

# COMP S380F Lecture 7: Filter, Spring Security

---

Dr. Keith Lee

*School of Science and Technology*

*Hong Kong Metropolitan University*

# Overview of this lecture

## Java Servlet Filter

- Using filter for authentication
  - Authentication filter example: *HelloSpringAuthfilter*

## Spring Security

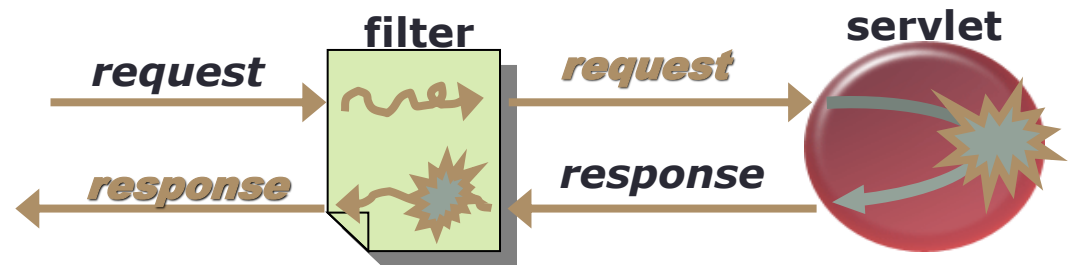
- Features
- **User stores:** in-memory, relational database, LDAP
  - Simple in-memory authentication example: *HelloSpringSecurity1*
  - Relational DB authentication example: *HelloSpringSecurity2*
- `<http>` element:
  - `<intercept-url>`, `<form-login>`, `<logout>`, `<remember-me>`
  - Using Spring Expression Language (SpEL)
- View layer security (using JSP taglib “security”)
  - `<security:authorize>`, `<security:authentication>`
- Incorporating all-in-one example : *HelloSpringSecurity3*

# Java Servlet Filter

- A **servlet filter** is a Java class that can do the following:
  - To **intercept requests** from a client before they access a resource at the back end of the web application
  - To **manipulate responses** from server before they are sent back to the client
- Introduced in Servlet Specification 2.3

Example uses:

- Authentication filters
- Logging and Auditing filters
- Image conversion filters
- Data compression filters
- Encryption filters

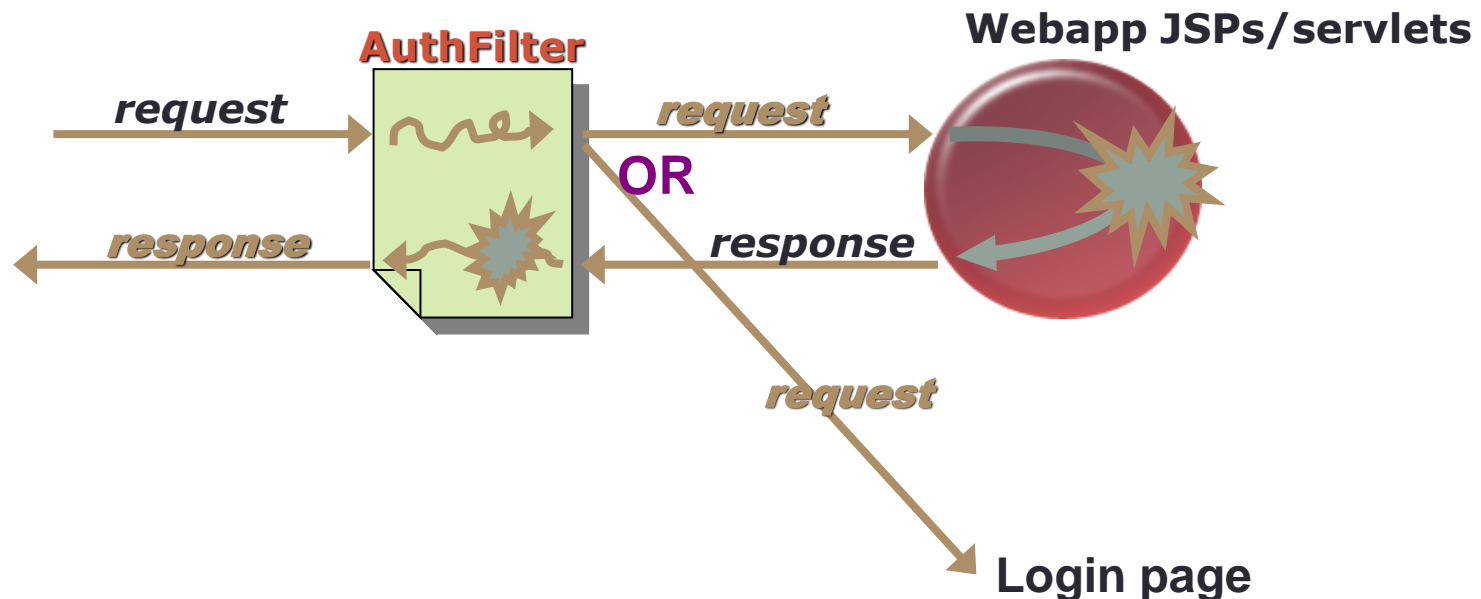


## Example: Authentication

- An application is built with various JSPs and/or servlets.
- Before the clients can use your application, they must log in as an authenticated user.
- **SOLUTION 1:**
  - Create a servlet that serves as the “interface servlet” (callback URL) in your application.
  - This servlet must make sure that the user is authenticated, and if not, then forward them to a URL for log-in.
- **PROBLEM:**
  - We need a separate servlet layer that handles every request from the users of your webapp.
- **BETTER SOLUTION?**

## Example: Authentication with Filter

- Instead of creating a separate servlet, create a filter that is used before using each of your web application's JSPs/servlets.
- This filter checks whether the user is authenticated:
  - If yes, passes on this information to your main webapp's JSPs / servlets.
  - If no, then forwards the request to a log-in page.

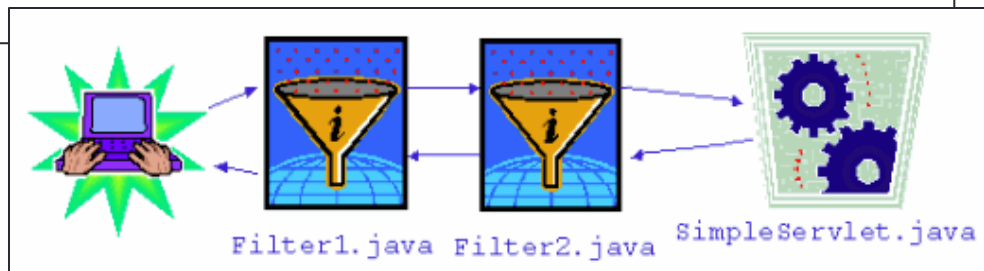


# Step 1: Defining a filter in web.xml

```
<filter>
  <filter-name>filter1</filter-name>
  <filter-class>com.example.Filter1</filter-class>
</filter>
<filter-mapping>
  <filter-name>filter1</filter-name>
  <url-pattern>/mylink</url-pattern>
</filter-mapping>
```

*... Similar for filter2; filter is applied in order of web.xml ...*

```
<servlet>
  <servlet-name>simpleServlet</servlet-name>
  <servlet-class>com.example.SimpleServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>simpleServlet</servlet-name>
  <url-pattern>/mylink</url-pattern>
</servlet-mapping>
```



## Step 2: Creating a Filter class

```
public class Filter1 implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain)
        throws IOException, ServletException {
        out.println("Filtering request ... ");
        chain.doFilter(request, response);
        out.println("Filtering response ... ");
    }

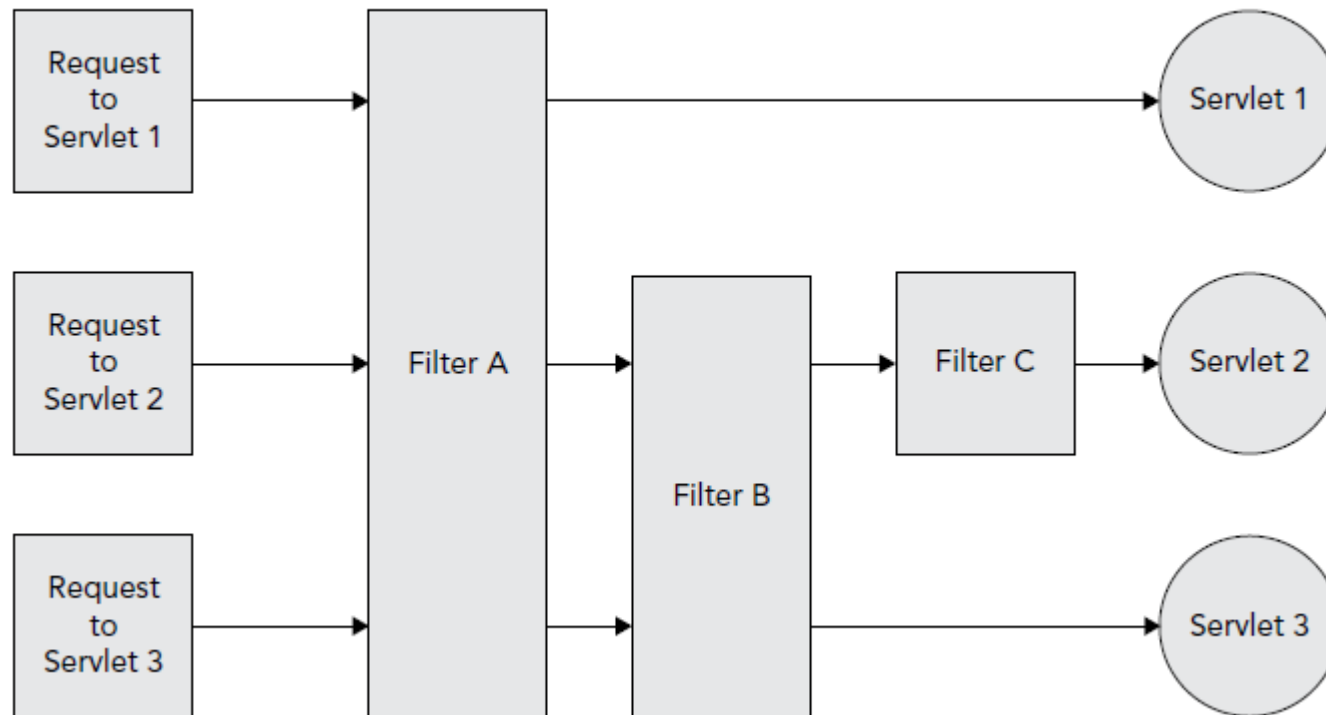
    @Override
    public void init(FilterConfig fc) throws ServletException {}

    @Override
    public void destroy() {}

}
```

# Servlet Filter Chain

- We may have a chain of filters (to appear in Lab exercises).





# Spring MVC guestbook application

- Example webapp: *HelloSpringAuthfilter*

## Model

GuestBookEntry.java

Integer id;  
String name;  
String message;  
Date date;

## Controller

GuestBookController

IndexController

## View

GuestBook.jsp

AddComment.jsp

EditComment.jsp

/guestbook/view, /guestbook/

/guestbook/add

/guestbook/edit?id=1

[Logout](#)

## Guest Book

- keith (2017-03-22): [\[Edit\]](#)  
This is a test message.

[Add Comment](#)

[Logout](#)

## Add Comment

Name:   
Message:

[Logout](#)

## Edit Comment

Name: keith  
Message:

# RedirectView object returned by controller

- Controller **IndexController** redirects request on URL <base URL>/ to the URL <base URL>/guestbook:

```
@Controller
public class IndexController {

    @GetMapping
    public View index() {
        return new RedirectView("/guestbook", true);
    }
}
```

- The **RedirectView** object redirects the client to another URL.
- The URL can be absolute or relative.
- By default, URL is **relative to the server URL** (e.g., localhost:8080) instead of the application's base URL (e.g., localhost:8080/appname).
- The second boolean argument "**true**" makes the URL **context-relative**.

# Controller: AuthenticationController

- We add a controller `AuthenticationController` that contains the user database (a `Map` object) and business logic for authentication.

**@Controller**

AuthenticationController.java

```
public class AuthenticationController {  
  
    private static final Map<String, String> userDatabase = new ConcurrentHashMap<>();  
  
    static {  
        userDatabase.put("keith", "keithpw");  
        userDatabase.put("john", "johnpw");  
    }  
  
    private ModelAndView guestbookRedirect() {  
        return new ModelAndView(new RedirectView("/guestbook", true));  
    }  
  
    ...  
}
```

- The function `guestbookRedirect()` redirects the client to the guestbook view page. Here, we return a **ModelAndView** object for **URL redirect**.

# Controller: AuthenticationController (cont')

```
@GetMapping("/login")
public ModelAndView login(HttpSession session) {
    if (session.getAttribute("username") != null) {
        return guestbookRedirect();
    }
    ModelAndView mav = new ModelAndView("login");
    mav.addObject("loginFailed", false);
    mav.addObject("loginForm", new Form());
    return mav;
}
```

```
public static class Form {

    private String username;
    private String password;
```

```
// getters and setters for username & password
```

```
}
```

AuthenticationController.java

Already logged in

You don't need to set modelAttribute if you use the default form-backing object attribute name "command".

login.jsp

```
<form:form method="post" modelAttribute="loginForm" >
    <form:label path="username">Username:</form:label> <br />
    <form:input type="text" path="username" /><br /><br />
    <form:label path="password">Password:</form:label><br />
    <form:input type="password" path="password" /><br /><br />
    <input type="submit" value="Log In"/>
</form:form>
```

# Controller: AuthenticationController (cont')

**@PostMapping("/login")**

```
public ModelAndView login(HttpSession session, HttpServletRequest request, Form form) {  
    if (session.getAttribute("username") != null) {  
        return guestbookRedirect();  
    }  
    if (form.getUsername() == null || form.getPassword() == null  
        || !userDatabase.containsKey(form.getUsername())  
        || !form.getPassword().equals(userDatabase.get(form.getUsername()))) {  
        ModelAndView mav = new ModelAndView("login");  
        mav.addObject("loginFailed", true);  
        mav.addObject("loginForm", new Form());  
        return mav;  
    }
```

Already logged in

For showing login failure message

```
    session.setAttribute("username", form.getUsername());
```

```
    request.changeSessionId();
```

```
    return guestbookRedirect();  
}
```

If login is successful, do  
Session migration

**@GetMapping("/logout")**

```
public String logout(HttpSession session) {  
    session.invalidate();  
    return "redirect:/login";  
}
```

Logout through a hyperlink to /logout

Alternative way for URL redirect

AuthenticationController.java

# Filter class for authentication

AuthFilter.java

```
public class AuthFilter implements Filter {  
  
    @Override  
    public void doFilter(ServletRequest sr, ServletResponse sr1, FilterChain chain)  
        throws IOException, ServletException {  
        HttpServletRequest request = (HttpServletRequest) sr;  
        HttpServletResponse response = (HttpServletResponse) sr1;  
        HttpSession session = request.getSession(false);  
  
        if (session == null || session.getAttribute("username") == null) {  
            response.sendRedirect(request.getContextPath() + "/login");  
        } else {  
            chain.doFilter(sr, sr1);  
        }  
    }  
  
    @Override  
    public void init(FilterConfig fc) throws ServletException { }  
  
    @Override  
    public void destroy() { }  
}
```

The **false** value ensures that a new session will not be created.

Get the context root of the web app

# Defining the filter in web.xml

- We need to define the filter in the deployment descriptor (web.xml):

```
<filter>
  <filter-name>authFilter</filter-name>
  <filter-class>hkmu.comps380f.filter.AuthFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>authFilter</filter-name>
  <url-pattern>/guestbook/*</url-pattern>
</filter-mapping>
```

web.xml

- authFilter** will intercept all requests and responses for the URLs matching **/guestbook/\***
- This example shows how filter can be used for authenticating users.
- If there are more types of user accounts, the login code will become much more complicated, and we also need a lot of work to customize the content that each type of users can see.

# SPRING SECURITY

---



# Spring Security

- Provide Enterprise-level authentication and authorization services
- **Authentication** is based on implementation of **GrantedAuthority** interface
  - Logging in with a username and password
  - Different user type: **ROLE\_USER**, **ROLE\_ADMIN**, etc.
- **Authorization** is based on Access Control List
  - Access control list is a list of access control entities, each of them identifies a user or user group, and specifies the access rights allowed, denied, or audited for that user or user group.
- We will focus on **Authentication**.
- Spring Security was originally the ACEGI project, but ACEGI requires a lot of XML configurations.
- ACEGI is rebranded as Spring Security around Spring 2.0 release.
- Simplified configuration with Security namespace and **configuration by convention**.

# Spring Security: Features

- Declarative security
  - Keep security details out of your code
- Authentication
  - Against virtually any user store: in-memory, relational DB (database), LDAP, X.509 client certificate, OpenID, etc.
- Web URL and method authorization
- Support for anonymous sessions, concurrent sessions, remember-me, channel-enforcement (HTTP/HTTPS) and more
- Spring-based, but can be used for non-Spring web frameworks
- Provides a **security** namespace for Spring
  - Much less XML configuration is required
- Supports SpEL (Spring Expression Language)

Current version: **Spring Security 5**

# Spring Security: Maven dependencies

**Include the following Maven dependencies:**

- spring-security-core
- **spring-security-web**
- **spring-security-config**

**Optional Maven dependencies:**

- spring-security-taglibs
- spring-security-ldap
- spring-security-acl
- spring-security-cas-client
- spring-security-openid
- ...

# Simple example: Guestbook web application

- Example webapp: *HelloSpringSecurity1*

## Model

GuestBookEntry.java

```
Integer id;  
String name;  
String message;  
Date date;
```

## Controller

GuestBookController

IndexController

## View

GuestBook.jsp

AddComment.jsp

EditComment.jsp

/guestbook/view, /guestbook/

/guestbook/add

/guestbook/edit?id=1

Log out

## Guest Book

- keith (2020-03-20): [\[Edit\]](#)  
This is a test message.

[Add Comment](#)

We don't need AuthFilter,  
AuthenticationController, and  
login.jsp.

# DelegatingFilterProxy in web.xml

- We need to define the filter “springSecurityFilterChain” in the deployment descriptor (web.xml):

web.xml

```
<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>
```

- We proxy all requests (URL /\*) to a Spring bean with ID “springSecurityFilterChain” (You must use this name only).

# Root Spring application context: Spring Security

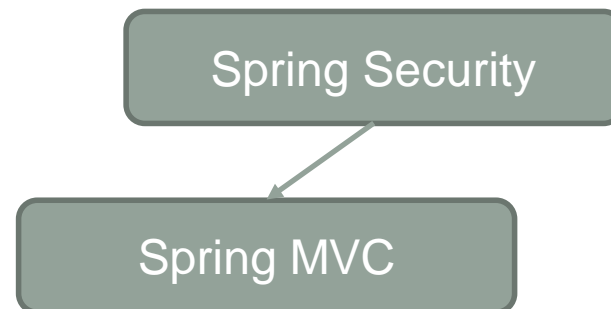
- We will set up Spring Security in the **root** Spring application context.
- Its Spring XML configuration file is **/WEB-INF/spring/security.xml**

web.xml

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring/*.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

- In this example, we have two Spring application contexts:
  - **root**: for Spring Security
  - DispatcherServlet: for Spring MVC



# Spring XML configuration: security.xml

- Spring XML configuration file is **/WEB-INF/spring/security.xml**

security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<b:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:b="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security.xsd"
">
...
```

- Note that the **default XML namespace** is changed to the Spring **security** namespace, instead of the **beans** namespace. It is because we will use mostly the **security** namespace in this XML file.
- We use the **prefix b** for the **beans** namespace.

# The <http /> tag

security.xml (cont')

<http />

In Spring Security 5, a password encoder is required by default. Add {noop} as prefix for plain text password.

```
<user-service>
  <user name="keith" password="{noop}keithpw"
    authorities="ROLE_USER, ROLE_ADMIN" />
  <user name="john" password="{noop}johnpw" authorities="ROLE_USER" />
</user-service>
</b:beans>
```

- **Require authentication to every URL** in your webapp
- **Generate a login** form for you.

<base URL>/**login**

<base URL>/**login?error**

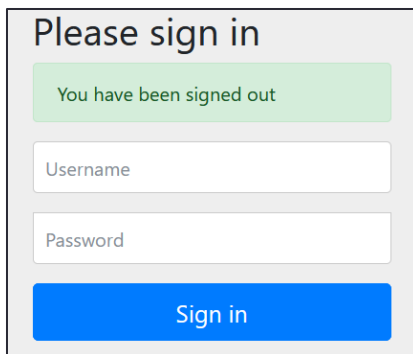
- **Form-based authentication:** *in-memory* user store <user-service>
  - User authorities are prefixed with **ROLE\_**



# Spring XML configuration: security.xml (cont')

- Allow the user to logout (**<base URL>/logout**)

**<base URL>/login?logout** (URL is changed after logout)



Please sign in

You have been signed out

Username

Password

Sign in

- **CSRF (Cross-site request forgery) attack prevention**
  - CSRF forces an end user to execute unwanted actions on a web application in which they're currently authenticated.
- **Session Fixation protection**
  - URL rewriting is disabled for session tracking

# CSRF token in HTML forms

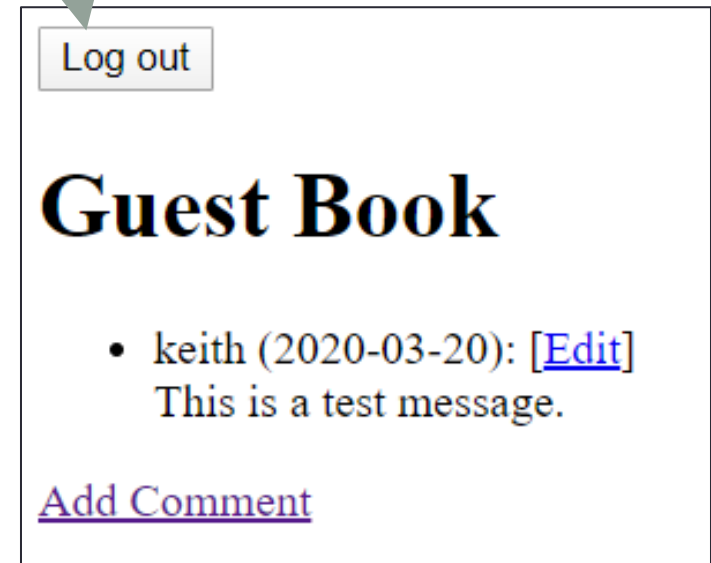
- Use HTML form instead of hyperlink for logout (**<base URL>/logout**)

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}" method="post">
  <input type="submit" value="Log out" />
  <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>
```

1. If you use the ordinary HTML form tag, it is **necessary** to add the hidden input control for the **CSRF token**.

2. If you use the **<form:form>** tag, Spring Security will automatically generate a CSRF form field.

➤ Except for a **multipart** form



Log out

## Guest Book

- keith (2020-03-20): [\[Edit\]](#)  
This is a test message.

[Add Comment](#)

# User stores

- Spring Security is extremely flexible and is capable of authenticating users against virtually any data store.
- 1. In-memory authentication** (webapp example: *HelloSpringSecurity1*)
  - Suitable for debugging and developer testing purposes
  - Not suitable for production application
- 2. Relational database**
  - It is better to maintain user data in a database
- 3. LDAP (Lightweight Directory Access Protocol)**
  - LDAP is a software protocol for enabling anyone to locate organizations, individuals, and other resources such as files and devices in a network, whether on the public Internet or on a corporate intranet.
  - Commonly used in enterprise environment
- 4. Many more including custom user store implementations

# Relational database: Apache Derby

Apache Derby is an open source relational database implemented entirely in Java and available under the Apache License, Version 2.0.

Some key advantages include:

- Derby has a small footprint -- about 3.5 megabytes for the base engine and embedded JDBC driver.
- Derby is based on the Java, JDBC, and SQL standards.
  - Not as powerful as MySQL, Oracle Database
- Derby provides an embedded JDBC driver that lets you embed Derby in any Java-based solution.
- Derby also supports the more familiar client/server mode with the Derby Network Client JDBC driver and Derby Network Server.
- Derby is easy to install, deploy, and use.

Reference: <https://db.apache.org/derby/>

# Setting up Derby database in Apache NetBeans

- Visit the Derby website at <http://db.apache.org/derby/>
- Download Derby for Java 8 or higher (choose the lib.zip file):

## For Java 8 and Higher

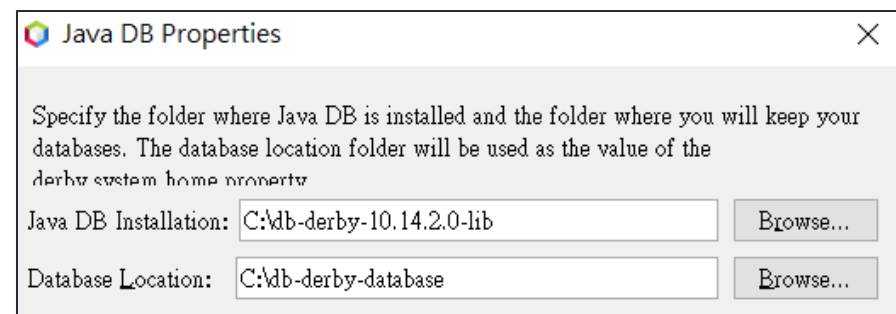
- [10.14.2.0](#) (May 3, 2018 / SVN 1828579)
- [10.13.1.1](#) (October 25, 2016 / SVN 1766613)

- Unzip this lib.zip file to any location you choose.
- In **NetBeans**, select the **Services** tab, expand the **Databases** entry, right-click on **JavaDB**, select **Properties**.

➤ For the Derby location, enter the directory where you installed the Derby library.

➤ For the databases location, you can enter any directory.

This location is where your created databases are stored.



## Setting up Derby database in Apache NetBeans (con't)

- **Webapp example: *HelloSpringSecurity2***
- We can set up a Derby database in NetBeans, as follows:
  1. Open the **Services** panel in NetBeans.
  2. Under **Databases**, right-click on “JavaDB” and select “Create Database”:
    - Database name: Account
    - Username: nbuser
    - Password: nbuser

This creates a Derby database.
- 3. Right-click the newly created database, and select “Connect”
- 4. Right-click the database, and select “Execute Command”
- 5. Run the SQL statements to create tables and insert data into them

# Derby SQL statement

- The database has two tables: **users** and **user\_roles**

src/main/resources/create\_user.sql

```
CREATE TABLE users (  
    username VARCHAR(50) NOT NULL,  
    password VARCHAR(50) NOT NULL,  
    PRIMARY KEY (username)  
);  
  
CREATE TABLE user_roles (  
    user_role_id INTEGER NOT NULL  
    GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),  
    username VARCHAR(50) NOT NULL,  
    role VARCHAR(50) NOT NULL,  
    PRIMARY KEY (user_role_id),  
    FOREIGN KEY (username) REFERENCES users(username)  
);  
  
INSERT INTO users VALUES ('keith', '{noop}keithpw');  
INSERT INTO user_roles(username, role) VALUES ('keith', 'ROLE_USER');  
INSERT INTO user_roles(username, role) VALUES ('keith', 'ROLE_ADMIN');  
  
INSERT INTO users VALUES ('john', '{noop}johnpw');  
INSERT INTO user_roles(username, role) VALUES ('john', 'ROLE_USER');
```

# Configure the data source

- Include the following Maven dependencies:

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derby</artifactId>
  <version>10.14.2.0</version>
</dependency>
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.14.2.0</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${spring.framework.version}</version>
</dependency>
```

pom.xml

- Add a dataSource bean, which accesses the Derby database.

```
<b:bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <b:property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />
  <b:property name="url" value="jdbc:derby://localhost:1527/account" />
  <b:property name="username" value="nbuser" />
  <b:property name="password" value="nbuser" />
</b:bean>
```

security.xml



# Configure the authentication-provider

- Replace the <user-service> tag with the <authentication-manager> tag

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="dataSource"
      users-by-username-query=
        "SELECT username, password, true FROM users WHERE username=?"
      authorities-by-username-query=
        "SELECT username, role FROM user_roles WHERE username=?" />
  </authentication-provider>
</authentication-manager>
```

security.xml

- Spring Security expects certain user tables exist and there are 3 default SQL statements to query these tables, **taking a username parameter**.
- If we don't follow those table definitions, we need to override the SQLs.
  - users-by-username-query: Retrieve a user's **username**, **password**, and **whether or not they are enabled** (we hardcode it to true).
  - authorities-by-username-query: Retrieve user's granted authorities.
  - group-authorities-by-username-query: Retrieve authorities granted to a user as a member of a group (we did not use and did not override).

# Enabling web security using <http>

- The **<http>** tag is the central configuration element for web security.

```
<http auto-config="true" use-expressions="true">  
    // web security configuration  
</http>
```

- **auto-config** automatically includes the following tags:

- **<http-basic />**: Support HTTP basic authentication

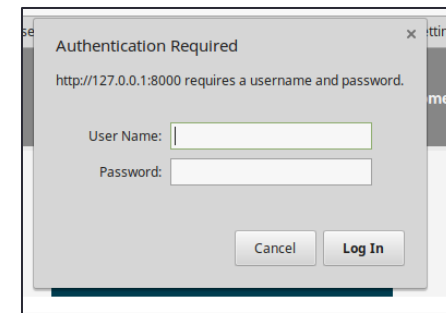
- When opening a website with HTTP basic authentication, the server will send back a header requesting authentication for a given resource. You will be asked to provide your username and password in order to access the website.

- **<form-login />**: Create a login-form (default or user-defined)

- **<logout />**: Create logout service (default or user-defined)

- **use-expressions** allows us to configure with **Spring EL expressions**

- Its default value is true.



# Securing by URL using `<intercept-url>`

We use `<intercept-url>` tags to secure web pages by matching requests.

```
<http auto-config="true">  
  <intercept-url pattern="/guestbook/edit" access="hasRole('ADMIN')" />  
  <intercept-url pattern="/guestbook/**" access="hasAnyRole('USER','ADMIN')" />  
</http>
```

`<intercept-url>` may include the following attributes:

- **pattern**: The URL to secure (**\*\*** will include any level of subdirectories)
- **access**: The expression to use to secure the URL
  - We can use SpEL here (see next slide)
- **requires-channel**: Can be “http” or “https” depending on whether a particular URL pattern should be accessed over HTTP or HTTPS.
- **method**: HTTP method of the incoming request for matching
- **filters="none"** : Matched request will bypass the security filter chain.

Note that the request matching is done **in the declaration order.**

# Spring Expression Language (SpEL)

- Spring Security extends the Spring Expression Language (SpEL) with several security-specific expressions.
- We can use SpEL in the **access** attribute of the `<intercept-url>` tag.
- Here, **role** is a user authority with the prefix "ROLE\_" removed.

Expression	Description
principal	Allows direct access to the principal object representing the current user
authentication	The user's authentication object
hasRole( <b>role</b> )	True if the current user has the given role.
hasAnyRole( <b>list of roles</b> )	True if the current user has any of the given roles (as a comma-separated list of strings)
hasIpAddress(IP address)	True if the request comes from the given IP address

- We may use logical operator in SpEL like **and**, **or**:  
`<intercept-url pattern="/spitter/**" access="hasRole('STUDENT') and hasIpAddress('192.168.1.2')"/>`

# Spring Expression Language (SpEL) (cont')

- The **remember-me** functionality allows a user to log in once and then be remembered by the application when the user come back to it later.

Expression	Description
permitAll	Always evaluates to true
denyAll	Always evaluates to false
isAnonymous()	True if the current user is an anonymous user
isRememberMe()	True if the current user is a <b>remember-me</b> user
isAuthenticated()	True if the user is not anonymous
isFullyAuthenticated()	True if the user is not anonymous, or not authenticated with remember-me

## <intercept-url> example

We use the <intercept-url> tag to secure web pages by matching requests.

### Webapp example: *HelloSpringSecurity3*

security.xml

```
<http auto-config="true">
  <intercept-url pattern="/guestbook/edit" access="hasRole('ADMIN')" />
  <intercept-url pattern="/guestbook/**" access="hasAnyRole('USER','ADMIN')" />
</http>
```

- The first tag restricts that only user with ROLE\_ADMIN can access the URL “/guestbook/edit”
  - A user without ROLE\_ADMIN (e.g., john) gets an HTTP status 403.
- The second tag allows use with either ROLE\_USER and ROLE\_ADMIN to access all URL starting with “/guestbook/”, except “/guestbook/edit”.
  - In the pattern, \* means one level of subdirectory, and \*\* means any level of subdirectory.

# Form-based authentication: Login

- We can customize login page using `<form-login>`:

```

<http auto-config="true">
  <form-login login-page="/login"
    authentication-failure-url="/login?error"
    username-parameter="username"
    password-parameter="password" />
</http>
  
```

security.xml

## Login

User:

Password:

Remember Me: ☐

- We add the a controller method for the URL `"/login"` in `IndexController`, and a custom login page `"/WEB-INF/jsp/view/login.jsp"`.

- In fact, `username` and `password` are default.

```

<h1>Login</h1>
<form action="login" method='POST'>
  User: <input type='text' name='username'><br />
  Password: <input type='password' name='password' /><br />
  Remember Me: <input type="checkbox" name="remember-me" /><br />
  <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
  <input name="submit" type="submit" value="Log In" /><br />
</form>
  
```

IndexController.java

```

@GetMapping("/login")
public String login() {
    return "login";
}
  
```

login.jsp

# Form-based authentication: Logout

- We can define custom logout service using <logout>, as follows.

```
<http auto-config="true">  
  <logout logout-url="/logout"  
    logout-success-url="/login?logout"  
    invalidate-session="true"  
    delete-cookies="JSESSIONID" />  
</http>
```

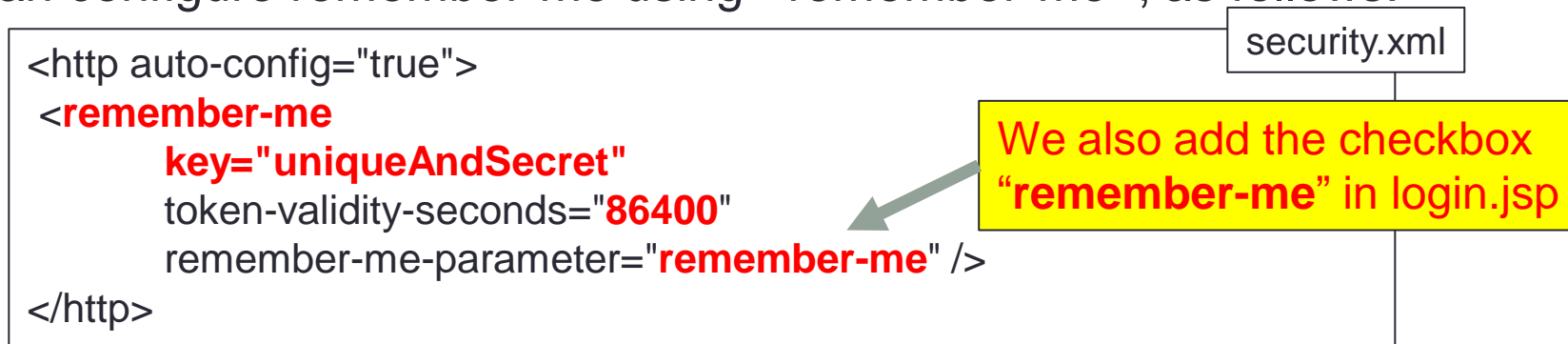
security.xml

- The above code will do the following when the user logout:
  - Invalidate the user's session
  - Remove the session cookie "JSESSIONID"



# Form-based authentication: Remember-me

- The **remember-me** functionality allows a user to log in once and then be remembered by the application when the user come back to it later.
  - A authenticated user can close the browser without being logged out.
- We can configure remember-me using <remember-me>, as follows:



- There are different ways of configuring the remember-me functionality.
- The above code **creates an additional "remember-me" cookie** with:
  - **username**: to identify the logged-in principal
  - **expirationTime**: to expire the cookie; the default is 2 weeks, and we change it to 1 day (i.e., 86400 seconds).
  - **MD5 hash**: of the previous 2 values + password + predefined **key**

## More on authentication

- **Problem:** The password is stored as plain text in the database, which is not secured.
- **Solution:**
  - Spring Security offers <password-encoder> to hash the passwords using different password hashing scheme (note that password hashing is a one-way process in the sense that the hashed password cannot be “decoded” back to the original password).
  - We can store the hashed password in database.
  - When a user logs in, the entered password will be hashed using the same scheme and then be compared with the database’s one.

# View layer security using JSP taglib “security”

- Spring Security provides a JSP tag library for
  - Restricting the display of certain content by user’s authority
  - Accessing the current authentication object

base.jspf

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

- You need the Maven dependency: **spring-security-taglibs**

JSP tag	What it does
<security:accesscontrollist>	Conditionally render its body content if the user is granted authorities by an access control list
<security:authentication>	Render details about the current authentication
<security:authorize>	Conditionally render its body content if the user is granted certain authorities, or if a SpEL expression evaluates to true

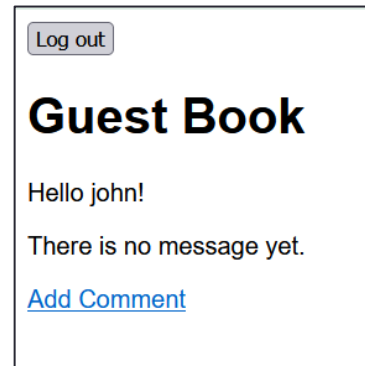
# View layer security: Example

- We added the following JSP code to GuestBook.jsp

GuestBook.jsp

```
<p>Hello <security:authentication property="principal.username" />!/</p>  
<security:authorize access="isAuthenticated() and principal.username=='keith'">  
  <p>This paragraph can only be seen by keith</p>  
</security:authorize>
```

- If “john” logs in, the page is:



- If “keith” logs in, the page is:

