

***Spring Security***

**Training Assignments**

|  |  |
| --- | --- |
| Document Code | 25e-BM/HR/HDCV/FSOFT |
| Version | 1.1 |
| Effective Date | 20/11/2012 |

**HCM, 05/2020**

RECORD OF CHANGES

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Effective Date | Change Description | Reason | Reviewer | Approver |
|  | 21/May/2020 | Create a new Lab | Create new |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Contents

[Objectives 4](#_Toc41333458)

[Business needs 4](#_Toc41333459)

[Working Environments 4](#_Toc41333460)

[Product Architecture 5](#_Toc41333461)

[Technologies 5](#_Toc41333462)

[Database Relationship 5](#_Toc41333463)

[Assignment Descriptions 7](#_Toc41333464)

[Descriptions and Guidelines: 10](#_Toc41333465)

|  |  |
| --- | --- |
|  | **CODE: JSFW.L.L103 (Blogs)**  **TYPE: LONG**  **LOC: N/A**  **DURATION: 360 MINUTES** |

# **Objectives**

After finishing the following exercises, trainees will:

* Understand Spring Core (IoC, DI)
* Practice with JDBC Templates to connect database in spring
* Use JPA for object-relational mapping
* Practice with the Spring MVC framework architecture
* Know how to write a Web application with Spring MVC Framework and Spring Data JPA
* Know how to write a Web application with Spring Security

# **Business needs**

Create a Web application based on HTML, CSS, JavaScript, Ajax, Servlet, JSP, Spring framework

The blog system that we are going to develop is a single user system. The owner of the system will be able to perform the following actions:

* Login and logout
* Create, update and delete posts
* Published, unpublished and archive posts
* Approve and delete comments

All other users are guest users who can perform the following actions:

* Read posts
* Create comments

Additional Requirements for this system include:

* The homepage of the system should display a list of the most recent posts.
* If a page contains more than 10 posts, they should be displayed in pages.
* The system should display a post together with its comments.
* The system should be able to list posts with a specified tag.
* The system should show a cloud of tags indicating their use frequencies.
* The system should show a list of most recent comments.

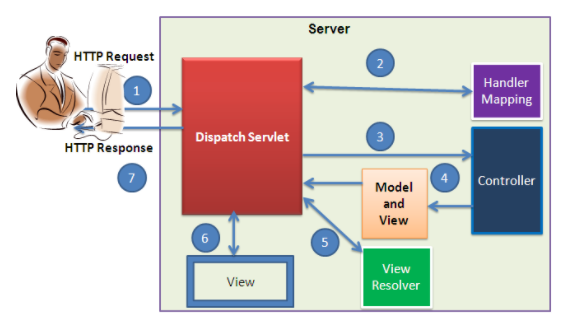
# Working Environments

* JDK 1.8 +
* Maven 3+
* Eclipse IDE or STS,
* MySQL DB Server/MySQL,
* Apache TomCat 7 or later,
* Internet connection,

Delivery: Source code, deployment and testing, reviewing evidences packaged in a compress archive.

# Product Architecture

Web applications are by nature distributed applications, meaning that they are programs that run on more than one computer and communicate through a network or server. Specifically, web applications are accessed with a web browser and are popular because of the ease of using the browser as a user client.



The product is implemented using MVC Pattern base on Spring Web MVC.

# Technologies

The product implements one or more technology:

* HTML & CSS
* JavaScript & Ajax
* MVC & JSP Model
* Spring Framework: IoC, Bean, SpEL, Spring Data JPA, Interceptor, Validation, Spring Sercutity.

# Database Relationship

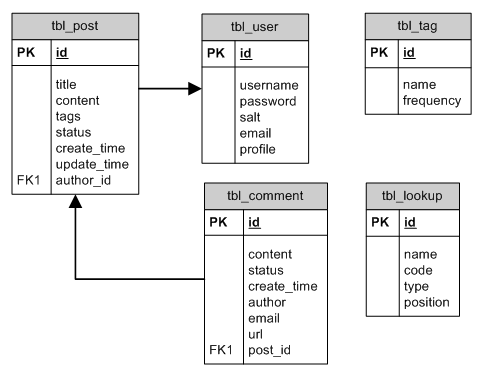
Based on the analysis of the requirements, we decide to use the following database tables to store the persistent data for our blog application:

* ***tbl\_user*** stores the user information, including username and password.
* ***tbl\_post*** stores the blog post information. It mainly consists of the following columns:
* title: required, title of the post;
* content: required, body content of the post which uses the [Markdown format](http://daringfireball.net/projects/markdown/syntax);
* status: required, status of the post, which can be one of following values:

1. meaning the post is in draft and is not visible to public;
2. meaning the post is published to public;
3. meaning the post is outdated and is not visible in the post list (still accessible individually, though).

* tags: optional, a list of comma-separated words categorizing the post.
* ***tbl\_comment*** stores the post comment information. Each comment is associated with a post and mainly consists of the following columns:
* author: required, the author name;
* email: required, the author email;
* url: optional, the author website URL;
* content: required, the comment content in plain text format.
* status: required, status of the comment, which indicates whether the comment is approved (value 2) or not (value 1).
* ***tbl\_tag*** stores post tag frequency information that is needed to implement the tag cloud feature. The table mainly contains the following columns:
* name: required, the unique tag name;
* frequency: required, the number of times that the tag appears in posts.
* ***tbl\_lookup*** stores generic lookup information. It is essentially a map between integer values and text strings. The former is the data representation in our code, while the latter is the corresponding presentation to end users. For example, we use integer 1 to represent the draft post status and stringDraft to display this status to end users. This table mainly contains the following columns:
* name: the textual representation of the data item that is to be displayed to end users;
* code: the integer representation of the data item;
* type: the type of the data item;
* position: the relative display order of the data item among other items of the same type.

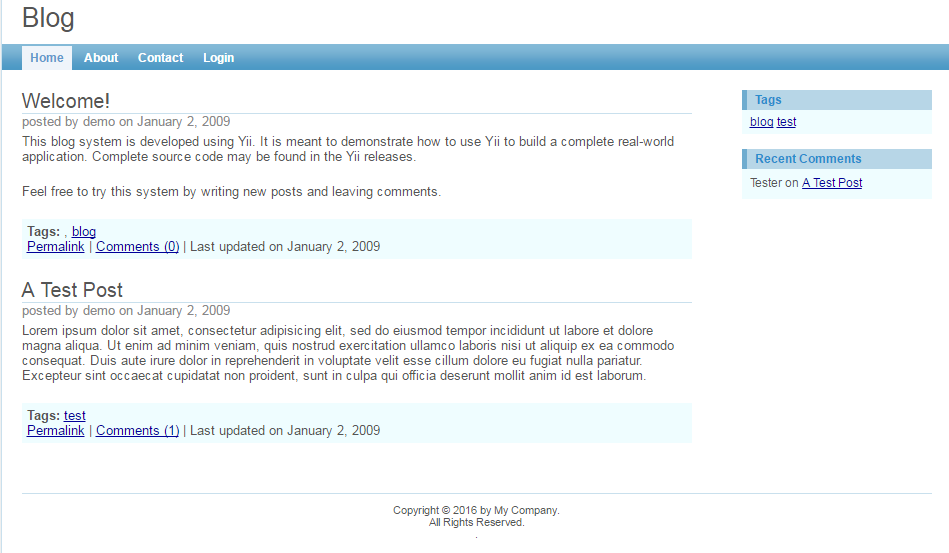
The following entity-relation (ER) diagram shows the table structure and relationships about the above tables



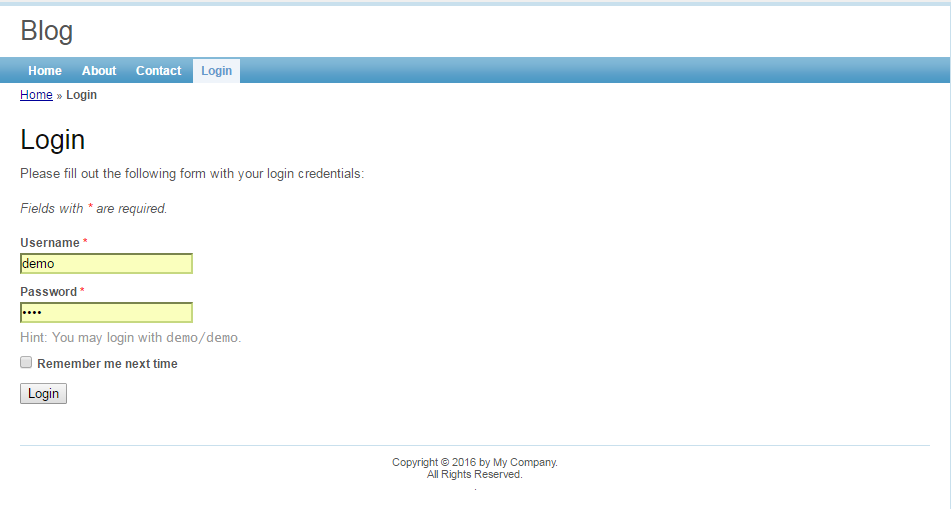
# Assignment Descriptions

1. ***Create the pages as below:***

* Home page:



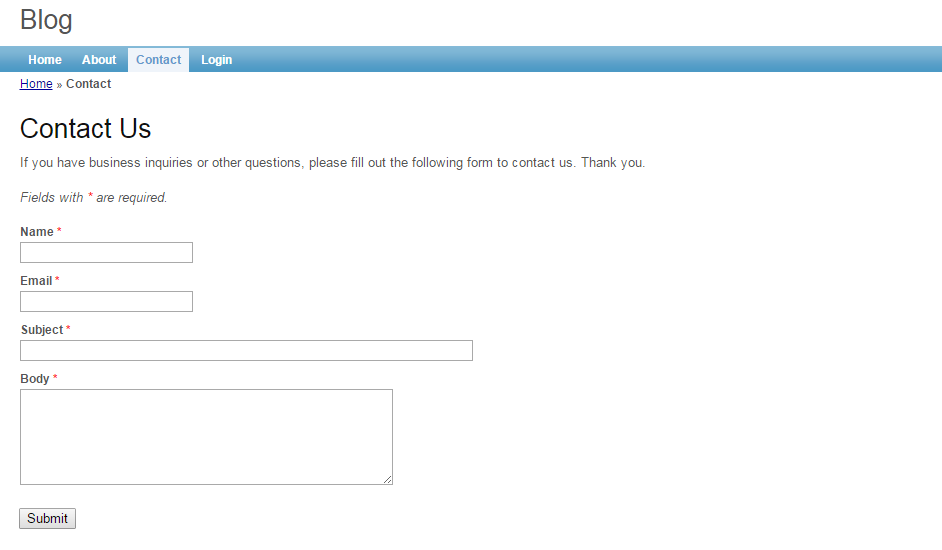
* Login page:



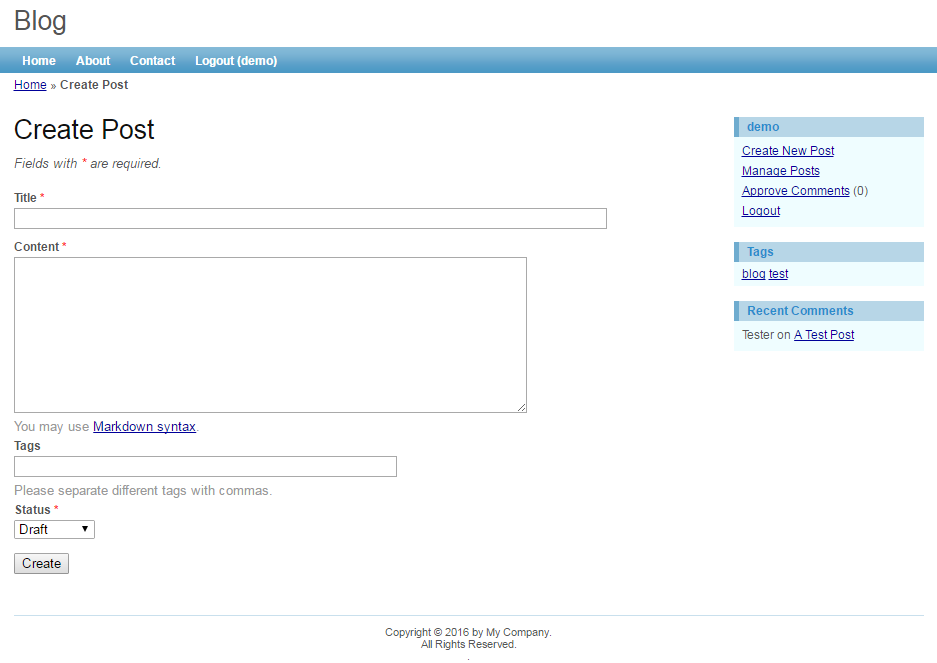
* About page



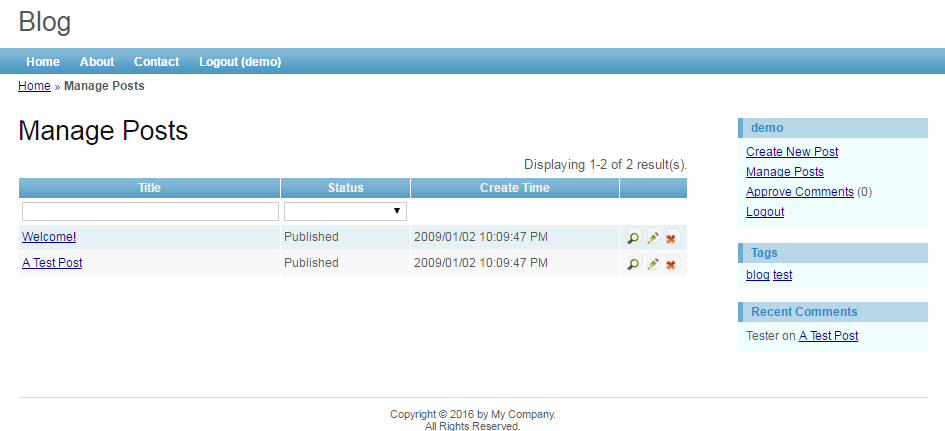
* Contact Page



* Create post

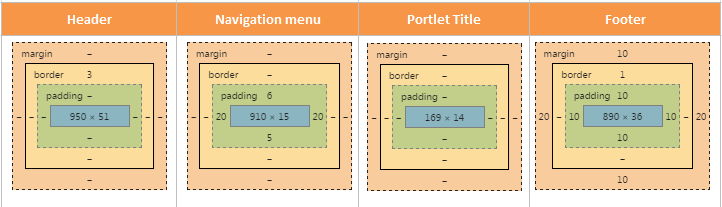


* Manage post page



1. **Screen Requirement:**

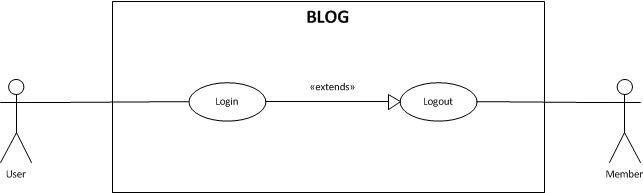
CSS Box Model as below:



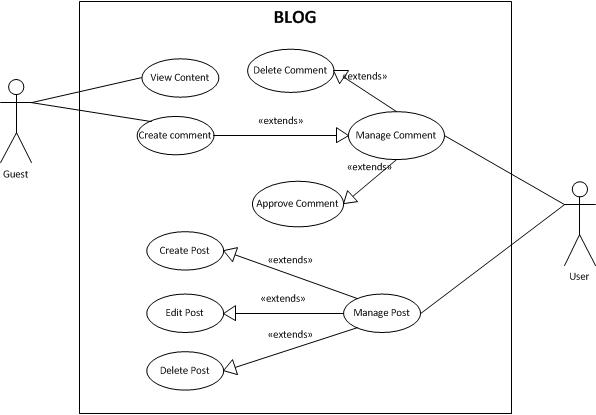
* Let design based The screen illustrated.
* Add User must be a popup.
* Placeholder text in inputs has a light gray color. You need to define the placeholder for input items on the above screen.
* Add a new file fStyle.css to customize your screen, it will basd on convention of CSS style.
* The **Index** page must contain 3 parts: Banner, Menu, Body

1. **Functional Requirement:**

As you may have found that the skeleton application already provides user authentication by checking if the username and password are both ***demo*** or ***admin***. In this section, we will modify the corresponding code so that the authentication is done against the User database table.



In our blog application, a post may be displayed among a list of posts or by itself. Managing posts mainly refers to listing posts in an administrative view that allows us to see posts with all statuses, updating them and deleting them. In this assignment, you must finish the feature in use case as below:



# Descriptions and Guidelines:

Step1: Jar dependencies

Add the following jars dependencies in your pom.xml file.

1. <dependency>
2. <groupId>org.springframework.security</groupId>
3. <artifactId>spring-security-web</artifactId>
4. <version>5.0.0.RELEASE</version>
5. </dependency>
6. <dependency>
7. <groupId>org.springframework.security</groupId>
8. <artifactId>spring-security-config</artifactId>
9. <version>5.0.0.RELEASE</version>
10. </dependency>

Step 2: Entity classes

Create two @Entity classes, named as User and Authorities, to map with database tables as follows.

**User.java**

1. **package** fa.training.model;
2. **import** java.util.Collection;
3. **import** java.util.HashSet;
4. **import** java.util.Set;
5. **import** javax.persistence.CascadeType;
6. **import** javax.persistence.Column;
7. **import** javax.persistence.Entity;
8. **import** javax.persistence.FetchType;
9. **import** javax.persistence.GeneratedValue;
10. **import** javax.persistence.GenerationType;
11. **import** javax.persistence.Id;
12. **import** javax.persistence.OneToMany;
13. **import** javax.persistence.Table;
14. /\*\*
15. \* Class model User, map with table tbl\_user
16. \* Declare fields of tbl\_user, getter and setter, Constructor.
17. \* **@author** KyLH
18. \*
19. \*/
20. @Entity
21. @Table(name = "tbl\_user")
22. public class User {
24. // Declare field id, auto generate.
25. @Id
26. @GeneratedValue(strategy = GenerationType.***IDENTITY***)
27. @Column(name = "id")
28. private int id;
30. // Declare field username
31. @Column(name = "username")
32. **private** String username;
33. // Declare field password
34. @Column(name = "password")
35. **private** String password;
36. // Declare field email
37. @Column(name = "email")
38. **private** String email;
39. // Declare field profile
40. @Column(name = "profile")
41. **private** String profile;
43. // Declare foreign key authorities table
44. @OneToMany(cascade = CascadeType.***ALL***, mappedBy = "user", fetch = FetchType.***EAGER***)
45. **private** Collection<Authorities> authorities;
47. // Declare foreign key post table
48. @OneToMany(cascade = CascadeType.***ALL***, mappedBy = "user", fetch = FetchType.***LAZY***)
49. **private** Set<Post> post = **new** HashSet<>();
51. /\*\*
52. \* constructor
53. \*/
54. **public** User() {
55. **super**();
56. // **TODO** Auto-generated constructor stub
57. }
59. /\*\*
60. \* constructor
61. \* **@param** id
62. \* **@param** username
63. \* **@param** password
64. \* **@param** email
65. \* **@param** profile
66. \* **@param** authorities
67. \*/
68. **public** User(**int** id, String username, String password, String email, String profile,
69. Set<Authorities> authorities) {
70. **super**();
71. **this**.id = id;
72. **this**.username = username;
73. **this**.password = password;
74. **this**.email = email;
75. **this**.profile = profile;
76. **this**.authorities = authorities;
77. }
79. /\*\*
80. \* function get id
81. \* **@return** id
82. \*/
83. **public** **int** getId() {
84. **return** id;
85. }
87. /\*\*
88. \* function set id
89. \* **@param** id
90. \*/
91. **public** **void** setId(**int** id) {
92. **this**.id = id;
93. }
95. /\*\*
96. \* function get username
97. \* **@return** username
98. \*/
99. **public** String getUsername() {
100. **return** username;
101. }
103. /\*\*
104. \* function set username
105. \* **@param** username
106. \*/
107. **public** **void** setUsername(String username) {
108. **this**.username = username;
109. }
111. /\*\*
112. \* function get password
113. \* **@return** password
114. \*/
115. **public** String getPassword() {
116. **return** password;
117. }
119. /\*\*
120. \* function set password
121. \* **@param** password
122. \*/
123. **public** **void** setPassword(String password) {
124. **this**.password = password;
125. }
127. /\*\*
128. \* function get email
129. \* **@return** email
130. \*/
131. **public** String getEmail() {
132. **return** email;
133. }
135. /\*\*
136. \* function set email
137. \* **@param** email
138. \*/
139. **public** **void** setEmail(String email) {
140. **this**.email = email;
141. }
143. /\*\*
144. \* function get profile
145. \* **@return** profile
146. \*/
147. **public** String getProfile() {
148. **return** profile;
149. }
151. /\*\*
152. \* function set profile
153. \* **@param** profile
154. \*/
155. **public** **void** setProfile(String profile) {
156. **this**.profile = profile;
157. }
159. /\*\*
160. \* function get authority
161. \* **@return** authorities
162. \*/
163. **public** Collection<Authorities> getAuthorities() {
164. **return** authorities;
165. }
167. /\*\*
168. \* function set authority
169. \* **@param** authorities
170. \*/
171. **public** **void** setAuthorities(Collection<Authorities> authorities) {
172. **this**.authorities = authorities;
173. }
175. /\*\*
176. \* function get all post belong this user
177. \* **@return** post
178. \*/
179. **public** Set<Post> getPost() {
180. **return** post;
181. }
183. /\*\*
184. \* function set post belong this user
185. \* **@param** post
186. \*/
187. **public** **void** setPost(Set<Post> post) {
188. **this**.post = post;
189. }
190. }

**Authorities.java**

1. package fa.training.model;
2. import javax.persistence.Entity;
3. import javax.persistence.Id;
4. import javax.persistence.JoinColumn;
5. import javax.persistence.ManyToOne;
6. import javax.persistence.Table;
7. /\*\*
8. \* Class model User, map with table authorities
9. \* Declare fields of authorities, getter and setter, Constructor.
10. \* @author KyLH
11. \*
12. \*/
13. @Entity
14. @Table(name = "authorities")
15. public class Authorities {
17. // Declare field id, auto generate.
18. @Id
19. private String authority;
21. // Declare foreign key to table tbl\_user
22. @ManyToOne
23. @JoinColumn(name = "userid")
24. private User user;
26. public Authorities() {
28. }
29. public Authorities(String name) {
30. this.authority = name;
31. }
33. /\*\*
34. \* function get authority
35. \* @return authority
36. \*/
37. public String getAuthority() {
38. return authority;
39. }
41. /\*\*
42. \* function set authority
43. \* @param authority
44. \*/
45. public void setAuthority(String authority) {
46. this.authority = authority;
47. }
49. /\*\*
50. \* function get user
51. \* @return user
52. \*/
53. public User getUser() {
54. return user;
55. }
57. /\*\*
58. \* function set user
59. \* @param user
60. \*/
61. public void setUser(User user) {
62. this.user = user;
63. }
64. }

Step 3: Hibernate configuration

First, create a properties file under **src/main/resources** folder and define the database connection, hibernate properties as follows.

**database.properties**

1. # MySQL connection properties
2. jdbc.driverClassName = com.mysql.jdbc.Driver
3. jdbc.url = jdbc:mysql://localhost:3306/blogmanagement
4. jdbc.username = root
5. jdbc.password = root
6. # Hibernate properties
7. hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
8. hibernate.show\_sql = true
9. hibernate.format\_sql = true
10. hibernate.hbm2ddl.auto = update
11. org.hibernate.jpa.internal.TransactionImpl

Next, create a @Configuration class and define the @Bean method for LocalSessionFactoryBean as follows. In Spring based application, LocalSessionFactoryBean class is used to create a Hibernate SessionFactory.

**AppContext.java**

1. package fa.training.config;
2. import java.util.Properties;
3. import javax.sql.DataSource;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.context.annotation.Bean;
6. import org.springframework.context.annotation.ComponentScan;
7. import org.springframework.context.annotation.Configuration;
8. import org.springframework.context.annotation.PropertySource;
9. import org.springframework.core.env.Environment;
10. import org.springframework.jdbc.datasource.DriverManagerDataSource;
11. import org.springframework.orm.hibernate5.HibernateTransactionManager;
12. import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
13. import org.springframework.transaction.annotation.EnableTransactionManagement;
14. @Configuration
15. @PropertySource("classpath:database.properties")
16. @EnableTransactionManagement
17. @ComponentScan(basePackages = { "fa.training" })
18. public class AppContext {
19. @Autowired
20. private Environment environment;
21. @Bean
22. public LocalSessionFactoryBean sessionFactory() {
23. LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
24. sessionFactory.setDataSource(dataSource());
25. sessionFactory.setPackagesToScan(new String[] { "fa.training.model" });
26. sessionFactory.setHibernateProperties(hibernateProperties());
27. return sessionFactory;
28. }
29. @Bean
30. public DataSource dataSource() {
31. DriverManagerDataSource dataSource = new DriverManagerDataSource();
32. dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.driverClassName"));
33. dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
34. dataSource.setUsername(environment.getRequiredProperty("jdbc.username"));
35. dataSource.setPassword(environment.getRequiredProperty("jdbc.password"));
36. return dataSource;
37. }
38. private Properties hibernateProperties() {
39. Properties properties = new Properties();
40. properties.put("hibernate.dialect", environment.getRequiredProperty("hibernate.dialect"));
41. properties.put("hibernate.show\_sql", environment.getRequiredProperty("hibernate.show\_sql"));
42. properties.put("hibernate.format\_sql", environment.getRequiredProperty("hibernate.format\_sql"));
43. properties.put("hibernate.hbm2ddl.auto", environment.getRequiredProperty("hibernate.hbm2ddl.auto"));
44. return properties;
45. }
46. @Bean
47. public HibernateTransactionManager getTransactionManager() {
48. HibernateTransactionManager transactionManager = new HibernateTransactionManager();
49. transactionManager.setSessionFactory(sessionFactory().getObject());
50. return transactionManager;
51. }
53. }

Step 4: Repository classes

Create @Repository classes **under fa.training.dao** package as follows.

**UserDetailsDao.java**

1. package fa.training.dao;
2. import fa.training.model.User;
3. /\*\*
4. \* Interface UserDetailsDao. This interface consist of 1 function access with database.
5. \* 1.findUserByUsername
6. \* @author KyLH
7. \*
8. \*/
9. public interface UserDetailsDao {
11. /\*\*
12. \* function find user by username
13. \*
14. \* @param username
15. \* @return user
16. \*/
17. User findUserByUsername(String username);
18. }

**UserDetailsDaoImp.java**

1. package fa.training.dao;
2. import javax.persistence.Query;
3. import javax.persistence.criteria.CriteriaBuilder;
4. import javax.persistence.criteria.CriteriaQuery;
5. import javax.persistence.criteria.Root;
6. import org.hibernate.Session;
7. import org.hibernate.SessionFactory;
8. import org.springframework.beans.factory.annotation.Autowired;
9. import org.springframework.stereotype.Repository;
10. import fa.training.model.User;
11. /\*\*
12. \* Class implement UserDetailsDao. This class consist of 1 function access with database.
13. \* 1.findUserByUsername
14. \* @author KyLH
15. \*
16. \*/
17. @Repository
18. public class UserDetailsDaoImp implements UserDetailsDao {
20. // Declare SessionFactory
21. @Autowired
22. private SessionFactory sessionFactory;
24. /\*\*
25. \* function find user by username
26. \*
27. \* @param username
28. \* @return user
29. \*/
30. @Override
31. public User findUserByUsername(String username) {
32. // TODO Auto-generated method stub
33. try {
34. // get Current session.
35. Session session = sessionFactory.getCurrentSession();
36. // get CriteriaBuilder.
37. CriteriaBuilder cb = session.getCriteriaBuilder();
38. // create Query with Model User.
39. CriteriaQuery<User> cq = cb.createQuery(User.class);
40. Root<User> root = cq.from(User.class);
41. // select query
42. cq.select(root).where(cb.equal(root.get("username"), username));
43. // create query
44. Query query = session.createQuery(cq);
46. if (query.getResultList().size() == 0) {
47. return null;
48. }
49. User user = (User) query.getResultList().get(0);
50. return user;
51. } catch (Exception e) {
52. return null;
53. }
54. }
56. }

Step 5: UserDetailsService or Service class

To create a custom user service, you need to implement the UserDetailsService interface and override the loadUserByUsername() method. Create @Service class under **fa.training.service** package as follows.

**UserDetailsServiceImp.java**

1. package fa.training.service;
2. import org.slf4j.Logger;
3. import org.slf4j.LoggerFactory;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.security.core.userdetails.UserDetails;
6. import org.springframework.security.core.userdetails.UserDetailsService;
7. import org.springframework.security.core.userdetails.UsernameNotFoundException;
8. import org.springframework.stereotype.Service;
9. import org.springframework.transaction.annotation.Transactional;
10. import fa.training.dao.UserDetailsDao;
11. import fa.training.dto.UserDetailsDTO;
12. import fa.training.model.User;
13. /\*\*
14. \* Class UserDetails Service, consist of 1 function:
15. \* 1. loadUserByUsername
16. \* @author KyLH
17. \*
18. \*/
19. @Service("userDetailsService")
20. public class UserDetailsServiceImp implements UserDetailsService {
21. private static final Logger Log = LoggerFactory.getLogger(UserDetailsServiceImp.class);
23. // Declare UserDetailDao
24. @Autowired
25. private UserDetailsDao userDetailsDao;
27. /\*\*
28. \* function get user by username
29. \* @param username
30. \* @return user detail
31. \*/
32. @Transactional(readOnly = true)
33. @Override
34. public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
35. // TODO Auto-generated method stub
36. User user = userDetailsDao.findUserByUsername(username);
37. if (user == null) {
38. throw new UsernameNotFoundException("User not found.");
39. } else {
40. Log.info("loadUserByUsername() : {}", username);
41. return new UserDetailsDTO(user);
42. }
43. }
44. }

Step 6: Spring Security configuration

To configure Spring Security in Spring MVC application you need to create a @Configuration class by extending the WebSecurityConfigurerAdapter class and annotate it with @EnableWebSecurity as follows.

**WebSecurityConfig.java**

1. package fa.training.config;
2. import org.springframework.beans.factory.annotation.Autowired;
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
5. import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
6. import org.springframework.security.config.annotation.web.builders.HttpSecurity;
7. import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
8. import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
9. import org.springframework.security.core.userdetails.UserDetailsService;
10. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
11. /\*\*
12. \* Class Config web security
13. \* @author KyLH
14. \*
15. \*/
16. @EnableWebSecurity
17. public class WebSecurityConfig extends WebSecurityConfigurerAdapter{
19. // Declare UserDetailsService
20. @Autowired
21. private UserDetailsService userDetailsService;
23. /\*\*
24. \* function bcrypt encode password
25. \* @return BCryptPasswordEncoder
26. \*/
27. @Bean
28. public BCryptPasswordEncoder passwordEncoder() {
29. return new BCryptPasswordEncoder();
30. }
32. /\*\*
33. \* function provide authenrity to user
34. \* @return auProvider
35. \*/
36. @Bean
37. public DaoAuthenticationProvider authenticationProvider() {
38. DaoAuthenticationProvider auProvider = new DaoAuthenticationProvider();
39. auProvider.setUserDetailsService(userDetailsService);
40. auProvider.setPasswordEncoder(passwordEncoder());
41. return auProvider;
42. }
44. /\*\*
45. \* function configure
46. \* @param auth
47. \*/
48. @Override
49. protected void configure(AuthenticationManagerBuilder auth) throws Exception {
50. // TODO Auto-generated method stub
51. auth.authenticationProvider(authenticationProvider());
52. }
54. /\*\*
55. \* function http config
56. \* @param http
57. \*/
58. @Override
59. protected void configure(HttpSecurity http) throws Exception {
60. // TODO Auto-generated method stub
61. http.authorizeRequests()
62. .antMatchers("/").permitAll()
63. .antMatchers("/AboutPage").permitAll()
64. .antMatchers("/LoginPage").permitAll()
65. .antMatchers("/ContactPage").hasAnyRole("ADMIN","USER")
66. .antMatchers("/ManagePostPage").hasAnyRole("ADMIN","USER")
67. .antMatchers("/CreatePostPage").hasAnyRole("ADMIN","USER")
68. .antMatchers("/ManageCommentPage/\*\*").hasAnyRole("ADMIN","USER")
69. .antMatchers("/ManageCommentPage").hasAnyRole("ADMIN","USER")
70. .antMatchers("/EditPostPage").hasAnyRole("ADMIN","USER")
71. .and()
72. .formLogin().loginPage("/LoginPage")
73. .usernameParameter("username").passwordParameter("password")
74. .loginProcessingUrl("/doLogin")
75. .successForwardUrl("/postLogin")
76. .failureUrl("/loginFailed").and().logout().logoutUrl("/doLogout")
77. .logoutSuccessUrl("/logout").permitAll()
78. .and().csrf().disable();
79. }
80. }

Next, create SecurityWebApplicationInitializer class by extending the AbstractSecurityWebApplicationInitializer to register the springSecurityFilterChain filter.

**SecurityWebApplicationInitializer.java**

1. package fa.training.config;
2. import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;
3. /\*\*
4. \* Class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer
5. \* @author KyLH
6. \*
7. \*/
8. public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer{
9. }

Step 7: Spring MVC configuration

In this lab, we are using the JSP views. So create a @Configuration class and override the configureViewResolvers() method to register the JSP view resolver.

Also, you can override the addViewControllers() method to map and render the default login page generated by Spring Security.

**AppConfig.java**

1. package fa.training.config;
2. import org.springframework.context.annotation.Bean;
3. import org.springframework.context.annotation.ComponentScan;
4. import org.springframework.context.annotation.Configuration;
5. import org.springframework.web.servlet.config.annotation.EnableWebMvc;
6. import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
7. import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;
8. import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
9. import org.springframework.web.servlet.view.tiles3.TilesConfigurer;
10. import org.springframework.web.servlet.view.tiles3.TilesViewResolver;
11. /\*\*
12. \* Class Config webMvc and resources
13. \* @author KyLH
14. \*
15. \*/
16. @Configuration
17. @EnableWebMvc
18. @ComponentScan(basePackages = { "fa.training" })
19. public class AppConfig implements WebMvcConfigurer {
20. /\*\*
21. \* Configure TilesConfigurer.
22. \*/
23. @Bean
24. public TilesConfigurer tilesConfigurer() {
25. TilesConfigurer tilesConfigurer = new TilesConfigurer();
26. tilesConfigurer.setDefinitions(new String[] { "/WEB-INF/views/\*\*/tiles.xml" });
27. tilesConfigurer.setCheckRefresh(true);
28. return tilesConfigurer;
29. }
30. /\*\*
31. \* Configure ViewResolvers to deliver preferred views.
32. \*/
33. public void configureViewResolvers(ViewResolverRegistry registry) {
34. TilesViewResolver viewResolver = new TilesViewResolver();
35. registry.viewResolver(viewResolver);
36. }
37. /\*\*
38. \* Function handle url of lib
39. \*/
40. public void addResourceHandlers(ResourceHandlerRegistry registry) {
41. registry.addResourceHandler("/resources/\*\*").addResourceLocations("/resources/");
42. }
43. }

Step 8: Servlet container Initialization and configuration

In Spring MVC, The DispatcherServlet needs to be declared and mapped for processing all requests either using java or web.xmlconfiguration.

In a Servlet 3.0+ environment, you can use AbstractAnnotationConfigDispatcherServletInitializer class to register and initialize the DispatcherServlet programmatically as follows.

**SpringMvcDispatcherServletInitializer.java**

1. package fa.training.config;
2. import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
3. /\*\*
4. \* AbstractAnnotationConfigDispatcherServletInitializer class to register and initialize the DispatcherServlet
5. \* @author KyLH
6. \*
7. \*/
8. public class SpringMvcDispatcherServletInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
9. @Override
10. protected Class<?>[] getRootConfigClasses() {
11. // TODO Auto-generated method stub
12. return new Class[] {AppContext.class ,WebSecurityConfig.class};
13. }
14. @Override
15. protected Class<?>[] getServletConfigClasses() {
16. // TODO Auto-generated method stub
17. return new Class[] { AppConfig.class };
18. }
19. @Override
20. protected String[] getServletMappings() {
21. // TODO Auto-generated method stub
22. return new String[] { "/" };
23. }
24. }

Step 9: Controller class

Create a simple @Controller class under **fa.training.controller** package as follows.

**UserController.java**

1. **package** fa.training.controller;
2. **import** java.security.Principal;
3. **import** java.util.Arrays;
4. **import** javax.servlet.http.HttpSession;
5. **import** javax.validation.Valid;
6. **import** org.slf4j.Logger;
7. **import** org.slf4j.LoggerFactory;
8. **import** org.springframework.beans.factory.annotation.Autowired;
9. **import** org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
10. **import** org.springframework.security.core.Authentication;
11. **import** org.springframework.security.core.context.SecurityContextHolder;
12. **import** org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
13. **import** org.springframework.stereotype.Controller;
14. **import** org.springframework.ui.Model;
15. **import** org.springframework.validation.BindingResult;
16. **import** org.springframework.web.bind.annotation.GetMapping;
17. **import** org.springframework.web.bind.annotation.ModelAttribute;
18. **import** org.springframework.web.bind.annotation.PostMapping;
19. **import** org.springframework.web.bind.annotation.RequestMapping;
20. **import** org.springframework.web.bind.annotation.RequestMethod;
21. **import** org.springframework.web.bind.annotation.SessionAttributes;
22. **import** org.springframework.web.bind.support.SessionStatus;
23. **import** org.springframework.web.servlet.ModelAndView;
24. **import** fa.training.dto.UserDetailsDTO;
25. **import** fa.training.model.Authorities;
26. **import** fa.training.model.User;
27. **import** fa.training.service.CommentService;
28. **import** fa.training.service.UserService;
29. **import** fa.training.validator.UserValidator;
30. /\*\*
31. \* This Class User Controller Consist of many mapping process functions.
32. \*
33. \* **@author** KyLH
34. \*
35. \*/
36. @SessionAttributes({ "currentUser" })
37. @Controller
38. **public** **class** UserController {
39. // Declare Comment Service
40. @Autowired
41. **private** CommentService commentService;
42. **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(UserController.**class**);
43. **private** **static** **int** *error*;
44. // Declare User validator
45. @Autowired
46. **private** UserValidator userValidator;
47. // Declare User service
48. @Autowired
49. **private** UserService userService;
50. /\*\*
51. \* function to BCrypt encode password
52. \*
53. \* **@param** password
54. \* **@return** String
55. \*/
56. **public** String passwordEncoder(String password) {
57. **return** **new** BCryptPasswordEncoder().encode(password);
58. }
59. /\*\*
60. \* function Get Mapping with direct: /LoginPage
61. \*
62. \* **@param** principal
63. \* **@return**
64. \*/
65. @GetMapping("/LoginPage")
66. **public** String loginPage(Principal principal, Model result) {
67. System.***out***.println(*error*);
68. **if** (*error* == 1) {
69. result.addAttribute("Result", "fail\_create");
70. } **else** **if** (*error* == 2) {
71. result.addAttribute("Result", "success\_create");
72. } **else** **if** (*error* == 3) {
73. result.addAttribute("Result", "fail\_login");
74. }
75. error = 0;
76. **return** principal == **null** ? "LoginPage" : "redirect:/";
77. }
78. /\*\*
79. \* function Post Mapping with direct: /postLogin
80. \*
81. \* **@param** model
82. \* **@param** session
83. \* **@return** String "redirect:/index"
84. \*/
85. @PostMapping("postLogin")
86. **public** String postLogin(Model model, HttpSession session) {
87. ***log***.info("postLogin()");
88. // declare Authentication
89. Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();
90. // cast Authentication to UsernamePasswordAuthenticationToken
91. UsernamePasswordAuthenticationToken token = (UsernamePasswordAuthenticationToken) (authentication);
92. // call function validatePrinciple
93. validatePrinciple(token.getPrincipal());
94. // cast UsernamePasswordAuthenticationToken to User
95. User loggedInUser = ((UserDetailsDTO) token.getPrincipal()).getUserDetails();
96. model.addAttribute("currentUser", loggedInUser.getUsername());
97. session.setAttribute("userId", loggedInUser.getId());
98. session.setAttribute("userName", loggedInUser.getUsername());
99. session.setAttribute("userEmail", loggedInUser.getEmail());
100. session.setAttribute("totalComment",
101. commentService.countAllCommentByUserId((**int**) session.getAttribute("userId")));
102. **return** "redirect:/index";
103. }
104. /\*\*
105. \* function to validate principal
106. \*
107. \* **@param** principal
108. \*/
109. **private** **void** validatePrinciple(Object principal) {
110. **if** (!(principal **instanceof** UserDetailsDTO)) {
111. **throw** **new** IllegalArgumentException("Principal can not be null!");
112. }
113. }
114. /\*\*
115. \* function Get Mapping with direct: /loginFailed
116. \*
117. \* **@param** model
118. \* **@return** String "LoginPage"
119. \*/
120. @RequestMapping(value = "loginFailed", method = RequestMethod.***GET***)
121. **public** String loginError(Model model) {
122. ***log***.info("Login attempt failed");
123. *error* = 3;
124. model.addAttribute("error", "true");
125. **return** "redirect:/LoginPage";
126. }
127. /\*\*
128. \* function Get Mapping with direct: /Logout logout this user
129. \*
130. \* **@param** session
131. \* **@param** session2
132. \* **@return** new ModelAndView
133. \*/
134. @GetMapping("Logout")
135. **public** ModelAndView logoutPage(SessionStatus session, HttpSession session2) {
136. SecurityContextHolder.*getContext*().setAuthentication(**null**);
137. *error* = 0;
138. session.setComplete();
139. session2.invalidate();
140. **return** **new** ModelAndView("redirect:/LoginPage");
141. }
142. /\*\*
143. \* function Post Mapping with direct: /createUser
144. \*
145. \* **@param** user
146. \* **@param** theBindingResult
147. \* **@return** String "redirect:/LoginPage"
148. \*/
149. @PostMapping("createUser")
150. **public** String createUser(@Valid @ModelAttribute("User") User user, BindingResult theBindingResult) {
151. userValidator.validate(user, theBindingResult);
152. **if** (theBindingResult.hasErrors()) {
153. *error* = 1;
154. **return** "redirect:/LoginPage";
155. } **else** {
156. user.setPassword(passwordEncoder(user.getPassword()));
157. // Create authority
158. Authorities authority = **new** Authorities("ROLE\_USER");
159. authority.setUser(user);
160. user.setAuthorities(Arrays.*asList*(authority));
161. userService.createUser(user);
162. *error* = 2;
163. **return** "redirect:/LoginPage";
164. }
165. }
166. }

**UserValidator.java**

1. package fa.training.service;
2. import java.util.List;
3. import javax.transaction.Transactional;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.stereotype.Service;
6. import fa.training.dao.UserDAO;
7. import fa.training.model.User;
8. /\*\*
9. \* Class User Service, connect with UserDAOImpl, consist of 3 function:
10. \* 1. getUsers 2. createUser 3. findUserByUsername
11. \* @author KyLH
12. \*
13. \*/
14. @Service
15. public class UserService {
17. // Declare UserDAO
18. @Autowired
19. private UserDAO userDAO;
21. /\*\*
22. \* function get all user
23. \* @return list user
24. \*/
25. @Transactional
26. public List<User> getUsers() {
27. return userDAO.getAllUsers();
28. }
29. /\*\*
30. \* function create user
31. \* @param user
32. \*/
33. @Transactional
34. public void createUser(User user) {
35. userDAO.createUser(user);
36. }
38. /\*\*
39. \* function find user by username
40. \* @param username
41. \* @return user
42. \*/
43. @Transactional
44. public User findUserByUsername(String username) {
45. return userDAO.findUserByUsername(username);
46. }
47. }

Step 10: DTO class

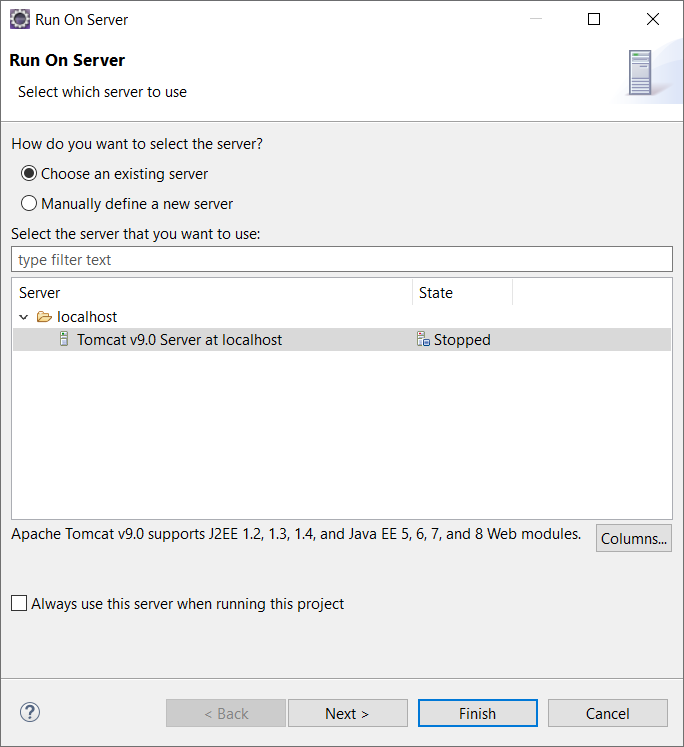
Create a simple DTO class under **fa.training.dto** package as follows.

**UserDetailsDTO.java**

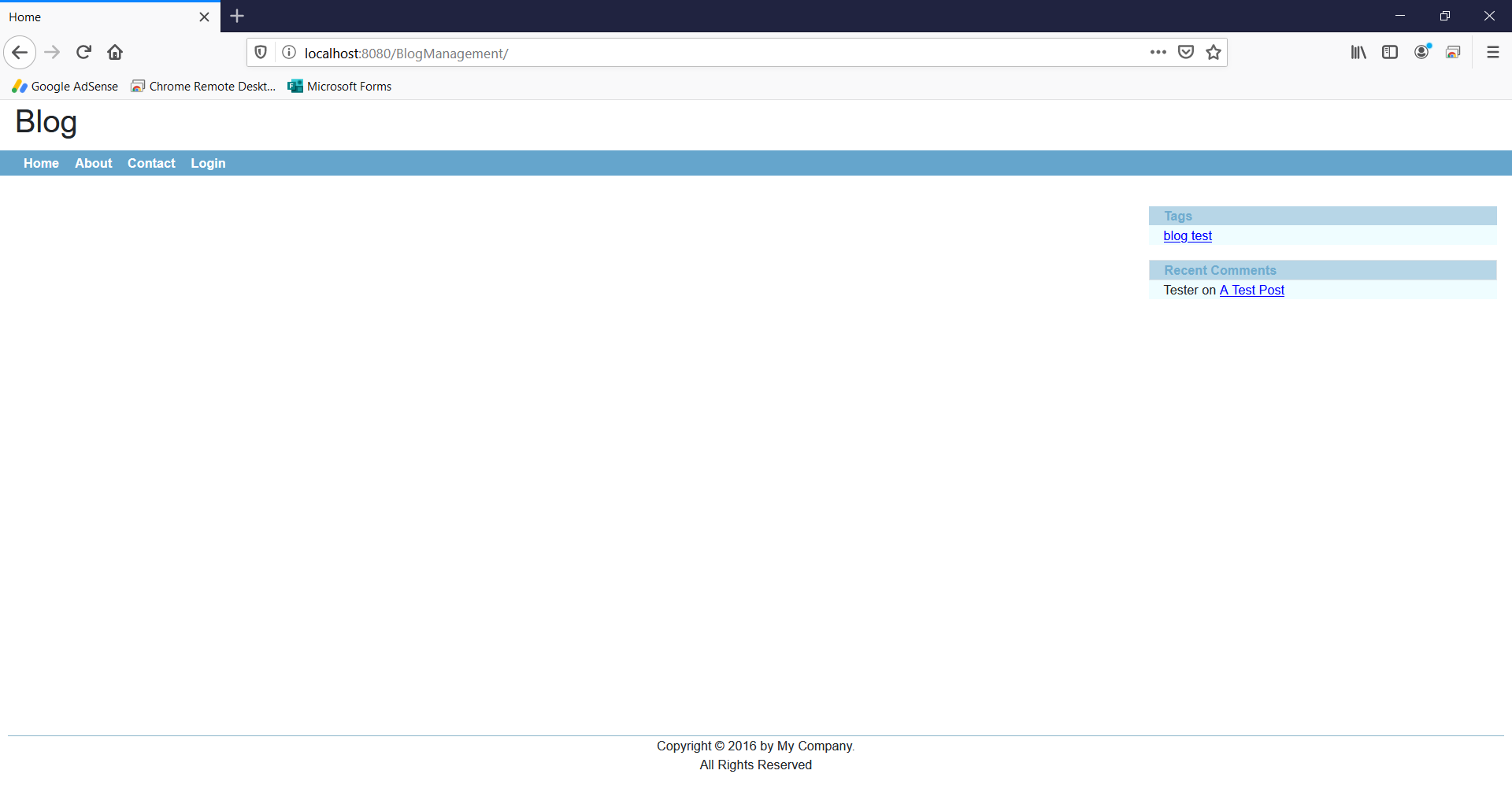
1. package fa.training.dto;
2. import java.util.Collection;
3. import java.util.stream.Collectors;
4. import org.springframework.security.core.GrantedAuthority;
5. import org.springframework.security.core.authority.SimpleGrantedAuthority;
6. import org.springframework.security.core.userdetails.UserDetails;
7. import fa.training.model.User;
8. /\*\*
9. \* Class UserDetailsDTO implements UserDeatails This class consist of 9
10. \* function. 1. UserDetailsDTO 2.getAuthorities 3.getUserDetails 4.getPassword
11. \* 5.getUsername 6.isAccountNonExpired 7.isAccountNonLocked
12. \* 8.isCredentialsNonExpired 9.isEnabled
13. \*
14. \* @author KyLH
15. \*
16. \*/
17. public class UserDetailsDTO implements UserDetails {
19. // Declare User
20. private User user;
22. /\*\*
23. \* Constructor
24. \* @param user
25. \*/
26. public UserDetailsDTO(User user) {
27. this.user = user;
28. // TODO Auto-generated constructor stub
29. }
31. /\*\*
32. \* function get authority
33. \* @return authority
34. \*/
35. @Override
36. public Collection<? extends GrantedAuthority> getAuthorities() {
37. // TODO Auto-generated method stub
38. return user.getAuthorities().stream()
39. .map(authority -> new SimpleGrantedAuthority(authority.getAuthority().toString()))
40. .collect(Collectors.toList());
41. }
43. /\*\*
44. \* function get detail of user
45. \* @return user
46. \*/
47. public User getUserDetails() {
48. return user;
49. }
51. /\*\*
52. \* function get password
53. \* @return password
54. \*/
55. @Override
56. public String getPassword() {
57. // TODO Auto-generated method stub
58. return user.getPassword();
59. }
61. /\*\*
62. \* fuction get username
63. \* @return username
64. \*/
65. @Override
66. public String getUsername() {
67. // TODO Auto-generated method stub
68. return user.getUsername();
69. }
71. /\*\*
72. \* function set AccountNonExpired
73. \* @return true
74. \*/
75. @Override
76. public boolean isAccountNonExpired() {
77. // TODO Auto-generated method stub
78. return true;
79. }
81. /\*\*
82. \* function set AccountNonLocked
83. \* @return true
84. \*/
85. @Override
86. public boolean isAccountNonLocked() {
87. // TODO Auto-generated method stub
88. return true;
89. }
91. /\*\*
92. \* function set CredentialsNonExpired
93. \* @return true
94. \*/
95. @Override
96. public boolean isCredentialsNonExpired() {
97. // TODO Auto-generated method stub
98. return true;
99. }
101. /\*\*
102. \* function set AccountEnabled
103. \* @return true
104. \*/
105. @Override
106. public boolean isEnabled() {
107. // TODO Auto-generated method stub
108. return true;
109. }
110. }

Step 11: Run application

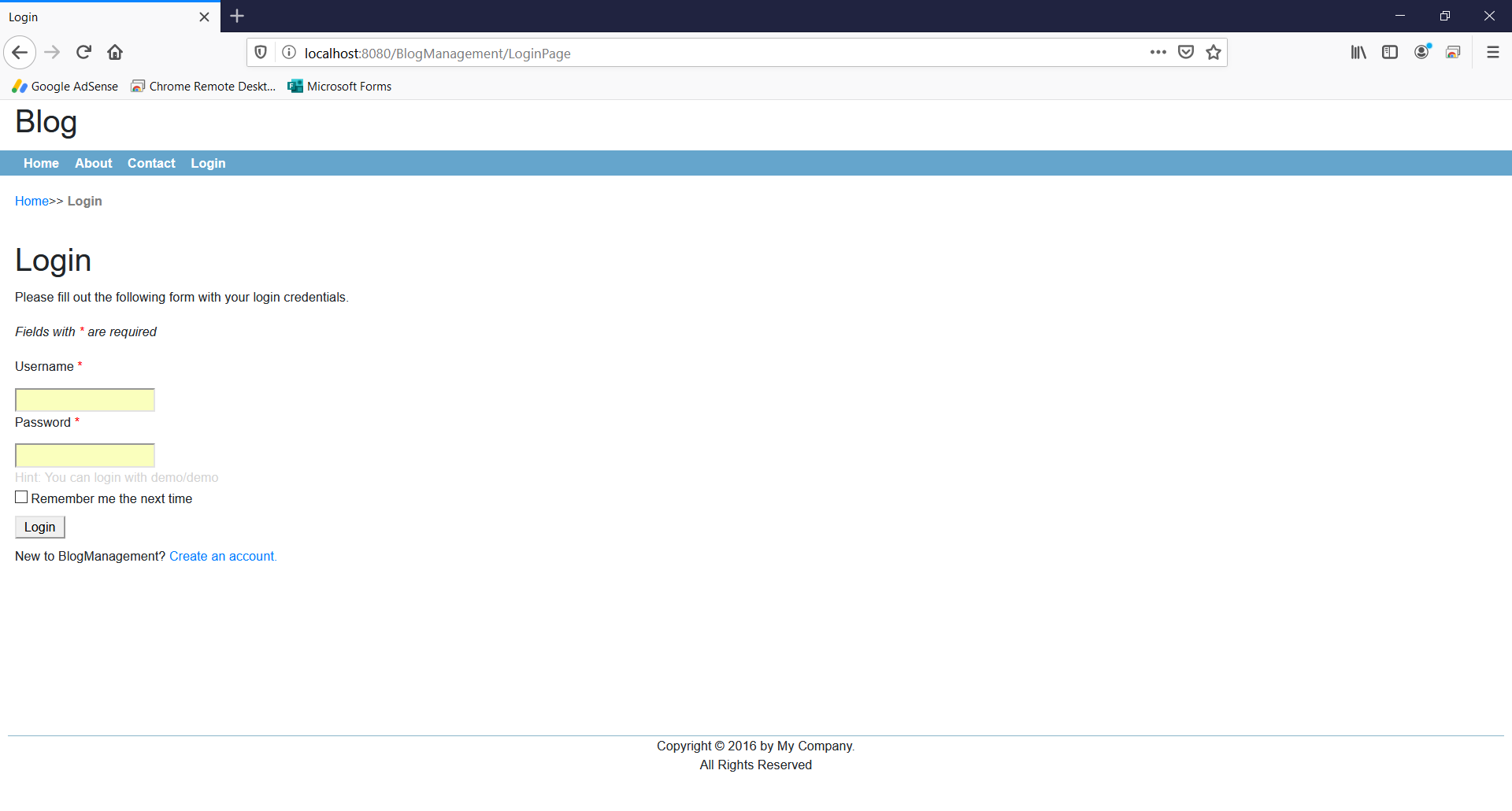
Right clcik project -> Run As -> Run on Server -> Finish



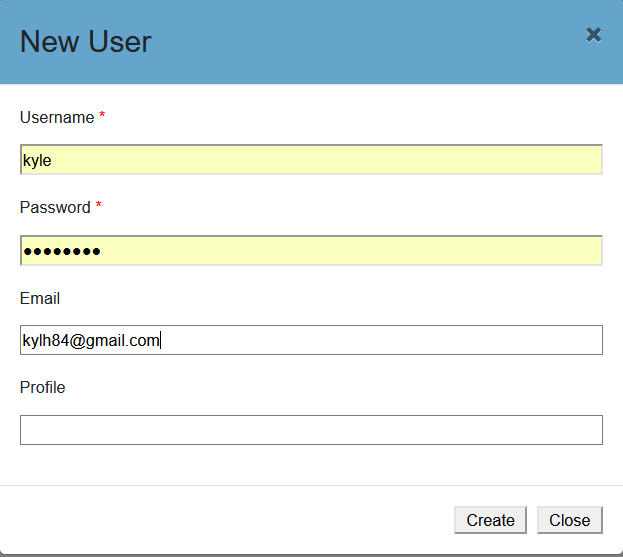
Enter the http://localhost:8080 URL in browser's address bar to test our application. On entering the URL, you will see the home page as follows.



Click ‘Login’, you will see the login page as follows

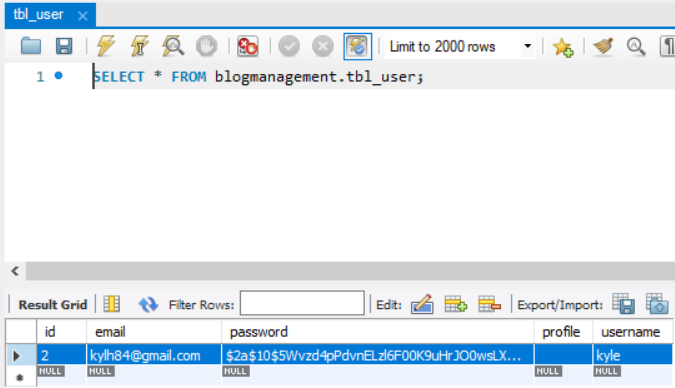


Click ‘Create an account’, you will see the new user dialog as follows

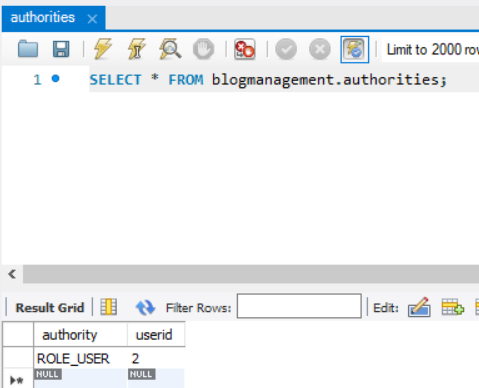


On successful sign up, you will see the data as follows.

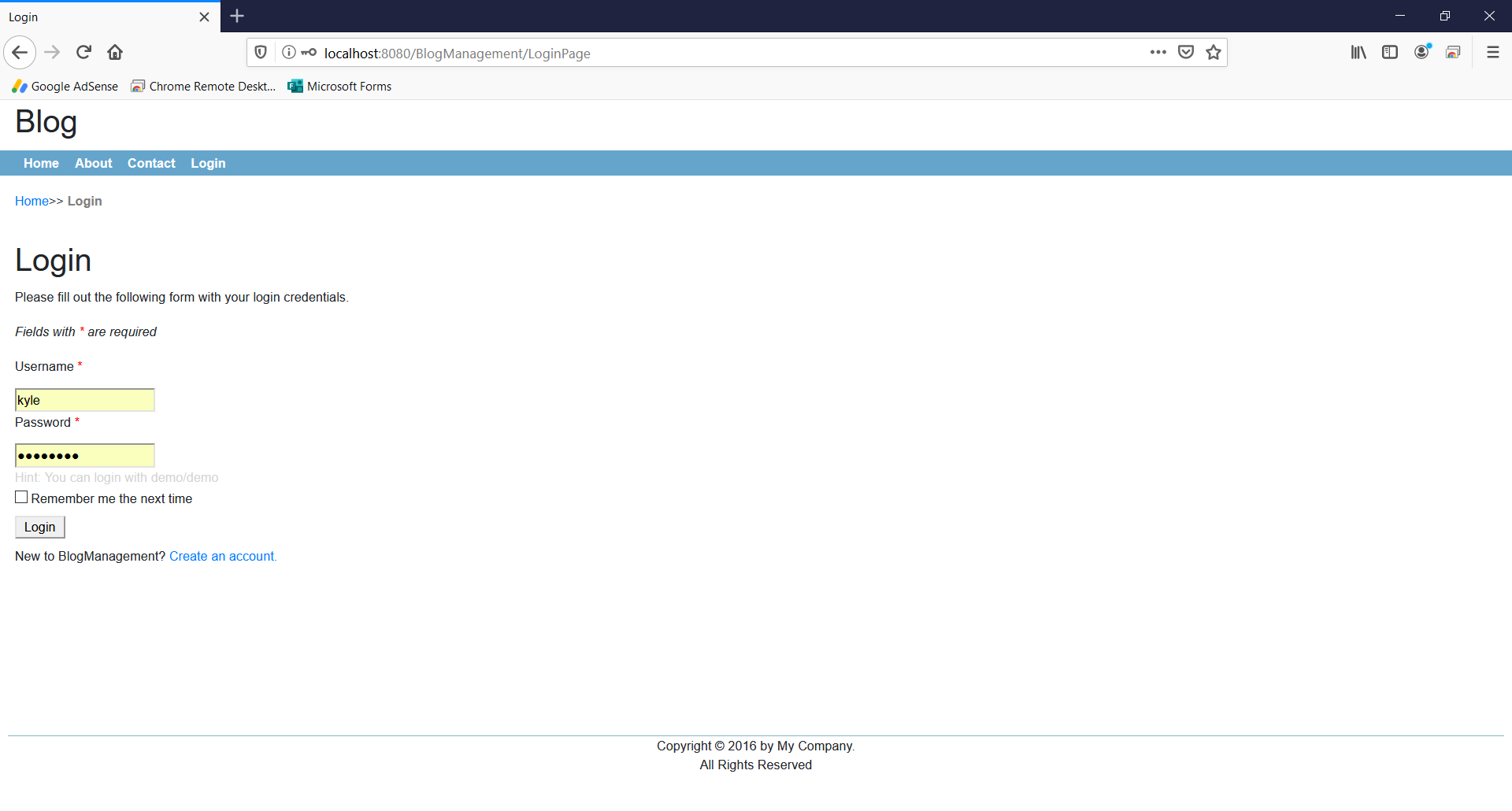
**Tbl\_user**



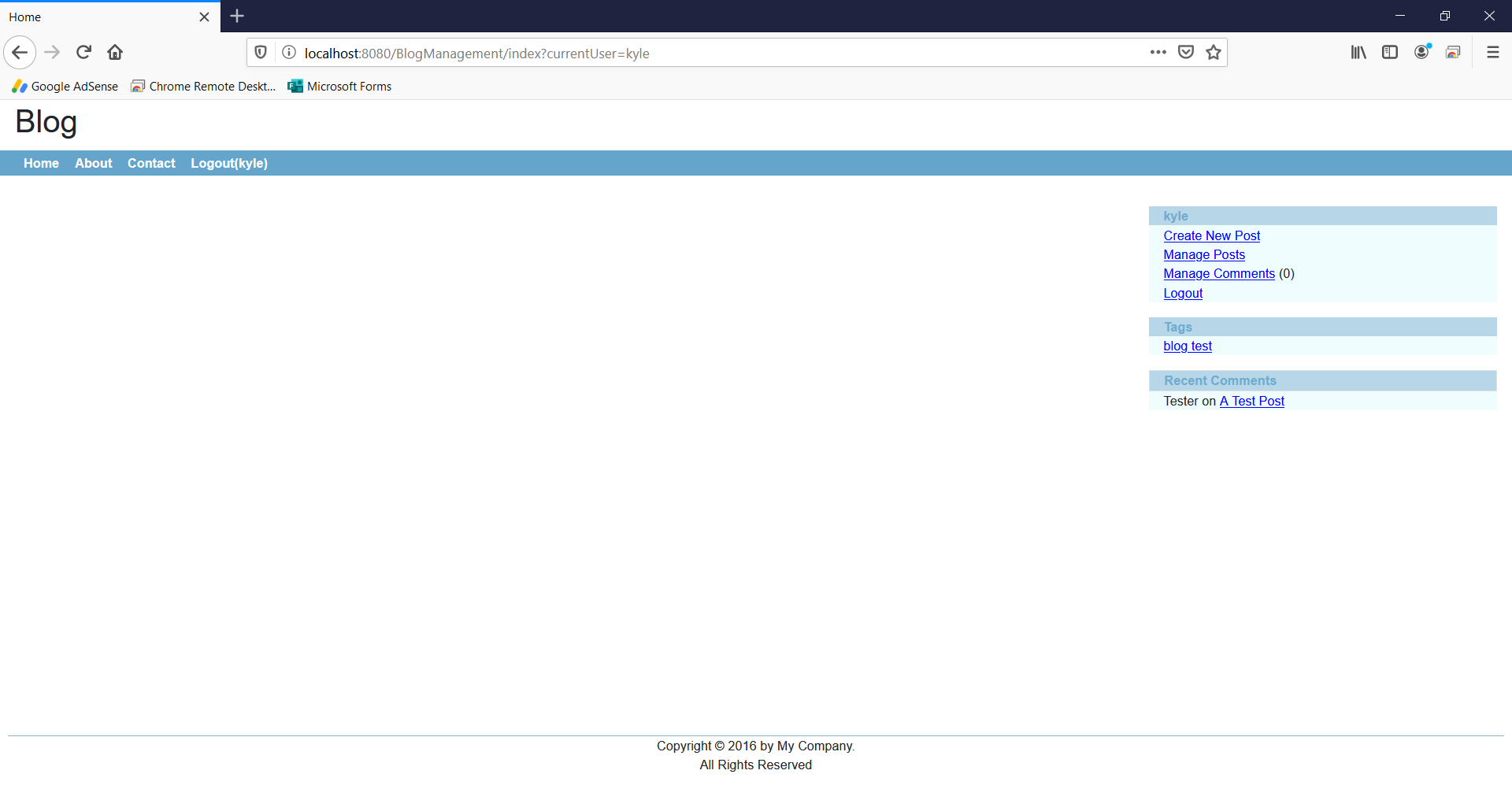
**authorities**



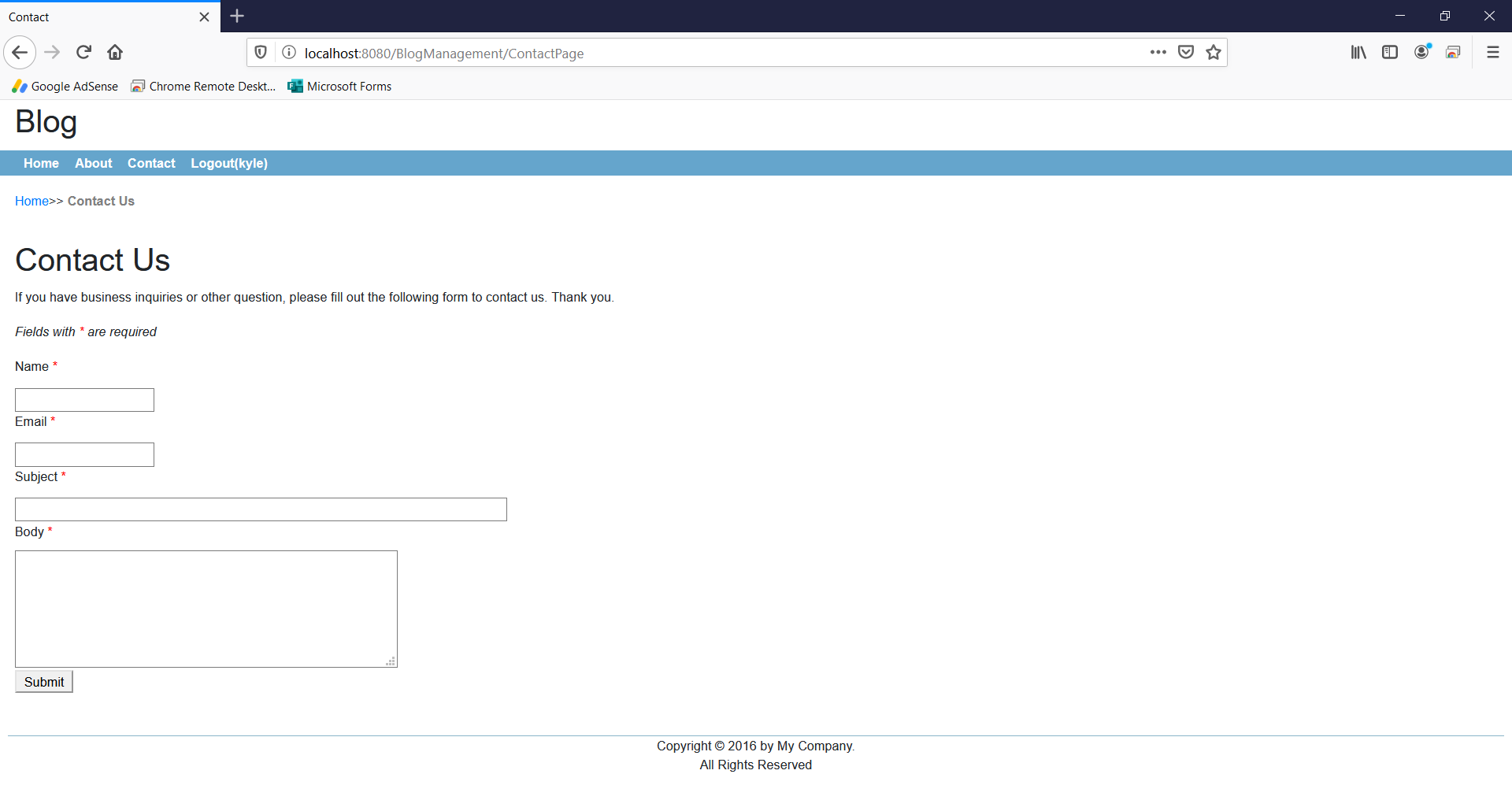
Click ‘Contact’ menu before login, the system will redirect ‘login’ page



On successful login, you will see the index page as follows.



Click ‘Contact’ menu, you will see the contact page as follows



**-- THE END --**