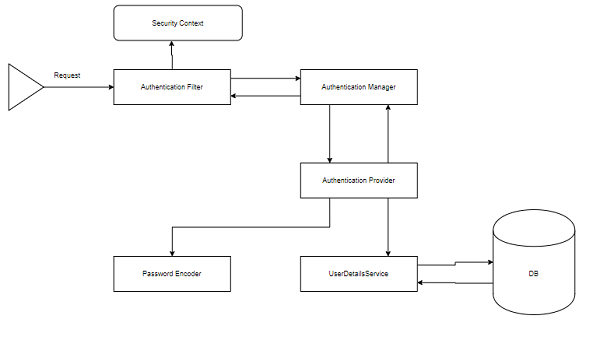
**https://www.tutorialspoint.com/spring\_security**

* Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.
* Authentication is the process of knowing and identifying the user that wants to access.
* **Authorization** is the process to allow authority to perform actions in the application.
* Components of Spring Security Architecture



### **AuthenticationFilter**

This is the filter that intercepts requests and attempts to authenticate it. In Spring Security, it converts the request to an Authentication Object and delegates the authentication to the AuthenticationManager.

### **AuthenticationManager**

It is the main strategy interface for authentication. It uses the lone method authenticate() to authenticate the request. The authenticate() method performs the authentication and returns an Authentication Object on successful authentication or throw an AuthenticationException in case of authentication failure. If the method can’t decide, it will return null.

### **AuthenticationProvider**

* The AuthenticationManager is implemented by the ProviderManager which delegates the process to one or more AuthenticationProvider instances.
* Any class implementing the AuthenticationProvider interface must implement the two methods – authenticate() and supports().
* It is used to check if the particular authentication type is supported by our AuthenticationProvider implementation class. If it is supported it returns true or else false.
* Next, the authenticate() method. Here is where the authentication occurs. If the authentication type is supported, the process of authentication is started.
* Here is this class can use the loadUserByUsername() method of the **UserDetailsService** implementation. If the user is not found, it can throw a UsernameNotFoundException.
* If the user is found, then the authentication details of the user are used to authenticate the user. For example, in the basic authentication scenario, the password provided by the user may be checked with the password in the database. If they are found to match with each other, it is a success scenario. Then we can return an Authentication object from this method which will be stored in the Security Context

### **UserDetailsService**

It is one of the core interfaces of Spring Security. The authentication of any request mostly depends on the implementation of the UserDetailsService interface. It is most commonly used in database backed authentication to retrieve user data. The data is retrieved with the implementation of the lone loadUserByUsername() method where we can provide our logic to fetch the user details for a user. The method will throw a UsernameNotFoundException if the user is not found.

### **PasswordEncoder**

Until Spring Security 4, the use of PasswordEncoder was optional. The user could store plain text passwords using in-memory authentication. But Spring Security 5 has mandated the use of PasswordEncoder to store passwords. This encodes the user’s password using one its many implementations. The most common of its implementations is the BCryptPasswordEncoder. Also, we can use an instance of the NoOpPasswordEncoder for our development purposes. It will allow passwords to be stored in plain text. But it is not supposed to be used for production or real-world applications.

### **Spring Security Context**

This is where the details of the currently authenticated user are stored on successful authentication. The authentication object is then available throughout the application for the session. So, if we need the username or any other user details, we need to get the SecurityContext first. This is done with the SecurityContextHolder, a helper class, which provides access to the security context. We can use the setAuthentication() and getAuthentication() methods for storing and retrieving the user details respectively.

### **Login with a Database**

As we discussed, Spring Security automatically provides an in-memory authentication implementation by default. We can override this by authenticating users whose details are stored in a database. In this case, while authenticating a user, we can verify the credentials provided by the user against those in the database for authentication. We can also let new users register in our application and store their credentials in the same database. Also, we can provide methods to change or update their passwords or roles or other data. As a result, this provides us with persistent user data which can be used for longer periods of time.

### **Login Attempts Limit**

To limit login attempts in our application we can use Spring Security’s isAccountNonLocked property. Spring Security’s UserDetails provides us with that property. We can set up an authentication method wherein, if any user or someone else provides incorrect credentials for more than a certain number of times, we can lock their account. Spring Security disables authentication for a locked user even if the user provides correct credentials. This is an in-built feature provided by Spring Security. We can store the number of incorrect login attempts in our database. Then against each incorrect authentication attempt, we can update and check with the database table. When the number of such attempts exceeds a given number, we can lock the user out of their account. Consequently, the user will not be able to log in again until their account is unlocked.

**User.java**

package com.tutorial.spring.security.formlogin.model;

import java.util.Collection;

import java.util.List;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

import org.springframework.security.core.GrantedAuthority;

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

@Entity

@Table(name = "users")

public class User implements UserDetails {

/\*\*

\*

\*/

private static final long serialVersionUID = 1L;

@Id

private String username;

private String password; @Column(name = "account\_non\_locked")

private boolean accountNonLocked;

public User() {

}

public User(String username, String password, boolean accountNonLocked) {

this.username = username;

this.password = password;

this.accountNonLocked = accountNonLocked;

}

@Override

public Collection< extends GrantedAuthority> getAuthorities() {

return List.of(() -> "read");

}

@Override

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

@Override

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

@Override

public boolean isAccountNonExpired() {

return true;

}

@Override

public boolean isAccountNonLocked() {

return accountNonLocked;

}

@Override public boolean isCredentialsNonExpired() {

return true;

}

@Override public boolean isEnabled() {

return true;

}

public void setAccountNonLocked(Boolean accountNonLocked) {

this.accountNonLocked = accountNonLocked;

}

public boolean getAccountNonLocked() {

return accountNonLocked;

}

}

**Attempts.java**

package com.tutorial.spring.security.formlogin.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Attempts {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String username;

private int attempts;

/\*\*

\* @return the id

\*/

public int getId() {

return id;

}

/\*\*

\* @param id the id to set

\*/

public void setId(int id) {

this.id = id;

}

/\*\*

\* @return the username

\*/

public String getUsername() {

return username;

}

/\*\*

\* @param username the username to set

\*/

public void setUsername(String username) {

this.username = username;

}

/\*\*

\* @return the attempts

\*/

public int getAttempts() {

return attempts;

}

/\*\*

\* @param attempts the attempts to set

\*/

public void setAttempts(int attempts) {

this.attempts = attempts;

}

}

**UserRepository.java**

package com.tutorial.spring.security.formlogin.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.tutorial.spring.security.formlogin.model.User;

@Repository public interface UserRepository extends JpaRepository<User, String> {

Optional<User> findUserByUsername(String username);

}

**AttemptsRepository.java**

package com.tutorial.spring.security.formlogin.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.tutorial.spring.security.formlogin.model.Attempts;

@Repository

public interface AttemptsRepository extends JpaRepository<Attempts, Integer> {

Optional<Attempts> findAttemptsByUsername(String username);

}

### **Configuration**

Since we are going to use a custom login form, we have to override the default configuration of Spring Security. To do this we create our configuration class which extends the WebSecurityConfigurerAdapter class of Spring Security.

package com.tutorial.spring.security.formlogin.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

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

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

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

@Configuration

public class ApplicationConfig extends WebSecurityConfigurerAdapter {

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.csrf().disable()

.authorizeRequests().antMatchers("/register\*\*")

.permitAll() .anyRequest().authenticated()

.and()

.formLogin() .loginPage("/login")

.permitAll()

.and()

.logout() .invalidateHttpSession(true)

.clearAuthentication(true) .permitAll();

}

}

Here we did two things−

* First, we have specified the implementation of the PasswordEncoder interface that we are going to use. We have used an instance of BCryptPasswordEncoder to encode our passwords for this example. The PasswordEncoder interface has many implementations and we can use any of them. We have chosen BCryptPasswordEncoder here as it the most commonly used implementation. It uses the very strong BCrypt hashing algorithm to encode the passwords. It does so by incorporating a salt to protect against rainbow table attacks. In addition to this, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power.
* Secondly, we have overridden the configure() method to provide our implementation of the login method.
  + Whenever we use a custom form for authentication in place of the one provided by Spring Security, we have to inform Spring Security of it using the formLogin() method.
  + We then also specify our login URL – /login. We will map the URL to our custom login page in our Controller later.
  + We have also specified that the endpoints starting with /register, /login and the logout page need not be protected. We did so using the permitAll() method. This allows everyone to access these endpoints. Other than these endpoints, all endpoints are to be authenticated(). That is to say, users must be logged in to access all the other endpoints.
  + On logout, we have specified that the session is to be invalidated and authentication stored in the application’s SecurityContext be cleared.

### **Security Setup**

Now, we will setup our authentication process. We are going to setup authentication using a database and locking of user accounts.

Let’s create our implementation of UserDetailsService first. As we have discussed before, we need to provide our custom implementation for authentication using a database. This is because, Spring Security, as we know, only provides an in-memory authentication implementation by default. Therefore, we need to override that implementation with our database based process. To do so, we need to override the loadUserByUsername() method of UserDetailsService.

### **UserDetailsService**

package com.tutorial.spring.security.formlogin.security;

import java.util.Optional;

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

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

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

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

import org.springframework.security.provisioning.UserDetailsManager;

import org.springframework.stereotype.Service;

import com.tutorial.spring.security.formlogin.model.User;

import com.tutorial.spring.security.formlogin.repository.UserRepository;

@Service

public class SecurityUserDetailsService implements UserDetailsService {

@Autowired

private UserRepository userRepository;

@Override

public UserDetails loadUserByUsername(String username)

throws UsernameNotFoundException {

User user = userRepository.findUserByUsername(username)

.orElseThrow(() -< new UsernameNotFoundException("User not present"));

return user;

}

public void createUser(UserDetails user) {

userRepository.save((User) user);

}

}

### **Authentication Provider**

We will now implement our custom authentication provider. It will implement the AuthenticationProvider interface. We have two methods here that we have to override and implement.

package com.tutorial.spring.security.formlogin.security;

import java.util.Optional;

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

import org.springframework.security.authentication.AuthenticationProvider;

import org.springframework.security.authentication.BadCredentialsException;

import org.springframework.security.authentication.LockedException;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.AuthenticationException;

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

import org.springframework.stereotype.Component;

import com.tutorial.spring.security.formlogin.model.Attempts;

import com.tutorial.spring.security.formlogin.model.User;

import com.tutorial.spring.security.formlogin.repository.AttemptsRepository;

import com.tutorial.spring.security.formlogin.repository.UserRepository;

@Component public class AuthProvider implements AuthenticationProvider {

private static final int ATTEMPTS\_LIMIT = 3;

@Autowired private SecurityUserDetailsService userDetailsService;

@Autowired private PasswordEncoder passwordEncoder;

@Autowired private AttemptsRepository attemptsRepository;

@Autowired private UserRepository userRepository;

@Override

public Authentication authenticate(Authentication authentication)

throws AuthenticationException {

String username = authentication.getName();

Optional<Attempts>

userAttempts = attemptsRepository.findAttemptsByUsername(username);

if (userAttempts.isPresent()) {

Attempts attempts = userAttempts.get();

attempts.setAttempts(0); attemptsRepository.save(attempts);

}

}

private void processFailedAttempts(String username, User user) {

Optional<Attempts>

userAttempts = attemptsRepository.findAttemptsByUsername(username);

if (userAttempts.isEmpty()) {

Attempts attempts = new Attempts();

attempts.setUsername(username);

attempts.setAttempts(1);

attemptsRepository.save(attempts);

} else {

Attempts attempts = userAttempts.get();

attempts.setAttempts(attempts.getAttempts() + 1);

attemptsRepository.save(attempts);

if (attempts.getAttempts() + 1 >

ATTEMPTS\_LIMIT) {

user.setAccountNonLocked(false);

userRepository.save(user);

throw new LockedException("Too many invalid attempts. Account is locked!!");

}

}

}

@Override public boolean supports(Class<?> authentication) {

return true;

}

}

* authenticate() − This method returns a fully authenticated object including credentials on successful authentication. This object is then stored in the SecurityContext. To perform authentication we will use the loaduserByUsername() method of the SecurityUserDetailsService class of our Application. Here we perform multiple things −
  + First, we extract the user credentials from the Authentication request object which is passed as a parameter to our function. This authentication object was prepared by the AuthenticationFilter class and passed down the AuthenticationProvider through the AuthenticationManager.
  + We also fetch the user details from the database using the loadUserByUsername() method.
  + Now, first, we check if the user account has been locked due to previous failed authentication attempts. If we find that the account is locked, we throw a LockedException, and the user will be unable to authenticate unless the account is unlocked again.
  + If the account is not locked, we match the provided password along with the one stored against the user in the database. This is done using the matches() method of the PasswordEncoder interface.
  + If the passwords match, and the account has not been locked by then, we return a fully authenticated object. Here we have used an instance UsernamePasswordAuthenticationToken class (as it is a username-password authentication) that implements Authentication. Meanwhile, we also reset the attempts counter to 0.
  + On the other hand, if the password doesn’t match, we check for a few conditions −
    - If it is the user’s first attempt, then, probably his name would not be in the database. We check for this using the method findAttemptsByUsername() from the AttemptsRepository.
    - If not found, we make an entry for the user in the database, with the number of attempts set to one.
    - If a user is found, then we increase the number of attempts by 1.
    - We then check against the maximum number of failed attempts allowed, using a constant value we defined earlier.
    - If the number is more than the allowed number of attempts, then the user is locked our of the application and a LockedException is thrown.
* supports() − We also have the supports method that checks if our authentication type is supported by our AuthenticationProvider implementation class. It returns true, false, or null if it matches, doesn’t match, or if it can’t decide respectively. We have hardcoded it to be true for now.

### **Controller**

package com.tutorial.spring.security.formlogin.controller;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpSession;

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

import org.springframework.http.MediaType;

import org.springframework.security.authentication.BadCredentialsException;

import org.springframework.security.authentication.LockedException;

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.RequestParam;

import com.tutorial.spring.security.formlogin.model.User;

import com.tutorial.spring.security.formlogin.security.SecurityUserDetailsService;

@Controller

public class HelloController {

@Autowired private SecurityUserDetailsService userDetailsManager;

@Autowired

private PasswordEncoder passwordEncoder;

@GetMapping("/")

public String index() {

return "index";

}

@GetMapping("/login")

public String login(HttpServletRequest request, HttpSession session) {

session.setAttribute(

"error", getErrorMessage(request, "SPRING\_SECURITY\_LAST\_EXCEPTION")

);

return "login";

}

@GetMapping("/register")

public String register() {

return "register";

}

@PostMapping(

value = "/register",

consumes = MediaType.APPLICATION\_FORM\_URLENCODED\_VALUE, produces = {

MediaType.APPLICATION\_ATOM\_XML\_VALUE, MediaType.APPLICATION\_JSON\_VALUE }

)

public void addUser(@RequestParam Map<String, String> body) {

User user = new User(); user.setUsername(body.get("username"));

user.setPassword(passwordEncoder.encode(body.get("password")));

user.setAccountNonLocked(true); userDetailsManager.createUser(user);

}

private String getErrorMessage(HttpServletRequest request, String key) {

Exception exception = (Exception) request.getSession().getAttribute(key);

String error = "";

if (exception instanceof BadCredentialsException) {

error = "Invalid username and password!";

} else if (exception instanceof LockedException) {

error = exception.getMessage();

} else {

error = "Invalid username and password!";

}

return error;

}

}