Shared Session

The Heureka! architecture is used as a base for the collaboration support tool Pool[2]. Thus, almost all microserives implementing functional requirements need to identify the user. A unique authentication for every microservice implies a lot of problems: There are more possible problem sources, maintenance becomes hard to apply and a user needs to enter his or her credentials every time the user switch between microservices. The latter is serious usability issue and can hinder adaption. Therefore, the Heureka! architecture implements an OAuth 2 provider that can be used to share a users session between microservices.

The microservice *Drops* implements a secured session handling and an OAuth 2 provider that trusts all microservices that are deployed inside the Heureka! infrastructure. Therefore, it implements an OAuth 2 handshake using grant type Authorization Code that redirects to a OAuth client with an authorization code without asking the users permission. The OAuth client has to be part of the internal microservice network to receive an access token. Thus, the users data is kept save without asking for the users permission on each time the user switches between microservices.

Drops does not implement the original Authorization Code workflow. So, there is no way to receive an access token from without the internal microservice network.

Authentication of microservices becomes a critical security challenge for the Heureka! architecture. *Drops* can trust a microservice if and only if the microservice is hosted by the organization hosting the Heureka! architecture. Undoubtedly, microservices can implement security issues, but since the HeurekaA architecture addresses socio-technical organizations, we need to ensure clean implementation of microservices during a quality assurance process before deployment. For now, all microservices are deployed using a virtual network at the one server, so no external communication (using the internet or at least LAN) is needed. Thus, authentication is implemented using a naive microservice id and secret combination. This combination is send only for requesting the access token.

Ensure that the authentication data of your microservice is secret for each deployment (test, staging, live)! Additonally, do **not** save these data in any kind of client application, like javascript Web-Apps! These types of application will be discussed later.

OAuth 2 handshake

Drops allows the user to initiate a server session. A encrypted HTTP cookie is used to store all information needed to identify the user on server side. Additionally, *Drops* implements an OAuth2 provider. Thus, another microservice is able to `REDIRECT` to *Drops* in order to request an authorization code.

1. Request from `Client` to `Microservice`
2. `REDIRECT` from `Microservice` to `Drops`
3. `Drops` checks if the redirecting `Microservice` is trusted (the URL contains an identifier)
4. `Drops` generates the `Authorization code`
5. `Drops` redirects back to the `Microservice` OAuth2 endpoint (given as parameter AND saved in `Drops` database)

6. `Microservice` is able to initiate a direct REST-based communication to `Drops` and requests the `Access Token` without redirect
7. `Drops` responses with the users data
8. `Microservice` can initiate its own user session

Web Apps

Some microservices will implement a Web-App architecture. Such applications are based on HTML, CSS and Javascript. Mostly, there is also some backend implementing a RESTful API to save and synchronize data entered by users. If the microservice credentials are saved on client devices, our concept of trusted OAuth2 handshake has serious security issues since we are not able to prevent misuse of these credentials. Thus, the backend systems have to handle the trusted OAuth2 handshake. The question remains of how to implement the user session for the frontend. Short answer: You won't need a frontend session. The user session is needed for three purposes: (1) Control access, (2) use the user information for further handling, and (3) display the user. First, you have to keep in mind, that your and your users data is protected by the backend. So, if a user tries to request specific data using your RESTful API, your backend can grant or forbid access. Thus, if a user is entering spaces that are not allowed for this user, your backend won't send data, but it can send a status `403 FORBIDDEN` and an optional message. That way your frontend can handle the access control. Second, to handle the currently logged in user, your backend has to implement a special route that (1) is secured and (2) return the `UUID` of the user. Thus, the frontend can use the user in forms and other interaction elements. If your system has to display the user (case (3)), you should use widgets for user display prepared by *Drops*. In case your frontend is delivered by a special webserver with no connection to your backend system, you have to implement the trusted OAuth2 handshake for the delivering webserver to receive the users `UUID`.

OAuth Message broker

If the trusted OAuth2 handshake was successful, a users session can be established. An important requirement is to keep the session synchronized. That means, updates of the exchanged user information have to be cascaded, and also the logout event. For this purpose, the Heureka! infrastructure hosts a message broker system. *Drops* publishes these events:

```
type: LOGOUT
body: UUID of user
```

```
type: user.UPDATE
body: UUID of user
```

```
type: user.CREATE
body: UUID of user
```

```
type: user.DELETE
body: UUID of user
```

Your microservice can listen to these events for the purpose of synchronization.