COMP S380F Lecture 9: Spring Security, Spring Profiles

Dr. Keith Lee

School of Science and Technology

Hong Kong Metropolitan University

Overview of this lecture

- Java Servlet Filter
- Spring Security and its features (***HelloSpringSecurity ver. 1***)
- **User stores:**
  - Simple in-memory authentication (***HelloSpringSecurity ver. 2***)
  - **Spring data sources**:
    - JDBC driver-based, JNDI, Pooled, Embedded
  - **Spring profiles** feature example (***HelloSpringSecurity ver. 3***)

- Configuring **`SecurityFilterChain`** bean: (***HelloSpringSecurity ver. 4***)
  - `.authorizeHttpRequests()`, `.formLogin()`, `.logout()`, `.rememberMe()`
  - Using Spring Expression Language (SpEL)

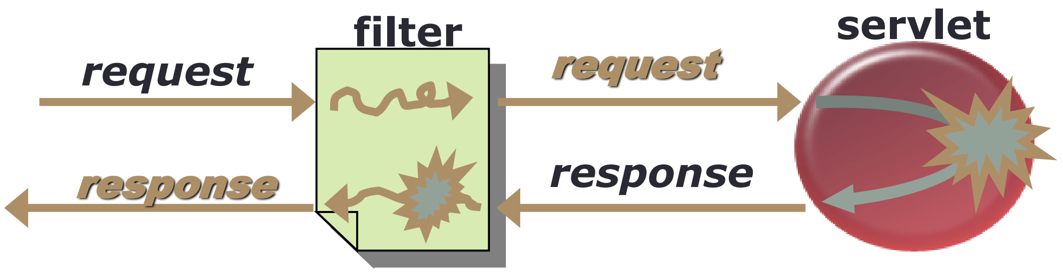
- View layer security (using **JSP taglib `security`**)
  - `<security:authorize>, <security:authentication>`

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

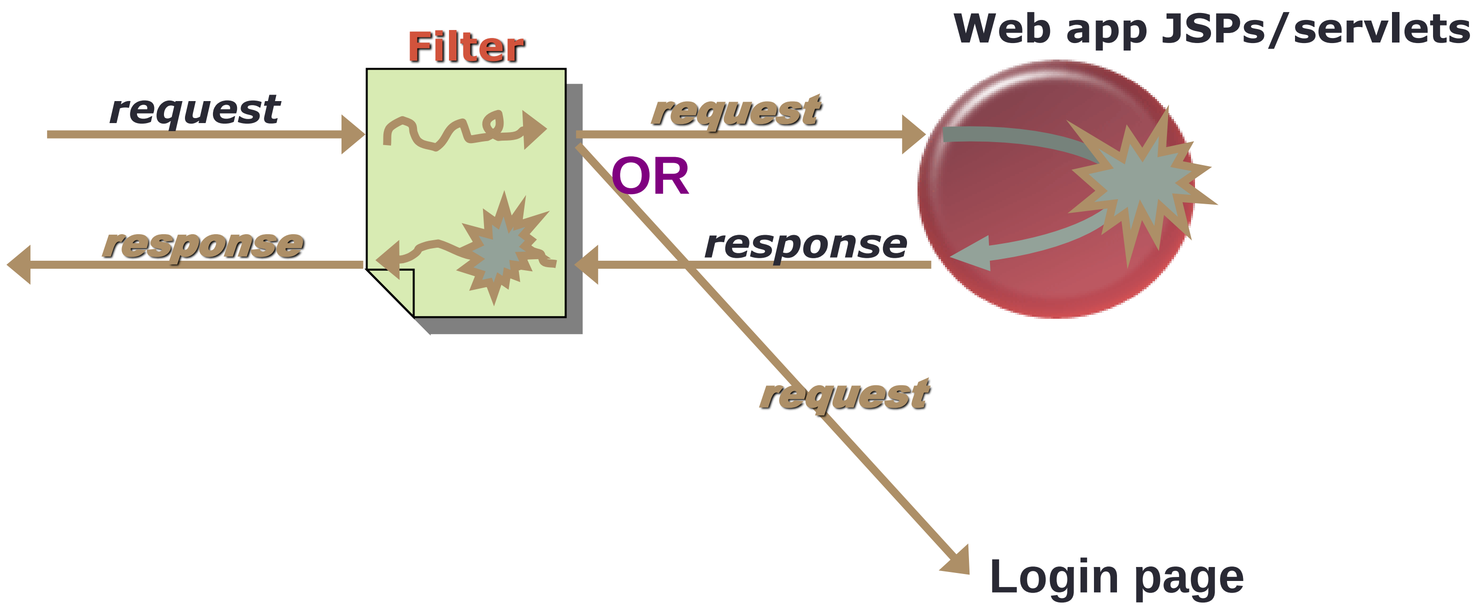
Example uses:

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



Example: Authentication with Filter

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



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**.

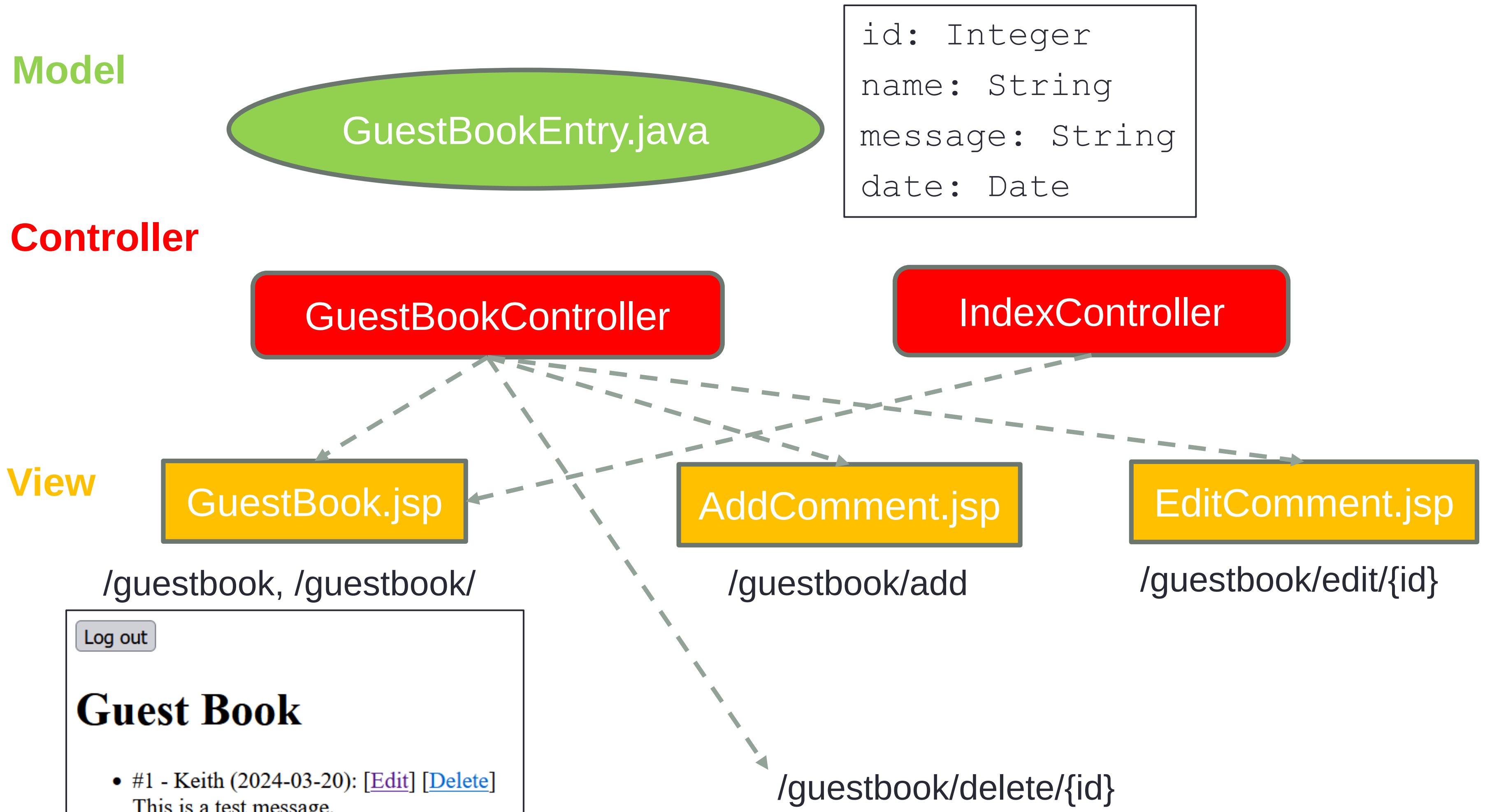
- Spring Security was originally the ACEGI project, but ACEGI requires a lot of XML configurations.
- ACEGI is rebranded as Spring Security around Spring 2.0 release.
- Simplified configuration with **Security namespace** and **configuration by convention**.

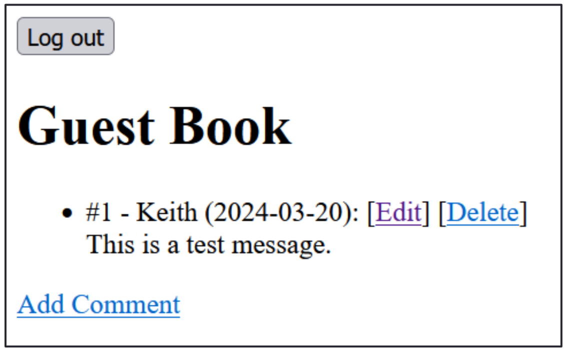
Spring Security: Features

- 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)
- Based on a filter class `org.springframework.web.filter.DelegatingFilterProxy`

Simple example: Guestbook web application

Web app example: lecture09-hellospringsecurity (commit ver. 1)




Log out

Guest Book

- #1 - Keith (2024-03-20): [Edit] [Delete]
  This is a test message.

Add Comment

Configuration for Spring Security in Spring Boot

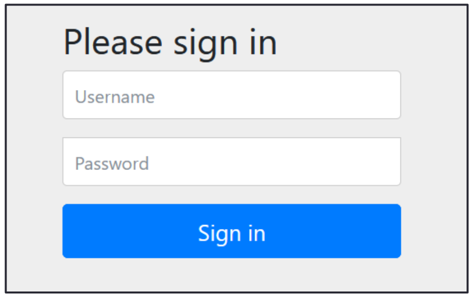
- Include the starter dependency for Spring Security:

```
implementation 'org.springframework.boot:spring-boot-starter-security'
```

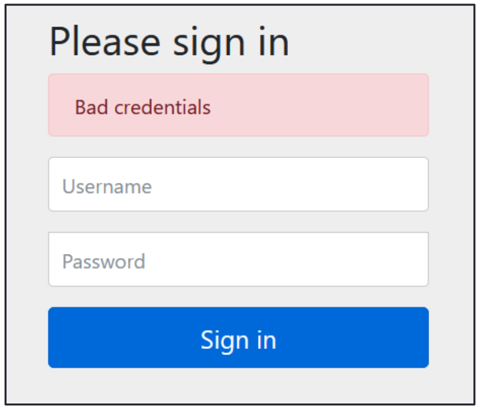
Effect:

- **Require authentication to every webapp URL** (even static contents like CSS)
- **Generate a login** form:

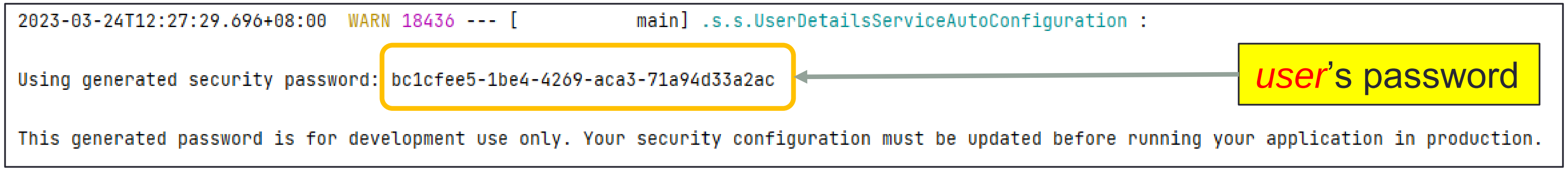
`<base URL>/login`



`<base URL>/login?error`



- **Form-based authentication**:
  - ➢ Default user account with username *user*
  - ➢ Password generated at project start time (can be found in the **console**).

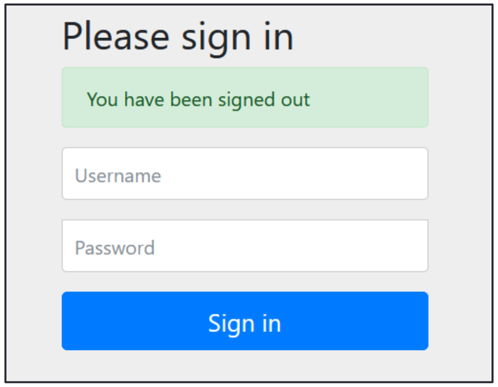


```
2023-03-24T12:27:29.696+08:00  WARN 18436 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: bc1cfee5-1be4-4269-aca3-71a94d33a2ac

This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

user's password

Configuration for Spring Security in Spring Boot (cont')

- Allow the user to logout (**`<base URL>/logout`**)
  - ➢ After logout, the URL will be updated to: `<base URL>/`**`login?logout`**
- **CSRF (Cross-site request forgery) attack prevention**
  - ➢ CSRF protects end users from executing unwanted actions on a web application in which they're currently authenticated.
- **Session Fixation protection**
  - ➢ URL rewriting is disabled for session tracking


Please sign in

You have been signed out

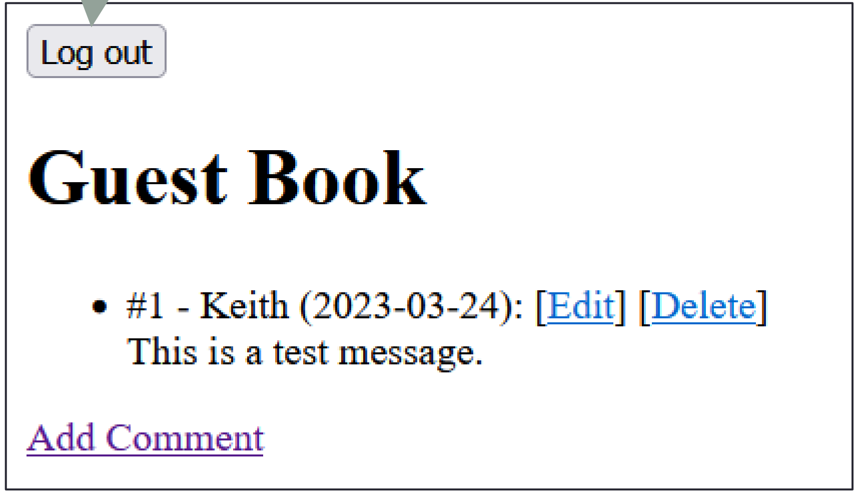
CSRF token in HTML forms

- 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.



Guest Book

- #1 - Keith (2023-03-24): [Edit] [Delete]
  This is a test message.

Add Comment

User stores for Spring Security

- Very flexible: Can authenticate users against virtually any data store.

1. **In-memory authentication** (webapp example: ***commit ver. 2***)
   - ➢ Suitable for debugging and developer testing purposes
   - ➢ Not suitable for production application
2. **Relational database** (webapp example: ***commit ver. 3***)
   - ➢ We can store the data in a database, which can even be an embedded/in-memory 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

In-memory user store

lecture09-hellospringsecurity (commit ver. 2)

package `hkmu.comps380f.config`

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withUsername("keith")
                .password("{noop}keithpw").roles("ADMIN", "USER").build();
        UserDetails user2 = User.withUsername("john")
                .password("{noop}johnpw").roles("USER").build();
        InMemoryUserDetailsManager userDetailsManager
                = new InMemoryUserDetailsManager();
        userDetailsManager.createUser(user1);
        userDetailsManager.createUser(user2);
        return userDetailsManager;
    }
}
```

- `@Configuration`: This class contains Spring bean definition with `@Bean`.
- `@EnableWebSecurity`: This class has Spring Security configuration.

In-memory user store (cont’)

hkmu.comps380f.config.**SecurityConfig**

```
@Bean
public UserDetailsService userDetailsService() {
    UserDetails user1 = User.withUsername("keith")
            .password("{noop}keithpw").roles("ADMIN", "USER").build();
    UserDetails user2 = User.withUsername("john")
            .password("{noop}johnpw").roles("USER").build();
    InMemoryUserDetailsManager userDetailsManager
            = new InMemoryUserDetailsManager();
    userDetailsManager.createUser(user1);
    userDetailsManager.createUser(user2);
    return userDetailsManager;
}
```

- `UserDetailsService` is used as a User DAO.
- All user roles are automatically prefixed with **`ROLE_`**
  - ➢ `.roles("ADMIN", "USER")` is equivalent to

    `.authorities("ROLE_ADMIN", "ROLE_USER")`
- **`{noop}`**: Password is stored in plain text (without a password encoder).

User store using a Spring data store

```
hkmu.comps380f.config.SecurityConfig
```

```
@Autowired
@Bean
public UserDetailsService jdbcUserDetailsService(DataSource dataSource) {
    String usersByUsernameQuery
            = "SELECT username, password, true FROM users WHERE username=?";
    String authsByUserQuery
            = "SELECT username, role FROM user_roles WHERE username=?";
    JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
    users.setUsersByUsernameQuery(usersByUsernameQuery);
    users.setAuthoritiesByUsernameQuery(authsByUserQuery);
    return users;
}
```

- `UserDetailsService` can be defined using a **data source** (`Autowired`).
- The above bean replaces **`InMemoryUserDetailsManager`** with a **`JdbcUserDetailsManager`** which has to be used with a **`DataSource`** .
- Spring Security expects default user tables, and there are 3 default SQL statements to query these tables, **taking a username parameter**.

Details: https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/jdbc.html#servlet-authentication-jdbc

User store using a Spring data store (cont')

- We do not follow the default user tables, and will define our own tables.

/resources/sql/schema.sql

```
# ...

DROP TABLE IF EXISTS user_roles;
DROP TABLE IF EXISTS users;
CREATE TABLE users (
    username VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL,
    PRIMARY KEY (username)
);
CREATE TABLE user_roles (
    user_role_id INTEGER GENERATED ALWAYS AS IDENTITY,
    username VARCHAR(50) NOT NULL,
    role VARCHAR(50) NOT NULL,
    PRIMARY KEY (user_role_id),
    FOREIGN KEY (username) REFERENCES users(username)
);
```

/resources/sql/data.sql

```
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');
```

User store using a Spring data store (cont’)

```
hkmu.comps380f.config.SecurityConfig
```

```
@Autowired
@Bean
public UserDetailsService jdbcUserDetailsService(DataSource dataSource) {
    String usersByUsernameQuery
            = "SELECT username, password, true FROM users WHERE username=?";
    String authsByUsernameQuery
            = "SELECT username, role FROM user_roles WHERE username=?";
    JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
    users.setUsersByUsernameQuery(usersByUsernameQuery);
    users.setAuthoritiesByUsernameQuery(authsByUsernameQuery);
    return users;
}
```

- If we don’t follow those table definitions, we need to override the SQLs.

1. `usersByUsernameQuery`: Retrieve a user’s **username**, **password**, and **whether or not they are enabled** (**we hardcode it to true here**).
2. `authoritiesByUsernameQuery`: Retrieve user’s authorities (**roles**).
3. `groupAuthoritiesByUsernameQuery`: Retrieve authorities granted to a user as a member of a **group** (we did not use and override here).

Data source 1: JDBC driver-based data source

/resources/application-qa.properties

```
spring.datasource.url=jdbc:h2:./Data/myDB;AUTO_SERVER=TRUE
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
```

- The simplest data source you can configure in Spring is one that's defined through a JDBC driver.
- You should at least specify the URL by setting the `spring.datasource.url` property. Otherwise, Spring Boot tries to auto-configure an embedded database (see Data source 4).
- Other examples of `spring.datasource.url`:
  - ➢ `jdbc:h2:tcp://localhost/~/test`
  - ➢ `jdbc:mysql://localhost/test`
- Spring Boot can deduce the JDBC driver class for most databases from the URL. If you need to specify a specific class, you can use the `spring.datasource.driver-class-name` property.

Details: https://docs.spring.io/spring-boot/docs/current/reference/html/data.html#data.sql.datasource.configuration

Data source 2: JNDI data source

- Jakarta EE application servers allow you to configure data sources to be retrieved via **JNDI (Java Naming and Directory Interface)**.
  - ➢ JNDI is a directory service that allows Java software to discover and look up data and objects via a name.
- **Benefits:**
  - ➢ JNDI data sources can be managed completely **external to web app**, allowing the web app to ask for a data source when it's ready to access the database.
  - ➢ Data sources managed in an application server (AS) are often **pooled** for greater performance (i.e., it draws its connection from a database connection pool) and can be hot-swapped by system administrators.
- `spring.datasource.jndi-name` can be used as an alternative to `spring.datasource.url`, e.g., to access a JBoss AS-defined DataSource

```
spring.datasource.jndi-name=java:jboss/datasources/customers
spring.datasource.username=sa
spring.datasource.password=password
```

name of the resource in JNDI

Data source 3: Pooled data source

- Spring supports Database Connection Pools to share a pool of open connections:
  - **HikariCP** (Spring Boot’s default, https://github.com/brettwooldridge/HikariCP)
  - **Apache Commons DBCP 2** (http://commons.apache.org/proper/commons-dbcp)
  - **c3p0** (http://sourceforge.net/projects/c3p0)
- E.g., **HikariCP**’s data source:

/resources/**application-prod.properties**

```
spring.datasource.url=jdbc:h2:./Data/myDB;AUTO_SERVER=TRUE
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password

spring.datasource.hikari.auto-commit=true
spring.datasource.hikari.idle-timeout=10000
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.max-lifetime=30000
spring.datasource.hikari.pool-name=HikariCP-1
```

JDBC driver for H2 database

max. time for a connection to be idle (in milliseconds)

at most 20 (idle and in-use) connections is allowed

Controls maximum lifetime of a connection in the pool.
Strongly recommended to set this value and it should be less than any DB connection time limit.

More configurations: https://github.com/brettwooldridge/HikariCP?tab=readme-ov-file#gear-configuration-knobs-baby

Data source 4: Embedded database

- An embedded database runs as part of your application instead of as a separate database server that your application connects to.
- It is in-memory, and thus cannot be accessed directly in IntelliJ.
- Not suitable for production but perfect for development & testing:
  - ➢ You can populate your database with test data, and reset the database every time you restart your application or run your tests.
- E.g.,

/resources/**application-dev.properties**

```
spring.datasource.url=jdbc:h2:mem:myDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
```

lecture09-hellospringsecurity (commit ver. 3)

Spring profiles

- We may need different data source beans in different environment.
  - E.g., development, quality assurance (QA), production.
- We can set different **Spring profile** for different data source.
- Spring Boot allows us to set **profile-specific properties files**, which should be named in the format **`application-{profile}.properties`**.
- We can select a profile (e.g., dev, qa, prod) by setting spring.profiles.active:

/resources/**application.properties**

```
spring.profiles.active=dev
# ...
```

- Spring Boot will load
  - the properties in **`application.properties`** file for all profiles, and
  - the properties in **`application-{profile}.properties`** only for the specified profile.
- In our web app,
  dev: embedded database, qa: JDBC data source, prod: pooled data source

Configuring web security using `SecurityFilterChain`

- The bean `SecurityFilterChain` is auto-configured by default, as follows.
- **`http`** allows configuring web-based security for specific http requests.

`hkmu.comps380f.config.SecurityConfig`

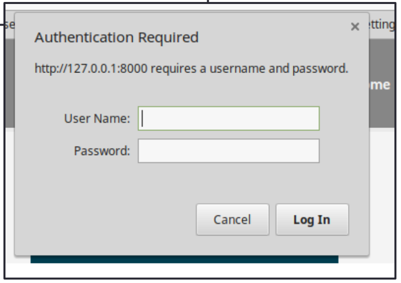
```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
        throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            .anyRequest().authenticated()
        )
        .formLogin(withDefaults())
        .httpBasic(withDefaults());
    return http.build();
}
```

Lambda expression:

`(Parameters) -> { Body }`

`->` separates parameters & body (Java statement)

1. Any request to web app requires the user to be authenticated
2. Lets users authenticate with form-based login
3. Supports HTTP basic authentication



Authentication Required

http://127.0.0.1:8000 requires a username and password.

User Name:

Password:

➢ When opening a website, the server will send back a header requesting authentication. Clients will be asked to provide username and password.

Reference: https://docs.spring.io/spring-security/reference/servlet/configuration/java.html#jc-httpsecurity
Lambda Expression tutorial: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Securing URL patterns

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
        throws Exception {
    http
        // ...
        .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/resources/**", "/signup", "/about").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/db/**").access(
         new WebExpressionAuthorizationManager("hasRole('ADMIN') and hasRole('DBA')"))
                .anyRequest().denyAll()
        );
    return http.build();
}
```

We use **`.authorizeHttpRequests()`** to secure web pages by matching requests:

- Note that the request matching is done **in the declaration order.**
- **`.requestMatchers`**: URL patterns to secure (****** will include any level of subdirectories)
- **`.anyRequest().denyAll()`**: Any URL that has not already been matched on is denied access. This is a good strategy if you do not want to accidentally forget to update your authorization rules.

Reference: https://docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html

Spring Expression Language (SpEL)

- Spring Security extends the Spring Expression Language (SpEL) with several security-specific expressions.
- 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(*role*)	True if the current user has the given role.
hasAnyRole(*list of roles*)	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 use SpEL in `.access()`, where logical operator **and**, **or** can be used:

```
http.authorizeHttpRequests(authorize -> authorize
    // ...
    .requestMatchers("/db/**").access(
        new WebExpressionAuthorizationManager("hasRole('ADMIN') and hasRole('DBA')"));
```

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 **remember-me** user
isAuthenticated()	True if the user is not anonymous
isFullyAuthenticated()	True if the user is not anonymous, or not authenticated with remember-me

- Spring Security extends SpEL to **`SecurityExpressionOperations`** such that we can use expression-based security **methods**:

➢ E.g., denyAll(), permitAll(), hasAnyRole(String... roles), …

Documentation: https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/access/expression/SecurityExpressionOperations.html

lecture09-hellospringsecurity (commit ver. 4)

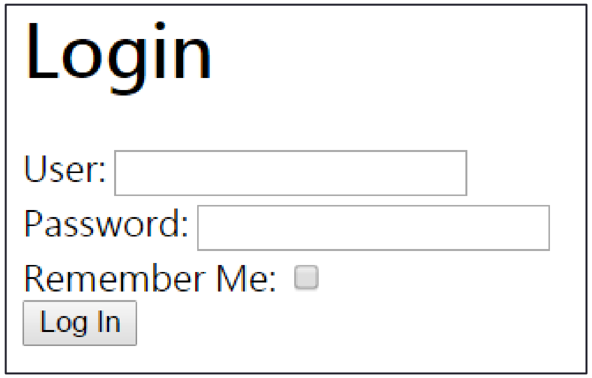
Our web app example

Our web app example's `.authorizeHttpRequests()` for secure web pages by matching requests.

```
http
  .authorizeHttpRequests(authorize -> authorize
    .requestMatchers("/guestbook/edit/**", "/guestbook/delete/**").hasRole("ADMIN")
    .requestMatchers("/guestbook/**").hasAnyRole("USER","ADMIN")
    .anyRequest().permitAll()
  )
```

- The first match restricts that only users with **`ROLE_ADMIN`** can access the URL `/guestbook/edit/**` and `/guestbook/delete/**`
  - `*` means one level of subdirectory
  - `**` means any level of subdirectory.
  - A user without **`ROLE_ADMIN`** (e.g., **john**) gets an HTTP status 403.
- The second match allows users with either **`ROLE_USER`** and **`ROLE_ADMIN`** to access all URL starting with `/guestbook/`, except URLs in the first match.
- The third match permits all users to access any URLs not previously matched.

Form-based authentication: Login


Login

User:

Password:

Remember Me: ☐

Log In

- We use `.formLogin()` to customize login page:

```
http
    .formLogin(form -> form
        .loginPage("/login")
        .failureUrl("/login?error")
        .usernameParameter("username")
        .passwordParameter("password")
        .permitAll()
    )
```

Default values

IndexController.java

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

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

login.jsp

```
<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>
```

Form-based authentication: Logout

- We use `.logout()` to customize logout service:

```
http
    .logout(logout -> logout
        .logoutUrl("/logout")
        .logoutSuccessUrl("/login?logout")
        .invalidateHttpSession(true)
        .deleteCookies("JSESSIONID")
    )
```

- The above code will do the followings:
  - ➢ Sets the logout URL to `/logout`

  When a user successfully logged out:
  - ➢ Redirects the user to the URL `/login?logout`
  - ➢ Invalidates the user’s session
  - ➢ Removes 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 use `.rememberMe()` to configure the remember-me functionality:

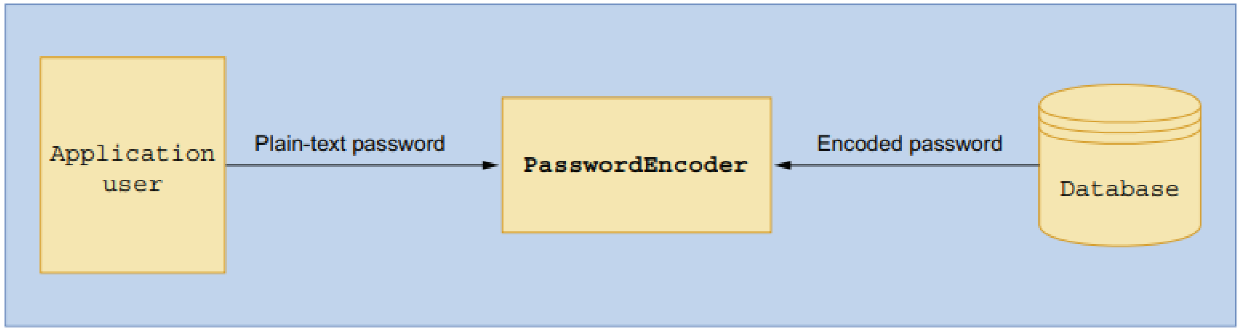
```
http
  .rememberMe(remember -> remember
        .key("uniqueAndSecret")
        .tokenValiditySeconds(86400)
        .rememberMeParameter("remember-me")
)
```

We also add the checkbox “**remember-me**” in login.jsp

- 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 logins, 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

build.gradle

```
implementation 'org.springframework.boot:spring-security-taglibs'
```

base.jspf

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

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

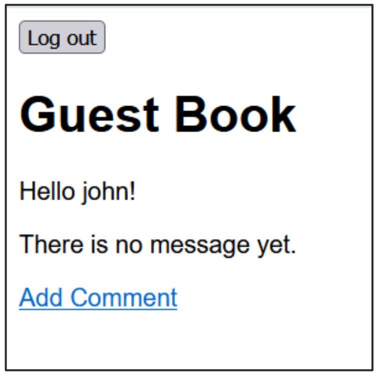
View layer security: Example

- 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:



Log out

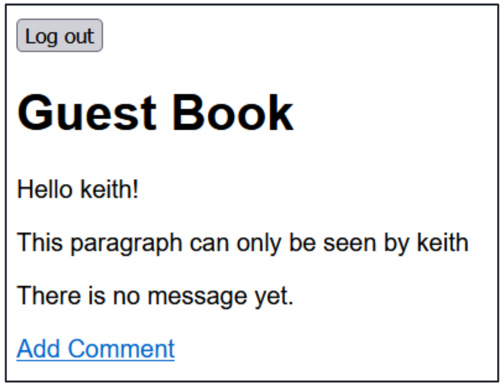
Guest Book

Hello john!

There is no message yet.

Add Comment

- If "**keith**" logs in, the page is:



Log out

Guest Book

Hello keith!

This paragraph can only be seen by keith

There is no message yet.

Add Comment