Chapter 4- Securing Spring

* 4.1 ENABLING SPRING SECURITY:
  + Add spring boot security starter

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

* + You will get the following autoconfigured features by adding this starter:
    - All HTTP request paths require authentication
    - No specific roles or authorities are required
    - There’s no login page
    - Authentication is prompted with HTTP basic authentication
    - There’s only one user; the username is user.
  + This is not enough. You need to configure Spring security to at least do the following:
    - Prompt for authentication with login page, instead of an HTTP basic dialog box.
    - Provide for multiple users, and enable a registration page so new Taco Cloud Customers can sign up.
    - Apply different security rules for different request paths. The homepage and registration pages, for example, shouldn’t require authentication at all.
    - Let’s start by configuring a proper user store so that you can have multiple users.
* 4.2 CONFIGURING SPRING SECURITY
  + Writing a barebone configuration class

package tacos.security;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

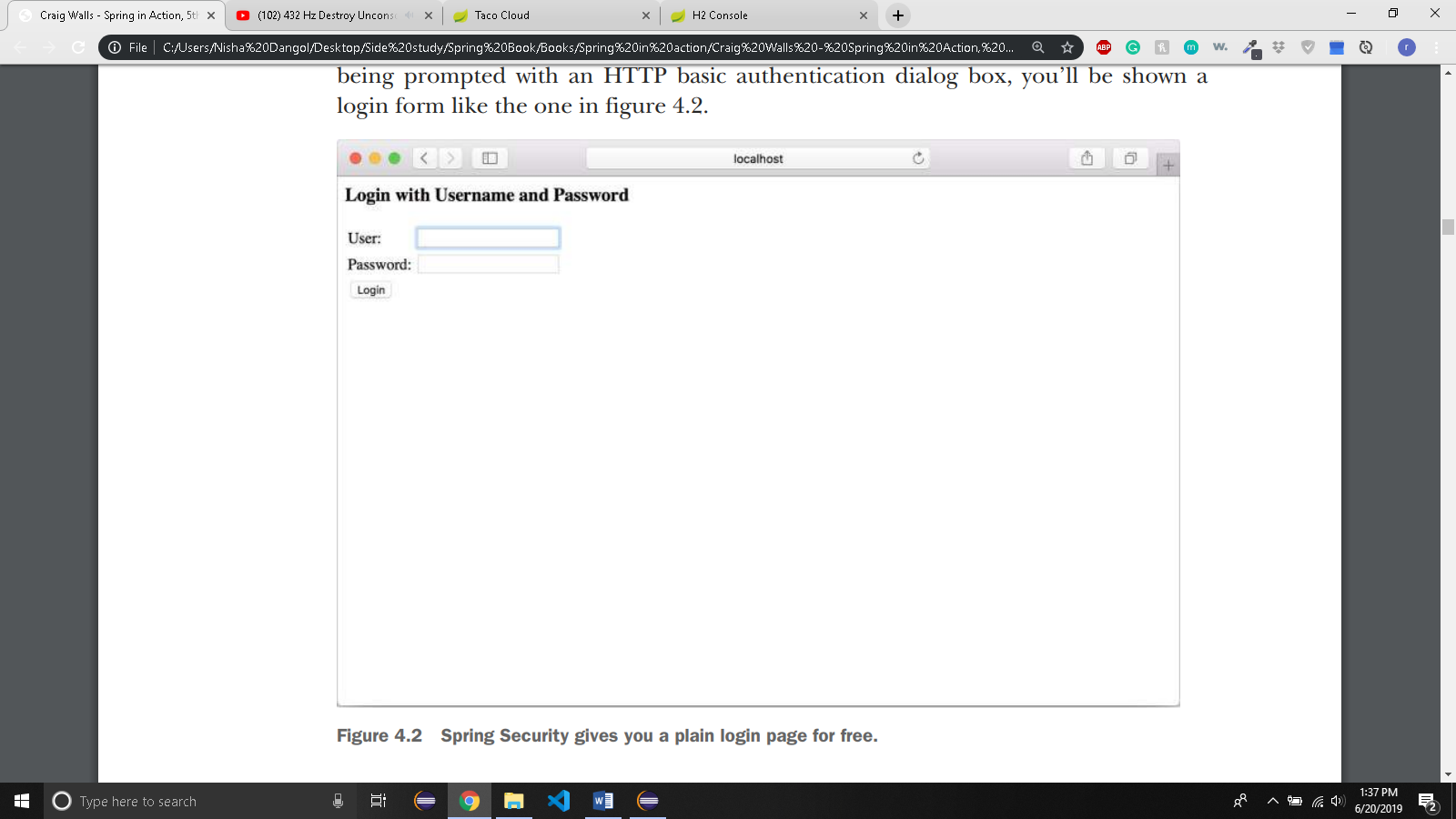
*@Configuration*

*@EnableWebSecurity*

public class SecurityConfig extends WebSecurityConfigurerAdapter {

}

* + - This lets you see a login form just like this:



* + Spring Security offers several ways to configure a user store
    - An In-memory user store
    - A JDBC-based user store
    - An LDAP-backed user store
    - A custom user details service
  + No matter which user store you choose, you can configure it by overriding configure() method defined in WebSecurityConfigurerAdapter configuration base class.
* 4.2.1 IN-MEMORY USER STORE
  + Use it if you have only handful of users
  + It may be simple enough to define those users as part of security configuration
  + Lets configure two users Rohan and Roshik in in-memory user store

package tacos.security;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

*@Configuration*

*@EnableWebSecurity*

public class SecurityConfig extends WebSecurityConfigurerAdapter {

/\* This method is used to configure the user store

\*AuthenticationManagerBuilder employs builder-styole API to configure authentication details

\*/

*@Override*

protected void configure(AuthenticationManagerBuilder auth) throws Exception{

/\*

\* call to inMemoryAuthentication() method specefies user information directly

\* in the security configuration itself. Each call to withUser() starts the

\* configuration for a User.class The username, password and the granted

\* authorities are set.

\*/

auth

.inMemoryAuthentication()

.withUser("Rohan")

.password("Rohan9841")

.authorities("ROLE\_USER")

.and()

.withUser("Roshi")

.password("Roshik9841")

.authorities("ROLE\_USER");

}

}

* + - Convenient for testing purpose
    - Hard to edit users
      * Have to make changes and rebuild and redeploy the application.
    - In taco-cloud application, you want user to be able to register with the application and manage their own user accounts. So, let’s look at the database backed user store
* 4.2.2 JDBC BASED USER STORE
  + User information is oftern maintained in relational database and a JDBC based user store seems appropriate.
  + The configuration for jdbc authentication automatically gives us table and queries. There are basically 3 queries:
    - Authenticate the user
      * The authentication query selects the username, password and enabled status
    - Look up user’s granted authorities for authorization purpose
      * The authorities query selects zero or more rows containing the username and a granted authority
    - Look up authorities granted to a user as a member of a group.
      * The group authorities query selects zero or more rows, each with a group ID, a group name, and an authority.
  + However, if you want to configure your own queries, you can do something like this:

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

*@Configuration*

*@EnableWebSecurity*

public class SecurityConfig extends WebSecurityConfigurerAdapter {

*@Autowired*

DataSource dataSource;

/\* This method is used to configure the user store

\*AuthenticationManagerBuilder employs builder-styole API to configure authentication details

\*/

*@Override*

protected void configure(AuthenticationManagerBuilder auth) throws Exception{

auth

.jdbcAuthentication()

.dataSource(dataSource)

.usersByUsernameQuery(

"select username, password, enabled from Users" +

"where username = ?"

)

.authoritiesByUsernameQuery(

"select username, authority from UserAuthorities"+

"where username = ?"

);

}

}

* + In this case, you only override the authentication and basic authorization queries. But you can override the group authorities query by calling groupAuthoritiesByUsername() with a custom query
  + All the queries must have username as their only parameter
  + WORKING WITH ENCODED PASSWORDS:
    - If the password stored in database is stored in plain text, the hackers might get access easily.
    - If you encode the password in the database, it will fail authentication as it won’t match the plain text password submitted by user.
    - To solve this, you need to specify the password encoder by calling the passwordEncoder() method.

auth

.jdbcAuthentication()

.dataSource(dataSource)

.usersByUsernameQuery(

"select username, password, enabled from Users" +

"where username = ?"

)

.authoritiesByUsernameQuery(

"select username, authority from UserAuthorities"+

"where username = ?"

)

.passwordEncoder(new StandardPasswordEncoder("53cr3t"));

* + - The passwordEncoder accepts any implementation of spring security’s passwordEncoder interface. Some of the implementations are:
      * BCryptPasswordEncoder – Applies bcrypt strong hashing ecryption.
      * NoOpPasswordEncoder – Applies no encoding.
      * Pbkdf2PasswordEncoder – Applies PBKDF2 encryption.
      * SCryptPasswordEncoder – Applies scrypt hashing encryption.
      * StandardPasswordEncoder – Applies SHA-256 hashing encryption.
    - You can choose an implementations or even provide your own custom implementation. The PasswordEncoder interface is as follow:

public interface PasswordEncoder {

String encode(CharSequence rawPassword);

boolean matches(CharSequence rawPassword, String encodedPassword);

}

* + - No matter the implementation, the password in the database is never decoded. Instead the password entered by the user at login is encoded using the same algorithm and them compared with the encoded password in the database. The comparison is done by PasswordEncoder matches() method.
* 4.2.3 LDAP BACKED USER STORE:
  + Use ldapAuthentication() method to configure the Ldap-backed user store.

@Override

Protected void configure(AuthenticationManagerBuilder auth) throws Exception{

auth

.ldapAuthentication()

.userSearchFilter(“(uid = {0})”)

.groupSearchFilter(“member = {0}”);

* + The userSearchFilter() and groupSearchFilter() methods are used to provide filters for the base LDAP queries, which are used to search for users and groups.
  + By default, the base queries for both users ad groups are empty, indicating that the serach will be done from the root of the LDAP hierarchy. But you can change that by specifying a query base:

@Override

Protected void Configure(AuthenticationManagerBuilder auth) throws Exception{

auth

.ldapAuthentication()

.userSearchBase(“ou = people”)

.userSearchFilter(“(“uid = {0})”)

.groupSearchBase(“ou = groups”)

.groupSearchFilter(“member = {0}”);

}

* + The userSearchBase() provides basic query for finding users.
  + The groupSearchBase() method specifies basic query for finding groups.
  + Rather than search from root, this example specifies, that users by searched for where the organizational unit is people. Groups should be searched for where the organizational unit is groups.
  + The default strategy for authenticating against LDAP is to perform a bind operation, authenticating user directly to the LDAP server.
  + Another option is to perform a comparison operation. This involves sending entered password to the LDAP directory and asking the server to compare the password against a user’s password attribute. Because the comparison is done within the LDAP server, the actual password remains secret.
  + If you want to do the password comparison, you can declare passwordCompare() method
  + By default, the password enterd in the login form will be compared with the value of the userPassword attribute in the user’s LDAP entry. If the password is kept in a different attribute, you can specify the password attribute’s name with passwordAttribute().

auth

.ldapAuthentication()

.userSearchBase("ou = people")

.userSearchFilter("(uid = {0})")

.groupSearchBase("ou = groups")

.groupSearchFilter("member={0}")

.passwordCompare()

.passwordEncoder(new BCryptPasswordEncoder())

.passwordAttribute("passcode");

* + We are saying that passcode attribute is what should be compared with given password.
  + But the attempted password is still passed across the wire to the LDAP server and could be intercepted by a hacker. To prevent that you can specify an encryption strategy by called the passwordEncoder() method.
  + In the above example, passwords are encrypted using the bcrypt password hashing function. This assumes that the passwords are also encrypted using bcrypt in the LDAP server.
  + REFERING TO A REMOTE LDAP SERVER:
    - By default, spring security’s LDAP authentication assumes that the LDAP server is listening on port 33389 on localhost.
    - But if your LDAP server is on another machine, you can use the contextSource() method to configure the location.

auth

.ldapAuthentication()

.userSearchBase("ou = people")

.userSearchFilter("(uid = {0})")

.groupSearchBase("ou = groups")

.groupSearchFilter("member={0}")

.passwordCompare()

.passwordEncoder(new BCryptPasswordEncoder())

.passwordAttribute("passcode")

.contextSource() .url("ldap://tacocloud.com:389/dc=tacocloud,dc=com");

* + CONFIGURING AN EMBEDDED LDAP SERVER
    - Spring security can provide an embedded LDAP server for you.
    - Instead of setting url to remote LDAP server, you can specify the root suffix for the embedded server via the root() method.

auth

.ldapAuthentication()

.userSearchBase("ou = people")

.userSearchFilter("(uid = {0})")

.groupSearchBase("ou = groups")

.groupSearchFilter("member={0}")

.passwordCompare()

.passwordEncoder(new BCryptPasswordEncoder())

.passwordAttribute("passcode")

.contextSource.root("dc=tacocloud,dc=com");

* + - When LDAP server starts, it will attempt to load data from any LDIF files that it can find in the classpath. LDIF(LDAP Data Interchange Format) is a standard way of representing LDAP data in a plain text file.
    - Each record has one or more lines containing name:value pair.
    - Records are separated from each other by blank lines.
    - If you don’t want spring to go around looking for any LDIF files it can find, you can explicitly define which LDIF() file to load by calling ldif() method.

auth

.ldapAuthentication()

.userSearchBase("ou = people")

.userSearchFilter("(uid = {0})")

.groupSearchBase("ou = groups")

.groupSearchFilter("member={0}")

.passwordCompare()

.passwordEncoder(new BCryptPasswordEncoder())

.passwordAttribute("passcode")

.contextSource.root("dc=tacocloud,dc=com")

.ldif("classpath:users.ldif");

* + - * Here, you ask LDAP server to load its content from the users.ldif file at the root of the classpath.
      * Here’s an LDIF file that you could use to load the embedded LDAP server with user data

dn: ou=groups,dc=tacocloud,dc=com

objectclass: top

objectclass: organizationalUnit

ou: groups dn: ou=people,dc=tacocloud,dc=com

objectclass: top

objectclass: organizationalUnit

ou: people dn: uid=buzz,ou=people,dc=tacocloud,dc=com objectclass: top

objectclass: person

objectclass: organizationalPerson

objectclass: inetOrgPerson

cn: Buzz Lightyear

sn: Lightyear

uid: buzz

userPassword: password

dn: cn=tacocloud,ou=groups,dc=tacocloud,dc=com

objectclass: top

objectclass: groupOfNames

cn: tacocloud

member: uid=buzz,ou=people,dc=tacocloud,dc=com

* + - * Taco cloud application needs to create and configure a custom user details service.
* 4.2.4 CUSTOMIZING USER AUTHENTICATION
  + DEFINING THE USER DOMAIN AND PERSISTENCE
    - When Taco cloud customers register with the application, they will give you their full name, address and phone no. You can use this information to pre-populate the order form. To capture all of that information, you will create a User class, as follows:

package tacos;

import java.util.Arrays;

import java.util.Collection;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.core.userdetails.UserDetails;

import lombok.AccessLevel;

import lombok.Data;

import lombok.NoArgsConstructor;

import lombok.RequiredArgsConstructor;

@Data

*@NoArgsConstructor*(access = *AccessLevel*.***PRIVATE***, force = true)

*@RequiredArgsConstructor*

*@Entity*

public class User implements UserDetails {

private static final long ***serialVersionUID*** = 1L;

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***AUTO***)

private Long id;

private final String username;

private final String password;

private final String fullName;

private final String street;

private final String city;

private final String state;

private final String zip;

private final String phoneNumber;

/\*returns collection indicating that all users will have been granted ROLE\_USER authority\*/

*@Override*

public Collection<? extends GrantedAuthority> getAuthorities() {

return Arrays.*asList*(new SimpleGrantedAuthority("ROLE\_USER"));

}

*@Override*

public boolean isAccountNonExpired() {

return true;

}

*@Override*

public boolean isAccountNonLocked() {

return true;

}

*@Override*

public boolean isCredentialsNonExpired() {

return true;

}

*@Override*

public boolean isEnabled() {

return true;

}

}

* + - You can now define userRepository:

package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.User;

public interface UserRepository extends CrudRepository<User, Long> {

User findByUsername(String username);

}

* + - * findByUsername() method will be used in user details service to look up a User by their username.
    - CREATING A USER-DETAILS SERVICE
      * Spring security’s UserDetailsService is a straightforward interface:

Public interface userDetailsService{

UserDetails loadUserByUsername(String username) Throws UsernameNotFoundException;

}

* + - * + This implementation of this interface are given a user’s username and are expected to either return a UserDetails object or throw an exception.
        + Because your User class implements UserDetails, and because UserRepository provides a findByUsername() method, they’re perfectly suitable for use in a custom UserDetailsService implementation.
      * The following is the UserDetailsService you will use in Taco cloud application:

package tacos.security;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

import tacos.User;

import tacos.data.UserRepository;

/\*Stereotype annotation to flag this class for spring's component scanning\*/

@Service

public class UserRepositoryUserDetailsService implements UserDetailsService {

private UserRepository userRepo;

public UserRepositoryUserDetailsService(UserRepository userRepo) {

this.userRepo = userRepo;

}

/\*

\* called findByUsername() from userRepo to look up a user. The

\* loadUserByUsername must never return null so if the method returns null then

\* it will throw UsernameNotFoundExcpetion.

\*/

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

User user = userRepo.findByUsername(username);

if(user != null) {

return user;

}

throw new UsernameNotFoundException("User '" + username + "' not found");

}

}

* + - * You need to configure your custom user details service with Spring security

package tacos.security;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

/\* This method is used to configure the user store

\*AuthenticationManagerBuilder employs builder-styole API to configure authentication details

\*/

@Autowired

private UserDetailsService userDetailsService;

/\*

\* Configure password encoder for JDBC-based authentication to encode the

\* password in the database.

\*/

@Bean

public PasswordEncoder encoder() {

return new BCryptPasswordEncoder();

}

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception{

/\*

\* o passing the UserDetailsService instance to userDetailsService() method that is

\* autowired into SecurityConfig

\* o injecting passwordencoder into user details service configuration by calling passwordEncoder()

\*

\*/

auth

.userDetailsService(userDetailsService)

.passwordEncoder(encoder());

}

}

* + - * It appears that encoder() method is called and its return value is passed to passwordEncoder().
      * However, since encoder() method is annotated with @bean, it will be used to declare a PasswordEncoder bean in the spring application context. Any calls to encoder() will then be intercepted to return the bean instance from the application context.
      * Now you need to create a registration page for Taco cloud patrons to register with the application.
    - REGISTERING USERS
      * The RegistrationController class below presents and processes registration forms.

package tacos.security;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import tacos.data.UserRepository;

*@Controller*

*@RequestMapping*("/register")

public class RegistrationController {

private UserRepository userRepo;

private PasswordEncoder passwordEncoder;

*@Autowired*

public RegistrationController(UserRepository userRepo, PasswordEncoder passwordEncoder){

this.userRepo = userRepo;

this.passwordEncoder = passwordEncoder;

}

*@GetMapping*

public String registrationForm() {

return "registration";

}

*@PostMapping*

public String processRegistration(RegistrationForm form) {

userRepo.save(form.toUser(passwordEncoder));

return "redirect:/login";

}

}

* + - * The following is the thymeleaf template that defines the view:

<!DOCTYPE html>

<html xmlns = "http://www.w3.org"

xmlns:th = "http://www.thymeleaf.org">

<head>

<title>Taco Cloud</title>

</head>

<body>

<h1>Register</h1>

<img th:src = "@{/images/TacoCloud.png}"/>

<form method = "POST" th:action = "@{/register}" id = "registerForm">

<label for = "username">Username: </label>

<input type = "text" name = "username"/><br/>

<label for = "password">Password: </label>

<input type = "password" name = "password"/><br/>

<label for = "confirm">Confirm password: </label>

<input type = "password" name = "confirm"/><br/>

<label for = "fullname">Full name: </label>

<input type = "text" name = "fullname"/><br/>

<label for = "street">Street: </label>

<input type = "text" name = "street"/><br/>

<label for = "city">City: </label>

<input type = "text" name = "city"/><br/>

<label for = "state">State: </label>

<input type = "text" name = "state"/><br/>

<label for = "zip">Zip: </label>

<input type = "text" name = "phone"/><br/>

<label for = "phone">Phone: </label>

<input type = "text" name = "phone"/><br/>

<input type = "submit" value = "Register"/>

</form>

</body>

</html>

* + - * When the form is submitted, the HTTP POST request will be handled by processRegistration() method. The RegistrationForm object given to processRegistration() is bound to the request data and is defined with the following class:
* package tacos.security;
* import org.springframework.security.crypto.password.PasswordEncoder;
* import lombok.Data;
* import tacos.User;
* @Data
* public class RegistrationForm {
* private String username;
* private String password;
* private String fullname;
* private String street;
* private String city;
* private String state;
* private String zip;
* private String phone;

* /\*
* \* uses the properties to create a new User object, which is what
* \* processRegistration() will save,using the injected UserRepository
* \*/
* public User toUser(PasswordEncoder passwordEncoder) {
* return new User(
* username, passwordEncoder.encode(password),
* fullname, street, city, state, zip, phone
* );
* }
* }
  + - * toUser() method uses the instance variables to create a new User object which is what processRegistration() will save, using the injected UserRepository.
      * You know that RegistrationController is injected with a passwordEncoder. This is the exact same PasswordEncoder bean you declared in SecurityConfig class.
      * When processing the form submission, RegistrationController passes it to the toUser() method, which uses it to encode the password before saving it to the database. In this way, the submitted password is written in an encoded form, and the user details service will be able to authenticate against the encoded password.
      * Now, if you start the application you can’t even get to the registration page without being prompted to log in. That’s because, by default, all the requests require authentication. Let’s look at how web requests are intercepted and secured so that you can fix this problem.
* 4.3 SECURING WEB REQUESTS
  + Homepage, login page and registration page should be available to unauthenticated users
  + Let’s look at WebSecurityConfigurerAdapter’s other configure() method

@Override

Protected void configure(HttpSecurity http) throws Exception{

…

}

* + - This configure() method accepts HttpSecurity object which can be used to configure how security is handled at the web level.
    - You can configure following with HttpSecurity:
      * Requiring that certain security conditions be met before allowing a request to be served.
      * Configuring a custom login page.
      * Enabling users to log out of the application
      * Configuring cross-site request forgery protection.
* 4.3.1 SECURING REQUESTS
  + You need to ensure that the request for /design and /orders are only available to authenticated users. All other requests should be permitted for all users.
  + The following configure() implementation does exactly that

*@Override*

protected void configure(HttpSecurity http) throws Exception{

http

.authorizeRequests()

.antMatchers("/design","/orders")

.hasRole("ROLE\_USER")

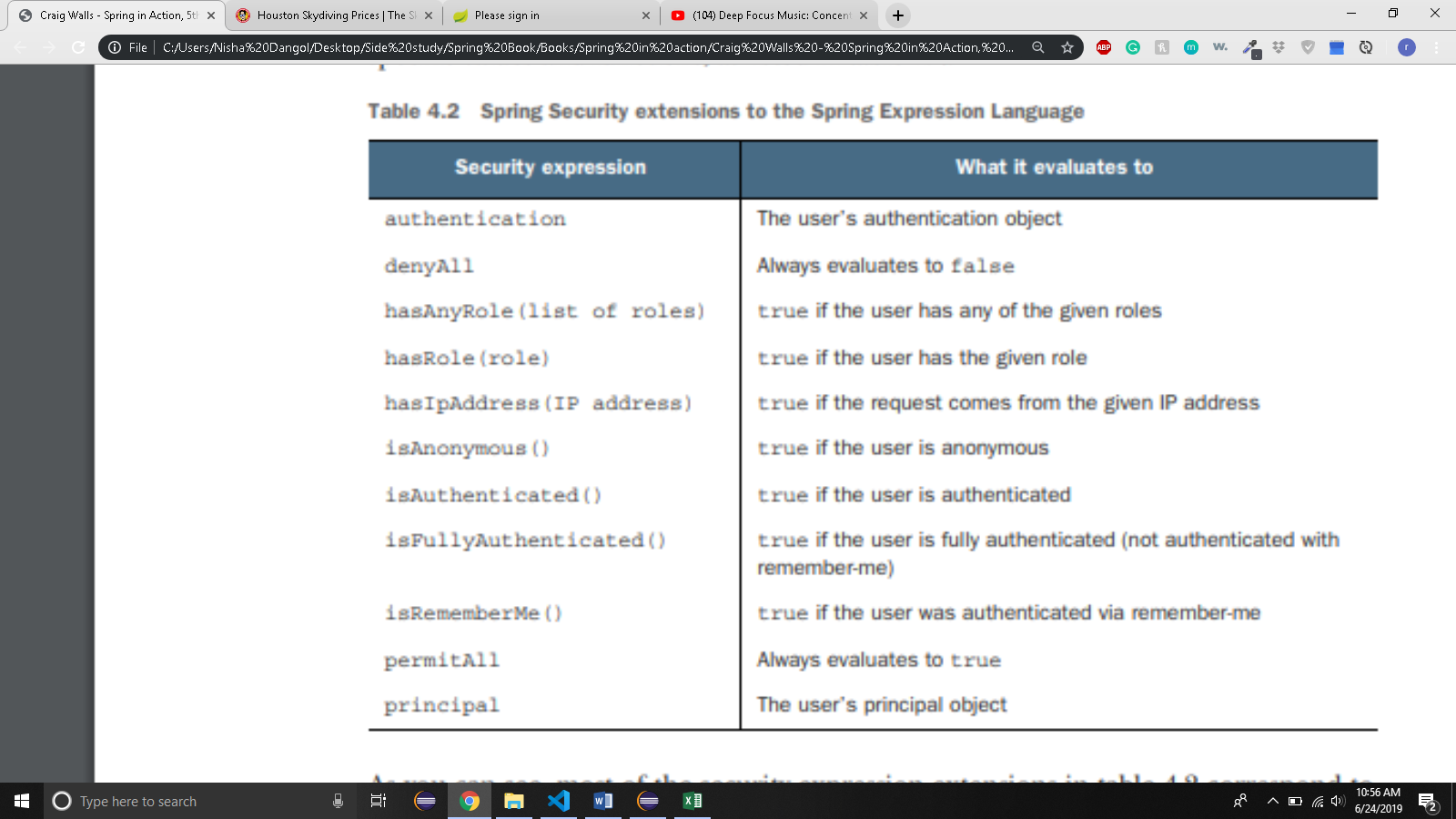
.antMatchers("/","/\*\*").permitAll();

}

* + - The call to authorizeRequests() returns an object (ExpressionInterceptUrlRegistry) on which you can specify URL paths and patterns and the security requirements for those paths.
    - In this case you specify two security rules:
      * Requests for /design and /orders should be for users with a granted authority of ROLE\_USER.
      * All other requests should be permitted to all users.
    - Security rules declared first take precedence over those declared lower down.
    - The hasRole() and permitAll() are just a couple of methods for declaring security requirements for request paths. We have more:



* + - The above security rules are self limiting. You can use access() method to provide a SpEL expression to declare richer security rules. Spring security extends SpEL to include several security specific values and functions.



* + - You can rewrite the configure method as follows:

*@Override*

protected void configure(HttpSecurity http) throws Exception{

http

.authorizeRequests()

.antMatchers("/design","/orders")

.access("hasRole('ROLE\_USER')")

.antMatchers("/","/\*\*")..access("permitAll");

}

* + - This makes coding very flexible. For instance, suppose that you only wanted users with ROLE\_USER authority to create new Tacos on Tuesdays, you can rewrite the expression as below:

@Override

Protected void configure(HttpSecurity http) throws Exception{

http

.authorizeRequest()

.antMatchers(“/design”,”/orders”)

.access(“hasRole(‘ROLE\_USER’) &&” +

“T(java.util.Calendar).getInstance().get(“ +

“T(java.util.Calendar).DAY\_OF\_WEEK) = = “+

“T(java.util.Calendar). TUESDAY”)

.antMatchers(“/”,”/\*\*”).access(“permitAll”);

}

* + - Simple use of access() and the SpEL expression met the authorization needs for Taco Cloud applications
    - Let’s customize login page
* 4.3.2 CREATING A CUSTOM LOGIN PAGE
  + To replace the built in login page, you first need to tell spring security what path your custom login page will be at. That can be done by calling formLogin() on the HttpSecurity object passed into configure();

*@Override*

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/design", "/orders")

.access("hasRole('ROLE\_USER')")

.antMatchers("/", "/\*\*").access("permitAll")

.and()

.formLogin()

.loginPage("/login");

* + The and() method signifies that you’re finished with the authorization configuration and are ready to apply some additional HTTP configuration. You’ll use and() several times as you begin new sections of configuration.
  + You call formLogin() to start configuring you custom login form.
  + The call to loginPage() after that designates the path where you custom login page will be provided.
    - When spring security determines that the user is unauthenticated and needs to log in, it will redirect them to this path.
  + You controller will do nothing but provide view. So you can just declare it in webconfig.

@Override

public void addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/").setViewName("home");

registry.addViewController("/login");

* + Finally you need to define login page view itself

<!DOCTYPE html>

<html xmlns = "www.w3.org/1999/xhtml"

xmlns:th = "www.thymeleaf.org">

<head>

<title>Taco Cloud</title>

</head>

<body>

<h1>Login</h1>

<img th:src = "@{/images/TacoCloud.png}"/>

<div th:if = "${error}">

Unable to login. Check your username and password.

</div>

<p>New here? Click

<a th:href = "@{/register}">here</a> to register

</p>

<form method = "POST" th:action = "@{/login}" id = "loginForm">

<label for = "username">Username: </label>

<input type = "text" name = "username" id = "username"/><br/>

<label for = "password">Password: </label>

<input type = "password" name = "password" id = "password"/><br/>

<input type = "submit" value = "Login"/>

</form>

</body>

</html>

* + By default, Spring Security listens for login requests at /login and expects that the username and password fields be names username and password. The is configurable however. For example, the following configuration customizes the path and field names:

.and()

.formLogin()

.loginPage(“/login”)

.loginProcessingUrl(“/authenticate”)

.usernameParameter(“user”)

.passwordParameter(“pwd”)

* + Here you specify spring security should listen for requests to /authenticate to handle login submissions. Also, the username and password fields should now be named user and pwd.
  + By default, a successful login will take the user directly to the page that they were navigating to when Spring Security determinded that they needed to log in.
  + If the user were to directly navigate to the login page, a successful login would take them to the root path(homepage.
    - But You can specify a default success page:

.and()

.formLogin()

.loginPage("/login")

.defaultSuccessUrl("/design");

* + - Now, After successful log in, the user will be directed to the /design page.
  + You can also force the user to the design page after login, even if they were navigating elsewhere prior to logging in, by passing true as a second parameter.

and()

.formLogin()

.loginPage("/login")

.defaultSuccessUrl("/design",true);

* LOGGING OUT
  + You simply need to call logout on the HttpSecurity object

.and()

.logout()

.logoutSuccessUrl("/");

* In this case, users will be sent to homepage after logout
  + This sets up a security filter that intercepts POST requests to /logout. Therefore, to provide logout capability, you just need to add a logout form and button to the views in your application:

<form method = "POST" th:action = "@{/logout}">

<input type = "submit" value = "Logout"/>

</form>

* + When the user clicks the button, their session will be cleared and they will be logged out of the application. By default they will be redirected to the login page where they can log in again. But you can change that by logoutSuccessFilter().
* 4.3.4 PREVENTING CORSS-SITE REQUEST FORGERY(CSRF)
  + CSRF is a common security attack that involves subjecting a user to code on a maliciously designed web page that automatically (and usually secretly) submits a form to another application on behalf of a user who is often the victim of the attack.
  + To protect against such attacks, applications can generate a CSRF token upon displaying a form, place that token in a hidden field, and then stow it for later use on the server. When the form is submitted, the token is sent back to the server along with the rest of the form data.
  + The request is then intercepted by the server and compared with the token that was originally generated. If the token matches, the request is allowed to proceed. Otherwise, the form must have been rendered by an evil website without knowledge of the token generated by the server.
  + Spring security has built in CSRF protection. It is enabled by default and you don’t need to explicitly configure it. You only need to make sure that any forms your application submits include a field name \_csrf that contains the CSRF token.
  + Spring security even makes that easy by placing the CSRF token in a request attribute with the name \_csrf. Therefore, you could render the CSRF token in a hidden field with the following in a Thymeleaf template.
    - <input type = “hidden” name = “\_csrf” th:value = “${\_csrf.token}”/>
  + If you’re using spring mvc’s usp tag library or Thymeleaf with spring security dialect, you don’t even need to explicitly write hidden field.
  + In thymeleaf, you just need to make sure that one of the attributes of the <form> element is prefixed as a Thymeleaf attribute. For example, th:action attribute is all you need for Thymeleaf to render the hidden field for you.
  + You can disable csrf by doing following  
    http.csrf().disable();

http.headers().frameOptions().disable();

* You do the above so that you can connect to the h2 database. Other wise, it won’t let you.
* 4.4 KNOWING YOUR USER
  + Often you need to know who logged in so that you can tailor their experience.
  + For example, in OrderController, when you initially create the Order object that’s bound to the order form, it’d be nice if you could prepopulate the Order with the user’s name and address so they don’t have to reenter it for each order.
  + Perhaps, even more important, when you save their order, you should associate the Order entity with the User that created the order.
  + To achieve the desired connection between an Order entity and a User entity, you need to add a new propery in the Order class:

@ManyToOne

Private User user;

* + - The @ManyToOne annotation indicates that ann order belongs to a single user and conversely that a user may have many orders.
  + In OrderController, the processOrder() is responsible for saving an order. It will need to tbe modified to determine who the authenticated user is and to call setUser() on the Order object to connect the order with the user.
  + There are several ways to determine who the user is. Some of the most common ways are:
    - Inject a Principal object into the controller method
    - Inject an Authentication object into the controller method
    - Use SecurityContextHolder to get at the security context
    - Use an @AuthenticationPrincipal annotated method
  + For example, you could modify processOrder() to accept a java.security.Principal as a parameter. You could then use the principal name to look up the user from a UserRepository:

@PostMapping

Public String processOrder(@Valid Order order, Errors errors, SessionStatus sessionStatus, Principal principal){

…….

User user = UserRepository.findByUsername(principal.getName());

order.setUser(user);

…….

}

* + This works fine but litter the code. You can trim down some of the security specific code by modifying processOrder() to accept an Authentication object as a parameter instead of Principal;

@PostMapping

Public String processOrder(@Valid Order order, Errors erros, SessionStatus sessionStatus, Authentication authentication){

…..

User user = (User) authentication.getPrincipal();

Order.setUser(user);

….

}

* + With the Authentication, you can call getPrincipal() to get the principal object which, in this case, is a User. Note that getPrincipal() returns a java.util.Object so you need to cast it to User.
  + Perhaps, The cleanest solution of all is to simply accept a User object in processOrder(), but annotate it will @AuthenticationPrincipal so that it will be the authentication’s principal:

@PostMapping

Public String processOrder(@Valid Order order, Errors errors, SessionStatus sessionStatus, @AuthenticationPrincipal User user){

If(errors.hasErrors()){

Return “orderForm”;

}

order.setUser(user);

orderRepo.save(order);

sessionStatus.setComplete();

return “redirect:/”;

}

* + @AuthenticationPrincipal doen’t require a cast and it limits the security specific code to the annotation itself. By the time you get the User object in processOrder(), it’s ready to be used and assigned to the Order.