OAuth2 Handshake

Setting up the OAuth2 handshake requires two steps: (1) You need to register your microservice as an OAuth2 client in *Drops*. To do so, you have to contact the administrator of the Heureka! architecture and please her / him to add your microservice to the *Drops* database. If you setup a development system, you are the administrator by yourself. In that case consider the description below. (2) You have to implement your part of the handshake.

The OAuth handshake implements only the **exchange of user data**. It does **not** implements a users session for your microservice. Thus, after implementing the OAuth handshake, you will have access to the user data and you can **use the libraries you know** to implement a **session for the user and your microservice**.

> The `Cookie` with the name `VCA_POOL_DROPS` is encrypted by *Drops*. Do **not** try to use this cookie! Instead, implement a session handled by your own application.

> This implementation just removes the question for consent of the user for sharing the personal data. So, clients libaries can still be used without additional code work.

Setup a microservice as OAuth2 client in *Drops*

Considering the discussion about the number of systems for one microservices, keep in mind that you normally have to register only one OAuth client for your microservice. No matter how many systems are part of your microservice.

1. You have to log into Drops as an administrator. Use the Heureka! console to to configure your user as an administrator.
2. Create the microservice as an OAuth2 client: Open the form using the menu (`Admin` > `Oauth Clients`) and enter an `ID`, a `Secret`, a `Redirect URL`, and a `Grant type` for the new service.

> The `ID` can be any unique identifier, for example the microservices name.

> The `Secret` should be known only to *Drops* and the new microservice. Thus, I would recommend to generate a key using KeePass enter it into the form, save it in a KeePass database, and enter it to the microservices deployment configuration.

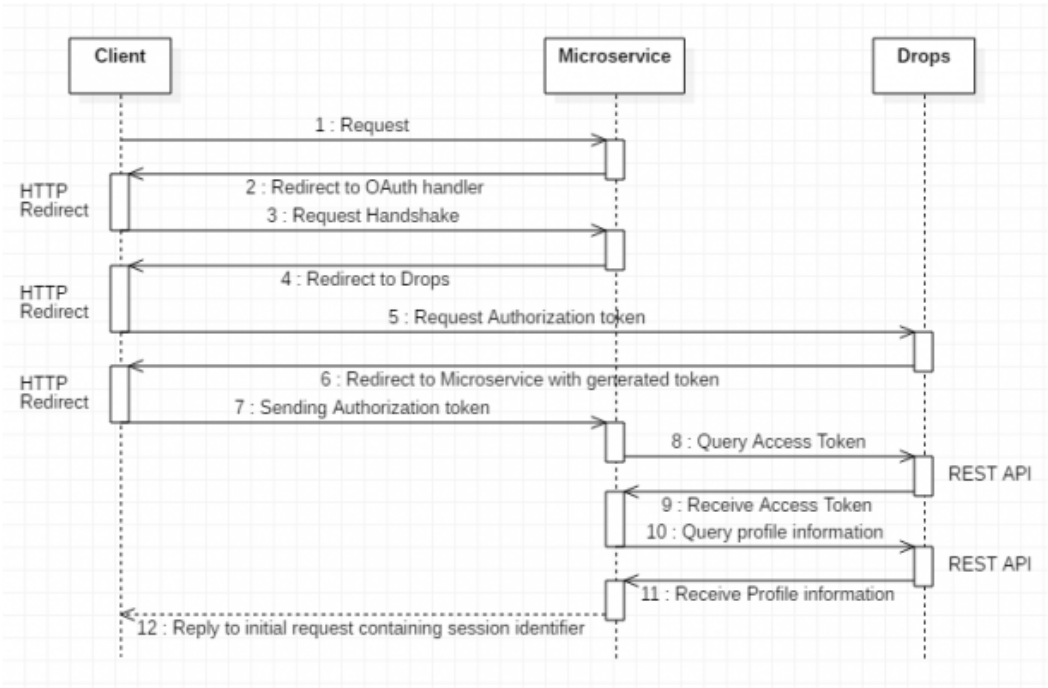
> The `Redirect URL` will be defined by the microservice developer and should be given the Heureka! architectures administrator. The given URL identifies the endpoint that is used by *Drops* to redirect the users client back, if the authorization code has been successfully created.

> The chosen `Grant types` define the possible authorization workflows possible between *Drops* and the microservice. Currently, *Drops* allows only `authorization code`.

Protocol flow

The RFC 6749 defines multiple possible interactions between clients and OAuth provider. A general workflow is defined in §1.2 of the protocol.

Drops implements the `authorization code` handshake. Thus, the client has to redirect to *Drops*, which redirects the user to the login page, if no session exists. Otherwise, *Drops* will validate the requesting microservice, generates an `authorization code`, and redirects back to the requesting microservice with the `authorization code` attached. Using this code, the service is able to request an `access token` that can be used to query information about *Drops* currently logged in user. At this point, the microservice is able to create its own user session. Handling of this additional user session should be synchronized with the *Drops* session, thus we implemented a so called `OAuth message broker`.



Endpoints

Implementation of the OAuth2 handshake requires to know the endpoints of *Drops*, but also to know which enpoints have to be implemented.

The following endpoints of *Drops* can be used:

- `drops.authorization.code = ${drops.url.base}/oauth2/code/get?client_id=${ID}&response_type=code&state=${any_context_string}&redirect_uri=${redirect_uri}&ajax=false`

- `drops.access.token = ${drops.url.base}/oauth2/access_token`
- `drops.get.profile = ${drops.url.base}/oauth2/rest/profile?access_token=${drops.access.token}`

> You have to replace the `${drops.url.base}` by the host and potentially path to the deployed *Drops* microservice.

There are some parameter to consider. First, to get an `authorization_code` *Drops* needs to identify your service. For this purpose, add the `ID` of your microservice and the `redirect_uri` are required in the query string. Furthermore, you can attach a `state` that will be returned to you, to encode some context information, like the current page of the user. Additionally, the optional boolean parameter `ajax` encodes, if the response should be `JSON` encoded in any case (including the case no user is currently logged in) or if Drops is allowed to redirect in some cases to the login page: `${drops.url.base}/oauth2/code/get?client_id=${ID}&response_type=code&state=${any_context_string}&redirect_uri=${redirect_uri}&ajax=false`.

The access token endpoint expects some query parameter: `grant_type`, `client_id`, `client_secret`, `redirect_uri`, and `code`. While the `grant_type` has to be the currently chosen one (e.g. `authorization_code`), the next three parameter identify the microservice and have to be the same as added to *Drops*. The `code` parameter has to contain the received `authorization code`.

Last, requesting the profile information requires to hold a valid `access token` that has to be attached to the request as a query parameter.

Additionally, you have to prepare an endpoint by yourself, that takes an `authorization_code` and initiate the next step using the `authorization_code`. *Drops* appends the `authorization_code` to the given `Redirect URL`, thus you are free to design your URLs.

Example endpoints: `https://ms.de/` (takes the code as part of the path), or `https://ms.de?code=` (expects the code as a query parameter with the name code).

> If you are implementing a frontend application that is using REST calls to communicate with a backend system, you need to set the parameter `ajax=true` to receive `JSON` in all cases (success and failure). Thus, you can handle the response by yourself.

> If you are running your microservice on another port than the Heureka! platform, *Drops* will throw some `CORS` errors (https://developer.mozilla.org/de/docs/Web/HTTP/CORS/Errors). You can solve the issue by adding your domain name and the chosen port to the allowed origins array of the Drops backend. Alternatively, you can add it to the configured server names.

Example

An example controller implemented using Play2 Framework and written in Scala could have the following functions:

```
package controllers

import javax.inject._

import models.AccessToken
```

```
import play.api._
import play.api.libs.json.Json
import play.api.mvc._
import play.api.libs.ws._
import play.api.Configuration

import scala.concurrent.ExecutionContext
import scala.concurrent.ExecutionContext.Implicits.global

class HomeController @Inject() (ws: WSClient,conf : Configuration) extends Controller {

  /**
   * Create an Action to render an HTML page with a welcome message.
   * The configuration in the `routes` file means that this method
   * will be called when the application receives a `GET` request with
   * a path of `/`.
   */
  def index = Action {
    Ok(views.html.index("Your new application is ready."))
  }

  def login = Action {
    val url = conf.getString("drops.url.base").get + conf.getString("drops.url.code").get +
      conf.getString("drops.client_id").get
    Redirect(url)
  }

  def receiveCode(code: String) = Action.async {
    val url = conf.getString("drops.url.base").get + conf.getString("drops.url.accessToken").get
    val clientId = conf.getString("drops.client_id").get
    val clientSecret = conf.getString("drops.client_secret").get

    val accessToken = ws.url(url).withQueryString(
      "grant_type" -> "authorization_code",
      "client_id" -> clientId,
      "client_secret" -> clientSecret,
      "code" -> code,
      "redirect_uri" -> "http://localhost:8080/endpoint?code="
    ).get().map(response => response.status match {
      case 200 => AccessToken(response.json)
      case _ => println(response.status);throw new Exception
                // Todo: throw meaningful exception considering the returned error message and status c
    })

    accessToken.flatMap(token => {
      val url = conf.getString("drops.url.base").get + conf.getString("drops.url.profile").get

      ws.url(url).withQueryString(
        "access_token" -> token.content
      ).get().map(response => response.status match {
        case 200 => Ok(
          Json.obj("status" -> "success", "code" -> code, "token" -> token.content, "user" -> response.
        )
        case _ => Ok(
          Json.obj("status" -> "error", "code" -> code, "token" -> token.content, "response-status" ->
        )
      })
    })
  }
}
```