OAuth 2.0 Hands-on

(SecAppDev 2014 version.)

Prerequisites

Make sure the following prerequisites are met before starting the hands-on session.

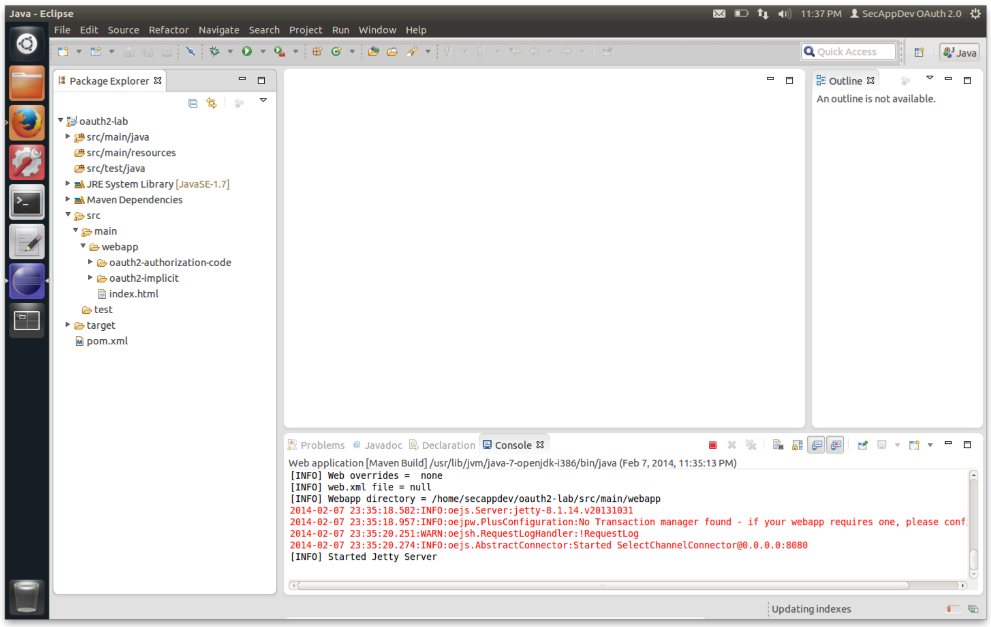
- **Get a Google account.** The examples in this lab set up an OAuth client to the Google APIs. You need a working Google account to register your own client applications and to get results back from the APIs. If you don't have one already, you can create a dummy account at https://accounts.google.com/SignUp.
- **Start up the VM image.** A VM image will be distributed to you containing an IDE and the necessary files to get started. This image is based on Ubuntu Linux. It contains an IDE (Eclipse) and the necessary files to get started with the hands-on session.

References

- The presentation: http://prezi.com/2uxj3_30cts1/oauth-20-2014/
- The OAuth 2.0 Spec: http://tools.ietf.org/html/rfc6749

Overview of the VM image

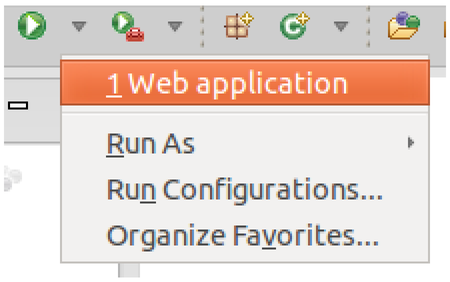
The screen shot below shows the virtual machine running the Eclipse IDE.



```
Web application [Maven Build] /usr/lib/jvm/java-7-openjdk-i386/bin/java (Feb 7, 2014, 11:35:13 PM)
[INFO] Web overrides =  none
[INFO] web.xml file = null
[INFO] Webapp directory = /home/secappdev/oauth2-lab/src/main/webapp
2014-02-07 23:35:18.582:INFO:oejs.Server:jetty-8.1.14.v20131031
2014-02-07 23:35:18.957:INFO:oejpw.PlusConfiguration:No Transaction manager found - if your webapp requires one, please conf
2014-02-07 23:35:20.251:WARN:oejsh.RequestLogHandler:!RequestLog
2014-02-07 23:35:20.274:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

A Java Maven web application was configured with the necessary files. This application can be founds in `/home/secappdev/oauth2-lab`.

- In the folder `src/main/webapp` are the html and javascript files of the application.
- The folder `src/main/java` contains the class `OAuth2Util` with utility methods.
- To start the application from the command line, go into the directory and type `mvn jetty:run`
- To start the web application from within the IDE, click the Run button ( ) and choose the first target. It is possible to debug the application as well.



The username and password for the default environment is: `secappdev` / `SecAppDev2014`. Sudo access is enabled for this user.

1. Use the Google OAuth2 playground

The Google OAuth playground shows the required calls to access a Google API, including the requests to the OAuth 2.0 endpoints. They can show you what you need to provide to make the calls yourself.

- Open the Firefox browser and visit the OAuth2 Playground (https://developers.google.com/oauthplayground/).
- Select a Google API you want to try out (e.g. the contacts API).
- Go through the steps to get an authorization code, to translated it into an access and refresh token and to call the actual service. In the right frame notice the calls being made.

As you can see, the OAuth2 Playground is a handy tool to take a sneak preview at the Google APIs and how to call them.

2. Use the implicit grant flow (Javascript)

In this lab, you will create a Javascript application that shows the names of the contacts of the person that logs in. The necessary files to get started can be found in `src/main/webapp/oauth2-implicit`.

- First your project has to be registered in the Google Cloud Console (https://cloud.google.com/console/project). Use the following settings:
  - API to use: Contacts API. (Scope: https://www.google.com/m8/feeds)
  - Create a client ID for a web application. The Javascript origin should be http://localhost:8080/ (used for CORS - Cross-Origin Resource Sharing) and the redirect URI http://localhost:8080/oauth2-implicit/callback.html
  - Don't forget to enter a name for your application for the consent screen.
- Click the Run button in Eclipse and start the embedded webserver. Visit http://localhost:8080 to find out what the application already does. In the next steps, you have to complete the application.
- Open `oauth2-implicit.js`. Modify the `getLoginUrl()` function to generate the correct login url.
  - The Google OAuth authorization endpoint is located at: https://accounts.google.com/o/oauth2/auth
  - Add the correct parameters to request an access code (response_type, client_id, redirect_uri, scope, state).
  - You can generate a secure random state using the provided `generateSecureRandomState()` function. (Note: in Internet Explorer, this function falls back to an insecure random generator.)
  - For bonus points: be sure to URL encode all the parameter values.
- After your changes, the login button should open a popup window where you can login with your Google account. After logging in and granting access, the popup loads the callback page with a number of OAuth parameters in the hash of the URL. The next step is to make sure the access token is passed back to the original window.
  - First, modify the `parseHash()` function in `callback.html` to get the parameters out of the hash. Fill in a correct regular expression to extract the (name)=(value) pairs.
  - Make sure these parameters are passed onto the original window. You can call its `setOAuthParameters()` function to do that. The window can be retrieved using `window.opener`. Afterwards the popup window can close itself.
  - Don't forget to validate the incoming state!
- Finally, update the `setOAuthParameters()` and `loadContacts()` functions to use the access token.
- Now you can move on to the next lab. If there is any time remaining at the end, you can take a look at how to handle errors:
  - What if the user cancels the request (or another OAuth error code is returned).
  - What if the access token expires.

3. Use the authorization code grant flow (Java only)

For this example, we didn’t use a web framework: the complete solution consists of a couple of JSP pages calling static utility methods. (In real life you probably want to use a framework. However with the number of frameworks out there, choosing one that everybody knows/likes is virtually impossible.)

- Create an additional client ID in the Google Cloud console.
  - Javascript origin:http://localhost:8080/ (used for CORS - Cross-Origin Resource Sharing).
  - Redirect URI: http://localhost:8080/oauth2-authorization-code/callback.jsp

- Start the application using `mvn clean jetty:run`. Visit the authorization code grant flow to see what happens.

- Provide an implementation of `OAuth2Util.getLoginUrl()` to provide a redirect to the login page.
  - The Google OAuth authorization endpoint is located at: https://accounts.google.com/o/oauth2/auth
  - Add the correct parameters to request an authorization code (response_type, client_id, redirect_uri, scope, state).
  - A quick way to generate a cryptographically strong pseudo random state is using `UUID.randomUUID()`.
  - For bonus points: be sure to URL encode all the parameter values.

- Restart the Jetty server and try this out. If all goes well, after logging in you should end up on the callback page, but without showing the access token.

- Provide an implementation of `OAuth2Util.retrieveAccessToken()` to fetch an access token from Google.
  - You can use `sendHttpPost()` to send a POST request with parameters and `parseJson()` to parse the JSON response.
  - Use the following endpoint to get the access token: https://accounts.google.com/o/oauth2/token
  - Pass the following parameter: client_id, client_secret, redirect_uri, grant_type, code. (Note: client_id and client_secret can also be passed using HTTP basic authentication).
  - Don't forget to validate the incoming state!

- Restart the Jetty server and try this out. If all goes well, the access token will be shown.

- If you have time left:
  - Provide error handling (what if the user cancels, what if the authorization code has expired).
  - Use the refresh token to get a new access token.
  - Use the access token in a call to a Google API.

4. Roll your own

If you don't know Java, you could try to work out a working example in the language of your choice. I will be happy to assist you.