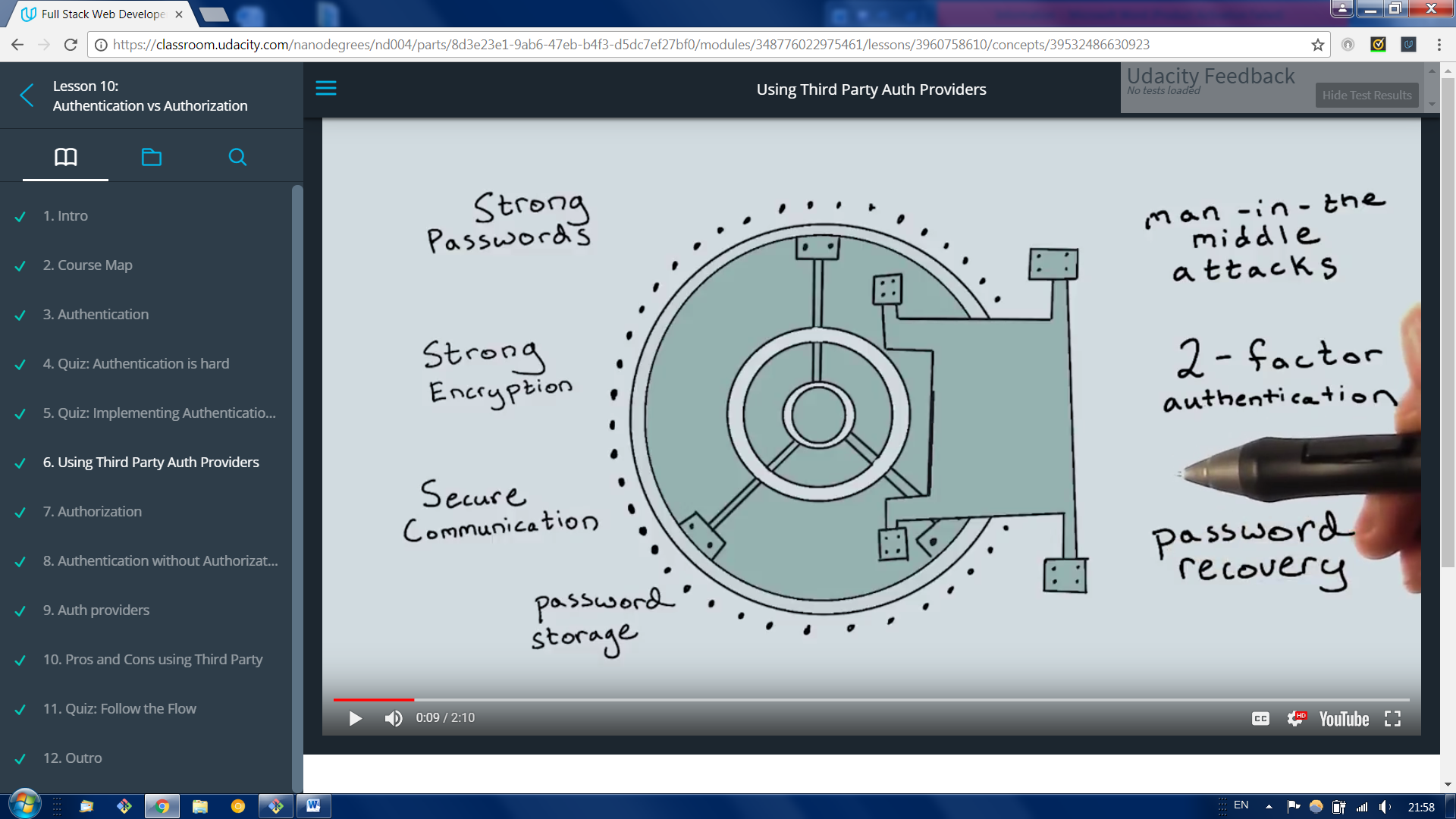
Authentication is the process of finding you are who you say you are (authenticity )



It is a lot of a security concerns , developers created free and open solution to this problems.

When authentication is successful we can move to another topic which is Authorization

Authorization checking if you have the right to access resource you need.

Auth providers

**OAuth** is an open standard for access delegation, commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords.[1] This mechanism is used by companies such as **Google, Facebook, Microsoft and Twitter** to permit the users to share information about their accounts with third party applications or websites.

<https://en.wikipedia.org/wiki/OAuth>

**Pros and Cons**

**Pros**

* Outsorce auth handling to Oauth providers
* Easier to register users ( the don’t have to fiil another form and store password to this application)

**Neutral**

* Users need to have a 3rd party account

**Con’s**

* User don’t rus you app/site with the data provided by third part Oauth provider (ex fecebook)
  + = keep aouth scope minimal and use only what is necessary for your app
* Limited/restricted internet access ( local authentication might be better in this case)
* Security provided by auth is good but you might need stronger yet

Creating google Sign in

Flow in web security refers to the way the information flows between client servers and Oauth providers.

Oauth2

1.Client side Auth Flow

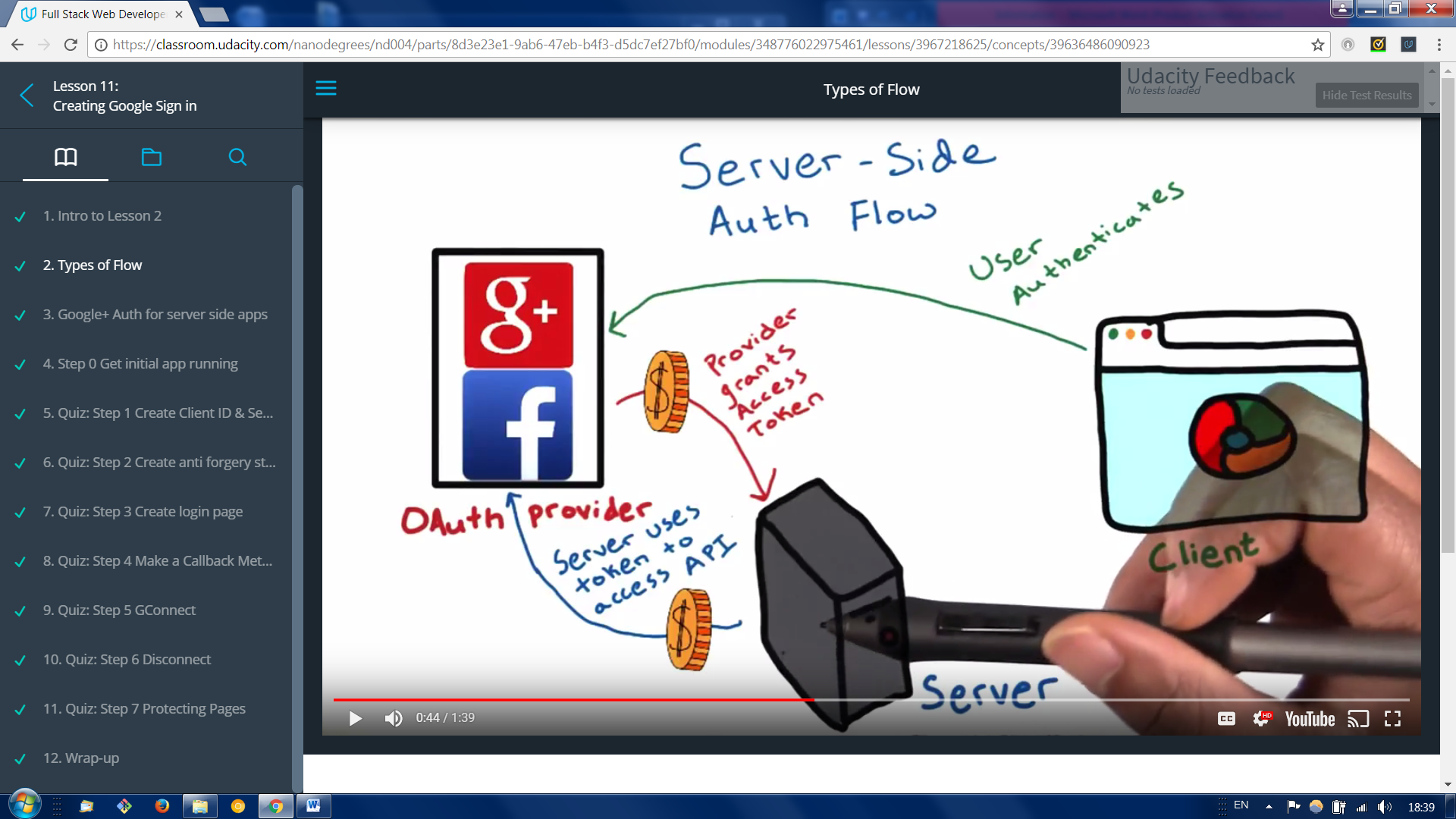
All the code is initiated from user browser side through java script this is useful for single page browser based web applications.

It is quick and easy but lot of trust is on browser or mobile device

2. Mobile authentications

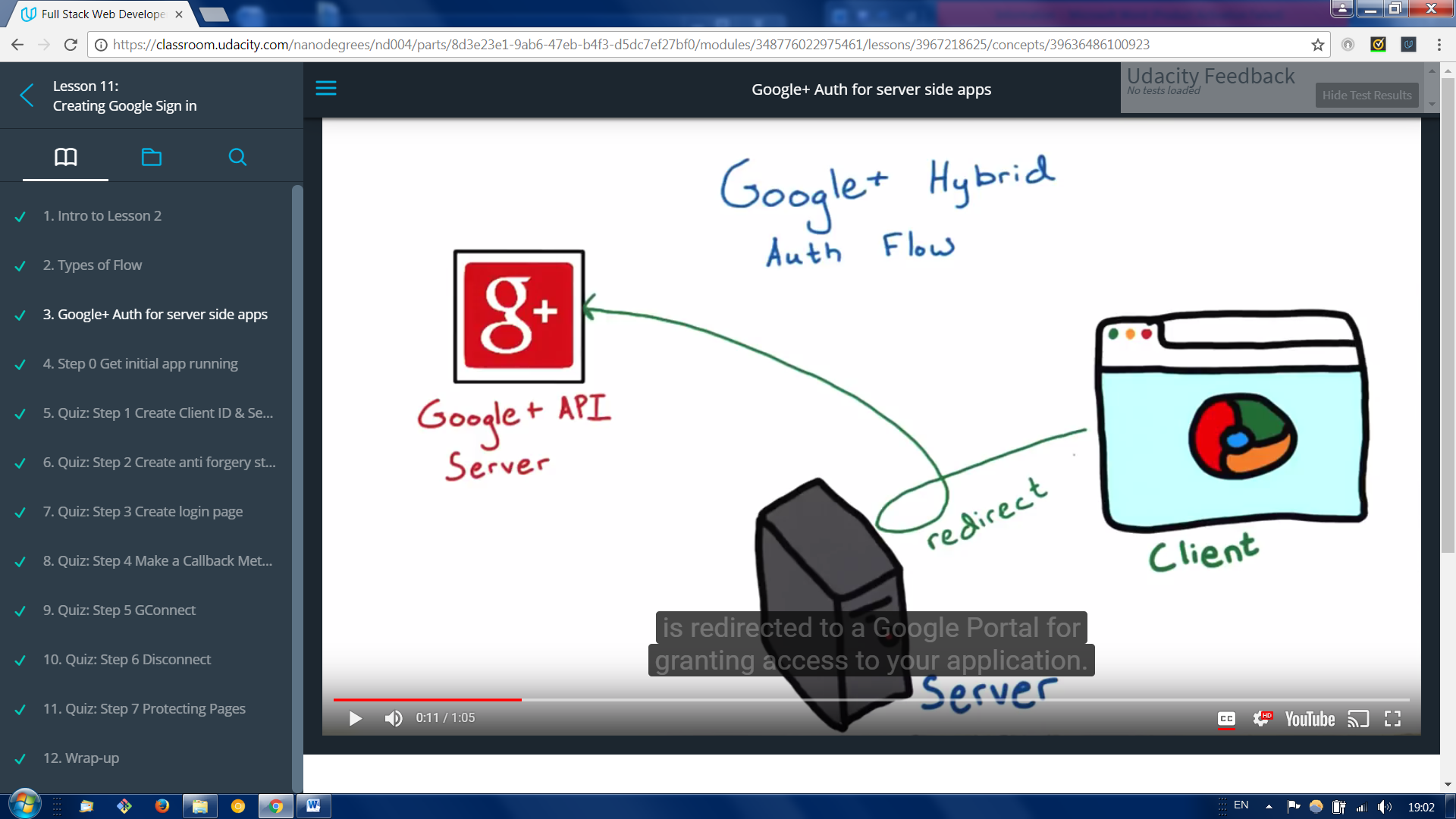
Mobile application can obtain authentication

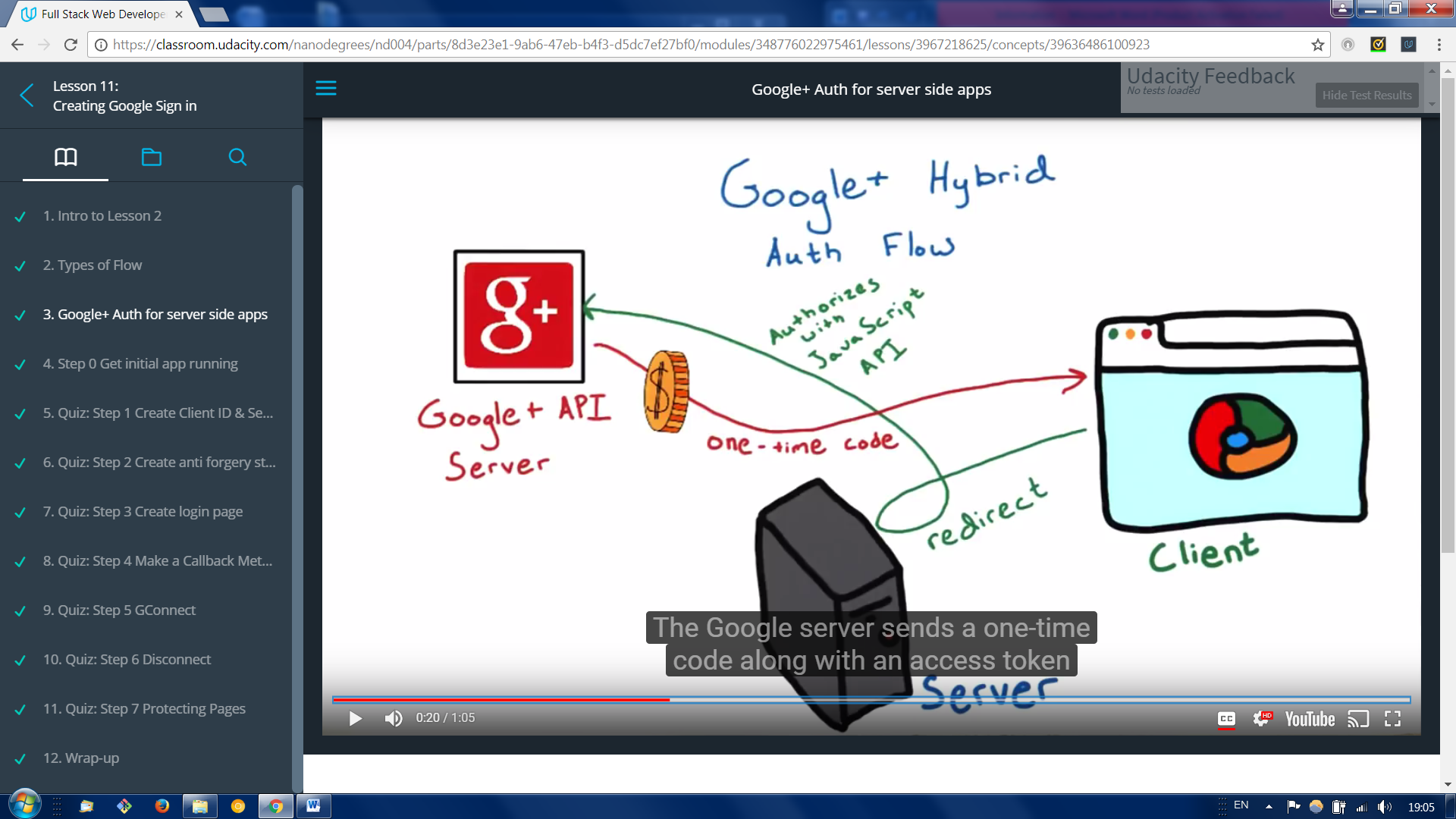
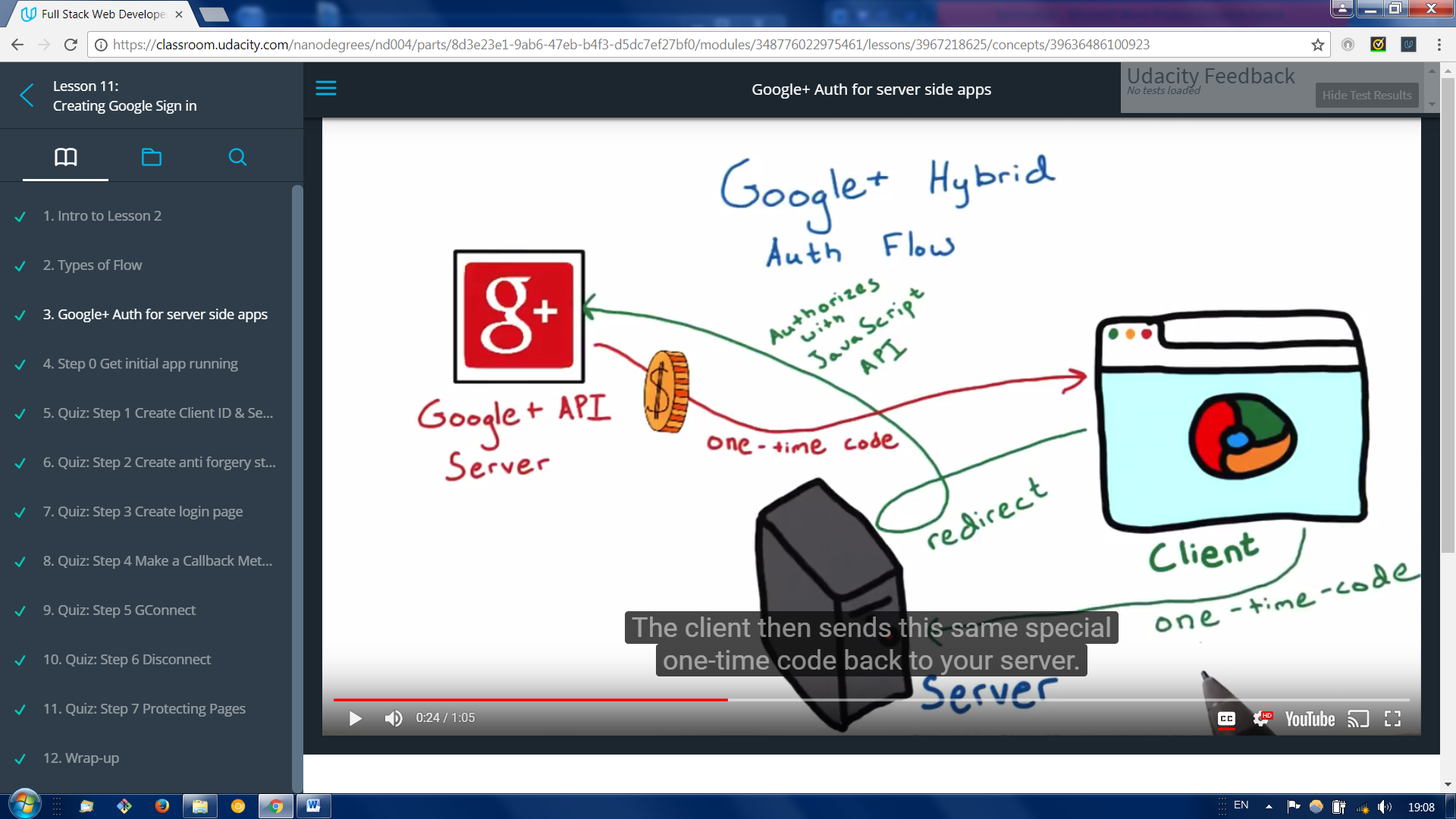
3. Server-side Auth Flow gives more power to the server based application, server now responsible for secure session checking and secure storage of this access tokens



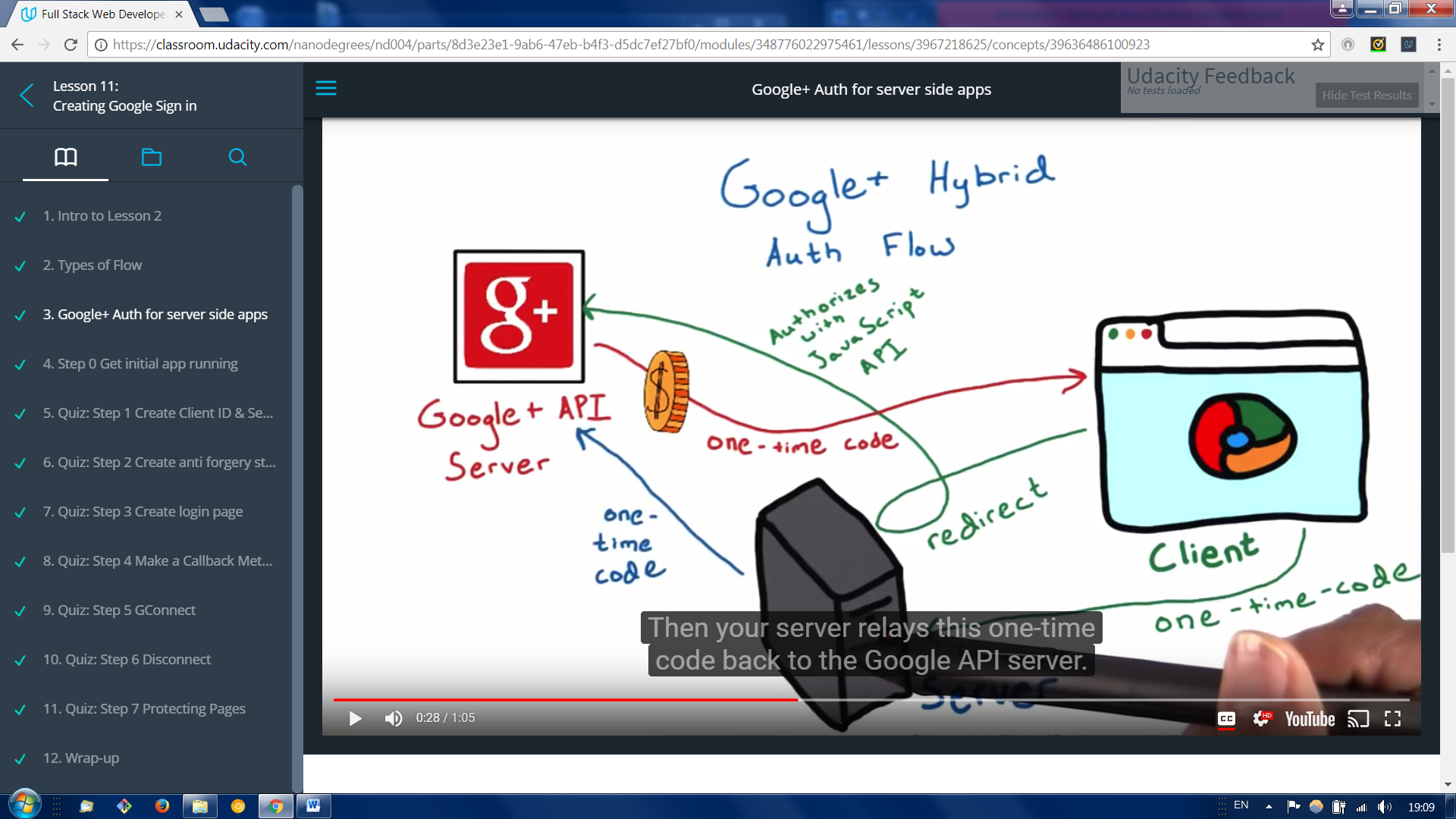
Server flow allows the server to obtain the access token to allow the server to make API requests on behalf of the user. The user has the option to set a time out or revoke access to these tokens at any time.

Google Auth use hybridized flow for loggins that requires authentication to happen on the client , but allows the server to make API calls on behalf of the client.

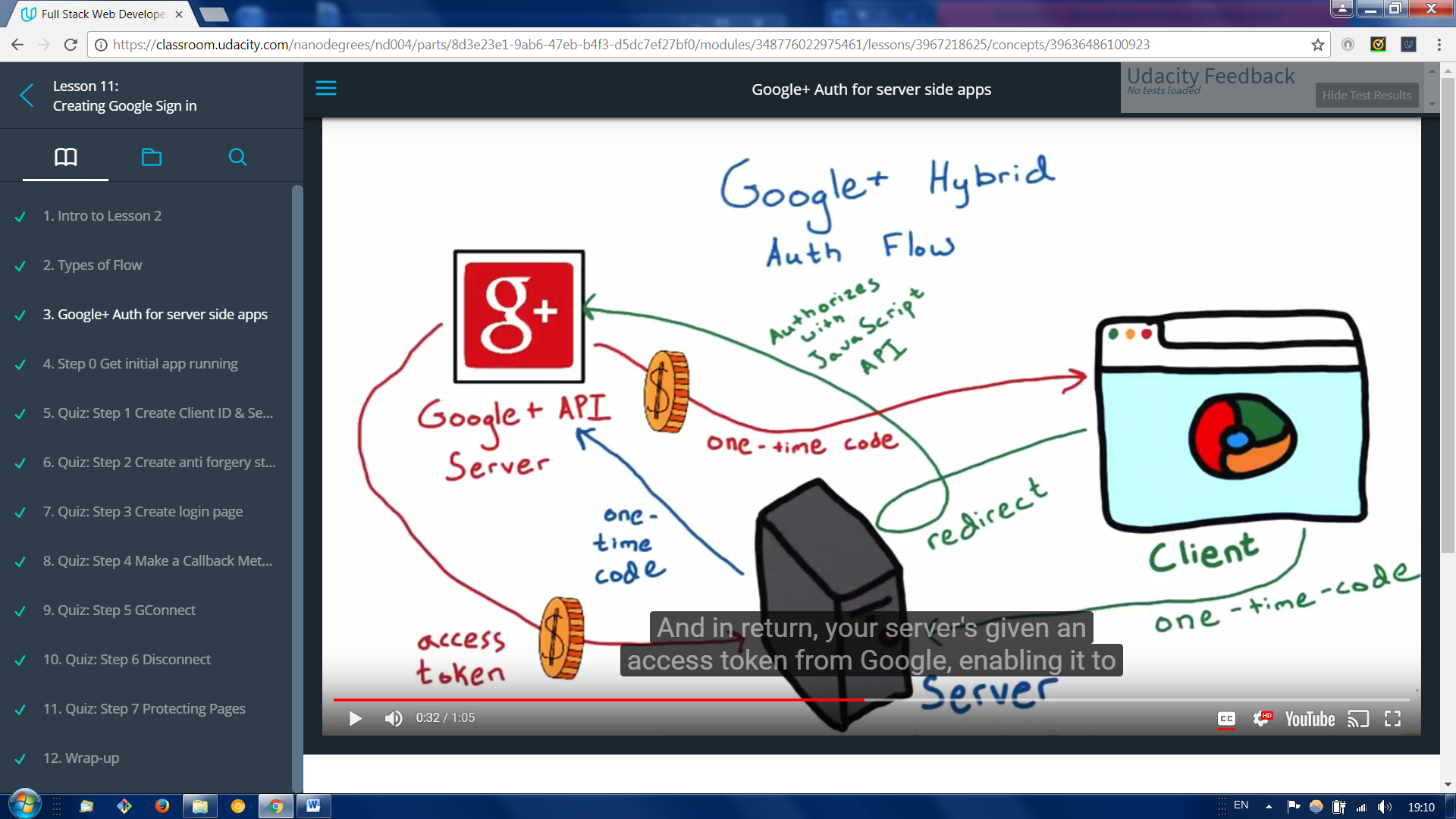


1. A user opts to log in with the Google account . and is redirected to google portal for granting access to your application. The user authorizes your app on the client side using the javaScript API client.
2.  The Google server sends a one- time code along with an access token back to the client.
3. 

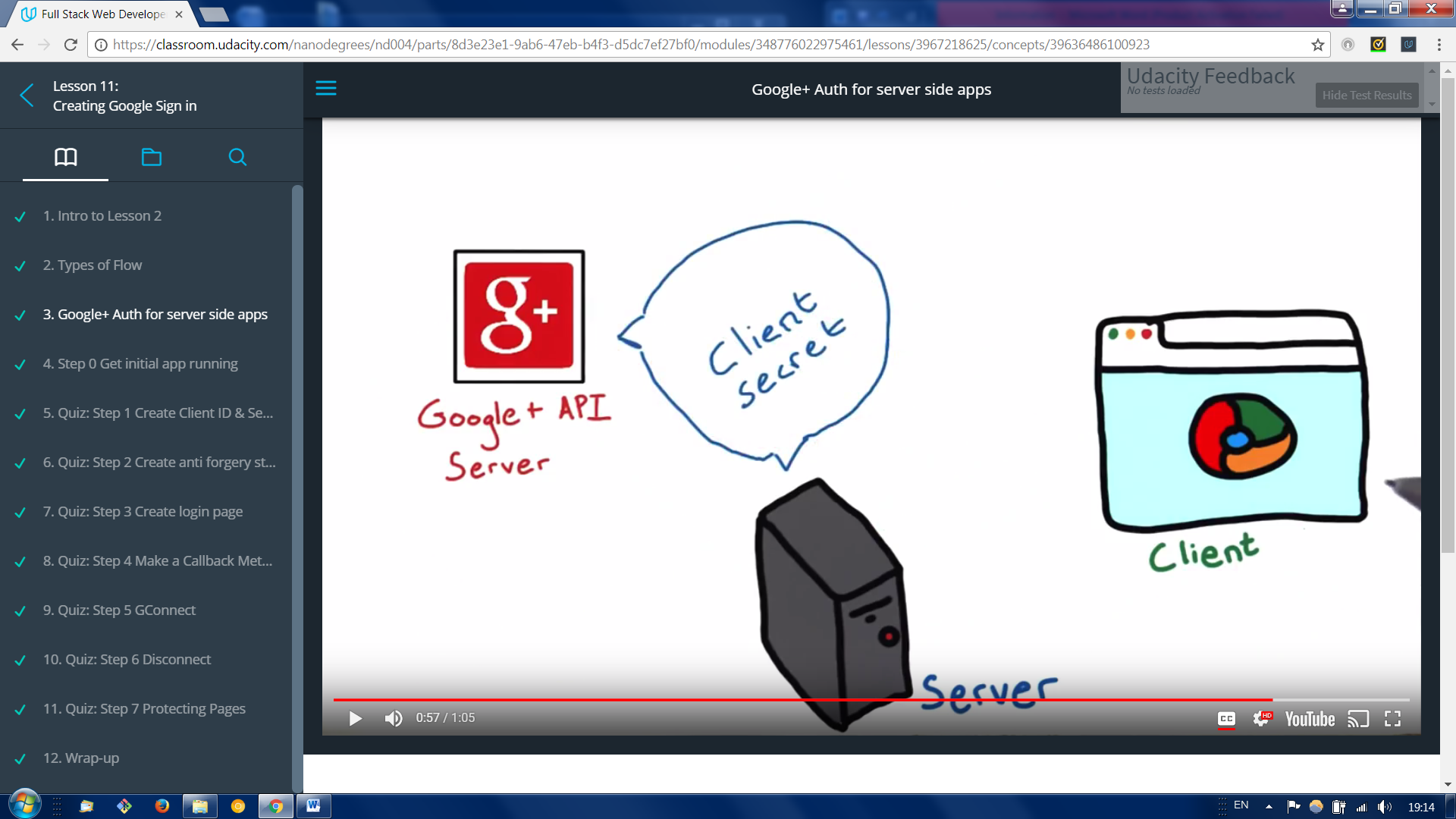
The client then sends this same special one-time code back to your server.

1. 

Then your server relays this one-time code back to Google API server

1. 
2. And in return , your server’s given an access token from Google, enabling it to make its own API calls, which can be done even when the user is offline.

This one-time code flow has a security advantage over a pure server side flow This is because, with one-time codes, Google provides tokens directly to your server, without any intermediaries. Even if one time code is discovered, it is extremely hard to use without your application’s client secret.



A client secret is a special code Google issues to verify your application.

2 Creating Login through Google API

**Step 1**

**Create Client ID & Secret**

<https://console.developers.google.com>

create new project , and select credential where you need to create new client ID , web application should be selected or specified as interface changed through this course.

We have to go through consent screen where we specify our email and product name .

Now in authorized javascript origins we have to specify our url address ( <http://localhost:5000> this is needed for our local version to work)

Now we have project called RestaurantMenu App with

client id: **652245020503-o4lvag05eho5uunqjocf7htpqbo3qahp.apps.googleusercontent.com**

Client secret : **N27nsg\_5btm4serq84TIlS49**

**Step 2**

**Create anti forgery state token**. We nee to be sure that it’s actually the user who came up with that request and it isn’t someone tricking them into sending it.

Anti forgery tokens protect the security of your users by preventing anti-forgery request attacks.

First step is to create a unique session token that your client side code returns alongside the Google generated authorization code. You will verify this unique session token with your server when a request is made to verify that the user is making the request and not a malicious script.

**Step 3**

**Create login button**

Google ID Tokens may contain the following fields (known as *claims*):

|  |  |  |
| --- | --- | --- |
| Claim | Provided | Description |
| iss | always | The Issuer Identifier for the Issuer of the response. Always https://accounts.google.com or accounts.google.com for Google ID tokens. |
| at\_hash |  | Access token hash. Provides validation that the access token is tied to the identity token. If the ID token is issued with an access token in the server flow, this is always included. This can be used as an alternate mechanism to protect against cross-site request forgery attacks, but if you follow [Step 1](https://developers.google.com/identity/protocols/OpenIDConnect#createxsrftoken)and [Step 3](https://developers.google.com/identity/protocols/OpenIDConnect#confirmxsrftoken) it is not necessary to verify the access token. |
| email\_verified |  | True if the user's e-mail address has been verified; otherwise false. |
| sub | always | An identifier for the user, unique among all Google accounts and never reused. A Google account can have multiple emails at different points in time, but the sub value is never changed. Use sub within your application as the unique-identifier key for the user. |
| azp |  | The client\_id of the authorized presenter. This claim is only needed when the party requesting the ID token is not the same as the audience of the ID token. This may be the case at Google for hybrid apps where a web application and Android app have a different client\_id but share the same project. |
| email |  | The user's email address. This may not be unique and is not suitable for use as a primary key. Provided only if your scope included the string "email". |
| profile |  | The URL of the user's profile page. Might be provided when:   * The request scope included the string "profile" * The ID token is returned from a token refresh   When profile claims are present, you can use them to update your app's user records. Note that this claim is never guaranteed to be present. |
| picture |  | The URL of the user's profile picture. Might be provided when:   * The request scope included the string "profile" * The ID token is returned from a token refresh   When picture claims are present, you can use them to update your app's user records. Note that this claim is never guaranteed to be present. |
| name |  | The user's full name, in a displayable form. Might be provided when:   * The request scope included the string "profile" * The ID token is returned from a token refresh   When name claims are present, you can use them to update your app's user records. Note that this claim is never guaranteed to be present. |
| aud | always | Identifies the audience that this ID token is intended for. It must be one of the OAuth 2.0 client IDs of your application. |
| iat | always | The time the ID token was issued, represented in Unix time (integer seconds). |
| exp | always | The time the ID token expires, represented in Unix time (integer seconds). |
| nonce |  | The value of the nonce supplied by your app in the authentication request. You should enforce protection against replay attacks by ensuring it is presented only once. |
| hd |  | The hosted G Suite domain of the user. Provided only if the user belongs to a hosted domain. |

data-scope="openid email" inside the login button in login.html