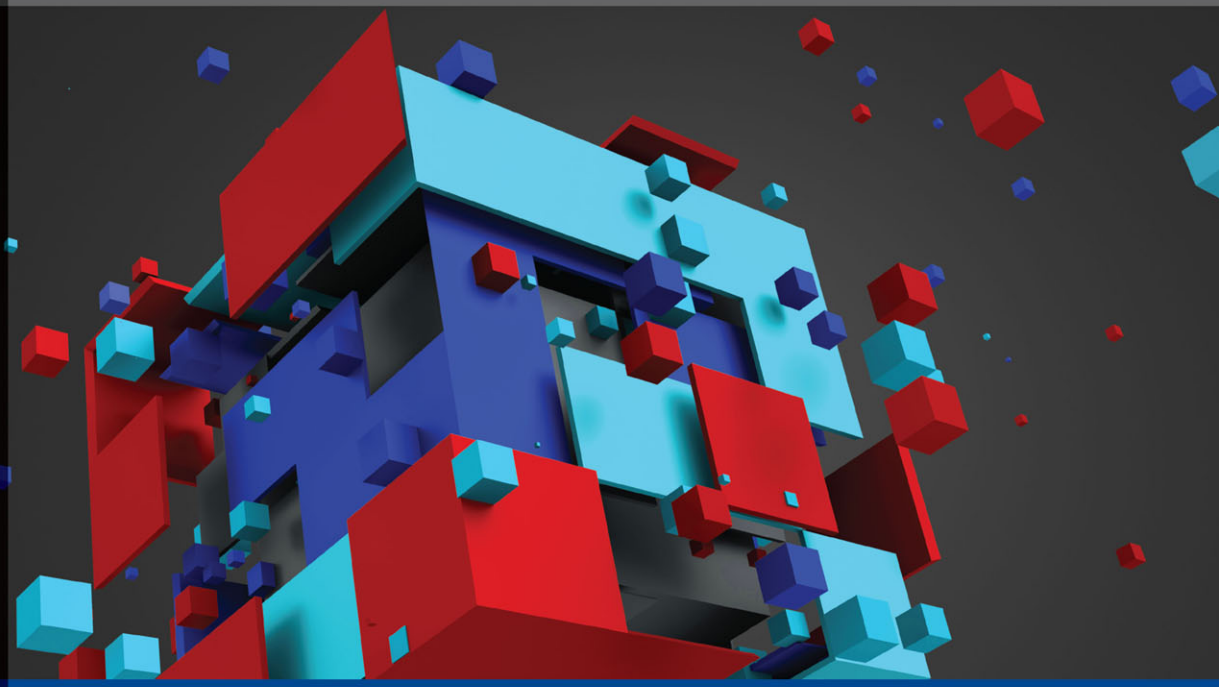
"This book is an essential guide for every architect and developer deploying secure, business-critical solutions on Azure."

—*from the foreword by* **Scott Guthrie,**
Executive Vice President, Cloud + AI Group, Microsoft

Microsoft

Designing and Developing Secure Azure Solutions



Best practices

Michael Howard • Simone Curzi • Heinrich Gantenbein

Microsoft

Designing and Developing Secure Azure Solutions

Michael Howard
Heinrich Gantenbein
Simone Curzi

Microsoft Entra

Around the time of this writing, Microsoft introduced Microsoft Entra, which is a family of identity and access solutions. These include Azure AD, Microsoft Entra Permissions Management, and Microsoft Entra Verified ID. You can learn more about Microsoft Entra at https://docs.microsoft.com/en-us/entra/.

Microsoft Authentication Library

The Microsoft Authentication Library (MSAL) enables developers to obtain tokens from the Microsoft identity platform to authenticate users and access secure web APIs. MSAL is used by Microsoft products such as Office 365. This might sound a little silly, but if you simply use MSAL, you are 90 percent on your way to building a client solution that takes advantage of modern identity, which is why we will use it in our sample application.

Tip The following website explains all the current authentication libraries and SDKs available for various Microsoft technologies, with a strong emphasis on MSAL: https://azsec.tech/libs.

With MSAL, all the hard work, bug fixes, and subtle protocol issues are handled for you. When the OAuth2 protocol is updated and new features are made available, MSAL is updated too, by people who understand OAuth2 well. For example, near the time of this writing, it was announced that OAuth2 will soon support proof of possession (PoP) to help protect against token replay. When this happens, MSAL will be updated accordingly.

MSAL can target many common programming languages and environments, including the following:

- Android
- Angular
- Go
- iOS and macOS
- Java
- JavaScript
- .NET
- Node
- Python
- React

MSAL targets clients or applications behaving as clients. For example, a web browser running Angular acts as a client when it accesses a server. Similarly, in the case of an Azure Function accessing Microsoft Graph, the function acts as a client.

Note In OAuth2, a user is different from a client. A user is a human being, and a client is the software used by a human being or some other principal, such as a process.

Active Directory Authentication Library

An older library, Active Directory Authentication Library (ADAL), integrates with Azure AD only. However, ADAL has been deprecated in favor of MSAL. MSAL integrates with Azure AD, Microsoft personal accounts, and Azure AD Business to Customer (B2C), which can authenticate users using other identity providers such as Facebook, Google, and Twitter. So, you should not use ADAL, and if you do use ADAL, you must migrate to MSAL as soon as possible. You can read about the differences between ADAL and MSAL at https://azsec.tech/htc.

MSAL supports continuous access evaluation (CAE). What this means is if the user's situation changes—for example, malware is detected on their laptop—access to a protected resource can then be denied quickly. Here is another example: Suppose an employee logs in to their work laptop at home and accesses a protected resource. Then, the employee puts their laptop to sleep, packs it up, and takes it to a coffee shop, where they continue working while sipping on a latte. Because their situation changed, they might be prompted to log on again and maybe even satisfy a second authentication factor such as a phone app challenge.

MSAL also supports multiple application types—for example, web apps, mobile apps, native apps, daemons, and services. This is important because different application types have different security implications and use different OAuth2 flows, which we will explain in the "Flows" section.

Tip If you cannot use MSAL—for example, you use a programming language not supported by MSAL—you should use an actively maintained library instead. For options, see https://openid.net/developers/certified/.

Now, let's look at some sample JavaScript code. The following code creates an MSAL object. As you can see here, MSAL performs an immense amount of heavy lifting so you do not have to, in a small amount of code.

```
const msalConfig = {
       auth: {
          clientId: "d1361825-38d2-413a-9d2c-00e79a3a9221",
          authority: "https://login.microsoftonline.com/common",
          redirectUri: "https://localhost:4242",
}, <snip>
const myMSALObj = new msal.PublicClientApplication(msalConfig);
```

The `msalConfig` section shown here is important. (In this example, it is abbreviated.) It includes the following parameters:

- **clientId** This is the application (client) ID value in the Azure Portal for this application. You can find it in Azure AD by selecting App Registrations and then the name of the application.

Important `clientId` is not a secret. We cannot stress this enough: the client ID does not need to be protected. This is different from a confidential client's credential, which *is* a secret and must be protected. We discuss confidential clients in the "Flows" section later in the chapter.

- **authority** This is the cloud instance. The associated URL is a discovery URL used by Azure AD to locate information about your application and will change depending on the types of users you are signing in. Here, the URL contains `/common`, which indicates a work, school, or personal Microsoft account. Other options are as follows:
  - **/organizations** This is for work and school accounts.
  - **/consumers** This is for personal Microsoft accounts (Xbox, Outlook.com, and so on).
  - **/<tenant>** This is the GUID for the tenant. It logs in users of a specific organization only.
- **redirectUri** This is the URI to return after the access token is issued. This example uses `localhost` as the redirect URI, which indicates that this code is not production ready. When the code is ready for production, this parameter is replaced with a real URI.

Tip For a full explanation of `msalConfig`, see https://azsec.tech/cfg. (Note that much of this data is found when you create an application registration.) Also, a complete explanation of all the configuration options is available here: https://azsec.tech/kj2.

Real-world experience: Debugging OpenID Connect and OAuth2

We all know that sometimes things go wrong in code. Thankfully, MSAL—especially MSAL JS—has excellent debug logging. If things are not working as expected in your client MSAL code, open the Developer Tools page (press Ctrl+Shift+I in Edge) and click the Console tab. You will find any errors or warnings from MSAL in the console.

The preceding code also instantiates a `PublicClientApplication` class. This is because the code is JavaScript running in a browser. As such, it cannot securely store secrets, so it cannot be a confidential client. If you look at the MSAL JavaScript source code, you will notice there is no confidential client support, because supporting a confidential client is not possible.

After you create the MSAL object, you implement the following code to sign in the user:

```
const loginRequest = {
    scopes: ["User.Read"]
};
myMSALObj.loginPopup(loginRequest)
  .then(handleResponse)
  .catch(error => {
    console.error(error);

  });
```

Finally, you use code like this to obtain access tokens:

```
request.account = myMSALObj.getAccountByUsername(username);

return myMSALObj.acquireTokenSilent(request)
    .catch(error => {
        console.warn("silent token acquisition fails. acquiring token using popup");
        if (error instanceof msal.InteractionRequiredAuthError) {
            // fallback to interaction when silent call fails
            return myMSALObj.acquireTokenPopup(request)
                .then(tokenResponse => {
                    console.log(tokenResponse);
                    return tokenResponse;
                }).catch(error => {
                    console.error(error);
                });
        } else {
            console.warn(error);
        }
});
```

Note For .NET, MSAL is implemented in `Microsoft.Identity.Client`.

Tip A more complete version of this sample is available at https://azsec.tech/jsx, and the source code and documentation for the JavaScript MSAL library can be found at https://azsec.tech/jsm. If you are new to MSAL, consider reading the article "Overview of the Microsoft Authentication Library (MSAL)" at https://azsec.tech/msal, as this provides a broad overview of its use.

Where is your application hosted?

This might seem obvious, but your application code—regardless of whether it is mobile, native, a web API, or something else—does not need to be hosted in Azure. For example, suppose you write an application in C# that runs on your users' desktops and accesses Cosmos DB in Azure. Under the hood, this application would perform all the appropriate authentication and authorization using MSAL and Azure AD.

OAuth2 roles

The OAuth2 "ceremony" involves various parties, or roles:

- **Resource owner** This is often someone who grants a client application the ability to access data that belongs to them. For example, suppose Toby grants an email client read-only access to his email. In this case, the resource is email, and the resource owner is Toby.
- **Resource server** This is a protected resource. Continuing with the example of Toby's email, an email server is the resource server, and it is accessible using REST APIs.
- **Client** This is an application attempting to access resources on a resource server. The application could be a web browser used by a user, a VM using a Managed Identity, and so on.
- **Authorization server** This authenticates the client and issues tokens for access to resource servers. In our Azure example, Azure AD is the authorization server. Authorization servers are often called *identity providers*.

Flows

OAuth2 defines various interactions among the roles defined in the preceding section, and these interactions are called *flows*. An example of a flow is the interaction between a client application and an authorization server. When using Azure AD, the result of these flows is a JWT that can be used to grant some level of access to a resource. Note that we say *a resource* and not *resources*. We will explain the distinction in the upcoming section on tokens.

Note You will often see references to *grant flows*. The terms *flow* and *grant flow* are interchangeable.

There are various types of OAuth2 flows. Each flow has the same set of goals: to obtain a token that represents a set of permissions to act on behalf of a user, and to perform one or more tasks on the user's behalf. Obtaining a token involves two steps:

1. Getting the token from an identity provider such as Azure AD or Facebook
2. Using the token to perform a task on behalf of the user—for example, to read a user's email address or, in the case of Facebook, to post an image to the user's timeline

The various flows supported by OAuth2 vary only in the first step (getting the token). This is because different application types and deployment scenarios dictate which flows will work and which will not.

Three key OAuth2 flows are as follows:

- **Authorization code flow** This is presently the most secure, and therefore the preferred, flow because it does not expose credentials to any other processes or services (except the OAuth2 identity provider—for example, Azure AD—which already has your password identifier and is trusted). You can learn more about it here: https://azsec.tech/acf.

Proof of Key for Code Exchange

Authorization code flows can be made even more secure through the use of an extension called Proof of Key for Code Exchange (PKCE), pronounced *pixie*. MSAL automatically uses PKCE whenever it can.

PKCE helps mitigate cross-site request forgery (CSRF) and other injection attacks, especially against public clients (explained in the upcoming section about client types). To do so, it uses an ephemeral client-generated secret called a *code verifier*. PKCE is defined in RFC 7363, located here: https://azsec.tech/pkce.

- **Client credentials flow** This flow is often used when two services, such as two daemons, talk to each other. It permits the client web service to use its own credentials to authenticate when calling another web service. These clients are called *confidential clients* because they can store and protect credentials. For more information about this flow, see https://azsec.tech/ccf.

Warning Never publish client credentials flow credentials in your source code, embed them in web pages, or hard-code them in a client application that runs on user's devices or machines.

Tip You can also use an X.509 certificate and private key instead of a shared secret as a client application's credential. You should use X.509 certificates and private keys in production rather than shared secrets.

- **On-behalf-of (OBO) flow** If you are familiar with Windows impersonation, the OBO flow is a close, but more secure, cousin. (We say more secure because in Windows, impersonation does not constrain access. So, for example, if Cheryl connects to a Windows service that impersonates her, the service can do anything that Cheryl can do on that machine.) OBO is used when an application calls a service or web API, which in turn needs to call another service or web API. The idea is to propagate the delegated user identity and permissions through the request chain. At the time of this writing, the OBO flow works only for user accounts, not for service principals. For more information about the OBO flow, see https://azsec.tech/obo.

Note OBO cannot be used with Azure AD B2C. For more information, see https://azsec.tech/s8z.