Degree Thesis, Åland University of Applied Sciences, Bachelor of Information Technology

AUTHORIZATION OF USERS IN A WEB APPLICATION

using OAuth-flow against Azure AD

Tomas Holmberg

2022:30

Datum för godkännande: 18.05.2022
Handledare: Björn-Erik Zetterman

EXAMENSARBETE
Högskolan på Åland

Degree Programme:	Informationsteknik
Author:	Tomas Holmberg
Title:	Auktorisering av användare i en webbapplikation genom användning av OAuth-flöde emot Azure AD
Academic Supervisor:	Björn Erik-Zetterman
Commissioned by:	Carus

Abstract

Syftet med detta examensarbete är att dokumentera auktorisering flödet vid en webbapplikation som begär åtkomst av skyddade resurser. Alla applikationer är registrerade hos Azure AD och är konfigurerade att ge en specifik organisations åtkomst.

Applikationerna är utvecklad i MSAL Javascript (frontend) och vi använder oss av Java Spring Boot för vår backend API.

Resultatet är en webbapplikation som anropar backend API:et som svarar med “Hello World” om användaren har behörighet.

Keywords

Javascript, API, Java, Spring Boot, Azure, JWT

Serial number:	ISSN:	Language:	Number of pages:
2022:30	1458-1531	English	34 pages

Handed in:	Date of presentation:	Approved:
27.04.2022	13.05.2022	18.05.2022

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Bachelor of Information Technology
Author:	Tomas Holmberg
Title:	Authorization of Users in a Web Application Using OAuth-flow against Azure AD
Academic Supervisor:	Björn Erik-Zetterman
Commissioned by:	Carus

Abstract

The purpose of this thesis is to document the authorization flow at a web application requesting access to protected resources. All applications are registered at Azure AD and are set to allow a specific organization to access them.

The application is developed in MSAL JavaScript (frontend) and we are using Java Spring Boot (backend).

The result is a web application calling the backend API, responding with "Hello World" if the user has authority.

Keywords

Javascript, API, Java, Spring Boot, Azure, JWT

Serial number:	ISSN:	Language:	Number of pages:
2022:30	1458-1531	English	34 pages

Handed in:	Date of presentation:	Approved:
27.04.2022	13.05.2022	18.05.2022

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

The purpose of this thesis is to document and develop a single-page web application with an OAuth-flow against Azure Active Directory. The application provides a login functionality that redirects to Microsoft's login to authenticate the user with a password and mobile authentication that remembers the unit you are using for up to 30 days. You can then make a call to the backend API. The API will be developed in Java Spring Boot, since it is a request from the customer. It will handle the validation process of the token received from our frontend, after validation is done it will give or deny access to one or multiple protected resources depending on how the token is defined.

I'm also creating a test organization on Azure and registering everything to make sure the applications can work together.

Our proof of concept is to get a response from the protected resource with a simple "Hello World".

1.2 Background

I was assigned to develop and document a web application and an API both registered at Azure Active Directory, the important part of my work was to demonstrate the OAuth flow against Azure. And the web application itself did not need to have much functionality, but was required to handle login calls and make a call to the protected resource.

Since the web application does not have a lot of requirements I went to Microsoft and grabbed one of their examples that provided you with an interface and sign-in functionality. This way I was able to focus on what was important, to gain access to the protected resources.

2. THEORY

2.1 Identity

The digital identity represents who we are when dealing with online activities and transactions. Due to the rapid growth of online services, users need to manage an increasing number of different identities (Warren et al., 2007).

A human-computer interaction refers to how a computer program or website is designed to make it clear and simple for users to interact with it. Identity management systems must be easy to use and have an intuitive interface. If the system is only designed to satisfy service provider requirements, it should also take into account the requirements of users; otherwise, it will be inconvenient and difficult to use for users when managing their identities (Alzomai, 2011).

2.2 Security

In today's applications security is of great importance because applications are often available over various networks and connected to the cloud. This increases the vulnerability to security threats and breaches. So how do we make ourselves less vulnerable? Most identity management systems will offer multi-factor authentication, we can have policies requiring strong passwords or updating passwords every 6 months, keeping software up-to-date, and the list goes on.

There are 3 categories of methodology to verify a user: something you know, something you are, or something you have (*Understanding the Three Factors of Authentication*, n.d.).

2.3 Azure AD

Azure Active Directory (Azure AD) is Microsoft's cloud service for access management, identity management, and application management. With this application, users or organizations can have access to a cloud-based directory with the ability to manage users, groups, devices, apps, and other IT resources with a single set of common tools. Currently, Azure AD has four different types of licenses. Each type of license provides a different set of features and capabilities (Ajburnle, n.d.).

2.3.1 Role-based access control (Azure RBAC)

First off you will be needing an Azure AD Premium P1 or P2 license to do this[1].

Roles, in general, are really all about permissions to help manage your organization, resetting passwords, adding more members, and so on. But we can also use it for role-based access control. This way we can control access to resources by assigning roles on Azure. A role assignment consists of three elements:

- security principal
- role definition
- scope

This is how permissions are enforced.

2.3.2 Security principal

The security principal is an object that represents a user, group, service principal, or managed identity that is requesting access to a resource. We are capable of assigning a role to any of these security principles, figure 1 shows all levels of the security principal.

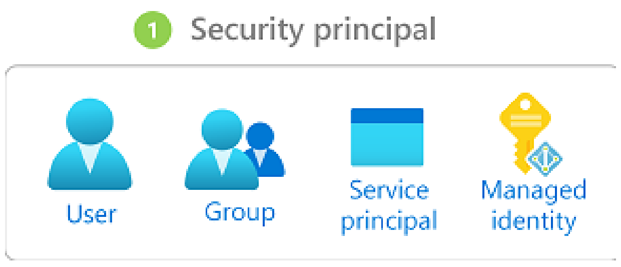
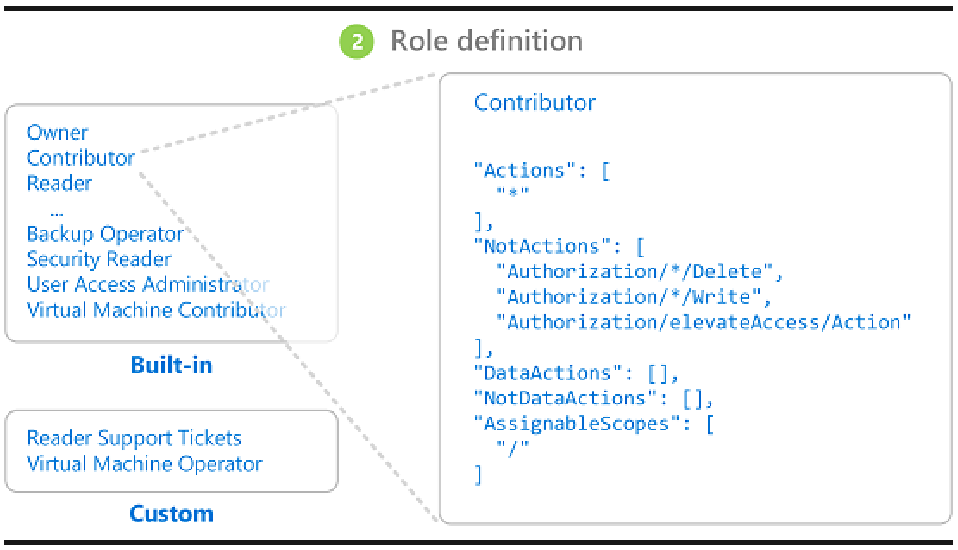


Figure 1. Showing all levels of the Security principal (rolyon, n.d.-b)

[1] https://azure.microsoft.com/en-us/pricing/details/active-directory/

2.3.3 Role definition

Role definitions are a set of permissions that apply to a role. It is usually simply referred to as a role. The role definition describes the actions that may be carried out by that role, such as reading, writing, and deleting. Roles may be of a high level, such as those of the owner, or they may be specific, such as those of the virtual machine reader. The Azure platform offers several built-in roles as well as the ability to create our own custom roles, figure 2 shows an example of a built-in role definition.



```
"Actions": [
  "*"
],
"NotActions": [
  "Authorization/*/Delete",
  "Authorization/*/Write",
  "Authorization/elevateAccess/Action"
],
"DataActions": [],
"NotDataActions": [],
"AssignableScopes": [
  "/"
]
```

Figure 2. Showing an example of how a built-in definition can look (rolyon, n.d.-b)

2.3.4 Scope

Scopes is the set of resources that the access applies to. You are capable of additionally restricting the actions permitted by a role by defining a scope when assigning it. This can be useful if you want to make someone a contributor, but only for one particular resource group.

It is possible to specify a scope in Azure at four different levels:

- management group
- subscription
- resource group
- resource

In order to construct scopes, the parent-child relationship is taken into consideration. This means that you can assign roles at any of the levels of scope, figure 3 shows the different levels of a scope.

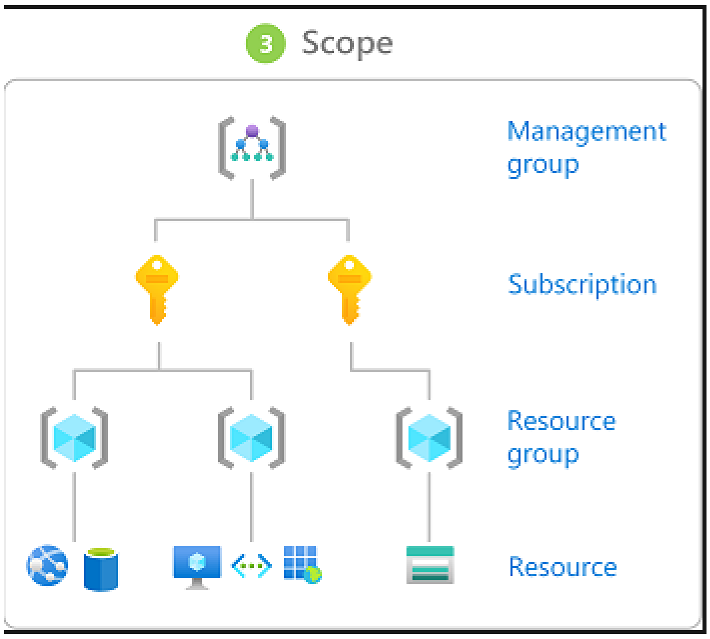


Figure 3. Showing the different levels of Scope (rolyon, n.d.-b)

2.3.5 Role Assignments

Basically, a role assignment is a process of assigning a role definition to a security principal at a particular scope in order to grant that principal access to certain resources. By creating a role assignment, access can be granted, and by removing it, access can be revoked.

An example of a role assignment is shown in figure 4. This example illustrates the Marketing team being assigned to the Pharma-Sales resource group and having the role of Contributor assigned to their role. Therefore, the users in the Marketing group will be able to create or manage any Azure resource in the Pharma-Sales resource group. Marketing users are not provided access to resources outside of the pharma-sales resource group unless they are part of another role assignment (rolyon, n.d.-a).

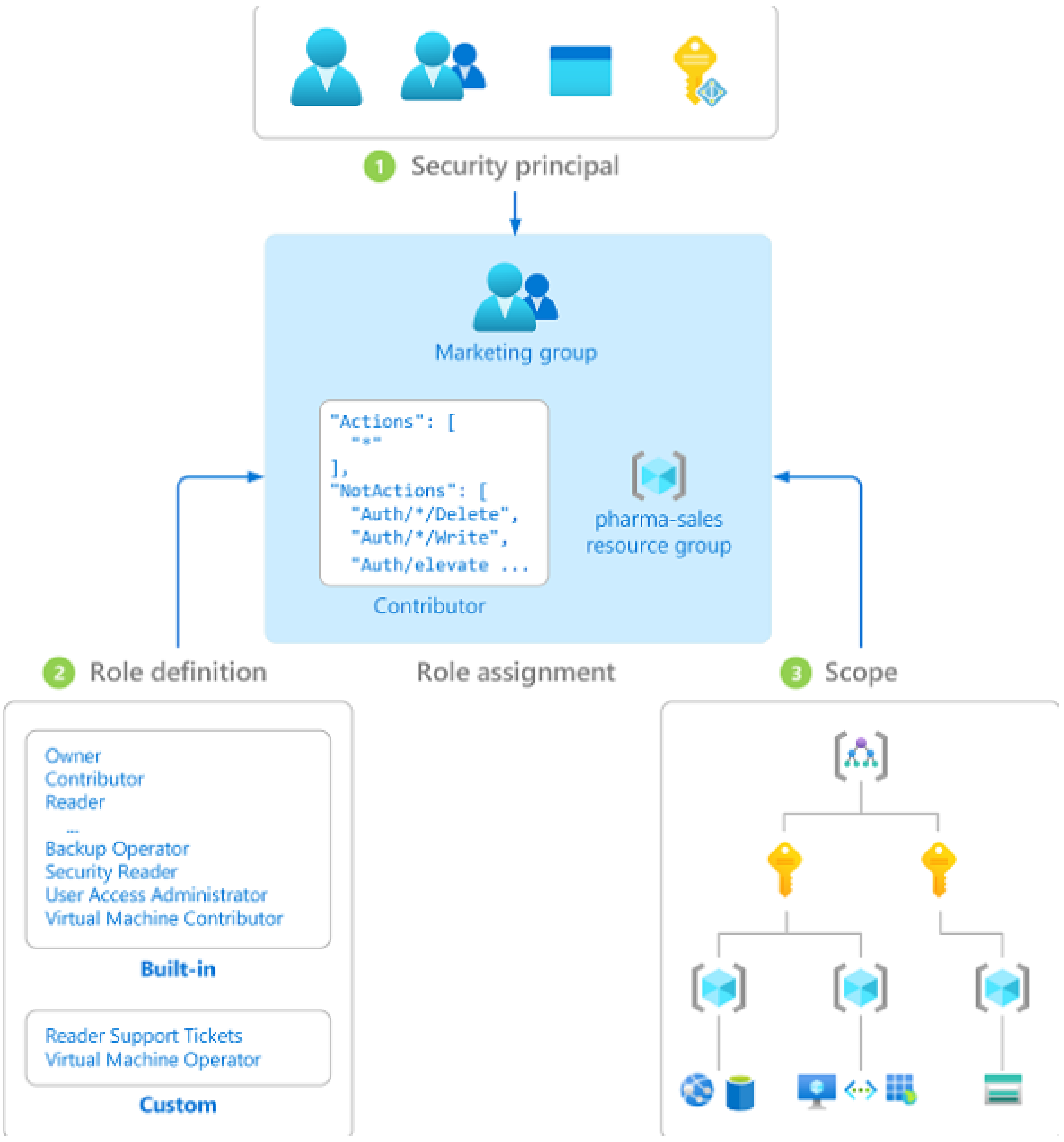


Figure 4. Showing an example of a Roles assignment being built (rolyon, n.d.-a).

2.4 Microsoft Identity Platform

Embedding the Microsoft identity platform in your applications lets you build applications that your customers and users can log into using their Microsoft identities or social accounts, and provide authorized access to your APIs or Microsoft APIs like Microsoft Graph (rwike, n.d.).

There are several segments that create the Microsoft identity platform, including

- OpenID Connect and OAuth 2.0-compatible authentication service, which enables developers to authenticate several different types of identities, including
    1. Business or school accounts provisioned by Azure Active Directory
    2. Accounts with Microsoft services such as Skype, Xbox, and Outlook
    3. Social or local accounts when they are used with Azure AD B2C
- Open-source libraries: Authentication libraries for Microsoft (MSAL) and other standards-compliant libraries
- Application management portal: The Azure portal allows users to register and configure themselves on top of other Azure management capabilities
- Application configuration API and PowerShell: You can automate DevOps tasks by programming your application's configuration using Microsoft graph API and PowerShell.
- Developer content: A collection of technical documentation, quickstarts, tutorials, how-to guides, and code samples.

For developers, Microsoft's identity platform integrates modern innovations in the identity and security space, including passwordless authentication, step-up authentication, and conditional access. Microsoft identity platform applications already take advantage of such innovations, so you do not have to implement them yourself.

The Microsoft identity platform allows you to write code once and have it reach any user. It is possible to create an application once and have it work across multiple platforms or to create an application that acts both as a client and as a resource application (API). Figure 5 shows a graph of the different applications you are capable of building using the Microsoft identity platform.

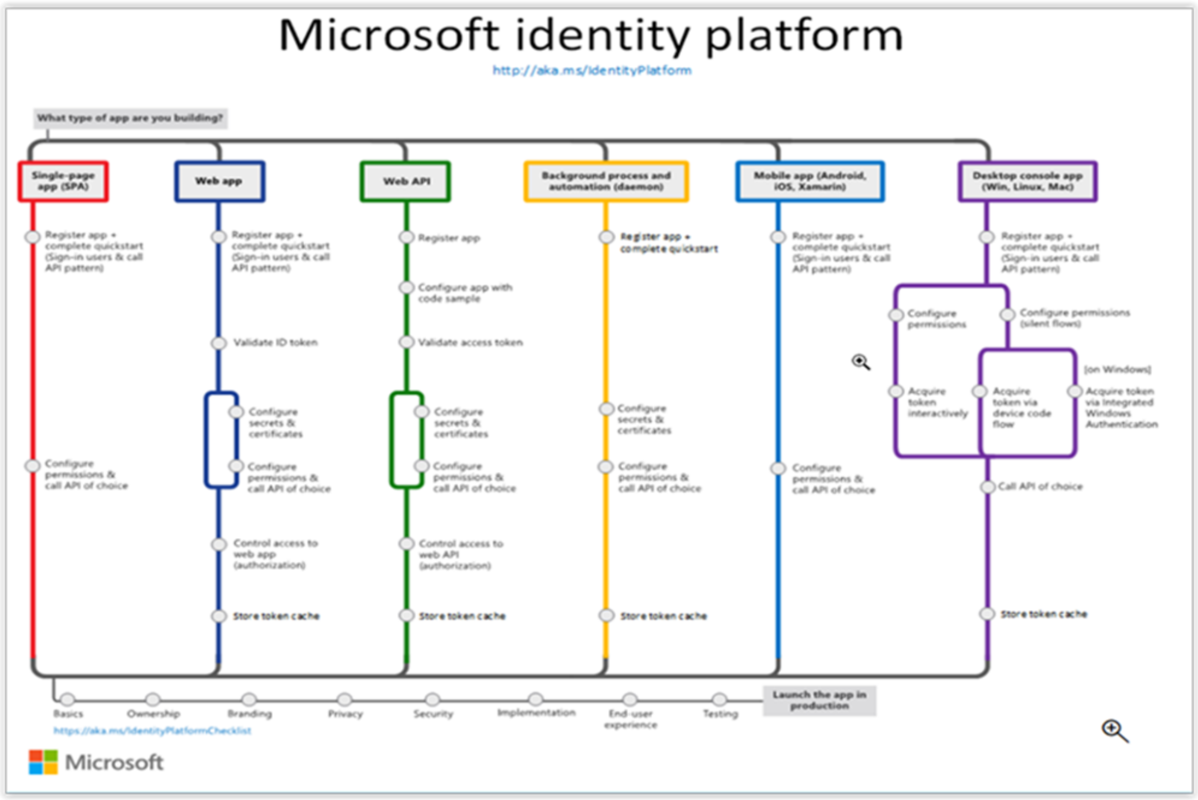

Microsoft identity platform
http://aka.ms/IdentityPlatform
What type of app are you building?
Single-page app (SPA)
Web app
Web API
Background process and automation (daemon)
Mobile app (Android, iOS, Xamarin)
Desktop console app (Win, Linux, Mac)
Register app + complete quickstart (Sign-in users & call API pattern)
Configure permissions & call API of choice
Register app + complete quickstart (Sign-in users & call API pattern)
Validate ID token
Configure secrets & certificates
Configure permissions & call API of choice
Control access to web app (authorization)
Store token cache
Register app
Configure app with code sample
Validate access token
Configure secrets & certificates
Configure permissions & call API of choice
Control access to web API (authorization)
Store token cache
Register app + complete quickstart
Configure secrets & certificates
Configure permissions & call API of choice
Store token cache
Register app + complete quickstart (Sign-in users & call API pattern)
Configure permissions & call API of choice
Register app + complete quickstart (Sign-in users & call API pattern)
Configure permissions
Configure permissions (silent flows)
[on Windows]
Acquire token interactively
Acquire token via device code flow
Acquire token via Integrated Windows Authentication
Call API of choice
Store token cache
Basics
Ownership
Branding
Privacy
Security
Implementation
End-user experience
Testing
Launch the app in production
https://aka.ms/IdentityPlatformChecklist
Microsoft

Figure 5. Metro map showing several application scenarios in Microsoft identity platform (rwike, n.d.).

2.5 OAuth2

OAuth2 stands for "Open Authorization" and is today a standard design to allow an application or website to access resources hosted by other web applications on behalf of a user. Open Authorization does not share users' credentials but instead uses access tokens to prove the identity between consumers and service providers (*RFC 6749 - the OAuth 2.0 Authorization Framework*, n.d.).

2.5.1 OAuth 2.0 Scopes

Scopes are an important concept in OAuth 2.0. They are used to specify exactly the reason for which access to resources may be granted. Acceptable scope values, and which resources they relate to, are dependent on the resource server, in our case, the resource server is Azure Active Directory.

2.5.2 How OAuth2 works

At the most basic level, the Client must acquire its own credentials, a client id, and client secret from the authorization server in order to identify and authenticate itself when requesting an access token (Leiba, 2012).

1. The client requests authorization (request) from the authorization server, supplying the client id and secret as identification. It also provides the scopes and an endpoint URI to send the access token or the authorization code to.
2. The authorization server authenticates the client and verifies that the requested scopes are permitted.
3. The resource owner interacts with the authorization server to grant access
4. The authorization server redirects back to the client with either an authorization code or access token, depending on the grant type. A refresh token may also be returned.
5. With the access token, the client can now request access to the resource from the resource server.

The simplest example of OAuth in action is one website saying "Hey, do you want to log in to our website with another website's login?". Figure 6 summarizes this data flow graphically.

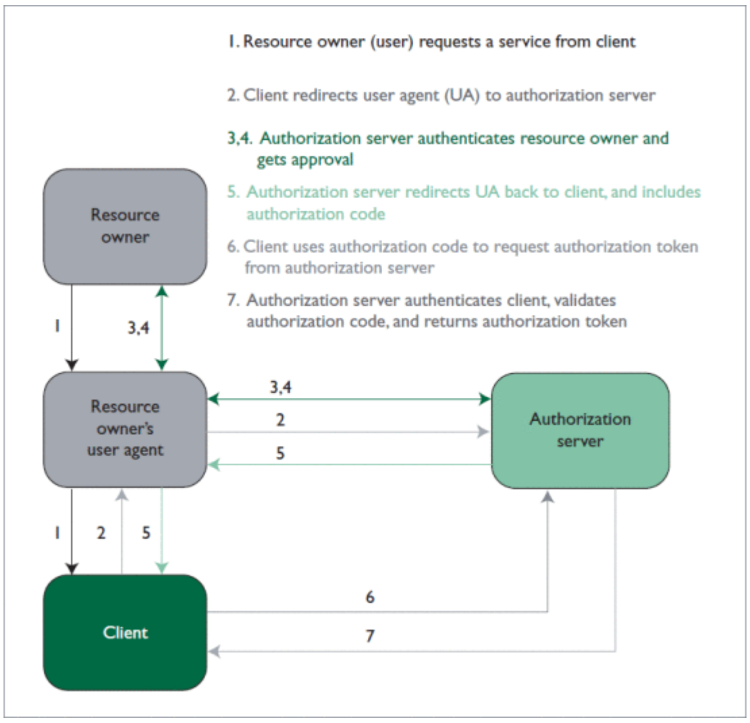

1. Resource owner (user) requests a service from client
2. Client redirects user agent (UA) to authorization server
3,4. Authorization server authenticates resource owner and gets approval
5. Authorization server redirects UA back to client, and includes authorization code
6. Client uses authorization code to request authorization token from authorization server
7. Authorization server authenticates client, validates authorization code, and returns authorization token
Resource owner
Resource owner's user agent
Authorization server
Client
1
3,4
2
5
6
7

Figure 6. An OAuth transaction. A Facebook user wants to import Gmail contacts into Facebook without giving Facebook her email username or password (Leiba, 2012).

2.5.3 Grant type

Grant types will be described here, focusing on the most common grant types. Grants are the set of steps a client has to perform to get resource access authorization (*What Is OAuth 2.0 and What Does It Do for You?* n.d.). The authorization framework provides several grant types to address different scenarios.

1. Authorization Code grant, the authorization server returns a single-use authorization code to the client, which is then exchanged for an access token. This is normally the best option for traditional web applications where the exchange can securely happen on the server-side. However, the client's secret cannot be stored securely, and therefore the authentication during the exchange is limited to the use of the client id alone.
2. Authorization code grant with proof key for code exchange (*RFC 7636 - Proof Key for Code Exchange by OAuth Public Clients*, n.d.). This authorization flow is similar to the previous one, but with additional steps to make it more secure for mobile/native applications and SPAs.

2.6 JSON Web Token

"A JSON Web Token (JWT) is an open standard that defines a compact and self-contained way to transmit information securely and reliably between parties as a JSON object." (RFC 7519)

Information transmitted by JWT can be verified and trusted since JWTs are digitally signed. It is possible to sign JWTs using a secret with the HMAC algorithm or using a public/private key pair with RSA or ECDSA.

In terms of these JWTs, they can be encrypted to ensure that there is also secrecy between the parties. The importance of signing JWTs will be discussed in this portion. Signed tokens allow users to verify the integrity of the claims that are contained in them, while encrypted tokens conceal those claims from third parties. In the case of signing tokens using a public/private key pair, the signature also provides a guarantee that the tokens were only signed by the party holding the private key.

2.6.1 When should JWTs be used?

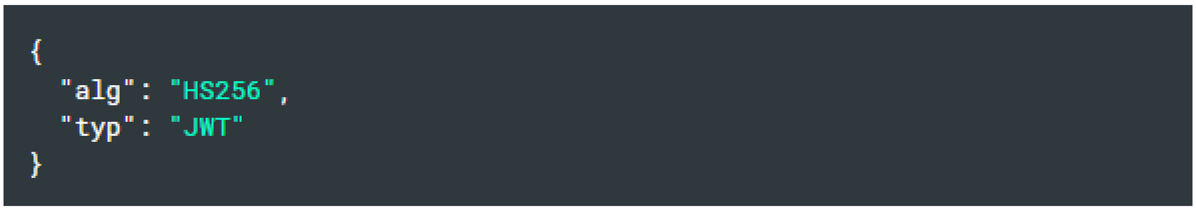
- In the majority of cases, JWT is used for authentication purposes. Following the first login, every subsequent request will include a JWT, which will allow the user to access all routes, services, and resources that are permitted by that token. JWT has become a widely popular feature for Single Sign-On these days, largely because of its small overhead and its ability to be used across all sorts of different domains with ease.

- Exchange of information between parties, JWT is a good method for sending information securely. Since JWTs can be signed, we can verify whether or not the content has been modified in any way.

2.6.2 The structure of JWT

The JWT is made up of three parts, each separated by a dot, which is its compact form. Therefore, a JWT is typically displayed in this manner: xxxxx.yyyyy.zzzzz.

- Header - In general, the header is composed of two parts, the type of token and the signature algorithm that will be used. Figure 7 shows an example of a header.

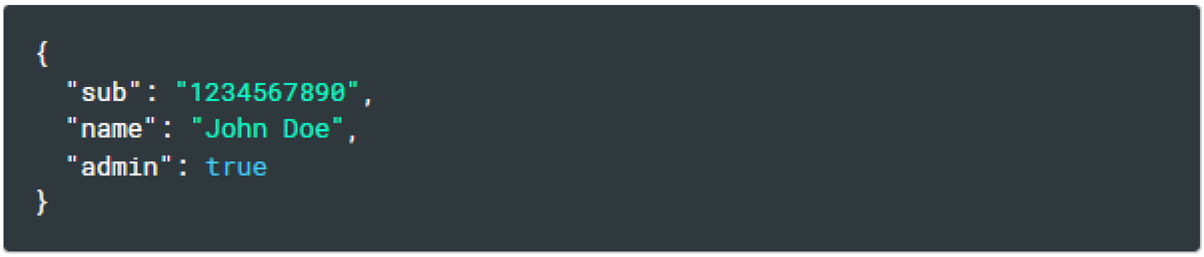


```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Figure 7. Showing how a decoded header may look.

- The payload is the second part of the token, in which the claims can be found. A claim is a statement that describes something about a person or organization (usually the user) and includes additional data. There are three types of claims, as follows: registered, public and private. Figure 8 shows an example of a payload
    1. A registered set of claims is a predefined set of claims that are not mandatory but are recommended to provide a set of useful and interoperable claims. Among those types are issuer (iss), expiration time (exp), subject (sub), and audience (aud).

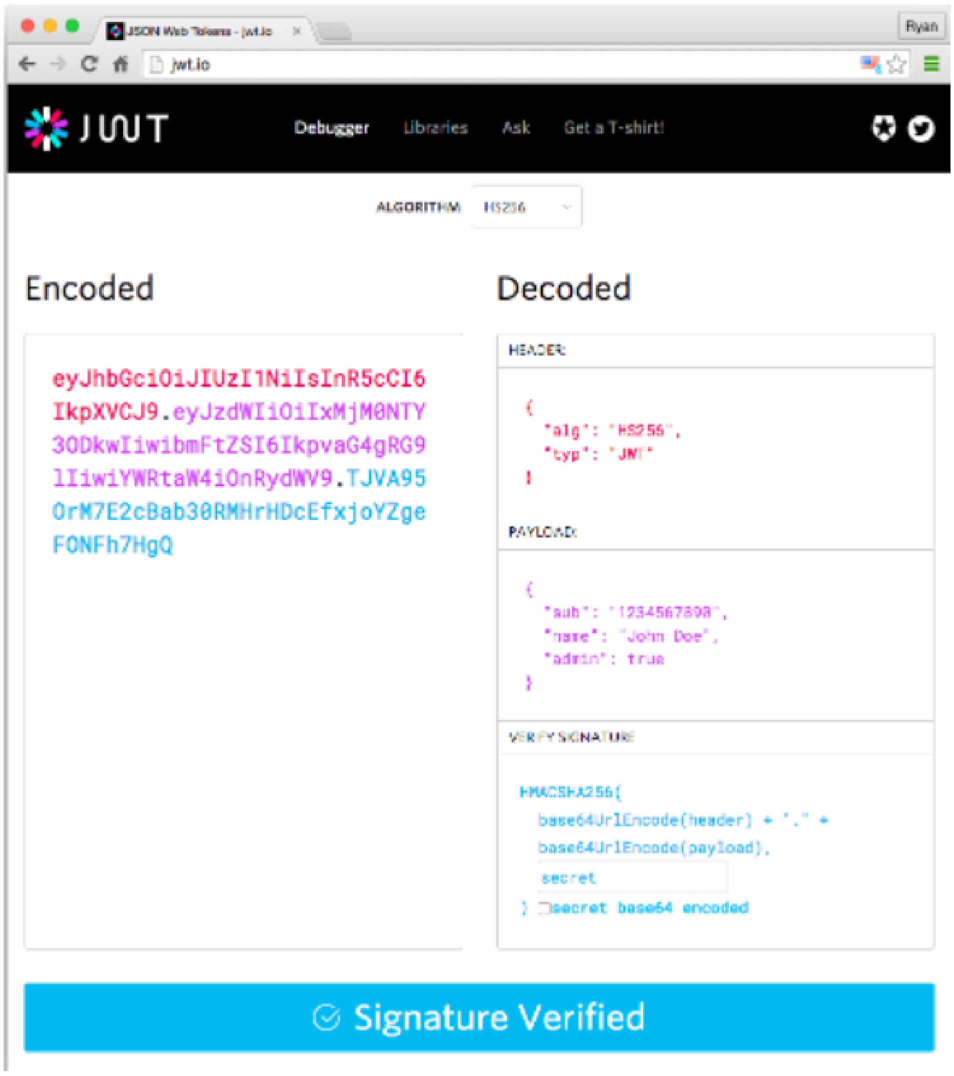
2. Public claims are defined at will by those using them, so they can be defined however they like. To ensure we eliminate collisions with the JWT registry, they should be defined as URIs instead of JWT registry entries or be defined as names that contain collision-resistant namespaces.
3. A private claim is a claim which is made up of custom information to share with other parties who agree to share it, and which is neither registered nor public.



```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Figure 8. Showing how a decoded payload may look.

- Signature - It is necessary to use the encoded header and the encoded payload as well as a secret and the algorithm provided in the header, in order to create the signature part. With the signet, we are able to ensure that the message was not modified along the way, and in the case of tokens signed with a private key, we can verify that the sender of the JWT is who they claim to be. Figure 9 shows an example of a complete token decoded at jwt.io.



```
eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpXVCJ9.eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6IkpvaG4gRG9
lIiwiYWRtaW4iOnRydWV9.TJVA95
OrM7E2cBab30RMHrHDcEfxjoYZge
FONFh7HgQ
```

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) secret base64 encoded
```

Figure 9. Showing a decoded example token at jwt.io (jwt.io).

2.6.3 How does JWT work?

In an authentication process, a JSON Web Token will be returned to the user when they are successfully logged in with their credentials. Because tokens are credentials, precautions must be taken in order to ensure their security. The rule of thumb is that a token should not be kept longer than needed, and sensitive session data should never be stored in the browser's memory due to a lack of security (*RFC 7519 - JSON Web Token (JWT)*, n.d.).

As soon as the user attempts to access a protected route or resource, the user agent should send the JWT, usually as a part of the Authorization header using a Bearer scheme as the bearer. This can be considered a stateless authorization mechanism. The server's protected routes will check each Authorization header for a valid JWT, and if one is present, the user will be allowed to access the protected resource. If the JWT contains the necessary information, the need to query the database for certain operations may also be reduced, but this may not always be true.

When sending JWT tokens through HTTP headers, you need to prevent them from growing too large. In some cases, bigger headers can be rejected by the server. You may need an alternative solution if you want to embed all the user's permissions in a JWT token.

2.6.4 Why should we use JWT

As JSON is less verbose than XML[2], when it is encoded its size is also smaller, making JWT more compact than SAML[3]. This makes JWT a good choice to be passed in HTML and HTTP environments.

Security-wise, SWT can only be symmetrically signed by a shared secret using the HMAC[4] algorithm. However, JWT and SAML tokens can use a public/private key pair in the form of an x.509 certificate[5] for signing. Signing XML with an XML digital signature without introducing obscure security holes is very difficult when compared to the simplicity of signing JSON.

JSON parsers are common in most programming languages because they map directly to objects. Conversely, XML doesn’t have a natural document-to-object mapping. This makes it easier to work with JWT than SAML assertions.

Regarding usage, JWT is used at the Internet scale. This highlights the ease of client-side processing of the JWT on multiple platforms, especially mobile (Auth, n.d.).

[2] https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction
[3] https://developers.onelogin.com/saml
[4]https://www.geeksforgeeks.org/hmac-algorithm-in-computer-network/#:~:text=HMAC%20algorithm%20stands%20for%20Hashed,uses%20the%20Hashing%20concept%20twice.
[5]https://sectigo.com/resource-library/what-is-x509-certificate#:~:text=Share%20this-,An%20X.,internet%20communications%20and%20computer%20networking.

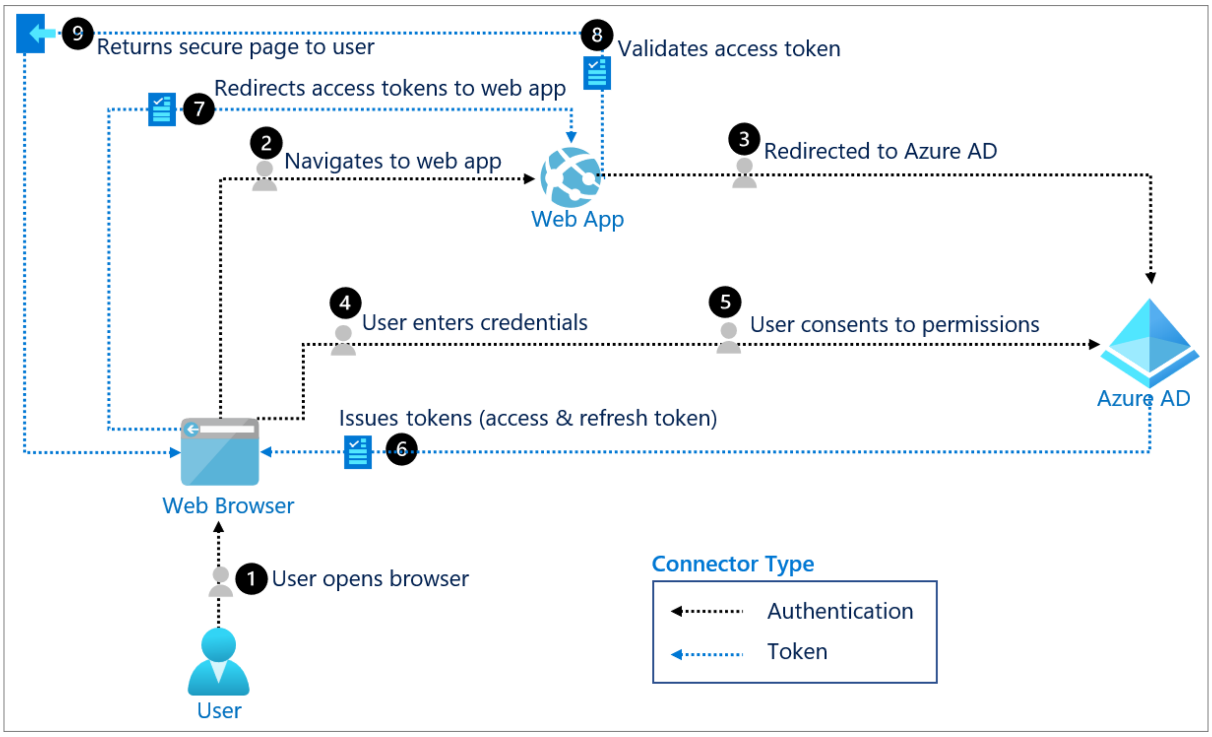
2.7 OpenID Connect

There are numerous types of authentication protocols that can be used for any application, but OpenID Connect ((*OpenID Connect Core 1.0 Incorporating Errata Set 1*, n.d.)) is one of the main ones (along with OAuth2). The OIDC provides identity services using standardized message flow standards defined by OAuth2.

The design goal of OIDC is “making simple things simple and complicated things possible”. As a result of OIDC, web developers can authenticate their users on a variety of websites and applications without needing to maintain password files on their servers. As a result of this method, the app builder has a secure way to verify the identity of the person logging into the browser or native application that is connected to the application.

The authentication process of the user must be done at an identity provider where the credentials or session of the user is checked. This can only be done with a trusted agent. The native app will generally launch the system browser to perform this task. Embedded views are not considered trusted as there is nothing that prevents the app from prying into the user's password through the embedded view.

Additionally to authentication, consent can be requested from the user. Consent is the user’s explicit permission to allow an application to access protected resources. The difference between consent and authentication is that consent only needs to be provided once for a resource. Consent remains valid until the user or administrator revokes it, figure 10 shows the authentication flow to a web application registered at Azure.



9 Returns secure page to user
8 Validates access token
7 Redirects access tokens to web app
2 Navigates to web app
3 Redirected to Azure AD
Web App
4 User enters credentials
5 User consents to permissions
Azure AD
Issues tokens (access & refresh token)
6
Web Browser
1 User opens browser
User
Connector Type
Authentication
Token

Figure 10. Showing the authentication flow of a user signing into their web application that is connected to Azure AD (BarbaraSelden, n.d.).

2.8 Spring Boot

Both small services and monolithic applications can be developed with Spring Boot. There are several types of web applications and services that can be used, including APIs or web applications, command-line programs, and orchestration software. Ultimately, Spring Boot orchestrates the Spring Framework. Dependency Injection and aspect-oriented programming are among the features of the Spring Framework for application-level infrastructure. As a result, building blocks become higher value concepts. This may include transactions, security, and more, depending on the module (*Spring Boot Infographic - Download Form*, n.d.).

2.9 Multi-Factor Authentication

Multi-Factor Authentication is a process when the user signing in is prompted for an additional form of identification, such as a code to their cell phone or a fingerprint scan. MFA works by requiring two or more of the following authentication methods (*Understanding the Three Factors of Authentication*, n.d.).

- password, “Something you know”

- a trusted device that is not easily duplicated, like a phone or a hardware key "Something you have"
- biometric authentication, like a fingerprint or a face scan "Something you are"

Applications or services that use MFA, do not need to be changed, since the verification prompts are a part of Azure AD sign-in. With Azure AD there are seven different types of multi-factor authentication options, such as Microsoft Authentication app, Windows Hello for business, FIDO2 security key, Oath hardware token, Oath software token, SMS, and voice call.

Azure's MFA can easily be configured to suit your organization by adding conditional access policies to define events or applications that require MFA. Policies can allow regular sign-in when the user is on the corporate network or a registered device but prompt for additional verification factors when the user is remote or on a personal device. Figure 11 provides a summary graphically of policies.

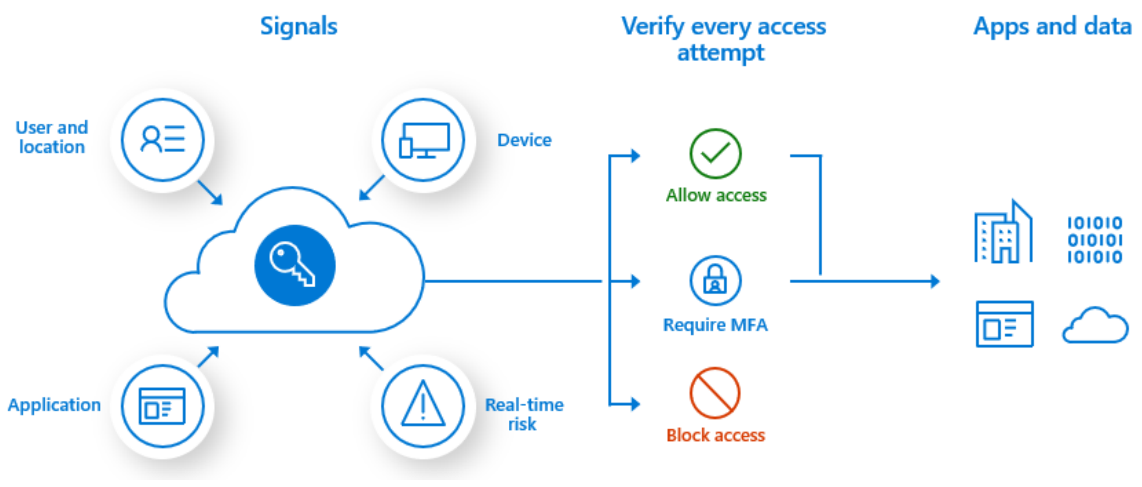


Figure 11. Depending on the policies defined by the organization the user will be allowed, prompted, or blocked when trying to sign in (Justinha, n.d.).

2.10 Javascript

JavaScript is a flexible, dynamic, prototype-based, easy-to-learn, easy-to-use, and versatile programming language that is used primarily on the world wide web. It should be noted that in spite of its initial focus on assisting in the generation of dynamic content for the web, the language has found its way into numerous other applications as well.

2.11 Cross-Origin Resource Sharing

As the name suggests, Cross-Origin Resource Sharing is a mechanism that relies on an HTTP header to allow servers to point browsers to other origin sites (such as domains, schemes, or ports) from which resources can be loaded. Moreover, CORS also relies on a mechanism by which browsers conduct a "preflight" request to the server hosting the cross-origin resource, in order to test whether the server will permit the actual request to go through. When the browser sends the preflight request, it sends headers that indicate the HTTP (Fielding, 1999) method that will be used in the request as well as headers that will be used in the request itself, figure 12 shows a demonstration of a preflight request and the actual request (*Cross-Origin Resource Sharing (CORS)*, n.d.).

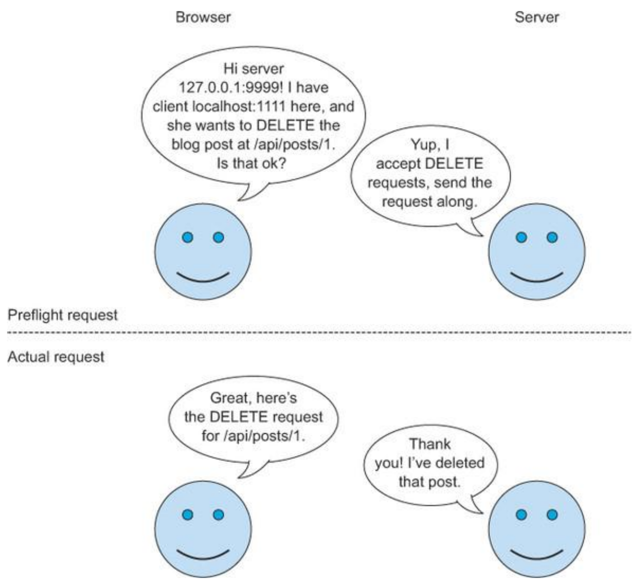

Browser
Server
Hi server 127.0.0.1:9999! I have client localhost:1111 here, and she wants to DELETE the blog post at /api/posts/1. Is that ok?
Yup, I accept DELETE requests, send the request along.
Preflight request
Actual request
Great, here's the DELETE request for /api/posts/1.
Thank you! I've deleted that post.

Figure 12. Explanation of a preflight request and actual request between browser and server (Chapter 4. Handling Preflight Requests, n.d.).

3. REQUIREMENTS FOR SOFTWARE

The requested application that was assigned to me was to include a frontend with login functionality and authenticate the user against Azure. After authentication, we should then receive a token that could be used to request access to protected resources (API). The backend API should then validate the token against Azure to grant or deny access to the user doing the request. Figure 13 shows how Carus would like the flow to work.

1. Web Application with login functionality and authentication against Azure
2. Getting an Access Token from Azure
3. Requesting resources from the API
4. Sending the token to a backend API
5. API validating the token against Azure
6. API sending back the requested resource.

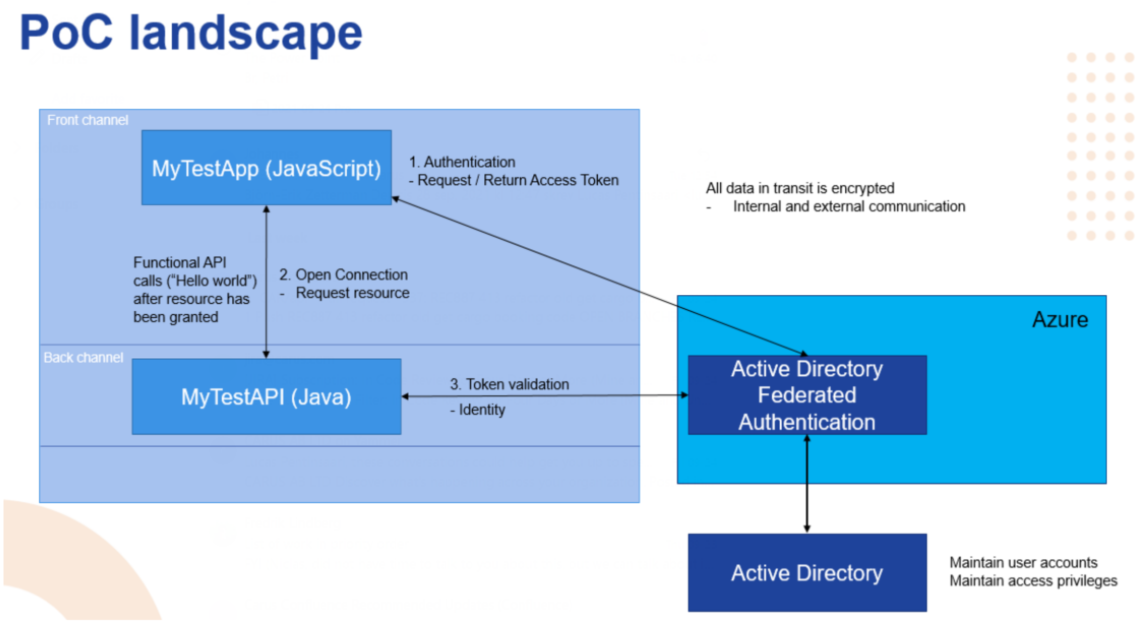

PoC landscape
Front channel
MyTestApp (JavaScript)
1. Authentication
- Request / Return Access Token
All data in transit is encrypted
- Internal and external communication
Functional API calls ("Hello world") after resource has been granted
2. Open Connection
- Request resource
Azure
Back channel
MyTestAPI (Java)
3. Token validation
- Identity
Active Directory Federated Authentication
Active Directory
Maintain user accounts
Maintain access privileges

Figure 13. Image provided to me from an employee at Carus.

4. IMPLEMENTATION

This chapter describes the actual implementation of the software done in this project.

4.1 Registering applications at Azure

First thing needed is a tenant (organization) existing or new that is up to you. When our tenant is set up we are able to register applications, you are able to specify who can use your application, this is called a sign-in audience, but the default option is “Accounts in this organizational directory only”. Once registered your applications are given a unique client-id (application-id). You have now established a trust relationship between your application and the Microsoft identity platform, your application now trusts Microsoft and not the other way around.

When registering APIs we have the same process as before, the difference is that we add scopes to this application. First, if we set our Application ID URI, the default one will be “api://<application-client-id>” but there is also other supported patterns. Now we can add a scope to our application, there are two important fields such as the name and who can consent. The naming convention is usually “resource.operation.constraint” for example Employees.Read.All. And that is it, your API has now been exposed and you are able to connect it to your other applications.

4.2 First Prototype

Microsoft is providing a lot of examples for different scenarios. I chose an example with MSAL JavaScript that provides a frontend and a backend API. But since we want a Spring Boot backend we had to remake the provided backend and implement our own.

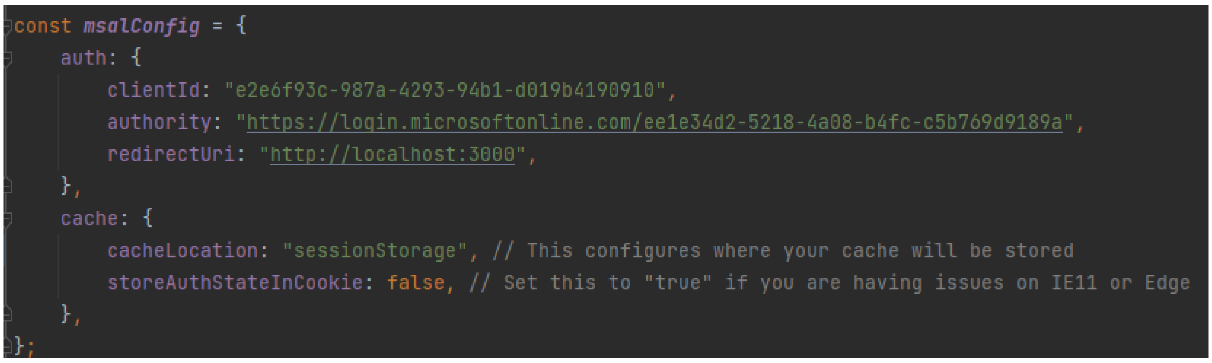
Before deleting the API we checked that the registered applications at Azure actually communicated.

4.3 Implementing Azure to the code

Azure is a set of services provided by Microsoft, which have different cloud integrations to use. In the following chapters the consumed integrations will be described.

4.3.1 Implementing Azure to our frontend

This chapter will display how easy it is to configure our frontend application with our SPA application ID, and who has the authority to sign in to it or the tenant (the organization). Now everyone in your organization has the authority to log in to the web application. This does not mean everyone has the authority to use the resources, figure 14 shows the connection between Azure AD and the web application.

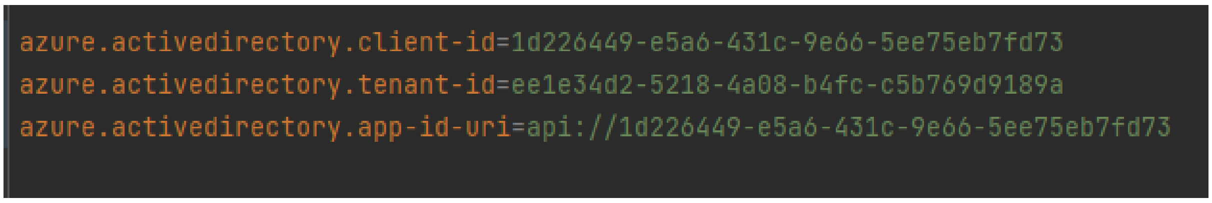


```
const msalConfig = {
    auth: {
        clientId: "e2e6f93c-987a-4293-94b1-d019b4190910",
        authority: "https://login.microsoftonline.com/ee1e34d2-5218-4a08-b4fc-c5b769d9189a",
        redirectUri: "http://localhost:3000",
    },
    cache: {
        cacheLocation: "sessionStorage", // This configures where your cache will be stored
        storeAuthStateInCookie: false, // Set this to "true" if you are having issues on IE11 or Edge
    },
};
```

Figure 14. Showing the implementation of Azure on our frontend.

4.3.1 Implementing Azure to our backend APIs

So with defining the API client-id (application-id) along with the tenant-id and the app-id-uri in our application properties we are now allowing users from this organization to use this API, figure 15 shows the connection between Azure AD and the API.



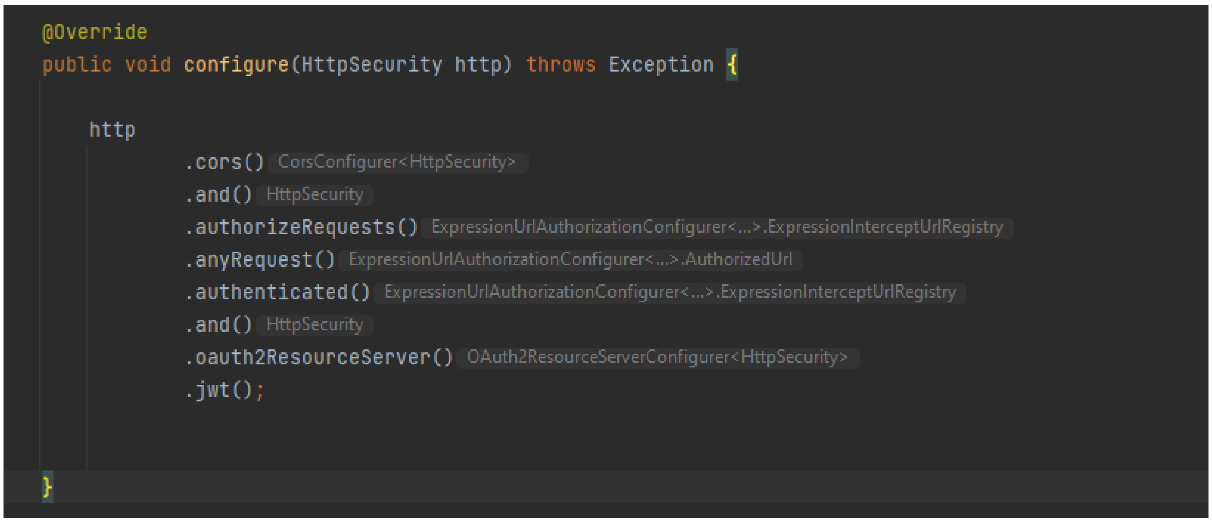
```
azure.activedirectory.client-id=1d226449-e5a6-431c-9e66-5ee75eb7fd73
azure.activedirectory.tenant-id=ee1e34d2-5218-4a08-b4fc-c5b769d9189a
azure.activedirectory.app-id-uri=api://1d226449-e5a6-431c-9e66-5ee75eb7fd73
```

Figure 15. Showing the implementation of Azure in our backend.

If we don’t want people to have access to all our resources. It is possible to use a group(role) claim from the token to assign what a person is allowed to use. So for example we can use groups to decide who is allowed to use what. This way we are able to have consumers, developers, etc. sign in to the same web application but assign what resources they are allowed to use. We talked about this in chapter 2.6.1.

4.4 Problems with CORS

I had issues during the implementation of CORS, running solutions back and forth between Google searches with no result. I did this for more than a week and had over 6000 lines of code changed and removed, then I finally realized I had two configurations, one that implemented CORS and the other overriding the first one to remove CORS again, figure 16 shows my current implementation of CORS.

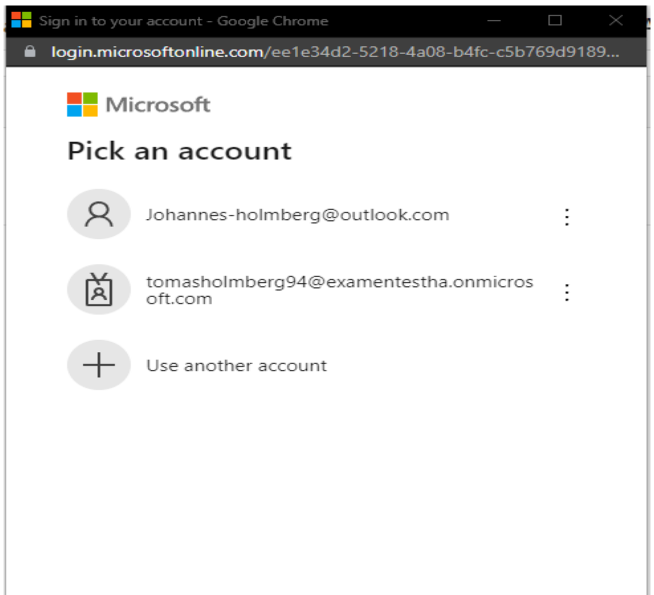


```
@Override
public void configure(HttpSecurity http) throws Exception {

    http
            .cors() CorsConfigurer<HttpSecurity>
            .and() HttpSecurity
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .anyRequest() ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
            .authenticated() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .and() HttpSecurity
            .oauth2ResourceServer() OAuth2ResourceServerConfigurer<HttpSecurity>
            .jwt();

}
```

Figure 16. Showing the implementation of CORS in the backend.

4.5 User interface

Not really much to say about the interface. We took it from Microsoft and didn't do any design changes. What we really wanted was the popup login. See figure 17.

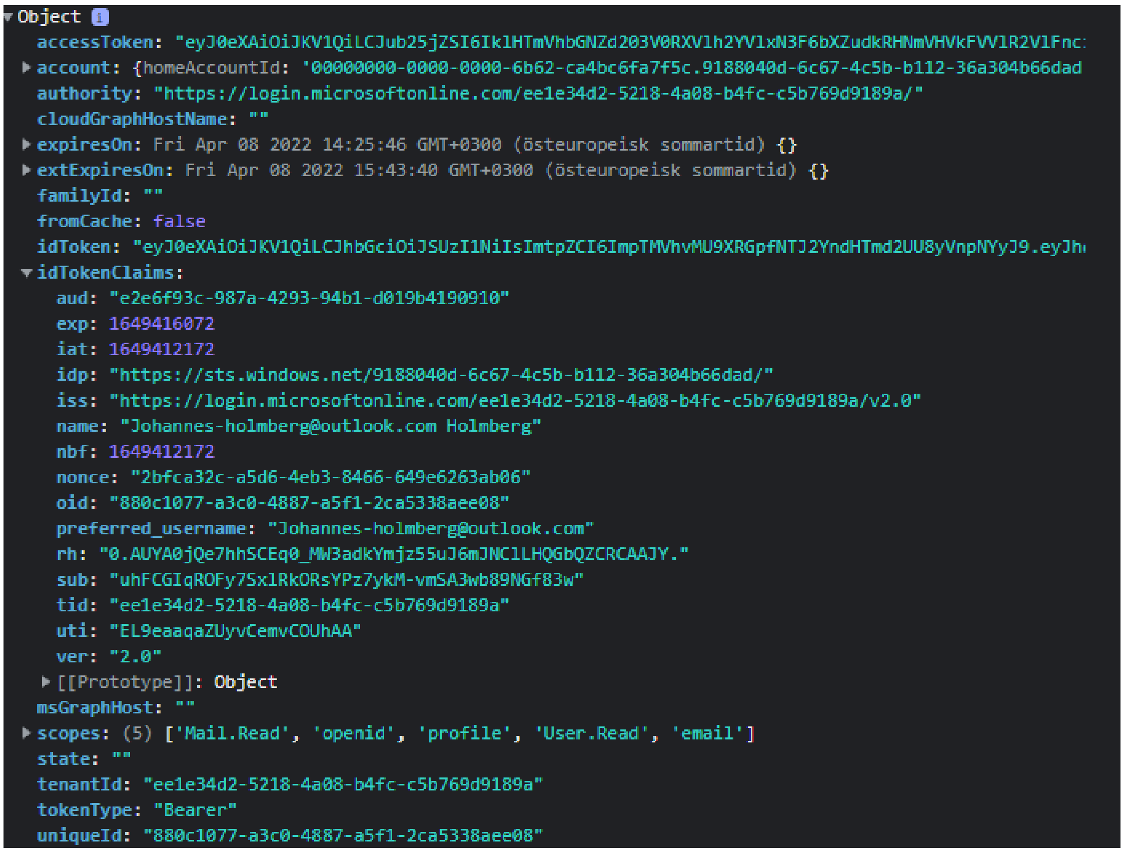


Sign in to your account - Google Chrome
login.microsoftonline.com/ee1e34d2-5218-4a08-b4fc-c5b769d9189...
Microsoft
Pick an account
Johannes-holmberg@outlook.com
tomasholmberg94@examentestha.onmicrosoft.com
Use another account

Figure 17. Showing Microsoft popup-login

4.6 Final product

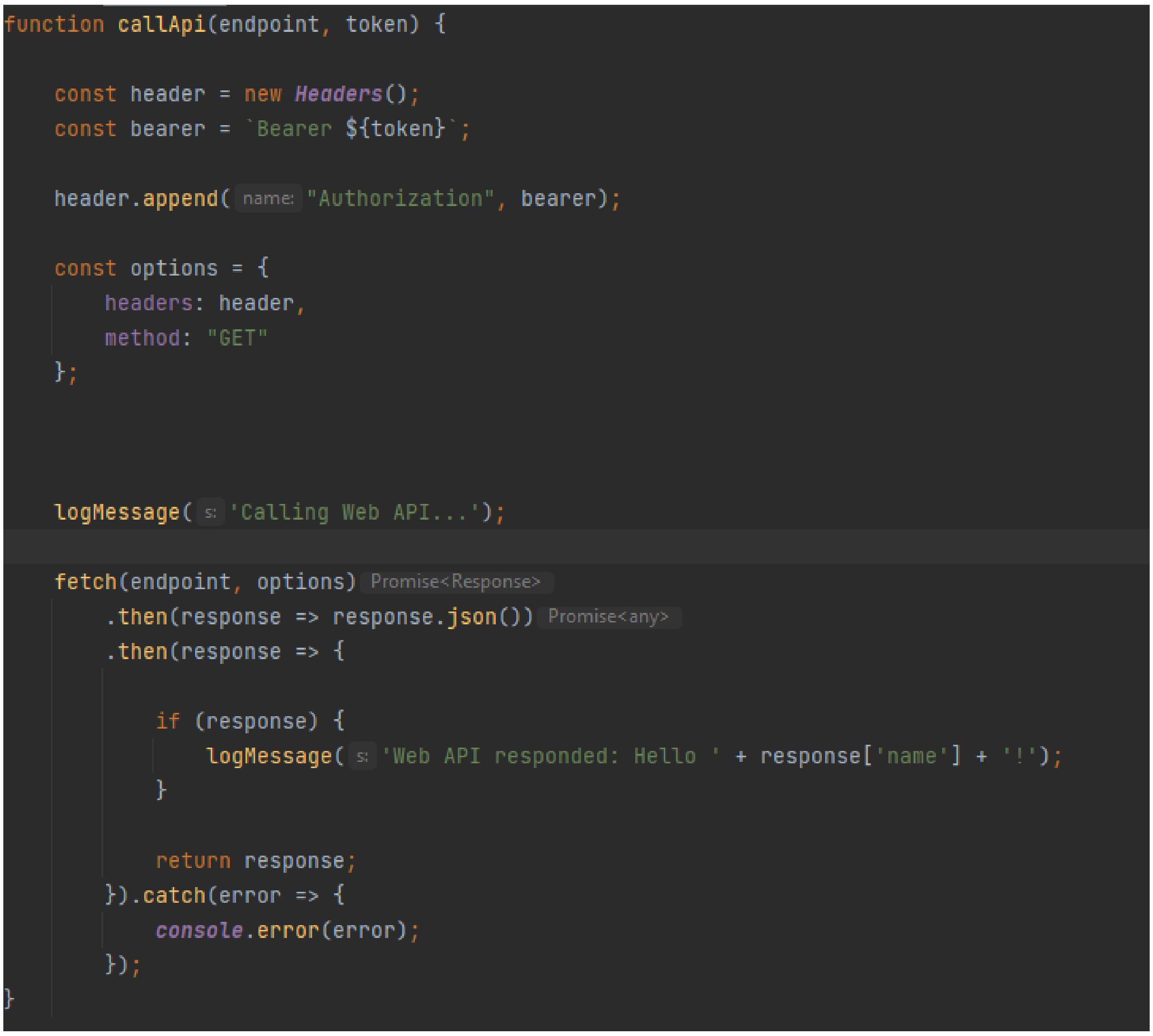
When a user successfully logged in to our web application, the application received this token (see figure 18) from Azure.



```
▼Object
   accessToken: "eyJ0eXAiOiJKV1QiLCJub25jZSI6IklHTmVhbGNZd203V0RXV1h2YV1xN3F6bXZudkRHNmVHVkFVV1R2V1Fnc
 ▶ account: {homeAccountId: '00000000-0000-0000-6b62-ca4bc6fa7f5c.9188040d-6c67-4c5b-b112-36a304b66dad
   authority: "https://login.microsoftonline.com/ee1e34d2-5218-4a08-b4fc-c5b769d9189a/"
   cloudGraphHostName: ""
 ▶ expiresOn: Fri Apr 08 2022 14:25:46 GMT+0300 (östeuropeisk sommartid) {}
 ▶ extExpiresOn: Fri Apr 08 2022 15:43:40 GMT+0300 (östeuropeisk sommartid) {}
   familyId: ""
   fromCache: false
   idToken: "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImpTMVhvMU9XRGpfNTJ2YndHTmd2UU8yVnpNYyJ9.eyJh
 ▼ idTokenClaims:
     aud: "e2e6f93c-987a-4293-94b1-d019b4190910"
     exp: 1649416072
     iat: 1649412172
     idp: "https://sts.windows.net/9188040d-6c67-4c5b-b112-36a304b66dad/"
     iss: "https://login.microsoftonline.com/ee1e34d2-5218-4a08-b4fc-c5b769d9189a/v2.0"
     name: "Johannes-holmberg@outlook.com Holmberg"
     nbf: 1649412172
     nonce: "2bfca32c-a5d6-4eb3-8466-649e6263ab06"
     oid: "880c1077-a3c0-4887-a5f1-2ca5338aee08"
     preferred_username: "Johannes-holmberg@outlook.com"
     rh: "0.AUYA0jQe7hhSCEq0_MW3adkYmjz55uJ6mJNClLHQGbQZCRCAAJY."
     sub: "uhFCGIqROFy7SxlRkORsYPz7ykM-vmSA3wb89NGf83w"
     tid: "ee1e34d2-5218-4a08-b4fc-c5b769d9189a"
     uti: "EL9eaaqaZUyvCemvCOUhAA"
     ver: "2.0"
   ▶ [[Prototype]]: Object
   msGraphHost: ""
 ▶ scopes: (5) ['Mail.Read', 'openid', 'profile', 'User.Read', 'email']
   state: ""
   tenantId: "ee1e34d2-5218-4a08-b4fc-c5b769d9189a"
   tokenType: "Bearer"
   uniqueId: "880c1077-a3c0-4887-a5f1-2ca5338aee08"
```

Figure 18. Showing the token we are receiving after successfully logging in

Using the token, another call can be made to get resources that are available for this specific user, identified by the given token. The next step is to try to access resources at our backend by clicking "Call web API" which initiates the "callApi" function (see figure 19).

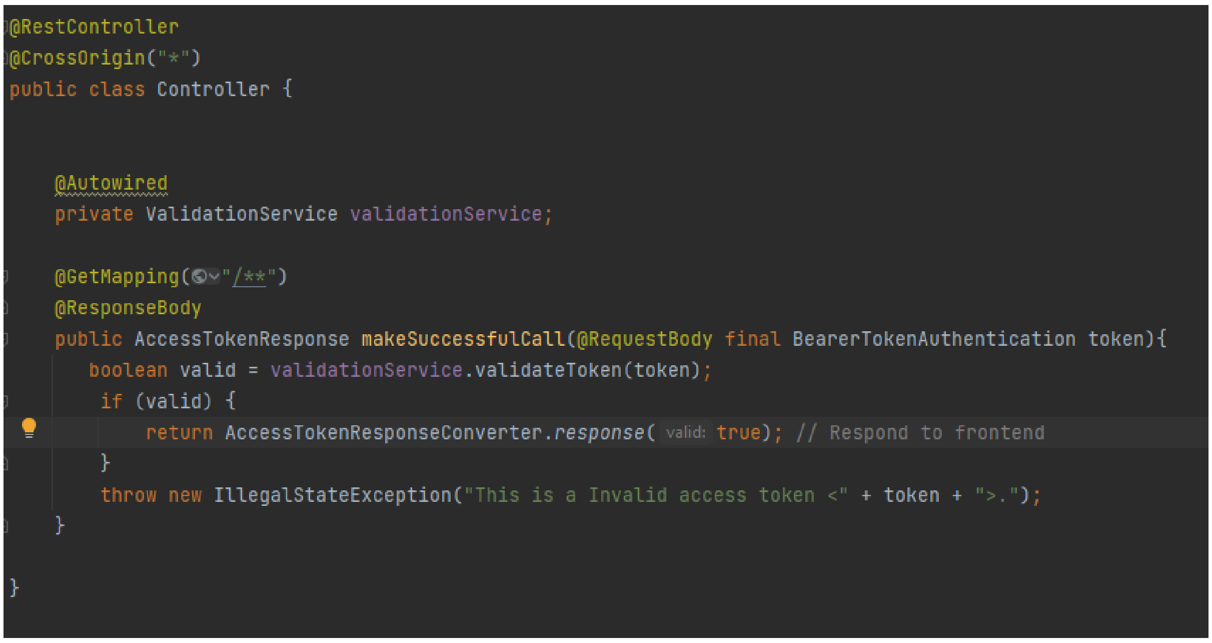


```
function callApi(endpoint, token) {

    const header = new Headers();
    const bearer = `Bearer ${token}`;

    header.append( name: "Authorization", bearer);

    const options = {
        headers: header,
        method: "GET"
    };



    logMessage( s: 'Calling Web API...');

    fetch(endpoint, options) Promise<Response>
        .then(response => response.json()) Promise<any>
        .then(response => {

            if (response) {
                logMessage( s: 'Web API responded: Hello ' + response['name'] + '!');
            }

            return response;
        }).catch(error => {
            console.error(error);
        });
}
```

Figure 19. Showing our frontend calling the backend API

This function sets the Authorization header with the token, as discussed in chapter 2.9.3. What actually happens in the call API function, is that a new header is added to the response to the backend, sent from the user/frontend (as seen in figure 19). By sending the token/bearer, another API can consume data in another call.

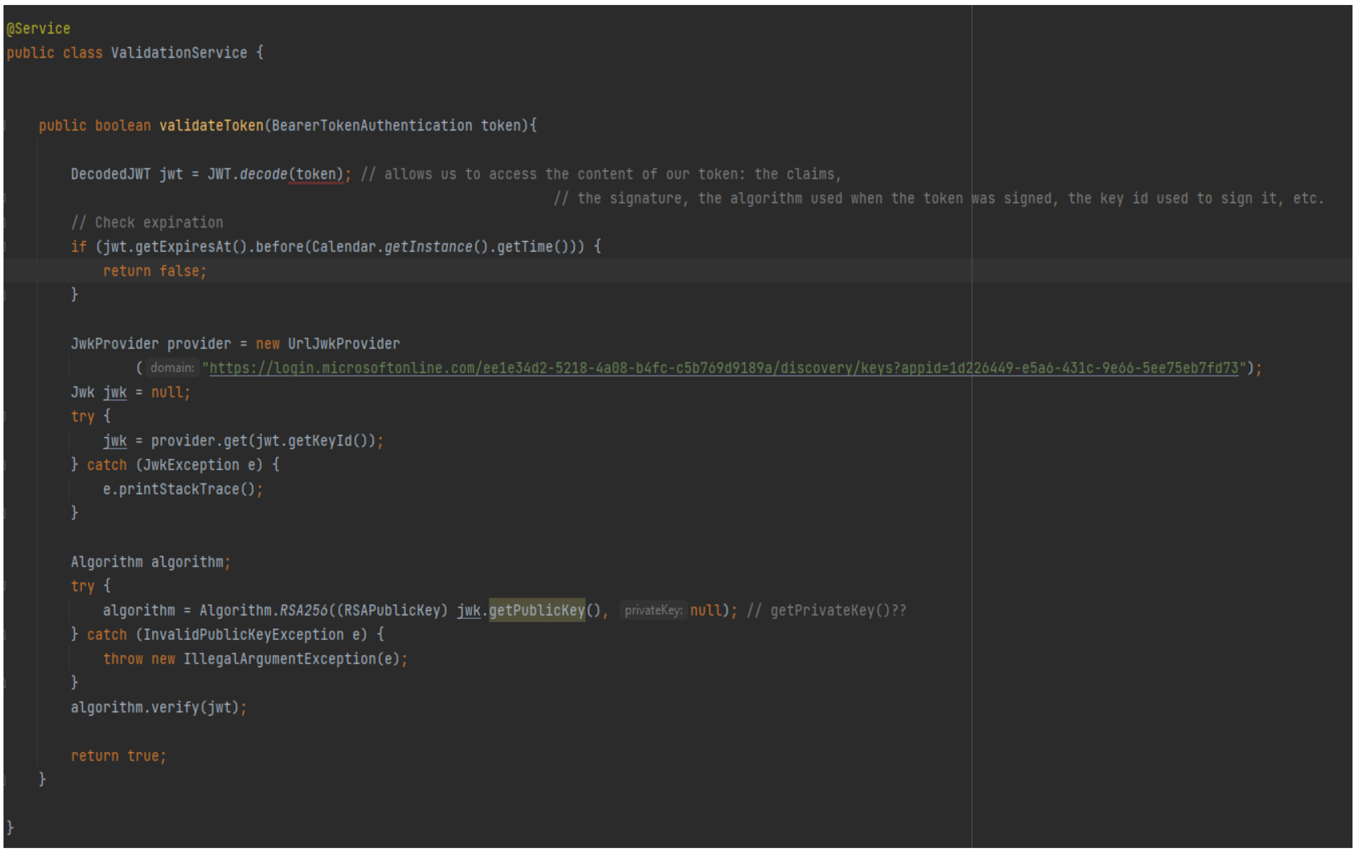
Now our Controller (see figure 20) picks up the token sent from our frontend, the object *BearerObjectAuthentication* named *token*, and consumes the given secret to make the next step, controlling the access token validity.



```
@RestController
@CrossOrigin("*")
public class Controller {


    @Autowired
    private ValidationService validationService;

    @GetMapping("/**")
    @ResponseBody
    public AccessTokenResponse makeSuccessfulCall(@RequestBody final BearerTokenAuthentication token){
        boolean valid = validationService.validateToken(token);
        if (valid) {
            return AccessTokenResponseConverter.response( valid: true); // Respond to frontend
        }
        throw new IllegalStateException("This is a Invalid access token <" + token + ">.");
    }

}
```

Figure 20. Showing our backend controller

The following step is trying to validate the token in our function “validateToken” (see figure 21).

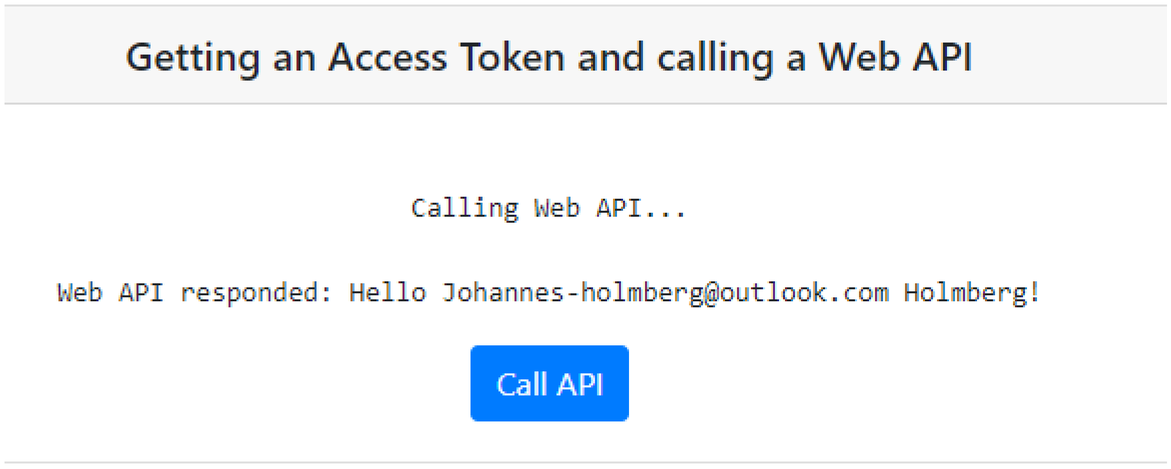


```
@Service
public class ValidationService {


    public boolean validateToken(BearerTokenAuthentication token){

        DecodedJWT jwt = JWT.decode(token); // allows us to access the content of our token: the claims,
                                           // the signature, the algorithm used when the token was signed, the key id used to sign it, etc.
        // Check expiration
        if (jwt.getExpiresAt().before(Calendar.getInstance().getTime())) {
            return false;
        }

        JwkProvider provider = new UrlJwkProvider
                ( domain: "https://login.microsoftonline.com/ee1e34d2-5218-4a08-b4fc-c5b769d9189a/discovery/keys?appid=1d226449-e5a6-431c-9e66-5ee75eb7fd73");
        Jwk jwk = null;
        try {
            jwk = provider.get(jwt.getKeyId());
        } catch (JwkException e) {
            e.printStackTrace();
        }

        Algorithm algorithm;
        try {
            algorithm = Algorithm.RSA256((RSAPublicKey) jwk.getPublicKey(),  privateKey: null); // getPrivateKey()??
        } catch (InvalidPublicKeyException e) {
            throw new IllegalArgumentException(e);
        }
        algorithm.verify(jwt);

        return true;
    }

}
```

Figure 21. Showing our backend validating the JWT token.

Unfortunately, my backend is not working correctly and I'm not able to validate my token. I've used jwt.io to make sure my token is valid, but I'm not sure how to break down my token to the correct format that is necessary for my function. I am however able to do this in my the frontend, but then I'm not able to pass it to my API since it is no longer a token.

So for demonstration purposes, I will use a Microsoft code example[6] so we can see what a final result would look like. Microsoft is however using JavaScript in their backend, so it is not what is requested from us. Figure 22 will show how a successful call would have looked.


Getting an Access Token and calling a Web API

```
Calling Web API...

Web API responded: Hello Johannes-holmberg@outlook.com Holmberg!
```

Figure 22. Showing a successful call to the backend with a response of the username of the caller.

[6] https://docs.microsoft.com/en-us/azure/active-directory/develop/sample-v2-code

5. CONCLUSION

Azure AD on paper seems to be the perfect way to distribute your resources within your organization and even to your customers. You can easily add and revoke access to groups or users. It's easy to add new resources at Azure, and you can easily implement Azure to your existing resources to connect them.

JWT might be one of the best ways to validate and send information between parties. It's almost impossible to modify anything within the token since it is signed, in our case by Azure. I say almost because, yes, you are able to modify the token, but it would no longer be valid after the modification and is therefore useless. Also if you would gain access to the secret signing key from Azure, then you are able to modify the token, and it would still be valid.

Since the backend right now is written in JavaScript we would like to change this to the desired Java Spring as requested by the customer.

REFERENCE LIST

Ajburnle. (n.d.). *What is Azure Active Directory?* Retrieved May 2, 2022, from https://docs.microsoft.com/en-us/azure/active-directory/fundamentals/active-directory-whatis

Alzomai, M. H. (2011). *Identity management : strengthening one-time password authentication through usability* [Phd, Queensland University of Technology]. https://eprints.qut.edu.au/46213/

Auth. (n.d.). *JSON Web Tokens*. Auth0 Docs. Retrieved April 26, 2022, from https://auth0.com/docs/secure/tokens/json-web-tokens

BarbaraSelden. (n.d.). *OpenID Connect authentication with Azure Active Directory*. Retrieved May 2, 2022, from https://docs.microsoft.com/en-us/azure/active-directory/fundamentals/auth-oidc

Chapter 4. Handling preflight requests. (n.d.). Retrieved May 10, 2022, from https://livebook.manning.com/book/cors-in-action/chapter-4/13

Cross-Origin Resource Sharing (CORS). (n.d.). Retrieved April 21, 2022, from https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Fielding, R. (1999). Hypertext transfer protocol-HTTP/1.1. IETF RFC 2616. *http://www.ietf.org/rfc/rfc2616.txt*. https://ci.nii.ac.jp/naid/10030667381/

Justinha. (n.d.). *Azure AD Multi-Factor Authentication overview*. Retrieved May 10, 2022, from https://docs.microsoft.com/en-us/azure/active-directory/authentication/concept-mfa-howitworks

Leiba, B. (2012). OAuth Web Authorization Protocol. *IEEE Internet Computing*, *16*(1), 74–77.

OpenID Connect Core 1.0 incorporating errata set 1. (n.d.). Retrieved May 2, 2022, from https://openid.net/specs/openid-connect-core-1_0.html

RFC 6749 - the OAuth 2.0 authorization framework. (n.d.). Retrieved May 2, 2022, from https://datatracker.ietf.org/doc/html/rfc6749

RFC 7519 - JSON Web Token (JWT). (n.d.). Retrieved May 2, 2022, from https://datatracker.ietf.org/doc/html/rfc7519

RFC 7636 - proof Key for Code Exchange by OAuth Public Clients. (n.d.). Retrieved April 21, 2022, from https://datatracker.ietf.org/doc/rfc7636/

rolyon. (n.d.-a). *Overview of Azure Active Directory role-based access control (RBAC)*. Retrieved May 9, 2022, from https://docs.microsoft.com/en-us/azure/active-directory/roles/custom-overview

rolyon. (n.d.-b). *What is Azure role-based access control (Azure RBAC)?* Retrieved May 2, 2022, from https://docs.microsoft.com/en-us/azure/role-based-access-control/overview

rwike. (n.d.). *Microsoft identity platform overview - Azure - Microsoft identity platform*. Retrieved April 21, 2022, from https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-overview

Spring Boot infographic - download form. (n.d.). JAX London 2022. Retrieved May 2, 2022, from https://jaxlondon.com/spring-boot-infographic-download-form/

Understanding the three factors of authentication. (n.d.). Retrieved May 2, 2022, from https://www.pearsonitcertification.com/articles/article.aspx?p=1718488

Warren, J., Roddick, J., Steketee, C., Brankovic, L., Coddington, P., & Wendelborn, A. (2007). *Usability and Privacy in Identity Management Architectures* (J. Warren, J. Roddick, C. Steketee, L. Brankovic, P. Coddington, & A. Wendelborn (Eds.); pp. 143–152). Australian Computer Society.

What is OAuth 2.0 and what does it do for you? (n.d.). auth0. Retrieved May 2, 2022, from https://auth0.com/intro-to-iam/what-is-oauth-2/