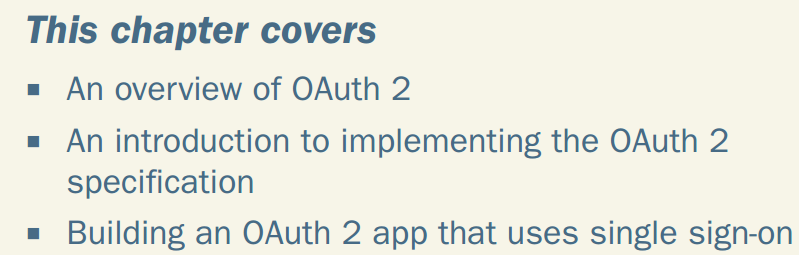
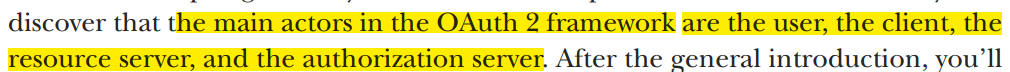
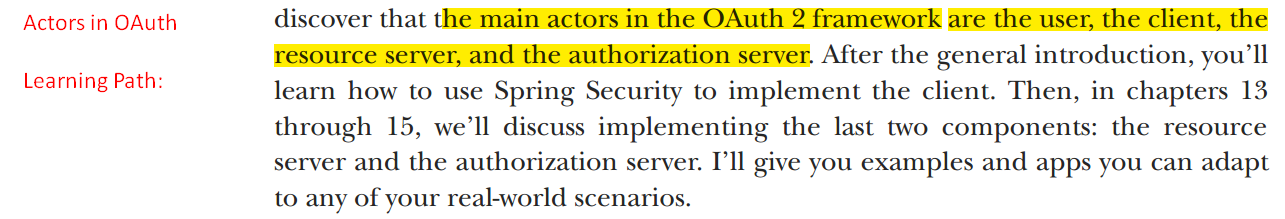
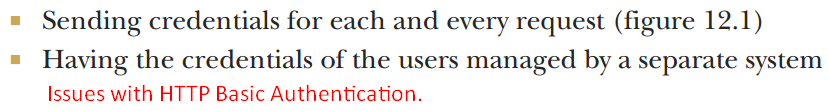
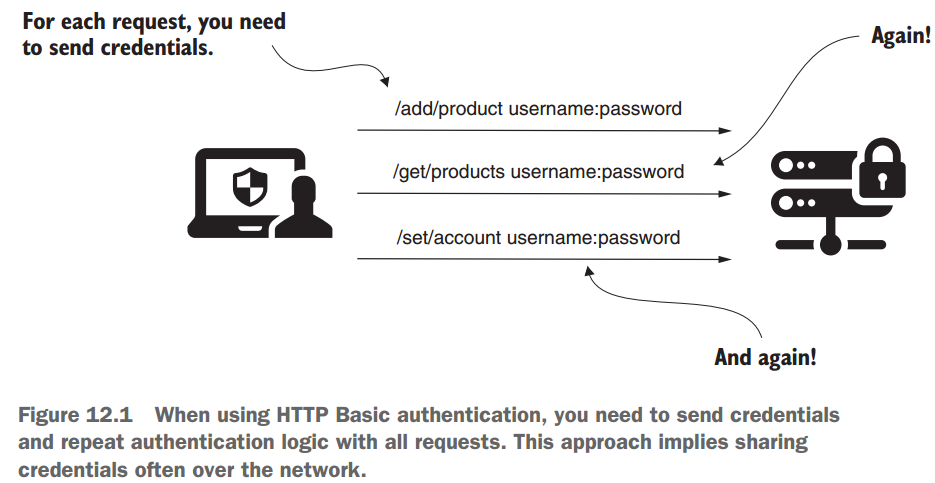
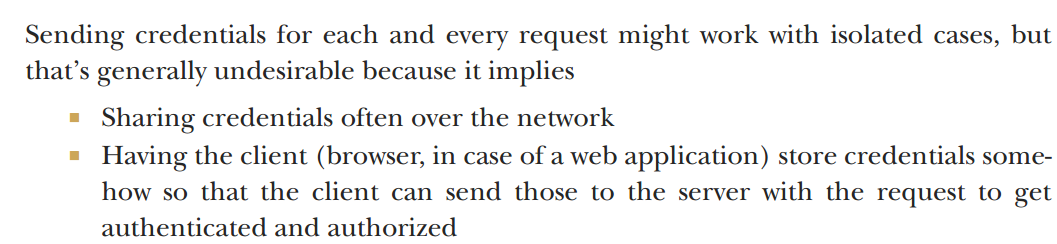
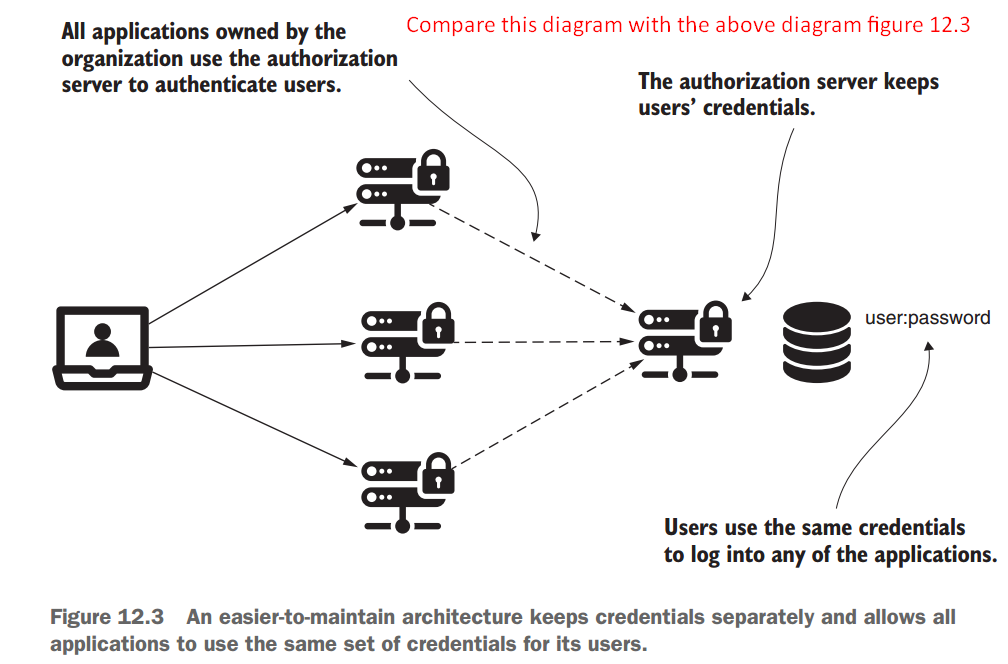
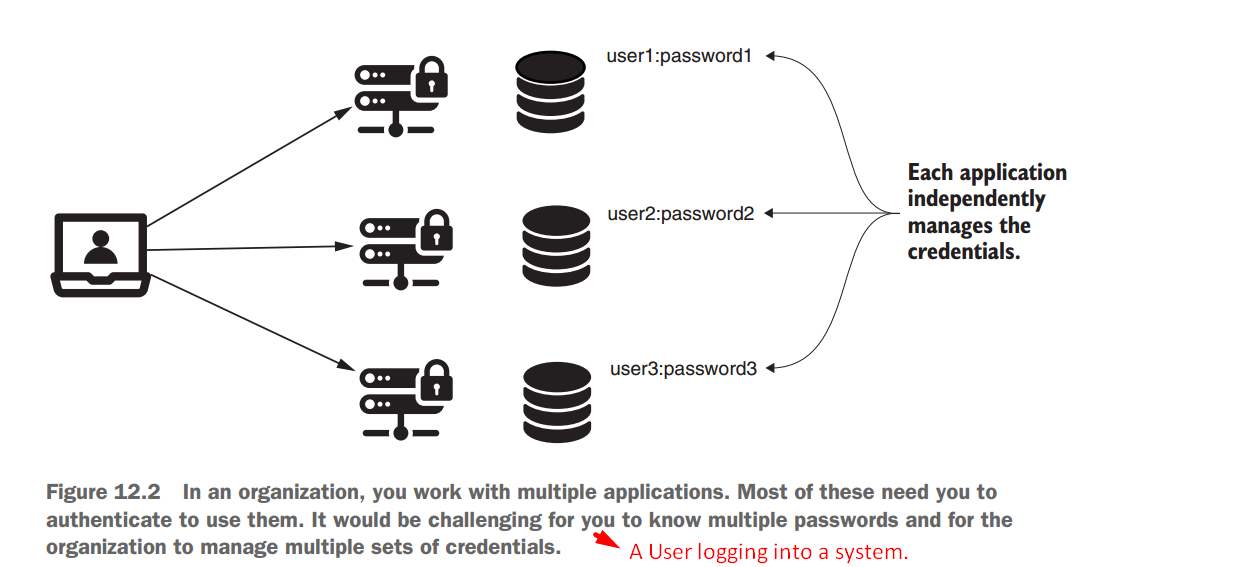
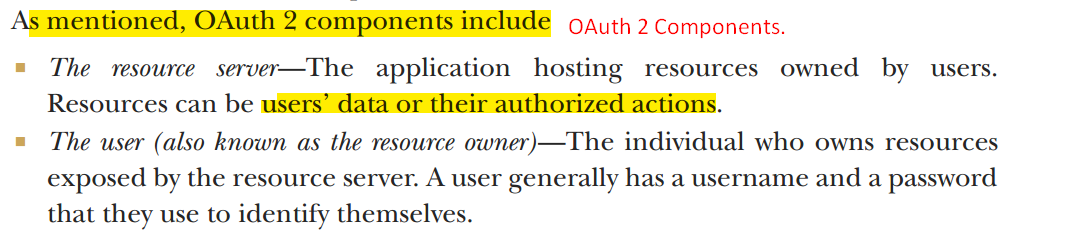
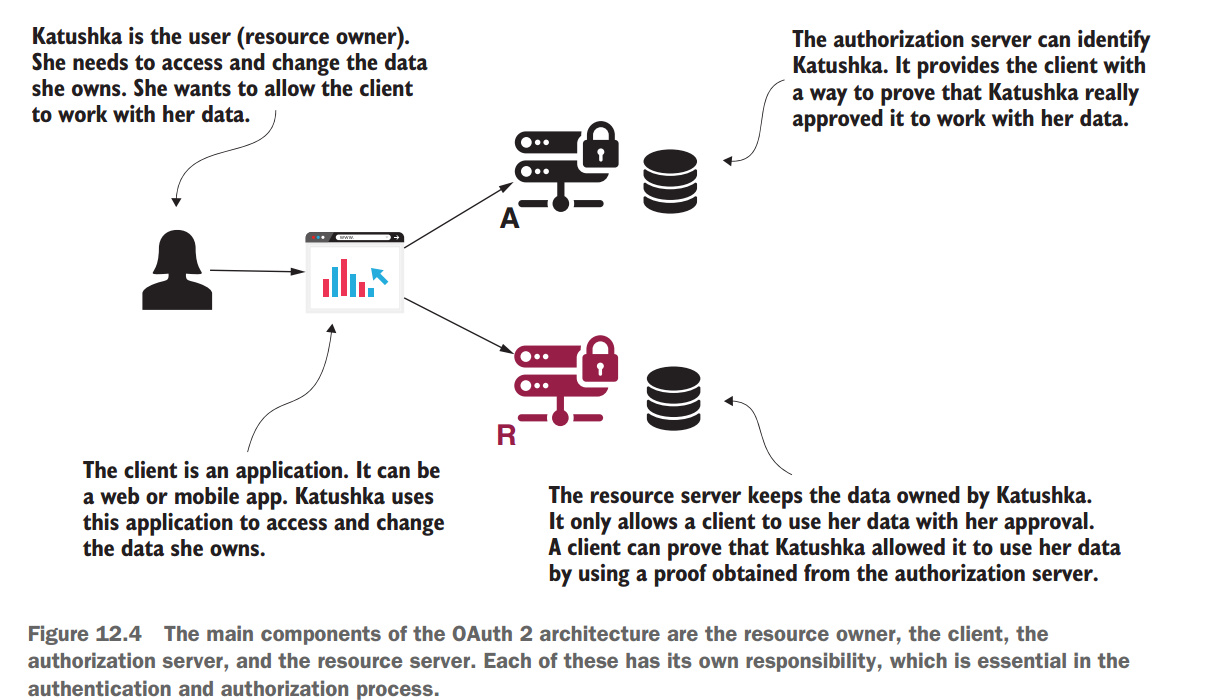
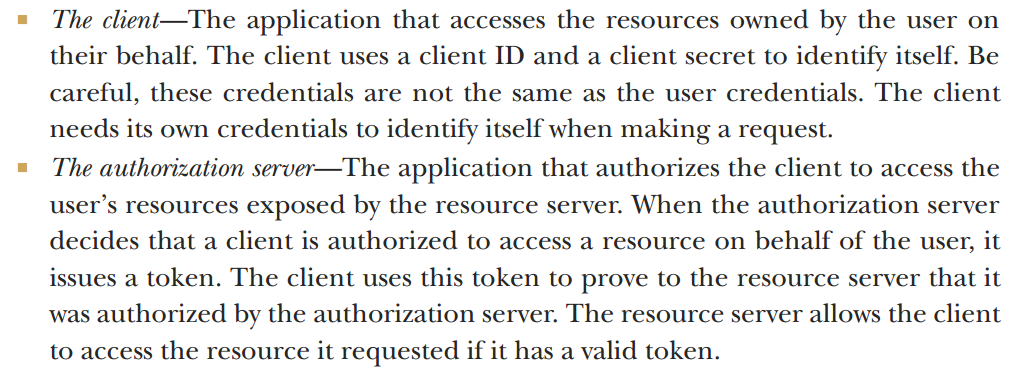
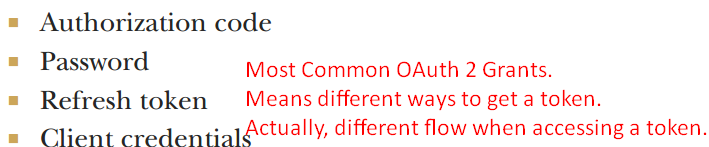
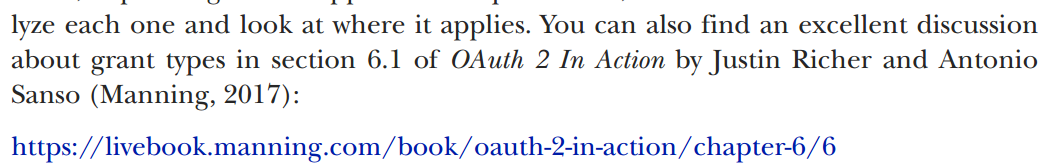
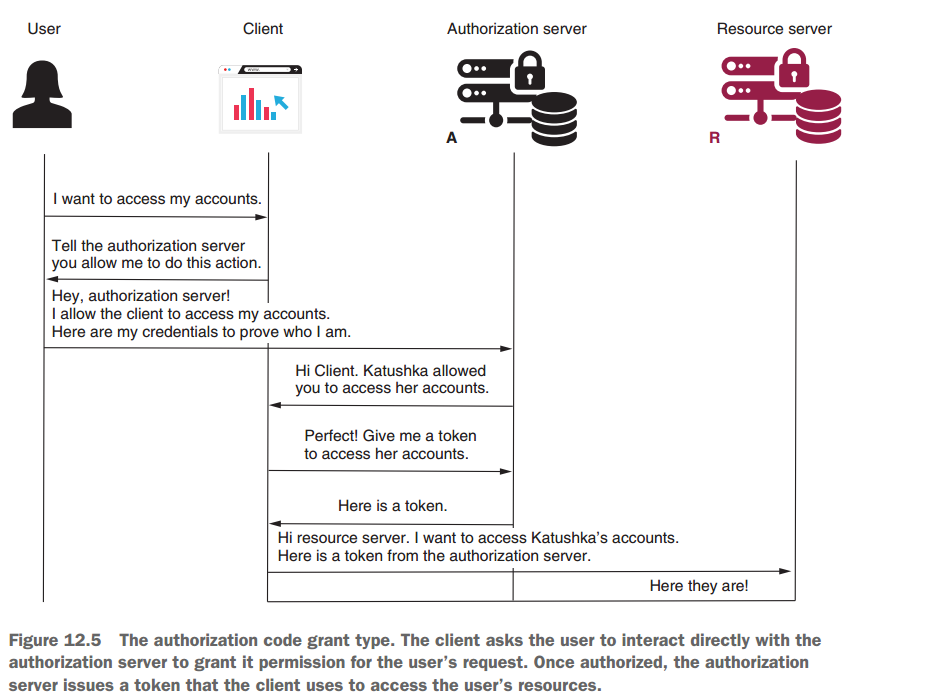
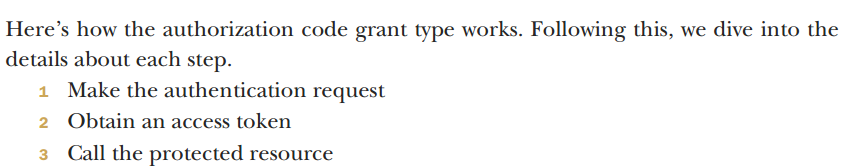
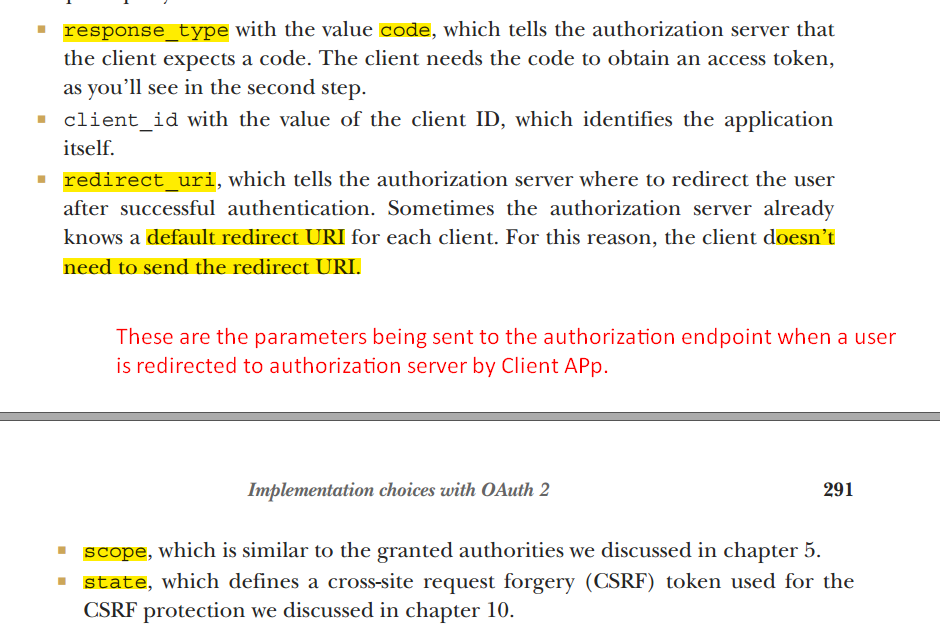
1. 
2. Actors in Oauth:  
   
3. 
4. 
5. **Purpose**:
   1. To secure web app.
6. In most cases, Oauth 2 is referred to as an **Authorization Framework (or a specification framework)** whose primary purpose is to allow a 3rd party website or app to access a resource.
7. Also called **Delegation Protocol.**
8. It is not specification implementation or a library.
9. 
10. 
11. 
12. Get rid of these two points from our applications’ architecture as they weaken security by making credentials vulnerable.
13. In most cases, we want a separate system to manage user credentials.
14. 
15. 
16. *12.2 The components of the OAuth 2 authentication*  
    *architecture*
17. **Agenda**:
    1. We will discuss the components that act in Oauth 2 authentication implementations.
    2. Different flows that cause different interactions b/w these components.
18. 
19. 
20. 
21. *12.3 Implementation choices with OAuth 2*
22. **Agenda**:
    1. Oauth 2 implies multiple ***possible authentication flows***, and you need to know which one applies to your case.
    2. How to apply Oauth 2, depending on the architecture of your application.
23. What is Oauth 2?
    1. Oauth 2 refers to using tokens for authorization.
    2. Tokens are like access cards using what you can access specific resources.
24. Oauth offers multiple ways to obtain a token called 🡺 **Grants**.
25. 
26. To read live  
    <https://livebook.manning.com/book/oauth-2-in-action/chapter-6/6>
27. 
28. *12.3.1 Implementing the authorization code grant type*
29. 
30. 
31. STEP 1: MAKING THE AUTHENTICATION REQUEST WITH THE AUTHORIZATION CODE GRANT TYPE

**Scenario**: A user wants to access his secured resource from a different app or web site.

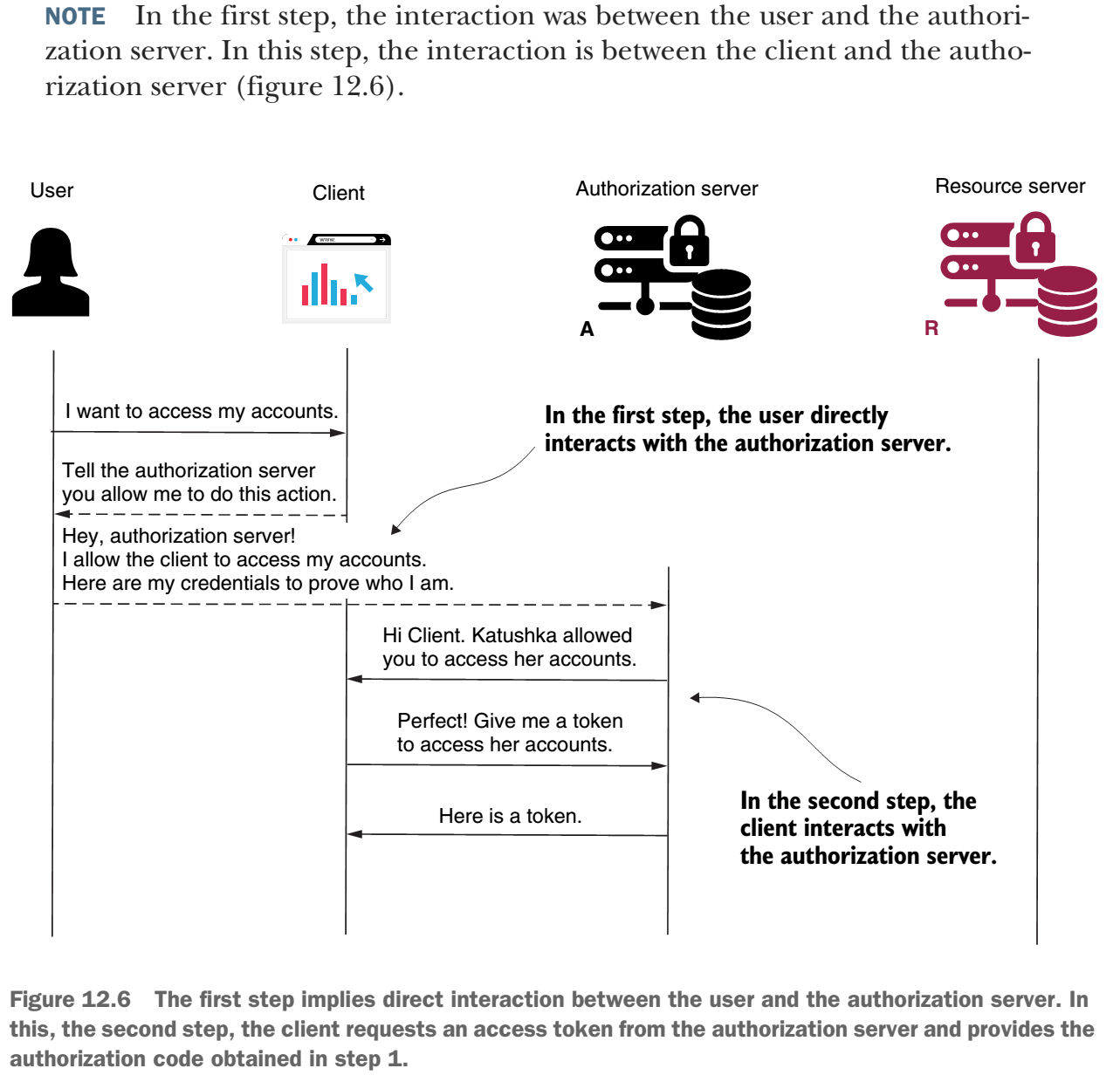
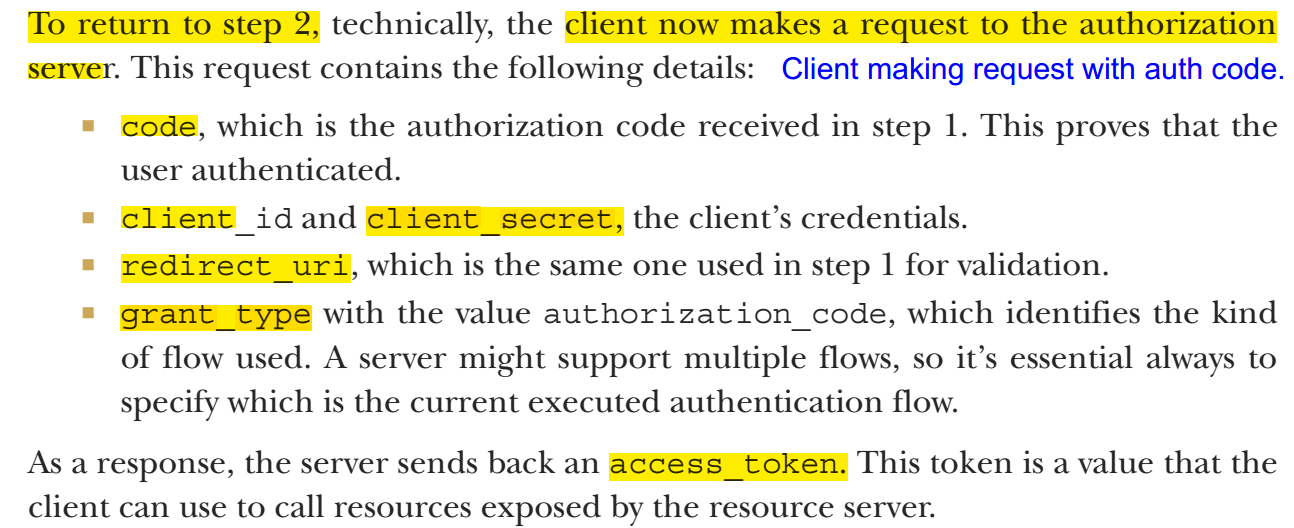
* 1. The user is on the client app.
  2. User tries to access secured resource on a different server from client app.
  3. Client app redirects the user to the end point of the authorization server of the resource server.
  4. The authorization server will represent a login page to the user where user enters credentials.

**NOTE**: The user doesn’t enter his credentials on the Client app.

* 1. During redirection to authorization endpoint, client sends the following details
     1. 
  2. After successful authentication, the authorization server calls back the client on the redirect URI and provides   
     a code and the state value.
  3. The client checks that the state value is the same as the one it sent in the request to confirm that it was not someone else attempting to call the redirect URI.
  4. Then the client uses the code to obtain an access token as presented in step 2.

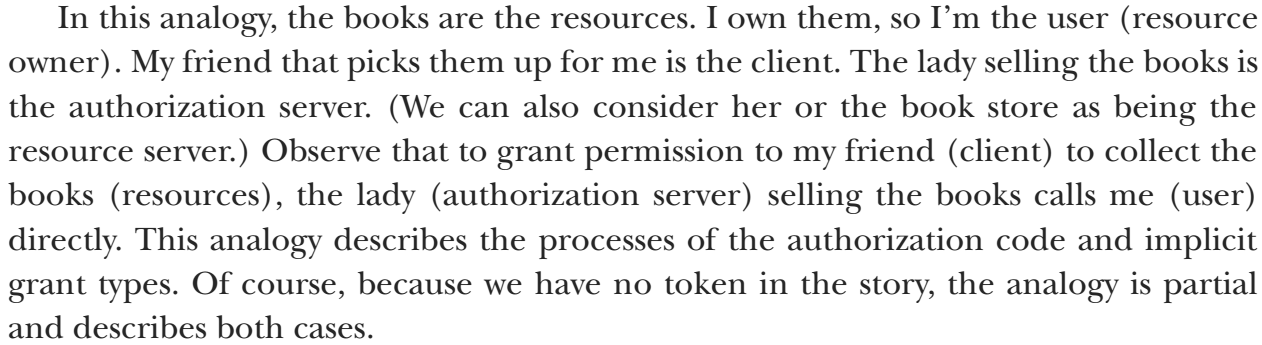
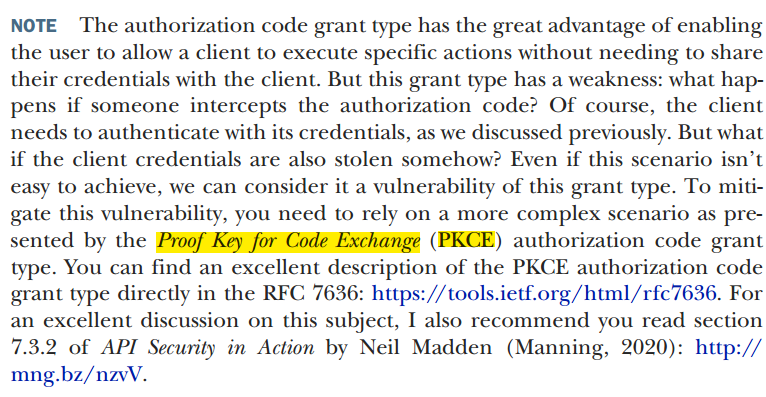
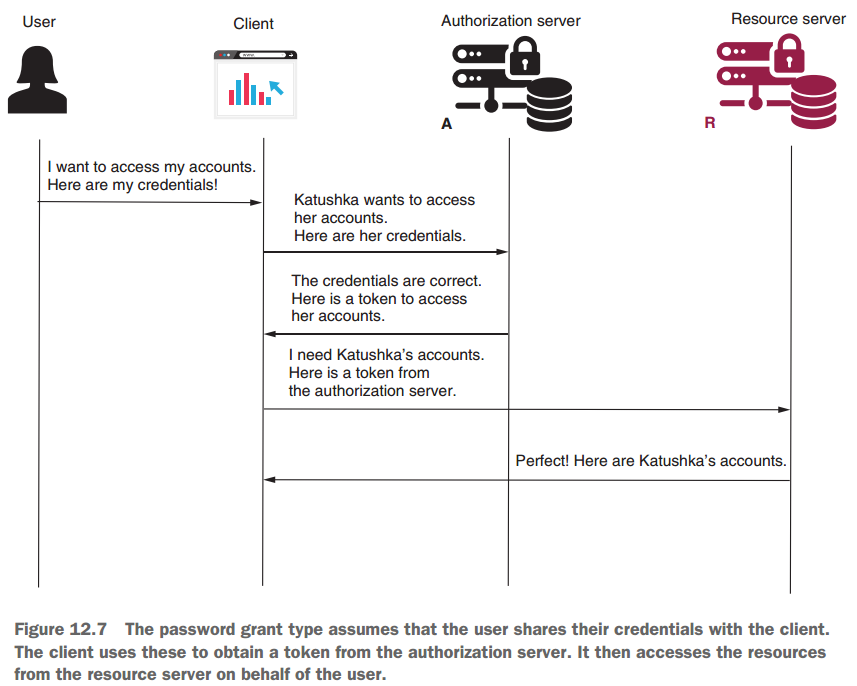
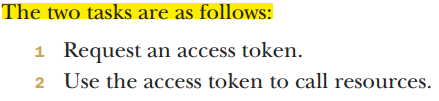
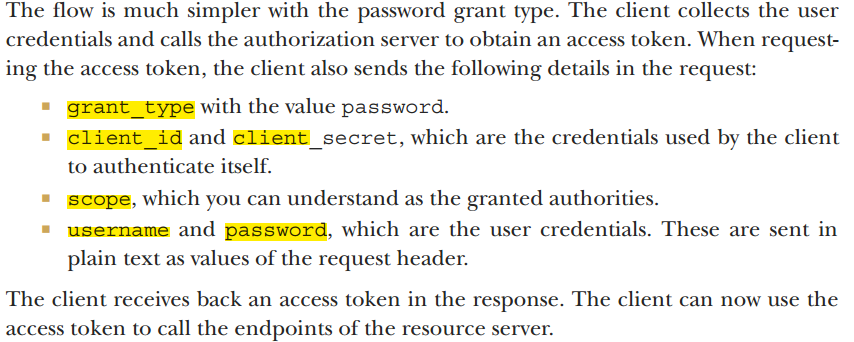
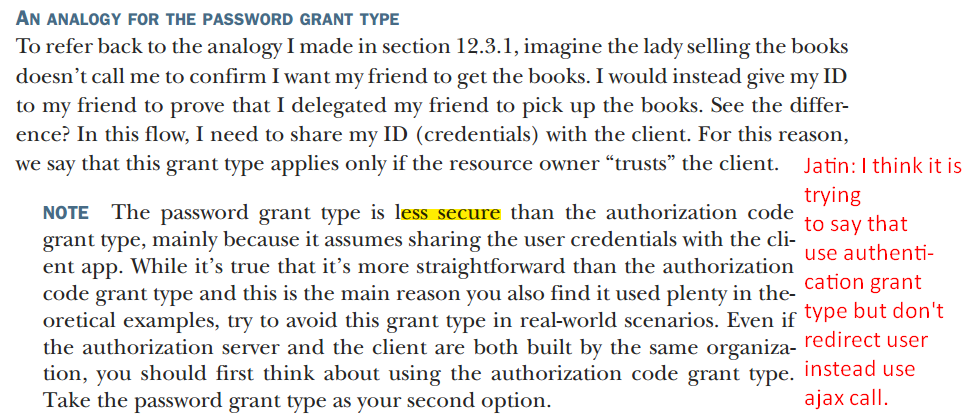
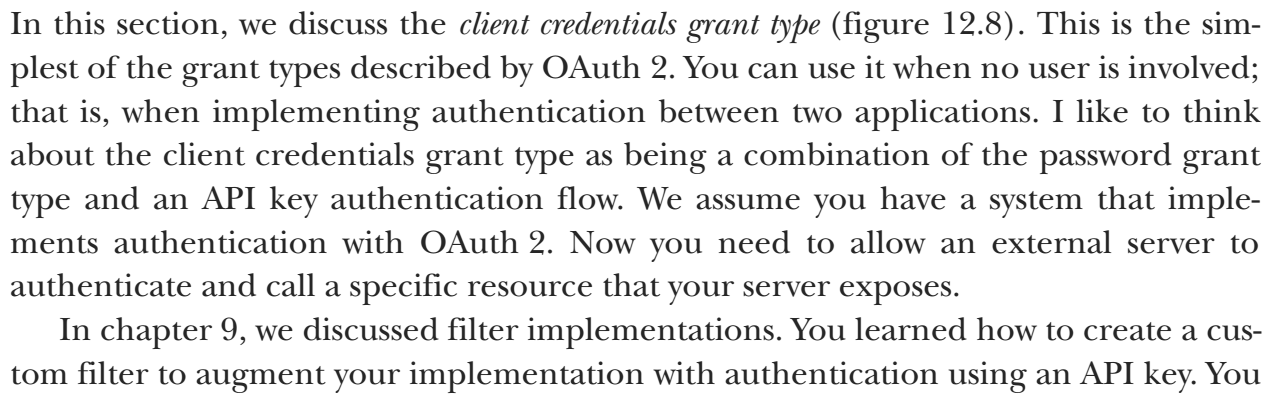
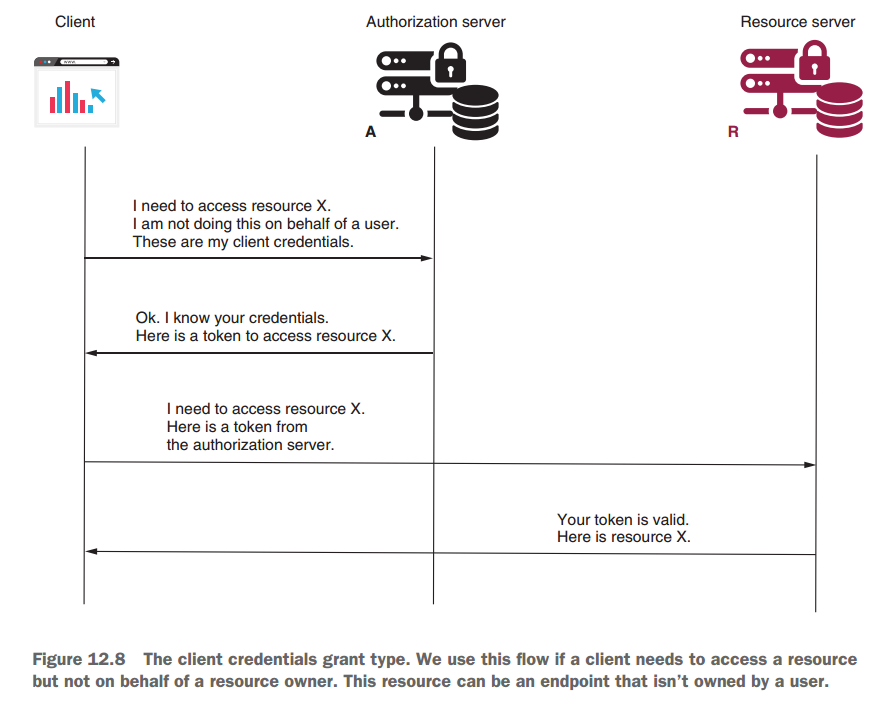
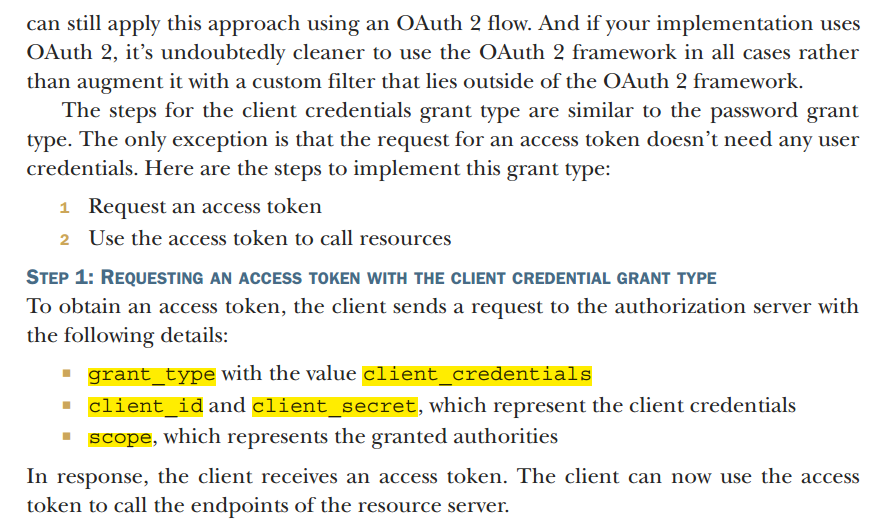
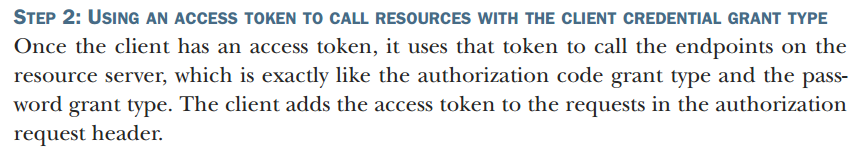
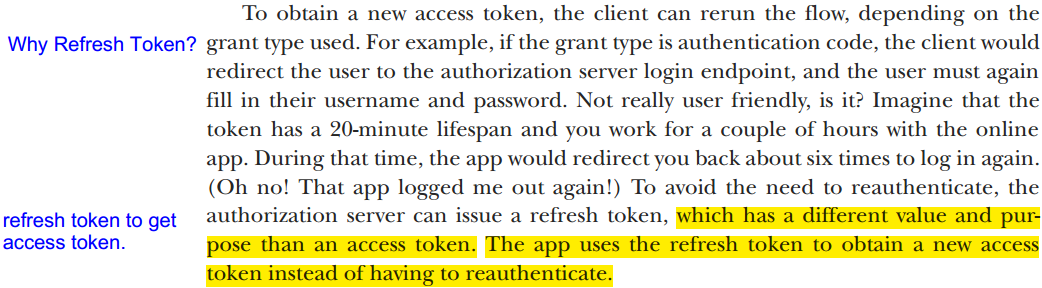
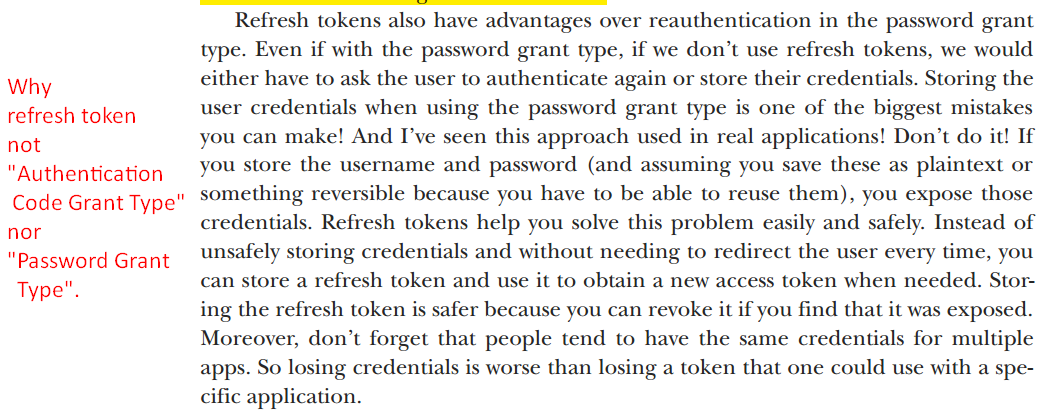
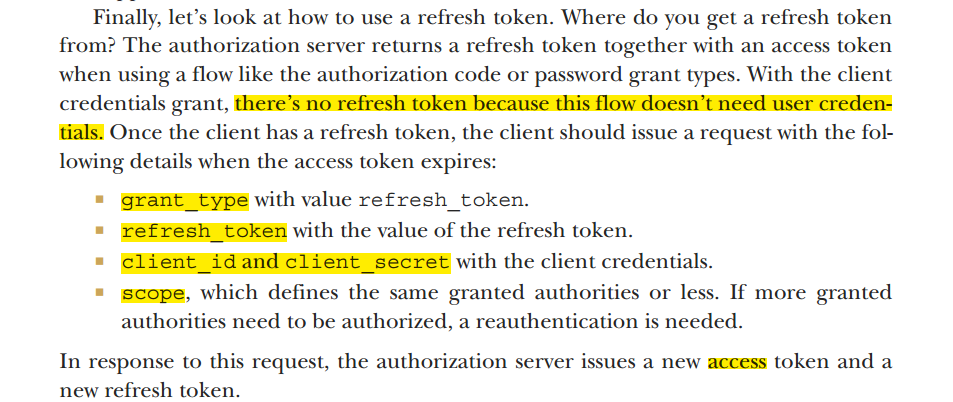
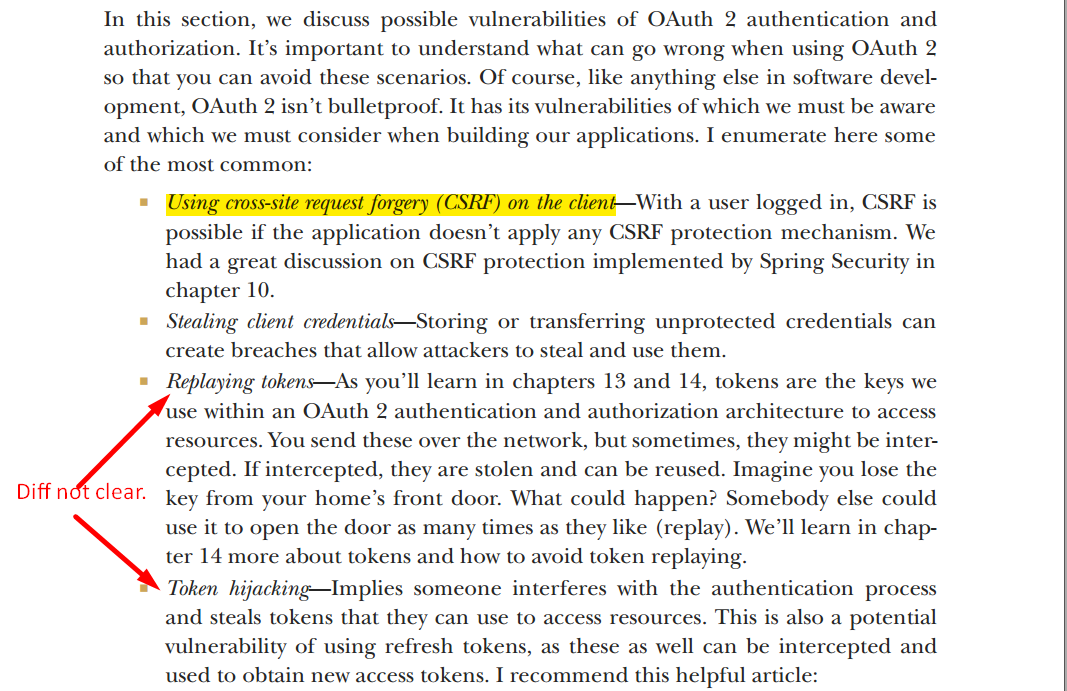
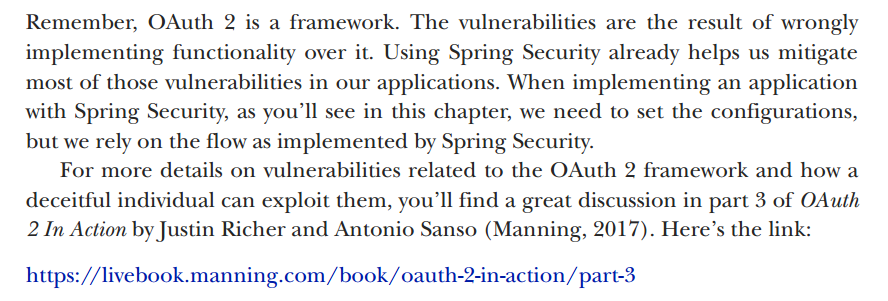
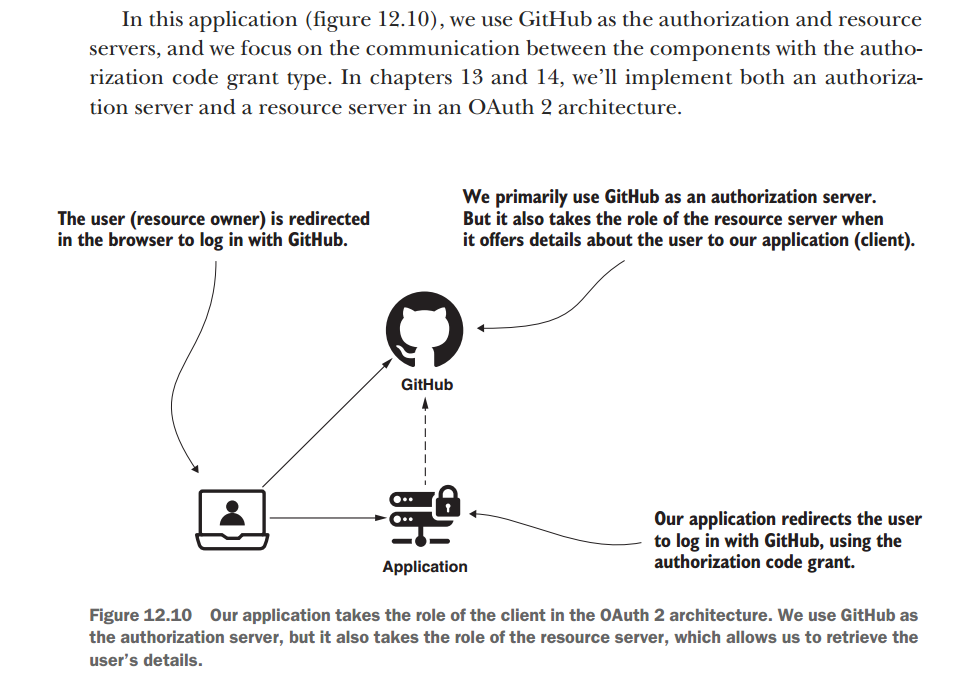
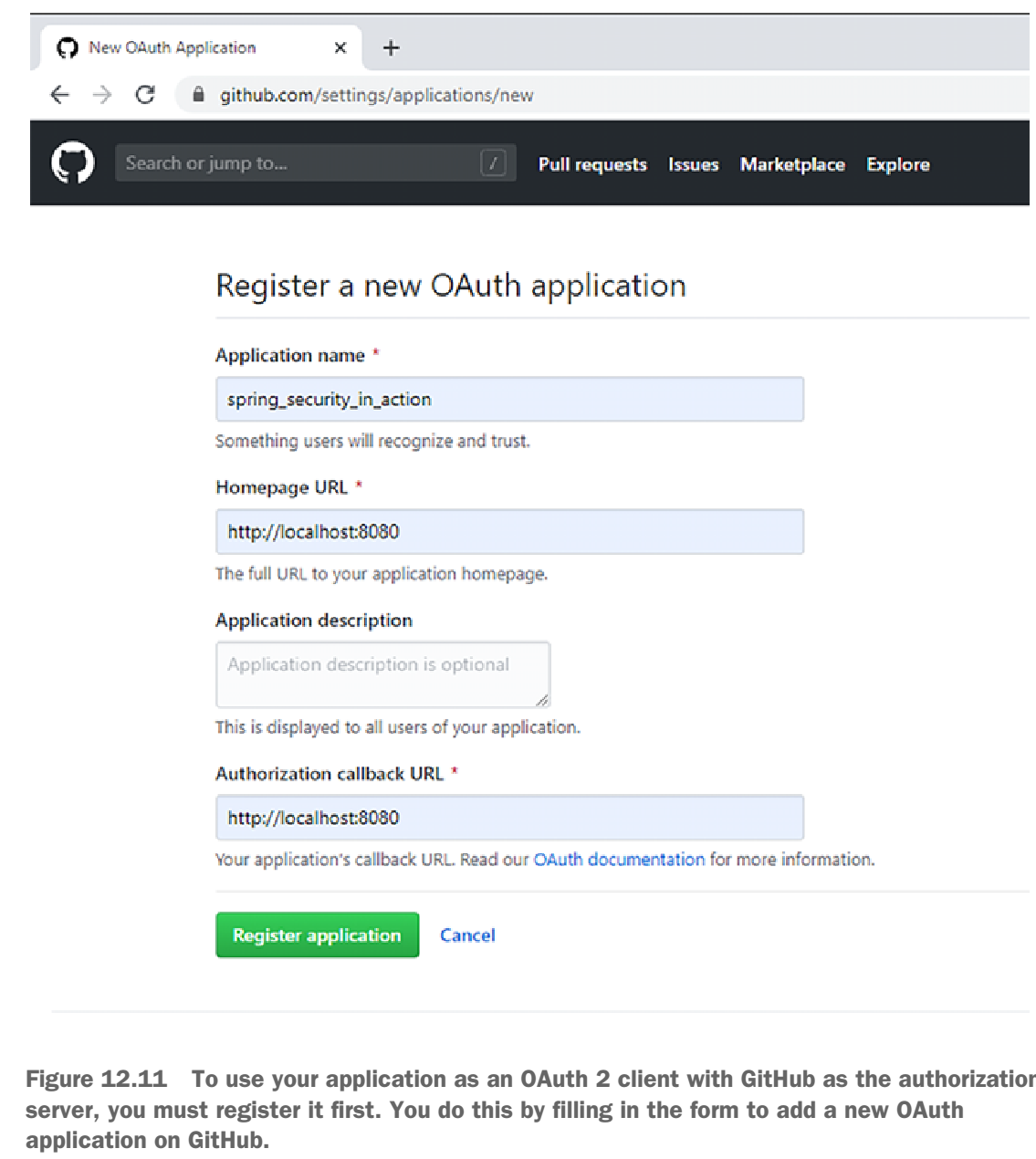
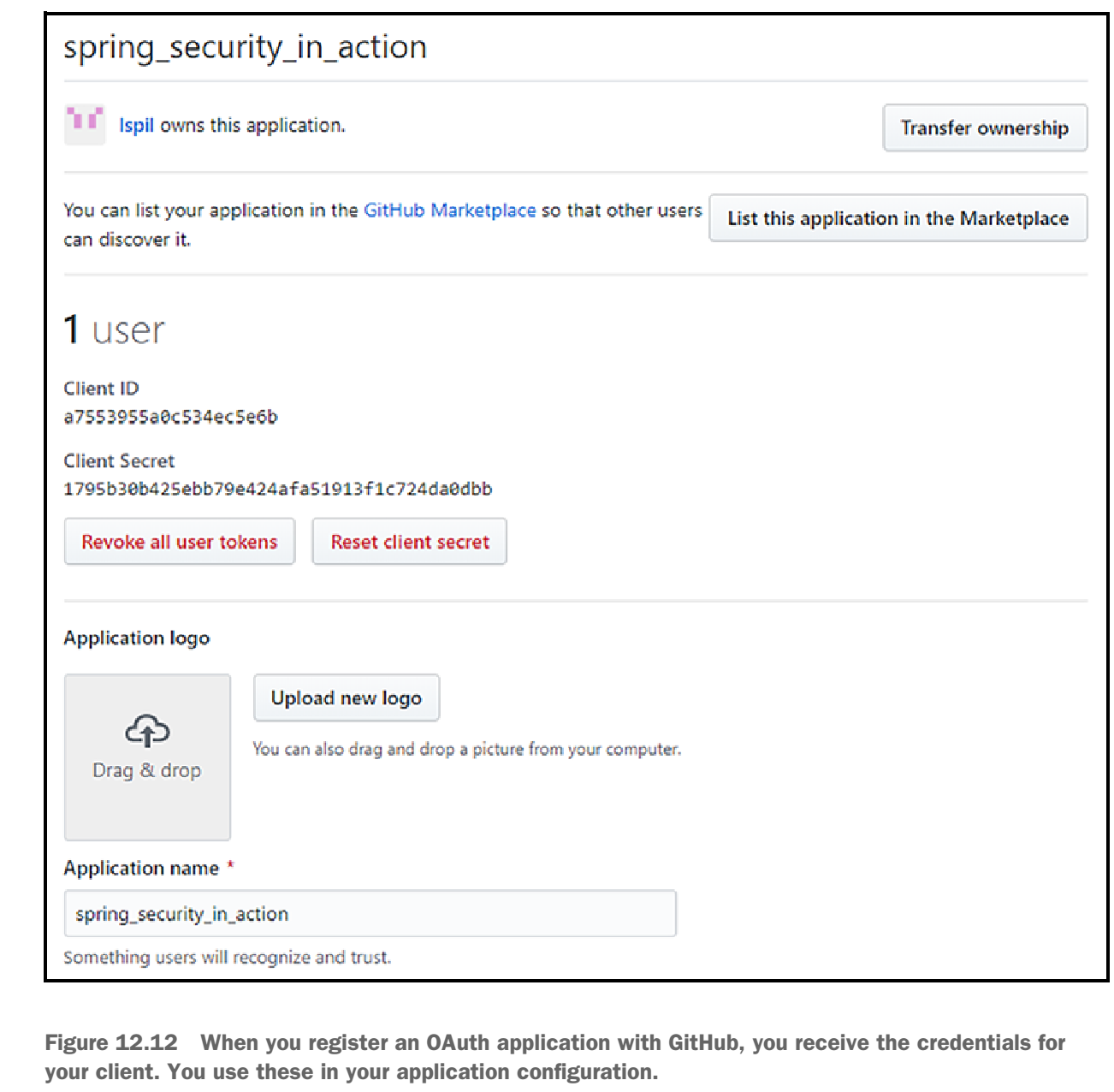
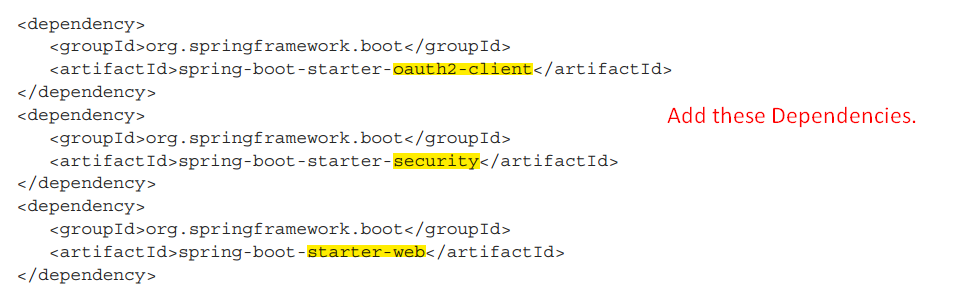
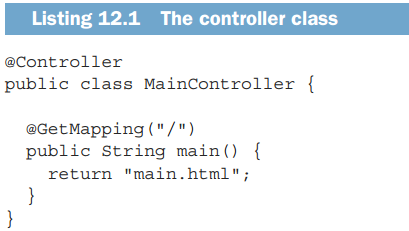
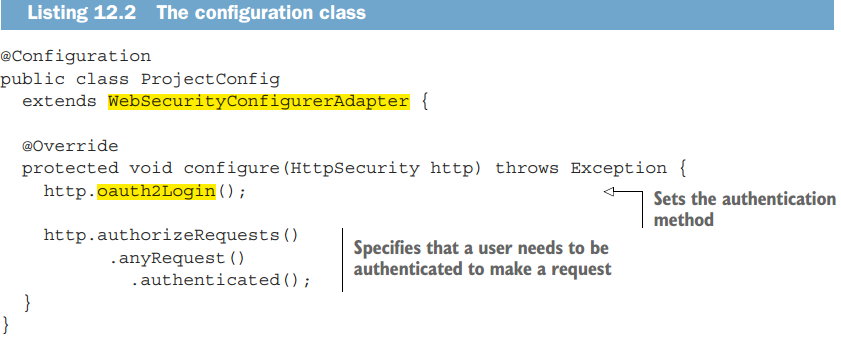
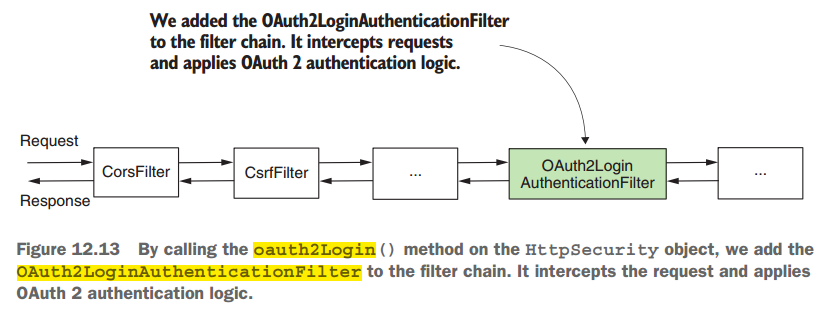
1. STEP 2: OBTAINING AN ACCESS TOKEN WITH THE AUTHORIZATION CODE GRANT TYPE

**NOTE**: The code obtained from the step 1 is the client’s proof that the user is authenticated.  
That is why it is called authorization code grant type.

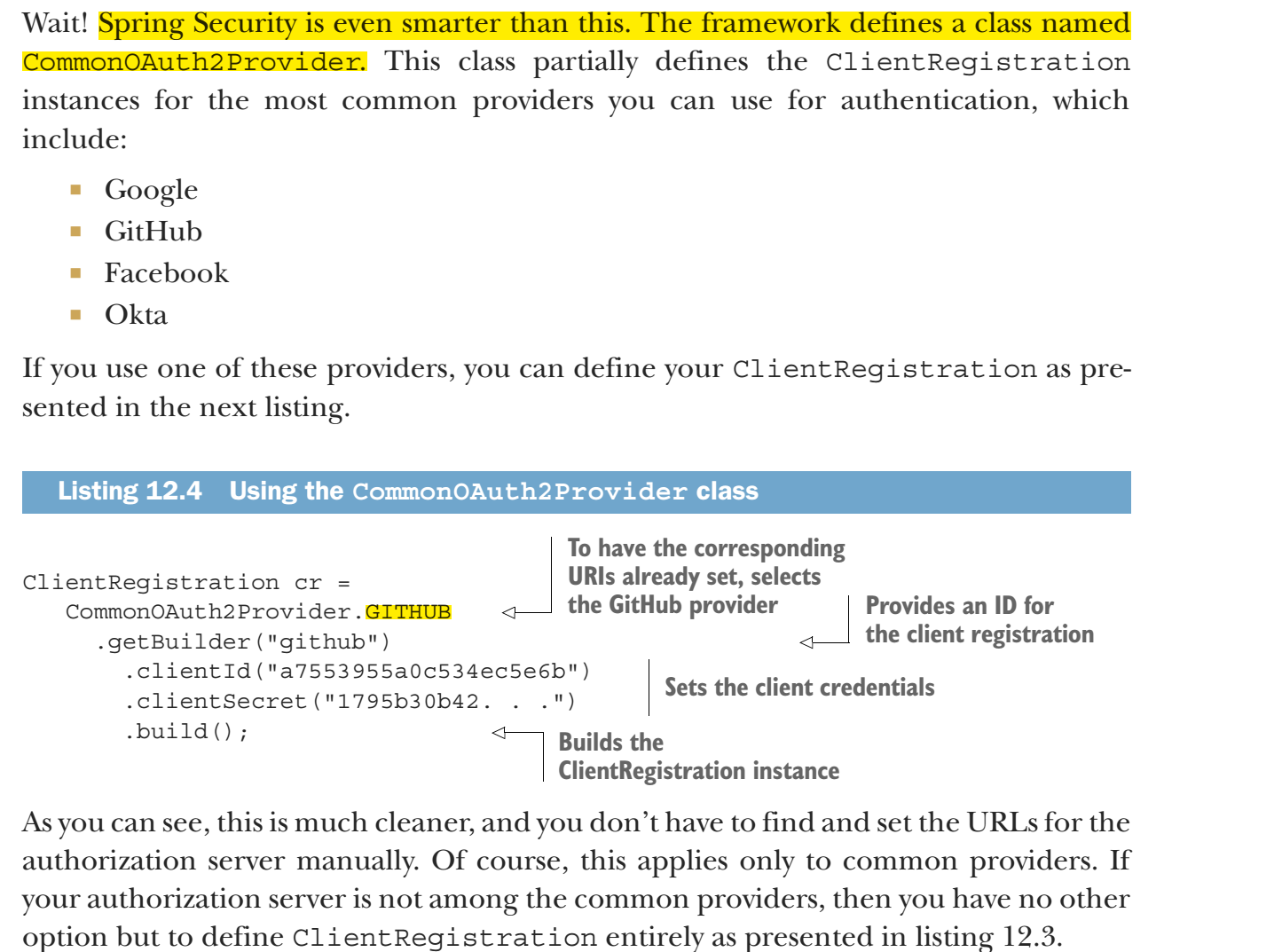
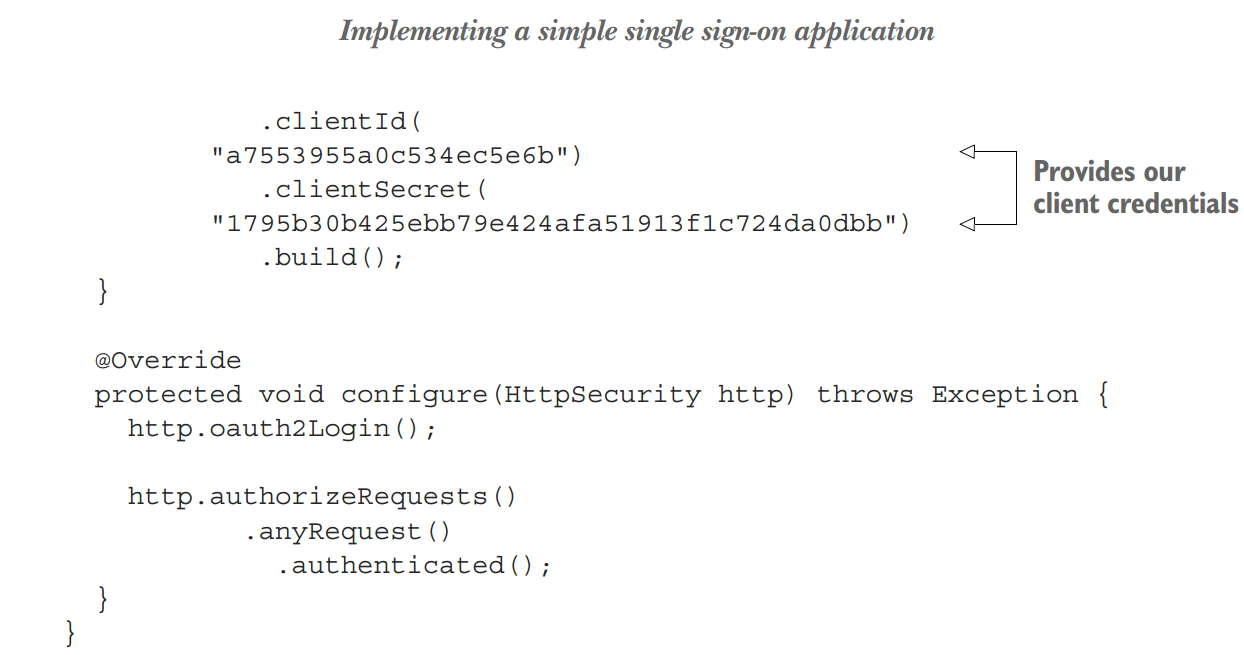
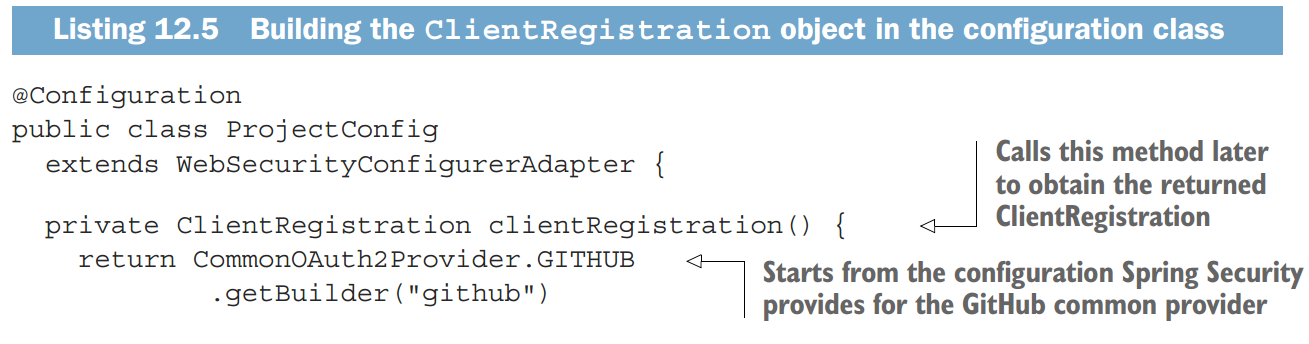
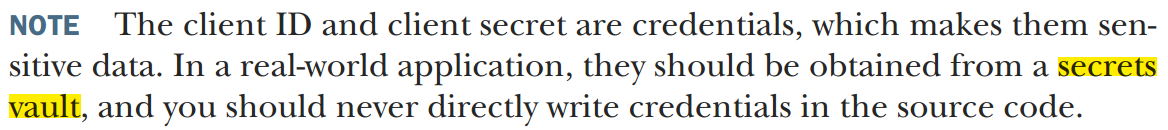
* 1. Now the client app calls the authorization server with the code to get an access token.  
     
  2. 

1. STEP 3: CALLING THE PROTECTED RESOURCE WITH THE AUTHORIZATION CODE GRANT TYPE

Now calling for the protected resources.

1. Client makes request by setting access\_token in the Authorization request header.
2. AN ANALOGY FOR THE GRANT TYPE AUTHORIZATION CODE
3. I end this section with an analogy for this flow.
4. 
5. 
6. *12.3.2 Implementing the password grant type*
7. Also called “Resource Owner Credentials Grant Type”.
8. Applications using this flow assume that the client collects the user credentials and uses these to authenticate and obtain an access token from the authorization server.
9. 
10. **Application of this Grant Type**:
    1. You use this authentication flow only if the client and authorization server are built and maintained by the same organization.
    2. Why?
    3. **Scenario** Where to use: Let’s assume you build a microservices system, and you decide to separate the authentication responsibility as a different microservice to enhance scalability and keep responsibilities separated for each service.  
       Here using “Authorization Code Grant Type” would redirect a user of your app (Web/Mobile) which is strange as both resource and authorization servers are from the same organization.  
       So, using “Password Grant Type” and presenting a login page to the user is best.
11. 
12. STEP 1: REQUESTING AN ACCESS TOKEN WHEN USING THE PASSWORD GRANT TYPE
13. 
14. STEP 2: USING AN ACCESS TOKEN TO CALL RESOURCES WHEN USING THE PASSWORD GRANT TYPE
15. Once the client has an access token, it uses token to call the endpoints on the resource server, which is exactly like the authorization grant type.   
    The client adds the access token to the requests in the authorization request header.
16. AN ANALOGY FOR THE PASSWORD GRANT TYPE
17. 
18. *12.3.3 Implementing the client credentials grant type*
19. **Application**: When no user is involved. When implementing authentication b/w two applications.
20. Simplest Grant Type.
21. 
22. 
23. 
24. 
25. *Using refresh tokens to obtain new access tokens*
26. Up to now, you learned that the result of an Oauth 2 flow, which we also call **Grant**, is an access token. But we didn’t say much about this token.
27. In the end, Oauth 2 doesn’t assume a specific implementation for tokens.
28. What you will learn now is that a token, no matter how it is implemented, can expire.
29. It is not mandatory - you can create tokens with an infinite lifespan - but in general, you should make these as short lived as possible.
30. If token doesn’t expire and it is stolen, you can imagine what would happen.
31. 
32. 
33. Jatin: First note that we use refresh token to avoid using user credentials in any way.   
    Now in Client Credentials Grant Type, user’s no credentials are involved so we don’t use refresh token.  
    
34. *12.4 The sins of OAuth 2*
35. **Agenda**: we will discuss possible vulnerabilities of Oauth 2 authentication and authorization.
36. It is possible to understand what can go wrong when using Oauth 2 so that you can avoid these scenarios.
37. Of course, like anything else in software development, Oauth 2 is not bulletproof. It has its
38. 
39. 
40. *12.5 Implementing a simple single sign-on application*
41. **Agenda**:
    1. Implement the first app of our book that uses the Oauth 2 framework with Spring Boot and Spring Security.
42. 
43. *12.5.1 Managing the authorization server*
44. **Agenda**:
    1. Implement Authorization Server.
    2. We will not implement our own authorization server but use an existing one 🡺 Github
45. As we will Github Authorization Server, so
    1. Out app will not manage its users.
    2. Any user having Github account can log into our app.
46. Oauth app (Client App) should register itself with Github Authorization Server to get client id and client secrete so that a client can authenticate and authorization server can identify the app.
47. using the following link, register.  
    <https://github.com/settings/applications/new>
48. When adding a new Oauth App, specify
    1. The name for the app.
    2. The link to which Github will make the call back to your app.
49. 
50. 
51. 
52. *12.5.2 Starting the implementation*
53. **Agenda**:
    1. Implement an SSO (Single Sign-On) App.
54. 
55. Create a web page to secure.
56. A controller and a simple HTML page that represents our app.
57. 
58. Let’s set the security configurations to allow our application to use the login with Github.
59. Let’s write configuration class.
60. 
    1. httpBasic(), formLogin() and oauth2Login() simply adds a new authentication filter to the filter chain.
    2. oauth2Login() adds filter 🡺 OAuth2LoginAuthenticationFilter.
    3. This filter intercepts requests and applies the needed logic for Oauth 2 Authentication.
61. 
62. *12.5.3 Implementing ClientRegistration*
63. **Agenda**:
    1. Implement the link b/w the Oauth 2 Client and the authorization server.
64. We need to establish that Github is our authorization server.
65. For this purpose, Spring Security defines the **ClientRegistration** Contract.
    1. **ClientRegistration** interface represents the client in the Oauth 2 architecture.
    2. For a client, define the following details.
       1. Client ID and secrete.
       2. Grant type.
       3. Redirect URI.
       4. Scopes.
67.   
    Above in the listing, we defined
    1. **Scopes**: Granted Authorities.
    2. Client Name, Secret, Registration ID of my choice.
    3. URLs of the authorization Server.
       1. Authorization URI
       2. Token URI: For access token & refresh token.
       3. User Info URI:  
          Doc to get URLs if the authorization server is not developed by you.

https://developer.github.com/apps/building-oauth-apps/authorizing-oauth-apps/

1. 
2. 
3. 
4. **ClientRepository**:
   1. By implementing this interface, we represent Oauth 2 Client for Spring Security.
5. *12.5.4 Implementing ClientRegistrationRepository*
6. **Brief**:
   1. :
      1. Instance for Spring Security to use for authentication.