LAB- Spring Security Basics

Step 1 -- Securing an App using Spring boot with Basic and Form Based Authentication

In this step you will be working on **01-security-basics-start** project.

Open HelloController.java and observe it.

Open WebSecurityConfiguration.java and observe that it is empty now.

This application is not yet secured.

Now start this application and make a request to http://localhost:8080/hello?name=Siva

Observe that you get the response as expected.

Now we want to secure this application.

Open pom.xml and complete TODO-1 in it.

Now restart the application. You should observe that there is a password logged into the console.

Now make a request to http://localhost:8080/hello?name=Siva

You should be redirected to login page at http://localhost:8080/login

Give username as **user** and password which is printed on the console as credentials and login

You should be shown the the response from the controller.

We don't want the default username as user. We want to configure our own username and password. Open application.yml and configure the below:

```
spring:
  security:
    user:
      name: siva
      password: secret
```

Now restart the application, restart the browser and make a request http://localhost:8080/hello?name=Siva

You should be able to login using siva/secret

We have used form based authentication till now which is the default.
Why is form based authentication the default?

How did my application get security just by adding security Starter Dependency?

Open **SecurityAutoConfiguration.java** and observe that it is importing **SpringBootWebSecurityConfiguration.java**

It looks like below where it is defining a Configuration file which extends **WebSecurityConfigurerAdapter**

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(WebSecurityConfigurerAdapter.class)
@ConditionalOnMissingBean(WebSecurityConfigurerAdapter.class)
@ConditionalOnWebApplication(type = Type.SERVLET)
public class SpringBootWebSecurityConfiguration {

    @Configuration(proxyBeanMethods = false)
    @Order(SecurityProperties.BASIC_AUTH_ORDER)
    static class DefaultConfigurerAdapter extends WebSecurityConfigurerAdapter {

    }

}
```

Open **WebSecurityConfigurerAdapter.java** and observe the below method where formlogin is configured first

```
protected void configure(HttpSecurity http) throws Exception {
            logger.debug("Using default configure(HttpSecurity). If subclassed this
will potentially override subclass configure(HttpSecurity).");

            http
                  .authorizeRequests()
                        .anyRequest().authenticated()
                        .and()
                  .formLogin().and()
                  .httpBasic();
      }
```

Above method is configuring both form based and basic Authentication.

Now we want to enable Basic Authentication.

If we define our own WebsecurityConfigurerAdapter, boot will not configure it.

Open WebSecurityConfiguration.java under com.way2learnonline package and complete the TODO-2 and TODO-3

Now restart the application and give a request to http://localhost:8080/hello?name=Siva

Your browser should ask for username/password in a pop up window.

Step 2 -- Securing an App using Spring boot with Digest Authentication

Now We want to configure 2 SecurityFilterChains. One for Digest and other for Basic Authentication.

Open AdminSecurityConfiguration.java and complete TODO-4 to TODO-10. Make sure that you complete TODOs in order.

Observe that we have configured Order as 1 for AdminSecurityConfiguration and 2 for WebSecurityConfiguration. Can you tell why?

Now run the application in debug mode.

Open DelegatingFilterProxy.java and keep a break point at keep a break point at the last line of `doFilter() method.`

Now give a request to http://localhost:8080/admin/hello?name=Siva

Observe that there will be 2 Filter chains as shown below:

this	DelegatingFilterProxyRegistrationBean$1 (id=119)
beanName	null
contextAttribute	null
delegate	FilterChainProxy (id=158)
beanName	"springSecurityFilterChain" (id=225)
environment	StandardServletEnvironment (id=161)
filterChains	ArrayList<E> (id=226)
[0]	DefaultSecurityFilterChain (id=244)
[1]	DefaultSecurityFilterChain (id=245)

Observe that First filter chain I mapped to /admin/**

Observe that there is DigestAuthenticationFilter in the first chain.

delegate	FilterChainProxy (id=158)
beanName	"springSecurityFilterChain" (id=225)
environment	StandardServletEnvironment (id=161)
filterChains	ArrayList<E> (id=226)
[0]	DefaultSecurityFilterChain (id=244)
filters	ArrayList<E> (id=252)
[0]	WebAsyncManagerIntegrationFilter (id=259)
[1]	SecurityContextPersistenceFilter (id=260)
[2]	CsrfFilter (id=261)
[3]	LogoutFilter (id=262)
[4]	DigestAuthenticationFilter (id=263)
[5]	RequestCacheAwareFilter (id=264)
[6]	SecurityContextHolderAwareRequestFilter (id=265)
[7]	AnonymousAuthenticationFilter (id=266)
[8]	SessionManagementFilter (id=267)
[9]	ExceptionTranslationFilter (id=268)
[10]	FilterSecurityInterceptor (id=269)
requestMatcher	AntPathRequestMatcher (id=253)

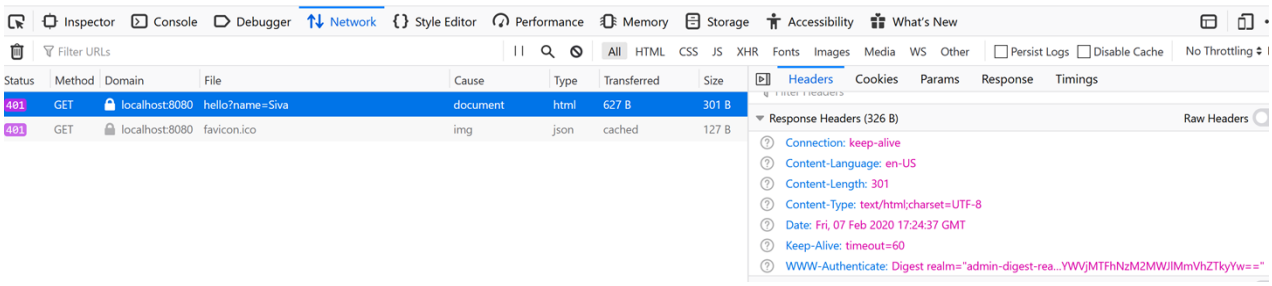
Given admin credentials as admin/adminsecret . You should be able to see the response successfully.

Restart your browser now.

Right click on Firefox and open "Inspect Element" and click on network.

Now give a request to http://localhost:8080/admin/hello?name=Siva
It should ask for username/password. Click on cancel and observe the network tab

You should observe WWW-Authenticate header as shown below in the response headers



This proves that the server is requesting for digest authentication

Step 3– Configuring Credentials in database

we want to use h2 as embedded database.

Open pom.xml and observe that we have already added h2 as a dependency using below xml

```
<dependency>
	<groupId>com.h2database</groupId>
	<artifactId>h2</artifactId>

</dependency>
```

To enable h2 console, add the below in application.yml

```
spring:
  h2:
    console:
      path: /h2-console
      enabled: true
```

Open WebSecurityConfiguration.java and modify complete TODO-11
Open data.sql and schema.sql in src/main/resources and obsere the tables DDL and inserts
These sql files will be automatically executed by spring boot on embedded database.

Why is {noop} used in passwords?

That's it. We have now configured our security configuration to use jdbc authentication.

Now give a request to http://localhost:8080/hello?name=Siva and use the credentials from data.sql

You should be successfully logged in using credentials from database.

Step 4– Configuring Custom Form login

We want to enable Form login for any url other than /admin/**

Open WebSecurityConfiguration.java and modify the configure method as shown below:

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

		http
```

```
                .authorizeRequests()
                .antMatchers("/register","/login","/h2-console/**","/mylogin").permitAll()
                        .anyRequest().authenticated()
                        .and()
                        .formLogin().loginPage("/mylogin").and().csrf().disable();

        }
```

Observe that we have disabled csrf.

Open MvcConfiguration.java and fully uncomment it. Observe how we have configured viewcontroller for /mylogin

Open mylogin.html in src/main/resources/templates and observe it.

Restart the application now .
Give a request to http://localhost:8080/hello?name=Siva. You should be redirected to /mylogin and you should see that the custom login page is dispayed

But css is not deisplayed properly.
We don't want security for static files like css

So, Open WebsecurityConfiguration.java and override the configure method as shown below:

```
@Override
public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/css/**","/webjars/**");
}
```

Restart the application now .
Give a request to http://localhost:8080/hello?name=Siva. You should be redirected to /mylogin and you should see that the custom login page is dispayed properly with css.

You should be able to login successfully.

Restart the application now .
Give a request to http://localhost:8080/hello?name=Siva. You should be redirected to /mylogin and you should see that the custom login page is dispayed

Step 4– Enabling Https for Spring Boot Application

Create your own keystore using the below command

keytool -genkey -alias tomcat -storetype PKCS12 -keyalg RSA -keysize 2048 -keystore mykeystore.p12

Create Give the keystore password as tomcat

Copy the created mystore.p12 into src/main/resources.

Configure below in application.yml

```yaml
server:
  port: 8443
  ssl:
    key-store-password: tomcat
    key-store: classpath:mykeystore.p12
    key-store-type: PKCS12
    key-alias: tomcat
```

Now give a request to https://localhost:8443/hello?name=Siva
You should be able to login and see the page

What if some one gives a request to http://localhost:8080/hello?name=Siva
We want the redirect to https://localhost:8443/hello?name=Siva

Open Application.java and complete TODO-13

Now restart the application and give a request to http://localhost:8080/hello?name=Siva

You should be redirected to login page which uses https and port 8443

Step 5– Externalizing secrets to Vault

In this step, we will be storing our sensitive information in vault.

We will be using HashiCorp's Vault.

It can be downloaded from https://www.vaultproject.io/downloads/

But I have given you the downloaded zip file to you in micro-lab-docs folder. Extract it.

Copy **vault.hcl** given to you under the extracted folder.
Open vault.hcl and observe the configuration . It is configured to listen at 8200 and tls is disabled

Use the below command to start vault

Vault server –config ./vault.hcl

Our Vault server now is running, but since this is its first run, we need to initialize it.

```
Set VAULT_ADDR=http://localhost:8200
vault operator init
```

After issuing the above command, we should see a message like this:

```
Unseal Key 1: NvEnMTc7idNhU1mE8wkDaISR3TvzF9ML/Q9vUtpwTvgd
Unseal Key 2: q7/Twre6t71esCemSNqEVM1TEu4mBOz94V1x6yWudQ36
Unseal Key 3: ALbT2By9EtQplKxRmzuxOABhrsfvO1LS20Gvq0p7XOx1
Unseal Key 4: 3PeI5BNPtqOXrJBiMNNf5AVp+FKR1wCuUjqB3TWDrCBL
Unseal Key 5: LADHv6EYMH+CL2xlHlMmLNSZ3FX6Fw2G6kSMUF1Wd02K

Initial Root Token: s.SIcC2NZp4N4lw1Sgm7t9PsJF
```

The five first lines are the master key shares that we will later use to unseal Vault's storage. **Please note that Vault only displays the master key shares will during initialization – and never more. Take note and store them safely or we'll lose access to our secrets upon server restart!**

Now execute the below command:

```
Set VAULT_TOKEN= <root token value>
```

Let's see our server status now that we have initialized it, with the following command:

Execute the following command:

vault status

you should see as shown below:

```
D:\micro-2020\micro-lab-docs\vault_1.3.2_windows_amd64>vault status
Key                Value
---                -----
Seal Type          shamir
Initialized        true
Sealed             true
Total Shares       5
Threshold          3
Unseal Progress    0/3
Unseal Nonce       n/a
Version            1.3.2
HA Enabled         false
```

Observe that the vault is sealed

Observe the unseal progress: “0/3” means that Vault needs three shares, but got none so far. Let's move ahead and provide it with our shares.

We now unseal Vault so we can start using its secret services. We need to provide any three of the five key shares in order to complete the unseal process:

```
vault operator unseal <key share 1 value>
vault operator unseal <key share 2 value>
vault operator unseal <key share 3 value>
```

After executing all the 3 commands, value should be unsealed

Now You need to enable the KV secrets engine at path **secret**

Execute the following commands

vault secrets enable -path=secret/ kv

Now put a key value pair into vault using below command:

```
vault kv put secret/myapp keystorepassword=tomcat
```

Till now, we have started valut and stored `keystorepassword=tomcat` in vault

Now we want out spring boot app to read secrets from valut.

Open pom.xml and complete TODO-14

Now create a file with name bootstrap.yml under src/main/resources and configure it will following:

```
spring:
  application:
    name: myapp
  cloud:
    vault:
      host: localhost
      port: 8200
      scheme: http
     # authentication: token
      token: s.yZPi3ogu1BI585ZHw9Dqh2EZ
```

Now Replace the key token in bootstrap.yml with the root token which you got when you initialized vault

In application.yml replace the value of server.ssl.key-store-password with `${keystorepassword}`

Now start your application and give a request to http://localhost:8080/hello?name=siva

Everything should work fine and you should be redirected to login page which uses https

This means that the keystore password is retrieved from the vault successfully.

CONGRATULATIONS YOU KNOW HOW TO SECURE YOUR APPLICATION USING SPRING BOOT