Spring Security Lab 4

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A utility to extract zip files

Make sure you have a valid Gmail account and a GitHub account (https://github.com/signup)

2 OAuth2 / OIDC background

In the traditional approach to authentication, users interacting with unique web apps or services (for e.g. Facebook, Reddit, Twitter, GitHub, Google, etc) will need to register a user account (typically involving a username (or email) / password combination) to identify themselves to that particular service. The need to create multiple accounts each with its individual username/password combination gives rise to a variety of security issues:

a) Users will pick simple passwords in order to be able to simplify the memorization of a large number of different passwords. This makes the passwords easier to crack should there be a compromise of the password list on the hosting site and inappropriate password hashing technology is applied.

b) User may instead elect to create a single, complex password and reuse the same password on multiple accounts. Although this is better than the first approach, should this single password be compromised, all the other accounts of that user on other services are automatically compromised as well.

In some situations (for e.g. banking or payment gateway applications), it is ideal and necessary that users create a separate, unique account for authenticating themselves to the application as this account will be used to authorize financial transactions as well as access sensitive financial information.

In most other cases, apps simply need to verify that a user attempting to access the resources from the service is valid in some way. For the app to handle account registration and verification on its own, it would need to provide the following functionality:

a) Provide a registration / account creation page for the user to enter relevant details (username, email, password, DOB, etc) to create a valid user account
b) Provide a way to validate the user registering (for e.g. sending a code or OTP to the user's registered email or phone)
c) Store the username/password and other user-related info securely with appropriate password hashing algorithms applied
d) Provide a safe and valid way for the user to reset or change password (a very common occurrence)
e) Perform password validation correctly every time the user attempts to login into the system to access sensitive resources
f) Any other related account maintenance / security activities, depending on the nature of the app

Such functionality can incur significant overhead on system development and resource consumption for the app provider, particularly when they are dealing with a large number of users and the functionality described needs to be performed in a highly scalable and secure manner.

In view of this, Single Sign On (SSO) has become popular in recent years where users can now logon (sign on) to multiple, unique different services using the credentials from a single service. In effect, app / service providers now delegate or outsource the activity of user registration / authentication to a 3rd party that is capable of performing all these activities in a secure manner.
OAuth2 (and in particular OIDC, the authentication layer that rides on top of OAuth2) has become very popular in recent years for supporting SSO.

Spring Security 5 is capable of supporting sign in with virtually any service provider that conforms ot the OAuth2 / OIDC protocol by simply providing the service details in configuration. Out of the box, Spring Security 5 offers baseline configuration for Facebook, Google, GitHub, and Okta: which makes it extremely easy to configure them.

3 Setting up OIDC workflow with Google

Set up an OAuth Client on Google API console by following the instructions here:

https://developers.google.com/identity/protocols/oauth2/openid-connect

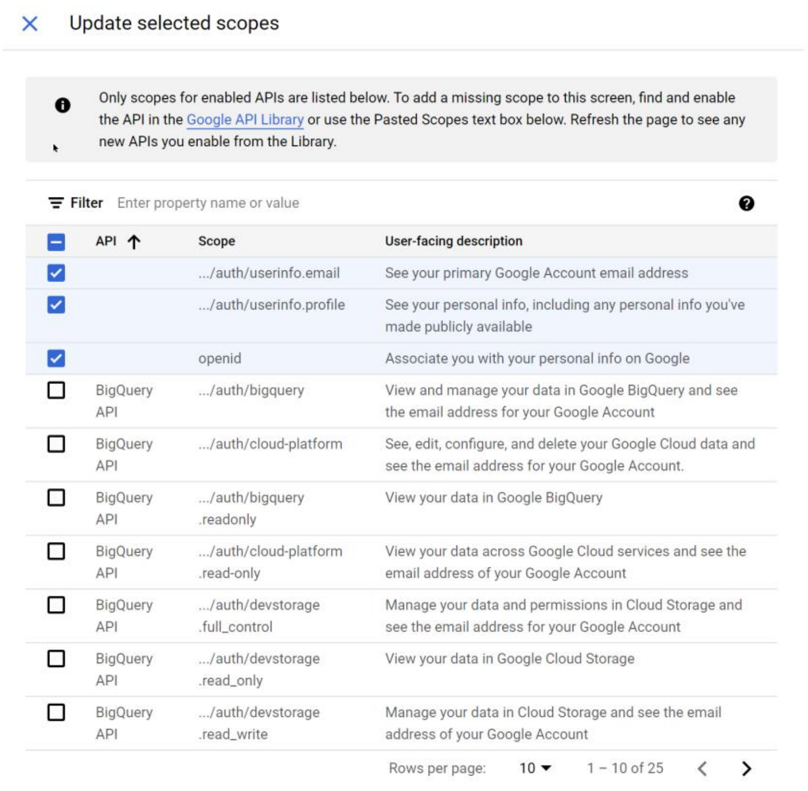
First, create a new project for this demo with any particular name
https://developers.google.com/workspace/guides/create-project

Then, customize the OAuth Consent screen.
Select an External User Type.
Enter a suitable App name and suitable values for the other fields. For the app domain entries, you can just use any valid URL, for e.g.:
Application home page: https://www.paynet.my/
Application privacy link: https://paynet.my/media.html
Application terms of service: https://paynet.my/references.html

For authorized domains, use the domain that you picked in the app domain entries, for e.g.
`paynet.my`

Click Save and Continue

At the Scopes page, you will specify the permissions that users who want to authenticate to your app will provide to your app. Click Add or Remove Scopes and select the first 3 scopes, then click Update.


Update selected scopes

Only scopes for enabled APIs are listed below. To add a missing scope to this screen, find and enable the API in the Google API Library or use the Pasted Scopes text box below. Refresh the page to see any new APIs you enable from the Library.

Filter Enter property name or value

	API ↑	Scope	User-facing description
✓		.../auth/userinfo.email	See your primary Google Account email address
✓		.../auth/userinfo.profile	See your personal info, including any personal info you've made publicly available
✓		openid	Associate you with your personal info on Google
☐	BigQuery API	.../auth/bigquery	View and manage your data in Google BigQuery and see the email address for your Google Account
☐	BigQuery API	.../auth/cloud-platform	See, edit, configure, and delete your Google Cloud data and see the email address for your Google Account.
☐	BigQuery API	.../auth/bigquery .readonly	View your data in Google BigQuery
☐	BigQuery API	.../auth/cloud-platform .read-only	View your data across Google Cloud services and see the email address of your Google Account
☐	BigQuery API	.../auth/devstorage .full_control	Manage your data and permissions in Cloud Storage and see the email address for your Google Account
☐	BigQuery API	.../auth/devstorage .read_only	View your data in Google Cloud Storage
☐	BigQuery API	.../auth/devstorage .read_write	Manage your data in Cloud Storage and see the email address of your Google Account

Rows per page: 10 1 – 10 of 25

Then click Save and Continue.

At the Test Users page, click Add Users and add in any Gmail address that you have wish you intend to use to test this app. Click Add, then click Save and Continue. Finally click Back to Dashboard.

In the left hand pane, select Credentials to bring up the Credentials page then select Create Credentials. Select OAuth client ID.

For application type, select Web Application.
For the Name, enter any name you wish.
Click on Add URI in the section for Authorized redirect URIs
Enter this value
http://localhost:8080/login/oauth2/code/google
then click Create

This is the authorization callback URL on your app, whereby the final call to return the authorization code as part of the OAuth2 / OIDC flow is performed. The value provided is the default configured value in Spring Security, so you should not map this path for any other purpose in your app.

You should now see a box with the Client ID and Client Secret. Copy down both these values which you will later use in the configuration of your Spring Boot app. Then click Ok. You can later click on the name of your Web application in the OAuth 2.0 Client IDs list if you wish to view these values again.

4 Running through the Google OIDC workflow

The main folder for this lab is `OIDC-Flow-Demo`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `OIDCFlowDemo`
Group: `com.workshop.security`
Artifact: `OIDCFlowDemo`
Version: `0.0.1-SNAPSHOT`
Description: `Demo Google OIDC workflow`
Package: `com.workshop.security`

Add the following dependencies:
```
Web -> Spring Web
Template Engines -> Thymeleaf
Developer Tools -> Lombok
Developer Tools -> Spring Boot DevTools
Security -> OAuth2 Client
```

Adding in the OAuth2 Client dependency automatically adds in all the relevant Spring Security dependencies, so we no longer need to explicitly specify this.

Add in this additional dependency in the project POM to allow us to add in Spring Security dialect specific tags in the Thymeleaf template. Remember to do a Maven -> Update Project.

```
<dependency>
      <groupId>org.thymeleaf.extras</groupId>
      <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
```

In `src/main/resources/templates`, place these files:

```
add.html
delete.html
home.html
update.html
view.html
```

In the package `com.workshop.security` in `src/main/java`, place these files:

```
WebMvcConfiguration
```

Here, we have created a skeleton structure for simple employee management system accessed via a Thymeleaf app. The configuration file `WebMvcConfiguration` provides a simple direct mapping for the URL paths to the view names, which are Thymeleaf HTML templates.

Start up the app in the usual manner from the dashboard. Note down the random generated security password.

Navigate to http://localhost:8080/ and you will be presented with the standard Spring autogenerated login page. Login in with the username of `user` and the random password that you noted down earlier. Navigate around the various blank pages. Again, notice that if you remove the single cookie that is currently used to track this session, you will be redirected again to the autogenerated login page the next time you attempt to navigate to any page.

We then need to add the client credentials (client-id and client-secret) that we obtained previously to `application.properties`. The relevant Spring Security properties are prefixed with `spring.security.oauth2.client.registration` followed by the `registrationid`, then the name of the client property. Both `client-id` and `client-secret` are needed at the minimum for a basic OAuth flow. The various OAuth / OIDC providers that are supported out-of-the-box by Spring Security are Facebook, Google, GitHub and Okta (`google / github / facebook / okta`) . As Spring Security framework uses the RestTemplate library under the hood to communicate with Google's authorization servers, we set the logging for this library to debug to allow us to trace the flow of messages in the workflow.

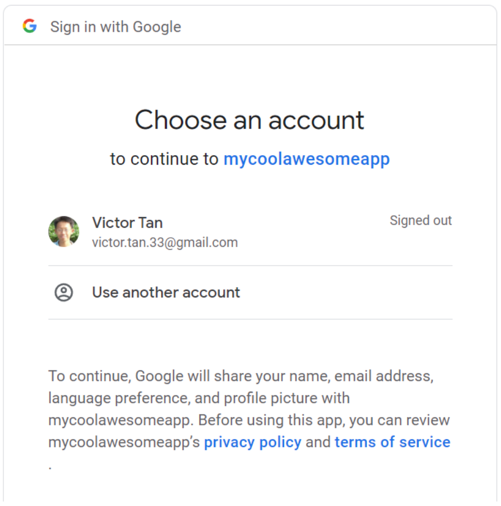
```
spring.security.oauth2.client.registration.google.client-id=your-id
spring.security.oauth2.client.registration.google.client-secret=your-secret
logging.level.org.springframework.web.client.RestTemplate=DEBUG
```

Adding these properties for at least one client will enable the OAuth2ClientAutoConfiguration class which sets up all the necessary beans. You can configure one or more client security properties if your app is going to provide the user multiple options for logging in via an OAuth2 / OIDC workflow.

With no initial custom security configuration, then by default, authentication is required for all requests to the app. The inclusion of the OAuth2 Client starter dependency means Spring that any incoming unauthenticated request for any resource will kickstart the entire OAuth2 / OIDC workflow where the user will be redirected to the OAuth2 login page of the specific configured provider. Once the user has logged in successfully and authorized the client (the web app) to access specific resources from his account, the browser will be redirected back to the requested resource.

When you are done, restart the app.
Open the Network tab in Chrome DevTools and check the Preserve Log checkbox in order to preserve tracking across page redirects.
Clear the browser cache and remove the single tracking cookie, then make a request again to http://localhost:8080/

If you had previously signed in to your Gmail account on the browser that you are performing this test on, it is likely that the long term cookies that authenticate your access to the Google server are being submitted as part of the OIDC authentication workflow from your browser. In that case you will be presented with a screen similar to the following (which already indicates that you are recognized by the Google server):


Sign in with Google

Choose an account

to continue to mycoolawesomeapp

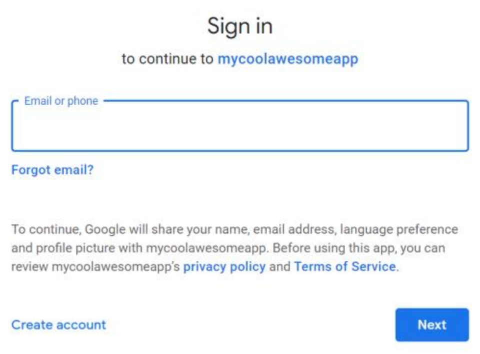
Victor Tan
victor.tan.33@gmail.com
Signed out

Use another account

To continue, Google will share your name, email address, language preference, and profile picture with mycoolawesomeapp. Before using this app, you can review mycoolawesomeapp's privacy policy and terms of service
.

In that case, clear all the cookies from all Google domains (including youtube) from the cookie dialog box accessible from the address bar.

Then make a request again to http://localhost:8080/
This time you should see a screen similar to the following:


Sign in

to continue to mycoolawesomeapp

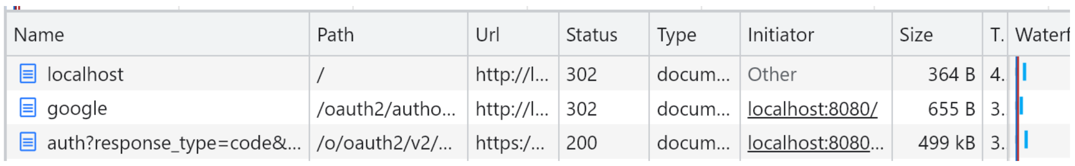
Email or phone

Forgot email?

To continue, Google will share your name, email address, language preference and profile picture with mycoolawesomeapp. Before using this app, you can review mycoolawesomeapp's privacy policy and Terms of Service.

Create account
Next

If you click on the name of the app, as well as the privacy policy and Terms of Service, you will obtain info or be navigated to the links that you had configured earlier in the Google API console.
You should also be able to trace the interaction between the browser, your app and Google's authorization server in the Network tab.



Name	Path	Url	Status	Type	Initiator	Size	T.	Waterf
localhost	/	http://l...	302	docum...	Other	364 B	4.	
google	/oauth2/autho...	http://l...	302	docum...	localhost:8080/	655 B	3.	
auth?response_type=code&...	/o/oauth2/v2/...	https:/...	200	docum...	localhost:8080...	499 kB	3.	

Detailed outline of the authorization flow:

1. For the initial attempt to access the application root path (localhost:8080), it returns a 302 redirect to http://localhost:8080/oauth2/authorization/google as well as setting a new JSESSIONID cookie to begin session tracking. This happens since there is no security configuration, which means by default, access to any resource requires authentication. For a normal Spring Security, this would normally result in a redirect to the default login path (`/login`) which returns the autogenerated login page. However, since we are using Spring OAuth2 Client here, the redirect is to http://localhost:8080/oauth2/authorization/google instead. This is the default path to which a Spring Security handler method is mapped to so that the entire OAuth2 / OIDC workflow is kickstarted when a GET request is sent here.

2. When the browser is redirected to here, this new session cookie is submitted in the request. The response provides further redirection onwards to the Google login server authorization endpoint with a URL similar to the following:

```
https://accounts.google.com/o/oauth2/v2/auth?
response_type=code&
client_id=xxx &
scope=openid profile email&
state=yyyy&
redirect_uri=http://localhost:8080/login/oauth2/code/google&
nonce=zzzzz
```

See
https://developers.google.com/identity/protocols/oauth2/openid-connect#sendauthrequest
for more info on the meaning of the specific query parameters above

- The `response_type=code` indicates that this is the start of the OAuth2 authorization code flow process and is used along with the `scope` parameter to determine whether OIDC flow is also involved.
- The `client_id=xxx` specifies the client id obtained earlier when obtaining OAuth2 client IDs in the GCP.
- The `scope=openid profile email`. The first value must be `openid` to indicate that is really an OIDC flow, while the `profile` and `email` values are used to indicate that the ID token that is eventually issued should include the user's default profile claims as well as the email claims. This is in line with the configuration for the OAuth Client ID that you performed earlier - and allows the app to access this info about the authenticated user
- The `state=yyyy` is used for CSRF protection
- The `redirect_uri` value is the Authorization callback URL that we specified for our OAuth Client when we initially registered it. For the case of Spring, this will be by default:

  ```
  http://localhost:8080/login/oauth2/code/xxxx
  ```

where xxx is the specific OIDC provider

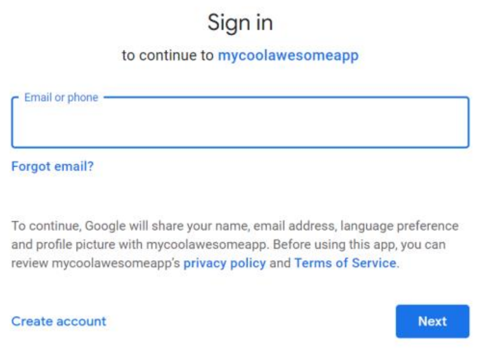
- The `nonce=zzzzz` is a nonce generated by Spring Security to protect against replay attacks

Note that the Discovery Document provides a list of all the relevant endpoints used by an OIDC provider service (such as Google) that is used in the OIDC workflow.
https://developers.google.com/identity/protocols/oauth2/openid-connect#discovery

The latest values for this document are available at:
https://accounts.google.com/.well-known/openid-configuration

For e.g. we can see here that the authorization endpoint is
https://accounts.google.com/o/oauth2/v2/auth

3. The request to this URL at the Google authorization server results in response that returns the standard login page with the registered name of the client app shown as well as links to the privacy policy and Terms of Service for the app (all which would have seen up during the registration of the OAuth Client).


Sign in

to continue to mycoolawesomeapp

Email or phone

Forgot email?

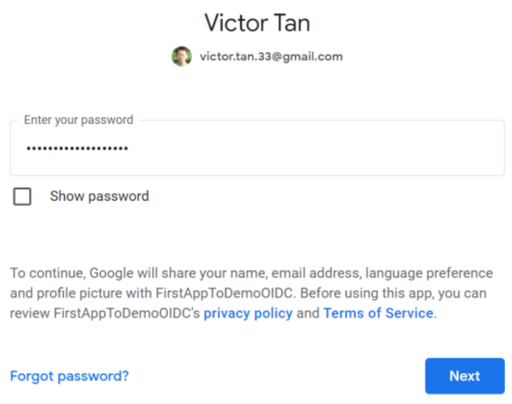
To continue, Google will share your name, email address, language preference and profile picture with mycoolawesomeapp. Before using this app, you can review mycoolawesomeapp's privacy policy and Terms of Service.

Create account

Next

4. The response from the authorization server to the GET at the URL above will also include some initial cookies as well a `x-auto-login` header that is used to auto-login a user in Chrome. When Chrome receives a response with this header, it will redirect to Google's OIDC consent page.

Typing in the email address and clicking next will cause the Javascript on the page to display a field to obtain the password.


Victor Tan

victor.tan.33@gmail.com

Enter your password

••••••••••••••••••

Show password

To continue, Google will share your name, email address, language preference and profile picture with FirstAppToDemoOIDC. Before using this app, you can review FirstAppToDemoOIDC's privacy policy and Terms of Service.

5. Clicking Next will trigger a series of redirect calls through a series of URLs shown below. All of these redirects are internal to the Google authentication workflow and not part of the official OIDC workflow (which does not explicitly specify how the authentication is implemented at the authorization server end).

https://accounts.google.com/CheckCookie?
continue=https://accounts.google.com/signin/oauth/consent?
authuser=unknown&
part=xxxx&
service=lso&
hl=en-GB&
checkedDomains=youtube&
checkConnection=youtube:97:0&
client_id=10740582430-
tc6tktoe06p0grhgko1hi92lf2qv5vcm.apps.googleusercontent.com&chtml=LoginDoneHtml&
gidl=EgIIAA

https://accounts.youtube.com/accounts/SetSID?ssdc=1
&sidt=xxxx&
continue=https://accounts.google.com/signin/oauth/consent?authuser=0&
part=xxxxx&
&tcc=1
&dbus=MY

https://accounts.google.com.my/accounts/SetSID?ssdc=1&
sidt=xxx&
continue=https://accounts.google.com/signin/oauth/consent?authuser=0&
part=xxx&
tcc=1

https://accounts.google.com/signin/oauth/consent?authuser=0&part=XXXXX

6. When the authentication completes successfully, the Google account owner whose credentials were used to login will receive an email to his registered email address notifying him that the specified OAuth Client app now has authorization to access specific data stored in his account.

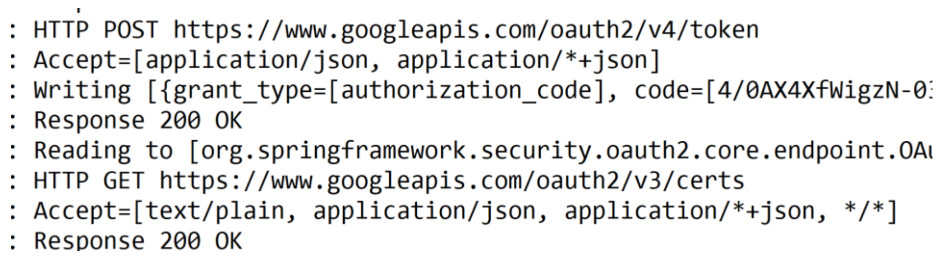
7. This is then follow with a redirect is back to the Authorization call back URL for the app that we had registered earlier.

http://localhost:8080/login/oauth2/code/google?
`state=xxxx`&
`code=yyyyy`&
scope=email profile openid https://www.googleapis.com/auth/userinfo.profile https://www.googleapis.com/auth/userinfo.email&
authuser=0&
prompt=consent

Spring Security framework will confirm that the CSRF token received back in `state=xxxx` is identical to the one it sent out earlier in step 3 earlier

8. Once Spring Security has received back the authorization code in this final redirect, it will now itself make a call to the token endpoint at the Google resource server to retrieve back an access token and an ID token. It will also need to make a call to the retrieve the public keys necessary to verify the signature on the tokens (which are in JWT format). You can see this from the server log output

See
https://developers.google.com/identity/protocols/oauth2/openid-connect#exchangecode



```
: HTTP POST https://www.googleapis.com/oauth2/v4/token
: Accept=[application/json, application/*+json]
: Writing [{grant_type=[authorization_code], code=[4/0AX4XfWigzN-0
: Response 200 OK
: Reading to [org.springframework.security.oauth2.core.endpoint.OA
: HTTP GET https://www.googleapis.com/oauth2/v3/certs
: Accept=[text/plain, application/json, application/*+json, */*]
: Response 200 OK
```

Again, the current values of the endpoints to invoke in order to send the token and retrieve the certs are given by the `token_endpoint` and `jwks_uri` values in the latest Discovery document.
https://accounts.google.com/.well-known/openid-configuration

9. The request to this URL from the browser will include the JSESSIONID cookie generated earlier in step 2. The response will include a new JSESSIONID cookie to track the authenticated user in a newly created session as well as provide a redirect back to the application root path (localhost:8080).

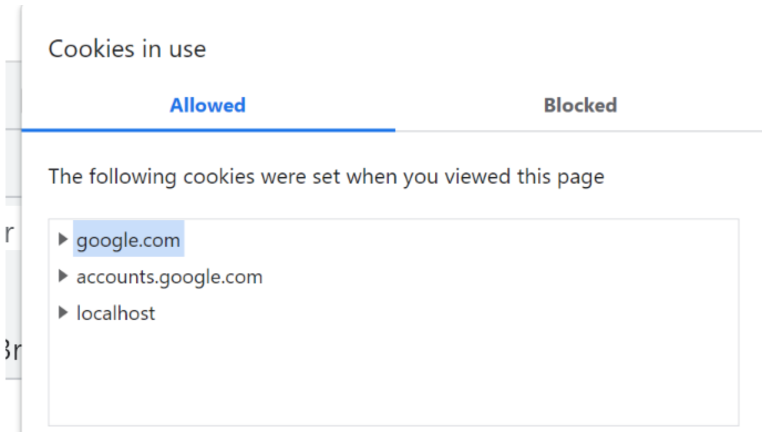
10. At this point, the app will now be in possession of both the access token as well as ID token in a POST request to the token endpoint. Subsequently the app code can extract user information from the ID token, and then decide how to authenticate the user based on information persisted locally by the app as well the information available from the token.

See Steps 4 - 6
https://developers.google.com/identity/protocols/oauth2/openid-connect

11. The final request from the browser to localhost:8080 includes the new JSESSIONID cookie and the response will return the view mapped to this path (i.e `home.html`). This cookie can be

viewed from the cookies dialog box accessible from the address bar and also from the Application tab in Chrome DevTools. All further interaction between the browser and the service will include this JSESSIONID cookie to track the session (identical to the situation for an authenticated user accessing a web app secured using typical Spring Security settings).

12. In addition, you should be able to also see all the cookies returned in the previous interactions with Google authorization server in the browser cache (chrome://settings/siteData?searchSubpage=google.com), or in the cookies dialog box accessible from the address bar. It will not be accessible in the Application tab in Chrome DevTools however, since the current domain that the browser is accessing is localhost (and not Google).


Cookies in use

The following cookies were set when you viewed this page

- google.com
- accounts.google.com
- localhost

13. If the single JSESSIONID cookie is now removed, a subsequent request to the application root will return a response that sets a new JSESSIONID cookie as well as redirect to http://localhost:8080/oauth2/authorization/google , as the service is no longer able to identify the user.

14. This is again followed with a redirection onwards to the Google login server OAuth endpoint as we have seen earlier.

https://accounts.google.com/o/oauth2/v2/auth?
```
response_type=code&
client_id=xxx &
scope=openid profile email&
state=yyyy&
redirect_uri=http://localhost:8080/login/oauth2/code/google&
nonce=zzzzz
```

The request to this URL will include all the cookies it cached from google.com and accounts.google.com from the previous initial successful authentication, which identifies the request as coming from an authenticated user. The standard Google login page is no longer displayed, and no further redirection through a variety of URLs is performed. Instead there is an immediate redirect back to the Authorization call back URL for the app that we had registered earlier.

http://localhost:8080/login/oauth2/code/google?
state=xxxx&

code=yyyyy&
scope=email profile openid https://www.googleapis.com/auth/userinfo.profile https://www.googleapis.com/auth/userinfo.email&
authuser=0&
prompt=consent

15. Everything from this point onwards will proceed as before. Any further requests again to the home page (at the application root path) and corresponding responses will continue to include the newly created session cookie to identify the user to the service (the Google cookies will obviously not be submitted because this is a different domain).

16. If we now remove this new session cookie again along with all the third party cookies from Google (using either the cookie dialog box or simply clearing the browser cache), then any further attempt to access the application root will result in a standard Google login page being displayed again (since there is no longer any means of identifying the user to either the service or the Google Authorization server).

Two things to keep in mind:

- To clear out the browse cache (including cookies) - More tools -> Clear Browsing data
- Attempt to navigate to the app home page (localhost:8080) in an incognito window. The default setting for the incognito window in Chrome is that it blocks third party cookies (such as the one returned by the Google authorization server for tracking the user on the browser) and also does not use any cookies from a previous session. Thus, navigating to the app home page (localhost:8080) in an incognito window will always force the user to reauthenticate at the Google login page, even if the user had previously authenticated successfully in a normal browser tab.

5 Running through the GitHub OAuth2 workflow

GitHub does not officially support OIDC (unlike Google or Keycloak) although the authentication flow is very similar to that of OIDC. The authentication mechanism is based on the OAuth2 authorization code flow.

Follow these instructions for creating a OAuth app:
https://docs.github.com/en/developers/apps/building-oauth-apps/creating-an-oauth-app

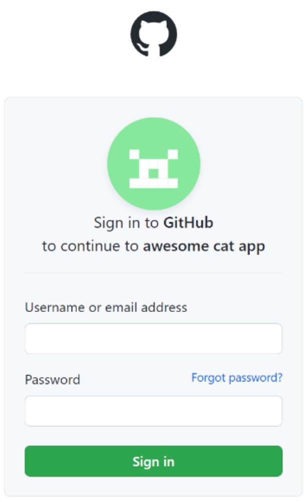
Enter any details you want for the various fields, with the most important field being the authorization callback:
http://localhost:8080/login/oauth2/code/github

At the application main page, select generate a new client secret. You will be required to reauthenticate to complete this process.

Copy down the newly generated client-id and client-secret and use this to configure the `application.properties` with the following new values:

```
spring.security.oauth2.client.registration.github.client-id=my-id
spring.security.oauth2.client.registration.github.client-secret=my-secret
logging.level.org.springframework.web.client.RestTemplate=DEBUG
```

When you are done, restart the app.
Open the Network tab in Chrome DevTools and check the Preserve Log checkbox in order to preserve tracking across page redirects.
Clear the browser cache and remove any tracking cookies, then make a request again to http://localhost:8080/
You should see a login page similar to the following:



If you are seeing a more complete authorization page, this means that your cookies from the previous GitHub authentication are being sent with your request. As usual, clear all the cookies from the GitHub domain from the cookie dialog box accessible from the address bar. Then navigate again to http://localhost:8080/

The flow for authorization is briefly summarized here:
https://docs.github.com/en/developers/apps/building-oauth-apps/authorizing-oauth-apps

The detailed outline is as follows:

1. For the initial attempt to access the application root path (localhost:8080), it returns a 302 redirect to http://localhost:8080/oauth2/authorization/github as well as setting a new JSESSIONID cookie to begin session tracking. This happens since there is no security configuration, which means by default, access to any resource requires authentication. For a normal Spring Security, this would normally result in a redirect to the default login path (/login) which returns the autogenerated login page. However, since we are using Spring OAuth2 Client here, the redirect is to: http://localhost:8080/oauth2/authorization/github instead. This is the default path to which a Spring Security handler method is mapped to so that the entire OAuth2 authorization workflow is kickstarted when a GET request is sent here.

2. When the browser is redirected to here, this new session cookie is submitted in the request. The response provides further redirection onwards to the GitHub authorization server with a URL similar to the following:

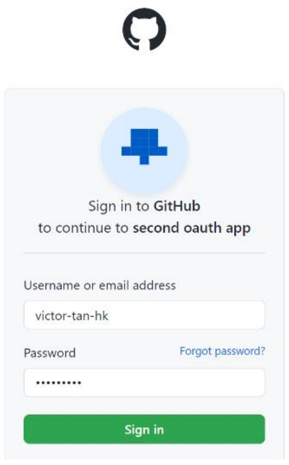
https://github.com/login/oauth/authorize?response_type=code&
client_id=`client-id`&
scope=read:user&
state=`hash-value`&
redirect_uri=http://localhost:8080/login/oauth2/code/github
(where *client-id* refers to the client id generated for the OAuth App that we registered earlier)

- The `response_type=code` indicates that the client expects an authorization code, which is a standard step in the OAuth2 authorization code flow process
- The `scope=read:user` is similar conceptually to OIDC scopes, where for a true OIDC implementation the value would be `scope=openid`. Here it simply means that the OAuth app is requesting the ability to read user info
- The `state` parameter is used for CSRF protection
- The `redirect_uri` value is the Authorization callback URL that we specified for our OAuth App when we initially registered it

3. The request to this URL at the GitHub authorization server results in response that provides a another redirection to:

https://github.com/login?client_id=*client-id*&
return_to=/login/oauth/authorize?client_id=*client-id*&
redirect_uri=http://localhost:8080/login/oauth2/code/github&
response_type=code&
scope=read:user&
state=*hash-value*

The response from the authorization server to the GET at the URL above will include the standard GitHub OAuth login page as well as 4 cookies: `_device_id`, `_octo`, `logged_in` and `_gh_sess` which will be used for session tracking the current user


Sign in to GitHub
to continue to second oauth app

Username or email address
victor-tan-hk

Password

•••••••••

4. Clicking Sign in sends a POST to https://github.com/session which includes the 4 cookies it had received earlier as well as a new `tz` cookie. in The form data includes the login name

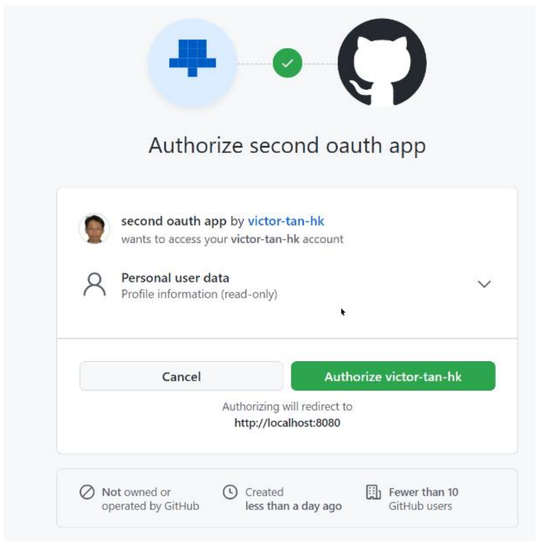
and password as well as a variety of key-value pairs that perform specific security functions, such as `timestamp`, `timestamp_secret`, `client_id`, `authenticity_token`

5. If the authentication credentials are validated correctly, a response is sent back which includes a redirect to the URL specified in the `return_to` parameter of the earlier URL:

https://github.com/login/oauth/authorize?
client_id=*`client-id`*&
redirect_uri=http://localhost:8080/login/oauth2/code/github&
response_type=code&
scope=read:user&
state=*hash-value*

In addition the response, includes many more new cookies: `has_recent_activity`, `user_session`, `_Host-user_session_same _site`, `color_mode`, `logged_in`, `dotcom_user`. In addition, `_gh_sess` is set to a new value to indicate the start of a new session to track the successfully authenticated user.

6. The next request to this redirect URL will includes all these new cookies. The response includes some changes to existing cookies, including `_gh_sess` which is again set to a new value to indicate the start of a new session to track the successfully authenticated user. An authorization page is also returned that specifies the specific portions of the current user's GitHub account that the OAuth app will be authorized to access.



Authorize second oauth app

second oauth app by victor-tan-hk
wants to access your victor-tan-hk account

Personal user data
Profile information (read-only)

Authorizing will redirect to
http://localhost:8080

Not owned or operated by GitHub
Created less than a day ago
Fewer than 10 GitHub users

7. Clicking the Authorize button will now result in a response that redirects back to the Authorization callback URL that we specified for our OAuth App when we initially registered it

[http://localhost:8080/login/oauth2/code/github?code=*code-value*&state=*hash-value*](http://localhost:8080/login/oauth2/code/github?code=code-value&state=hash-value)

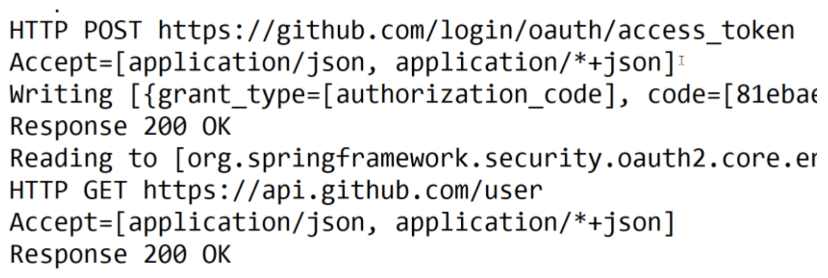
Here, `code-value` is the authorization code sent to the callback URL in the standard authorization code flow (or grant type): see Authorization code flow

At this point, the GitHub account owner whose credentials were used to login to the OAuth server will receive an email to his registered email address notifying him that the OAuth App now has authorization to access specific data stored in his account.

8. The request to this URL from the browser will include the JSESSIONID cookie generated earlier in step 2. The response will include a new JSESSIONID cookie to track the authenticated user in a newly created session as well as provide a redirect back to the application root path (localhost:8080).

9. At this point, the service will now be in possession of the authorization code and uses that (along with its client credentials - client ID and client secret) to request the access token it needs by making a POST to separate endpoint of the authorization server. Finally, it makes a request to a different endpoint along with the access token in order to obtain the required resources (in this case, it will be personal information about the authenticated user). See Step 2 and 3

https://docs.github.com/en/developers/apps/building-oauth-apps/authorizing-oauth-apps

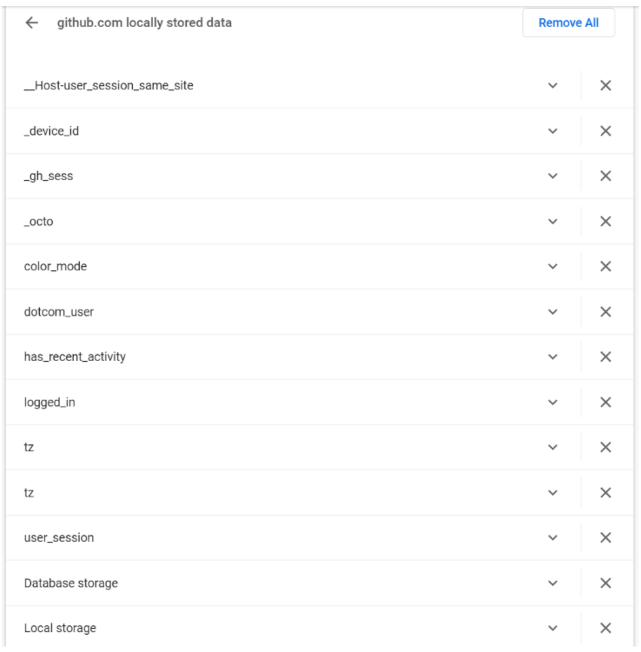


```
HTTP POST https://github.com/login/oauth/access_token
Accept=[application/json, application/*+json]
Writing [{grant_type=[authorization_code], code=[81ebae
Response 200 OK
Reading to [org.springframework.security.oauth2.core.e
HTTP GET https://api.github.com/user
Accept=[application/json, application/*+json]
Response 200 OK
```

10. The final request from the browser to localhost:8080 includes the new JSESSIONID cookie and the response will return the view mapped to this path. This cookie can be viewed from the cookies dialog box accessible from the address bar and also from the Application tab in Chrome DevTools.

11. All further interaction between the browser and the service will include this JSESSIONID cookie to track the session (identical to the situation for an authenticated user accessing a web app secured using typical Spring Security settings).

12. In addition, you should be able to also see all the cookies returned in the previous interactions with the GitHub authorization server in the browser cache (chrome://settings/cookies/detail?site=github.com), even though they may not be visible in the cookies dialog box accessible from the address bar or the Application tab in Chrome DevTools, since the current domain that the browser is accessing is localhost (and not GitHub).



__Host-user_session_same_site
_device_id
_gh_sess
_octo
color_mode
dotcom_user
has_recent_activity
logged_in
tz
tz
user_session
Database storage
Local storage

13. If the single JSESSIONID cookie is now removed, a subsequent request to the application root will return a response that sets a new JSESSIONID cookie as well as redirect to http://localhost:8080/oauth2/authorization , as the service is no longer able to identify the user.

14. This is again followed with a redirection onwards to the GitHub authorization server at the same URL as before:

https://github.com/login/oauth/authorize?
client_id=`client-id`&
redirect_uri=http://localhost:8080/login/oauth2/code/github&
response_type=code&
scope=read:user&
state=*hash-value*

The request to this URL will include all the GitHub cookies it cached from the previous initial successful authentication, which identifies the request as coming from an authenticated user and therefore the OAuth login page is not displayed. The response will now set a new value for `_gh_sess` to indicate the start of a new session to track the authenticated user.

15. It will then redirect to:

http://localhost:8080/login/oauth2/code/github?code=*code-value*&state=*hash-value*

The request to this URL will include the new JESSIONID Cookie set recently, and as in the previous case, the response will include another new JESSIONID cookie value to indicate the start of a new session with the newly authenticated user. However, this time the GitHub cookies from the browser will be viewable in the cookies dialog box accessible from the address bar.

16. Any further requests again to the home page (at the application root path) and corresponding responses will continue to include this new session cookie to identify the user to the service (the GitHub cookies will obviously not be submitted because this is a different domain).

17. If we now remove this new session cookie again along with all the third party cookies from GitHub (using either the cookie dialog box or simply clearing the browser cache), then any further attempt to access the application root will result in a the GitHub OAuth login page being displayed again (since there is no longer any means of identifying the user to either the service or the GitHub Authorization server).

18. To replay the exact sequence of steps in this authorization workflow process from Step 1, go to the Developer Settings for the relevant OAuth App just created and select Revoke all user tokens.

6 Obtaining authenticated user info from Google OIDC

Each OAuth2 / OIDC provider has different ways of providing user-related info after the authentication workflow process completes.

Reset `application.properties` to the values you used earlier for Google OIDC. You can remove the DEBUG property.

```
spring.security.oauth2.client.registration.google.client-id=your-id
spring.security.oauth2.client.registration.google.client-secret=your-secret
```

For the case of Google OIDC, the actual Authentication object that is instantiated is OAuth2AuthenticationToken which is extended from AbstractAuthenticationToken that implements the key Authentication interface. The `getPrincipal()` call on this object in turn returns a type of DefaultOidcUser which implements the OidcUser interface. From DefaultOidcUser, we can then extract the ID Token (that contains all the related authentication information at the end of an OIDC flow), and from that token extract a series of claims.

In the package `com.workshop.security` in `src/main/java`, place these files:

`MainController`

Clear out the single session tracking cookie, restart the app and navigate to the `view` page specifically. Check for the log output from the server side which provides us specific info extracted from all the classes discussed earlier.

The list of claims in an ID token returned from a successful OIDC flow using Google is shown at:
Topic 5. Obtain user information from the ID token
https://developers.google.com/identity/protocols/oauth2/openid-connect

Points to note

- The value of `user.getName()` is identical to the value of the `sub` claim in the ID token since is the unique identifier for the user, unique among all Google accounts
- Most of the relevant information in the claims can also be obtained via the various getter methods in OidcUser that it inherits from StandardClaimAccessor - for e.g. `email`, `email_verified`, `family_name`, `given_name` - so technically there is no need to extract the claims explicitly via `getIdToken`, `getClaims`, etc
- The authorities of the OidcUser are basically the scope strings returned from one of the steps in the OIDC workflow.

```
scope=email profile openid
https://www.googleapis.com/auth/userinfo.profile
https://www.googleapis.com/auth/userinfo.email
```

This scope is actually not related to the authorization rights of the user in Google, but related to the authorization rights of the OAuth client with regards to which user resources is can access (in this, the user's email and profile).

In addition, Spring automatically populates an additional standard authority of ROLE_USER so that the current authenticated user can be authorized appropriately using the typical authorization constraints that are role-based (for e.g. `hasRole, hasAnyRole, hasAuthority, hasAnyAuthority`)

The amount of information available in the claims from the ID token is directly related to the personal info which the account owner has decided to make public to be shared with others:
https://support.google.com/accounts/answer/6304920

7 Obtaining authenticated user info from GitHub OAuth2

Reset `application.properties` to the values you used earlier for Google OIDC. You can remove the DEBUG property.

```
spring.security.oauth2.client.registration.github.client-id=my-id
spring.security.oauth2.client.registration.github.client-secret=my-secret
```

Clear out the single session tracking cookie, restart the app and navigate to the `view` page specifically. Notice now that we get an exception reported back in the Whitelabel Error page, as the classes that we used to extract user info from an OIDC workflow is no longer valid for the case of OAuth2 workflow.

We will make some changes to our code to reflect the need to use a different class (OAuth2User) to cast the principal type returned from the Authentication.

In the package `com.workshop.security` in `src/main/java`, make these changes:

`MainController-v2`

Restart the app and navigate to the `view` page specifically.
This time you should see the relevant log output on the server pertaining to the authenticated user

Notice here that the amount of user-related information is significantly more than in the case of Google. The amount of user-related information that can be retrieved from the various tokens (access token for the case of OAuth2 and ID token for the case of OIDC) will vary greatly between providers. This is something you will need to take into account when deciding which 3rd party OAuth2 / OIDC providers to integrate into the authentication mechanism of your app.

8 Customizing a login page for OAuth2 / OIDC login providers

At the moment, the redirection to the authorization endpoint of the configured OAuth2 / OIDC provider happens with the first request to the application root path. This is because with no initial custom security configuration, then by default, authentication is required for all requests to the app. The inclusion of the OAuth2 Client starter dependency means Spring that any incoming unauthenticated request for any resource will kickstart the entire OAuth2 / OIDC workflow where the user will be redirected to the OAuth2 / OIDC login page of the specific configured provider.

We can configure the OAuth2 login process in the similar way that we did for a standard login process, by extending on the builder method chaining for HttpSecurity in the main security configuration class.

In the package `com.workshop.security` in `src/main/java`, place this file:

`MainSecurityConfig`

In the package `com.workshop.security` in `src/main/java`, make changes to this file:

`MainController-v3`

In `src/main/resources/templates`, place this file:

`login.html`

In `src/main/resources/templates`, make modifications to these files:

`view-v2`
`home-v2`

Reconfigure `application.properties` with the `client-id` and `client-secret` for both GitHub and Google respectively:

```
spring.security.oauth2.client.registration.google.client-id=your-google-id
spring.security.oauth2.client.registration.google.client-secret=your-google-
secret

spring.security.oauth2.client.registration.github.client-id=my-github-id
spring.security.oauth2.client.registration.github.client-secret=my-github-secret
```

We add a logout button to the home page to trigger a logout action (i.e. to invalidate the current active session and force the user to reauthenticate again at the login page), and add this to our builder configuration for HttpSecurity in `MainSecurityConfig`. This configuration here specifies a customized login page for both the normal and OAuth type logins. This ensures that any attempt to access any resource that does not have a `permitAll()` specified on it will result in a redirect to this customized login page.

```
@Override
protected void configure(HttpSecurity http) throws Exception {

  http.authorizeRequests()
    .antMatchers("/", "/login", "/oauth/**").permitAll()
    .anyRequest().authenticated()
  .and()
    .formLogin().loginPage("/login")
  .and()
     .oauth2Login().loginPage("/login")
   .and()
     .logout().logoutUrl("/performLogout")
     .logoutSuccessUrl("/login?logout=true");

}
```

We add two links in the customized login page to the standard authorization endpoints in our app to kickstart the authentication workflow for Google and GitHub respectively.

MainController provides logic to distinguish between successful logins from Google and GitHub. Since successful authentications will from both these sources will result in a different instantiation of the core OAuth2User interface (with DefaultOidcUser being a subclass of DefaultOAuth2User), we can use a basic type check to distinguish between authenticated users from these two OAuth2 providers.

We then cast the Principal object to the correct type and extract the relevant data from them to populate the 4 parameters (`realUserName`, `userEmail`, `userLocation` and `userCompany`) that we place in the Model to be subsequently retrieved and displayed in `view.html`. Notice how we use a `th:if` conditional attribute in this view to decide whether to display text regarding company and location, since this information is only available from a GitHub login.

Restart the app.

Select the logout button from the home page to invalidate the current session and redirect to the customized login page. Verify that the correct user info is retrieved and displayed on the view page depending on the specific OAuth2 login utilized (Google or GitHub).

Note that this approach for distinguishing between Google or GitHub on the basis of a type check works only because these two providers employ different workflows for authentication (Google -> OIDC, GitHub -> OAuth2). If you provide login options for two (or more) OIDC providers (e.g. Paypal, Yahoo, Microsoft, etc), you will need some other means of distinguishing between them. For e.g. the `iss` claim will typically hold the name of the provider:
`"iss": "https://accounts.google.com"`
https://developers.google.com/identity/protocols/oauth2/openid-connect#an-id-tokens-payload

9 Integrating with standard Spring authentication mechanisms

We have now seen how to configure and perform an OAuth2 / OIDC login as well as a standard username / password login via the standard Spring Security authentication mechanism (which we have already covered in detail in previous labs). In typical scenarios, we will usually combine both approaches so that users have an option of registering a user account with us and logging in using that account or alternatively using an existing account they have with a known OAuth / OIDC provider (of which there are many in the market now: Apple, Facebook, Instagram, Twitter, etc

https://en.wikipedia.org/wiki/List_of_OAuth_providers

In `src/main/resources/templates`, make these changes:

```
login-v2
home-v3
view (revert back to original version)
```

In the package `com.workshop.security` in `src/main/java`, make these changes:

```
MainSecurityConfig-v2
MainController-v4
```

In the package `com.workshop.security` in `src/main/java`, place these files:

```
CustomUserDetails
CustomUserDetailsService
StringConstantsHolder
```

We add additional HTML form elements to the customized login page for a standard login, including the ability to conditionally display messages in accordance to the query parameters. The login options for Google and GitHub are still maintained.

We move the display of the various variables related to user info to the home page and make their display completely conditional on them not being null. The view page is restored to its normal state.

We add a standard in-memory authentication using hardcoded usernames and passwords from the StringConstantsHolder class. As expected, user login via the standard customized login page will be based on this information.

We provide two additional classes: CustomUserDetails and CustomUserDetailsService to provide and retrieve additional information about the users (in addition to their basic username and password). All of the relevant information is also hardcoded. Of course, in a real-life app, all of this (along with the username / password info) would be persisted to and retrieved from a backend database table - but we keep it simple here in this example.

Restart the app.

Verify that you are now able to login using the standard in-memory authentication route in addition to GitHub and Google, and that the correct user related information (such as email, location, etc) are displayed correctly on the home page. Try logging in and out with different user accounts.

10 Enforcing access control on OAuth2 / OIDC users

We can also enforce access control on the OAuth2 / OIDC users by examining the authorities / roles that are assigned to them by Spring Security after the authentication completes successfully. As we saw earlier, for the case of Google OIDC, these are the authorities which are the same for every user:

- `ROLE_USER`
- `SCOPE_https://www.googleapis.com/auth/userinfo.email`
- `SCOPE_https://www.googleapis.com/auth/userinfo.profile`
- `SCOPE_openid`

And for the case of GitHub OAuth2

- `ROLE_USER`
- `SCOPE_read:user`

Notice that both have authorities ROLE_USER, which means they both have the USER role. So we can use this role when we want to express authorization constraints that include both of them, and use their unique authorities for authorization constraints that apply specifically only to them.

In the package `com.workshop.security` in `src/main/java`, make these changes:

`MainSecurityConfig-v3`

Create a folder `error` in `src/main/resources/templates` and place these new files there:

`403.html`
`404.html`

Add this following property to the existing properties in `application.properties`:

```
server.error.whitelabel.enabled=false
```

We have now added in authorization constraints to various antMatcher paths in the main security configuration based on the roles / authorities of authenticated users in the system:

```
      // Anyone with role ADMIN or USER can access - this means essentially
everyone
      .antMatchers("/view").hasAnyRole(StringConstantsHolder.role1,
StringConstantsHolder.role2)

      // Only the GitHub or the Google user can access
      .antMatchers("/add").hasAnyAuthority("SCOPE_read:user", "SCOPE_openid")

      // Only the GitHub user can access
      .antMatchers("/update").hasAuthority("SCOPE_read:user")

      // Only role ADMIN can access
```

```
       .antMatchers("/delete").hasRole(StringConstantsHolder.role2)

```

We have also configured a custom page to handle any 403 Forbidden Whitelabel error pages that are returned as a result of AuthorizationExceptions.

Restart the app.

Verify that all the authenticated users are only able to access the views authorized for them in accordance to the authorization constraints described above. Check that the custom page is returned in cases where the authorization constraint on a view is violated and also when an attempt is made to access a resource that does not exist (for e.g. http://localhost:8080/asdfasdfasdf)