

---

# UNIT 12 SPRING SECURITY CONFIGURATION

---

## Structure

- 12.0 Introduction
- 12.1 Objective
- 12.2 Introduction to Web Securities
  - 12.2.1 Introduction of Java Cryptography Architecture (JCA)
  - 12.2.2 Introduction of Java Secure Socket Extension (JSSE)
- 12.3 Issues and Challenges of Web Security
- 12.4 Spring Security Overview
- 12.5 Java Based Configuration
- 12.6 Create Spring Initializer Class
- 12.7 Create Controller and View
- 12.8 Run Application
- 12.9 Summary
- 12.10 Solutions/Answers to Check Your Progress
- 12.11 References/Further Reading

---

## 12.0 INTRODUCTION

---

In unit 9, Spring Boot Application and a simple application without any security considerations are described. This unit will describe security features like authentication and authorization that create a secured Java Enterprise Application. The following sections will explain JCA and JSSE to secure our data at various levels. This unit will also describe how spring security simplifies the security implementation as compared to non-spring projects.

---

## 12.1 OBJECTIVE

---

After going through this unit, you should be able to:

- ... secure the data using JCA,
- ... establish Secure communication between client and server using JSSE,
- ... explain common web application security vulnerabilities,
- ... perform Authentication (AuthN) and Authorization (AuthZ),
- ... describe building blocks of Spring Security, and
- ... develop the secured applications using spring security.

---

## 12.2 INTRODUCTION TO WEB SECURITIES

---

Security is one of the topmost concern for any application. It includes many aspects such as sensitive data safety, robust website against hacking, sql injection, csrf etc. A different set of tools are used for various aspects of application security. This section describes how to secure sensitive information on backend applications, i.e. passwords

in the database, critical records etc. With the emergence of cloud services, security has become an indispensable requirement for sensitive information on every platform.

### Encryption of data in-transit and at-rest

Data **in-transit** is data moving from/ through the internet or private network. Term Data protection in transit is the protection of data while it is moving from network to network or is transferred from a local storage device to a cloud storage device.

The term Data **at-rest** is data that is not moving from device to device or network to networks such as data stored on a hard drive, laptop or archived/stored in some other way. Data protection at rest aims to secure inactive data stored on any device or network.

Safeguarding sensitive data both **in-transit** and **at-rest** is vital for modern enterprises as attackers are persistently trying innovative ways to compromise systems and steal data. Data can be exposed to many security threats, both in-transit and at-rest, which requires safeguard against such threats. There are multiple approaches to protect data in-transit and at-rest. Encryption plays a vital role in data protection and is a very popular tool for securing data both in-transit and at-rest. Enterprises often encrypt sensitive data prior to move and/or use encrypted connections such as HTTPS, SSL, TLS, FTPS etc., to protect the data in transit. Enterprises can simply encrypt sensitive data prior to storing them and/or choose to encrypt the storage drive itself.

## 12.2.1 INTRODUCTION OF JAVA CRYPTOGRAPHY ARCHITECTURE (JCA)

### Cryptography

*Cryptography* is the art and science of making a cryptosystem that is capable of providing information security. Cryptography aims to provide the following:

- ... Secret Communication
- ... Authentic Information
- ... Store Sensitive Information

### Cryptography Primitives

*Cryptography primitives* are simply the tools and techniques in Cryptography that can be particularly used to provide a set of desired security services –

- ... Encryption
- ... Hash functions
- ... Message Authentication codes (MAC)
- ... Digital Signatures

### Java Cryptography Architecture (JCA)

The *Java Cryptography Architecture (JCA)* is based on “provider” architecture. It is a set of APIs to cryptography such as encryption, message digests, key generation and management, digital signatures, certificates etc. It enables you to encrypt and decrypt data in java, manage keys, sign and authenticate messages etc.

JCA provides some general-purpose classes and interfaces. JCA has “provider” based architecture; thus actual functionality and implementation are provided by providers. The encryption algorithm used for encryption and decryption in Cipher class depends on the concrete provider used.

### Core Classes and Interfaces

Following java packages has been defined into Java cryptography API.

- ... javax.crypto
- ... javax.crypto.spec
- ... javax.crypto.interfaces

```
... java.security
... java.security.cert
... java.security.spec
... java.security.interfaces
```

Above defines, packages contain many core classes and interfaces. Some of the frequently used ones are as follows:

```
... Provider
... Cipher
... MessageDigest
... Signature
... Mac
... KeyPairGenerator
... KeyGenerator
... KeyStore
```

The following sections will give you insights of some frequently used classes and its feature.

### Provider

JCA has a “provider” based architecture. Java SDK provides a default provider. Default providers may not support the encryption algorithms you want to use. In order to use a different provider instead of the default, Java Cryptography API provides a method to set the desired provider. Cryptography provider can be set as below-

```
Security.addProvider(...)
```

### Cipher

The ***Cipher (javax.crypto.Cipher)*** class is the core of JCA framework. This class provides cryptographic cipher for encryption and decryption. Cipher’s `getInstance` method provides cipher object for encryption and decryption. Method `getInstance` expects the name of the requested transformation. A transformation can be any form either "algorithm/mode/padding" or "algorithm".

Cipher instance can be created using `getInstance` method as –

```
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

Encryption/Decryption example using Cipher Class

```
package com.ignou;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.security.GeneralSecurityException;
import java.util.Base64;
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
public class EncryptDecrypt
{
    public static byte[] Encrypt(String secretKey, String plainText)
        throws GeneralSecurityException
    {
        byte[] keyBytes = secretKey.getBytes(Charset.forName("UTF-8"));
        if (keyBytes.length != 16)
        {
            throw new IllegalArgumentException("Allowed key size is 16");
        }
        IvParameterSpec paramSpec = new IvParameterSpec(new byte[16]);
```

```

SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES");
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, keySpec, paramSpec);
byte[] cipherTextBytes = cipher.doFinal(plainText.getBytes(Charset.forName("UTF-8")));
return cipherTextBytes;
}

public static byte[] Decrypt(String key, byte[] cipherText) throws GeneralSecurityException
{
    byte[] keyBytes = key.getBytes(Charset.forName("UTF-8"));
    if (keyBytes.length != 16)
    {
        throw new IllegalArgumentException("Allowed key size is 16");
    }
    IvParameterSpec paramSpec = new IvParameterSpec(new byte[16]);
    SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES");
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, keySpec, paramSpec);
    byte[] plainTextBytes = cipher.doFinal(cipherText);
    return plainTextBytes;
}

public static void main(String[] args)
{
    try
    {
        String key = "ThisIsASecretKey";
        String plainText = "Hello, Good Morning";
        byte[] ciphertextbytes = Encrypt(key, plainText);
        System.out.println("Cipher Text: " + Base64.getEncoder().encodeToString(ciphertextbytes));
        byte[] plaintextbytes = Decrypt(key, ciphertextbytes);
        String decryptedPlainText = new String(plaintextbytes, StandardCharsets.UTF_8);
        System.out.println("Decrypted Plain Text: "+decryptedPlainText);
    }
    catch (GeneralSecurityException e)
    {
        e.printStackTrace();
    }
}
}

```

**Output:** Program execution will provide the below output. As you can see by encryption, you are getting encrypted text corresponding to plain text. This Cipher text is used for confidentiality. This encrypted text has been decrypted to the plain text using Cipher class in Decrypt mode.

```

Cipher Text: ww5BTFTQIgKfRVb4u/yi8Xt62XCGE+J0Q9F1KjFltw=
Decrypted Plain Text: Hello, Good Morning

```

## Message Digest

**Message digests** are a secured one-way hash functions that take arbitrary-sized data and output fixed-length hash value. A common solution to validate the encrypted data on the way to you has not been modified is Message Digest. A message digest is a hash value calculated from message data. Steps to send encrypted data with message digest and validation of encrypted data is as-

- ... Calculate the message digest from data before encryption
- ... Encrypt both data and message digest and send across the wire
- ... Once encrypted data is received, decrypt it to get the message and message digest

- ... Calculate message digest for the received message and compare it with the received message digest
- ... If the comparison evaluates to true, there is a high probability (but not a 100% guarantee) that the data was not modified.

Java provides a class, named **MessageDigest** which belongs to the package `java.security`, to create a message digest for message data. This class supports algorithms such as SHA-1, SHA-256, SHA-384, SHA-512, MD5 algorithms to convert an arbitrary-length message to a fixed-length message digest. Sample code to generate message digest for multiple blocks of data is as –

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
```

```
byte[] msg1 = "This is msg1".getBytes("UTF-8");
md.update(msg1);
byte[] msg2 = "This is msg2".getBytes("UTF-8");
md.update(msg2);
```

```
// Call digest method to get the MD
byte[] digest = md.digest();
```

## Mac

The term **MAC** stands for Message Authentication Code. The integrity of information transmitted over an unsecured channel can be validated using MAC. A MAC is like a Message Digest but uses an additional key to encrypt the message digest. Thus, A MAC is a message digest encrypted with a secret key. MAC from a message can be created using Java Mac Class. To verify the MAC of a message, it requires the key along with original data. Hence, a MAC is a more protected way to guard a block of data from modification than a message digest. Java code to calculate the MAC of a message data is as –

```
package com.ignou;
import java.util.Base64;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
public class MAC
{
    public static void main(String[] args) throws Exception
    {
        Mac mac = Mac.getInstance("HmacSHA256");
        byte[] keyBytes = new byte[] {'T','h','i','s','A','S','e','c','r','e','t','K','e','y'};
        String algo = "RawBytes";
        SecretKeySpec key = new SecretKeySpec(keyBytes, algo);
        mac.init(key); // MAC instance initialized with key

        byte[] msg1 = "This is msg1".getBytes("UTF-8");
        mac.update(msg1); // Update that data for which mac should be calculated
        byte[] msg2 = "This is msg2".getBytes("UTF-8");
        mac.update(msg2); // Update that data for which mac should be calculated
        byte[] macBytes = mac.doFinal(); // Calculate MAC for given blocks of data
        System.out.println("Message Authentication Code(MAC): " +
            Base64.getEncoder().encodeToString(macBytes));
    }
}
```

Execution of the above program will give the MAC. MAC helps to validate the integrity of the received msg. The person who receives and knows the key, he/she can validate the integrity of msg.

## Output:

```
Message Authentication Code(MAC): IweNfSG12Dqw05u57fAL29Sy0aYyW5vs22dZUzg1uXo=
```

## Signature

A digital signature is a mathematical technique used to validate the authenticity and integrity of a message or digital document. Digital signatures provide the added assurances of evidence of origin, identity and status of an electronic document or message. It also ensures the non-repudiation, i.e. the author of the message can't later deny that they were the source. Digital Signatures are based on public-key cryptography. The signer uses his private key to create the signature and signature can be authenticated only by the signer public key.

The **Signature** (java.security.Signature) class is used to create Signature instance in java. A digital signature is an encrypted message digest with the private key of the signing authority. The signing authority may be the device, person or organization that is to sign the data.

To create a Signature instance, you call the Signature.getInstance(...) method. Signature instance creation is shown below:

```
Signature signature = Signature.getInstance("SHA256WithDSA");
```

Signature creation and verification example

```
package com.ignou;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
public class CreateAndVerifySignature
{
    public static void main(String[] args) throws Exception
    {
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
        // Initializing the key pair generator
        keyPairGen.initialize(2048);
        // Generate the pair of keys
        KeyPair pair = keyPairGen.generateKeyPair();
        // Get the private key from the key pair
        PrivateKey privateKey = pair.getPrivate();
        // Get the public key from the key pair
        PublicKey publicKey = pair.getPublic();
        // Create signature instance
        Signature sign = Signature.getInstance("SHA256withDSA");
        // Initialize with private key
        sign.initSign(privateKey);
        // Data that needs to be signed
        byte[] bytes = "This is my digital signed data".getBytes();
        // Update signature instance with data to be signed
        sign.update(bytes);
        // Create the signature
        byte[] signature = sign.sign();
        // Initiate the signature with public key to verify signature
        sign.initVerify(pair.getPublic());
        // Update signature instance with data for which signature needs to verify
        sign.update(bytes);
        // Verify the signature
        boolean isValidSignature = sign.verify(signature);

        if (isValidSignature)
        {
            System.out.println("Signature verified");
        }
        else
        {
        }
```

```

        System.out.println("Signature Failed");
    }
}

```

## 🔍 Check Your Progress 1

- 1) Write about JCA with some of the core classes and interface with example

.....  
 .....  
 .....  
 .....

- 2) Write the characteristics of Message Digest (MD) and Message Authentication Code (MAC) and compare MD and MAC.

.....  
 .....  
 .....  
 .....

- 3) Alice wants to send some confidential data securely to Bob over the network. Alice uses digital signature to sign the data and did the following where  $S(m)$  = digital signature of  $m$ .

- ... Compute  $S(m)$  using her private key
- ... Send  $m$ ,  $S(m)$  and her public key to Bob

Bob did as below upon receiving the confidential data:

- ... Bob receive  $S(m)$ ,  $m$  and Alice's public key
- ... Bob verify the  $S(m)$  using  $m$  and Alice's key.

Alice used private key to digitally sign the confidential data. Yet the above approach is attack prone. Figure out the possible attack and flaw in the above approach.

.....  
 .....  
 .....  
 .....

## 12.2.2 INTRODUCTION OF JAVA SECURE SOCKET EXTENSION (JSSE)

Data *in-transit* is vulnerable and can easily be modified or hacked by an unintended recipient. Private or critical data that travel across the network containing information such as password, credit card numbers etc., must be made unintelligible to unauthorized parties. Data integrity must be ensured while data transports. Many protocols have been designed to protect the privacy and integrity of in-transit data. The Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols are among them.

The Java Secure Socket Extension (JSSE) is a framework to enable secure internet communication. This framework provides an implementation for a Java version of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols. It also

provides API's for data encryption, server authentication, message integrity and optional client Authentication.

JSSE provides API framework along with its implementation. It acts as a building block to build secure web applications simply. It reduces the security vulnerabilities by abstracting the complex underlying security algorithms and handshaking mechanisms. The JSSE API boosts up the core network and cryptographic services defined by the `java.security` and `java.net` packages with add-on security.

Supported security protocols by JSSE API are as follows:

- ... TLS: version 1.0, 1.1, and 1.2
- ... SSL: version 3.0
- ... DTLS: versions 1.0 and 1.2

### JSSE Features and Benefits

As per the Oracle JSSE reference guide, important benefits and features of JSSE are as follows:

- ... A standard component of the JDK
- ... Provider-based architecture makes it flexible and extensible
- ... Provides classes to create secure channels such as `SSLSocket`, `SSLServerSocket`, and `SSLEngine`
- ... Initiation or verification of secure communication using cipher suite negotiation
- ... Support for client and server authentication, which is part of the normal SSL/TLS/DTLS handshaking
- ... Encapsulated HTTP in the SSL/TLS protocol, which allows access to web pages using HTTPS
- ... Server session management APIs
- ... Server name indication extension to facilitate secure connections to virtual servers.
- ... Provides support for Server Name Indication (SNI) Extension, which extends the SSL/TLS/DTLS protocols to indicate what server name the client is attempting to connect to during handshaking.
- ... Prevents man-in-the-middle attacks by providing support for endpoint identification during handshaking.

The following section explains the SSL and describes how to implement SSL in Java using JSSE (Java Secure Socket Extension) API.

JSSE provides an SSL toolkit for Java applications. In addition to the necessary classes and interfaces, JSSE provides a handy command-line debugging switch that you can use to watch the SSL protocol in action. A simple example with the below code can be executed in debug mode to watch the handshaking details.

```
package com.ignou;
public class SecureConnection
{
    public static void main(String[] args)
    {
        try
        {
            new java.net.URL("https://www.google.com").getContent();
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```



To run the above application in debug mode, we need to turn on SSL debugging. The application connects to the secure website <https://www.google.com> using the SSL protocol via HTTPS. In the command given below, the first option loads the HTTPS protocol handler, while the second option, the debug option, causes the program to print out its behavior.

```
java -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol -
Djavax.net.debug=sslcom.ignou.SecureConnection
```

The above SecureConnection program using the `java.net.URL` class demonstrates the easiest way to add SSL to your applications. This approach is useful, but it is not flexible enough to let you create a secure application that uses generic sockets. The next section describes SSL and its implementation using JSSE.

In simple terms, we can understand that SSL provides a secured connection between server and client. SSL encrypts the communication between two devices operating over a network connection to provide a secure channel. Secure communication between web browsers and web servers is a very common example of SSL. The following three main information security principles is required for secured communication over a network, and SSL fulfils these:

- ... Encryption: Protects the data transmissions between involved entities
- ... Authentication: Ensures that the server we connect to is actually the intended server
- ... Data integrity: guarantee that data exchange between two entities is not modified by the third party

## JSSE API

Java Security API is based on Factory Design Pattern extensively. JSSE provides many factory methods to instantiate. The following section will outline some important classes in JSSE API and the use of these classes.

### SSLSocketFactory

The `javax.net.ssl.SSLSocketFactory` is used to create `SSLSocket` objects. Methods in the `javax.net.ssl.SSLSocketFactory` class can be categorized into three categories. The first category consist of a single method ***static getDefault()***, that can retrieve the default SSL socket factory: ***static SocketFactorygetDefault()***

The second category consist of five methods that can be used to create `SSLSocket` instances:

- ... Socket createSocket(String host, int port)
- ... Socket createSocket(InetAddress host, int port)
- ... Socket createSocket(String host, int port, InetAddressclientHost, int clientPort)
- ... Socket createSocket(InetAddress host, int port, InetAddressclientHost, int clientPort)
- ... Socket createSocket(Socket socket, String host, int port, booleanautoClose)

The third category consists of two methods which return the list of SSL cipher suites that are enabled by default and the complete list of supported SSL cipher suites:

- ... String [] getDefaultCipherSuites()
- ... String [] getSupportedCipherSuites()

## SSLSocket

This *javax.net.ssl.SSLSocket* class extends *java.net.Socket* class in order to provide secure socket. SSL Sockets are normal stream sockets but with an added layer of security over a network transport protocol, such as TCP. Since it extends *java.net.Socket* class, thus it supports all the standard socket methods and adds methods specific to secure sockets. *SSLSocket* instances construct an SSL connection to a named host at a specified port. This allows binding the client side of the connection to a given address and port. Few important methods in *javax.net.ssl.SSLSocket* are listed below:

```
... String [] getEnabledCipherSuites()
... String [] getSupportedCipherSuites()
... void setEnabledCipherSuites(String [] suites)
... boolean getEnableSessionCreation()
... void setEnableSessionCreation(boolean flag)
... boolean getNeedClientAuth()
... void setNeedClientAuth(boolean need)
```

The methods below change the socket from client mode to server mode. This affects who initiates the SSL handshake and who authenticates first:

```
... boolean getUseClientMode()
... void setUseClientMode(boolean mode)
```

Method *void startHandshake()* forces an SSL handshake. It's possible, but not common, to force a new handshake operation in an existing connection.

### SSLServerSocketFactory

The *SSLServerSocketFactory* class creates *SSLServerSocket* instance. It is quite similar to *SSLSocketFactory*. Like *createSocket* method in *SSLSocketFactory*, *createServerSocket* method is there in *SSLServerSocketFactory* class to get *SSLServerSocket* instance.

### SSLServerSocket

This class is subclass of *ServerSockets* and provides secure server sockets using protocols such as the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols. *SSLServerSocket* creates SSL Sockets by accepting connections.

### Example

The class *SecureEchoServer* uses JSSE API to create a secure server socket. It listens on the server socket for connections from secure clients. When executing the *SecureEchoServer*, you must specify the keystore to use. The keystore contains the server's certificate.

```
package com.ignou;
public class SecureEchoServer
{
    public static void main(String[] arstring)
    {
        try
        {
            SSLServerSocketFactory sslserversocketfactory = (SSLServerSocketFactory)
            SSLServerSocketFactory.getDefault();
            SSLServerSocket sslserversocket = (SSLServerSocket)
            sslserversocketfactory.createServerSocket(8888);
            SSLSocket sslsocket = (SSLSocket) sslserversocket.accept();
            InputStream inputStream = sslsocket.getInputStream();
            InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
            BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
            String text = null;
```

```

        while ((text = bufferedreader.readLine()) != null)
        {
            System.out.println("Received From Client: "+text);
            System.out.flush();
        }
    }
    catch (Exception exception)
    {
        exception.printStackTrace();
    }
}
}

```

The *SecureEchoClient* code used JSSE to securely connect to the server. When running the *SecureEchoClient*, you must specify the truststore to use, which contains the list of trusted certificates.

```

package com.ignou;
public class SecureEchoClient
{
    public static void main(String[] arstring)
    {
        try
        {
            SSLSocketFactory sslsocketfactory = (SSLSocketFactory)
            SSLSocketFactory.getDefault();
            SSLSocket sslsocket = (SSLSocket) sslsocketfactory.createSocket("localhost", 8888);
            InputStream inputStream = System.in;
            InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
            BufferedReader bufferedreader = new BufferedReader(inputStreamReader);
            OutputStream outputStream = sslsocket.getOutputStream();
            OutputStreamWriter outputStreamWriter = new OutputStreamWriter(outputStream);
            BufferedWriter bufferedwriter = new BufferedWriter(outputStreamWriter);
            String text = null;
            System.out.println("Send Text to server");
            while ((text = bufferedreader.readLine()) != null)
            {
                bufferedwriter.write(text + '\n');
                bufferedwriter.flush();
            }
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}

```

The execution of the *SecureEchoServer* without keystore will give exception as **javax.net.ssl.SSLHandshakeException: no cipher suites in common** while execution of *SecureEchoClient* without trustStore will give exception as **javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake\_failure**.

Generate the self-signed SSL certificate using the Java keytool command and use the generated certificate to execute *SecureEchoServer* and *SecureEchoClient* code. Follow below steps to generate SSL certificate.

1. Open a command prompt or terminal
2. Run this command  
`keytool -genkey -keyalg RSA -alias localhost -keystore selfsigned.jks -validity <days> -keysize 2048`  
 Where <days> indicate the number of days for which the certificate will be valid.

3. Enter a password for the keystore. Note this password as you require this for configuring the server
4. Enter the other required details
5. When prompted with "Enter key" password for <localhost>, press Enter to use the same password as the keystore password
6. Run this command to verify the contents of the keystore  
keytool -list -v -keystore selfsigned.jks

Put this generated selfsigned.jks file into the root directory of your source code and execute the below command to run SecureEchoServer and SecureEchoClient.

```
java -Djavax.net.ssl.keyStore=selfsigned.jks -
Djavax.net.ssl.keyStorePassword=localhost com.ignou.SecureEchoServer
```

```
java -Djavax.net.ssl.trustStore=selfsigned.jks -
Djavax.net.ssl.trustStorePassword=localhost com.ignou.SecureEchoClient
```

**Note:** Use the same password you used to create a self-signed certificate in step 3 for keyStorePassword and trustStorePassword. I used localhost as a password in self-signed certificate generation; therefore so in the above command, I used the same.

On the client terminal, send some text to the server, and the server will echo the same text on its terminal.

### Check Your Progress 2

- 1) Explain JSSE with its features and benefit

.....

.....

.....

.....

- 2) List down the benefits of implementing SSL using JSSE over secure connection established using java.net.URL

.....

.....

.....

.....

- 3) List down the reasons for javax.net.ssl.SSLHandshakeException during the establishment of a secure connection

.....

.....

.....

.....

---

## 12.3 ISSUES AND CHALLENGES OF WEB SECURITY

---

With the advancement of the Web and other technology, the frequent usage of networks makes web applications vulnerable to a variety of threats. Advancements in web applications have also attracted malicious hackers and scammers, who are always coming up with new attack vectors. Hackers and attackers can potentially take various

path through the web application to cause risks to your business. Web application security deals specifically with the security surrounding websites, web applications and web services such as APIs.

## Common Web App Security Vulnerabilities

Web application security vulnerabilities can range from targeted database manipulation to large-scale network disruption. Few methods of attack which commonly exploit are:

... **Cross site scripting (XSS)** – Cross site scripting (XSS) is a kind of client-side code injection attack. Attacker executes malicious script into web browser of the victim by including malicious code into benign and trusted websites. Attacker mostly uses the forums, message boards and web pages that allow comments to perform XSS. The use of user's input without validation in the output generated by web page make the web page vulnerable to XSS. XSS attacks are possible in VBScript, ActiveX, Flash, and even CSS. XSS is most common in JavaScript, JavaScript has very restricted access to user's operation system and file system since most web browsers run JavaScript with restrictions. Malicious JavaScript can access all the objects to which a web page has access to. Web page access also includes user's cookies. Most often, session tokens are stored into cookies. Thus, if attackers obtain a user's session cookie, they can impersonate that user, perform actions on behalf of the user, and gain access to the user's sensitive data.

... **Cross-site request forgery (CSRF)** – Cross site request forgery, also known as CSRF or XSRF, is a type of attack in which attackers trick a victim into making a request that utilizes their authentication or authorization. The victim's level of permission decides the impact of CSRF attack. Execution of CSRF attack consists of mainly two-part.

- Trick the victim into clicking a link or loading a web page
- Sending a crafted, legitimate-looking request from victim's browser to the website

Tricking the victim into clicking a link or loading a web page is done through social engineering or using the malicious link. While sending a crafted request, attackers send the value chosen by themselves and include the cookies, if any, which is associated with the origin of the website. A CSRF attack exploits the fact that the browser sends the cookies to the website automatically with each request. CSRF attacks take place only in the case of an authenticated user. This means that the victim must be logged in for the attack to succeed.

... **SQL injection (SQLi)** – SQL Injections are among the most common threat to data security. SQL injection, also known as SQLi, is a type of injection attack that exploits the weakness of application security and allows attackers to execute malicious SQL statements. Malicious SQL statements execution enable attackers to access or delete records, change an application data-driven behavior. Attackers can use SQL Injection vulnerabilities to bypass application security measures. Any website or web application, which uses a relational database such as MySQL, Oracle, SQL Server etc., can be impacted by SQL Injection. Attackers use SQLi to gain access to unauthorized information, modify or create new user permissions, or otherwise manipulate or destroy sensitive data. Attackers use specially-crafted input to trick an application into modifying the SQL queries that the application asks the database to execute. Let us understand it by example.

Suppose that a developer has developed a web page to show the account number and balance into account for the current user's ID provided in a URL, Code snippet in java might be as below:

```
String accountBalanceQuery =
    "SELECT accountNumber, balance FROM accounts WHERE account_owner_id = "
    + request.getParameter("user_id");
try
{
    Statement statement = connection.createStatement();
    ResultSets = statement.executeQuery(accountBalanceQuery);
    while (rs.next())
    {
        page.addTableRow(rs.getInt("accountNumber"), rs.getFloat("balance"));
    }
}
catch (SQLException e) { ... }
```

For normal operation user visit the URL as below:

[https://securebanking/show\\_balances?user\\_id=1234](https://securebanking/show_balances?user_id=1234)

From the java code snippet, we can check that SQL query to find the account balance for user 1234 will be as below:

```
SELECT accountNumber, balance FROM accounts WHERE
account_owner_id = 1234
```

The above query is a valid query, and it will return only the intended result to show the balance for the user id 1234.

Now suppose, an attacker has identified the application security vulnerability and change the parameter user\_id as-

0 OR 1 = 1

Now the query will be as below:

```
SELECT accountNumber, balance FROM accounts WHERE account_owner_id = 0
OR 1=1
```

When the application asks this query to execute into the database, it will return the account balance for all the accounts into the database. The attacker now knows every user's account numbers and balances.

... **Denial-of-service (DoS)** – Denial-of-service attack is a type of attack which makes information systems, devices or other network resources non-responding. A denial-of-service condition is accomplished by flooding the targeted host or network with traffic until the target cannot respond or simply crashes, preventing access for legitimate users. DoS attack exploits a vulnerability in a program or website to force improper use of its resources or network connections, which also leads to a denial of service. There are two general methods of DoS attack: flooding services or crashing services. Flood attacks occur when the system receives too much traffic for the server to buffer, causing them to slow down and eventually stop.

... **Insecure Direct Object References (IDOR)** - Insecure direct object references (IDOR) are a cybersecurity issue that occurs when a web application exposes a reference to internal implementation. Internal implementation objects include files, database records, directories and database keys. For example, an IDOR vulnerability could happen if the URL of a transaction could be changed through client-side user input to show unauthorized data of another transaction. For example, let's consider that the web application displays transaction details using the following URL:

<https://www.example.com/transaction?id=7777>

A malicious attacker could try to substitute the id parameter value 7778 with other similar values, for example:

<https://www.example.com/transaction?id=7778>

transaction Id 7778 could be a valid transaction but belonging to a different user. The malicious attacker should not be authorized to see the transaction for id 7778. However, if the developer made an error, the attacker would see this transaction, and hence web app would have an insecure direct object reference vulnerability.

---

## 12.4 SPRING SECURITY OVERVIEW

---

Security is a very crucial aspect for any application to avoid malfunctioning, unauthorized access or hacking of crucial data. Spring security modules provide us with a framework to easily develop a secure application with very few configurations. Spring Security is a very robust and highly customizable authentication and authorization access-control framework. Along with Authentication and Authorization in Java applications, it also protects the application against attacks such as cross-site request forgery, session fixation, clickjacking etc. In short, Spring Security refers to a series of processes that intercept requests and delegate security processing to the designated Spring Security process. Let us understand the three important concepts before getting dive into Spring Security.

### Authentication

Authentication refers to the process of confirming user's identity whom he claims to be. It's about validating user's credentials such as Email/Username/UserId and password. The system validates whether the user is one whom he claims to be using user's credentials. The system authenticates the user identity through login ID and password. Authentication using login Id and password is the usual way to authenticate; still, there are other ways to authenticate. Various methods for authentication are as follows:

- ... Login Form
- ... HTTP authentication
- ... HTTP Digest
- ... X.509 Certificates
- ... Custom Authentication Methods

### Authorization

Authorization action takes place once a system has authenticated the user's identity. Authorization refers to the rules that determine who is allowed to do what. E.g. John may be authorized to create and delete databases, while Bob is only authorized to read. Authorization is a process of permitting required privileges to the user to access a specific resource in an application such as files, location, database etc. In simple terms, authorization evaluates a user's ability to access the system and up to what extent. There may be different approaches to implement authorization into an application. Different approaches to authorization include:

- ... **Token-based:** Users are granted a cryptographically signed token that specifies what privileges the user is granted and what resources they can access.
- ... **Role-Based Access Control (RBAC):** Users are assigned role/roles, and these roles specify what privileges users have. Users roles would restrict what resources they have access to.

- ... **Access Control List (ACL):** An ACL specifies which users have access to a particular resource. For instance, if a user wants to access a specific file or folder, their username or details should be mentioned in the ACL in order to be able to access certain data.

Various methods for authorization are as follows:

- ... Access controls for URLs
- ... Secure Objects and methods
- ... Access Control List (ACL)

### Servlet Filters

A Servlet filter is an object that can intercept HTTP requests. Filter object is invoked at preprocessing and postprocessing of a request. Filters are mainly used to perform cross-cutting concerns such as conversion, logging, compression, encryption, decryption, input validation, security implementation etc.

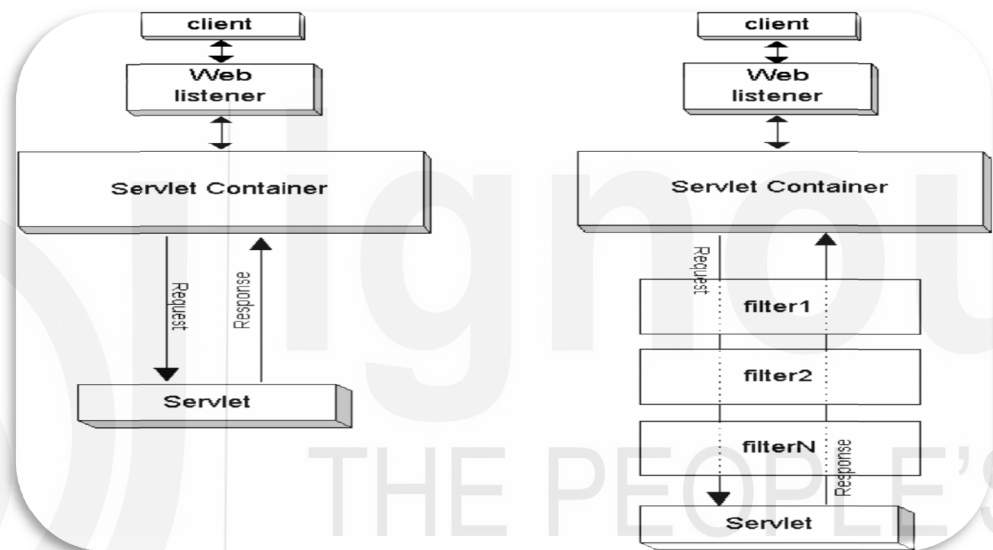


Figure 12.1: Servlet Filters

On the left of above image (Figure 12.1), we can see that there is no preprocessing and postprocessing of request, and thus request is directly reaching to servlet. If we want to implement the security, before request being processed by the servlet, DispatcherServlet in the context of spring, can be implemented using filters as shown above on the right side of the image.

In the context of Spring Application, no security implementation in DispatcherServlet and no one will like to implement security in all the controllers. Ideally, authentication and authorization should be done before a request hits the controller. Spring security uses various filters to implement security in Spring Applications.

### Spring 4 Security

The Spring Security stack is shown in the diagram(Figure 12.2)



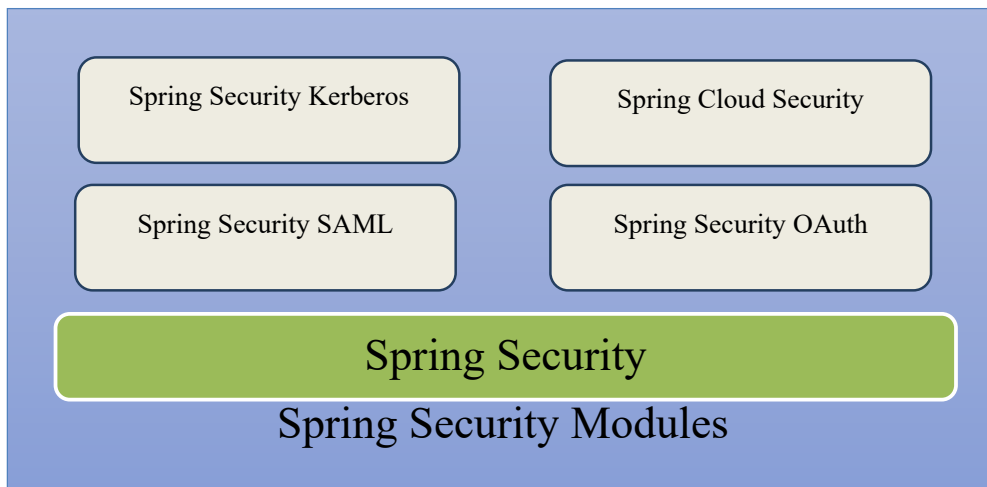


Figure 12.2: Spring Security Stack

Spring 4 Framework has the following modules to provide Security to the Spring-Based Applications:

- ... Spring Security
- ... Spring Security SAML
- ... Spring Security OAuth
- ... Spring Security Kerberos
- ... Spring Cloud Security

In Spring Framework, “Spring Security” module is the base module for the rest of the Spring Security modules. The following sections will outline some basics of “**Spring Security**” module.

Spring Security is one of the Spring Framework’s security modules. It is a Java SE/Java EE Security Framework to provide Authentication, Authorization, SSO and other Security features for Web Applications or Enterprise Applications.

### Spring 4 Security Features

*Spring Security Official website:* <https://projects.spring.io/spring-security/>

*Spring Security Documentation website:* <https://docs.spring.io/spring-security/site/docs/>

Spring 3.x Security Framework provides the following Features as per Spring Security Official website:

- ... Authentication and Authorization.
- ... Supports BASIC, Digest and Form-Based Authentication.
- ... Supports Cross-Site Request Forgery (CSRF) Implementation.
- ... Supports “Remember-Me” Feature through HTTP Cookies.
- ... Supports SSO (Single Sign-On) Implementation.
- ... Supports LDAP Authentication.
- ... Supports OpenID Authentication.
- ... Supports Implementation of ACLs
- ... Supports “Channel Security” that means automatically switching between HTTP and HTTPS.
- ... Supports I18N (Internationalization).
- ... Supports WS-Security using Spring Web Services.

- ... Supports Both XML Configuration and Annotations. Very Less or minimal XML Configuration.

Spring 4.x Security Framework supports the following new features:

- ... Supports WebSocket Security.
- ... Supports Spring Data Integration.
- ... CSRF Token Argument Resolver.

### Spring 4 Security Levels

Spring Security supports the following two Levels of Authorization-

- ... Method Level Authorization
- ... URL Level Authorization

#### NOTE

Spring Security supports “Method Level Security” by using AOP (Aspect-Oriented Programming) that means through Aspects. Spring Security supports “URL Level Security” by using Servlet filters.

### Spring 4 Security Advantages

Spring 4 Security Framework provides the following Advantages:

- ... Open Source Security Framework
- ... Comprehensive support for authentication and authorization
- ... Flexible, Easy to Develop and Unit Test the applications
- ... Protection against common tasks
- ... Declarative Security Programming
- ... Easy of Extendibility
- ... Ease of Maintenance
- ... CSRF protection
- ... We can develop Loosely-Coupled Applications.

### ☛ Check Your Progress 3

- 1) What is Spring Security? Explain in detail with its feature and benefit.

.....  
.....  
.....  
.....

- 2) What is Authentication (AuthN)? Write down various methods for AuthN.

.....  
.....  
.....  
.....

- 3) How does Authorization (AuthZ) make the system secured and restricted? List down various methods.

.....  
.....  
.....  
.....

## 12.5 JAVA BASED CONFIGURATION

Spring security uses Servlet Filter to implement the security in java applications. It creates a Servlet Filter named as *SpringSecurityFilterChain*. *SpringSecurityFilterChain* takes care of all the security concerns such as validating submitted username and passwords, application URLs protection, redirecting to the log in form to unauthenticated users, etc. Spring Security provides an abstract class *AbstractSecurityWebApplicationInitializer* that will ensure the *springSecurityFilterChain* Filter get registered for every URL in the application.

*SpringSecurityFilterChain* can be registered into spring application with the initialization of *AbstractSecurityWebApplicationInitializer* as:

```
import org.springframework.security.web.context.*;

public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer
{
}
}
```

With the latest Spring Security and/or Spring Boot versions, Spring Security is configured by having a class that:

- ... Is annotated with `@EnableWebSecurity`.
- ... Extends `WebSecurityConfigurerAdapter`, which basically provides a convenient base class for creating a `WebSecurityConfigurer` instance. The implementation allows customization by overriding methods. Here's what a typical `WebSecurityConfigurerAdapter` looks like:

```
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter { // (1)

    @Override
    protected void configure(HttpSecurity http) throws Exception { // (2)
        http
            .authorizeRequests()
            .antMatchers("/home").permitAll() // (3)
            .anyRequest().authenticated() // (4)
            .and()
            .formLogin() // (5)
            .loginPage("/login") // (5)
            .permitAll()
            .and()
            .logout() // (6)
            .permitAll()
            .and()
            .httpBasic(); // (7)
    }
}
```

1. A POJO class extending *WebSecurityConfigurerAdapter* and normal Spring `@Configuration` with the `@EnableWebSecurity` annotation
2. Overriding of `WebSecurityConfigurerAdapter`'s *configure(HttpSecurity)* method enables you to configure the FilterChain using DSL.
3. `permitAll()` allows all the users for the configured URI. User does not have to be authenticated. `antMatcher` allows to use wildcards such as `(*, \*\*, ?)` in the string. The request `"/home"` is allowed to all users without authentication.

4. All other requests except “/home”, the user to be authenticated first, i.e. the user needs to login.
5. As described, authentication methods in section 12.4, form login with a custom loginPage (/login, i.e. not Spring Security’s auto-generated one) is being used in the configuration. Login page should be accessible to anyone. Thus “/login” URI has been permitted to all using permitAll().
6. The default spring logout feature is configured and has been permitted to all.
7. Apart from form login, the configuration is also allowing for Basic Auth, i.e. sending in an HTTP Basic Auth Header to authenticate.

The above configuration creates SpringSecurityFilterChain which is accountable for security concerns such as validating submitted username and passwords, application URLs protection, redirecting to the log in form to unauthenticated users, etc. Java Configuration do the following for our application.

- ... Register required authentication for every URL
- ... Creates a login form
- ... Allow a user to be authenticated using form-based authentication as well as HTTP Basic Auth.
- ... Allow to logout
- ... Prevent from CSRF attack

## Securing the URLs

The most common methods to secure the URL are as below:

- ... authenticated(): This protects the URL which requires user to be authenticated. If the user is not logged in, the request will be redirected to the login page.
- ... permitAll(): This is applied for the URL’s which requires no security, such as css, javascript, login URL etc.
- ... hasRole(String role): This is used for authorization with respect to user’s role. It restricts to single role. “ROLE\_” will be appended into a given role. So if role value is as ”ADMIN”, comparison will be against “ROLE\_ADMIN”. An alternative is hasAuthority(String authority)
- ... hasAuthority(String Authority): This is similar to hasRole(String Role). No prefix will be appended in given authority. If authority is as “ADMIN”, comparison will be against “ADMIN”
- ... hasAnyRole(String... roles): This allows multiple roles. An alternative is hasAnyAuthority(String... authorities)

## HTTP Security

To enable HTTP Security in Spring, we need to extend the *WebSecurityConfigurerAdapter*. It allows to configure web-based security for specific http requests. By default all request will be secured but it can be restricted using requestMatcher(RequestMatcher) or other similar methods. The default configuration in the configure(HttpSecurity http) method is as:

```
protected void configure(HttpSecurity http) throws Exception
{
    http.authorizeRequests()
        .anyRequest().authenticated()
        .and().httpBasic();
}
```

## Form Login

Based on the features that are enabled in spring security, it generates a login page automatically. It uses standard values for the URLs such as /login and /logout to process the submitted login form and logout the user.

```
protected void configure(HttpSecurity http) throws Exception
{
    http.authorizeRequests()
        .anyRequest().authenticated()
        .and().formLogin()
        .loginPage("/login").permitAll();
}
```

## Authentication using Spring Security

Authentication is all about validating user credentials such as Username/User ID and password to verify the user's identity. We can either use in-memory-authentication or jdbc-authentication.

### In-Memory Authentication

In-Memory authentication can be configured using AuthenticationManagerBuilder as below:

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception
{
    auth.inMemoryAuthentication()
        .withUser("user").password(passwordEncoder().encode("password")).roles("USER")
        .and()
        .withUser("admin").password(passwordEncoder().encode("password")).roles("USER",
            "ADMIN");
}
```

### JDBC Authentication

To implement JDBC authentication, all you have to do is to define a data source within the application and use it as below:

```
@Autowired
private DataSource dataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception
{
    auth.jdbcAuthentication().dataSource(dataSource)
        .withDefaultSchema()
        .withUser("user").password(passwordEncoder().encode("password")).roles("USER")
        .and()
        .withUser("admin").password(passwordEncoder().encode("password")).roles("USER",
            "ADMIN");
}
```

As we can see, we're using the autoconfigured *DataSource*.

The *withDefaultSchema* directive adds a database script that will populate the default schema, allowing users and authorities to be stored. DDL statements are given for the HSQLDB database. The default schema for users and authorities is as below:

```
create table users
```

```
(
    username varchar_ignorecase(50) not null primary key,
    password varchar_ignorecase(50) not null,
    enabled boolean not null
);
create table authorities
(
    username varchar_ignorecase(50) not null,
    authority varchar_ignorecase(50) not null,
    constraint fk_authorities_users foreign key(username) references users(username)
);
```

Default Schema will not work for other databases except HSQLDB database. Check the spring.io website for schema corresponding to a database. The database schema can be checked at:

<https://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/appendix-schema.html>

### Customizing the search Queries

If Spring application is not using default schema for security, we need to provide custom queries to find user and authorities details as below:

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception
{
    auth.jdbcAuthentication()
        .dataSource(dataSource)
        .usersByUsernameQuery("select uuid,password,enabled "
            + "from users "
            + "where uuid= ?")
        .authoritiesByUsernameQuery("select uuid,authority "
            + "from authorities "
            + "where uuid= ?");
}
```

Customization of a search query in spring security is very easy, and two methods for that has been provided as *usersByUsernameQuery* and *authoritiesByUsernameQuery*.

---

## 12.6 CREATE SPRING INITIALIZER CLASS

---

AbstractAnnotationConfigDispatcherServletInitializer class has simplified the Spring initialization. We don't need to manually create contexts but just set appropriate config classes in `getRootConfigClasses()` and `getServletConfigClasses()` methods.

Let us consider that we're developing a web application, and we're going to use Spring MVC, Spring Security and Spring Data JPA. For this simple scenario, we would have at least three different config files. A `WebConfig` contains all our web-related configurations, such as `ViewResolvers`, `Controllers`, `ArgumentResolvers` etc. `RepositoryConfig` may define `datasource`, `EntityManagerFactory`, `PlatformTransactionManager` etc.

For gluing all these together, we have two options. First, we can define a typical hierarchical `ApplicationContext`, by adding `RepositoryConfig` and `SecurityConfig` in root context and `WebConfig` in their child context:

```

public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer
{
    @Override
    protected Class<?>[] getRootConfigClasses()
    {
        return new Class<?>[] { RepositoryConfig.class, SecurityConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses()
    {
        return new Class<?>[] { WebConfig.class };
    }
    @Override
    protected String[] getServletMappings()
    {
        return new String[] { "/" };
    }
}

```

If we have a single DispatcherServlet, we can add the WebConfig to the root context and make the servlet context empty:

```

public class ServletInitializer extends AbstractAnnotationConfigDispatcherServletInitializer
{
    @Override
    protected Class<?>[] getRootConfigClasses()
    {
        return new Class<?>[] { RepositoryConfig.class, SecurityConfig.class,
            WebConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses()
    {
        return null;
    }
    @Override
    protected String[] getServletMappings()
    {
        return new String[] { "/" };
    }
}

```

---

## 12.7 CREATE CONTROLLER AND VIEW

---

### Controllers and Mapping

Spring **@Controller** annotation is used to mark a class as web request handler in the Spring MVC application. While **@RestController** is a convenience annotation to mark a class as request handler for RESTful web services, **@RestController** is annotated with **@Controller** and **@ResponseBody**.

A simple example of **@Controller** annotated Class responsible to handle a Get Request method.

```

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

```

```

@Controller //(1)
@RequestMapping(value="/") //(2)
public class HomeController
{
    @GetMapping //(3)
    public ModelAndView index()
    {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");
        mv.getModel().put("message", "User, Spring Security Implemented Successfully !!");
        return mv;
    }
}

```

1. `@Controller` annotation is used to mark the class responsible to handle the web request forwarded from Dispatcher Servlet.
2. `@RequestMapping` annotation has been used to mark at class level to make class `HomeController` responsible to all web request for the URL pattern `“/”`
3. `@GetMapping` annotation has been used to mark the method `index()` to be invoked for Method Get and URL pattern `“/”`

### Views and Configuration

To render the view, *ViewResolver* configuration is required in Spring MVC Application or Spring Boot Application. For JSP, *ViewResolver* configuration is required in Spring MVC/Spring Boot application to resolve the location of jsp files. There are two ways to configure view resolver as-

... Add entries as shown below in application.properties  
spring.mvc.view.prefix=/WEB-INF/views/  
spring.mvc.view.suffix=.jsp

OR

... Configure *InternalResourceViewResolver* by extending *WebMvcConfigurerAdapter*

```

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

```

```

@Configuration
@EnableWebMvc
@ComponentScan
public class MvcConfiguration extends WebMvcConfigurerAdapter
{
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry)
    {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        resolver.setViewClass(JstlView.class);
        registry.viewResolver(resolver);
    }
}

```

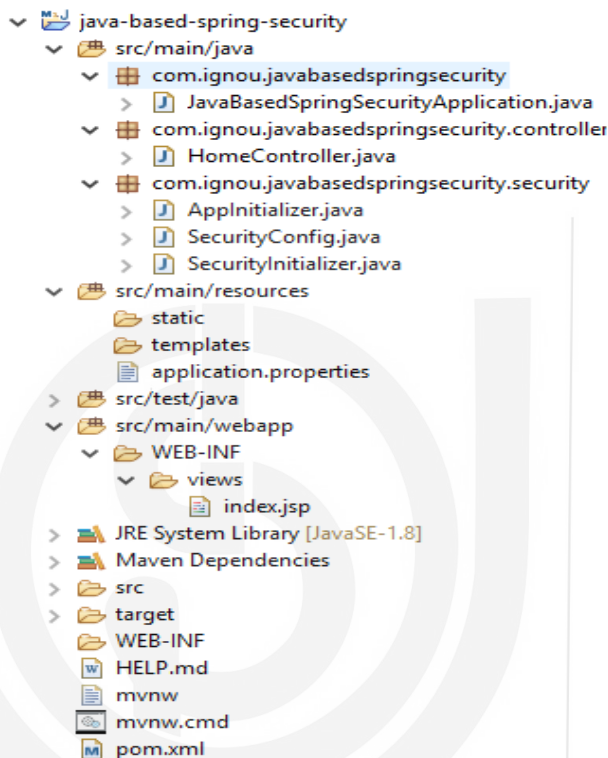


## 12.8 RUN APPLICATION

This section will explain the steps required to run the Spring Boot application with Spring Security. Tools and software required to write the code easily, manage the spring dependencies and execute the application is as -

- ... Java 8 or above is required
- ... Maven
- ... Eclipse IDE

**Step 1:** Create a web maven project and add the required dependencies such as



spring-boot-starter-web, spring-boot-starter-security in pom.xml. Directory Structure for Spring Project in eclipse will be as below –

**Step 2:** Spring Boot Application Initializer

```
package com.ignou.javabasedspringsecurity;
```

```
@SpringBootApplication
```

```
public class JavaBasedSpringSecurityApplication extends SpringBootServletInitializer
```

```
{
```

```
    @Override
```

```
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
```

```
    {
        return application.sources(JavaBasedSpringSecurityApplication.class);
    }
}
```

```
public static void main(String[] args)
```

```
{
```

Figure 2: Project Folder Structure In Eclipse

```
    SpringApplication.run(JavaBasedSpringSecurityApplication.class, args);
```

```
}
```

```
}
```

**Step 3: Spring Security Configuration**

```
package com.ignou.javabasedspringsecurity.security;
```

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter
{
    @Autowired
    PasswordEncoder passwordEncoder;
    @Bean
    public PasswordEncoder passwordEncoder()
    {
        return new BCryptPasswordEncoder();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
    {
        auth.inMemoryAuthentication().passwordEncoder(passwordEncoder).withUser("testuser")
        .password(passwordEncoder.encode("user@123")).roles("USER").and().withUser("testadmin")
        .password(passwordEncoder.encode("admin@123")).roles("USER", "ADMIN");
    }
    protected void configure(HttpSecurity http) throws Exception
    {
        http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();
    }
}
```

**Step 4: Initialize Spring Security**

```
package com.ignou.javabasedspringsecurity.security;
```

```
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer
{
    // No Code Needed Here
}
```

**Step 5: Include the spring security into App\_INITIALIZER**

```
package com.ignou.javabasedspringsecurity.security;
```

```
public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer
{
    @Override
    protected Class<?>[] getRootConfigClasses()
    {
        return new Class[] { SecurityConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses()
    {
        return null;
    }
    @Override
    protected String[] getServletMappings()
    {
        return new String[] { "/" };
    }
}
```

**Step 6: Controller Class and View**

```

package com.ignou.javabasedspringsecurity.controller;
@Controller
@RequestMapping(value="/")
public class HomeController
{
    @GetMapping
    public ModelAndView index()
    {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");
        mv.getModel().put("message", "User, Spring Security Implemented Successfully !!");
        return mv;
    }
}

```

For the view we will use a simple jsp as below

```

<!DOCTYPE html>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html lang="en">
<body>
<div>
<div>
<h1>Spring Boot Security Example</h1>
<h2>Hello ${message}</h2>
</div>
</div>
</body>
</html>

```

**Step 7: Spring Boot JspViewResolver configuration**

Add the below properties into application.properties file

```

spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp

```

**Step 8:** Run the file created into Step 2 named as `JavaBasedSpringSecurityApplication.java` to start the application. This file can be executed using java command or from eclipse IDE itself. After successful execution of the application, security configured application can be accessed using browser on `localhost:8080`

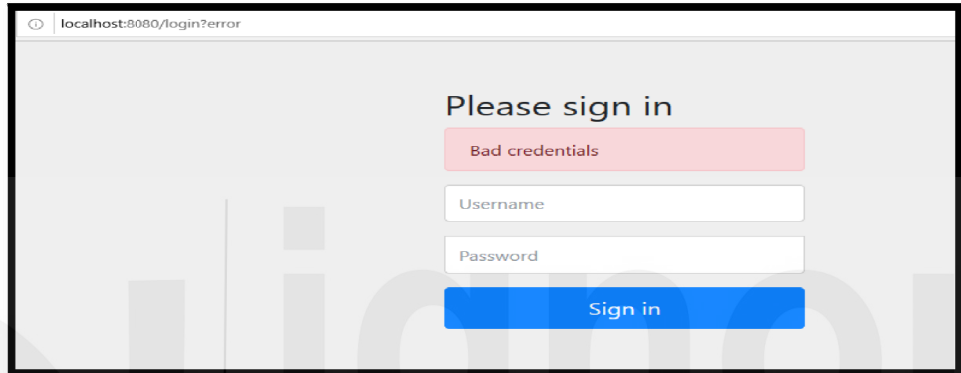
In Step 3 we configured the spring security and defined the *testuser* with credentials as *user@123* and *testadmin* with credentials as *admin@123*.

**Execution Output:**

If user tries to access `localhost:8080` and user is not logged in, spring will redirect to user on `localhost:8080/login` as shown below. This login page is spring security in-built.



If user provides **wrong credentials**, alert from spring security will be as below:



After **Successful login** user will get the below screen:



#### 🔗 Check Your Progress 4

1) What is SecurityFilterChain in Spring Security?

.....

.....

.....

.....

2) Write the Spring Security Configuration class code for below mentioned criteria-

|                                   |                                    |
|-----------------------------------|------------------------------------|
| http://www.securesite.com/static  | Open to everyone – CSS, JavaScript |
| http://www. securesite.com/login  | Open to everyone                   |
| http://www. securesite.com/user/  | ROLE_USER or ROLE_ADMIN            |
| http://www. securesite.com/admin/ | ROLE_ADMIN only                    |

- 3) Consider that Spring Authentication with default schema attribute **username** is not suitable for the application. Email attribute as the primary key is being used instead of username. Write the Spring Security Configuration class code to adapt the queries for the new schema as below:

```
CREATE TABLE users
(
    name VARCHAR(50) NOT NULL,
    email VARCHAR(50) NOT NULL,
    password VARCHAR(100) NOT NULL,
    enabled TINYINT NOT NULL DEFAULT 1,
    PRIMARY KEY (email)
);

CREATE TABLE authorities
(
    email VARCHAR(50) NOT NULL,
    authority VARCHAR(50) NOT NULL,
    FOREIGN KEY (email) REFERENCES users(email)
);
```

- 4) How to secure the URLs in Spring Security? Explain with available methods to make URL secure.

## 12.9 SUMMARY

In this unit, many concepts related to web security have been explained. You got to know about securing data at-rest and in-transit. In the unit JCA section described many useful concepts such as Cipher, Message Digest, Message Authentication Code, Signature. Using JCA API, the authenticity and integrity of the message can be verified. Further SSL and other useful concepts to secure the data in-transit is explained. The Spring Security provides a convenient and easy way to incorporate security in spring applications with minimum configuration. The concept of Authentication and Authorization has been explained to make application secure. Various methods such as `authenticate()`, `permitAll()`, `hasRole()`, `hasAuthority()` have been written to make URL secure in Spring Security. In-memory authentication and JDBC authentication has been described with the custom search query to incorporate user-defined schema for users and authorities. In the last section of this unit, you learned code to integrate spring security into the spring boot application.

## 12.10 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

### ☛ Check Your Progress 1

- 1) JCA has been described with the core classes and interface in section 12.2.1
- 2) Comparison of Message Digest and Message Authentication Code

| Message Digest  | Message Authentication Code  |
|---|--|
| A message digest algorithm takes a single input a message and produces a "message digest" (aka hash)                            | A Message Authentication Code algorithm takes two inputs, a message and a secret key, and produces a MAC                                     |
| It allows you to verify the integrity of the message  | It allows you to verify the integrity and the authenticity of the message  |
| Any change to the message will (ideally) result in a different hash being generated.  | Any change to the message <b>or</b> the secret key will (ideally) result in a different MAC being generated.                                 |
| An attacker that can replace the message and digest is fully capable of replacing the message and digest with a new valid pair. | Nobody without access to the secret should be able to generate a MAC calculation that verifies the integrity and authenticity of the message |

- 3) Man-in-the-middle attack is possible. Chuck, a man in the middle, attacks and intercepts all the messages sent by Alice. Chuck performs below-
  - Instead of forwarding message m, he forwards message n. Instead of the signature on m with Alice's private key, he forwards a signature on n with his private key.
  - Instead of Alice's public key he sends his public key to Bob. Bob will never see the difference as he does not know Alice's public key beforehand.

A flaw in the approach is that the public key should not be transferred along with the message. Public key should be available to the person prior to verification of the signature.

### ☛ Check Your Progress 2

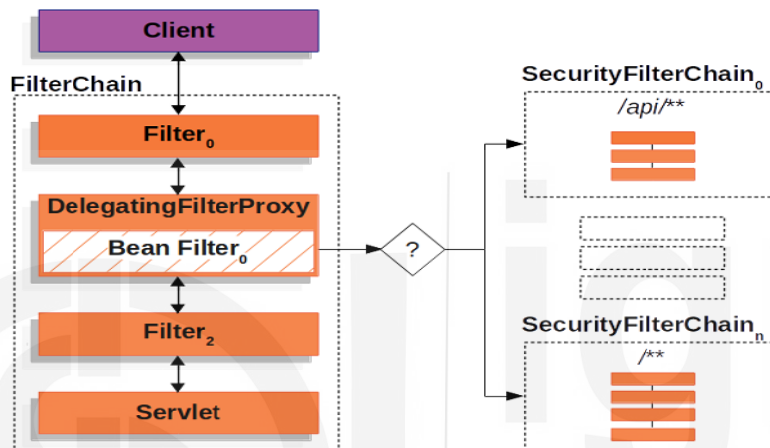
- 1) JSSE with the advantages has been written in section 12.2.2
- 2) `java.net.URL` class demonstrates the easiest way to add SSL to your applications. This approach is useful, but it is not flexible enough to let you create a secure application that uses generic sockets. Using JSSE API to implement SSL gives us full control over the protocols that need to be supported.
- 3) If `javax.net.ssl.SSLHandshakeException` occurs, it indicates that there is a `handshake_failure`. We haven't created or established the SSL certificate that the client and server should be referencing. Normally a certificate is obtained from a trusted certificate authority for a public site, but for testing and development purposes, it's common practice to create a self-signed certificate, which just means you are the issuer. These are typically issued to the localhost domain, so they'll function locally but not be trusted elsewhere.  
**Refer section 12.2.2 for keytool to generate keypair and how to use this to solve `javax.net.ssl.SSLHandshakeException` issue.**

### ☛ Check Your Progress 3

- 1) Refer section 12.4, Spring Security has been explained in detail.
- 2) Refer Authentication in Section 12.4
- 3) Refer Authorization in Section 12.4

### ☛ Check Your Progress 4

- 1) SecurityFilterChain is a FilterChain component which has zero or more Security Filters in an order.



Refer Section 12.5 for more details for SecurityFilterChain.

- 2) **Hint:** configure method definition in Spring Security Configuration will be as

```
@Override
protected void configure(HttpSecurity http) throws Exception
{
    http
        .authorizeRequests()
        .antMatchers("/static", "/register").permitAll()
        .antMatchers("/user/**").hasAnyRole("USER", "ADMIN") // can pass
        multiple roles
        .antMatchers("/admin/**").hasRole("ADMIN")
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .loginUrl("/login")
        .permitAll();
}
```

... /static and /register is allowed to everyone so permitAll() is used.

... /user/\*\* must be accessible to either “User” or “Admin” roles. Thus hasAnyRole("USER", "ADMIN") is used.

... /admin/\*\* must be accessible to the only Admin role. Thus hasRole("ADMIN") is used

... /login must be allowed for all so permitAll() is used.

- 3) We need to customize the search query. Spring security can adapt the customize search queries very easily. We simply have to provide our own

SQL statements when configuring the `AuthenticationManagerBuilder`. Two methods for customization has been provided as ***usersByUsernameQuery*** and ***authoritiesByUsernameQuery***. Refer to Section 12.5 for customizing the search query.

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
    throws Exception
{
    auth.jdbcAuthentication()
        .dataSource(dataSource)
        .usersByUsernameQuery("select email,password,enabled "
            + "from users "
            + "where email = ?")
        .authoritiesByUsernameQuery("select email,authority "
            + "from authorities "
            + "where email = ?");
}
```

4) Refer Securing the URLs in section 12.5

---

## 12.11 REFERENCES/FURTHER READING

---

- ... Craig Walls, “Spring Boot in action” Manning Publications, 2016.  
(<https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf>)
- ... Christian Bauer, Gavin King, and Gary Gregory, “Java Persistence with Hibernate”,Manning Publications, 2015.
- ... Ethan Marcotte, “Responsive Web Design”, Jeffrey Zeldman Publication, 2011([http://nadin.miem.edu.ru/images\\_2015/responsive-web-design-2nd-edition.pdf](http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf))
- ... Tomcy John, “Hands-On Spring Security 5 for Reactive Applications”,Packt Publishing,2018
- ... <https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html>
- ... <https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html>
- ... <https://docs.spring.io/spring-security/site/docs/3.1.x/reference/springsecurity.html>
- ... <https://spring.io/guides/topicals/spring-security-architecture>
- ... <http://tutorials.jenkov.com/java-cryptography/index.html>
- ... <https://www.cloudflare.com/learning/security/what-is-web-application-security/>
- ... <https://www.baeldung.com/java-ssl>
- ... <https://www.baeldung.com/java-security-overview>
- ... <https://www.baeldung.com/spring-security-login>
- ... <https://dzone.com/articles/spring-security-authentication>



---

# UNIT 13 CUSTOM LOGIN USING SECURITY

---

## Structure

- 13.0 Introduction
- 13.1 Objectives
- 13.2 Custom Login form creation
- 13.3 Spring Config for Custom Login Form
- 13.4 Create Request mapping and building Custom Login Form
- 13.5 Testing Custom Login Form
  - 13.5.1 Integration Test in Spring Boot
- 13.6 Adding Logout Support
- 13.7 Summary
- 13.8 Solutions/ Answer to Check Your Progress
- 13.9 References/Further Reading

---

## 13.0 INTRODUCTION

---

Unit 12 explained how to secure the spring application using spring security. The built-in login module of Spring Security authenticates the users and provides accessibility to the application. Default spring security does not require to create new jsp page for login since built-in login module already has default login page, which is used to authenticate the user. The default login page provided by spring security is not suitable for enterprise web application since the developer does not have control on designing and processing behavior. This unit describes how to customize the login page into spring security. The default login page, provided by Spring Security, has been used in the example written in unit 12, section 12.8. In this unit, the example executed in section 12.8 will be incorporated with a custom login page.

Testing is a very important phase of any application to ensure whether it works as per expected functionality or not. Integration testing and unit testing play a vital role in making the application error prone. This unit describes the annotation available in spring boot for Integration testing. The next unit will explain about unit test annotation available in spring boot.

Once user finishes all the activity on a web portal, he must be provided with a way to invalidate the session using logout functionality. Spring logout feature with various available methods has been explained in this unit.

---

## 13.1 OBJECTIVES

---

After going through this unit, you should be able to:

- ... create custom login page for spring security,
- ... configure the custom login page,
- ... execute the spring boot application with custom login page spring security, and
- ... develop logout support in Spring Applications.

## 13.2 CUSTOM LOGIN FORM CREATION

If no login form is configured into spring security, spring boot provides default login screen for spring security. Default login screen will be used to authenticate the user, and user will be allowed to access a resource. In an enterprise application default login screen is not suitable since enterprise does not have full control over design, processing URLs etc.

Customized login form creation and configuration is very easy in spring security. Custom login form in Spring Application is nothing but a simple html or jsp, like other web pages in java. A custom login form enables an enterprise to customize the login form as per their choice of design. It provides the developer to have full control over css and processing URL. A custom login form must be having below artifacts:

- ... Form action URL where the form is POSTed to trigger the authentication process
- ... Username input field for the username
- ... Password input field for the password

Sample custom login page is shown below. You can put the css styling as per application requirements. The following written custom login page will be used in the execution of the application into example.

### Form Action URL(Custom login Controller):

```
package com.ignou.javabasedspringsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping(value="/customlogin")

public class LoginController
{
    @GetMapping

    public String login()
    {
        return "customlogin";
    }
}
```

### Custom Login Form with CSS:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>
body {font-family: Arial, Helvetica, sans-serif;}
form {border: 3px solid #f1f1f1; width: 50%; margin: auto;}

input[type=text], input[type=password]
```

```
{
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #ccc;
  box-sizing: border-box;
}

button
{
  background-color: #4CAF50;
  color: white;
  padding: 14px 20px;
  margin: 8px 0;
  border: none;
  cursor: pointer;
  width: 100%;
}

button:hover
{
  opacity: 0.8;
}

.cancelbtn
{
  width: auto;
  padding: 10px 18px;
  background-color: #f44336;
}

.container
{
  padding: 16px;
  width: 50%;
  margin: auto;
}

.errormsg
{
  width: 100%;
  color: red;
  padding: 12px 0px;
  text-align: center;
}

span.psw
{
  float: right;
  padding-top: 16px;
}

</style>
</head>
<body>

<h2 style="text-align: center;">Spring Security Custom Login Form</h2>

<form action="/signin" method="post">
<div class="container">
  <c:if test="${param.error ne null}">
    <div class="errmsg"><b>Invalid
credentials</b></div>
  </c:if>
  <label for="uname"><b>Username</b></label>
  <input type="text" placeholder="Enter Username" name="username" required>

  <label for="psw"><b>Password</b></label>
  <input type="password" placeholder="Enter Password" name="password" required>

  <button type="submit">Login</button>
  <label>
  <input type="checkbox" checked="checked" name="remember"> Remember me
  </label>
</div>

<div class="container">
  <button type="button" class="cancelbtn">Cancel</button>
  <span class="psw">Forgot <a href="#">password?</a></span>

```

```

</div>
</form>

</body>
</html>

```

The login form created above has the following relevant artifacts:

- ... /signin – the URL where the form is POSTed to trigger the authentication process
- ... username – input field name for the username
- ... password – input field name for the password

Above jsp outputs the following login screen (Figure 13.1).

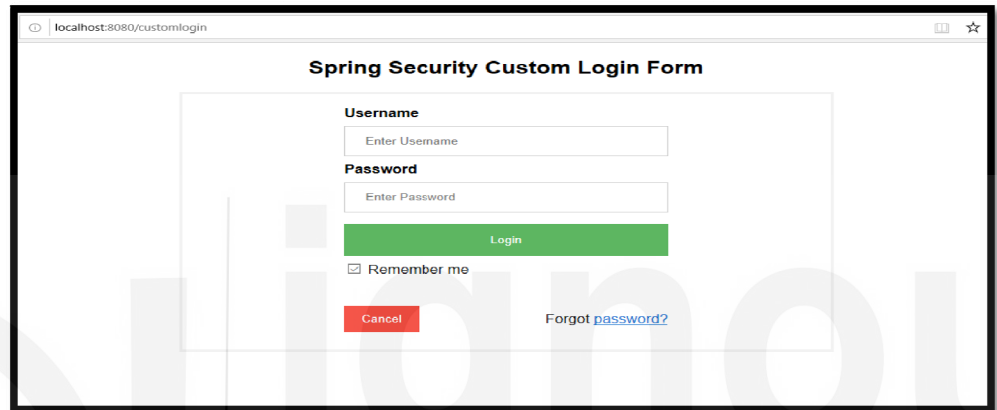


Figure13.1: Custom Login Page in Spring Security

### 13.3 SPRING Security CONFIG with Spring MVC FOR CUSTOM LOGIN FORM

Once you create the custom login page, you need to configure the custom login page in spring security. There is a detailed explanation regarding Java-based Spring Security configuration in section 12.5. Configuration for custom login page in spring security is very easy. You just need to set the loginPage in FormLoginConfigurer object provided by formLogin(). Sample Spring security configuration with custom login page is shown below.

```

package com.ignou.javabasedspringsecurity.security;
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Autowired
    PasswordEncoderpasswordEncoder;

    @Bean
    public PasswordEncoderpasswordEncoder ()
    {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
    {
        auth.inMemoryAuthentication().passwordEncoder(passwordEncoder)
            .withUser("testuser")
            .password(passwordEncoder.encode("user@123")).roles("USER").and()
            .withUser("testadmin").password(passwordEncoder.encode("admin@123"))
            .roles("USER", "ADMIN");
    }
}

```

```
protected void configure(HttpSecurity http) throws Exception
{
    http.csrf().disable()
    .authorizeRequests()
    .antMatchers("/customlogin").permitAll()
    .anyRequest().authenticated()
    .and()
    .formLogin()
    .loginPage("/customlogin")
    .loginProcessingUrl("/signin")
    .defaultSuccessUrl("/", true)
    .failureUrl("/customlogin?error=true");
}
}
```

In the above configuration, in-memory authentication has been configured with two users as testuser with Role USER and testadmin user with Role as USER, ADMIN both. The following section will explain the elements used to create the form login configuration.

### ***authorizeRequests()***

`<intercept-url>` element with `access="permitAll"` configures the authorization to **allow** all the requests on that particular path. Thus the requests for which no authorization is required, can be achieved without disabling the spring security using `permitAll()`.

### ***formLogin()***

There are several methods to configure the behavior of `formLogin()`.

- ... `loginPage()` – the custom login page
- ... `loginProcessingUrl()` – the url to submit the username and password to
- ... `defaultSuccessUrl()` – the landing page after a successful login
- ... `failureUrl()` – the landing page after an unsuccessful login

### **Check Your Progress 1**

- 1) Why is login form customization required in an application? Explain the customized login form configuration with sample code.

.....

.....

.....

.....

- 2) Define `loginPage()` and `loginProcessingUrl()` in Spring Boot Security configuration.

.....

.....

.....

.....

---

## **13.4 CREATE REQUEST MAPPING AND BUILDING CUSTOM LOGIN FORM**

---

Custom login form creation has been explained in Section 13.2. A very simple custom login form named as `customlogin.jsp` without any css is shown below

```

<html>
<body>
<div>
<center>Custom Spring Login Form</center>
<form action="/signin" method="post">
<input name="username" type="text" placeholder="Enter Username" />
<input name="password" type="text" placeholder="Enter password" />
<input type="submit" />
</form>
</div>
</body>
</html>

```

Custom login form must be mapped with controller in order to render the page to the user. The controller class is responsible for processing the request received from the dispatcher servlet. Spring provides annotation like `@Controller` to make a POJO class a controller in spring mvc application. `@RequestMapping` can be used at class level to map the controller which will be responsible for handling the request, and at the method level to map the method, which will be responsible for handling the requested URL. In previous code snippet for security configuration in section 13.3, `loginPage("/customlogin")` has been used. `"/customlogin"` should be mapped into a controller which will be responsible to process it and to return the corresponding view. Request mapping is shown below for `"/customlogin"` which return the view name as `customlogin`.

Spring view resolver resolves this view name as `customlogin.jsp` and it renders the jsp created into section 13.2

```

package com.ignou.javabasedspringsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping(value="/customlogin")
public class LoginController
{

    @GetMapping
    public String login()
    {
        return "customlogin";
    }

}

```

In above code snippet, `LoginController` is mapped with the URL pattern `/customlogin` and `login()` method is returning view name as `cutomlogin`. View resolver will resolve the view and corresponding view will be rendered.

---

## 13.5 TESTING CUSTOM LOGIN FORM

---

This section describes the execution of spring boot application with custom login form using spring security. Configuration code and request mapping code has been written

in code sample. This section also describes the annotations available in spring boot for unit testing.

Directory Structure for Spring Project in eclipse will be as below (Figure 13.2) –

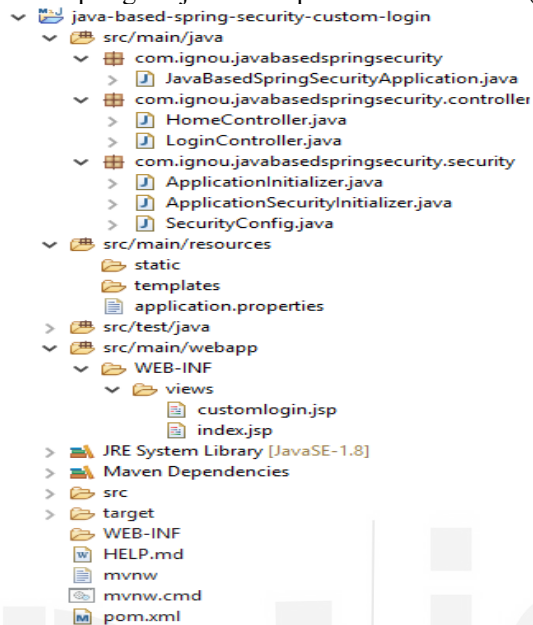


Figure 13.2: Project Structure In Eclipse

Spring Security Configuration for custom login form

```
package com.ignou.javabasedspringsecurity.security;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    PasswordEncoder passwordEncoder;
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication().passwordEncoder(passwordEncoder).withUser("testuser")
            .password(passwordEncoder.encode("user@123")).roles("USER").and().withUser("testadmin")
            .password(passwordEncoder.encode("admin@123")).roles("USER", "ADMIN");//(1)
    }
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/customlogin*").permitAll() //(2)
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/customlogin") //(3)
            .loginProcessingUrl("/signin") //(4)
            .defaultSuccessUrl("/", true)
            .failureUrl("/customlogin?error=true");
    }
}
```

In above code followings have been configured.

- 1) In-Memory authentication is configured for users testuser and testadmin.

- 2) /customlogin URL should be accessible to all. Thus permitAll() will allow all request to pass.
- 3) formLogin().loginPage("/customlogin") configures the customlogin page. Controller must have a mapping corresponding to Url pattern "/customlogin" to return customlogin page.
- 4) Custom login form processing URL is configured by loginProcessingUrl("/signin"). Custom login page form action must be "/signin"

### Controller Classes and View

```
package com.ignou.javabasedspringsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping(value="/customlogin")
public class LoginController
{

    @GetMapping
    public String login()
    {
        return "customlogin";
    }
}

package com.ignou.javabasedspringsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping(value="/")
public class HomeController
{
    @GetMapping
    public ModelAndView index()
    {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");
        mv.getModel().put("message", "User, Spring Security with custom login page implemented Successfully !!");
        return mv;
    }
}
```

### Custom login form with CSS

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>
body {font-family: Arial, Helvetica, sans-serif;}
form {border: 3px solid #f1f1f1; width: 50%; margin: auto;}

input[type=text], input[type=password]
{
    width: 100%;
    padding: 12px 20px;
    margin: 8px 0;
```



```

display: inline-block;
border: 1px solid #ccc;
box-sizing: border-box;
}

button
{
background-color: #4CAF50;
color: white;
padding: 14px 20px;
margin: 8px 0;
border: none;
cursor: pointer;
width: 100%;
}

button:hover
{
opacity: 0.8;
}

.cancelbtn
{
width: auto;
padding: 10px 18px;
background-color: #f44336;
}

.container
{
padding: 16px;
width: 50%;
margin: auto;
}

.errormsg
{
width: 100%;
color: red;
padding: 12px 0px;
text-align: center;
}

span.psw
{
float: right;
padding-top: 16px;
}

</style>
</head>
<body>

<h2 style="text-align: center;">Spring Security Custom Login Form</h2>

<form action="/signin" method="post">
<div class="container">
    <c:if test="${param.error ne null}">
        <div class="errormsg"><b>Invalid
credentials</b></div>
    </c:if>
    <label for="uname"><b>Username</b></label>
    <input type="text" placeholder="Enter Username" name="username" required>

    <label for="psw"><b>Password</b></label>
    <input type="password" placeholder="Enter Password" name="password" required>

    <button type="submit">Login</button>
    <label>
    <input type="checkbox" checked="checked" name="remember"> Remember me
    </label>
</div>

<div class="container">
<button type="button" class="cancelbtn">Cancel</button>
<span class="psw">Forgot <a href="#">password?</a></span>
</div>
</form>

</body>

```

&lt;/html&gt;

Execute the application as explained in Unit 12. In above example spring security has been configured with User as testuser with credential as user@123 and another User as testadmin with credential as admin@123.

Output:

If user tries to access localhost:8080 and user is not logged in, spring will redirect the user on localhost:8080/customlogin and custom login page will be rendered as shown below in Figure 13.2:

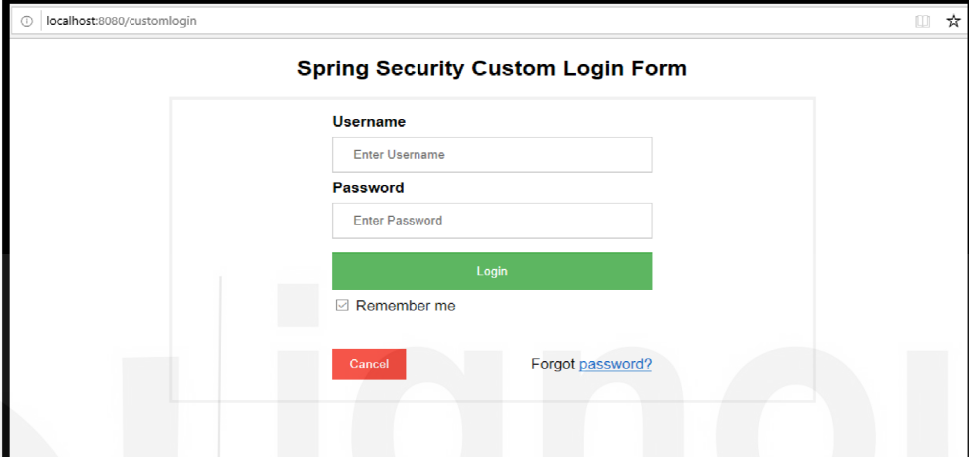
A screenshot of a web browser window showing the 'Spring Security Custom Login Form'. The browser's address bar displays 'localhost:8080/customlogin'. The form is centered on the page and contains the following elements: a title 'Spring Security Custom Login Form', a 'Username' label above a text input field with placeholder text 'Enter Username', a 'Password' label above a text input field with placeholder text 'Enter Password', a green 'Login' button, a checked 'Remember me' checkbox, a red 'Cancel' button, and a link 'Forgot password?'. The entire form is enclosed in a light gray border.

Figure 13.3: Custom Login Page In Spring Security

If user provides wrong credentials, alert from spring security will be as shown in Figure 13.4 :

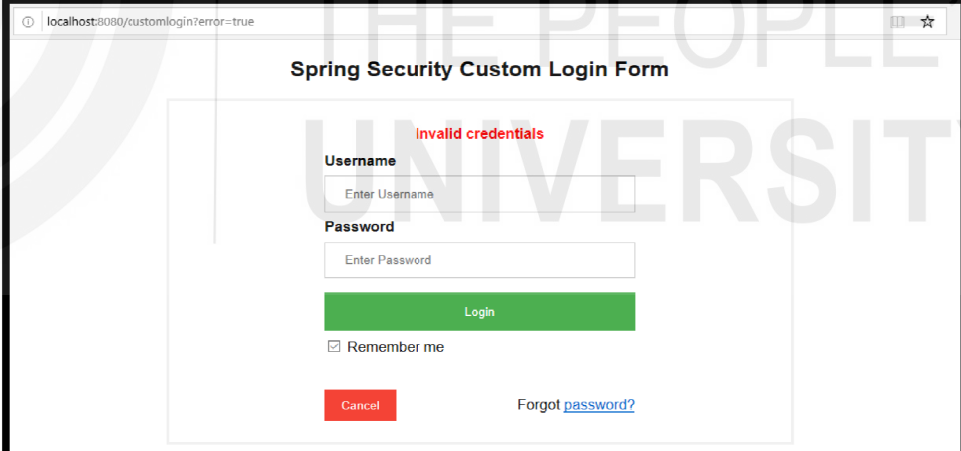
A screenshot of the same 'Spring Security Custom Login Form' as in Figure 13.3, but with an error message. The browser's address bar now shows 'localhost:8080/customlogin?error=true'. Above the 'Username' input field, the text 'Invalid credentials' is displayed in red. The rest of the form, including the 'Login' button, 'Remember me' checkbox, 'Cancel' button, and 'Forgot password?' link, remains the same.

Figure 13.4: Invalid Credentials Error Message

After Successful login user will be redirected to index.jsp and it will output as shown in Figure 13.5.

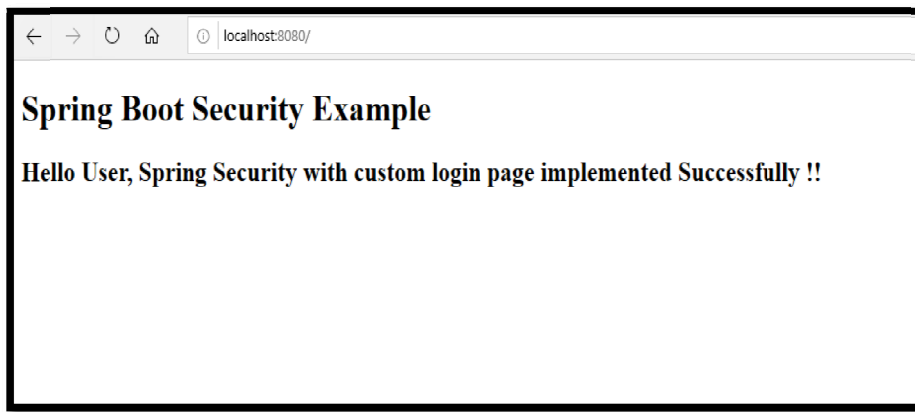


Figure 13.5: Success Screen After Authentication

### 13.5.1 Integration Test in Spring Boot

The following section describes the integration test in the spring boot application. Before getting dive into integration testing, let us define how integration test is different from unit test.

Unit testing aims at individual modules of an application without any interaction with dependencies to validate whether code is working as per the expected result or not. Integration testing aims to validate the code is working fine when different modules are combined as a group.

Previous sections described custom login form integration with spring security. Here integration test will be described for configured custom login form with spring security. Spring provides many annotations to perform integration test.

#### Spring Security and MockMvc Set Up

Spring Security with Spring MVC test can be used with the configuration of Spring Security FilterChainProxy as a Filter. The following set up performs the same.

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
class JavaBasedSpringSecurityApplicationTests
{
    @Autowired
    private MockMvc mockMvc;
}
```

**@RunWith:** `@RunWith(SpringRunner.class)` instructs JUnit to run using Spring's testing support. `SpringRunner` is the new name for `SpringJUnit4ClassRunner`.

**@SpringBootTest:** This annotation instructs Spring Boot to use main class annotated with `@SpringBootApplication` to start a Spring application context. Spring boot application testing requires nothing special to do. It's **ApplicationContext** that needs to be available to test Spring Boot application. `@RunWith` instructs the spring-test module to create an `ApplicationContext`. `SpringBootTest` loads complete application and injects all the beans, which can be slow.

**@AutoConfigureMockMvc:** Spring provides a way to simulate the HTTP request in which application code will be executed exactly the same way as it were processing a

real HTTP request but without any server start. For this simulation use, Spring's MockMvc and that can be configured using `@AutoConfigureMockMvc`.

### SecurityMockMvcRequestPostProcessors

RequestPostProcessor interface is provided by Spring MVC Test. These request processors can be used to modify the request. Spring Security has provided many implementations of RequestPostProcessor to make testing easy and efficient. Spring Security's RequestPostProcessor implementations can be used by static import of as below:

```
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
```

There are two ways to run the test cases as a specific User in Spring MVC Test.

- ... Running the test case as a specific user with RequestPostProcessor
- ... Running the test case as a specific user with Annotations

### Running the test case as a user with RequestPostProcessor

The following test will run as a specific user, which need not to exist since we are mocking the user, with the username "user", the password "password", and the role "ROLE\_USER":

```
@Test
public void testAlreadyLoggedInUser() throws Exception
{
    mvc.perform(get("/").with(user("user"))).andExpect(authenticated().withUsername("user"))
        .andExpect(authenticated().withRoles("USER"))
        .andExpect(model().attribute("message",
            "User, Spring Security with custom login page implemented Successfully !!"));
}
```

In the set up MockMvc is autoconfigured as mvc and it is being used to simulate the HTTP request. Here RequestPostProcessoruser() has been used to modified the get("/") request. User RequestPostProcessor can be customized for password and roles as –

```
mvc.perform(get("/").with(user("John").password("pass@123").roles("USER","ADMIN")))
```

Test can be run as an anonymous user with following configuration.

```
mvc.perform(get("/").with(anonymous()))
```

### Running the test case as a user with Annotation

Instead of RequestPostProcessor to create a user, annotation can be used for the same. The following test will run as a specific user, which does not need to exist since we are mocking the user, with the username "user", the password "password", and the role "ROLE\_USER":

```
@Test
@WithMockUser
public void testAlreadyLoggedInUser() throws Exception
{
    mvc.perform(get("/").andExpect(authenticated().withUsername("user"))
        .andExpect(authenticated().withRoles("USER"))
        .andExpect(model().attribute("message",
            "User, Spring Security with custom login page implemented Successfully !!"));
}
```

@WithMockUser can be customized with the following attributes:

```
... username: String
... authorities: String[]
... password: String
... roles: String[]
```

## SecurityMockMvcRequestBuilders

RequestBuilder interface is provided by Spring MVC Test. These request builders can be used to create the MockHttpServletRequest into the test. Few implementations of RequestBuilder has been provided by Spring Security to make testing easy and efficient. Spring Security's RequestBuilder implementations can be used by static import of as below:

```
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilders.*;
```

## Form Based Authentication Testing

Form based authentication can be tested very easily in Spring Boot with the help of SecurityMockMvcRequestBuilders. Below will submit a POST request to “/login” with the username as “user”, the password as “password” and valid CSRF token.

```
mvc.perform(formLogin())
```

Customization of formLogin() is very easy and an example of it is given below.

```
@Test
public void testFormLogin() throws Exception
{
    mvc.perform(formLogin("/signin").user("testadmin").password("admin@123").andExpect(
        status().isFound())
        .andExpect(redirectedUrl("/")).andExpect(authenticated().withUsername("testadmin"))
        .andExpect(authenticated().withRoles("ADMIN", "USER")));
}
```

In above configuration a POST request for “/signin” with the username as “testadmin” and the password as “admin@123” with a valid CSRF token will be submitted. Parameters name can be customized as follows -

```
mvc.perform(formLogin("/signin").user("uname","admin@gmail.com").password("password","admin@"))
```

## Logout Testing Using SecurityMockMvcRequestBuilders

Logout functionality can be tested very easily using SecurityMockMvcRequestBuilders. Example for Logout test is as followings-

```
mvc.perform(logout())
```

Logout processing URL can be customized as

```
mvc.perfrom(logout("/signout"))
```

Check the folder src/test for unit test cases. All unit test cases can be executed by right click on the project -> Run As -> JUnit Test as shown below screenshot in Figure 13.6

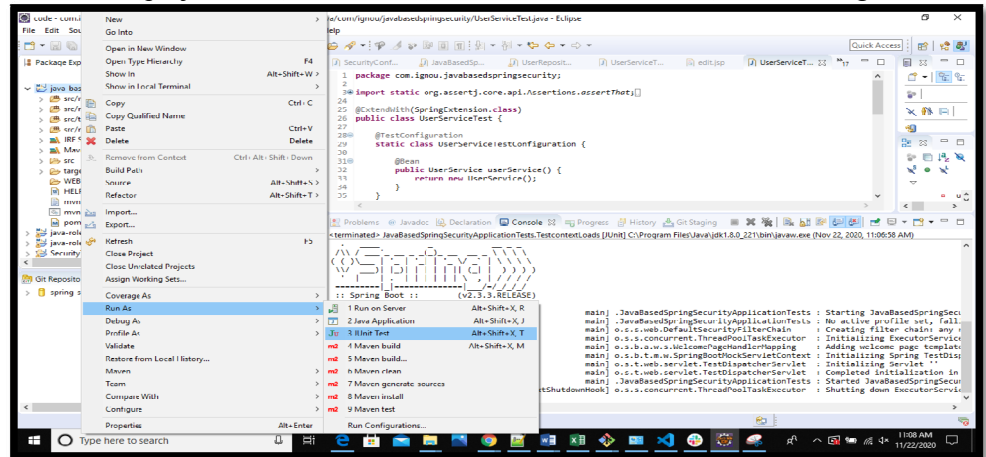


Figure 13.6: UnitCase Execution From Eclipse IDE

Successful execution of Unit test case will output the following. As you can see that all test cases are passed that is shown by green tick and green progress bar in the Figure 13.7.

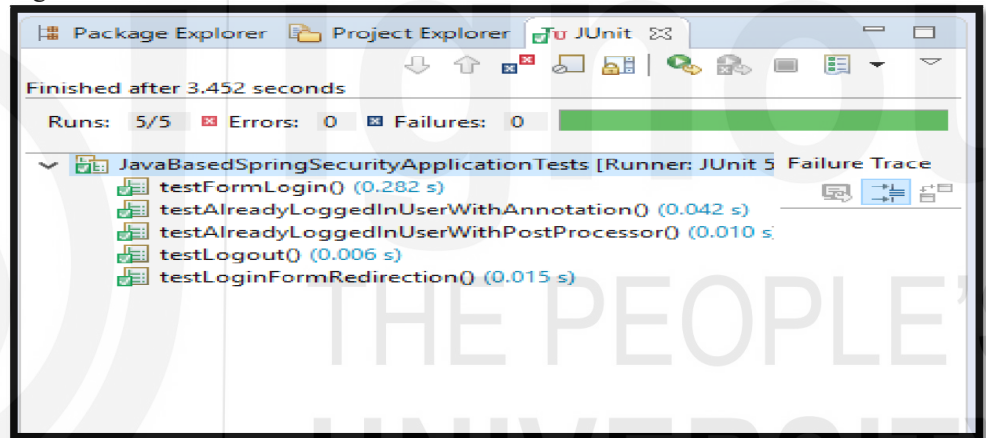


Figure 13.7: Unit Test Cases Execution Log

## 13.6 ADDING LOGOUT SUPPORT

The basic configuration of Spring Logout functionality using the logout() method is simple enough:

```
@Configuration
@EnableWebSecurity
public class SecSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Override
    protected void configure(final HttpSecurity http) throws Exception
    {
        http
            //...
        .logout()
            //...
    }
    //...
}
```

The above configuration enables default logout mechanism with processing url: /logout, which used to be /j\_spring\_security\_logout before Spring Security 4. We can

customize the logout feature in spring security. Let us see the advanced customization of spring logout.

### ***logoutSuccessUrl()***

On successfully logout operation, Spring Security will redirect the user to a specified page. By default, this is the root page ("/"), but this is configurable:

```
//...
.logout()
.logoutSuccessUrl("/successlogout")
//...
```

### ***logoutUrl()***

Like other defaults in Spring Security, logout mechanism has a default processing URL as well – /logout. To hide the information regarding which framework has been used to secure the application, it's better to change the default value. It can be done as below:

```
// ...
.logout()
.logoutUrl("/do_logout")
// ...
```

### ***invalidateHttpSession and deleteCookies***

These two advanced attributes control the session invalidation as well as a list of cookies to be deleted when the user logs out. Default value for invalidateHttpSession is true on logout.

```
//...
.logout()
.logoutUrl("/do_logout")
.invalidateHttpSession(true)
.deleteCookies("JSESSIONID")
// ...
```

Logout feature can be tested by enabling the logout option for the user. Add the logout option by doing the changes into index.jsp as shown below.

```
<!DOCTYPEhtml>
<%@taglibprefix="spring"uri="http://www.springframework.org/tags"%>
<html1lang="en">
<body>
    <div>
        <div>
            <astyle="text-align: left;" href="/Logout">logout</a>
            <h1>Spring Boot Security Example</h1>
            <h2>Hello ${message}</h2>
        </div>
    </div>
</body>
</html>
```

Logout configuration into WebSecurityConfig.java is described below.

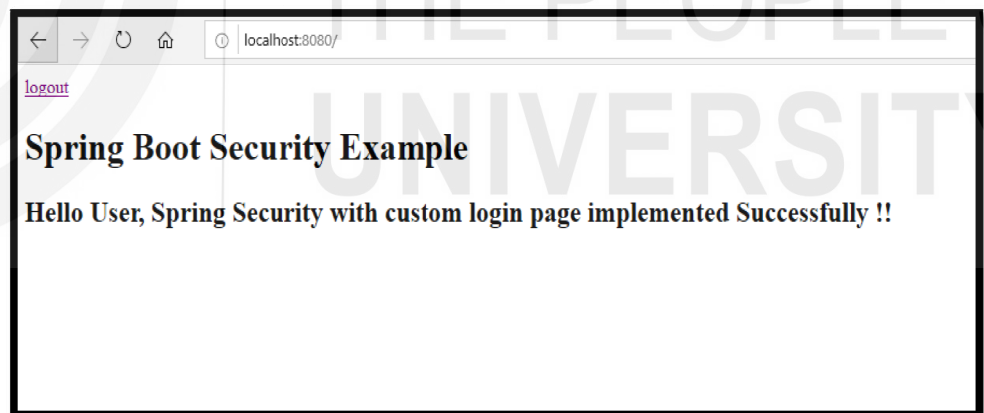
```

package com.ignou.javabasedspringsecurity.security;

@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Autowired
    PasswordEncoder passwordEncoder;
    @Bean
    public PasswordEncoder passwordEncoder()
    {
        return new BCryptPasswordEncoder();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
    {
        auth.inMemoryAuthentication().passwordEncoder(passwordEncoder).withUser("testuser")
            .password(passwordEncoder.encode("user@123")).roles("USER").and().withUser("testadmin")
            .password(passwordEncoder.encode("admin@123")).roles("USER", "ADMIN");
    }
    protected void configure(HttpSecurity http) throws Exception
    {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/customlogin*").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/customlogin")
            .loginProcessingUrl("/signin")
            .defaultSuccessUrl("/", true)
            .failureUrl("/customlogin?error=true")
            .and()
            .logout();
    }
}

```

Execute the application and after successful login you will see logout option as shown below in Figure 13.8:



**Figure 13.8: Success Screen with Logout Feature**

After clicking logout link, user will be logged out and redirected to login page as shown in Figure 13.9



Spring Security Custom Login Form

Username

Enter Username

Password

Enter Password

Login

☒ Remember me

Cancel

Forgot [password?](#)

Figure 13.9: Success Screen with Logout Feature

## Check Your Progress 2

- 1) Explain the usage of `@SpringBootTest`.  
.....  
.....  
.....  
.....
- 2) Write the unit test case to execute it as a user with `RequestPostProcessor` for the URL pattern `"/"` which returns model attribute with key as `"message"` and value as `"Hello World!!"`  
.....  
.....  
.....  
.....
- 3) Write the unit test case to execute it as a user with Annotation for the URL pattern `"/"` which returns model attribute with key as `"message"` and value as `"Hello World!!"`  
.....  
.....  
.....  
.....
- 4) Write Unit test case to test the custom login form having `"/signin"` as `loginProcessingUrl` and `"/"` as `defaultSuccessUrl`.  
.....  
.....  
.....  
.....

## 13.7 SUMMARY

Default Spring Security login form is not suitable for enterprises application since developers do not have full control over css and form design. Spring Security provides a way to configure custom login form in Spring Application. This unit has explained the configuration of custom login form with an example and its execution.

Unit test plays a vital role to ensure the correctness of code, and it identifies every defect that may arise before code is integrated with other modules. Many Spring Boot annotations available for Integration testing, such as `@SpringBootTest`, `@AutoConfigureMockMvc`, `@WithMockUser` has been explained.

Logout feature is an important feature from security perspective. At the end of this unit, Spring provided logout support has been explained with configurable properties.

## 13.8 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

### ☛ Check Your Progress 1

- 1) By default, spring boot provides default login screen for spring security. In enterprise application using default login screen, enterprise may not have full control over design, processing URLs etc. Custom login form in Spring Application is nothing but a simple html or jsp, like others web pages in java. Sample custom login form is shown below-

```
<html>
<body>
<div>
<center>Custom Spring Login Form</center>
<form action="/signin" method="post">
<input name="username" type="text" placeholder="Enter Username" />
<input name="password" type="text" placeholder="Enter password" />
<input type="submit" />
</form>
</div>
</body>
</html>
```

Configuration for custom login page in spring security is very easy. We just need to set the `loginPage` in `FormLoginConfigurer` object provided by `formLogin()`. Sample Spring security configuration with custom login page is shown below.

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Autowired
    PasswordEncoder passwordEncoder;
    @Bean
```

```

public PasswordEncoder passwordEncoder()
{
    return new BCryptPasswordEncoder();
}
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
    auth.inMemoryAuthentication().passwordEncoder(passwordEncoder).withUser("testuser")
        .password(passwordEncoder.encode("user@123")).roles("USER").and().withUser("testadmin")
        .password(passwordEncoder.encode("admin@123")).roles("USER", "ADMIN");
}
protected void configure(HttpSecurity http) throws Exception
{
    http.csrf().disable()
        .authorizeRequests()
        .antMatchers("/customlogin").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .loginPage("/customlogin")
        .loginProcessingUrl("/signin")
        .defaultSuccessUrl("/", true)
        .failureUrl("/customlogin?error=true");
}
}

```

- 2) loginPage("/dologin") instructs Spring Security to perform following-
  - ... Whenever authentication is required, its being redirected to /dologin
  - ... Login page is being rendered when /dologin is requested
  - ... It will redired to /dologin?error(if default configuration is used for error) in the case of authentication failure.
  - ... On successful logout, it will redirect to /dologin?logout (In the case fo default configuration used)

loginProcessingUrl(/signin) instruct the Spring Security to validate the submitted credentials sent for /signinUrl. By default it redirects the user back to the page from where user came. Request will neither be passed to Spring MVC nor controller.

## Check Your Progress 2

- 1) Spring-Boot provides @SpringBootTest annotation. This annotation provided spring boot feature in the test module and works by creating the ApplicationContext used in tests through SpringApplication. It starts the embedded server, creates a web environment and enables @Test methods to do integration testing.  
By default, no server is started by @SpringBootTest. It provides several options to add webEnvironment regarding how test cases should be executed.
  - ... MOCK(Default): Loads a web ApplicationContext and provides a mock web environment
  - ... RANDOM\_PORT: Loads a WebServerApplicationContext and provides a real web environment. The embedded server is started and listen to on a random port. This is the one that should be used for the integration test
  - ... DEFINED\_PORT: Loads a WebServerApplicationContext and provides a real web environment.
  - ... NONE: Loads an ApplicationContext by using SpringApplication but does not provide any web environment

- 2) RequestPostProcessor interface is provided by Spring MVC Test. These request processors can be used to modify the request. Many implementations of RequestPostProcessor has been provided by Spring Security to make testing easy and efficient. Spring Security's RequestPostProcessor implementations can be used by static import of as below:

Import static

```
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
```

Sample code to execute a unit test case with RequestPostProcessor is as followings-

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
class JavaBasedSpringSecurityApplicationTests
{
    @Autowired
    private MockMvc mvc;
    @Test
    public void testAlreadyLoggedInUserWithPostProcessor() throws Exception
    {
        mvc.perform(get("/").with(user("user"))).andExpect(authenticated().withUsername("user"))
            .andExpect(authenticated().withRoles("USER"))
            .andExpect(model().attribute("message", "Hello World!!"));
    }
}
```

- 3) Sample code with Annotation is as below:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
class JavaBasedSpringSecurityApplicationTests
{
    @Autowired
    private MockMvc mvc;
    @Test
    @WithMockUser()
    public void testAlreadyLoggedInUserWithAnnotation() throws Exception
    {
        mvc.perform(get("/").andExpect(authenticated().withUsername("user"))
            .andExpect(authenticated().withRoles("USER"))
            .andExpect(model().attribute("message", "Hello World!!")));
    }
}
```

- 4) Sample code with Annotation is as below:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
class JavaBasedSpringSecurityApplicationTests
{
    @Autowired
    private MockMvc mvc;
    @Test
    public void testFormLogin() throws Exception
    {
        mvc.perform(formLogin("/signin").user("testadmin").password("admin@123"))
            .andExpect(status().isFound())
            .andExpect(redirectedUrl("/")).andExpect(authenticated().withUsername("testadmin"))
            .andExpect(authenticated().withRoles("ADMIN", "USER"));
    }
}
```

---

## 13.9 REFERENCES/FURTHER READING

---

- ... Craig Walls, “Spring Boot in action” Manning Publications, 2016.  
(<https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf>)
- ... Christian Bauer, Gavin King, and Gary Gregory, “Java Persistence with  
Hibernate”,Manning Publications, 2015.
- ... Ethan Marcotte, “Responsive Web Design”, Jeffrey Zeldman Publication,  
2011([http://nadin.miem.edu.ru/images\\_2015/responsive-web-design-2nd-  
edition.pdf](http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf))
- ... Tomcy John, “Hands-On Spring Security 5 for Reactive Applications”,Packt  
Publishing,2018
- ... [https://docs.spring.io/spring-  
security/site/docs/4.2.19.RELEASE/guides/html5/form-javaconfig.html](https://docs.spring.io/spring-security/site/docs/4.2.19.RELEASE/guides/html5/form-javaconfig.html)
- ... <https://www.baeldung.com/spring-security-login>
- ... <https://www.baeldung.com/spring-boot-testing>
- ... [https://docs.spring.io/spring-  
framework/docs/current/reference/html/testing.html](https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html)



ignou  
THE PEOPLE'S  
UNIVERSITY

---

## UNIT 14    ROLE BASED LOGIN

---

### Structure

- 14.0 Introduction
- 14.1 Objectives
- 14.2 Display User Id and Roles – Overview
  - 14.2.1 User Details in a Controller
  - 14.2.2 Spring view layer security and User Details using JSP Taglibs
- 14.3 Roles based Login Example
- 14.4 Restrict Access based on Roles
- 14.5 Testing the Application
  - 14.5.1 Unit Tests in Spring Boot
    - 14.5.1.1 Data Layer or Repository Layer testing with `@DataJpaTest`
    - 14.5.1.2 Service Layer and Controller layer testing with `@MockBean`
- 14.6 Cross Site Request Forgery (CSRF)
- 14.7 Summary
- 14.8 Solutions/ Answer to Check Your Progress
- 14.9 References/Further Reading

---

## 14.0    INTRODUCTION

---

Role-Based access control (RBAC) allows us to define the accessibility based on user's roles and permissions. The roles in RBAC refer to the levels of access that user has to the resource. Roles and permissions are used to make a resource secured from unauthorized access. RBAC allows to define what end-users can perform at both broad and granular levels. While using RBAC, resource access of the users is analyzed, and users are grouped into roles based on common responsibilities and needs. Each user into the system is assigned with one or more roles and one or more permissions to each role. For an example, an admin can have permission to create a new post and edit the already created post while an editor may have permission to edit the post which already exists.

This Unit explains how to restrict resource access based on roles using the Spring security. Most of the applications display the logged in user details. This unit describes the various approaches to get logged in user details in controllers. It also describes how to retrieve security related information at view layer in jsp.

The previous unit explained how to write integration test cases using annotations provided by Spring. Good unit test cases make the codebase auto deployable and production ready since codebase can be validated with the help of unit test cases. This unit describes the unit test cases with available annotations in Spring.

Cross site request forgery, also known as CSRF or XSRF, is a type of attack in which attackers trick a victim to make a request that utilizes their authentication or authorization. The victim's level of permission decides the impact of CSRF attack. In the end of this Unit, Cross Site Request Forgery (CSRF) has been explained with an example and the solution to avoid CSRF attack.

---

## 14.1 OBJECTIVES

---

After going through this unit, you should be able to:

- ... explain Spring Security core component,
  - ... describe logged in user detail in controllers,
  - ... use Spring Security Expressions,
  - ... describe and use Authentication and Authorization information at view layer,
  - ... apply Role-Based Access Control and restrict the access based on roles, and
  - ... describe Cross Site Request Forgery (CSRF).
- 

## 14.2 DISPLAY USER ID AND ROLES – OVERVIEW

---

Various core classes and interface available in spring security to get security context are outlined. Spring Security core components are -

- ... **SecurityContextHolder:** A Class which provide access to SecurityContext
- ... **SecurityContext:** An Interface having Authentication and defining the minimum-security information associated with request
- ... **Authentication:** Represents the **Principal** in Spring security-specific way. The Spring Security **Principal** can only be retrieved as an Object from Authentication and needs to be cast to the correct UserDetails instance as –

```
UserDetails userDetails = (UserDetails)
authentication.getPrincipal();
System.out.println("User has authorities: "+
userDetails.getAuthorities());
```

- ... **GrantedAuthority:** Having the application wide permissions granted to a principal
- ... **UserDetails:** UserDetails has the required information to build an Authentication object from application's DAOs or other sources of security data.

The following sections explain how to retrieve user details in Spring Security. There are various ways to access logged-in user information in the Spring. Few ways are described below.

### 14.2.1 User Details in a Controller

In a `@Controller` annotated bean, principal can be defined as a method argument, and the Spring framework will resolve it correctly.

```
@Controller
public class UserSecurityController
{
    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String loggedInUserName(Principal principal)
    {
        return principal.getName();
    }
}
```

Instead of Principal, Authentication can be used. Authentication allows us to get the granted authorities using `getAuthorities()` method while Spring Security principal can only be retrieved as an Object and needs to be cast to the correct `UserDetails` instance. Sample code is as following-

```
@Controller
public class UserSecurityController
{
    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String loggeInUserName(Authentication authentication)
    {
        return authentication.getName();
    }
}
```

User Details can be retrieved from HTTP request as -

```
@Controller
public class UserSecurityController
{
    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String loggeInUserName(HttpServletRequest request)
    {
        Principal principal = request.getUserPrincipal();
        return principal.getName();
    }
}
```

User Details can be retrieved in a Bean as –

```
Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
String currentPrincipalName = authentication.getName();
```

### 14.2.2 Spring view layer security and User Details using JspTaglibs

Spring Security provides its own taglib for basic security support, such as retrieving security information and applying security constraints at view layer jsp. To use spring security features in jsp, you need to add the following tag library declaration into jsp-

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="security" %>
```

The tags provided by spring security to access security information and to secure the view layer of the application are as below:

- ... Authorize Tag
- ... Authentication Tag
- ... Accesscontrollist Tag
- ... Csrfinput Tag
- ... CsrffMetaTags Tag

**Authorize Tag:** This tag supports to secure information based on user's role or permission to access a URL. Authorize tag support two types of attributes.

- ... access
- ... url

In an application, view layer might need to display certain information based on user's role or based on authentication state. An example for the same is shown below.

```
<security:authorize access="!isAuthenticated()">
    Login
</security:authorize>
<security:authorize access="isAuthenticated()">
```



```

    Logout
</security:authorize>

```

Further, we can also display certain information based on user role as-

```

<security:authorize access="hasRole('ADMIN')">
    Delete Users
</security:authorize>

```

Value for access can be any of Spring Security Expression as below-

- ... `hasAnyRole('ADMIN','USER')` returns true if the current user has any of the listed roles
- ... `isAnonymous()` returns true if the current principal is an anonymous user
- ... `isRememberMe()` returns true if the current principal is a remember-me user
- ... `isFullyAuthenticated()` returns true if the user is authenticated and is neither anonymous nor a remember-me user

Using url attribute in authorize tag, at view layer we can check whether user is authorized to send request to certain URL:

```

<security:authorizeurl="/inventoryManagement">
<a href="/inventoryManagement">Manage Inventory</a>
</security:authorize>

```

**Authentication Tag:** This tag is not used to implement security but it allows access to the current Authentication object stored in the security context. It renders a property of the object directly in the JSP. If the principal property of the Authentication is an instance of Spring Security's UserDetails object, then username and roles can be accessed as –

```

... <security:authentication property="principal.username" />
... <security:authentication property="principal.authorities" />

```

**Accesscontrollist Tag:** This tag is used with Spring Security's ACL module. It checks list of required permissions for the specified domains. It executes only if current user has all the permissions.

**Csrfinput Tag:** For this tag to work, csrf protection should be enabled in spring application. If csrf is enabled, spring security inserts a csrf hidden form input inside `<form:form>` tag. But in case if we want to use html `<form></form>`, we need to put csrfinput tag inside `<form>` to create CSRF token as below –

```

<form method="post" action="/some/action">
<security:csrfInput />
    Name:<input type="text" name="username" />
...
</form>

```

**CsrfMetaTags Tag:** If we want to access CSRF token in javascript, we have to insert token as a meta tag. For this tag to work, csrf protection should be enabled in Spring Application.

```

<html>
<head>
<title>JavaScript with CSRF Protection</title>
<security:csrfMetaTags />
<script type="text/javascript" language="javascript">
    var csrfParameter =
    $("meta[name='_csrf_parameter']").attr("content");
    var csrfHeader = $("meta[name='_csrf_header']").attr("content");
    var csrfToken = $("meta[name='_csrf']").attr("content");
</script>
</head>
<body>
...

```

## ☛ Check Your Progress 1

- 1) What are Authorities in Authentication object?  
.....  
.....  
.....  
.....
- 2) Write the code to get logged in user's username from HttpServletRequest in a controller.  
.....  
.....  
.....  
.....
- 3) Explain with an example to display username and roles on jsp.  
.....  
.....  
.....  
.....

---

## 14.3 ROLES BASED LOGIN EXAMPLE

---

This section describes role-based login using Spring Security. Role based login means redirecting users to different URLs upon successful login according to the assigned role to the logged-in user. The following section explains an example with three types of users as Admin, Editor and normal User. Once user successfully logged into the system, based on role user will be redirected to its own url as –

- ... **Admin** will be redirected to **/admin**
- ... **Editor** will be redirected to **/editor**
- ... **User** will be redirected to **/home**

In previous spring security configurations, the success URL on successful authentication was configured using **defaultSuccessUrl(Sring s)**. With the use of Spring **defaultSuccessUrl("/homePage")**, every user will be redirected to homepage url irrespective of role. To have more control over the authentication success mechanism, **Spring Security** provides a component that has the direct responsibility of deciding what to do after a successful authentication – the **AuthenticationSuccessHandler**.

**AuthenticationSuccessHandler** is an Interface and Spring provides **SimpleUrlAuthenticationSuccessHandler**, which implement this interface. For role based login, you need to configure **successHandler(AuthenticationSuccessHandlersuccessHandler)** instead of **defaultSuccessUrl(Sring s)**. For custom success handler, you have two approaches.

1. **Implement *AuthenticationSuccessHandler*** interface to provide custom success handler.

2. **Extend**`SimpleUrlAuthenticationSuccessHandler` class and override the `handle()` method to provide the logic.

The below code sample is for custom success handler using approach 2.

```
public class CustomSuccessHandler extends SimpleUrlAuthenticationSuccessHandler
{
    private RedirectStrategy redirectStrategy = new DefaultRedirectStrategy();

    @Override
    protected void handle(HttpServletRequest request, HttpServletResponse response,
        Authentication authentication) throws IOException
    {
        String targetUrl = determineTargetUrl(authentication);

        if (response.isCommitted())
        {
            System.out.println("Can't redirect");
            return;
        }

        redirectStrategy.sendRedirect(request, response, targetUrl);
    }

    /**
     * This method extracts the roles of currently logged-in user and returns
     * appropriate URL according to his/her role.
     */
    protected String determineTargetUrl(Authentication authentication)
    {
        String url = "";
        Map<String, String> roleTargetUrlMap = new HashMap<>();
        roleTargetUrlMap.put("ROLE_USER", "/home");
        roleTargetUrlMap.put("ROLE_ADMIN", "/admin");
        roleTargetUrlMap.put("ROLE_Editor", "/editor");

        Collection<? extends GrantedAuthority> authorities =
            authentication.getAuthorities();
        List<String> roles = new ArrayList<String>();
        for (GrantedAuthority a : authorities)
        {
            roles.add(a.getAuthority());
        }

        if (isAdmin(roles))
        {
            url = "/admin";
        }
        elseif (isEditor(roles))
        {
            url = "/editor";
        }
        elseif (isUser(roles))
        {
            url = "/home";
        }
        else
        {
            url = "/accessDenied";
        }

        return url;
    }

    private boolean isUser(List<String> roles)
    {
        if (roles.contains("ROLE_USER"))
        {
            return true;
        }
        return false;
    }

    private boolean isAdmin(List<String> roles)
    {
        if (roles.contains("ROLE_ADMIN"))
    }
```

```

        {
            return true;
        }
        return false;
    }

    private boolean isEditor(List<String> roles)
    {
        if (roles.contains("ROLE_EDITOR"))
        {
            return true;
        }
        return false;
    }

    public void setRedirectStrategy(RedirectStrategy redirectStrategy)
    {
        this.redirectStrategy = redirectStrategy;
    }

    protected RedirectStrategy getRedirectStrategy()
    {
        return redirectStrategy;
    }
}

```

Above class overrides the handle method and redirects the user based on his role as follows –

- ... **Admin** will be redirected to **/admin**
- ... **Editor** will be redirected to **/editor**
- ... **User** will be redirected to **/home**

Spring Security Configuration for role-based login using successHandler is shown below.

```

@Override
protected void configure(final HttpSecurity http) throws Exception
{
    http
        .authorizeRequests()
            // endpoints
        .formLogin()
        .loginPage("/login.html")
        .loginProcessingUrl("/login")
        .successHandler(customSuccessHandler())
            // other configuration
    }
}

```

## 14.4 RESTRICT ACCESS BASED ON ROLES

Role-based access control (RBAC) is a mechanism that restricts system access. RBAC accomplishes this by assigning one or more "roles" to each user and giving each role different permissions. To protect sensitive data, mostly large-scale organizations use role-based access control to provide their employees with varying levels of access based on their roles and responsibilities.

The previous unit explained to secure the application using authentication and URL level security along with restrictions based on roles. The following sections explain to secure the application at service layer. Method-level security is used to restrict access based on Role. The last of this section explains to add security at view layer i.e.jsp.

Spring allows us to implement the security at the method level. Security applied at the method level restricts the unauthorized user to access the resource mapped by the method. How to enable method level security and restrict the method accessed based on role is explained below.

## @EnableGlobalMethodSecurity

To Enable the method level security in spring we need to enable global method security as below:

```
@Configuration
@EnableGlobalMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = true)
public class MethodSecurityConfig
    extends GlobalMethodSecurityConfiguration
{
}
```

- ... The prePostEnabled property enables Spring Security pre/post annotations
- ... The securedEnabled property determines if the @Secured annotation should be enabled
- ... The jsr250Enabled property allows us to use the @RoleAllowed annotation

## @Secured

@Secured annotation is used to implement method level security based on Role. This annotation accepts list of allowed roles to be permitted. Hence, a user can access a method if he/she has been assigned at least one role into the list.

```
@Secured("ROLE_USER")
public String getUsername()
{
    SecurityContext securityContext = SecurityContextHolder.getContext();
    return securityContext.getAuthentication().getName();
}
```

User who is having role as ROLE\_USER can execute the above method.

```
@Secured({ "ROLE_USER", "ROLE_ADMIN" })
public boolean isValidUsername(String username)
{
    return userRoleRepository.isValidUsername(username);
}
```

A user having Either "ROLE\_USER" OR "ROLE\_ADMIN" can invoke above method.

## @RolesAllowed Annotation

The @RolesAllowed annotation is the JSR-250's equivalent annotation of the @Secured annotation. Basically, we can use the @RolesAllowed annotation in a similar way as @Secured.

```
@RolesAllowed("ROLE_USER")
public String getUsername()
{
    //...
}

@RolesAllowed({ "ROLE_USER", "ROLE_ADMIN" })
public boolean isValidUsername(String username)
{
    //...
}
```

## @PreAuthorize and @PostAuthorize Annotations

Both `@PreAuthorize` and `@PostAuthorize` annotations provide expression-based access control. The `@PreAuthorize` annotation checks the given expression before entering into the method, whereas, the `@PostAuthorize` annotation verifies it after the execution of the method and could alter the result.

```
@PreAuthorize("hasRole('ROLE_VIEWER')")
public String getUsername()
{
    //...
}
@PreAuthorize("hasRole('ROLE_USER') or hasRole('ROLE_ADMIN')")
public boolean isValidUsername(String username)
{
    //...
}
@PreAuthorize("#username == authentication.principal.username")
public String getMyRoles(String username)
{
    //...
}
```

`getMyRoles()` method can be invoked by a user only if the value of the argument *username* is the same as current principal's username

`@PostAuthorize` annotation is similar to `@PreAuthorize` so it can be replaced with `@PostAuthorize`.

### Restriction at view layer

Access restriction based on role at view layer using spring security authorize tag is already explained. For details check the Section 14.2. Example is as follows-

```
<security:authorize access="!isAuthenticated()">
    Login
</security:authorize>
<security:authorize access="isAuthenticated()">
    Logout
</security:authorize>
```

Further we can also display certain information based on user role as-

```
<security:authorize access="hasRole('ADMIN')">
    Delete Users
</security:authorize>
```

### Check Your Progress 2

- 1) Describe role-based login. Write the sample code to configure the role-based login.

.....

.....

.....

.....

.....

- 2) What is Role Based Access Control (RBAC)? Write the sample configuration code to restrict the URL access based on role.

.....

.....

.....

- .....
- .....
- 3) Explain @Secured and @RolesAllowed. What is the difference between them?

.....

.....

.....

.....

.....

- 4) Which are all spring security annotations allowed to use SpEL?

.....

.....

.....

.....

.....

- 5) What's the difference between @Secured and @PreAuthorize in spring security?

.....

.....

.....

.....

.....

- 6) Explain spring security tag which restricts the access based on roles in jsp.

.....

.....

.....

.....

.....

## 14.5 TESTING THE APPLICATION

This section writes a complete example with three types of users as Admin, Editor and User. UserType, Role and functionality has been defined in the table as follows:

| User Type | Role                       | Functionality                             | Landing Page |
|-----------|----------------------------|---|--------------|
| Admin     | ROLE_ADMIN,<br>ROLE_EDITOR | Admin can create or edit a post/messages. | /admin       |
| Editor    | ROLE_EDITOR                | Editor can only edit the post/messages.   | /editor      |
| User      | ROLE_USER                  | User can only view the post/messages.     | /home        |

In units 12 and 13, In-Memory authentication has been configured. Here JDBC authentication with mysql database has been explained. Tools and software required

to write the code easily, manage the spring dependencies and execute the application is as -

- ... Java 8 or above is required
- ... Maven
- ... Eclipse IDE
- ... Mysql
- ... Hibernate
- ... JPA

Directory Structure for Spring Project in eclipse is shown below. A few of the code is shown in Figure 14.1.

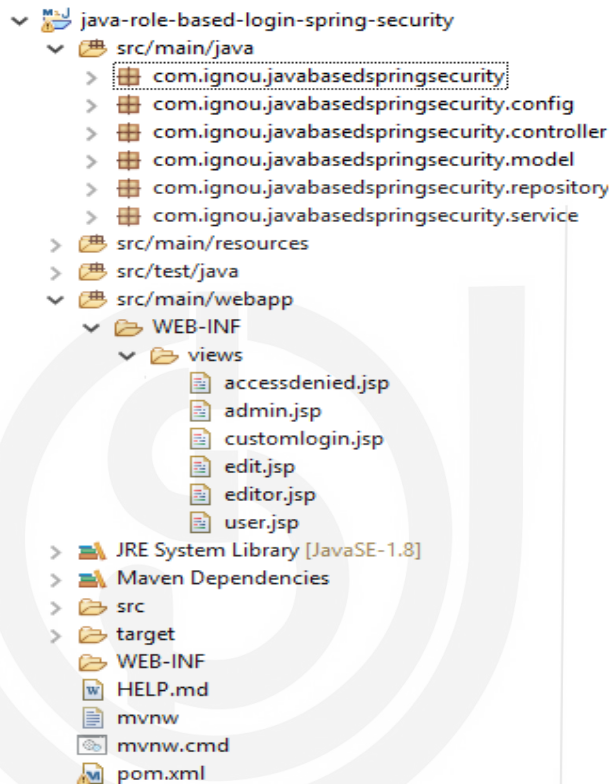


Figure 14.1: Project Folder Structure In Eclipse IDE

Spring Security Configuration classes in config packages – customSuccessHandler is already explained above for role-based login.

```
package com.ignou.javabasedspringsecurity.config;

@Component
public class CustomSuccessHandler extends SimpleUrlAuthenticationSuccessHandler
{
    private RedirectStrategy redirectStrategy = new DefaultRedirectStrategy();

    @Override
    protected void handle(HttpServletRequest request, HttpServletResponse response,
        Authentication authentication) throws IOException
    {
        String targetUrl = determineTargetUrl(authentication);

        if (response.isCommitted())
        {
            System.out.println("Can't redirect");
            return;
        }
    }
}
```



```

        redirectStrategy.sendRedirect(request, response, targetUrl);
    }

    /**
     * This method extracts the roles of currently logged-in user and returns
     * appropriate URL according to his/her role.
     */
    @protected String determineTargetUrl(Authentication authentication)
    {
        String url = "";
        Map<String, String>roleTargetUrlMap = new HashMap<>();
        roleTargetUrlMap.put("ROLE_USER", "/home");
        roleTargetUrlMap.put("ROLE_ADMIN", "/admin");
        roleTargetUrlMap.put("ROLE_Editor", "/editor");

        Collection<? extends GrantedAuthority>authorities =
            authentication.getAuthorities();

        List<String>roles = new ArrayList<String>();

        for (GrantedAuthoritya :authorities)
        {
            roles.add(a.getAuthority());
        }

        if (isAdmin(roles))
        {
            url = "/admin";
        }
        elseif (isEditor(roles))
        {
            url = "/editor";
        }
        elseif (isUser(roles))
        {
            url = "/home";
        }
        else
        {
            url = "/accessDenied";
        }

        returnurl;
    }

    privatebooleanisUser(List<String>roles)
    {
        if (roles.contains("ROLE_USER"))
        {
            returntrue;
        }
        returnfalse;
    }

    privatebooleanisAdmin(List<String>roles)
    {
        if (roles.contains("ROLE_ADMIN"))
        {
            returntrue;
        }
        returnfalse;
    }

    privatebooleanisEditor(List<String>roles)
    {
        if (roles.contains("ROLE_EDITOR"))
        {
            returntrue;
        }
        returnfalse;
    }

    publicvoidsetRedirectStrategy(RedirectStrategyredirectStrategy)
    {
        this.redirectStrategy = redirectStrategy;
    }

```

```

    protectedRedirectStrategygetRedirectStrategy()
    {
        returnredirectStrategy;
    }
}

```

### SecurityConfig.java

Complete Security configuration with successHandler(customSuccessHandler) is as below-

```

package com.ignou.javabasedspringsecurity.config;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter
{

    @Autowired
    private UserDetailsServiceImpl customUserService;

    @Autowired
    private CustomSuccessHandler customSuccessHandler;

    @Autowired
    PasswordEncoder passwordEncoder;

    @Bean
    public PasswordEncoder passwordEncoder()
    {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
    {
        auth
            .userDetailsService(customUserService)
            .passwordEncoder(passwordEncoder);
    }

    protected void configure(HttpSecurity http) throws Exception
    {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/customlogin*").permitAll()
            .antMatchers("/", "/home").hasRole("USER")
            .antMatchers("/editor/**").hasRole("EDITOR")
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/edit/**").hasAnyRole("ADMIN", "EDITOR")
            .antMatchers("/create/**").hasAnyRole("ADMIN")
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/customlogin")
            .loginProcessingUrl("/signin")
            .successHandler(customSuccessHandler)
            .failureUrl("/customlogin?error=true")
            .and()
            .logout()
            .and().exceptionHandling().accessDeniedPage("/accessdenied");
    }
}

```

### Controller package Classes

#### LoginController.java

```

package com.ignou.javabasedspringsecurity.controller;

@Controller
@RequestMapping(value="/customlogin")

```

```

public class LoginController
{
    @GetMapping
    public String login()
    {
        return "customlogin";
    }
}

```

### HomeController.java

```
package com.ignou.javabasedspringsecurity.controller;
```

```

@Controller
public class HomeController
{
    private static Map<Long, String> msgs = new HashMap<>();
    private static long msgCount = 0;

    static
    {
        msgs.put(new Long(1), "Java Cryptography Architecture (JCA)");
        msgs.put(new Long(2), "Java Secure Socket Extension (JSSE)");
        msgCount = 2;
    }

    @RequestMapping(value="/home", method=RequestMethod.GET)
    public String user(Model model)
    {
        model.addAttribute("msgs", msgs);
        return "user";
    }

    @RequestMapping(value="/editor", method=RequestMethod.GET)
    public String editor()
    {
        return "editor";
    }

    @RequestMapping(value="/admin", method=RequestMethod.GET)
    public String admin()
    {
        return "admin";
    }

    @RequestMapping(value="/edit", method=RequestMethod.GET)
    public String edit(Model model)
    {
        model.addAttribute("msgs", msgs);
        return "edit";
    }

    @RequestMapping(value="/edit", method=RequestMethod.POST)
    public String editPost(@ModelAttribute MsgPost msgPost, Model model)
    {
        long msgId = msgPost.getId();
        if (msgs.get(msgId) != null)
        {
            msgs.put(msgPost.getId(), msgPost.getContent());
        }
        model.addAttribute("msgs", msgs);
        return "edit";
    }

    @RequestMapping(value="/create", method=RequestMethod.POST)
    public String createPost(@ModelAttribute MsgPost msgPost, Model model)
    {
        msgCount += 1;
        msgs.put(msgCount, msgPost.getContent());
        model.addAttribute("msgs", msgs);
        return "edit";
    }
}

```

}

## Service package classes

## UserService.java

```

package com.ignou.javabasedspringsecurity.service;

@Service
@Transactional
public class UserService implements UserDetailsService
{
    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String email)
    throws UsernameNotFoundException
    {
        User user = userRepository.findByEmail(email).orElseThrow(() -
        > new UsernameNotFoundException(email + "not found"));
        return new org.springframework.security.core.userdetails.User(user.getEmail(),
        user.getPassword(), getAuthorities(user));
    }

    private static Collection<? extends GrantedAuthority> getAuthorities(User user)
    {
        String[] userRoles = user.getRoles().stream().map((role) ->
        role.getName()).toArray(String[]::new);
        Stream.of(userRoles).forEach(System.out::print);
        Collection<GrantedAuthority> authorities =
        AuthorityUtils.createAuthorityList(userRoles);
        return authorities;
    }
}

```

## Jsp in WEB-INF/views

Note: customlogin.jsp has been used same as explained in Unit 13.

## admin.jsp

```

<%@taglib uri="http://www.springframework.org/security/tags" prefix="security"%>
<%@page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-
8859-1"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Admin page</title>
</head>
<body>
    Dear <strong><security:authentication property="principal.username"/></strong>,
    Welcome to Admin Page.
    <p>You are having roles as
    <strong><security:authentication property="principal.authorities"/></strong>
    <p><a href="/edit">Edit</a>
    <p><a href="/Logout">Logout</a>
</body>
</html>

```

## editor.jsp

```

<%@taglib uri="http://www.springframework.org/security/tags" prefix="security"%>
<%@page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-
8859-1"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Editor page</title>
</head>

```

```

<body>
    Dear <strong><security:authenticationproperty="principal.username"/></strong>,
    Welcome to Editor Page.
    <p>You are having roles as
    <strong><security:authenticationproperty="principal.authorities"/></strong>
    <p><a href="/edit">Edit</a>
    <p><a href="/Logout">Logout</a>
</body>
</html>

```

### user.jsp

```

<%@taglib uri="http://www.springframework.org/security/tags"
    prefix="security"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Welcome page</title>
<style>
table.border,th.border,td.border
{
    margin-left: auto;
    margin-right: auto;
    border: 1px solid black;
}
</style>
</head>
<body>
    Dear
    <strong><security:authenticationproperty="principal.username"/></strong>,
    Welcome to Home Page.
    <p>
        You are having roles as <strong><security:authentication
            property="principal.authorities"/></strong>
    <table class="border">
        <tr>
            <th class="border">Msg Id</th>
            <th class="border">Message</th>
        </tr>
        <c:forEach items="${msgs}" var="msg">
            <tr>
                <td class="border">${msg.key}</td>
                <td class="border">${msg.value}</td>
            </tr>
        </c:forEach>
    </table>
    <p>
        <a href="/Logout">Logout</a>
</body>
</html>

```

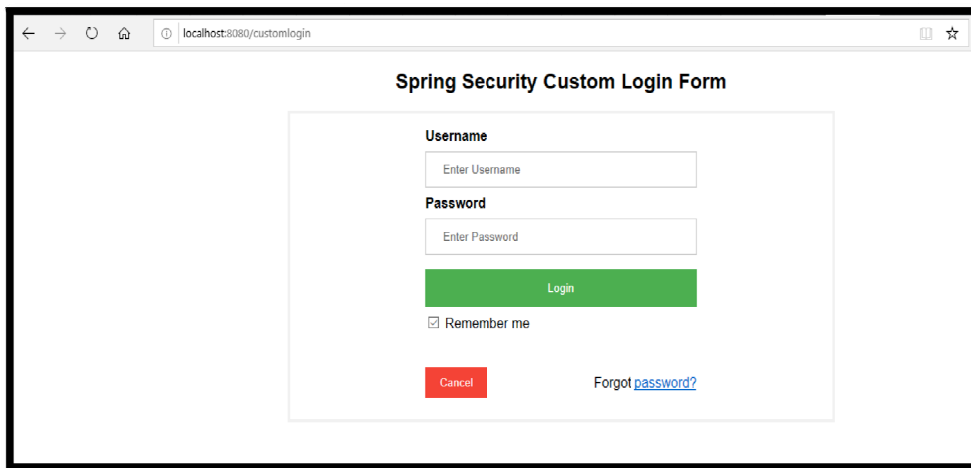
Execute the project either from eclipse or command line. After successful execution of the application, security configured application can be accessed using browser on localhost:8080

The following users are configured for the testing of role-based login example.

- ... Admin – UserName: **admin@gmail.com** Password: **admin**
- ... Editor – UserName: **editor@gmail.com** Password: **editor**
- ... User – UserName: **user@gmail.com** Password: **user**

### Execution Result

If a user tries to access **localhost:8080** and the user is not logged in, spring will redirect to the user on **localhost:8080/customlogin** as shown in Figure 14.2.



A screenshot of a web browser showing a custom login form titled "Spring Security Custom Login Form". The form is centered on the page and contains the following elements:

- Username:** A text input field with the placeholder "Enter Username".
- Password:** A text input field with the placeholder "Enter Password".
- Login:** A green button.
- Remember me:** A checked checkbox.
- Cancel:** A red button.
- Forgot password?:** A blue link.

**Figure 14.2: Spring Security Custom Login Page**

Admin Login (Shown in Figure 14.3).

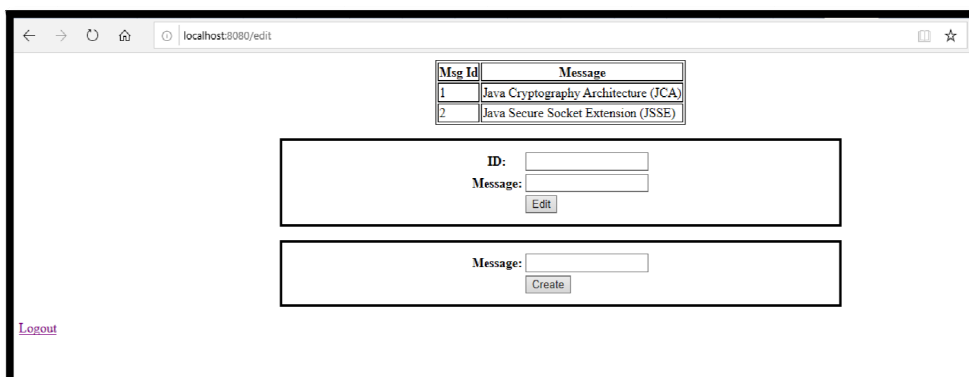


A screenshot of a web browser showing the admin user home screen. The address bar shows "localhost:8080/admin". The page content includes:

- A welcome message: "Dear admin@gmail.com, Welcome to Admin Page."
- Role information: "You are having roles as [ROLE\_ADMIN, ROLE\_EDITOR]"
- Two links: "Edit" and "Logout".

**Figure14.3: Admin User Home Screen**

On click on Edit button admin will see the below screen (Figure 14.4). Admin has only permission to create new messages as explained the feature of each user in the beginning of this section.



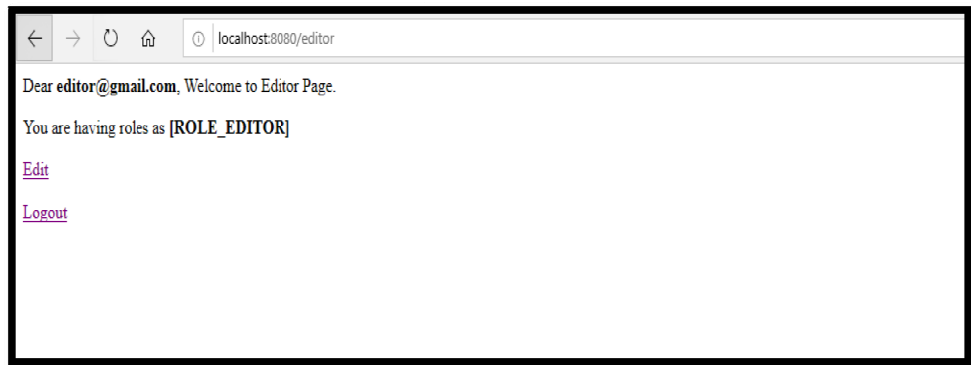
A screenshot of a web browser showing the "Edit and Create Screen For Admin User". The address bar shows "localhost:8080/edit". The page contains:

- A table with two columns: "Msg Id" and "Message".
 

| Msg Id | Message                              |
|--------|--------------------------------------|
| 1      | Java Cryptography Architecture (JCA) |
| 2      | Java Secure Socket Extension (JSSE)  |
- An "Edit" form with fields for "ID:" and "Message:" and an "Edit" button.
- A "Create" form with a "Message:" field and a "Create" button.
- A "Logout" link in the bottom left corner.

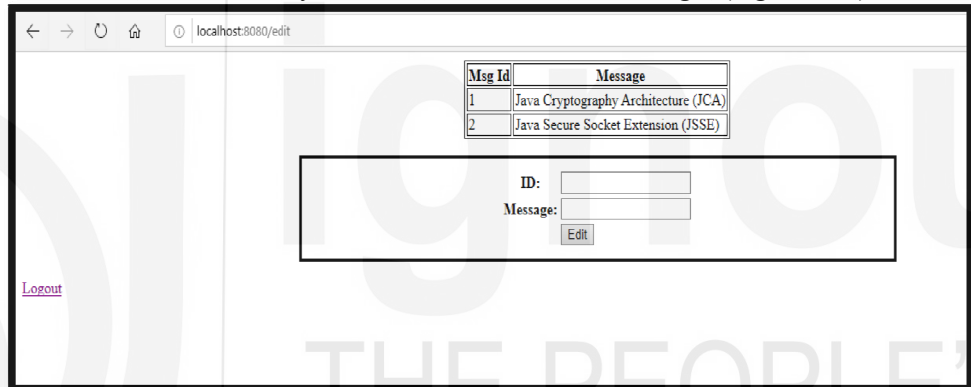
**Figure14.4: Edit and Create Screen For Admin User**

When Editor logs in, you will get the below screen (Figure 14.5).



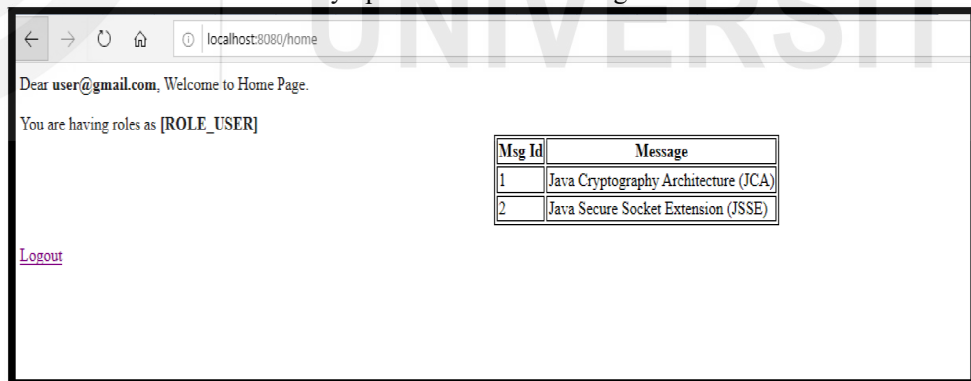
**Figure 14.5: Editor Role User Home Screen**

On click on Edit button, editor can only edit the already existing messages. Check edit.jsp file in which `<security:authorize access="hasRole('ADMIN')">` security tag has been used to allow only Admin user to create new messages (Figure 14.6).



**Figure14.6: Edit Screen from Editor Role User**

When a user logs in, he/she will get the below screen (Figure 14.7) as you can see that normal user does not have any option to edit the message.



**Figure14.7: General User Home Screen**

### 14.5.1 Unit Tests in Spring Boot

Production-ready code should have well-written Unit test cases as well as Integration Test. Good unit test cases should have code coverage of 80% or more. Writing good unit test cases is an art, and sometimes it's difficult to write good code coverage unit test cases. Spring provides an efficient way to write Unit Test cases as well Integration Test cases. In Unit 13, various annotations have been described for Integration Testing

with examples. `@SpringBootTest` is used for integration testing, and it is useful when we need to bootstrap the entire container. This section will describe the best practices to write unit test cases as well as available annotations will be described.

### **@SpringBootTest**

`@SpringBootTest`

```
class StudentEnrollmentUseCaseTest
```

```
{
```

```
    @Autowired
```

```
    private EnrollmentService enrollmentService;
```

```
    @Test
```

```
    void studentEnrollment() {
```

```
        Student student = new Student("Jack", "jack@mail.com");
```

```
        Student enrolledStudent = enrollmentService.enroll(student);
```

```
        assertThat(enrolledStudent.getEnrollmentNo(), isNotNull());
```

```
    }
```

```
}
```

A good test case takes only few milliseconds, but the above code might take 3-4 seconds. The above test case execution takes only a few milliseconds, and the rest of the 4 seconds is being taken by `@SpringBootTest` to set up the complete Spring Boot Application Context. In the above test case complete Spring Boot Application Context has been set up to autowire the `EnrollmentService` instance into the test case. Unit test cases set up time can be reduced a lot with the use of Mock instances and appropriate annotation instead of `@SpringBootTest`.

A well-structured web application consists of multiple layer such as Controllers, Services and Repository. Controllers are responsible to handle the client request forwarded by `dispatcherServlet`. The controller uses Service classes for business logic and Service classes uses Repository classes to interact with databases. Each layer can be tested independently with Unit test cases. The following section will describe the important annotations available in Spring to write the good unit test with a few milliseconds of execution time.

#### **14.5.1.1 Data Layer or Repository Layer testing with**

##### **@DataJpaTest**

An entity class named as `Student` which is having properties as `id` and `name`

```
@Entity
```

```
@Table(name = "student")
```

```
public class Student
```

```
{
```



```

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;

@Size(min = 3, max = 20)
private String name;

// constructors, getters and setters
}

```

### StudentRepository with Spring Data Jpa

```

@Repository
public interface StudentRepository extends JpaRepository<Student, Long>
{

    public Student findById(Long id);
    public List<Student>findByName(String name);

}

```

Persistence layer unit test cases code set up shown below

```

@DataJpaTest
public class UserRepositoryTest
{

    @Autowired
    private TestEntityManagerentityManager;

    @Autowired
    private UserRepositoryuserRepository;
    // test cases

}

```

@DataJpaTest focuses only on JPA components needed for testing the persistence layer. Instead of initializing the complete Spring Boot Application Context, as in the case of @SpringBootTest, initializes only the required configuration relevant to JPA tests. Thus, Unit test case set-up time is very less. @DataJpaTest does the following configurations.\

- ... setting Hibernate, Spring Data, and the DataSource
- ... configuring H2, an in-memory database
- ... turning on SQL logging
- ... performing an @EntityScan

A complete example of persistence layer unit test case using @DataJpaTest

```

@DataJpaTest
public class UserRepositoryTest
{

    @Autowired
    private TestEntityManagerentityManager;

    @Autowired
    private UserRepositoryuserRepository;

    @BeforeEach
    private void setUp()
    {
        User user = newUser();
        user.setEmail("testadmin@gmail.com");
        user.setName("Admin");
        user.setPassword("admin@123");
        entityManager.persist(user);
        entityManager.flush();
    }
}

```

```

@Test
public void findByEmailTest()
{
    String email = "testadmin@gmail.com";

    // Call userRepository to get user record
    Optional<User> found = userRepository.findByEmail(email);
    String getUserEmail = found.get().getEmail();

    assertThat(getUserEmail).isEqualTo(email);
}
}

```

### 14.5.1.2 Service Layer and Controller layer testing with @MockBean

In MVC project, Controllers are dependent on Services, and Services are dependent on Repositories. However, Controllers and Services should be testable without knowing the complete implementation of dependencies.

@MockBean can be used to mock the required dependency. Spring boot @MockBean annotation used to add mocks to a Spring ApplicationContext.

A complete example of Service layer unit test case with @MockBean

```

@ExtendWith(SpringExtension.class)
public class UserServiceTest
{
    @TestConfiguration
    static class UserServiceTestConfiguration
    {
        @Bean
        public UserService userService()
        {
            return new UserService();
        }
    }

    @MockBean
    private UserRepository userRepository;
    @Autowired
    private UserService userService;
    @BeforeEach
    public void setUp()
    {
        User user = new User();
        List<Role> roleList = new ArrayList<>();

        Role adminRole = new Role();
        adminRole.setName("Admin");
        Role editorRole = new Role();
        editorRole.setName("Editor");
        roleList.add(adminRole);
        roleList.add(editorRole);

        user.setEmail("testadmin@gmail.com");
        user.setName("Admin");
        user.setPassword("admin@123");
        user.setRoles(roleList);

        Mockito.when(userRepository.findByEmail(user.getEmail())).thenReturn(Optional.of(u
ser));
    }

    @Test
    public void whenValidName_thenUserShouldBeFoundTest()
    {
        String email = "testadmin@gmail.com";
        UserDetails found = userService.loadUserByUsername(email);
        assertThat(found.getUsername()).isEqualTo(email);
    }
}

```

All unit test cases can be executed by right click on the project -> Run As -> JUnit Test as shown below screenshot (Figure 14.8).

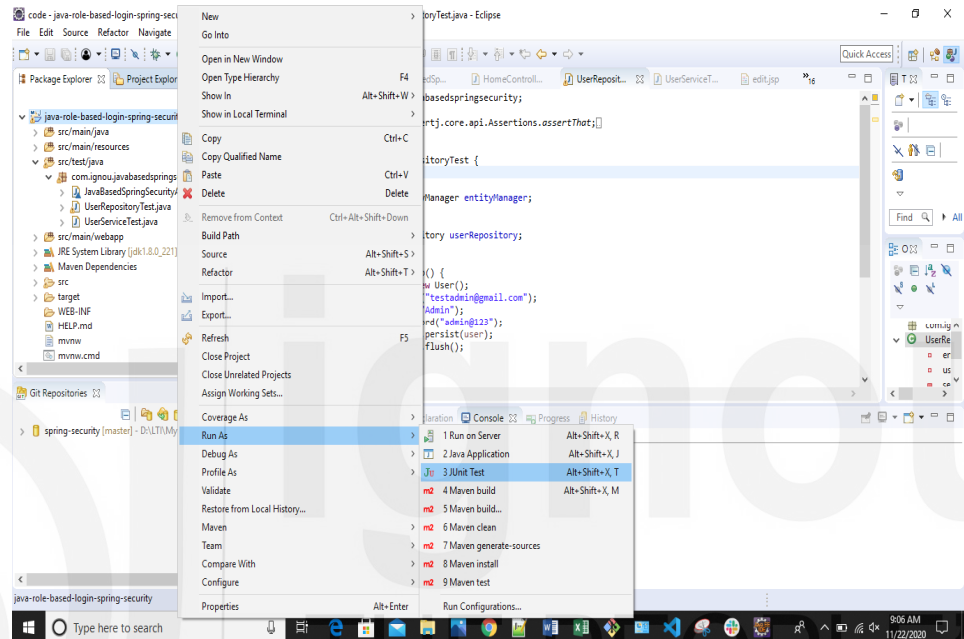


Figure 14.8: Unit Test Case Execution From Eclipse IDE

Successful execution of Unit Test case will output the following (Figure 14.9). As you can see all test cases are passed, which is shown by the green tick and green progress bar.

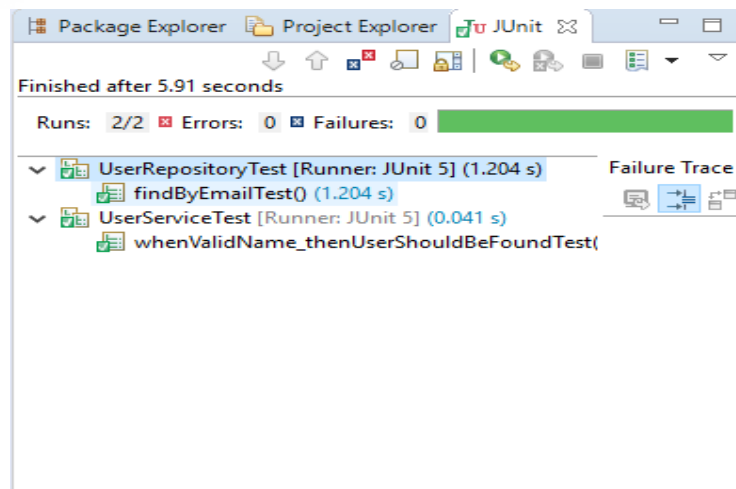


Figure 14.9: Unit Test Cases Execution Status

## 14.6 CROSS SITE REQUEST FORGERY (CSRF)

In Unit 12, section 12.3 CSRF concept and ways of its execution have been described. CSRF is a type of attack in which attackers trick a victim into making a request that utilizes their authentication or authorization. The victim's level of permission decides the impact of CSRF attack. Execution of CSRF attack consists of mainly two-part.

- ... Trick the victim into clicking a link or loading a web page
- ... Sending a crafted, legitimate-looking request from victim's browser to website

For a better understanding of CSRF, let us have a look at a concrete example of bank money transfer.

Suppose that bank's website provides a form to transfer money from the currently logged in user to another bank account.

For example, the HTTP request might look like:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&account=1234
```

Now consider that user authenticates to bank's website and then, without logging out, visits an evil website. The evil website contains an HTML page with the following form:

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden"
  name="amount"
  value="100.00"/>
<input type="hidden"
  name="account"
  value="evilsAccountNumber"/>
<input type="submit"
  value="Win Money!"/>
</form>
```

User likes to win money, and he clicks on the Win Money! Button. As user clicks on button, unintentionally he transferred \$100 to a malicious user. While evil website can not see your cookies, but the browser sent the cookie associated with bank along with the request.

The whole process can be automated using JavaScript, and onPage load script can be executed to perform CSRF. The most popular method to prevent Cross-site Request Forgery is to use a randomly generated token that is associated with a particular user and that is sent as a hidden value in every state-changing form in the web app. This token, called an anti-CSRF token (often abbreviated as CSRF token) or a synchronizer token. When a request is submitted, the server must look up the expected value for the parameter and compare it against the actual value in the request. If the values do not match, the request should fail.

### Spring Security CSRF Protection

Necessary steps to use Spring Security's CSRF protection are as follows-

- ... Use proper HTTP verb
- ... Configure CSRF protection

... Include the CSRF token

## Use Proper HTTP Verb

The use of proper HTTP verb plays a vital role to protect a website against CSRF attack. We need to be assured that the application is using PATCH, POST, PUT, and/or DELETE for anything that modifies state. This is not a limitation of Spring Security's support, but instead a general requirement for proper CSRF prevention.

## Configure CSRF protection

By default, CSRF protection is enabled in Spring java configuration. It can be disabled as below:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter
{
    @Override
    protected void configure(HttpSecurity http) throws Exception
    {
        http
            .csrf().disable();
    }
}
```

## Include the CSRF Token

The last step is to include the CSRF token all PATCH, POST, PUT, and DELETE methods. Csrfinput Tag is described into section 14.2 and used it in edit.jsp in Section 14.5. In edit.jsp we used two approach to include csrf token as follows-

```
... <input type="hidden" name="${_csrf.parameterName}"
value="${_csrf.token}" />
... <security:csrfInput/>
```

Following example will give you experimental and real feel of CSRF attack in the application executed in Section 14.5. Steps to perform CSRF attack simulation are as below-

**Step 1:** To simulate the evil website create a html file with below content. You can give it any name. Let us assume that filename is as **evil.html** and save this file to any location.

```
<html>
<body>
<form action="http://localhost:8080/edit" method="POST" id="attack">
    <input type="hidden" name="id" value="2">
    <input type="hidden" name="content" value="Hackers Msg">
    <input type="button"
onclick="document.getElementById('attack').submit()" value="Click me to Won
Prize"></input>
</form>
</body>
</html>
```

**Step 2:** Start the application tested into Section 14.5 and login as Admin or Editor user. Here consideration is that Admin user do login into the system (Figure 14.10).

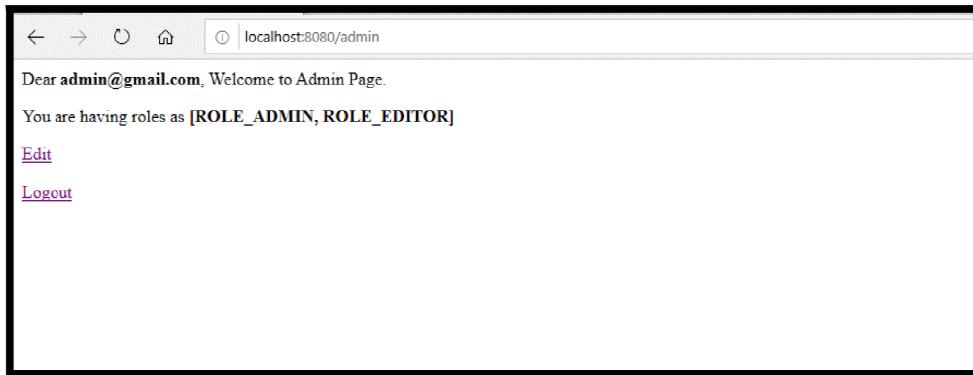


Figure 14.10: Admin Role User Home Screen

**Step 3:** For CSRF attack to place User must be logged in to the system. In step 2, Admin has logged in. Now open evil.html in the same browser in which admin has logged and click on the button. Once you click the button, the following screen (Figure 14.11) will appear with Hacker msg.

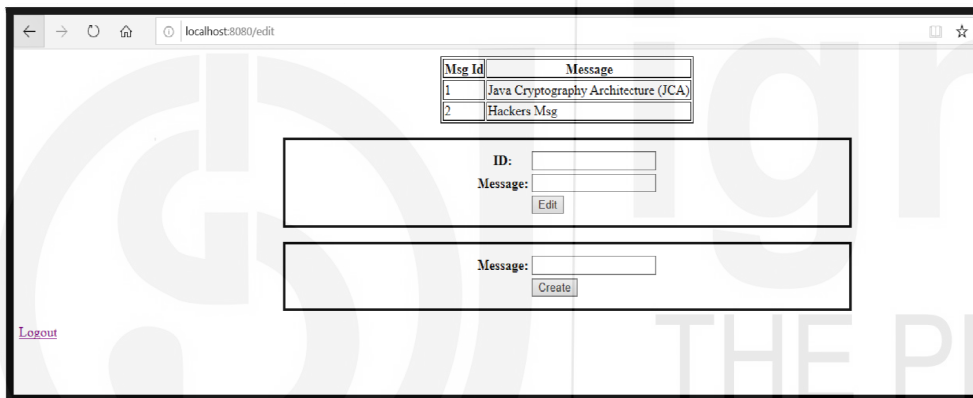


Figure 14.11: CSRF Attack

As you can see in above screen that malicious user has edited the msg. Close the evil.html. From admin login, click on edit link, and there also you will find the modified msg by malicious user.

Now we will make the required changes to protect the website against CSRF attacks. We have configured security in Security.java file. You can check that csrf is disabled there. You can just remove csrf.disable() and automatically csrf protection will be enabled. We have already included token in jsp so no need to make any changes in jsp. Perform all the steps again, and this time in step 3 you will get the below screen (Figure 14.12).

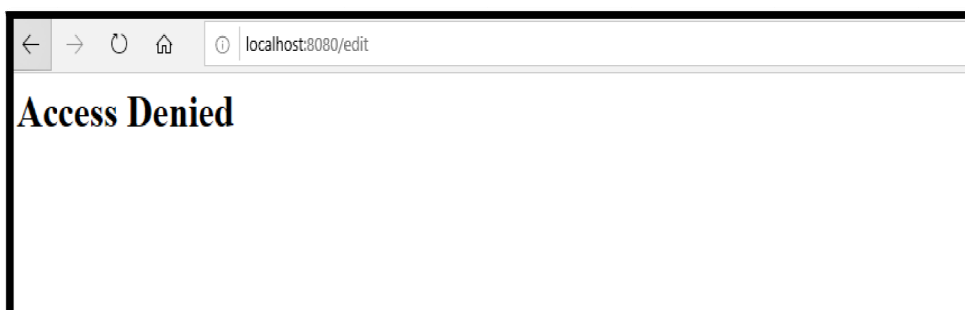


Figure 14.12: Website Protected Against CSRF Attack

As you can see now, the edit is being denied since we have enabled CSRF protection and the malicious user is unable to edit or create the msg.

You can try to create a new message as a malicious user by disabling csrf protection. Make the changes in evil.html.

### ☛ Check Your Progress 3

- 1) Explain @SpringBootTest. Why is it not suitable for unit test cases?

.....

.....

.....

.....

.....

- 2) Describe @MockBean with an example.

.....

.....

.....

.....

.....

- 3) What is Cross Site Request Forgery (CSRF)? What is the necessary condition for CSRF to take place?

.....

.....

.....

.....

.....

- 4) How do you make the Spring Application secured against CSRF attack? Discuss the necessary steps along with sample code to enable CSRF in Spring.

.....

.....

.....

.....

.....

---

## 14.7 SUMMARY

---

In the Unit, you learnt various core components of Spring Security such as SecurityContextHolder, SecurityContext, Authentication, Principal etc. Next you

learnt how to get logged in user details in controllers using Authentication and Principal and many more.

Gradually you learnt various Spring Security tags such as authorize, authentication, csrfInput etc. to secure the view layer jsp and display the user details on jsp using these tags.

Next you learnt about role-based login. In role-based login you learnt to redirect users on different URL based on assigned role of the User. You learnt about AuthenticationSuccessHandler and configuration using successHandler().

This Unit has described how to secure the service layer methods using various annotations such as @Secured, @RoleAllowed, @PreAuthorize, @PostAuthorize etc.

You executed a complete application in which you learnt how to implement the above concept programmatically. In the last of the unit, you learnt how to enable CSRF protection in Spring application. You got the real feel of CSRF attack with executed example, and this will enable you to understand CSRF attack easily.

## 14.8 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

### Check Your Progress 1

- 1) All Authentication implementations in Spring is having a list of GrantedAuthority objects. These represent the authorities that have been granted to the principal. AuthenticationManager inserts all the GrantedAuthority objects into the Authentication Object. Hierarchical Structure is as below (Figure 14.13).



Figure 14.13: Authorities in Authentication object

- 2) UserName using HttpServletRequest can be accessed as follows-

```

@Controller
public class UserSecurityController
{
    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String loggeInUserName(HttpServletRequest request)
    {
        Principal principal = request.getUserPrincipal();
        return principal.getName();
    }
}
  
```

- 3) Authentication tag can be used get logged in user details and authorities on jsp as below-



```
... <security:authentication property="principal.username" />
... <security:authentication property="principal.authorities" />
```

For details refer authentication tag in the Section 14.2

## ☛ Check Your Progress 2

- 1) **AuthenticationSuccessHandler** is an Interface, and Spring provides **SimpleUrlAuthenticationSuccessHandler**, which implement this interface. To create the custom success handler we can extend the **SimpleUrlAuthenticationSuccessHandler** class and can override the **handle()** method to provide the required logic to for redirect URL. For role based login, you need to configure **successHandler(AuthenticationSuccessHandler successHandler)** instead of **defaultSuccessUrl(String s)**. Configuration for custom success handler is as below:

```
@Override
protected void configure(final HttpSecurity http) throws Exception
{
    http
    .authorizeRequests()
        // endpoints
    .formLogin()
    .loginPage("/login.html")
    .loginProcessingUrl("/login")
    .successHandler(customSuccessHandler())
        // other configuration
}
```

- 2) For RBAC refer section 14.4. Configuration to restrict the URL access based on role is as below-

```
protected void configure(HttpSecurity http) throws Exception
{
    http.csrf().disable()
    .authorizeRequests()
    .antMatchers("/customlogin").permitAll()
    .antMatchers("/", "/home").hasRole("USER")
    .antMatchers("/editor/**").hasRole("EDITOR")
    .antMatchers("/admin/**").hasRole("ADMIN")
    .antMatchers("/edit/**").hasAnyRole("ADMIN", "EDITOR")
    .antMatchers("/create/**").hasAnyRole("ADMIN")
    .anyRequest().authenticated()
    .and()
    .formLogin()
    .loginPage("/customlogin")
    .loginProcessingUrl("/signin")
    .successHandler(customSuccessHandler)
    .failureUrl("/customlogin?error=true")
    .and()
    .logout()
    .and().exceptionHandling().accessDeniedPage("/accessdenied");
}
```

In above configuration

URL pattern “/” and “/home” is accessible only to user which is having role as **ROLE\_USER**.

URL pattern “/editor/\*\*” is accessible only to user having role as **ROLE\_EDITOR**

URL pattern “/admin/\*\*” is accessible only to user having role as **ROLE\_ADMIN**

URL pattern “/edit/\*\*” is accessible to user having role as ROLE\_ADMIN or ROLE\_EDITOR  
 URL pattern “/create/\*\*” is permissible only to user having role as ROLE\_ADMIN

- 3) @Secured and @RolesAllowed both annotations provide method-level security into Spring Beans. @Secured is Spring Security annotation from version 2.0 onwards Spring Security. But @RolesAllowed is JSR 250 annotation. Spring Security provides the support for JSR 250 annotation as well for method level security. @RolesAllowed provides role-based security only.
- 4) @PreAuthorize and @PostAuthorize allow to use SpEL. These annotations support expression attributes to allow pre and post-invocation authorization checks.
- 5) If you wanted to do something like access the method only if the user has Role1 and Role2 the you would have to use @PreAuthorize @PreAuthorize("hasRole('ROLE\_role1') and hasRole('ROLE\_role2')") Using @Secured({"role1", "role2"}) is treated as an OR
- 6) For details check the Section 14.2. Example is as follows-

```
<security:authorize access="!isAuthenticated()">
  Login
</security:authorize>
<security:authorize access="isAuthenticated()">
  Logout
</security:authorize>
```

Further we can also display certain information based on user role as-

```
<security:authorize access="hasRole('ADMIN')">
  Delete Users
</security:authorize>
```

### ☛ Check Your Progress 3

- 1) @SpringBootTest is used for integration testing and it is useful when we need to bootstrap the entire container. The @SpringBootTest annotation tells Spring Boot to look for a main configuration class (one with @SpringBootApplication, for instance) and use that to start a Spring application context. @SpringBootTest by default starts searching, a class annotated with @SpringBootConfiguration, in the current package of the test class and then searches upwards through the package structure. It reads the configuration from @SpringBootConfiguration to create an application context. This class is usually our main application class since the @SpringBootApplication annotation includes the @SpringBootConfiguration annotation. It then creates an application context very similar to the one that would be started in a production environment.
- 2) In the MVC project, Controllers are dependent on Services, and Services are dependent on Repositories. However, Controllers and Services should be testable without knowing the complete implementation of dependencies. @MockBean can be used to mock the required dependency. Spring boot @MockBean annotation used to add mocks to a Spring ApplicationContext.  
 ... @MockBean allows to mock a class or an interface.

- ... @MockBean can be used on fields in either @Configuration classes or test classes that are @RunWith the SpringRunner as well as a class level annotation.

@MockBean at field level

```
@RunWith(SpringRunner.class)
public class LoginControllerTest
{

    @MockBean
    private LoginService loginService;

}
```

@MockBean at class level

```
@RunWith(SpringRunner.class)
@MockBean(LoginService.class)
public class LoginControllerTest
{

    @Autowired
    private LoginService loginService;

}
```

- ... Mocks can be registered by type or by bean name.
- ... Any existing single bean of the same type defined in the context will be replaced by the mock. If no existing bean is defined a new one will be added.

- 3) For CSRF details, refer Unit 12 section 12.3 and Unit 14 section 14.6. Necessary condition for CSRF attack to succeed is that the victim must be logged in and attacker is able to trick the victim into clicking a link or loading a web page is done through social engineering or using a malicious link.
- 4) CSRF is a type of attack in which attackers trick a victim into making a request that utilizes their authentication or authorization. The victim's level of permission decides the impact of CSRF attack. Execution of CSRF attack consists of mainly two-part.
  - o Trick the victim into clicking a link or loading a web page
  - o Sending a crafted, legitimate-looking request from victim's browser to the website

Necessary steps to use Spring Security's CSRF protection are as follows-

- ... Use proper HTTP verb
- ... Configure CSRF protection
- ... Include the CSRF token

---

## 14.9 REFERENCES/FURTHER READING

---

- ... Craig Walls, "Spring Boot in action" Manning Publications, 2016.  
(<https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf>)

- ... Christian Bauer, Gavin King, and Gary Gregory, “Java Persistence with Hibernate”,Manning Publications, 2015.
- ... Ethan Marcotte, “Responsive Web Design”, Jeffrey Zeldman Publication, 2011([http://nadin.miem.edu.ru/images\\_2015/responsive-web-design-2nd-edition.pdf](http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf))
- ... Tomcy John, “Hands-On Spring Security 5 for Reactive Applications”,Packt Publishing,2018
- ... [https://www.baeldung.com/spring\\_redirect\\_after\\_login](https://www.baeldung.com/spring_redirect_after_login)
- ... <https://digitalguardian.com/blog/what-role-based-access-control-rbac-examples-benefits-and-more>
- ... <https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/csrf.html>
- ... <https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html>
- ... <https://www.baeldung.com/spring-security-csrf>
- ... <https://www.baeldung.com/java-spring-mockito-mock-mockbean>

