Adding Spring Security to CRM App

Introduction

In this lecture, you will learn how to integrate our CRM app with Spring Security.

GOALS

- Add login page to CRM app
- Restrict URL access based on roles
- Display content based on roles
- Store user id and encrypted password in the database

TECHNICAL APPROACH

- Use Spring all Java configuration (no xml)
- Use Maven for project dependency management

SECURITY INTEGRATION

Our CRM app will have a new login page. Only users with valid id/passwords will be able to login. The app will also restrict access based on roles.

- Employee role: users in this role will only be allowed to list customers.
- Manager role: users in this role will be allowed to list, add and update customers.
- Admin role: users in this role will be allowed to list, add, update and delete customers.

FULL SOURCE CODE

The full source code is available at this link:

http://www.luv2code.com/spring-crm-with-security

In the following sections, I'll walk through the source code and explain how it works.

Prerequisites

This tutorial assumes that you have already completed the Spring Security videos in the Spring-Hibernate course. This includes the Spring Security videos for JDBC authentication for plain-text passwords and encrypted passwords.

Overview of Steps

0. Download and Import the code
1. Run database scripts
2. Maven POM file
3. Database configuration file
4. Spring MVC Web Configuration Java code (all java, no xml)
5. Spring Security Configuration Java code (all java, no xml)
6. Display content based on roles

0. Download   and Import the Code

The code is provided as a full Maven project and you can easily import it into Eclipse.

TO IMPORT THE CODE TO ECLIPSE:

1. If you haven't done so already, download from:
http://www.luv2code.com/spring-crm-with-security
2. Unzip the file
3. In Eclipse, select **Import > Existing Maven Projects ...**
4. Browse to directory where you unzipped the code.
5. Click OK buttons etc to import code

REVIEW THE PROJECT STRUCTURE.

Make note of the following directories

- /src/main/java: contains the main java code
- /src/main/resources: contains the database configuration file
- /src/main/webapp: contains the web files (jsp, css etc)
- /sql-scripts: the database scripts for the app (customers and security accounts)

1. Run database scripts

In order to make sure we are all on the same page in terms of database schema names and user accounts/passwords, let's run the same database scripts.

MySQL WORKBENCH

In MySQL workbench, run the following database script:

```
/sql-scripts/customer-tracker.sql
```

This script will creates the database schema: web_customer_tracker. It will add test customers in the database.

Run the following script:

```
/sql-scripts/setup-spring-security-bcrypt-demo-database.sql
```

This script creates the database schema: **spring_security_demo_bcrypt**. The script creates the user accounts with encrypted passwords. It also includes the user roles.

User ID	Password	Roles
john	fun123	EMPLOYEE
mary	fun123	EMPLOYEE, MANAGER
susan	fun123	EMPLOYEE, ADMIN

2. Maven POM file

The Maven POM file, pom.xml contains the dependencies required for this project. Open the file and review the dependencies.

3. Database configuration files

3.1 Customer Tracker Configuration file

The database configuration file for the customer tracker is located at `/src/main/resources/persistence-mysql.properties`

When Maven performs a build, this file is copied with the WEB-INF/classes directory. We'll configure our Spring code to read this file from the classpath. The code will use information from this file to set up the database connection pool.

File: persistence-mysql.properties

```
#
# JDBC connection properties
#
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/web_customer_tracker?useSSL=false
jdbc.user=springstudent
jdbc.password=springstudent

#
# Connection pool properties
#
connection.pool.initialPoolSize=5
connection.pool.minPoolSize=5
connection.pool.maxPoolSize=20
connection.pool.maxIdleTime=3000

#
# Hibernate properties
#
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
hibernate.packagesToScan=com.luv2code.springdemo.entity
```

3.2 Spring Security Configuration file

The database configuration file for the security database is located at /src/main/resources/security-persistence-mysql.properties

This file is similar to the previous, except that is points to the Spring security database schema: spring_security_demo_bcrypt. This schema was created by the earlier database scripts. It holds the user ids, passwords and roles.

File: security-persistence-mysql.properties

```
#
# SPRING SECURITY CONFIGS
#
# JDBC connection properties
#
security.jdbc.driver=com.mysql.jdbc.Driver
security.jdbc.url=jdbc:mysql://localhost:3306/spring_security_demo_bcrypt?useSSL=false
security.jdbc.user=springstudent
security.jdbc.password=springstudent

#
# Connection pool properties
#
security.connection.pool.initialPoolSize=5
security.connection.pool.minPoolSize=5
security.connection.pool.maxPoolSize=20
security.connection.pool.maxIdleTime=3000
```

4. Spring MVC Web Configuration Java code (all java, no xml)

Instead of using web.xml to bootstrap the Spring MVC configuration. We'll make use of pure Java. This code is contained in the following files:

package: com.luv2code.springdemo.config

File: MySpringMvcDispatcherServletInitializer

This file specifies the configuration for the application. It also maps the Spring MVC dispatcher servlet to the root path.

```
	@Override
	protected Class<?>[] getServletConfigClasses() {
			return new Class[] { DemoAppConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
			return new String[] { "/" };
	}
```

I explained this file in detail earlier in the course. If you need a refresh, please see this video.

Section 47

https://www.udemy.com/spring-hibernate-tutorial/learn/v4/t/lecture/8355872?start=0 package: com.luv2code.springdemo.config

File: DemoAppConfig.java

The DemoAppConfig.java contains the configuration for the various beans needs for Hibernate DB access. This code was traditionally in a single xml file, spring-mvc-crud-servlet.xml.

At the beginning of the file, there are annotations for configuration

```
@Configuration
@EnableWebMvc
@EnableTransactionManagement
@ComponentScan("com.luv2code.springdemo")
@PropertySource({ "classpath:persistence-mysql.properties",
"classpath:security-persistence-mysql.properties" })
public class DemoAppConfig implements WebMvcConfigurer {
...
}
```

@Configuration: tells Spring this is a configuration class

@EnableWebMvc: is similar to the XML config <mvc:annotation-driven />

@EnableTransactionManagement: adds transaction manager support

@ComponentScan("com.luv2code.springdemo"): where to scan for components, recursively

@PropertySource({ "classpath:persistence-mysql.properties", "classpath:security-persistence-mysql.properties" }): the configuration files to load.

We'll use this to load the database configuration file from the classpath. As mentioned earlier, when Maven performs a build, the properties is copied with the WEB-INF/classes directory. The WEB-INF/classes directory is one of the classpath directories for Java web apps.

Note: I explained @Configuration, @EnableWebMvc and @ComponentScan earlier in the course. If you need a refresh, please see this video.

Section 47,

https://www.udemy.com/spring-hibernate-tutorial/learn/v4/t/lecture/8355870?start=0

Ok, let's continue to walk through the code.

We have this snippet code

```
@Autowired
private Environment env;
```

This is a special helper class provided by Spring. The Environment object, env, will be loaded with the properties file from the annotation: @PropertySource({ "classpath:persistence-mysql.properties" })

We'll use this later in the class to read the configs for JDBC, Hibernate and connection pooling.

Next, we define code for a view resolver. This is similar to what we had in the XML world.

```
    // define a bean for ViewResolver
        @Bean
    public ViewResolver viewResolver() {

             InternalResourceViewResolver viewResolver = new
InternalResourceViewResolver();

          viewResolver.setPrefix("/WEB-INF/view/");
          viewResolver.setSuffix(".jsp");

       return viewResolver;
    }
```

Next, we write code to create the datasource bean. This code reads the Environment env object to get the data. It then uses the data from env to set up configs for JDBC and connection pooling

```
    @Bean
    public DataSource myDataSource() {

       // create connection pool
        ComboPooledDataSource myDataSource = new ComboPooledDataSource();

           // set the jdbc driver
           try {
            myDataSource.setDriverClass("com.mysql.jdbc.Driver");
           }
           catch (PropertyVetoException exc) {
           throw new RuntimeException(exc);
           }
           // for sanity's sake, let's log url and user ... just to make sure
we are reading the data
        logger.info("jdbc.url=" + env.getProperty("jdbc.url"));
           logger.info("jdbc.user=" + env.getProperty("jdbc.user"));

           // set database connection props
        myDataSource.setJdbcUrl(env.getProperty("jdbc.url"));
        myDataSource.setUser(env.getProperty("jdbc.user"));
        myDataSource.setPassword(env.getProperty("jdbc.password"));

           // set connection pool props
myDataSource.setInitialPoolSize(Integer.parseInt(env.getProperty("connection.p
ool.initialPoolSize")));
myDataSource.setMinPoolSize(Integer.parseInt(env.getProperty("connection.pool.
minPoolSize")));
myDataSource.setMaxPoolSize(Integer.parseInt(env.getProperty("connection.pool.
maxPoolSize")));
myDataSource.setMaxIdleTime(Integer.parseInt(env.getProperty("connection.pool.
maxIdleTime")));
           return myDataSource;
    }
```

The next method handles the Hibernate properties.

```
    private Properties getHibernateProperties() {
            // set hibernate properties
            Properties props = new Properties();
          props.setProperty("hibernate.dialect",
env.getProperty("hibernate.dialect"));
        props.setProperty("hibernate.show_sql",
env.getProperty("hibernate.show_sql"));
            return props;
    }
```

The next method creates the Hibernate session factory based on the datasource and configuration properties

```
    @Bean
    public LocalSessionFactoryBean sessionFactory(){
              // create session factorys
          LocalSessionFactoryBean sessionFactory = new
LocalSessionFactoryBean();
              // set the properties
          sessionFactory.setDataSource(myDataSource());
sessionFactory.setPackagesToScan(env.getProperty("hibernate.packagesToScan"));
          sessionFactory.setHibernateProperties(getHibernateProperties());
              return sessionFactory;
    }
```

The next method creates the securityDataSource bean. This points to the security database for user id, passwords and roles.

```
    // define a bean for security data source
        @Bean
    public DataSource securityDataSource() {

            // create connection pool
        ComboPooledDataSource securityDataSource
                                    = new ComboPooledDataSource();

            // set the jdbc driver class

            try {

securityDataSource.setDriverClass(env.getProperty("security.jdbc.driver"));
            } catch (PropertyVetoException exc) {
            throw new RuntimeException(exc);
            }

            // log the connection props
            // for sanity's sake, log this info
            // just to make sure we are REALLY reading data from properties
file

        logger.info(">>> security.jdbc.url=" +
env.getProperty("security.jdbc.url"));
        logger.info(">>> security.jdbc.user=" +
env.getProperty("security.jdbc.user"));


            // set database connection props

        securityDataSource.setJdbcUrl(env.getProperty("security.jdbc.url"));
        securityDataSource.setUser(env.getProperty("security.jdbc.user"));

securityDataSource.setPassword(env.getProperty("security.jdbc.password"));

            // set connection pool props

      securityDataSource.setInitialPoolSize(
                getIntProperty("security.connection.pool.initialPoolSize"));

        securityDataSource.setMinPoolSize(
                getIntProperty("security.connection.pool.minPoolSize"));

            securityDataSource.setMaxPoolSize(
                getIntProperty("security.connection.pool.maxPoolSize"));

        securityDataSource.setMaxIdleTime(
                getIntProperty("security.connection.pool.maxIdleTime"));

            return securityDataSource;
    }
```

The next method configures the Hibernate transaction manager.

```
    @Bean
    @Autowired
    public HibernateTransactionManager transactionManager(SessionFactory
sessionFactory) {

            // setup transaction manager based on session factory
        HibernateTransactionManager txManager = new
HibernateTransactionManager();
        txManager.setSessionFactory(sessionFactory);

            return txManager;
    }
```

Finally, the app is going to use static web resources such as css, images, js etc. So we add a resource handler.

```
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
           registry
          .addResourceHandler("/resources/**")
          .addResourceLocations("/resources/");
    }
```

5. Spring Security Configuration Java code (all java, no xml)

Next, we need to configure Spring security. There are two important classes. DemoSecurityConfig.java and SecurityWebApplicationInitializer.java

package: com.luv2code.springdemo.config
File: SecurityWebApplicationInitializer

This file is required in order to register the Spring Security filters. We covered this in the Spring Security section. No changes to this file:

```
package com.luv2code.springdemo.config;

import
org.springframework.security.web.context.AbstractSecurityWebApplicationInitial
izer;

public class SecurityWebApplicationInitializer
                        extends AbstractSecurityWebApplicationInitializer {

}
```

package: com.luv2code.springdemo.config

File: DemoSecurityConfig.java

This file configures our users and also restricts access to the app based on roles. This file is mostly the same from previous spring security videos. There are just minor changes on which URLs to restrict access for.

We make use of the security datasource to access user accounts stored in the database.

```
@Configuration
@EnableWebSecurity
public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {

    // add a reference to our security data source
    @Autowired
    private DataSource securityDataSource;

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

             // use jdbc authentication ... oh yeah!!!
             auth.jdbcAuthentication().dataSource(securityDataSource);

    }
```

Then, we'll restrict access to users based on roles.

In this app, to make things exciting, we have the following app requirements:

- Employee role: users in this role will only be allowed to list customers.
- Manager role: users in this role will be allowed to list, add and update customers.
- Admin role: users in this role will be allowed to list, add, update and delete customers.

So, these requirements translated into the following code

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

    http.authorizeRequests()
        .antMatchers("/customer/showForm*").hasAnyRole("MANAGER", "ADMIN")
        .antMatchers("/customer/save*").hasAnyRole("MANAGER", "ADMIN")
        .antMatchers("/customer/delete").hasRole("ADMIN")
        .antMatchers("/customer/**").hasRole("EMPLOYEE")
        .antMatchers("/resources/**").permitAll()
      .and()
        .formLogin()
            .loginPage("/showMyLoginPage")
            .loginProcessingUrl("/authenticateTheUser")
            .permitAll()
       .and()
        .logout().permitAll()
       .and()
         .exceptionHandling().accessDeniedPage("/access-denied");

}
```

The ordering of antMatchers is important. You should list the most specific url patterns first and then add the more general patterns later. If you don't follow this approach, then you will not have proper access restrictions.

Everything else remains the same for login/logout etc.

6. Display content based based on roles

Based on the previous security configs in the CRM app, only certain users can perform certain operations such as Add, Update and Delete.

So we can change the view page to only display content based on the user roles.

For example, the view page list-customers.jsp

Only show the "Add Customer" button, if the user is a manager or admin.

We can make use of this code:

```
    <security:authorize access="hasAnyRole('MANAGER', 'ADMIN')">

            <!-- put new button: Add Customer -->

            <input type="button" value="Add Customer"
                onclick="window.location.href='showFormForAdd'; return false;"
                class="add-button"
            />

    </security:authorize>
```

Only show the "Update" link, if the user is a manager or admin.

```
    <security:authorize access="hasAnyRole('MANAGER', 'ADMIN')">
            <!-- display the update link -->
            <a href="${updateLink}">Update</a>
    </security:authorize>
```

Only show the "Delete" link, if the user is an admin.

Here's the code

```
    <security:authorize access="hasAnyRole('ADMIN')">
            <a href="${deleteLink}"
            onclick="if (!(confirm('Are you sure you want to delete this
customer?'))) return false">Delete</a>
    </security:authorize>
```

I'm assuming that you were able to run the CRM app earlier in the course and you were able to run the Spring Security simple web examples.

If so, then you can run the application on your server. Everything is ready to go!

When you run the app, you will need to log in with one of the accounts:

1. john
    - user id: john
    - password: fun123
    - Role: employee

2. mary
    - user id: mary
    - password: fun123
    - Role: employee, manager

3. susan
    - user id: susan
    - password: fun123
    - Role: employee, admin

Files

This section just contains details on which files are new and which ones remained the same.

Here's a list of files that were added to CRM project for all Java Configuration (no xml)

Java Configuration (no xml)

src/main/java/com/luv2code/springdemo/config

- DemoAppConfig.java
- MySpringMvcDispatcherServletInitializer.java

Properties file

/src/main/resources/persistence-mysql.properties

Here's a list of files that were added to CRM project for Spring security

Spring Security Configuration

src/main/java/com/luv2code/springdemo/config

- DemoSecurityConfig.java
- SecurityWebApplicationInitializer

Spring Security LoginController

src/main/java/com/luv2code/springdemo/controller

- LoginController

View pages

src/main/webapp/WEB-INF/view

- fancy-login.jsp
- access-denied.jsp

Spring Security RegistrationController

src/main/java/com/luv2code/springdemo/controller

- RegistrationController

Spring Security Registration Pages

src/main/webapp/WEB-INF/view

- registration-form.jsp
- registration-confirmation.jsp

No changes were made to: entity, dao, service, CustomerController

Full Source Code

The full source code is available at this link:

http://www.luv2code.com/spring-crm-with-security

That's it!