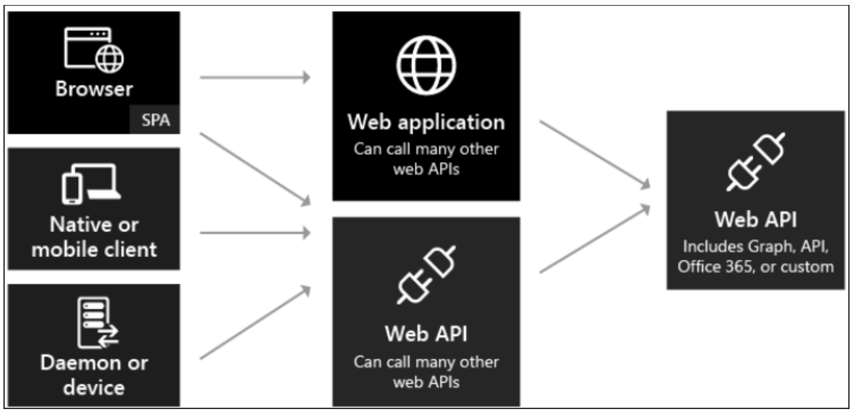
Security:

- Authentication and Authorization
- Understanding OAuth2 and OpenIdConnect
- Authenticate with JWT Bearer Token
- Using IdentityServer4
- Authenticating between microservices
- Implementing Role based and Policy based Authorization

The Big Picture

Most modern applications look more or less like this:


Browser
SPA
Native or mobile client
Daemon or device
Web application
Can call many other web APIs
Web API
Can call many other web APIs
Web API
Includes Graph, API, Office 365, or custom

The most common interactions are:

- Web browser to web application: A user needs to sign in to a web application that is secured by Azure AD.
- Single-page application (SPA): A user needs to sign in to a single-page application that is secured by Azure AD.
- Web application to web API: A web application needs to get resources from a web API secured by Azure AD.
- Native application to web API: A native application that runs on a phone, tablet, or PC needs to authenticate a user to get resources from a web API that is secured by Azure AD.
- Daemon or server application to web API: A daemon application or a server application with no web user interface needs to get resources from a web API secured by Azure AD.

API Access

Applications have two fundamental ways with which they communicate with APIs – using the application identity, or delegating the user's identity. Sometimes both methods need to be combined.

Basic Authentication Flow

In Basic Authentication username and password are passed in HTTP Header

End User Login Process:

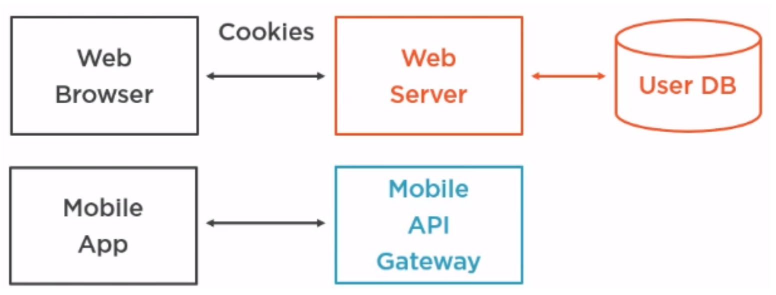
1. User visits login page
    a. Provides Username and Password
2. Username and password are sent to server
    a. Encrypted in transit (HTTPS)

   b. Authorization Header eg: Basic dXn1cjpasdfsasdf= (Basic Access Authentication with encrypted u/p)

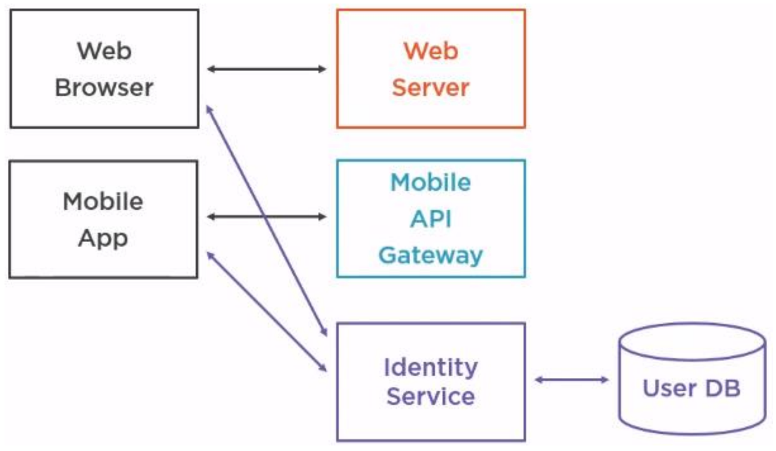
3. Server Validates the Password
   a. Database lookup
   b. Passwords are stored as hashes
4. Web server issues a cookie
   a. Stored in the browser
   b. Sent with every subsequent HTTP request
   c. Server validates the cookie

Problems with Basic Authentication in using for Microservices

1. Managing user credentials is complex and we have to deal with all of the following:
   - Password Hashing
   - User Registration
   - Password Reset
   - Brute force attack detection
   - Two Factor Authentication
2. Multiple microservices need authentication

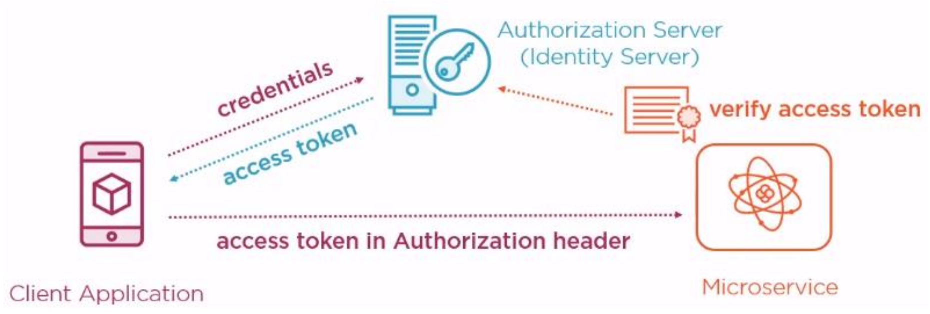


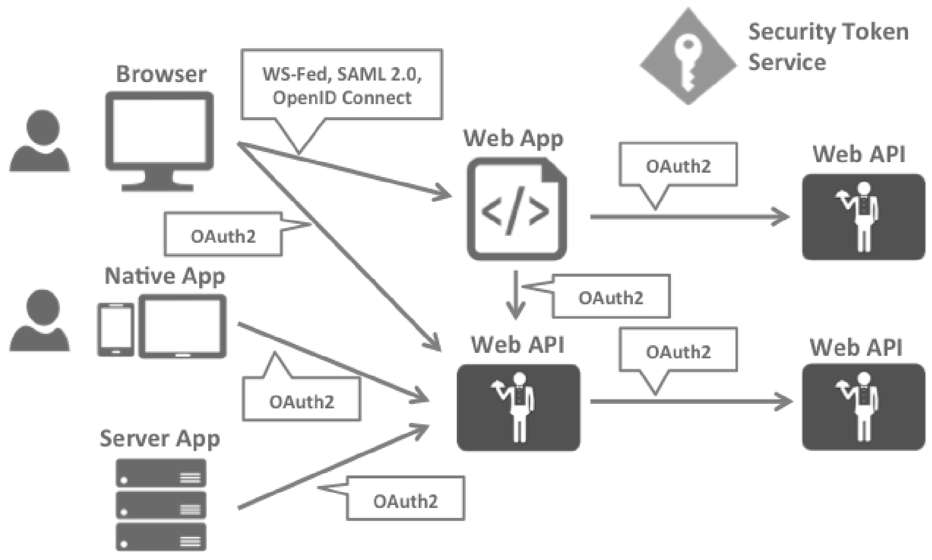
Ideally, we should delegate authentication to **third party identity service** which can handle all the concerns and simply tell us who is calling our microservice. There are some standard protocols which will help us in achieve this.



Solution: Authorization Server / Security Token Service

Restructuring the application to support an Authorization server or Security Token Service leads to the following architecture and protocols:





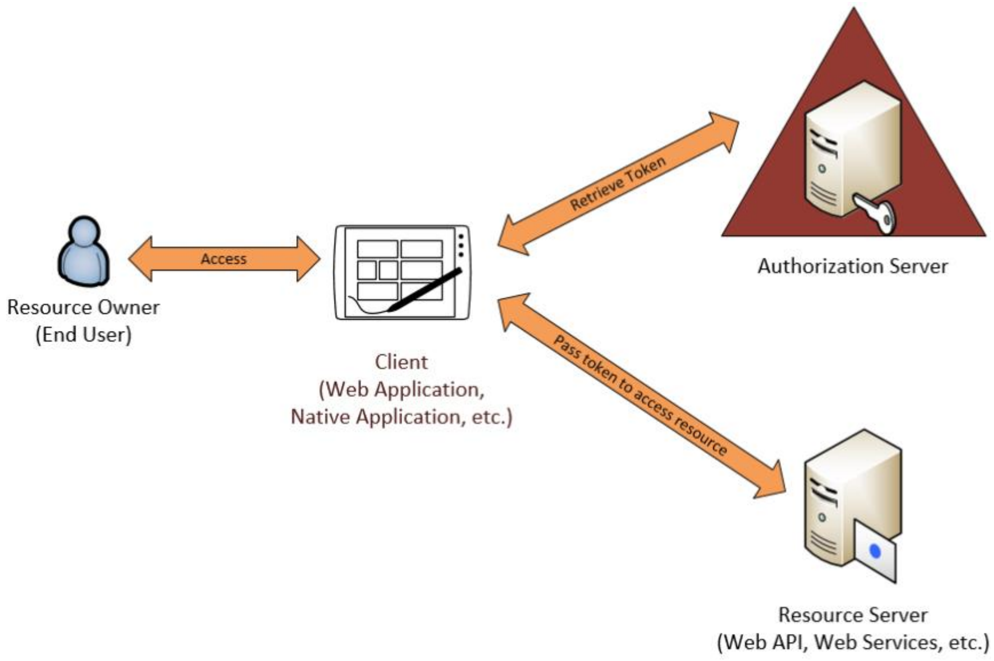
OAuth 2.0

- OAuth stands for Open Authorization. It is a free and open protocol.
- It works by **delegating user authentication** to the service that hosts the user account and authorizing third-party applications to access the user account. Example: Facebook, Twitter, Google, Microsoft Account, Linked-In etc…
- OAuth 2 provides authorization flows for **web, desktop applications** and **mobile** devices.
- It allows users (1st Party) to share their private resources with third party while keeping their own credentials secret from the third party. These resources could be the digital assets like photos, videos, contact lists and are usually stored with another service provider (2nd . OAuth does this by granting the requesting client applications a token after access is approved by the user. Each token grants limited access to a specific resource for a specific period.

Actors involved in OAuth 2:

1) **User / Resource Owner:** The resource owner is the **end user** who is giving access to some portion of his/her account.

2) **Authorization Server**: The server where the client application is registered and returns the access token after it gets consent from the resource owner for accessing protected resources hosted by a resource server.
3) **Resource Server:** The Web API or Web Service server which hosts the secured users protected resources and are protected by OAuth2. The resource server validates the access-token and serves the protected resources. Eg: Photo Sharing site, online bank service or any other service where **users private stuff** is kept.
4) **Third party Client Application:** The application that is attempting to get access to the user's account or a resource form Resource Server. It can be website, desktop or mobile application or a set-top box or anything connected to the web. Eg: Photo Printing Application/Website,



Authorization Grant Types

The authorization grant type depends on the method used by the application to request authorization, and the grant types supported by the API.

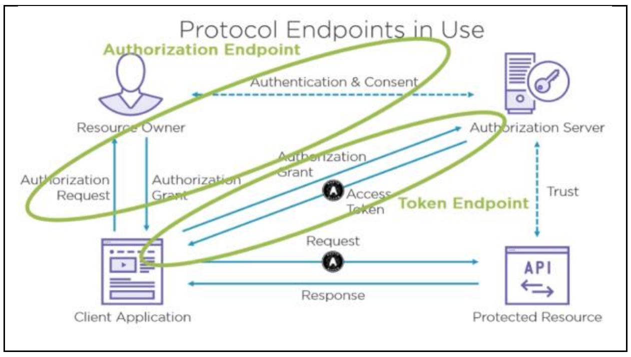
OAuth 2 defines four grant types, each of which is useful in different cases:

1. **Authorization Code.**
2. **Implicit**
3. **Client Credentials**
4. **Resource Owner Password Credentials (ROPC)**

1. Grant Type: Authorization Code

The authorization code grant type is the most commonly used because it is optimized **for server-side (web based) applications**, where source code is not publicly exposed, and Client Secret confidentiality can be maintained. This is a **redirection-based** flow, which means that the application must be capable of **interacting** with the user-agent (i.e. the user's **web browser**) and receiving API authorization codes that are routed through the user-agent.

The example explained above is Authorization Code Grant Type



Application Registration

Before using OAuth with our application, we must register our application with the Authorization Server. This is done through a registration form in the "developer" or "API" portion of the service's website, where we will provide the following information (and probably details about our application):

- Application Name
- Application Website
- Redirect URI or Callback URL

The redirect URI is where the service will redirect the user after they authorize (or deny) your application, and therefore the part of your application that will handle authorization codes or access tokens.

Client ID and Client Secret

Once our application is registered, the service will issue "client credentials" in the form of a client identifier and a client secret. The Client ID is a **publicly exposed** string that is used by the Authorization Server to identify the client application. The Client Secret is used to **authenticate the identity** of the client application to the service API when the application requests to access a user's account and must be kept private between the client application and the Authorization Server.

Step 1: Authorization Code Link:

The client application redirects the user to authorization server.

First, the user is given an authorization code link that looks like the following:

https://oauth.server.com/authorize?

response_type=code& -- code is fixed and means Authorization Code GrantType

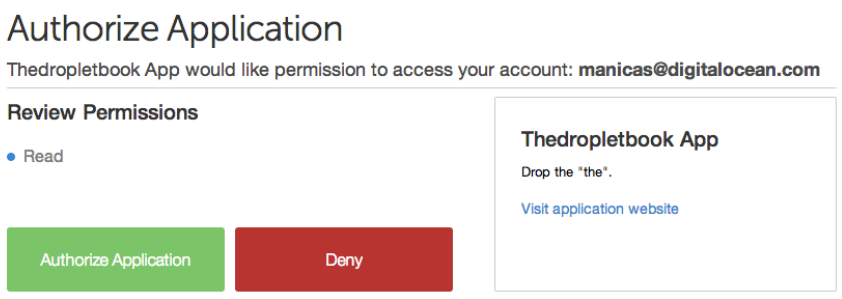
client_id=<CLIENT_ID>&

redirect_uri=https://www.clientapp.com/callback&

scope=api1.read, api2&

state=xyz

Step 2: User Authorizes Application


Authorize Application

Thedropletbook App would like permission to access your account: **manicas@digitalocean.com**

Review Permissions

- Read

Thedropletbook App

Drop the "the".

Visit application website

Step 3: Application Receives Authorization Code

If the user clicks "Authorize Application", the service redirects the user-agent to the application **redirect URI**, which was specified during the client registration, along with an *authorization code*. The redirect would look something like this:

https://www.clientapp.com/callback?code=<AUTHORIZATION_CODE>**&state**=xyz

Step 4: Swap the Authorization Code with Access Token

The application requests an access token from the API, by passing the authorization code along with authentication details, including the *client secret*, to the API token endpoint.

Here is an example request token endpoint:

Request Method: POST

content-type=application/x-www.form-urlencoded

Request URL: https://oauth.server.com/token?

client_id=CLIENT_ID**&**

client_secret=CLIENT_SECRET&

grant_type=**authorization_code**&

code=<AUTHORIZATION_CODE>&

redirect_uri=CALLBACK_URL

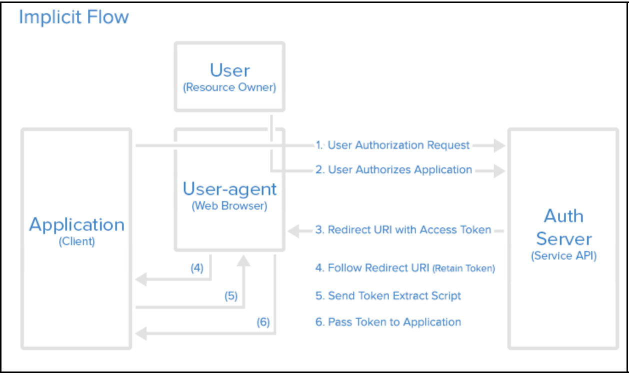
Step 5: Application Receives Access Token in response

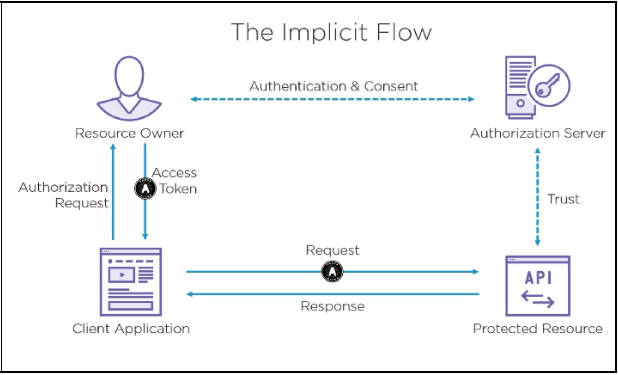
```
{
        "access_token":"<ACCESS_TOKEN>",
        "token_type":"bearer",
        "expires_in":2592000,
        "scope":"api1.read, api2",
        "uid":100101,
        "info": {"name":"Sandeep","email":"sandeepsoni@deccansoft.com"}
}
```

Step 6: Call the secure Resource Server API using the access token**.**

Grant Type: Implicit

The implicit grant type is used for **mobile apps and SPA web applications** where the client secret confidentiality is not guaranteed. The implicit grant type is also a redirection-based flow but the access token is given to the user-agent (browser) to forward to the application, so it may be exposed to the user and other applications on the user's device. Also, this flow does not authenticate the identity of the application, and relies on the redirect URI (that was registered with the service) to serve this purpose.


Implicit Flow
User
(Resource Owner)
Application
(Client)
User-agent
(Web Browser)
Auth
Server
(Service API)
1. User Authorization Request
2. User Authorizes Application
3. Redirect URI with Access Token
(4)
(5)
(6)
4. Follow Redirect URI (Retain Token)
5. Send Token Extract Script
6. Pass Token to Application



Note that Access Token is directly issued in Response to Client Application

Step 1: Implicit Authorization Link

https://oauth.server.com/authorize?

response_type=token&

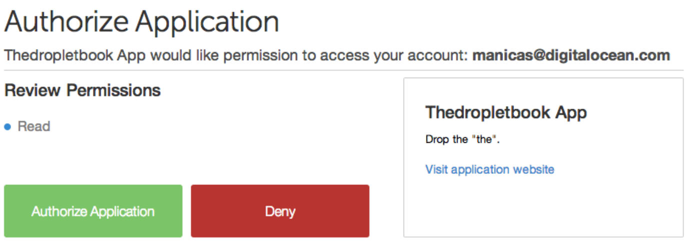
client_id=CLIENT_ID**&**

redirect_uri=https://www.clientapp.com/callback&

scope=ap1.read, api2&

state=abc

Step 2: After the user login the Authorization server will show the below content dialog.


Authorize Application

Thedropletbook App would like permission to access your account: **manicas@digitalocean.com**

Review Permissions

- Read

Authorize Application | Deny

Thedropletbook App

Drop the "the".

Visit application website

Step 3: User-agent Receives Access Token with Redirect URI

https://www.clientapp.com/callback?

access_token=ACCESS_TOKEN**&**

expires_in=3600&

token_type=example&

state=abc

Step 4: Call the secure Resource Server API using the access token**.**

Security Concerns with Implicit Grant time:

1. Access tokens exposed to resource owner.
2. Access tokens accessible to 3rd Party JavaScript.
3. No way to validate that access tokens are intended for client.

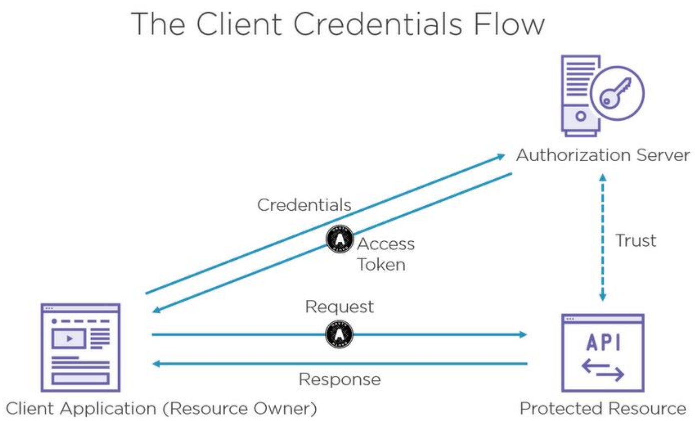
Grant Type: Client Credentials

Designed for client application who are resource owner. Eg: Console Application or Windows Services

Best for machine to machine (service to service) communication

The **client credentials grant type** provides an application a way to access its own service account.

Example: An application wants to update its registered description or redirect URI, or access other data stored in its service account via the API.



Step 1: Sent a request to Authorization Server

Method: POST

Content-Type: application/x-www.form-urlencoded

URL: https://oauth.example.com/token?

grant_type=client_credentials&

client_id=CLIENT_ID&

client_secret=CLIENT_SECRET

scope: api1.read

Step 2: User-agent Receives Access Token with Redirect URI

{

access_token=<ACCESS_TOKEN>**&**

expires_in=3600&

token_type=Bearer&

scope=api1.read

}

Step 3: Call the secure Resource Server API using the access token (with identity of client application)**.**

If the application credentials check out, the authorization server returns an access token to the application. Now the application is authorized to use its own account!

Grant Type: Resource Owner Password Credentials (ROPC)

Designed as a stop-gap for legacy applications.

Negates most of the benefits of OAuth.

Should not be used frequently and is deprecated.

The user provides their service credentials (username and password) directly to the client application which uses the credentials to obtain an access token from the Authorization Server. This grant type should only be enabled on the authorization server if other flows are not viable. It should only be used if the application is trusted by the user (e.g. it is owned by the service, or the user's desktop OS).

Step 1: Authorization Link

Method: **POST**

Content-Type: application/x-www.form-urlencoded

https://oauth.example.com/token?

grant_type=password&

username=USERNAME&

password=PASSWORD&

scope=api1.read, api2

Step 2: Application receives Access Token with Redirect URI

{

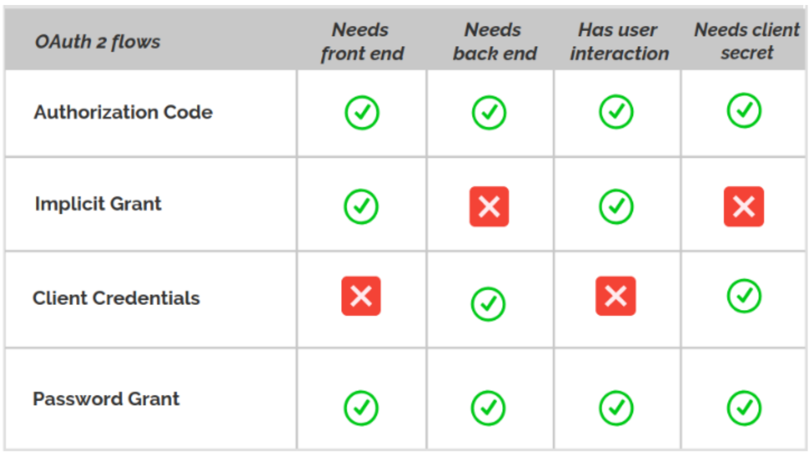
access_token=ACCESS_TOKEN**&**

expires_in=3600&

token_type=Bearer&

scope=api1.read

}



OAuth 2 flows	*Needs front end*	*Needs back end*	*Has user interaction*	*Needs client secret*
Authorization Code	✓	✓	✓	✓
Implicit Grant	✓	✗	✓	✗
Client Credentials	✗	✓	✗	✓
Password Grant	✓	✓	✓	✓

Grant Type = Refresh Tokens

- Swap for new tokens
- Allows for long-lived access
- Highly confidential
- User should be informed that refresh token is generating (model dialog)

Note that Refresh Tokens can be used by Only Grant Type: Authorization Code and Resource Owner Password Credentials (ROPC)

Step 1: Authorization Code Link: First, the user is given an authorization code link that looks like the following:

https://oauth.server.com/authorize?

response_type=code&

client_id=<CLIENT_ID>&

redirect_uri=https://www.clientapp.com/callback&

scope=api1.read, a, **offline_access** &

state=xyz

//….all steps…

Step 2: Application Receives Access Token

```
{
        "access_token":"ACCESS_TOKEN",
        "token_type":"bearer",
        "expires_in":2592000,
        "refresh_token":"REFRESH_TOKEN",
        "scope":"api1.read offline_access",
}
```

Step 3: Refresh Token Request

Method: POST

Content-Type: application/x-www.form-urlencoded

Authorization: Bearer czXcaGresasf33ljasdf

https://oauth.server.com/token?

grant_type=refresh_token&

refresh_token=<REFRESH_TOKEN>&

scope=api1.read,api2

Step 4: Application Receives Access Token

```
{
        "access_token":"NEW ACCESS_TOKEN",
        "token_type":"bearer",
        "expires_in":2592000,
        "refresh_token":"<NEW REFRESH_TOKEN>",
        "scope":"api1.read offline_access"
}
```

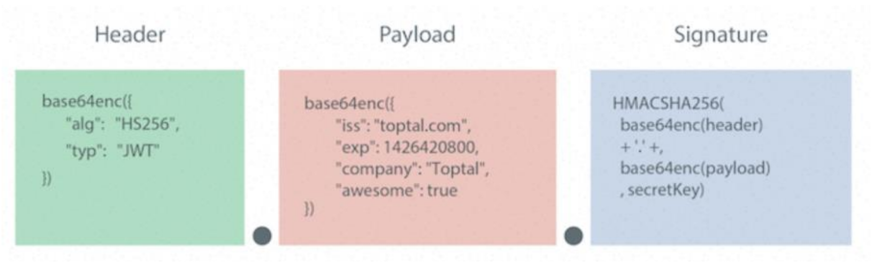
About JWT Token

JWT are an important piece in ensuring trust and security in your application. JWT allow claims, such as user data, to be represented in a secure manner.

A JSON Web Token (JWT) is a JSON object that is defined in RFC 7519 as a safe way to represent a set of information between two parties.

The token is composed of a header, a payload, and a signature.

Format of JWT Token = header.payload.signature



JSON Web Token example:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODVlLCJjb21wYW55IjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0.
yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK_ZXw

Steps to Create a JWT Token and Authenticate User:

Step 1. Create the HEADER

The header component of the JWT contains information about how the JWT signature should be computed.

JWT Header

```
{
  “alg”: “HS256”,
  “typ”: “JWT”
}
```

This JWT Header declares that the encoded object is a JSON Web Token, and that it is signed using the HMAC SHA-256 algorithm.

Once this is base64 encoded, we have the first part of our JWT.

Step 2. Create the PAYLOAD

In the context of JWT, a claim can be defined as a statement about an entity (typically, the user), as well as additional meta data about the token itself. The claim contains the information we want to transmit, and that the server can use to properly handle authentication. There are multiple claims we can provide;

These claims are **not** intended to be **mandatory** but rather to provide a starting point for a set of useful, interoperable claims.

- **exp**: Token expiration time defined in Unix time
- **iss**: The issuer of the token
- **sub**: The subject of the token
- **aud**: The audience of the token
- **nbf**: "Not before" time that identifies the time before which the JWT must not be accepted for processing
- **iat**: "Issued at" time, in Unix time, at which the token was issued
- **jti**: JWT ID claim provides a unique identifier for the JWT

Example Payload

```
{
  "iss": "toptal.com",
  "exp": 1426420800,
  "company": "deccansoft",
  "awesome": true
}
```

Step 3. Create the SIGNATURE

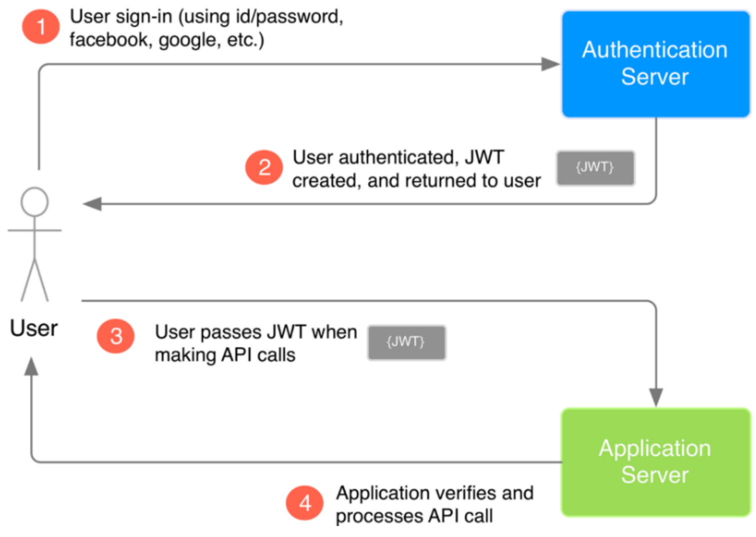
The signature is computed using the following pseudo code:

```
data = base64urlEncode(header) + "." + base64urlEncode(payload)
hashedData = HMACSHA256(data, secret)
```

Step 4: Create JWT Token

```
JWT Token = base64urlEncode( header ) + "." + base64urlEncode( payload ) + "." + base64urlEncode( hashedData )
```

How an application uses JWT to verify the authenticity of a user.


1 User sign-in (using id/password, facebook, google, etc.)
Authentication Server
2 User authenticated, JWT created, and returned to user
{JWT}
User
3 User passes JWT when making API calls
{JWT}
Application Server
4 Application verifies and processes API call

Open ID Connect (OIDC)

With OIDC, you can use a trusted external provider to prove to a given application that **you are who you say you are**, without ever having to grant that application access to your credentials.

OAuth 2.0 leaves a lot of details up to implementers. For instance, it supports scopes, but scope names are not specified. It supports access tokens, but the format of those tokens is not specified.

It's an Identity Layer on top of OAuth2.0

OpenID Connect *has become the leading standard for single sign-on and identity provision on the Internet.*
Its formula for success: *simple JSON-based identity tokens (JWT), delivered via OAuth 2.0 flows designed for web, browser-based and native / mobile applications.*
With OIDC, a number of specific scope names are defined that each produce different results.

About Scopes

Scopes are space-separated lists of identifiers used to specify what access privileges are being requested.

Built-In Scopes in OpenIDConnect

1. **openid**: This is mandatory scope
2. **profile**: requests access to **default profile*** claims
3. **email**: requests access to email and email_verified claims
4. **address**: requests access to address claim
5. **phone**: requests access to phone_number and phone_number_verified claim

Custom Scope Examples: deccansoft_api, deccansoft_api.read, deccansoft_api.write, etc..

*The **default profile** claims are:

- name
- family_name
- given_name
- middle_name
- nickname
- preferred_username
- profile
- picture
- website
- gender
- birthdate
- zoneinfo
- locale
- updated_at

Claims are name/value pairs that contain information about a user, as well meta-information about the OIDC service.

Open Id Tokens:

OIDC has Access Tokens and ID Tokens and since the specification dictates the token format, it makes it easier to work with tokens across implementations.

Token Types

1. id_token
2. access_token

id_token:

- The ID Token is a security token that contains **Claims** data about the **user** as saved with the Authentication Server.
- The ID Token is represented as a **JSON Web Token (JWT).**
- For example, if there's a client app that uses Google to log in users and to sync their calendars, Google sends an ID Token to the app that includes information about the user. The app then parses the token's contents and uses the information (including details like name and profile picture) to customize the user experience.
- ID Tokens should ***not*** be used to gain access to an API.

Identity Token – Header

```
{
        "alg": "RS256",
        "kid": "234234324324dfd34234sd3",
        "typ": "JWT"
}
```

Identity Token Payload

```
{
"nbf": 1511121000,                         //No Before
"exp": 1511281970,                         //Expiry
"iat": 1511121000,                         //Issued At
"auth_time": 1511121025,                    //When the end user actually authenticated
"iss": "https://oauth.server.com",         //Who has issued the token
"aud": "CLIENT_ID",                        //Intended Audience
"nonce": "n-0S6_WzA2Mj",                   //No more than once – This is like state parameter
"sub": "244sdfds00dfdfd320sdfdfd3433",     //Unique Identified for the user
“Idp”: "AuthServer"                        //Identity Provider who issued the Identity
"family_name": "Soni",
"given_name": "Sandeep",
"groups": [
  "Training",
  "Everyone"
],
"locale": "en-US",
"name": "Sandeep Soni",
"preferred_username": "sandeepsoni@deccansoft.com",
}
```

Among the claims encoded in the id_token is an expiration (**exp**), which **must be honored** as part of the validation process.

This token authenticates the user to the application. The **audience** (the aud claim) of the token is set to the application's identifier, which means that only this specific application should consume this token.

Access Tokens

- Access Tokens (which aren't always JWTs) are used to inform an API that the bearer of the token has been authorized to access the API and perform a predetermined set of actions (specified by the scopes granted).
- Access tokens are used as **bearer tokens**. A bearer token means that the bearer (who hold the access token) can access authorized resources without further identification. Because of this, it’s important that bearer tokens are protected.
- **If I can somehow get a hold of and “bear” *your* access token, I can pretend as you**.
- **In the Google example**, Google sends an Access Token to the app after the user logs in and provides consent for the app to read or write to their Google Calendar. Whenever the app wants to write to Google Calendar, it sends a request to the Google Calendar API, including the Access Token in the HTTP **Authorization** header.

- These tokens usually have a short lifespan (dictated by its expiration) for improved security. That is, when the access token expires, the user must authenticate again to get a new access token limiting the exposure of the fact that it’s a bearer token.
- **Access Tokens must *never* be used for authentication**. Access Tokens cannot tell if the user has authenticated. The only user information the Access Token possesses is the **user ID**, located in the **sub** claim.
- Your applications should treat Access Tokens as ***opaque strings*** since they are meant for APIs. Your application should *not* attempt to decode them or expect to receive tokens in a particular format.

```
{
  "iss": "https://oauth.example.com/",
  "sub": "244sdfds00dfdfd320sdfdfd3433",
  "aud": [
   "my-api-identifier",
   "https://www.clientapp.com/userinfo"
  ],
  "azp": "YOUR_CLIENT_ID",
  "exp": 1489179954,
  "iat": 1489143954,
  "scope": "openid profile email address phone read:appointments"
}
```

Note that the token does not contain any information about the user itself besides their ID (sub claim). It only contains authorization information about which actions the application is allowed to perform at the API (scope claim). This is what makes it useful for securing an API, but not for authenticating a user.

Response Type: (Same as Grant Type in OAuth2)

1. **Authorization:** response_type=**code**
2. **Implicit:** response_type=**id_token token** OR **response_type=id_token**
3. **Hybrid Flow:** response_type=**code token id_token (Recommended)**

Security in ASP.NET Core

In ASP.NET Core, authentication is handled by the **IAuthenticationService**, which is used by authentication middleware.

The authentication service uses registered **authentication handlers** to complete authentication-related actions.

Examples of authentication-related actions include:

- Authenticating a user.
- Responding when an unauthenticated user tries to access a restricted resource.

The following code **registers authentication services and handlers** for JWT bearer authentication schemes:

```
services.AddAuthentication(options =>
{
   options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme; //Bearer
   options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(x =>
{
   x.RequireHttpsMetadata = false;
   x.SaveToken = true;
   x.TokenValidationParameters = new TokenValidationParameters
   {
      IssuerSigningKey = new SymmetricSecurityKey(key),
      ValidateIssuerSigningKey = true,
      ValidateIssuer = false,
      ValidIssuers = new [] {"issuer1", "issuer2"},
      ValidateAudience = true,
      ValidAudiences = api1,
   };
});
```

The registered authentication handlers and their configuration options are called "**schemes**". Schemes are useful as a mechanism for referring to the **authentication, challenge, and forbid** behaviors of the associated handler. For example, an authorization policy can use scheme names to specify which authentication scheme should be used to authenticate the user. When configuring authentication, it's common to specify the default authentication scheme. The default scheme is used unless a resource requests a specific scheme.

Authenticate

An authentication scheme's authenticate action is responsible for constructing the user's identity based on request context.

Authenticate examples include:

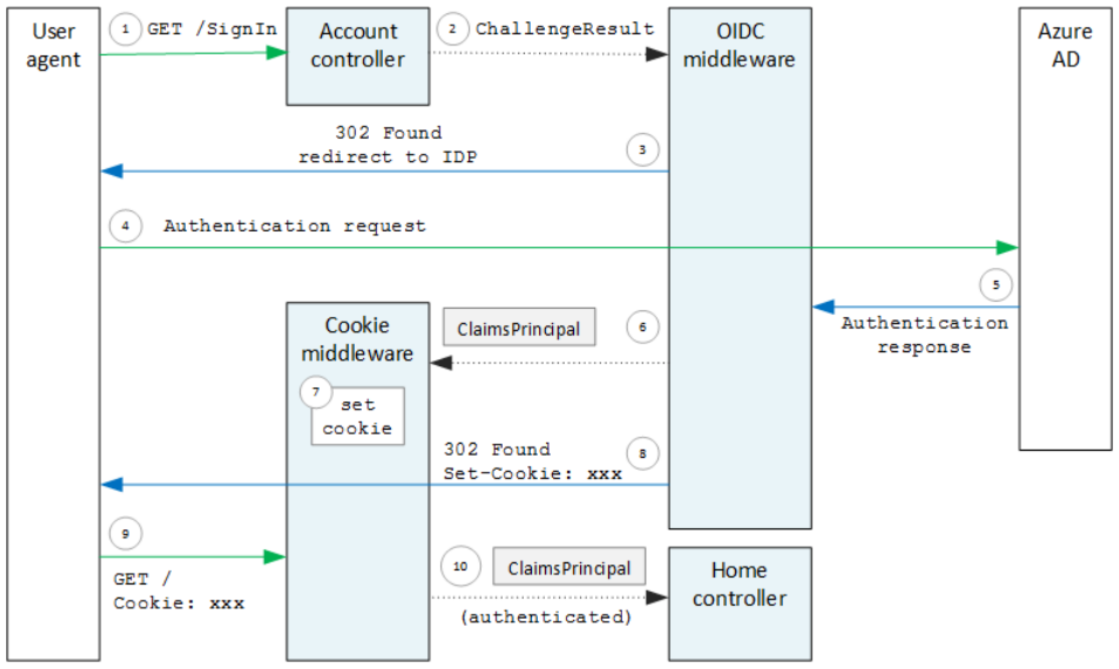
- A cookie authentication scheme constructing the user's identity from cookies.
- A JWT bearer scheme deserializing and validating a JWT bearer token to construct the user's identity.

Challenge

An authentication challenge is invoked by Authorization when an unauthenticated user requests an endpoint that requires authentication.

Authentication challenge examples include:

- A cookie authentication scheme redirecting the user to a login page.
- A JWT bearer scheme returning a 401 result with a www-authenticate: bearer header.



Following is a brief comparison of the various protocols:

- **OAuth vs OpenID Connect**: OAuth is used for authorization and OpenID Connect (OIDC) is used for authentication. OpenID Connect is built on top of OAuth 2.0, so the terminology and flow are similar between the two. You can even both authenticate a user (using OpenID Connect) and get authorization to access a protected resource that the user owns (using OAuth 2.0) in one request. For more information, see OAuth 2.0 and OpenID Connect protocols and OpenID Connect protocol.
- **OAuth vs SAML**: OAuth is used for authorization and SAML is used for authentication. See Microsoft identity platform and OAuth 2.0 SAML bearer assertion flow for more information on how the two protocols can be used together to both authenticate a user (using SAML) and get authorization to access a protected resource (using OAuth 2.0).
- **OpenID Connect vs SAML**: Both OpenID Connect and SAML are used to authenticate a user and are used to enable Single Sign On. SAML authentication is commonly used with identity providers such as Active Directory Federation Services (ADFS) federated to Azure AD and is therefore frequently used in enterprise applications. OpenID Connect is commonly used for apps that are purely in the cloud, such as mobile apps, web sites, and web APIs.

Securing WEB API using JWT Token – Basic Authentication

The good news is that authenticating with JWT tokens in ASP.NET Core is straightforward. Middleware exists in the **Microsoft.AspNetCore.Authentication.JwtBearer** package that does most of the work for us!

The following code registers authentication services and handlers for cookie and JWT bearer authentication schemes:

```
services.AddAuthentication(options =>
{
   options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme; //Bearer
   options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(x =>
{
   x.RequireHttpsMetadata = false;
   x.SaveToken = true;
   x.TokenValidationParameters = new TokenValidationParameters
   {
      ValidateIssuerSigningKey = true,
      IssuerSigningKey = new SymmetricSecurityKey(key),
      ValidateIssuer = false,
      ValidateAudience = false,
   };
});
```

Walkthrough:

1. **Create a New ASP.NET Core Web API Project**
2. **Edit appsettings.json**

```
{
   "Secret": "THIS IS USED TO SIGN AND VERIFY JWT TOKENS, REPLACE IT WITH YOUR OWN SECRET, IT CAN BE
ANY STRING"
}
```

3. Add the NuGet Package: "**Microsoft.AspNetCore.Authentication.JwtBearer**"
4. **Add the User, IUserSerivce and UserService classes to the project**

```
public class User
{
   public int Id { get; set; }
   public string Role { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Username { get; set; }
   public string Password { get; set; }
   public string Token { get; set; }
}
```

```
public interface IUserService
{
    User Authenticate(string username, string password);
    IEnumerable<User> GetAll();
}

public class UserService : IUserService
{
    // users hardcoded for simplicity, store in a db with hashed passwords in production applications
    private List<User> _users = new List<User>
    {
        new User { Id = 1, FirstName = "TestA", LastName = "UserA", Username = "testA", Password = "test",
Role="Admin" },
        new User { Id = 2, FirstName = "TestB", LastName = "UserB", Username = "testB", Password = "test",
Role="User" },
        new User { Id = 3, FirstName = "TestC", LastName = "UserC", Username = "testC", Password = "test",
Role="User" }
    };

    IConfiguration _config;;

  public UserService(IConfiguration config)
    {
        _config = cofig;
    }

    public User Authenticate(string username, string password)
    {
        var user = _users.SingleOrDefault(x => x.Username == username && x.Password == password);
        // return null if user not found
        if (user == null)
            return null;
        // authentication successful so generate jwt token
        var tokenHandler = new JwtSecurityTokenHandler();
        var key = Encoding.ASCII.GetBytes(_config["Secret"]);
        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(new Claim[]
            {
```

```
            new Claim(ClaimTypes.Name, user.Username.ToString()),
            new Claim(ClaimTypes.Role, user.Role),
            new Claim("aud","myclientid"),
            new Claim("iss","deccansoft.com")
         }),
         Expires = DateTime.UtcNow.AddDays(7),
         SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature)
      };
      var token = tokenHandler.CreateToken(tokenDescriptor);
      user.Token = tokenHandler.WriteToken(token);
      // remove password before returning
      user.Password = null;
      return user;
   }

   public IEnumerable<User> GetAll()
   {
      return _users.Select(x =>
      {
         x.Password = null;
         return x;
      });
   }
}
```

5. Add as API Controller to validate Username and Password and generate the JWT Token

```
[ApiController]
[Route("[controller]")]
public class UsersController : ControllerBase
{
   private IUserService _userService;
   public UsersController(IUserService userService)
   {
      _userService = userService;
   }

   [HttpGet("authenticate")]
   public IActionResult Authenticate(string username, string password)
```

```
    {
        var user = _userService.Authenticate(username, password);
        if (user == null)
            return BadRequest(new { message = "Username or password is incorrect" });
        return Ok(user);
    }

    [HttpGet]
    [Route("[action]")]
    [Authorize(Roles = "Admin")]
    public ActionResult<IEnumerable<User>> GetAll()
    {
        var users = _userService.GetAll();
        return Ok(users);
    }

    [HttpGet]
    [Route("[action]")]
    [Authorize()]
    public string GetUser()
    {
        return User.Identity.Name;
    }
}
```

6. Edit ConfigureService and Configure methods of Startup as below

```
using Microsoft.IdentityModel.Tokens;
using Microsoft.AspNetCore.Authentication.JwtBearer;

public void ConfigureServices(IServiceCollection services)
{
    // configure strongly typed settings objects
    // configure jwt authentication
    var key = Encoding.ASCII.GetBytes(Configuration["Secret"]);
    var auds = new List<string>() { "aud1", "aud2" };
    services.AddAuthentication(options =>
    {
        options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme; //Bearer
        options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(x =>
    {
        x.RequireHttpsMetadata = false;
        x.SaveToken = true;
        x.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(key),
            ValidateIssuer = false,
            ValidIssuer = "deccansoft.com",
            ValidateAudience = false,
            ValidAudiences = auds,
        };
    });

    // configure DI for application services
    services.AddScoped<IUserService, UserService>();
}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    . . .
    app.UseAuthentication();
    app.UseAuthorization();
    . . .
}
```

Try this in Postman

1. URL: http://localhost:5100/users/authenticate?username=testA&password=test
2. Call Secured Method: http://localhost:5100/users/getall

    Header: Authorization = Bearer **<Token from Previous Request>**