Spring MVC Tutorial – Contact List Application

Step 21: Hashing Passwords

Step 21: Hashing Passwords

Overview

The security that we have in place for our application is effective, but we still have some work to do to tighten things down. As it stands now, the passwords in the database are cleartext, which means that anyone with access to our database (either an attacker or an employee) can see (and take) the username and password for each account in the system. In this step, we will explore how to protect the password values using a technique called hashing.

Hashing vs. Encrypting

Two techniques that are used to hide the real value of data are **hash algorithms** and **encryption**. A hash function is an algorithm that takes an arbitrary block of data and returns a fixed-sized bit string. Hash functions are one-way, meaning that the process cannot be reversed. An encryption algorithm provides a 1-to-1 mapping between an arbitrary length input and output, so the mapping for the encryption is reversible.

Hash functions are used to protect password data in a database and we will take advantage of two properties of hash functions:

1. Hash functions are very difficult (virtually impossible) to reverse. In other words, if you have a hashed value, it is impossible to reverse the hash to determine what the original input was.
2. A given input will always map to the same hash value.

We will update our code so that it stores the hashed values of password instead of the cleartext value. We will also make updates to our login code so that the system will hash the password typed by the user before comparing it to the value in the database.

Preliminaries

Before we can effectively change how we are storing and checking passwords, we must update our application so that we can add and delete users. Our first step will be to add these capabilities using cleartext passwords. Once we have this new functionality completed, we will make changes to hash the passwords.

Model

First, we must create a model for our User. Our model will reflect the columns in the users table (user_id, username, password, enabled) and the authorities associated with the user as stored in the authorities table (this will be represented by an ArrayList of Strings). Create a new Java class called User in the model package and insert the following code:

```
package com.swcguild.contactlistmvc.model;
import java.util.ArrayList;

public class User {
    private int id;
    private String username;
    private String password;
    private ArrayList<String> authorities = new ArrayList<>();

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public ArrayList<String> getAuthorities() {
        return authorities;
    }
    public void setAuthorities(ArrayList<String> authorities) {
        this.authorities = authorities;
    }
    public void addAuthority(String authority) {
        authorities.add(authority);
    }
}
```

DAO

Next, we must create a DAO to handle the database interaction for users. We will do this in three steps:

1. Create a Java Interface for the DAO
2. Create an implementation of the DAO that talk to our database
3. Configure Spring DI to instantiate our DAO and inject the JdbcTemplate

Create a new Java Interface called **UserDao** in the **dao** package and insert the following code. We are only going to handle creating and deleting Users, and leave editing Users as an exercise for the reader (hint: it will follow the same pattern as editing Contacts):

```
package com.swcguild.contactlistmvc.dao;

import com.swcguild.contactlistmvc.model.User;

public interface UserDao {

    public User addUser(User newUser);

    public void deleteUser(String username);

}
```

Create a new Java Class called **UserDaoDbImpl** in the **dao** package and insert the following code:

Notes:

1. User data is held in both the users table and the authorities table. Therefore, when creating or deleting users, we must update both tables.
2. When adding a user, we must first put an entry in the users table and then put an entry for each of the user's authorities into the authorities table.
3. When deleting a user, we must first delete all of the user's authorities and then delete in the user's entry in the users table.

```java
package com.swcguild.contactlistmvc.dao;

import com.swcguild.contactlistmvc.model.User;
import java.util.ArrayList;
import org.springframework.jdbc.core.JdbcTemplate;

public class UserDaoDbImpl implements UserDao {

    // #1 - We need to update both the users and authorities tables
    private static final String SQL_INSERT_USER =
            "insert into users (username, password, enabled) values (?, ?, 1)";
    private static final String SQL_INSERT_AUTHORITY =
            "insert into authorities (username, authority) values (?, ?)";
    private static final String SQL_DELETE_USER =
            "delete from users where username = ?";
    private static final String SQL_DELETE_AUTHORITIES =
            "delete from authorities where username = ?";

    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public User addUser(User newUser) {
        // #2 - First insert user data into the users table and then insert data into
        //      the authorities table (failing to do so will result in foreign key
        //      constraint errors)
        jdbcTemplate.update(SQL_INSERT_USER, newUser.getUsername(), newUser.getPassword());
        newUser.setId(jdbcTemplate.queryForObject("select LAST_INSERT_ID()", Integer.class));

        // now insert user's roles
        ArrayList<String> authorities = newUser.getAuthorities();
        for(String authority : authorities) {
            jdbcTemplate.update(SQL_INSERT_AUTHORITY, newUser.getUsername(), authority);
        }

        return newUser;
    }

    @Override
    public void deleteUser(String username) {
        // #3 - First delete all authorities for this user
        jdbcTemplate.update(SQL_DELETE_AUTHORITIES, username);
        // #3 - Second delete the user - failing to do so will result in foreign
        //      key constraint errors
        jdbcTemplate.update(SQL_DELETE_USER, username);
    }
}
```

Finally, we must make changes to **spring-persistence.xml** so that Spring will create our new DAO and inject a JdbcTemplate into it. Add the following bean definition to spring-persistence.xml:

```
<bean id="userDao"
      class="com.swcguild.contactlistmvc.dao.UserDaoDbImpl">
      <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

Controller

We will now create a controller to handle the endpoints for adding and deleting Users. For adding Users, we will follow the pattern established earlier for adding Contact — we will have one endpoint that just displays the form for adding a User and we will have another endpoint that receives the POSTed data from the form and calls the DAO to create the new User.

For deleting Users, we will follow the pattern established earlier for deleting Contacts — we will have a page that lists the Users in the system and each User entry will have a Delete link next to it. The Delete link will be built so that it has the ID of the User and is connected to a controller endpoint that simply deletes the specified User from the database and then returns the view that lists the Users.

Create a new class called **UserController** in the **controller** package and enter the following code:

Notes:

1. This endpoint displays all the Users in the system.
2. This endpoint displays the Add User form.
3. This endpoint processes the data from the form, creates a new User object, and calls the DAO to create a new User in the database.
4. We are not using Spring Form tags in this example so we must get the form data from the HttpServletRequest object.
5. All Users in our system have the role ROLE_USER. Only add the role ROLE_ADMIN if the isAdmin checkbox is checked on the form.
6. This endpoint calls the DAO to delete the User specified by the username request parameter.

```java
package com.swcguild.contactlistmvc.controller;

import com.swcguild.contactlistmvc.dao.UserDao;
import com.swcguild.contactlistmvc.model.User;
import java.util.List;
import java.util.Map;
import javax.inject.Inject;
import javax.servlet.http.HttpServletRequest;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class UserController {

    private UserDao dao;

    @Inject
    public UserController(UserDao dao) {
        this.dao = dao;
    }

    // #1 - This endpoint retrieves all users from the database and puts the
    //      List of users on the model
    @RequestMapping(value="/displayUserList", method=RequestMethod.GET)
    public String displayUserList(Map<String, Object> model) {
        List users = dao.getAllUsers();
        model.put("users", users);
        return "displayUserList";
    }
    // #2 - This endpoint just displays the Add User form
    @RequestMapping(value="/displayUserForm", method=RequestMethod.GET)
    public String displayUserForm(Map<String, Object> model) {
        return "addUserForm";
    }
    // #3 - This endpoint processes the form data and creates a new User
    @RequestMapping(value="/addUser", method=RequestMethod.POST)
    public String addUser(HttpServletRequest req) {
        User newUser = new User();
        // #4 - This example uses a plain HTML form so we must get values from the request
        newUser.setUsername(req.getParameter("username"));
        newUser.setPassword(req.getParameter("password"));
        // #5 - All users have ROLE_USER, only add ROLE_ADMIN if the isAdmin box is checked
        newUser.addAuthority("ROLE_USER");
        if (null != req.getParameter("isAdmin")) {
            newUser.addAuthority("ROLE_ADMIN");
        }

        dao.addUser(newUser);

        return "redirect:displayUserList";
    }
    // #6 - This endpoint deletes the specified User
    @RequestMapping(value="/deleteUser", method=RequestMethod.GET)
    public String deletUser(@RequestParam("username") String username,
                            Map<String, Object> model) {
        dao.deleteUser(username);
        return "redirect:displayUserList";
    }
}
```

Views

We need to create two additional views to complete the Add and Delete User features of the application. The first view will list all of the users in the system, and each User entry will have a Delete link that allows that Users to be deleted from the system. There will also be a link to the Add User view on this page. Create a new JSP called **displayUserList.jsp** (make sure it is in the **jsp** folder) and enter the following code:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <title>Users</title>
    </head>
    <body>
        <h1>Users</h1>
        <a href="displayUserForm">Add a User</a><br/>
        <hr/>

        <c:forEach var="user" items="${users}">
            <c:out value="${user.username}"/> |
            <a href="deleteUser?username=${user.username}">Delete</a><br/><br/>
        </c:forEach>
    </body>
</html>
```

The second view will display the Add User form. Create a new JSP called **addUserForm.jsp** (make sure it is in the **jsp** folder) and enter the following code:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <title>Users</title>
    </head>
    <body>
        <h1>Add User Form</h1>

        <form method="POST" action="addUser">
            Username: <input type="text" name="username"/><br/>
            Password:&nbsp; <input type="password" name="password"/><br/>
            Admin User? <input type="checkbox" name="isAdmin" value="yes"/> <br/>
            <input type="submit" value="Add User"/><br/>
        </form>
    </body>
</html>
```

At this point, you should be able to Add and Delete Users in the system. For now, these Users will all have cleartext passwords, but we'll fix that in the next section.

Password Hashing

For new systems, the recommended hashing algorithm is BCrypt. BCrypt uses a random salt and key stretching in order to defend against brute force, dictionary, and precomputed rainbow table attacks. Spring Security also includes other algorithms in case you are working on a system that hashes passwords using something other than BCrypt. We will cover BCrypt in this example.

There are two parts involved in implementing password hashing in an application:

1. Hashing the submitted password when a user logs in and then checking the hashed value against the hashed value stored in the database.
2. Hashing the password of newly-created users and then storing the hashed password value in the database.

The first step will be handled by Spring Security and requires only a configuration change. We must handle the second step ourselves and this will require a small code change in our Add User process.

Configuration

We need to make two changes to our **spring-security.xml** file. First, we will define a bean for our BCryptPasswordEncoder and then we will modify our authentication-provider configuration so that it will use the BCryptPasswordEncoder when checking login credentials. Modify your **spring-security.xml** file so that it looks like this:

```
<!-- #1 - Spring Security XML namespace configuration -->
<beans:beans xmlns="http://www.springframework.org/schema/security"
             xmlns:beans="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-3.0.xsd
  http://www.springframework.org/schema/security
  http://www.springframework.org/schema/security/spring-security-3.1.xsd">

    <beans:bean id="webexpressionHandler"
           class=
"org.springframework.security.web.access.expression.DefaultWebSecurityExpressionHandler"/>
    <!-- PASSWORD ENCODER BEAN CHANGES START -->
    <beans:bean id="encoder"
                class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>
    <!-- PASSWORD ENCODER BEAN CHANGES END -->
    <!-- #2 - Make sure we don't need authorization to get to the login page -->
    <http pattern="/spring/login" security="none" />
    <!-- #3 - Authentication/login form and protected endpoint configuration -->
    <http auto-config="true" use-expressions="false">
        <!-- #3a - Login via html form, use Spring to do the security check -->
        <!-- #3b - Use the login page at this endpoint -->
        <!-- #3c - Redirect here is login fails -->
        <form-login login-processing-url="/j_spring_security_check"
                    login-page="/spring/login"
                    authentication-failure-url="/spring/login?login_error=1"/>
        <!-- #3d - Go back to login page when user logs out -->
        <logout logout-success-url="/spring/login" />
        <!-- #3e - Access to these endpoints require admin role -->
        <intercept-url pattern="/spring/editContactForm" access="ROLE_ADMIN" />
        <intercept-url pattern="/spring/displayNewContactForm" access="ROLE_ADMIN" />
        <intercept-url pattern="/spring/addContact" access="ROLE_ADMIN" />
        <intercept-url pattern="/spring/deleteContact" access="ROLE_ADMIN" />
        <intercept-url pattern="/spring/updateContact" access="ROLE_ADMIN" />
        <!-- #3f - Access to all other controller endpoints require user role -->
        <intercept-url pattern="/spring/**" access="ROLE_USER" />
    </http>
    <!-- #4 - Authentication Manager config -->
    <authentication-manager>
        <!-- #4a - Authentication Provider - we’re using the JDBC service -->
        <authentication-provider>
           <!-- AUTHENTICATION-PROVIDER CHANGES START -->
            <password-encoder ref="encoder"/>
           <!-- AUTHENTICATION-PROVIDER CHANGES END -->
            <!-- #4b - Tells Spring Security where to look for user information -->
            <!--       We use the dataSource defined in spring-persistence.xml  -->
            <!--       and we give Spring Security the query to use to lookup   -->
            <!--       the user’s credentials (get the password from the users  -->
            <!--       tables and get the roles from the authorities table      -->
            <jdbc-user-service id="userService"
                               data-source-ref="dataSource"
                               users-by-username-query=
          "select username, password, enabled from users where username=?"
                               authorities-by-username-query=
          "select username, authority from authorities where username=?" />
        </authentication-provider>
    </authentication-manager>
</beans:beans>
```

Code

We must also make changes to our code. First, we will update the UserController so that it hashes the password for new Users before storing the values in the database. Update your UserController to look like this:

Notes:

1. Create a private PasswordEncoder class variable. Make sure you import the `org.springframework.security.crypto.password.PasswordEncoder`. There is another PasswordEncoder interface in the Spring Security library but it has been deprecated.
2. Have Spring use auto-wired constructor injection to hand our class an instance of the PasswordEncoder. The BCryptPasswordEncoder spring-security.xml bean entry that we created in the last step will be handed to us at runtime (BCryptPasswordEncoder implements PasswordEncoder).
3. Get the cleartext password from the form, use the PasswordEncoder to hash the password, set the hashed value on the newly-created User object, and then save the User object to the database.

```java
package com.swcguild.contactlistmvc.controller;

import com.swcguild.contactlistmvc.dao.UserDao;
import com.swcguild.contactlistmvc.model.User;
import java.util.List;
import java.util.Map;
import javax.inject.Inject;
import javax.servlet.http.HttpServletRequest;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class UserController {

    private UserDao dao;
    // #1 - PasswordEncoder interface
    private PasswordEncoder encoder;

    // #2 - Inject a PasswordEncoder
    @Inject
    public UserController(UserDao dao, PasswordEncoder encoder) {
        this.dao = dao;
        this.encoder = encoder;
    }

    @RequestMapping(value="/displayUserList", method=RequestMethod.GET)
    public String displayUserList(Map<String, Object> model) {
        List users = dao.getAllUsers();
        model.put("users", users);
        return "displayUserList";
    }

    @RequestMapping(value="/displayUserForm", method=RequestMethod.GET)
    public String displayUserForm(Map<String, Object> model) {
        return "addUserForm";
    }

    @RequestMapping(value="/addUser", method=RequestMethod.POST)
    public String addUser(HttpServletRequest req) {
        User newUser = new User();
        newUser.setUsername(req.getParameter("username"));
        // #3 - Hash the password and then set it on the User object before saving it
        String clearPw = req.getParameter("password");
        String hashPw = encoder.encode(clearPw);
        newUser.setPassword(hashPw);
        newUser.addAuthority("ROLE_USER");
        if (null != req.getParameter("isAdmin")) {
            newUser.addAuthority("ROLE_ADMIN");
        }

        dao.addUser(newUser);
        return "redirect:displayUserList";
    }
    @RequestMapping(value="/deleteUser", method=RequestMethod.GET)
    public String deletUser(@RequestParam("username") String username,
                            Map<String, Object> model) {
        dao.deleteUser(username);
        return "redirect:displayUserList";
    }
}
```

Update Existing Passwords

Now that the system is configured to use hashed passwords, we must update the existing passwords in our database — if we don't, we'll be locked out of the system.

Create a new Java class called **PWEnc** in the controller package and insert the following code:

```
package com.swcguild.contactlistmvc.controller;

import org.springframework.security.authentication.encoding.Md5PasswordEncoder;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

public class PWEnc {

    public static void main(String[] args) {
        String clearTxtPw = "password";
        // BCrypt
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
        String hashedPw = encoder.encode(clearTxtPw);
        System.out.println(hashedPw);
    }
}
```

Paste in the values for each existing password in your database (one by one) and run this program (right-click the file and select Run File). The hashed password value will appear in the output window. Copy the hashed value and replace the clear text password in the database with this new one.

Update Endpoint Security Configuration

Finally, we have to protect the new endpoints (Diplay, Add, and Delete Users). To do this, we must update the **spring-security.xml** file.

Add the following intercept-url entries right after the entry for updateContact:

```
    <intercept-url pattern="/spring/displayUserForm" access="ROLE_ADMIN" />
    <intercept-url pattern="/spring/addUser" access="ROLE_ADMIN" />
    <intercept-url pattern="/spring/deleteUser" access="ROLE_ADMIN" />
```

Wrap-up

That does it for adding hashed passwords to our system. Here's what we covered in this step:

1. Added the ability to display the Users in the system
2. Added the ability to create Users and set user passwords and roles
3. Added the ability to delete Users from the system
4. Configured our system to use the BCrypt hashing algorithm for all passwords
5. Converted all existing passwords in the system from cleartext to hashed values
6. Protected the new endpoints (to Display, Add, and Delete Users) using Spring Security