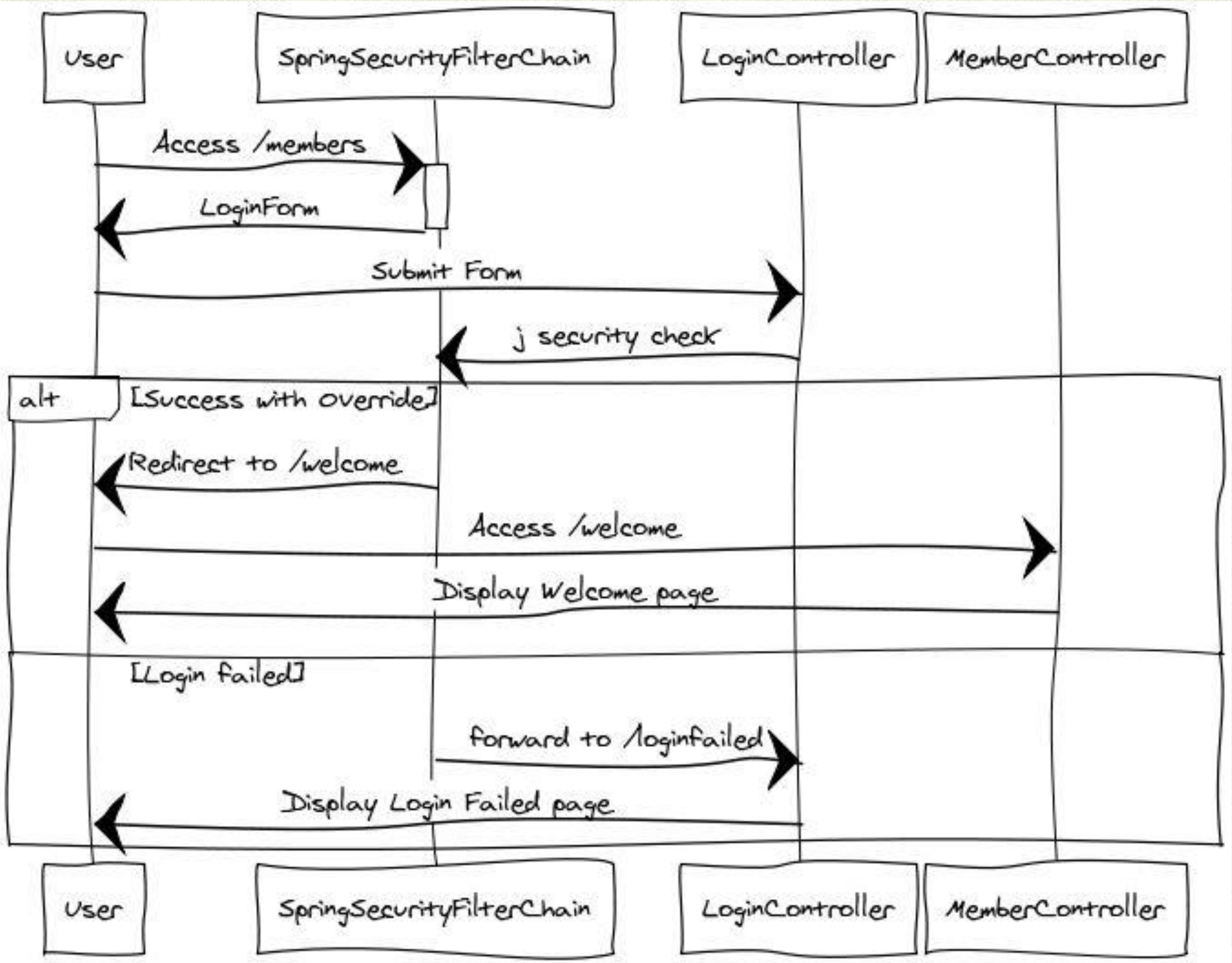
Browser

Web service client

AJAX application.



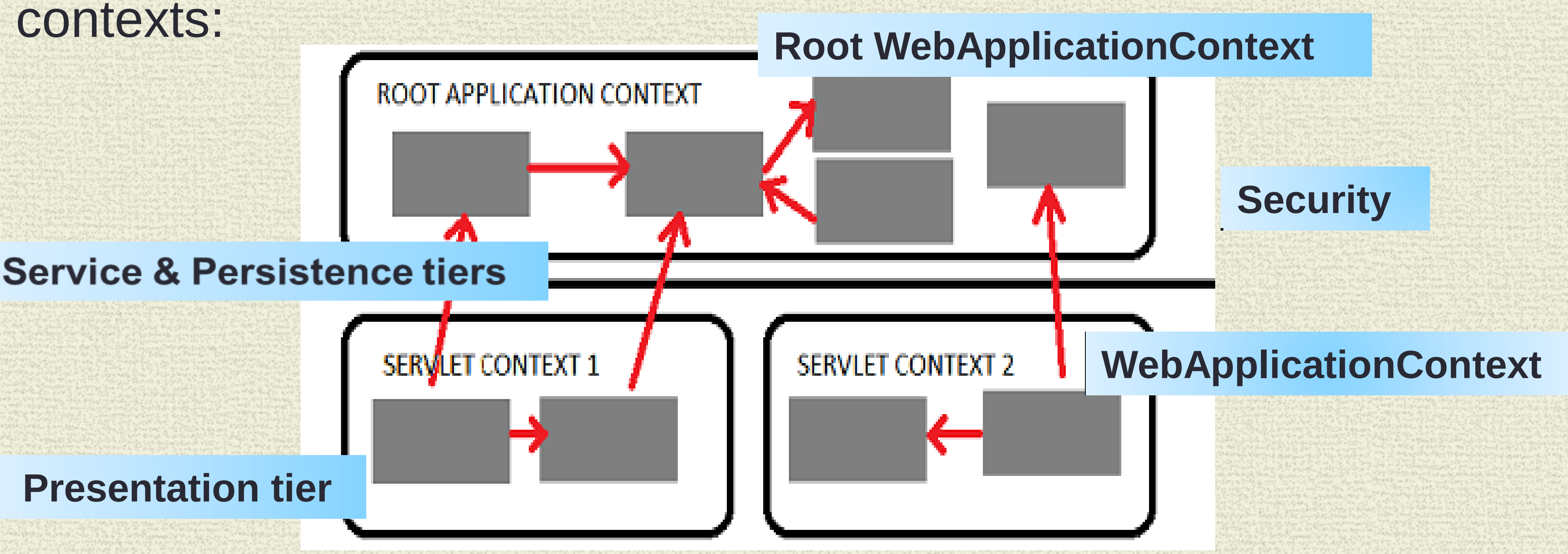
```
<!-- springSecurityFilterChain == an internal infrastructure
        bean created based on namespace enabling of security
        <http auto-config='true'> -->
<filter>
<filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```


User
SpringSecurityFilterChain
LoginController
MemberController
Access /members
LoginForm
Submit Form
j security check
alt
[Success with override]
Redirect to /welcome
Access /welcome
Display Welcome page
[Login failed]
forward to /loginfailed
Display Login Failed page
User
SpringSecurityFilterChain
LoginController
MemberController

Web Application Context

Spring has multilevel application context hierarchies.

Web apps by default have two hierarchy levels, root and servlet contexts:



- **Presentation tier has a WebApplicationContext [Servlet Context] which inherits all the resources already defined in the root WebApplicationContext [Services, Persistence]**

Spring Security web.xml

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                /WEB-INF/spring/context/applicationContext.xml
                /WEB-INF/spring/context/security-context.xml
        </param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<filter>
<filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
                org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
```

The security-context is loaded into the “root” WebApplicationContext as it is NOT Spring MVC specific [DispatcherServlet]

springSecurityFilterChain is an internal infrastructure bean created based on namespace enabling of security <http auto-config='true'>

Minimal [XML] configuration

Requires all users to be authenticated

Allows users to authenticate with form based login

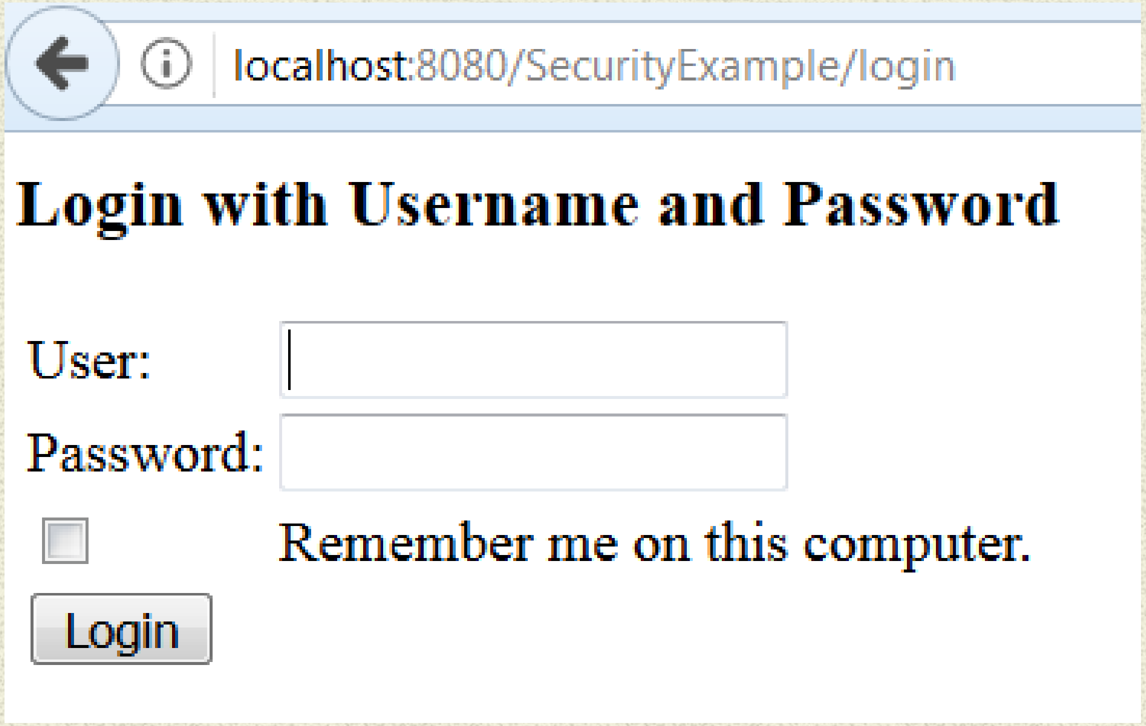
Allows users to authenticate with HTTP Basic authentication

```
<security:http use-expressions= "true“ >
   <intercept-url pattern="/**" access= “isFullyAuthenticated()“ />
    <form-login />
</security:http>
```

Default Form Based Login

<form-login /> generates a default login form

Available at URL: login


localhost:8080/SecurityExample/login

Login with Username and Password

User:

Password:

Remember me on this computer.

Overridden if attributes are set on <form-login />

Custom Login Configuration

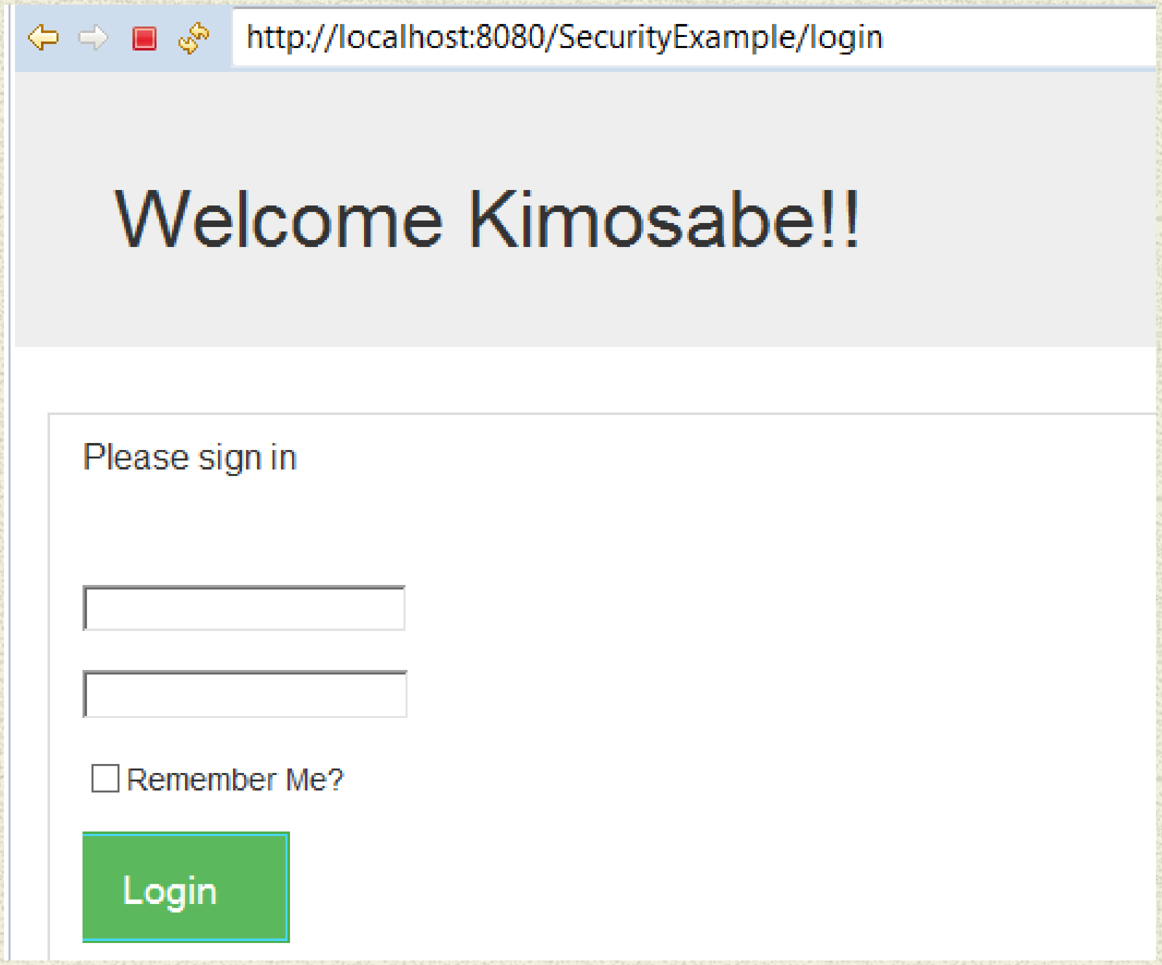
Security Version 4

```
 <security:form-login
  login-page="/login"
    default-target-url="/welcome"
                 always-use-default-target="true"
                 authentication-failure-url="/loginfailed"/>

         <security:logout logout-success-url="/logout"
                                     delete-cookies="JSESSIONID" />
```

```
<security:form-login login-page="/login"
        login-processing-url="/postLogin“
        username-parameter="username"
        password-parameter="password"
        default-target-url="/welcome"
       always-use-default-target="true"
       authentication-failure-url="/loginfailed"/>
     <security:logout logout-success-url="/logout"  logout-url= "/doLogout"/>
</security:http>
```

Default values in Version 4

Customized Login



http://localhost:8080/SecurityExample/login

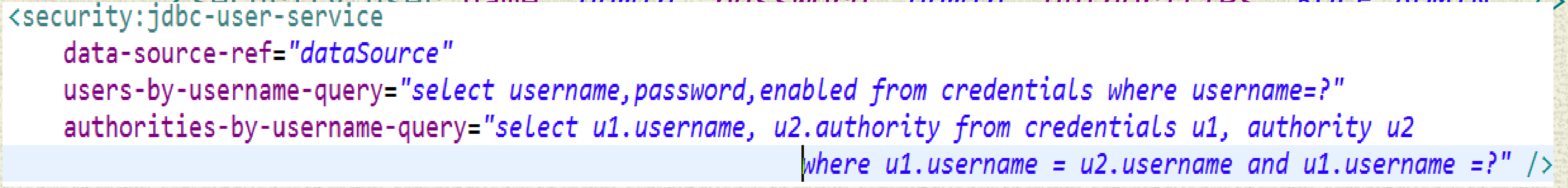
Welcome Kimosabe!!

Please sign in

☐ Remember Me?

security-context.xml
Configure User Credentials

```
<security:authentication-manager>
 <security:authentication-provider>
 <security:password-encoder ref="passwordEncoder" />
  <security:user-service>            *****
   <security:user name="admin" password="admin" authorities="ROLE_ADMIN" />
```



```
<security:jdbc-user-service
    data-source-ref="dataSource"
    users-by-username-query="select username,password,enabled from credentials where username=?"
    authorities-by-username-query="select u1.username, u2.authority from credentials u1, authority u2
                                   where u1.username = u2.username and u1.username =?" />
```

```
 </security:authentication-provider>
</security:authentication-manager>
```

***** replace with

`<jdbc-user-service ... />`

to use DBMS

NOTE: Database tables reflect Acegi Security model [see Demo]

Data at Rest

```
<security:password-encoder ref="passwordEncoder" />

public void save(Credentials credentials) {

      String encodedPassword =
            passwordEncoder.encode(credentials.getPassword());
      credentials.setPassword(encodedPassword);

      credentialsRepository.save(credentials);
}
```

security-context.xml Authorization

Enable Method level authorization. If here -APPLICATION Level scanned components. For WEB level - need to place in Dispatcher-servlet

```
<security:global-method-security pre-post-annotations="enabled"/>
```

security:http enables security filter mechanism.

name space configuration is activated

use-expressions enables SPEL syntax for URL level authorization

```
<security:http use-expressions="true">
    <security:intercept-url pattern="/members"
                    access="hasAnyRole('ROLE_ADMIN','ROLE_USER')" />
    <security:intercept-url pattern="/**"
                    access="permitAll" requires-channel="https"/>
```

HTTP – HTTPS Switching

All access will cause – a switch will occur to HTTPS

Manage SSL switching

```
<security:intercept-url pattern="/**"
                access="permitAll requires-channel="https"/>
```

DISCLAIMER

To be truly secure, an application should not use HTTP at all or even switch between HTTP and HTTPS. It should start in HTTPS (with the user entering an HTTPS URL) and use a secure connection throughout to avoid any possibility of man-in-the-middle attacks

security-context.xml [Optional]

```
Spring Assumes HTTP: Port 80 [8080] ... HTTPS: Port 443 [8443]
If otherwise need to configure the Ports:

<security:http use-expressions="true">

   <security:port-mappings>
      <security:port-mapping http=“9080" https=“9443"/>
 </security:port-mappings>
…

….
</security:http>
```

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**].

URL based Authorization

Patterns are always evaluated in the order they are defined

Configuration:

```
<security:intercept-url pattern="/members/add"
                 access= "hasRole('ROLE_ADMIN')" />
```

Method level Authorization

Configuration:

```
<security:global-method-security
                 pre-post-annotations="enabled"/>
```

MemberServiceImpl.java

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

Authorization failed Exception @ControllerAdvice example

```
@ControllerAdvice
public class ControllerExceptionHandler {

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

Cross Site Request Forgery (CSRF)

OWASP - CSRF

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: `<form:form>`

If NOT using form:form, use security tag:

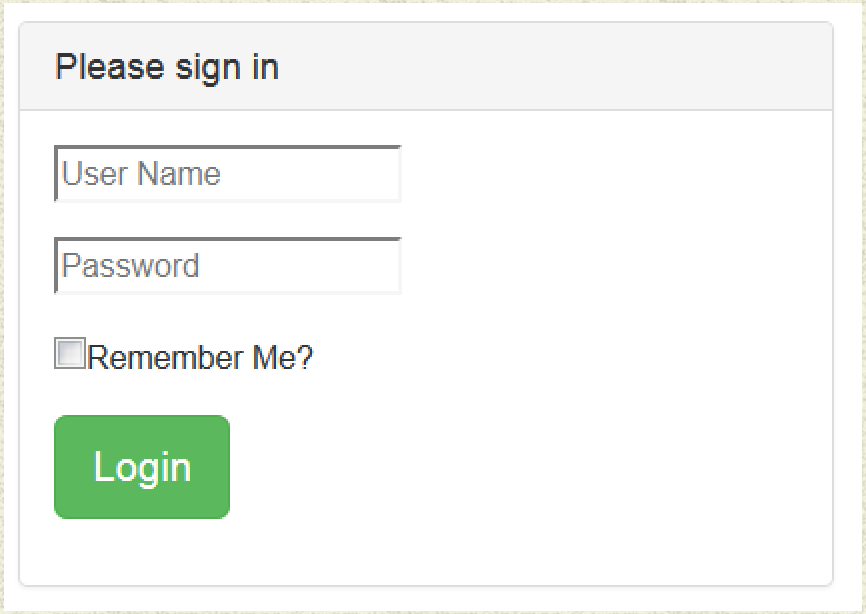
```
<security:csrfInput />
```

Spring - CSRF

Remember Me

AKA persistent-login authentication

Able to remember the identity of a principal between sessions.


Please sign in

User Name

Password

☐ Remember Me?

```
<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

    Combination of randomly generated series and token are persisted, making a brute force attack very unlikely.
  - Requires table persistent_logins in database

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

- `token-validity defaults to 14 days.`

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*