My lesson plan :

1 . Introduction to spring – security

2 . Explaining about : protection between two websites =>

**Passing credentials to third party applications**

One of the problem in a software security is how to make sure that system A is allowed to connect to system B. In other words, how to make sure that software applications are authorized to use one another. In case human to machine communication it’s fairly simple. User enters a website/mobile app and before doing anything it asks for valid credentials.

But what happens if we would like to have similar protection between two systems? To illustrate this problem, let’s suppose that we would like to login to [Dev.to](http://dev.to/) with GitHub account? (it’s possible now, but keep in mind that I don’t know how actually this integration works, I use it only to illustrate a **possible scenario**).

One solution for this problem would be to provide your GitHub credentials on [Dev.to](http://dev.to/) page. Then [Dev.to](http://dev.to/) would pass them (encrypted probably) to GitHub to check if they’re valid. Is it safe to show them to other system? Maybe that’s not the case for [Dev.to](http://dev.to/), but theoretically some untrusted party could save those credentials and mess around with them in future.

3 . Illustrate problems : after passing credentials to user :

And even if a middle man application is trusted one, there are still some use cases when such software would require to store user credentials.

For example, what happens when a GitHub user session expires? One solution would be to ask user once again for the credentials, but it’s not very user-friendly. Another would be to store credentials on [Dev.to](http://dev.to/) site and sent them if needed, which is even worse, which is a security violation.

Second case would be for a nightly jobs. Again, it’s maybe not the [Dev.to](http://dev.to/) case, but there are applications that do some batch jobs (e.g. process lots of data) during the night without user interaction. And if this job requires to fetch some protected data from other system it would need to store those credentials, which again should not happen

4 . Illustrate :

OAuth 2.0 flow (from user perspective) :

Going back to an example with [Dev.to](http://dev.to/) and GitHub, let’s say that I would like to check my repositories list on user information page (currently [Dev.to](http://dev.to/) doesn’t have this feature, but let’s say it has it). Therefore I go to settings page — <https://dev.to/settings>. It turns out that I’m not logged in so [Dev.to](http://dev.to/) redirects me to the login page.

During registration I’ve picked a GitHub option therefore now I also need to pick it. Now I’m redirected to the Github login page.

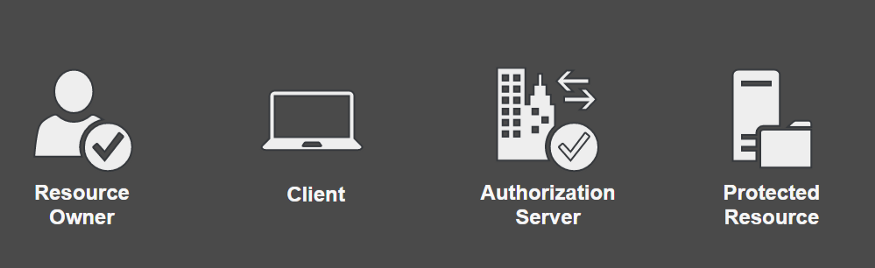
What’s important is that I’m now on GitHub page. When I’m checking the URL of a website in a browser, it’s no longer [dev.to](http://dev.to/), it’s [github.com](http://github.com/). So I could be sure that a credentials that I’ll provide in a second will not be seen by [Dev.to](http://dev.to/).

After providing my username and password I’m redirected back again to [Dev.to](http://dev.to/) website and I can do whatever I want there.

5 : Illustrate :

# OAuth 2.0 flow (Authorization grant, full picture)

* **Resource Owner** — usually it’s a person, a user. She/He grants access to *protected resource* to a *clients*.
* **Client** — it’s an application that on behalf of a resource owner would like to access a *protected resource*. An example of such client could be web browser, mobile or other third party application (in previous example it would be [*Dev.to*](http://dev.to/)).
* **Authorization Server** — is responsible for verifying the identity of a user and granting access tokens, credential needed to access *protected resource* by a client (I’ll explain what tokens are a little bit later).
* **Protected Resource** — is an application that a client wants to access (in previous example it would be *GitHub*). It authorized clients based on an access token provided by clients.



6. Illustrating what actually code is :

* client\_id - it's an identification of a client application, which tells authorization server (*GitHub* in this case) for which application access will be granted,
* response\_type - indicates a grant type (tells authorization how we would like to obtain access token, more about will be described going thru entire flow), a value here is code
* scope - represents a set of rights (what action or resource will be allowed by client to do) at protected resource, e.g. *read*, *write*, etc, in our example it's user,email,repo which means that [*Dev.to*](http://dev.to/) would like to have access to user, email address and repositories information,
* state - it's one time credential, usually randomly generated plain text, is necessary to match a client's request with authorization server's response
* redirect\_uri - after successful login, authorization server will redirect user to this page served by client application

# 7 . Explaining

# Refresh Token is ?

Up till now I’ve covered basic flow of the OAuth 2.0, which is obtaining the access token. In the part about JWT it was mentioned that tokens has an expire date, which means that after certain time they won’t be valid. And usually it’s after a short period of time, like couple of minutes.

To prevent client from asking resource owner again for credentials a new kind of a token was introduced — ***refresh token***.