AARHUS UNIVERSITET

Microservices and DevOps

DevOps and Container Technology

Security 101 - Authorization

Henrik Bærbak Christensen

AARHUS UNIVERSITET

Authorization

- Concerned with *ability to protect data and information from unauthorized access while still providing access to people/systems that are authorized*

- Authenticate actor
  - *Ok, you are Magnus*
- Authorize actor
  - *… and you have access to SkyCave*

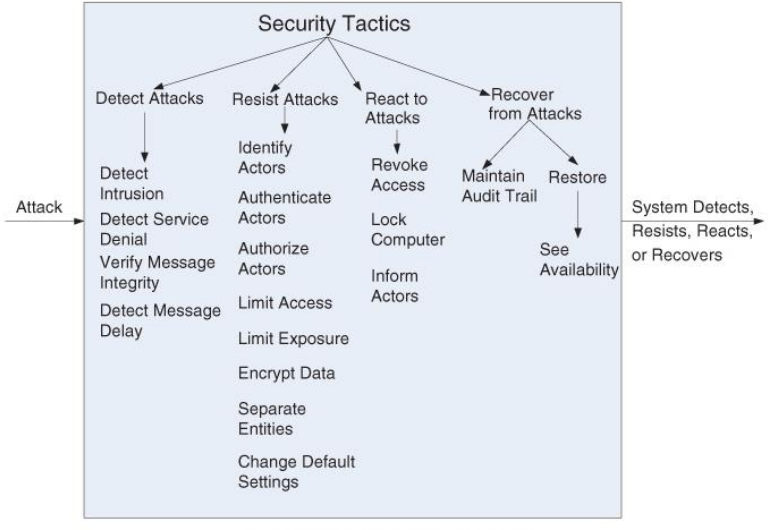
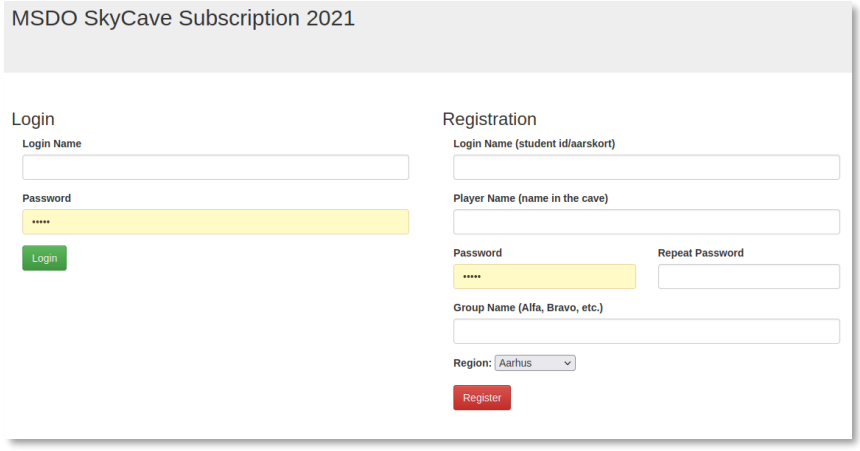


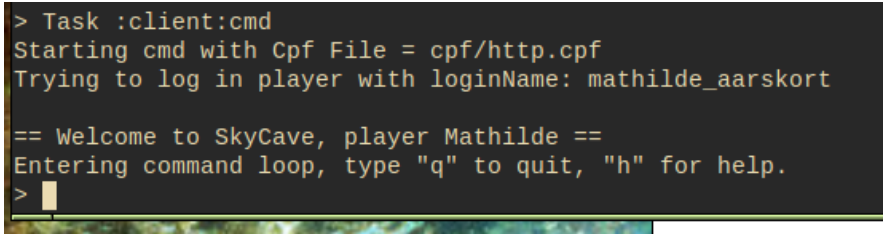
Figure 9.3. Security tactics

AARHUS UNIVERSITET

Our Backyard

- You register credentials in the course subscription service…
  - ‘Authenticate’
- The daemon asks it during cave login
  - ‘Authorize’





Architecturally

- Then our daemon service must talk to the subscription service to authorize ‘Mathilde’

- Simple, right?
  - Let Cmd send (loginName, password) to Daemon…
  - Deamon sends the (loginName, password) to AuthSrv and get a 200 OK or a 401 UNAUTHORIZED back?

MSDO until 2021

- This is actually exactly the way the SkyCave system operated up until 2021

You can verify your subscription using the SubscriptionService's RESTish interface by a GET request on a URL ala

```
http://cavereg.baerbak.com:7654/api/v2/auth?loginName=831720&password=fisk
```

- The daemon would make a GET request on the subscription service…

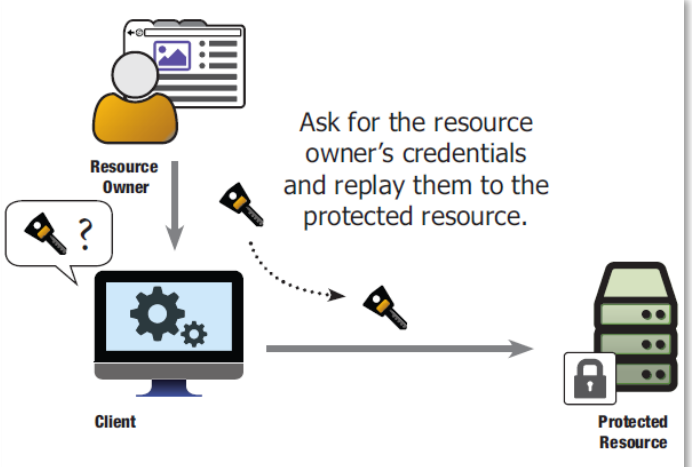


Figure 1.4 Ask for the resource owner's credentials, and replay them

AARHUS UNIVERSITET

The Issue

- The problem is: *Now Daemon has your credentials.* It can *impersonate* you in other contexts.

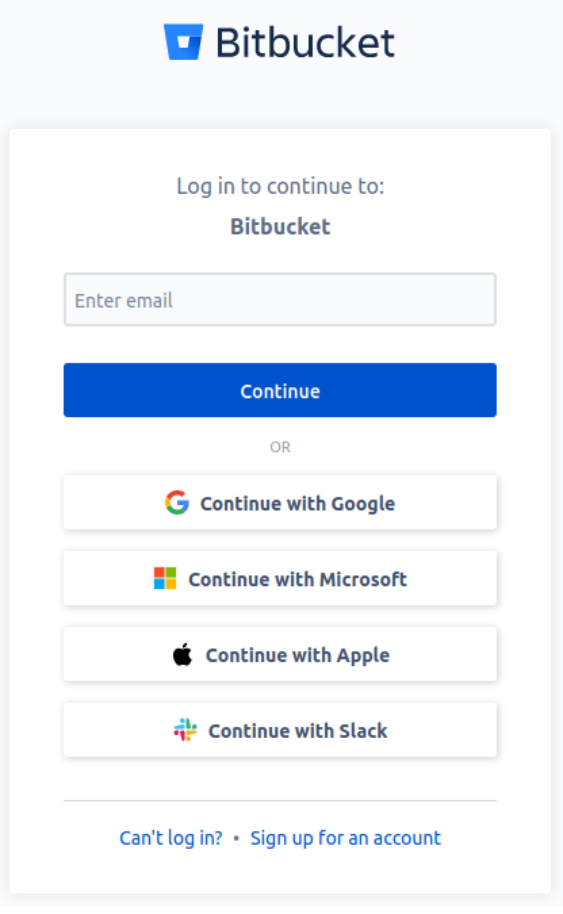
- In a growing web of services, it means your credentials are spread over an ever increasing set of services…
  - This does not really appear secure, does it?
- And you have to keep track of credentials for numerous services
  - Each having their own database of credentials
- ☹

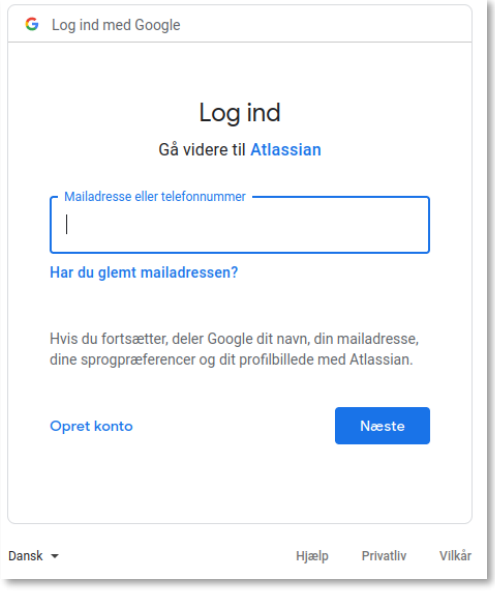
AARHUS UNIVERSITET

The Solution

- *Delegate* the authorization to a AuthServer
  - I.e. you provide credentials to the AuthSrv, **not the resource you want to access**
- The AuthSvr issues an **AccessToken** that is returned to the client (if you are authorized, of course)
  - Basically a unique ‘thing’ that states ‘you may use the resource’
- From now on, all communication client-to-resource includes that token
- The resource can always verify that the token is valid by requesting the AuthSvr
- And – it expires…

Example


Bitbucket
Log in to continue to:
Bitbucket
Enter email
Continue
OR
Continue with Google
Continue with Microsoft
Continue with Apple
Continue with Slack
Can't log in? • Sign up for an account


Log ind med Google
Log ind
Gå videre til Atlassian
Mailadresse eller telefonnummer
Har du glemt mailadressen?
Hvis du fortsætter, deler Google dit navn, din mailadresse, dine sprogpræferencer og dit profilbillede med Atlassian.
Opret konto
Næste
Dansk
Hjælp
Privatliv
Vilkår

AARHUS UNIVERSITET

(Privacy)

- As a sidenote, privacy is quite another quality attributes
  - If you use Google as AuthSrv, then Google knows exactly which services you access and when…

AARHUS UNIVERSITET

Authentication

The OAuth 2 Dance

It is a *protocol* !

And it is a HTTP protocol…

Terminology

- The terms takes a bit of getting-used-to…
- Resource Owner
  - That is me
- Protected Resource
  - That is my bank account
- Client
  - That is the bank's web site
- Authorization Server
  - That is NemID or …

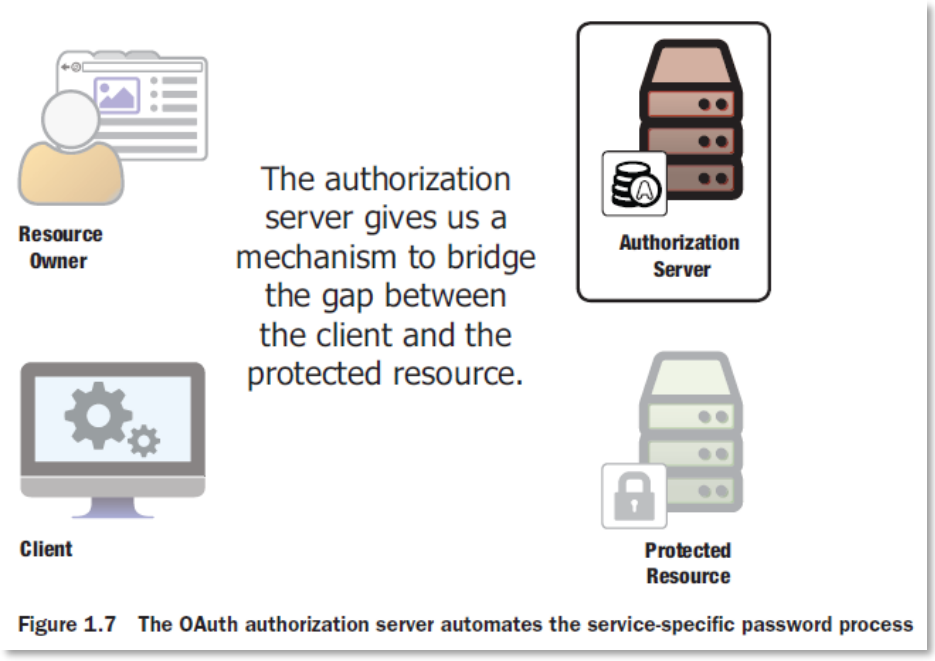


Figure 1.7 The OAuth authorization server automates the service-specific password process

Exercise

- I need access to my BitBucket account
- Who is who here?

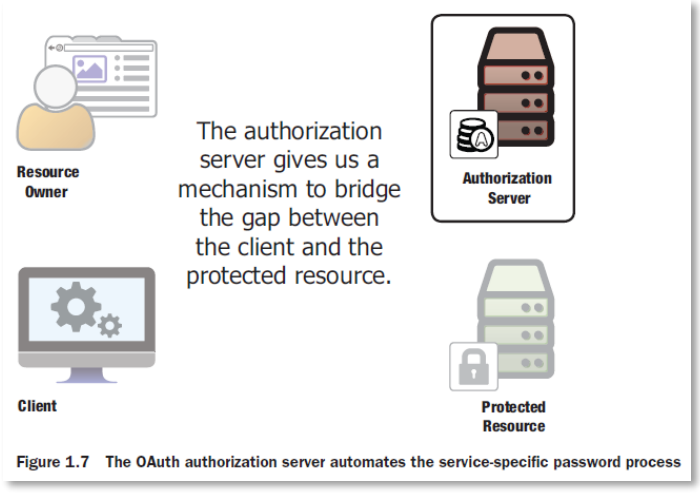
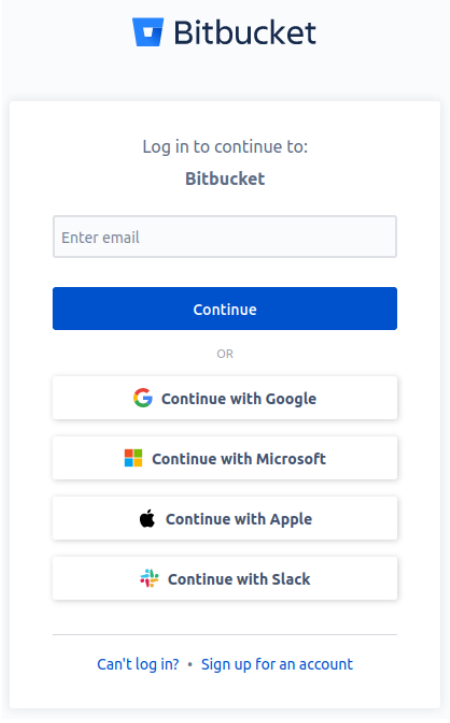
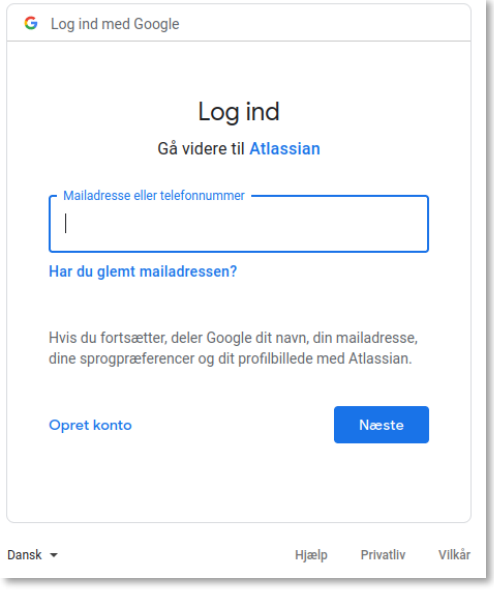


Figure 1.7 The OAuth authorization server automates the service-specific password process





Helicopter Perspective

- Overall…

1. The Resource Owner indicates to the Client that they would like the Client to act on their behalf (for example, "Go load my photos from that service so I can print them").
2. The Client requests authorization from the Resource Owner at the Authorization Server.
3. The Resource Owner grants authorization to the Client.
4. The Client receives a Token from the Authorization Server.
5. The Client presents the Token to the Protected Resource.

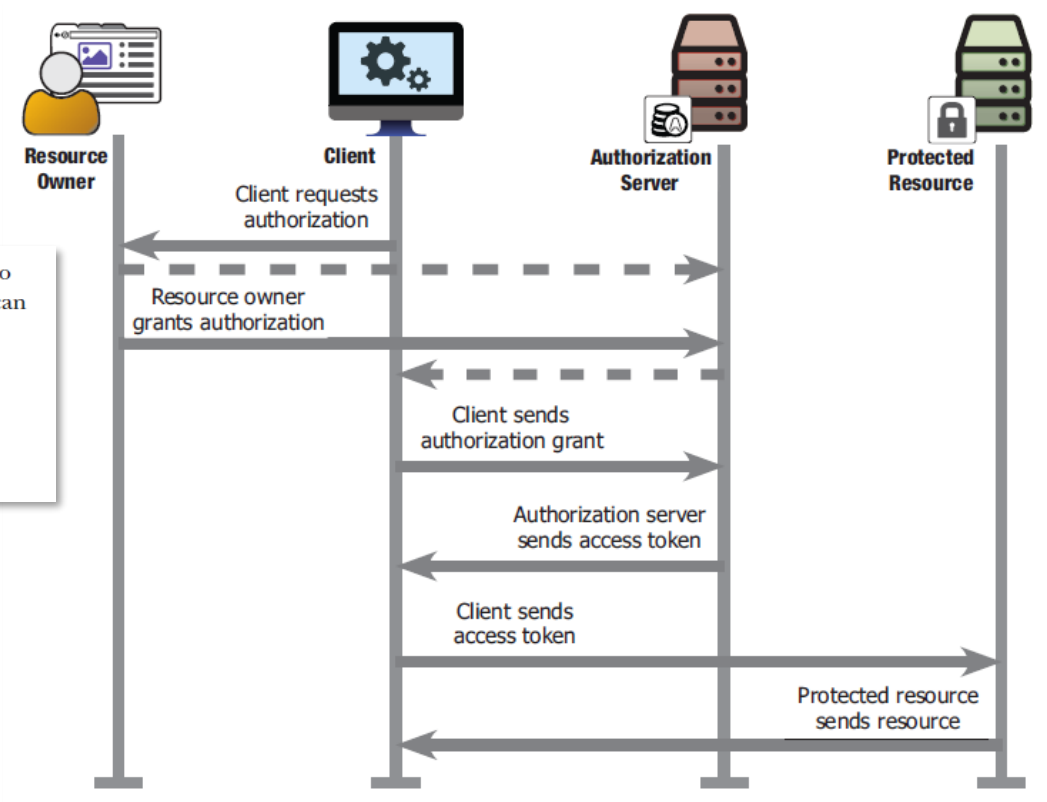


Figure 1.8 The OAuth process, at a high level

- And the details…

- Note the *redirects*…
  - OAuth is a protocol relying on the web and HTTP

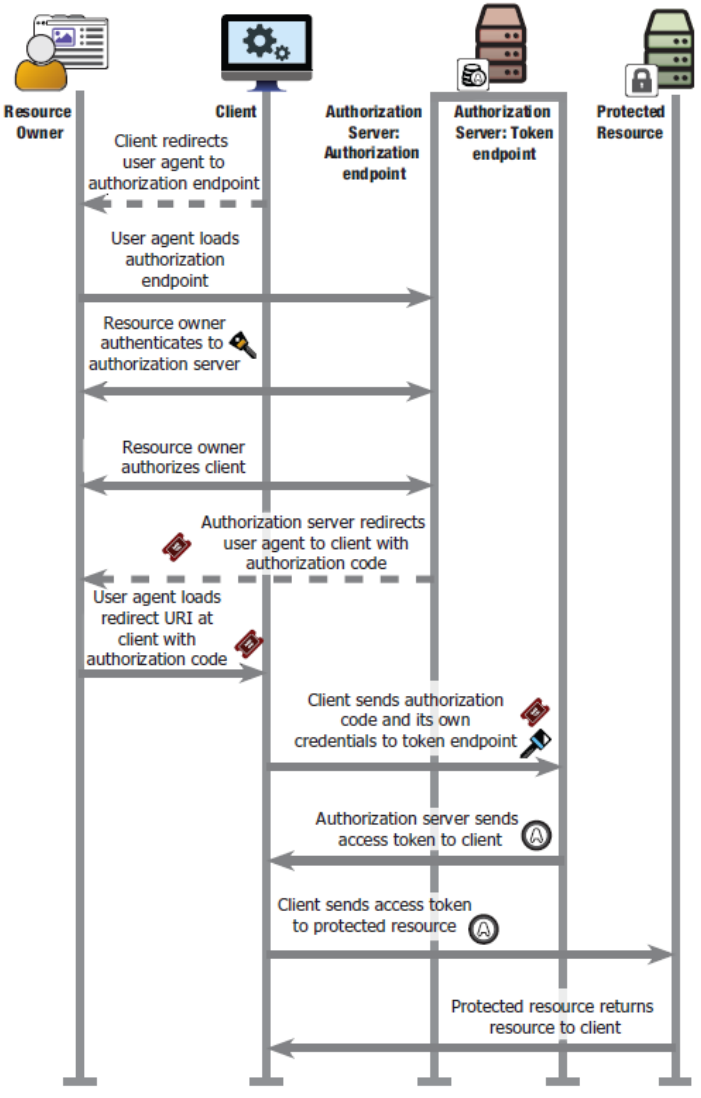


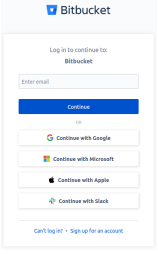
Figure 2.1 The authorization code grant in detail

The Details

AARHUS UNIVERSITET

Authorization Grant

- The OAuth Dance
  - Press ‘login’ make a *redirect* as answer
  - Forcing my browser to …



In the code:
9000 is the client
9001 is the AuthSvr
9002 is Protected Resource

```
HTTP/1.1 302 Moved Temporarily
x-powered-by: Express
Location: http://localhost:9001/authorize?response_type=code&scope=foo&client
_id=oauth-client-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&
state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1
Vary: Accept
Content-Type: text/html; charset=utf-8
Content-Length: 444
Date: Fri, 31 Jul 2015 20:50:19 GMT
Connection: keep-alive
```

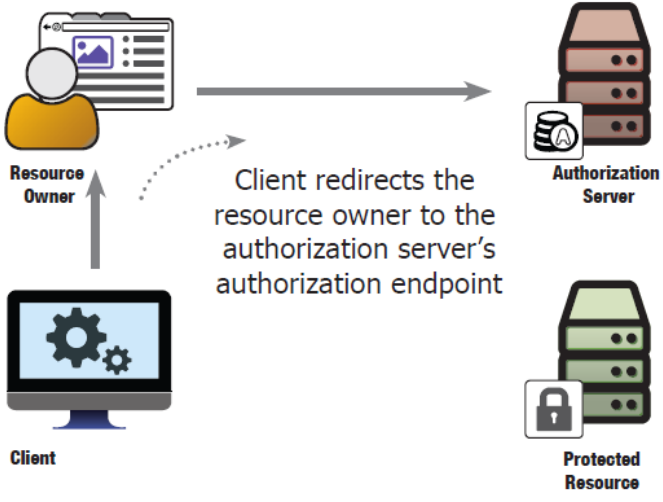
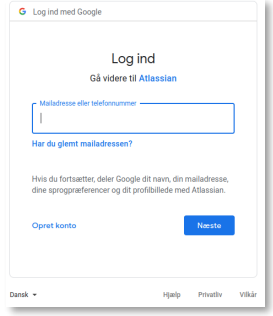


Figure 2.2 Sending the resource owner to the authorization server to start the process

Redirect

- … load the login page
  - Follow the requested redirect to the AuthServer

```
GET /authorize?response_type=code&scope=foo&client_id=oauth-client
-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%
2Fcallback&state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1 HTTP/1.1
Host: localhost:9001
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost:9000/
Connection: keep-alive
```



  - Now the resource owner logs in…

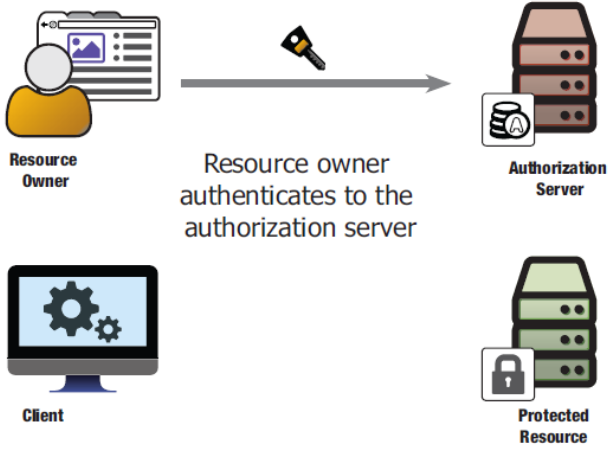


Figure 2.3 The resource owner logs in

AARHUS UNIVERSITET

Back to client

- AuthSrv redirect user *back* to client

Authorization code

```
HTTP 302 Found
Location: http://localhost:9000/oauth_callback?code=8V1pr0rJ&state=Lwt50DDQKU
B8U7jtfLQCVGDL9cnmwHH1
```

- … forcing the browser to request the client

```
GET /callback?code=8V1pr0rJ&state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1 HTTP/1.1
Host: localhost:9000
```

- … on the client's /callback route
  - (which must be defined for client to be OAuth compliant…)
- The *authorization code* is a one-time token

Call AuthSrv

- Client presents ‘auth code’ to AuthSrv using POST

```
POST /token
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic b2F1dGgtY2xpZW50LTE6b2F1dGgtY2xpZW50LXNlY3JldC0x

grant_type=authorization_code&
redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&code=8V1pr0rJ
```

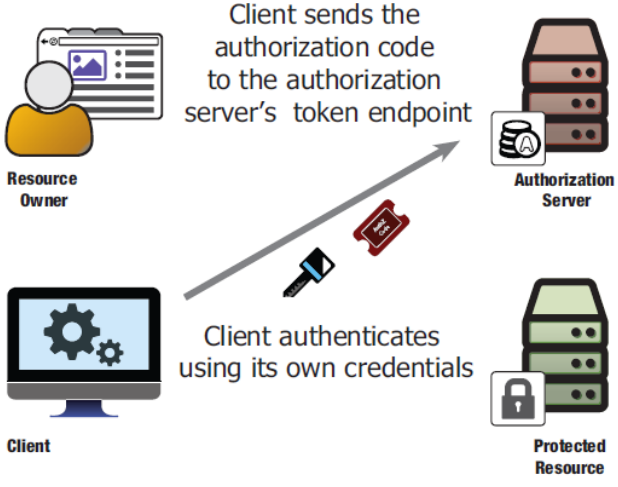


Figure 2.6 The client sends the code and its own credentials back to the authorization server

- Note: The client also presents its *own* credentials , so the AuthSrv can ensure it is talking with a valid client
  - I.e. the client app must in advance be registered in the AuthSrv!

AARHUS UNIVERSITET

(Side Note)

- Http basic access authentication (Wikipedia)

Basic access authentication

From Wikipedia, the free encyclopedia

In the context of an HTTP transaction, **basic access authentication** is a method for an HTTP user agent (e.g. a web browser) to provide a user name and password when making a request. In basic HTTP authentication, a request contains a header field in the form of `Authorization: Basic <credentials>`, where credentials is the Base64 encoding of ID and password joined by a single colon `:`.

- Java has library support for Base64

```
@Test
public void shouldDemonstrateBase64Encoding() {
  /*
  In the context of an HTTP transaction, basic access authentication is a method for an
  HTTP user agent (e.g. a web browser) to provide a user name interface and password
  when making a request.
  In basic HTTP authentication, a request contains a header field in the form of
  Authorization: Basic <credentials> credentials is the
  Base64 encoding of ID and password joined by a single colon
   */
  String idpwd = "mathilde_aarskort" + ":" + "333";
  String encodedString = Base64.getEncoder().encodeToString(idpwd.getBytes(StandardCharsets.UTF_8));
  assertThat(encodedString, is( value: "bWF0aGlsZGVfYWFyc2tvcnQ6MzMz"));

  byte[] decodedByteArray = Base64.getDecoder().decode(encodedString);
  String decodedString = new String(decodedByteArray, StandardCharsets.UTF_8);
  assertThat(decodedString, is(idpwd));
}
```

To finally get Access Token

- The reply of the POST /token

This token is returned in the HTTP response as a JSON object.

```
HTTP 200 OK
Date: Fri, 31 Jul 2015 21:19:03 GMT
Content-type: application/json

{
   "access_token": "987tghjkiu6trfghjuytrghj",
   "token_type": "Bearer"
}
```

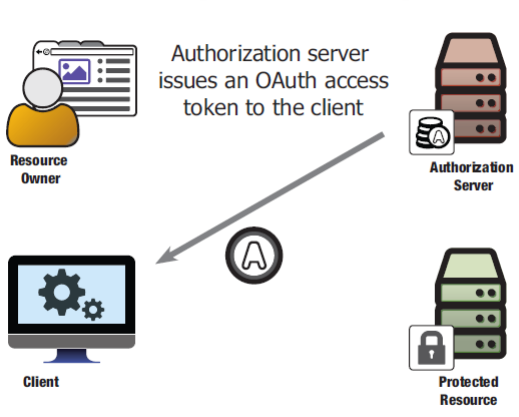


Figure 2.7 The client receives an access token

- This token embody ***This user is authorized*** and must be presented to all future calls to the Protected Resource
- That is, 'Client' must cache this token

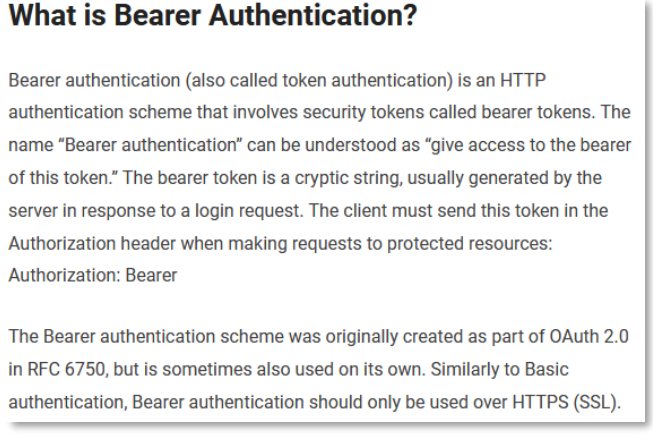
AARHUS UNIVERSITET

Access Token Use

- That is, all future calls to protected resource are like:

```
GET /resource HTTP/1.1
Host: localhost:9002
Accept: application/json
Connection: keep-alive
Authorization: Bearer 987tghjkiu6trfghjuytrghj
```

- I.e. a *bearer token* in HTTP terms
  - ‘the bearer of this token has access to…’
  - Like the card employees carry with them inside company buildings…



What is Bearer Authentication?

Bearer authentication (also called token authentication) is an HTTP authentication scheme that involves security tokens called bearer tokens. The name “Bearer authentication” can be understood as “give access to the bearer of this token.” The bearer token is a cryptic string, usually generated by the server in response to a login request. The client must send this token in the Authorization header when making requests to protected resources: Authorization: Bearer

The Bearer authentication scheme was originally created as part of OAuth 2.0 in RFC 6750, but is sometimes also used on its own. Similarly to Basic authentication, Bearer authentication should only be used over HTTPS (SSL).

AARHUS UNIVERSITET

The Access Token

- The access token is just a unique, opaque, bitstring
- But returned answer may include a lot of more info
  - Scope, expiration time, refresh token

This token is returned in the HTTP response as a JSON object.

```
HTTP 200 OK
Date: Fri, 31 Jul 2015 21:19:03 GMT
Content-type: application/json

{
   "access_token": "987tghjkiu6trfghjuytrghj",
   "token_type": "Bearer"
}
```

- Or you can encode info in the token itself
  - JSON Web Token (JWT)

```
const token = base64urlEncoding(header) + '.' + base64urlEncoding(payload) + '.' +
base64urlEncoding(signature)
```

AARHUS UNIVERSITET

Protected Resource

- So, the PR receives the token in *each request*
  - Must of course verify that it is a valid token!
  - And cache it to speed things upon on next requests…
  - May also include *scopes*, that is limit access to certain resources

> The protected resource can then parse the token out of the header, determine whether it's still valid, look up information regarding who authorized it and what it was authorized for, and return the response accordingly. A protected resource has a number of options for doing this token lookup, which we'll cover in greater depth in chapter 11. The simplest option is for the resource server and the authorization server to share a database that contains the token information. The authorization server writes new tokens into the store when they're generated, and the resource server reads tokens from the store when they're presented.

Phew…

- The process to get the access token is a *two step* protocol
  - 1. Get the one-time authorization code
    - GET /authorize
  - 2. Use client credentials + authorization code to get access token
    - POST /token

- Once the access token is provided the AuthSvr is not involved

AARHUS UNIVERSITET

Separation of Concerns

- Note that the client does not care how the AuthSvr does the authentication
  - It is fully delegated and the client is not involved
    - I can change my password and/or enable two-factor authentication
  - *Loose coupling*

AARHUS UNIVERSITET

Validating the Access Token

The Protected Resource’s view

Issue

- The PR is requested with an access token

```
GET /resource HTTP/1.1
Host: localhost:9002
Accept: application/json
Connection: keep-alive
Authorization: Bearer 987tghjkiu6trfghjuytrghj
```

- But how does the PR know that it is valid? And that it represents user A and not user B?
  - And what scopes have the user rights to access?

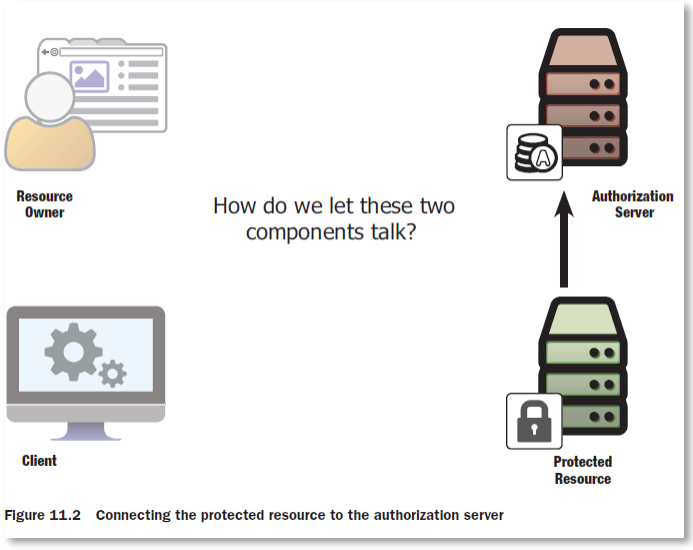


Figure 11.2 Connecting the protected resource to the authorization server

Solutions

- The simplest and often not possible solution…

> The simplest option is for the resource server and the authorization server to share a database that contains the token information. The authorization server writes new

- Obviously not relevant if we are using Google as AuthSvr!

- The other solution
  - *The inspection protocol*

AARHUS UNIVERSITET

Introspection protocol

- The AuthSvr provides the ability to validate the token
  - Is this token valid? What is the resource owner involved?

```
POST /introspect HTTP/1.1
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic
   cHJvdGVjdGVkLXJlc291cmNlLTE6cHJvdGVjdGVkLXJlc291cmNlLXNlY3JldC0x

token=987tghjkiu6trfghjuytrghj
```

- Again, this client must have been registered and present its credentials, and of course also present the token, that must be validated…

AARHUS UNIVERSITET

Introspection Reply

- Reply is a description of the token
  - Actually a JavaWebToken, JWT

```
HTTP 200 OK
Content-type: application/json

{
  "active": true,
  "scope": "foo bar baz",
  "client_id": "oauth-client-1",
  "username": "alice",
  "iss": "http://localhost:9001/",
  "sub": "alice",
  "aud": "http://localhost:/9002/",
  "iat": 1440538696,
  "exp": 1440538996,
}
```

  - Or you get 401 UNAUTHORIZED

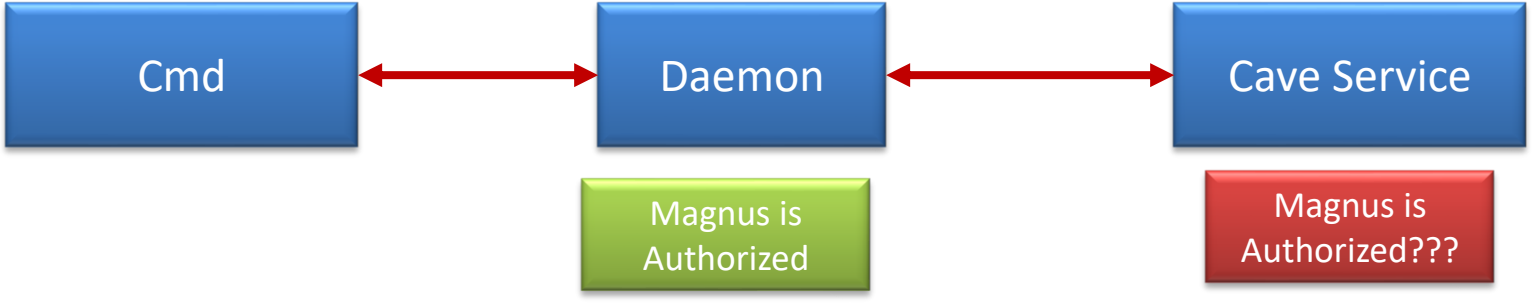
Table 11.1 Standard JSON web token claims

Claim Name	Claim Description
iss	The *issuer* of the token. This is an indicator of *who created this token*, and in many OAuth deployments this is the URL of the authorization server. This claim is a single string.
sub	The *subject* of the token. This is an indicator of *who the token is about*, and in many OAuth deployments this is a unique identifier for the resource owner. In most cases, the subject needs to be unique only within the scope of the issuer. This claim is a single string.
aud	The *audience* of the token. This is an indicator of *who is supposed to accept the token*, and in many OAuth deployments this includes the URI of the protected resource or protected resources that the token can be sent to. This claim can be either an array of strings or, if there's only one value, a single string with no array wrapping it.
exp	The *expiration* timestamp of the token. This is an indicator of *when the token will expire*, for deployments where the token will expire on its own. This claim is an integer of the number of seconds since the UNIX Epoch, midnight on January 1, 1970, in the Greenwich Mean Time (GMT) time zone.
nbf	The *not-before* timestamp of the token. This is an indicator of *when the token will begin to be valid*, for deployments where the token could be issued before it becomes valid. This claim is an integer of the number of seconds since the UNIX Epoch, midnight on January 1, 1970, in the GMT time zone.
iat	The *issued-at* timestamp of the token. This is an indicator of *when the token was created*, and is commonly the system timestamp of the issuer at the time of token creation. This claim is an integer of the number of seconds since the UNIX Epoch, midnight on January 1, 1970, in the GMT time zone.
jti	The *unique identifier* of the token. This is a value *unique to each token created by the issuer*, and it's often a cryptographically random value in order to prevent collisions. This value is also useful for preventing token guessing and replay attacks by adding a component of randomized entropy to the structured token that would not be available to an attacker.

AARHUS UNIVERSITET

Downstream Usage

- The inspection protocol is essential in a microservice context…
  - A 'front' service may pass the access token on to a downstream service
    - Which needs to verify the request is secure to perform
  - It can therefore always contact the AuthSvr to verify the received token (and cache it for further requests).



AARHUS UNIVERSITET

SkyCave Adaption

AARHUS UNIVERSITET

SkyCave Simplifications

- SkyCave is not a web based system, but Broker based
  - Redirects and stuff do not apply
- SkyCave is (already) a legacy system
  - The ‘daemon’ is the ‘API Gateway’ = single point of entry
  - That is, the client *does send credentials* to the daemon

- Design decision:
  - Keep the present centralized design
    - Client sends credentials to daemon
    - Daemon asks subscription service (AuthSvr) to authorize
  - Refactoring to a more correct design pending ☺ …

AARHUS UNIVERSITET

SkyCave Simplifications

- Simplifications
  - GET /authorize — Forward to AuthServer (auth token)
  - POST /token — Create and return Access Token
- .. Is replaced by a single **'POST /authorize'** to the AuthSrv **that provides the access token right away**
  - Simplifies design (and your work ☺)
- Java Connector Equivalent:
  - subscriptionService.authorize(loginName, pwd)
    - Which returns a subscriptionRecord with
      - getStatusCode() = 200 or 401
      - getAccessToken = "OAuth access token"

Protocol

```
Authorize a loginName
----

POST /api/v3/authorize

Accept: application/json
Authorization: Basic skycave_daemon:{daemon_pwd}

{
  "loginName": {login name}
  "password": {password}
}

Response:
  Status: 401 UNAUTHORIZED
    {
      "httpStatusCode": 401,
      "message": (error description)
    }

  Status: 500 INTERNAL SERVER ERROR
    {
      "httpStatusCode": 500,
      "message": (error description)
    }

  Status: 200 OK
    {
        "accessToken": "4ccb811e-df79-4385-a1ac-f3f2df647234",
        "httpStatusCode": 200,
        "message": "loginName (name) was authorized",
        "subscription": {
            "dateCreated": "2015-06-14 13:01 PM CEST",
            "groupName": "group-10",
            "groupToken": "Manganese946_Serbia419",
            "loginName": "name",
            "playerID": "a3607675-99b4-4ab7-8aa9-6f592676227c",
            "playerName": "EliaJørg",
            "region": "AALBORG"
        }
    }
```

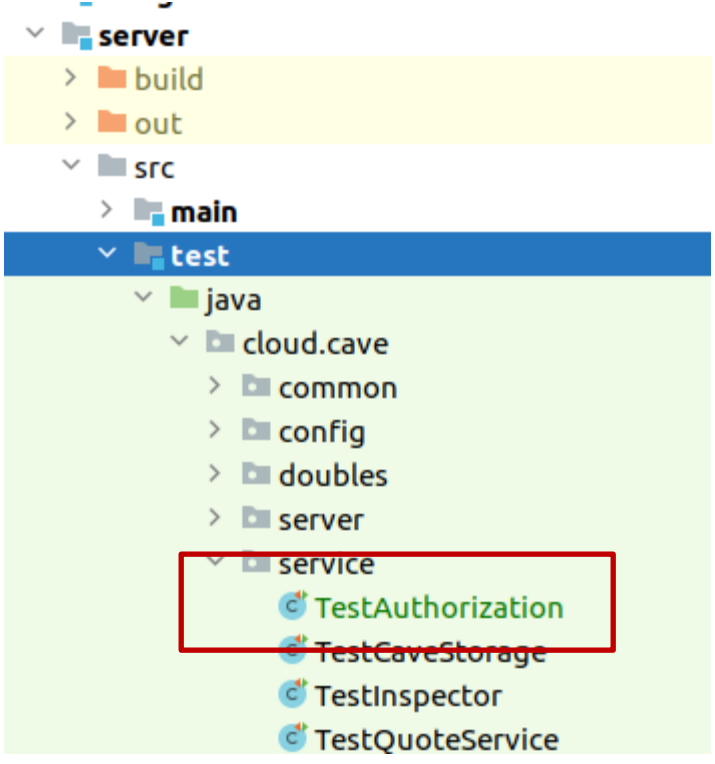
```
Comment:
  This POST request merges the OAuth 2's protocol
  of GET /authorize and POST /token in a single
  request; however the response carries a full
  SubscriptionRecord PODO as JSON.

  The OAuth relevant 'access token' is the value
  of key 'accessToken' in returned JSON.

  Returned status codes are the standard HTTP
  401 / UNAUTHORIZED and 200 / OK status codes,
  and is also replicated in the JSON payload.
  500 may be returned in case of server malfunction,
  typically internal persistent storage failure.

  The basic authentication is standard HTTP
  Base64 encoded of the literal "skycave_daemon" and
  the password for the daemon, separated by colon.
```

```
skycave_daemon:f8tqv56utx
```

AARHUS UNIVERSITET

Have a look at

- Find a few learning tests in:



Access Token

- New access tokens are issued upon *each /authorize call*
  - Access token must be presented to each call to a protected resource
    - It is actually a ‘session id’, identifying the current session a given player has within the cave…
    - Thus, it is presented to each ‘cmd – daemon’ interaction
      - Daemon is the protected resource, right…
- Access tokens do not expire in our subscription service
  - At least for now, maybe in the future…
- New login => New Access token
  - Thereby it mimics a session id; the old access token is invalidated…

AARHUS UNIVERSITET

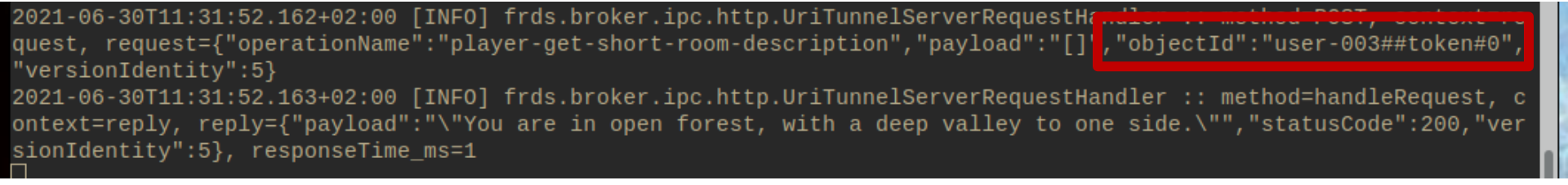
Access Token

- Broker Architecture issue
  - Broker sends requests to an ‘objectId’ and has no token field
    - But what is the objectId?
      - playerId identifies Magnus
      - accessToken identifies Magnus’ authorization
    - Is it the one? Is it the other?
  - Design decision
    - objectId is a mangling of **‘playerId##accessToken’**
      - Allows identifying ‘dual login’ handling
        - » That is, two+ clients competing to be ‘Magnus’
    - Daemon needs both to determine this situation…
      - PlayerId identifies Magnus, token the current session…

AARHUS UNIVERSITET

Example

- Mathilde moves north (with the stub subscription service, which issues tokens like “token#0”)



- Note: Not an exercise for you. It is taken care of in the provided code base
  - Just to justify the design decision…

AARHUS UNIVERSITET

/introspect

- Later in the course, you will be *strangling the daemon…*
  - Migrate from a monolith to a set of microservices
- Example
  - Daemon does *not handle messages in the room*
    - It asks a MessageService to do that on behalf of it
- A down stream service that needs to validate token
- *But we will return to this later…*

API

```
Introspect an AccessToken
---

POST /api/v3/introspect

Accept: application/json
Authentication: Basic skycave_service:{service_pwd}

{
  "token": {access token}
}

Response:
  Status: 401 UNAUTHORIZED
  {
       "httpStatusCode": 401,
       "message": "Could not introspect token {access token}"
  }

  Status: 200 OK
  {
      "accessToken": "6f9334b3-ced7-46ed-b4f8-002e49b15a42",
      "httpStatusCode": 200,
      "subscription": {
          "dateCreated": "2015-06-14 11:01 AM GMT",
          "groupName": "group-10",
          "groupToken": "Manganese946_Serbia419",
          "loginName": "rwar31t",
          "playerID": "a3607675-99b4-4ab7-8aa9-6f592676227c",
          "playerName": "EliaJørg",
          "region": "AALBORG"
      }
  }
```

```
Comment:
  This /introspect mimics OAuth 2.0 inspection of a token,
  used by downstream services to verify that a given bearer
  token indeed represents a valid authorization.

  Returned status codes are the standard HTTP
  401 / UNAUTHORIZED and 200 / OK status codes,
  and is also replicated in the JSON payload.
  500 may be returned in case of server malfunction,
  typically internal persistent storage failure.

  The basic authentication is standard HTTP
  Base64 encoded of the literal "skycave_service" and
  a password, separated by colon. The password is shared
  by all downstream services to simplify design.
```

A single token for all services

skycave_service:~~56utxf8tq~~

56utxf8tqv

AARHUS UNIVERSITET

Summary

- OAuth 2.0:
  - A *protocol* to authorize a client to access a protected resource
    - Rooted in the HTTP protocol
  - Without the client knowing the credentials; loose coupling
  - Parties: Resource owner, Client, AuthSvr, Protected Resource
  - Central artifact is the **access token**
    - Represents the resource owner when client access protected resource
    - The protocol outlines how the client receives the token
  - Any client that receives an access token can verify it using the introspection protocol

AARHUS UNIVERSITET

Summary

- In MSDO's SkyCave systems some simplifications have been made
  - Credentials *are* sent to the daemon
  - POST /authorize
    - Merges the two requests to get the access token
  - Client – daemon interactions are via the Broker pattern
    - objectId is a merge of (playerId, accessToken)
  - Downstream services can verify a token using POST /introspect