Implementing Passwordless Auth is now stupidly easy — Part 2

Chris Keogh · Follow
Published in CodeX
7 min read · Jul 15, 2021

Listen Share More

In my previous post, I outlined why we should consider moving away from using passwords to authenticate users instead of using Passwordless Authentication. In part two we'll explore how one might go about implementing a Passwordless Authentication flow using Dotnet core 3.1, IdentityServer4 and the FIDO2.NET library.

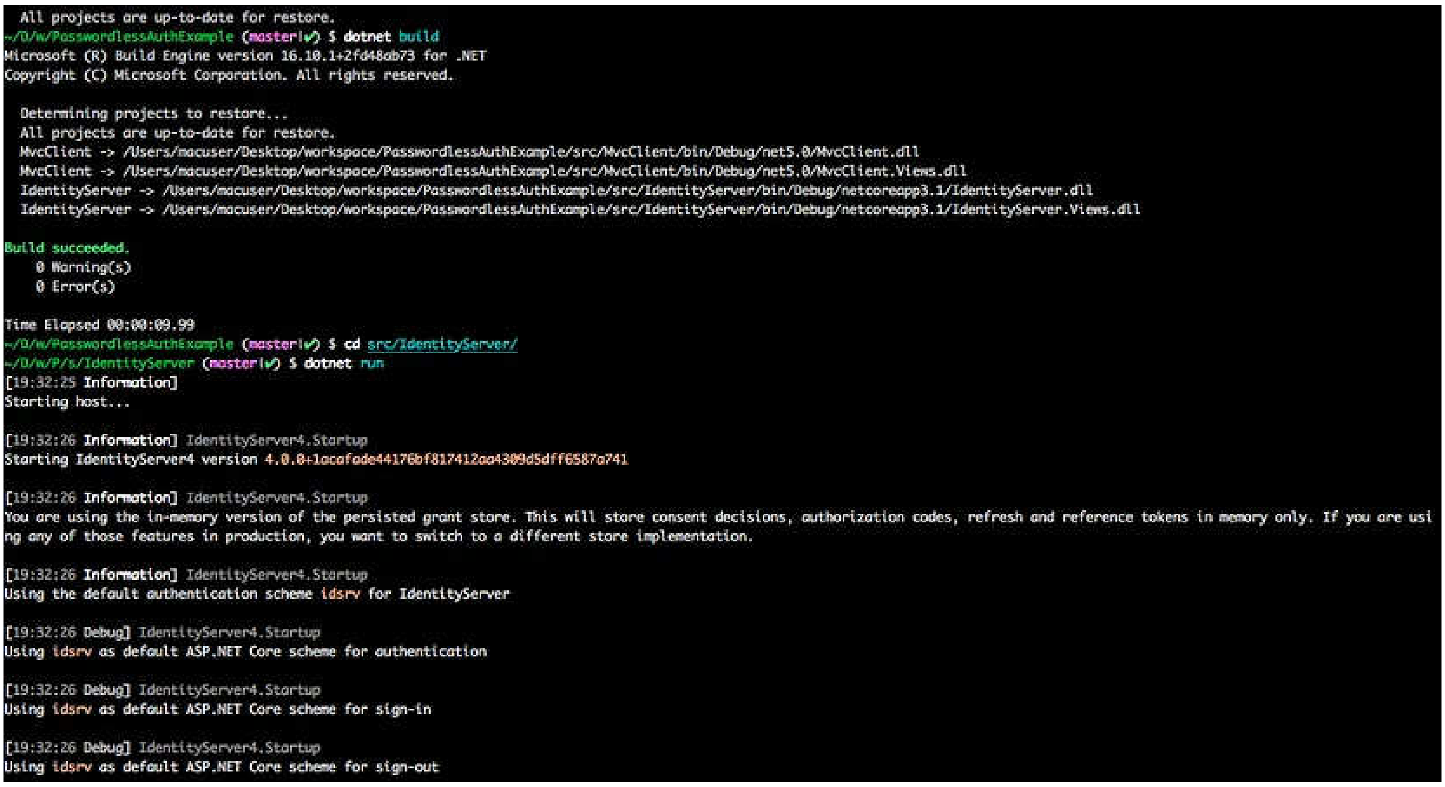
All of these tools are freely available as open-sourced technologies. To run the example used in this post, you'll either need some form of authenticator set-up, such as Windows Hello, Apple ID, or a Yubikey, or you can use the WebAuthn virtual environment available in Chrome's dev tools.

I've chosen IdentityServer4 because it is a mature open-source implementation of the Open ID Connect specification and is very easy to get up and running. The working example code used in this post, which is available on GitHub, is based on the template generation tooling that IdentityServer4 provides. I'm not going to go through all the steps to generate IdentityServer and the sample MvcClient referred to in this post because they are detailed well enough by IdentityServer's own quickstart documentation. However, if you're interested in starting from scratch, you only need to follow the first and third tutorials.

Rather than step through every single line of code, I'm going to frequently refer to the example GitHub repository while focusing on outlining the relevant request flows and highlighting any interesting code.

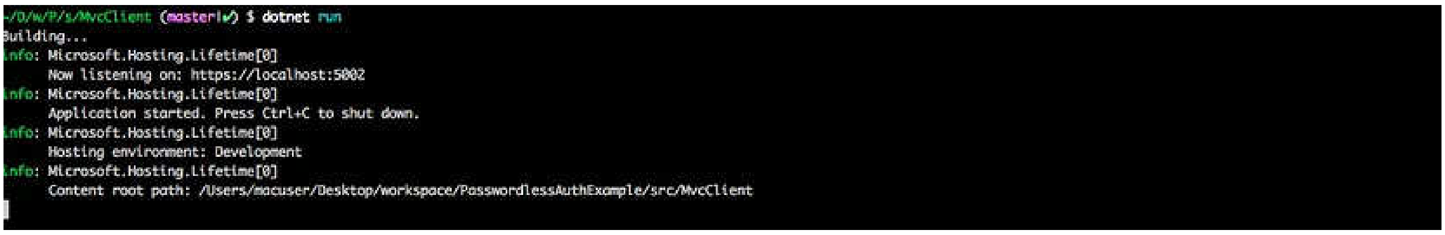
Getting Started

Firstly, make sure to clone the PasswordlessAuth GitHub repository which contains all of the code you'll need to get this up and running. Simply run dotnet restore and dotnet build before running the IdentityServer project:



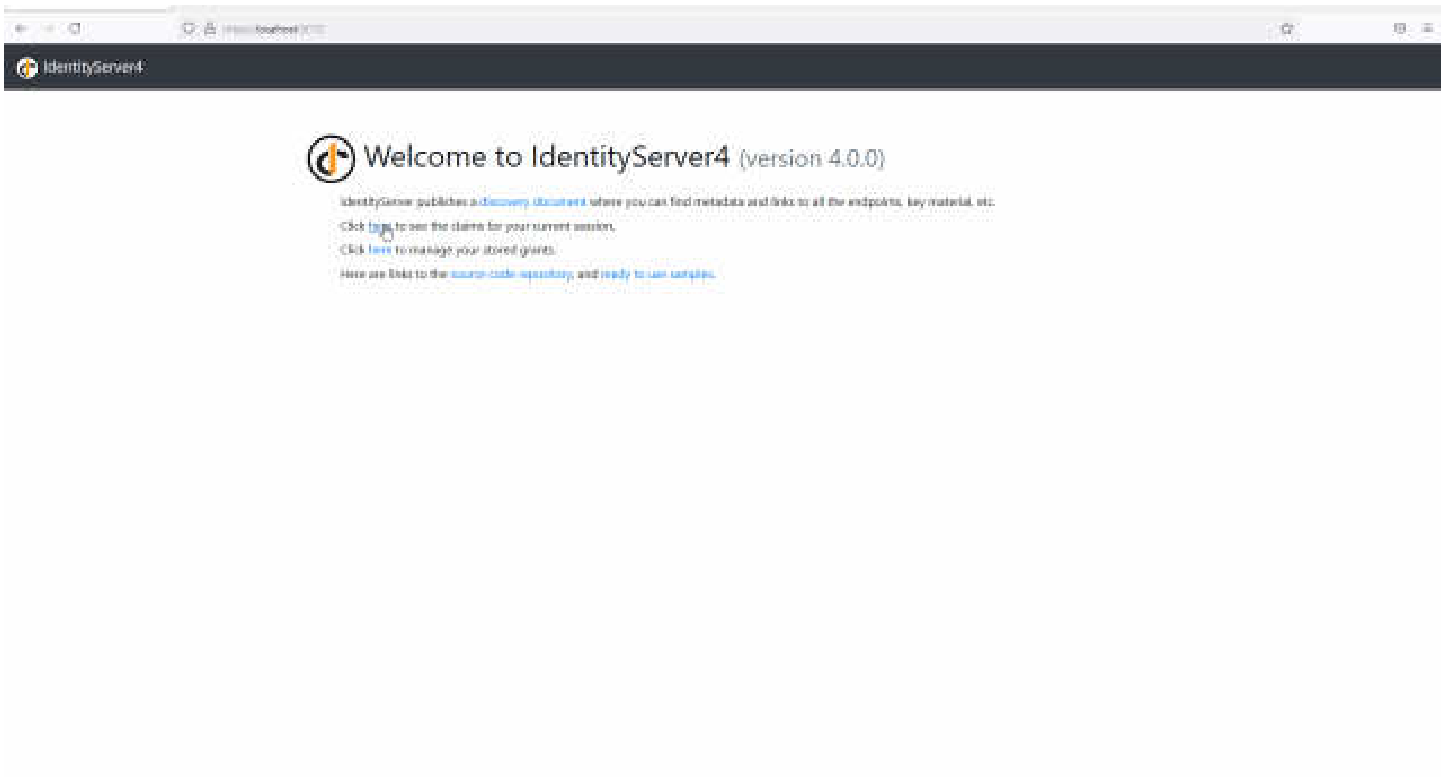
```
  All projects are up-to-date for restore.
~/D/w/PasswordlessAuthExample (master|✔) $ dotnet build
Microsoft (R) Build Engine version 16.10.1+2fd48ab73 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  MvcClient -> /Users/macuser/Desktop/workspace/PasswordlessAuthExample/src/MvcClient/bin/Debug/net5.0/MvcClient.dll
  MvcClient -> /Users/macuser/Desktop/workspace/PasswordlessAuthExample/src/MvcClient/bin/Debug/net5.0/MvcClient.Views.dll
  IdentityServer -> /Users/macuser/Desktop/workspace/PasswordlessAuthExample/src/IdentityServer/bin/Debug/netcoreapp3.1/IdentityServer.dll
  IdentityServer -> /Users/macuser/Desktop/workspace/PasswordlessAuthExample/src/IdentityServer/bin/Debug/netcoreapp3.1/IdentityServer.Views.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:09.99
~/D/w/PasswordlessAuthExample (master|✔) $ cd src/IdentityServer/
~/D/w/P/s/IdentityServer (master|✔) $ dotnet run
[19:32:25 Information]
Starting host...

[19:32:26 Information] IdentityServer4.Startup
Starting IdentityServer4 version 4.0.0+1acafade44176bf817412aa4309d5dff6587a741

[19:32:26 Information] IdentityServer4.Startup
You are using the in-memory version of the persisted grant store. This will store consent decisions, authorization codes, refresh and reference tokens in memory only. If you are usi
ng any of those features in production, you want to switch to a different store implementation.

[19:32:26 Information] IdentityServer4.Startup
Using the default authentication scheme idsrv for IdentityServer

[19:32:26 Debug] IdentityServer4.Startup
Using idsrv as default ASP.NET Core scheme for authentication

[19:32:26 Debug] IdentityServer4.Startup
Using idsrv as default ASP.NET Core scheme for sign-in

[19:32:26 Debug] IdentityServer4.Startup
Using idsrv as default ASP.NET Core scheme for sign-out
```

Make sure to also run the MvcClient sample application:



```
~/D/w/P/s/MvcClient (master|✔) $ dotnet run
Building...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5002
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /Users/macuser/Desktop/workspace/PasswordlessAuthExample/src/MvcClient
```

In our example, we have IdentityServer running on HTTPS port 5010 and the MvcClient sample running on HTTPS port 5002 (purely arbitrary selection of ports on my part). Once you have this up and running you should be able to log straight into IdentityServer with the username and password 'alice' and should now be able to sign into the MvcClient sample:



Welcome to IdentityServer4 (version 4.0.0)

IdentityServer publishes a discovery document where you can find metadata and links to all the endpoints, key material, etc.

Click here to see the claims for your current session.

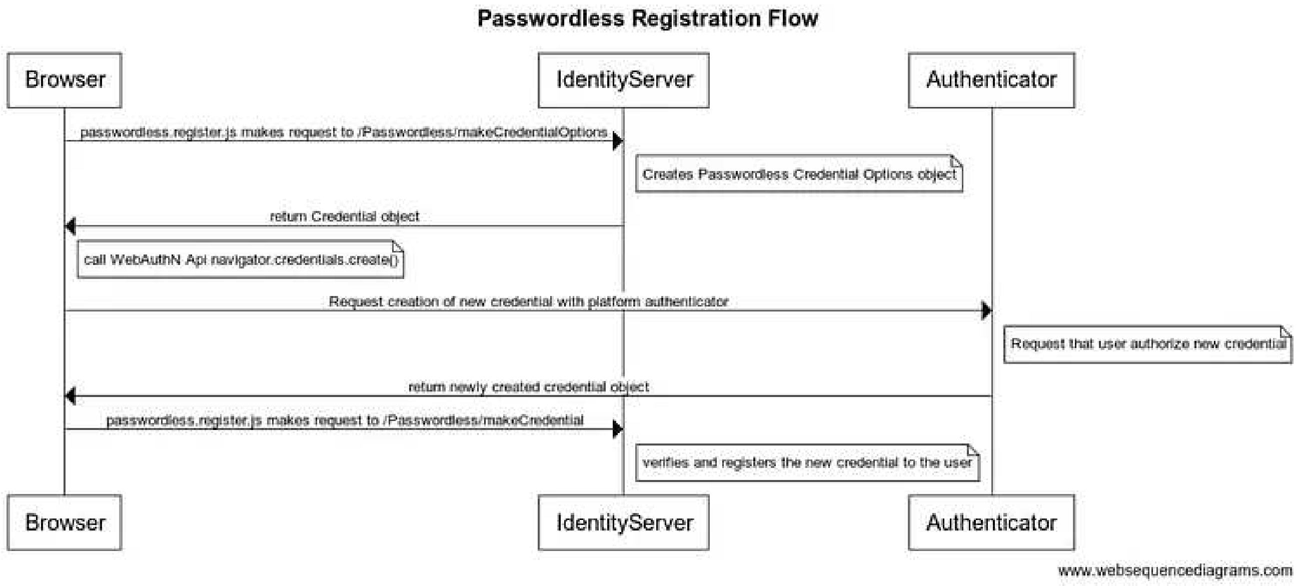
Click here to manage your stored grants.

Here are links to the source code repository, and ready to use samples.

An overview of the Passwordless flows

Now let's take a look at the Passwordless Authentication flows and how they work before examining how they've been implemented in this example. I've essentially taken the FIDO2.NET demo and implemented it into this project with some minor changes.

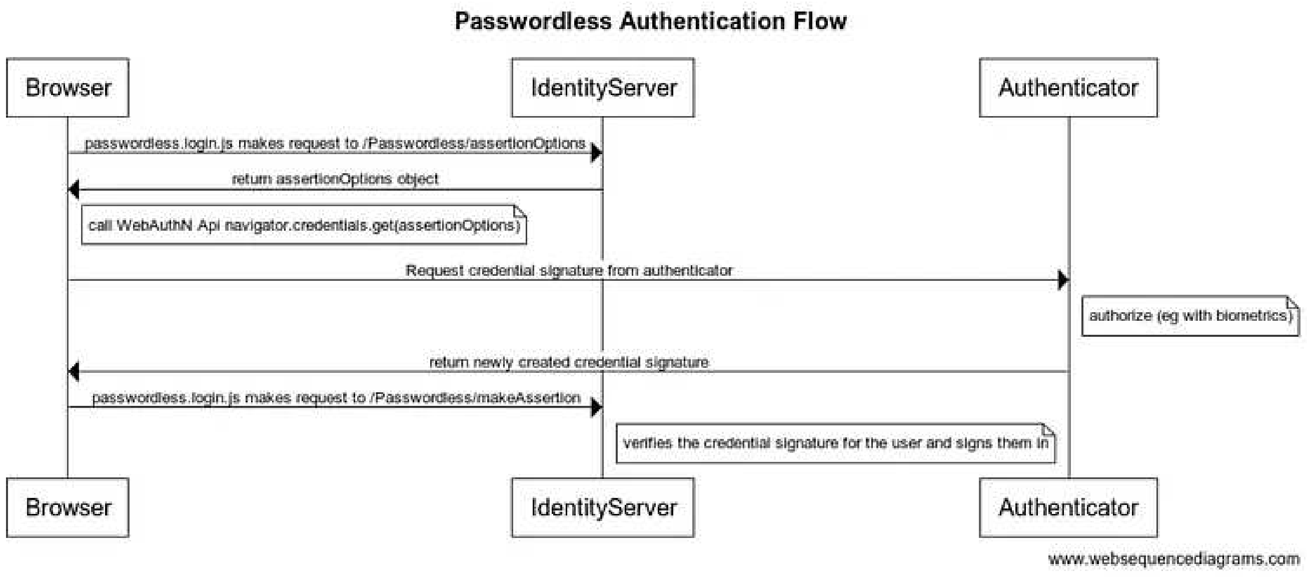
In my previous post, I briefly outlined the registration and authentication processes that together makeup Passwordless Authentication as described in the WebAuthn spec. Let's take a look at the implementation of these flows in our example, starting with the registration flow, which allows an existing IdentityServer user (in this case we'll stick with "alice") to set up Passwordless Authentication.


Passwordless Registration Flow
Browser
IdentityServer
Authenticator
passwordless.register.js makes request to /Passwordless/makeCredentialOptions
Creates Passwordless Credential Options object
return Credential object
call WebAuthN Api navigator.credentials.create()
Request creation of new credential with platform authenticator
Request that user authorize new credential
return newly created credential object
passwordless.register.js makes request to /Passwordless/makeCredential
verifies and registers the new credential to the user
Browser
IdentityServer
Authenticator
www.websequencediagrams.com

A high-level overview of our implementation of the passwordless registration flow.

We can see that IdentityServer hands off a credential options object which is used by the browser to create a new credential for an existing IdentityServer user. This credential is specific to IdentityServer's domain (localhost:5010) and cannot be used for any other website. The credential is generated by the Authenticator and authorised by the user through an authorisation gesture (eg. Windows Hello with biometric fingerprint scanning enabled for authorisation). The credential is made up of a public/private key pair. The private key is stored locally by the Authenticator and the public key is posted back to IdentityServer to be stored server-side. For a more in-depth technical explanation please see my previous post.

Following on from that, let's take a look at the actual authentication flow, available to our user once the registration flow has been completed.


Passwordless Authentication Flow
Browser
IdentityServer
Authenticator
passwordless.login.js makes request to /Passwordless/assertionOptions
return assertionOptions object
call WebAuthN Api navigator.credentials.get(assertionOptions)
Request credential signature from authenticator
authorize (eg with biometrics)
return newly created credential signature
passwordless.login.js makes request to /Passwordless/makeAssertion
verifies the credential signature for the user and signs them in
Browser
IdentityServer
Authenticator
www.websequencediagrams.com

A high-level overview of our implementation of the Passwordless Authentication flow.

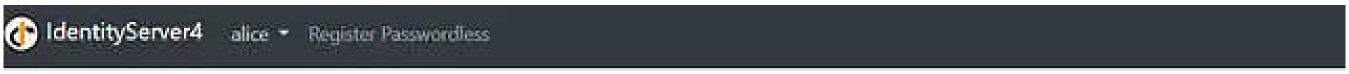
In a lot of ways, the authentication flow is similar to the registration flow. Our IdentityServer implementation hands off a set of assertionOptions to the browser, which then asks the Authenticator for the relevant credential. The user is asked for consent to allow the authenticator to pass the credential back to the browser and this credential is sent back to IdentityServer to verify that it is valid, before authenticating the user. I've used the term credential signature here where I should have used the term assertion signature. What is actually posted back to IdentityServer is a signature created by the authenticator using a private/public key algorithm. The private key sits within the user's authenticator and is used to generate a signature that IdentityServer — the relying party — validates with the associated public key it has stored server-side.

Getting our passwordless implementation up and running

Now we have a better understanding of what is actually involved during each of these flows, let's take a closer look at our example implementation.

I've added a new PasswordlessController to the controllers folder. I have essentially copy and pasted the demo controller from the FIDO2.NET library and made some minor changes, which I will come back into in a moment.

I've also added two new views to the IdentityServer project. One for passwordless login, and one for registration. I've also added a third button to the login view specifically for passwordless login. For registration, we'll add it to the shared navigation header and make it visible only once a user has signed in.



We'll also want to add two new endpoints to the Account controller to load our Passwordless views.

For any of this to work, we will need to add some Javascript that glues the flows together between the Authenticator, the Browser, and the Relying Party (IdentityServer). One file for passwordless login and one for registration. I've pretty much copied and pasted the demo javascript from the FIDO2.NET demo here also, with some very minor modifications.

For our passwordless flows to work we need some setup code for the FIDO2.NET library in the Startup class. FIDO2.NET requires the NewtonSoftJson package to work correctly, so once that's installed we need to add it:

```
services.AddControllersWithViews().AddNewtonsoftJson();
```

The remainder of the setup code:

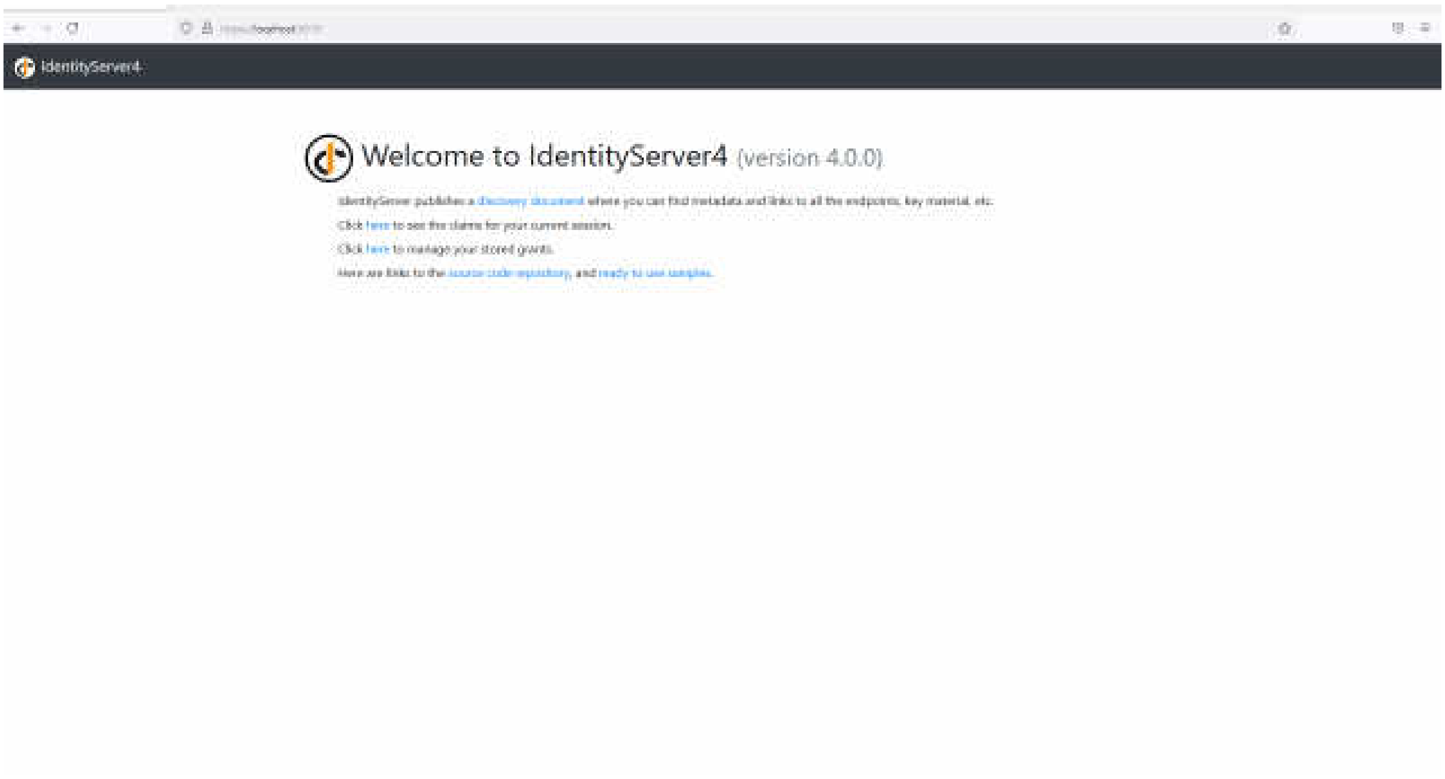
```
services.AddFido2(options =>
    {
        options.ServerDomain = Configuration["fido2:serverDomain"];
        options.ServerName = "IdentityServerPasswordlessTest";
        options.Origin = Configuration["fido2:origin"];
        options.TimestampDriftTolerance =
Configuration.GetValue<int>("fido2:timestampDriftTolerance");
        options.MDSAccessKey = Configuration["fido2:MDSAccessKey"];
        options.MDSCacheDirPath =
Configuration["fido2:MDSCacheDirPath"];
    })
    .AddCachedMetadataService(config =>
    {
        if
(!string.IsNullOrWhiteSpace(Configuration["fido2:MDSAccessKey"]))
        {

config.AddFidoMetadataRepository(Configuration["fido2:MDSAccessKey"]
  );
          }
          config.AddStaticMetadataRepository();
      });
```

We also need some configuration settings in appsettings.json:

```
  {
      "fido2": {
          "serverDomain": "localhost",
          "origin": "https://localhost:5010",
          "timestampDriftTolerance": 300000,
          "MDSAccessKey": null
      }
  }
```

In the interest of time, I have also removed the **required** attribute from the password property in the LoginInputModel class because we're going to reuse this class for passwordless login. It's probably better practice to create a separate PasswordlessInputModel, but we'll just modify what already exists.

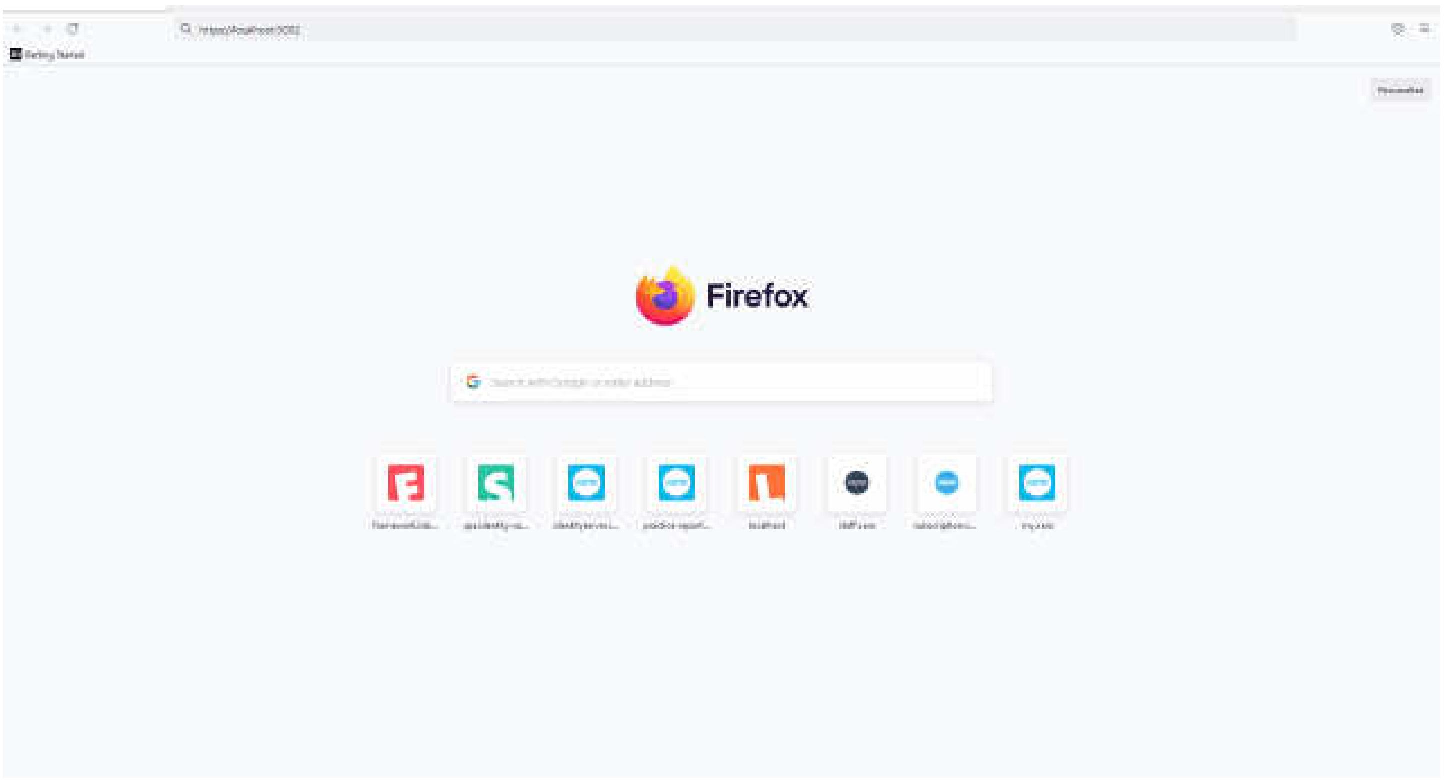
That should be everything required to get up and running to register:



IdentityServer publishes a discovery document where you can find metadata and links to all the endpoints, key material, etc.
Click here to see the claims for your current session.
Click here to manage your stored grants.
Here are links to the source code repository, and ready to use samples.

Signing in and then registering for the Passwordless Authentication flow.

… and subsequently, you should be able to sign in using the Passwordless Authentication flow:



Signing into our MvcSample app using the passwordless flow.

Integrating FIDO2.NET with IdentityServer4

There are two important pieces of code to take note of in the PasswordlessController. In the MakeCredentialOptions method we check to see whether the user exists in our IdentityServer users store (we can't authenticate a user that doesn't exist):

```
// user must already exist in Identity
var identityUser = _users.FindByUsername(username);
if (identityUser == null) {
    throw new Exception("User not found");
}
```

We also check whether the user is actually authenticated:

```
if (!HttpContext.User.IsAuthenticated())
{
    throw new Exception("User is not authenticated");
};
```

In this example, we actually have a separate in-memory user store for our passwordless users, courtesy of the FIDO2.NET library, DevelopmentInMemoryStore. We could probably do something to integrate this into our existing IdentityServer user store but for the sake of this example, and to save time, we'll keep them separate and rely on the above checks to keep things in sync.

I have added a new method to this controller called SignInOidc which looks like this:

```
async Task SignInOidc(string username)
{
    var user = _users.FindByUsername(username);
    await _events.RaiseAsync(new
UserLoginSuccessEvent(user.Username, user.SubjectId,
user.Username));

    AuthenticationProperties props = new AuthenticationProperties();
    // issue authentication cookie with subject ID and username
    var isUser = new IdentityServerUser(user.SubjectId)
    {
        DisplayName = user.Username
    };

    await AuthenticationManagerExtensions.SignInAsync(HttpContext,
isuser, props);
}
```

This method is called by the MakeAssertion method, which validates the assertion signature sent back by the browser during the authentication flow. If this is successful, SignInOidc is called which signs the user in resulting in the authentication and session cookies being written by IdentityServer.

```
// 5. Make the assertion
var res = await _fido2.MakeAssertionAsync(clientResponse, options,
creds.PublicKey, storedCounter,
    callback);

// 6. Store the updated counter
```

```
    var username =
System.Text.Encoding.UTF8.GetString(creds.UserId);
    await SignInOidc(username);
}
```

Wrapping up

That’s all there is to it to get the basic passwordless registration and authentication flows up and running using IdentityServer4, Dotnet core 3.1 and the FIDO2.NET library. Personally, I‘d like to see this promising technology become more widely adopted across the internet, and I hope I have been able to provide an entry-level overview of how the technology works as well as how it can be implemented.

Questions, comments and feedback are welcome so please don’t hesitate to reach out.

Authentication Software Development Webauthn Identityserver4 Security

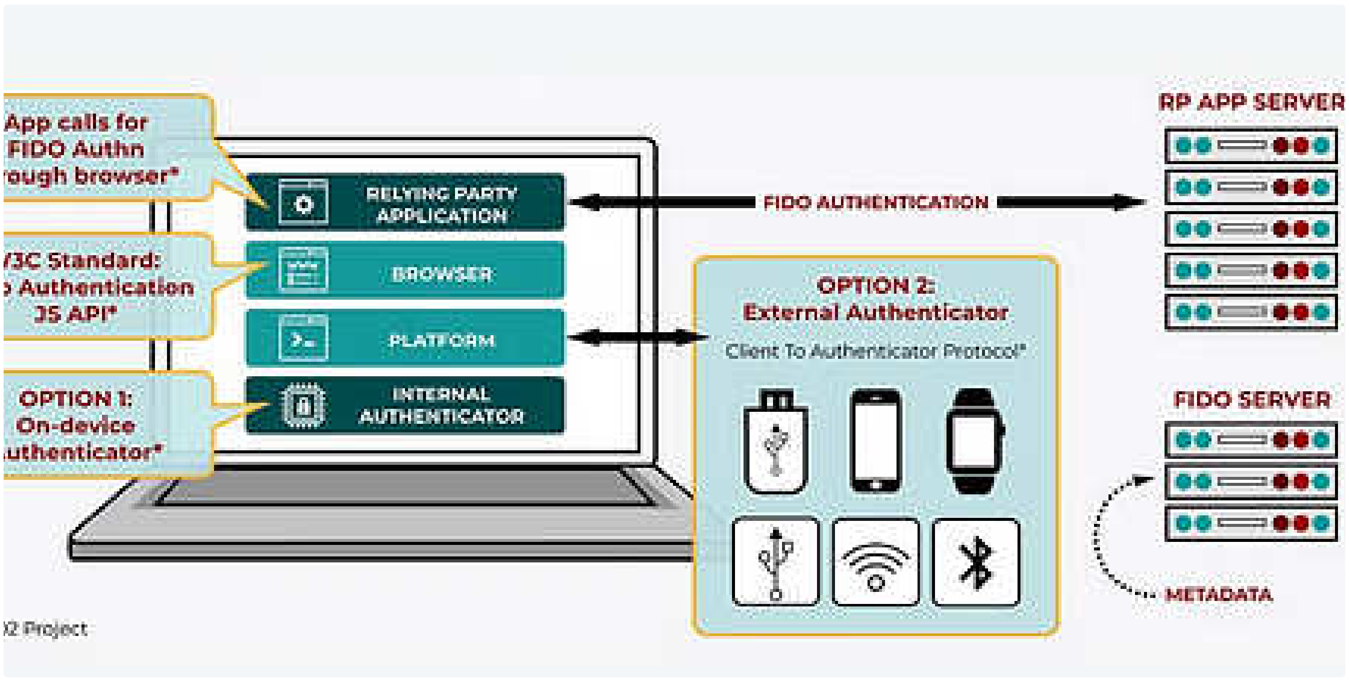
Follow

Written by Chris Keogh

3 Followers · Writer for CodeX

A software engineer and writer. Special interests include WebAuthN, OpenID Connect, Resilience Engineering and distributed systems.

More from Chris Keogh and CodeX



App calls for FIDO Authn rough browser*
/3C Standard: Authentication JS API*
OPTION 1: On-device uthenticator*
RELYING PARTY APPLICATION
BROWSER
PLATFORM
INTERNAL AUTHENTICATOR
FIDO AUTHENTICATION
OPTION 2: External Authenticator
Client To Authenticator Protocol*
RP APP SERVER
FIDO SERVER
METADATA
)2 Project

Chris Keogh

Implementing Passwordless Auth is now stupidly easy — Part 1

In this two part series I aim to provide an overview of what Passwordless Authentication is, why we should use it, and how we might...

7 min read · Jul 15, 2021

16

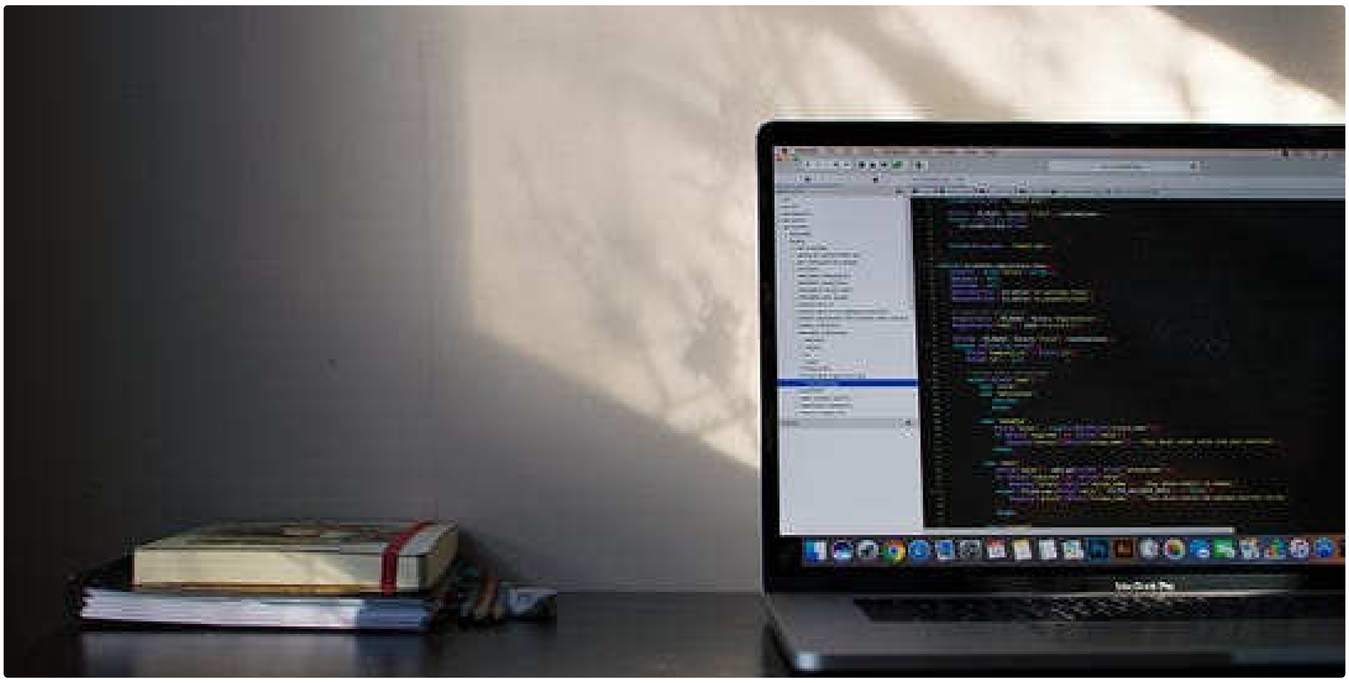
Christianlauer in CodeX

What is DataGPT and why should Data Engineers and Scientists care?

Are Data Experts loosing their Jobs due to AI?

✦ · 3 min read · Nov 10

869 12



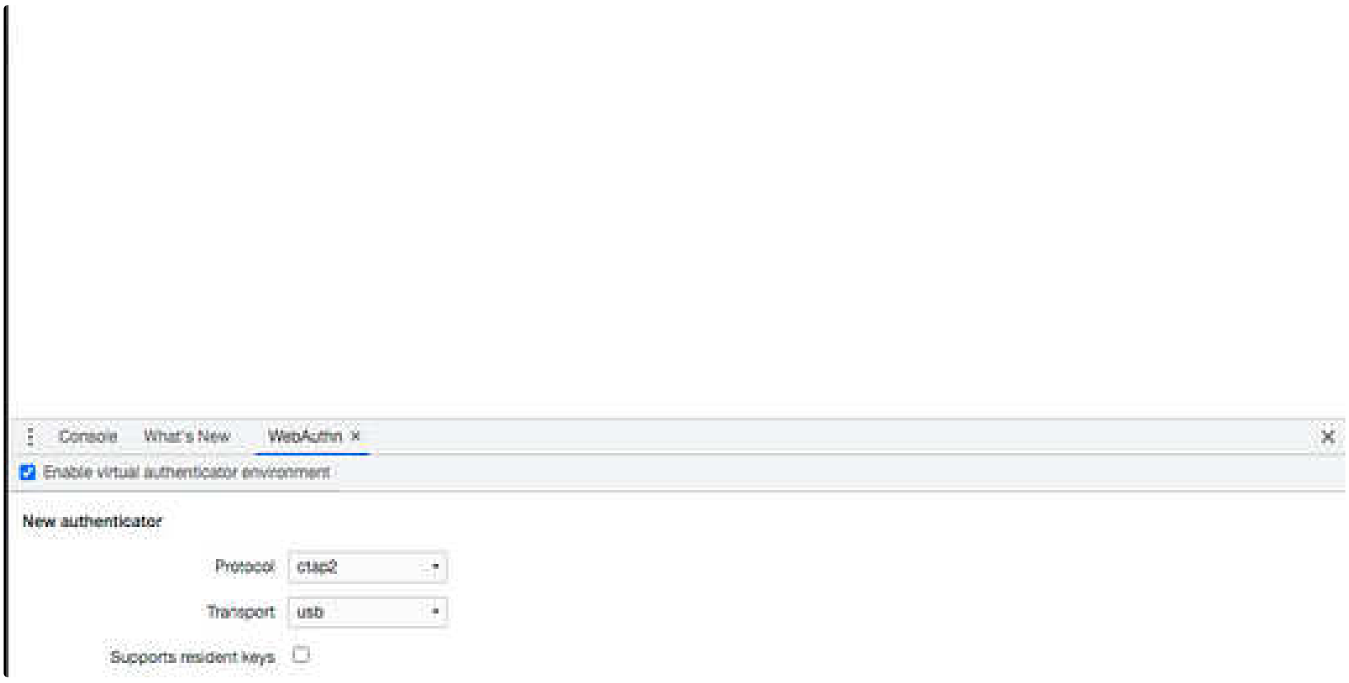
Anmol Tomar in CodeX

20 Pandas Codes To Elevate Your Data Analysis Skills

Data analysis is a cornerstone of decision-making in today's data-driven world, and Pandas is a powerful tool that empowers data analysts...

✦ · 5 min read · Nov 30

512 2



Chris Keogh

Implementing Passwordless Auth is now stupidly easy—part 3

In this post, I'm going to show how you can start experimenting with WebAuthn Passwordless Authentication without writing a single line of...

4 min read · Jul 21, 2021

56

See all from Chris Keogh

See all from CodeX

Recommended from Medium

Marc Kenneth Lomio & Melrose Mejidana

“Securing Angular Applications: A Guide to Implementing Refresh Tokens with IdentityServer4

Introduction.

3 min read · Aug 6

1

Dakshitha Ratnayake

Breaking Down Complex Authorization Beyond Login with ZITADEL

As we move towards a zero-trust mindset, the limitation of coarse-grained security measures like the traditional RBAC system become clear...

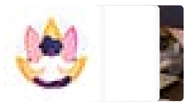
6 min read · Dec 1

2

Lists

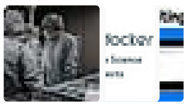
General Coding Knowledge

20 stories · 728 saves



Stories to Help You Grow as a Software Developer

19 stories · 668 saves

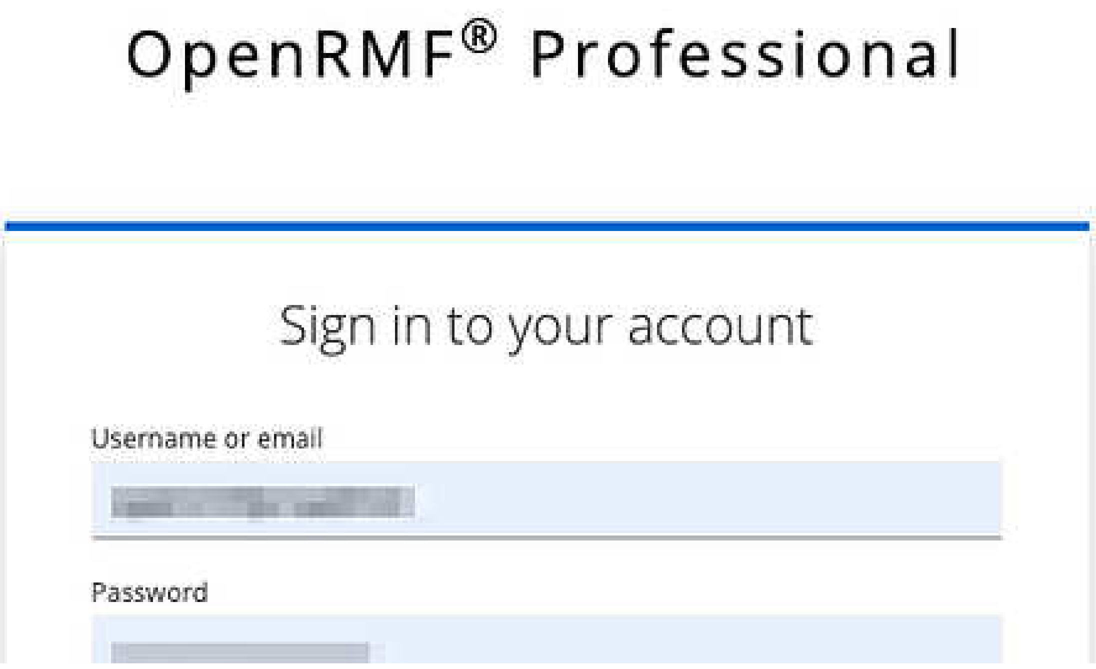


Coding & Development

11 stories · 349 saves

Apple's Vision Pro

7 stories · 38 saves



OpenRMF® Professional

Sign in to your account

Username or email

[redacted]

Password

[redacted]

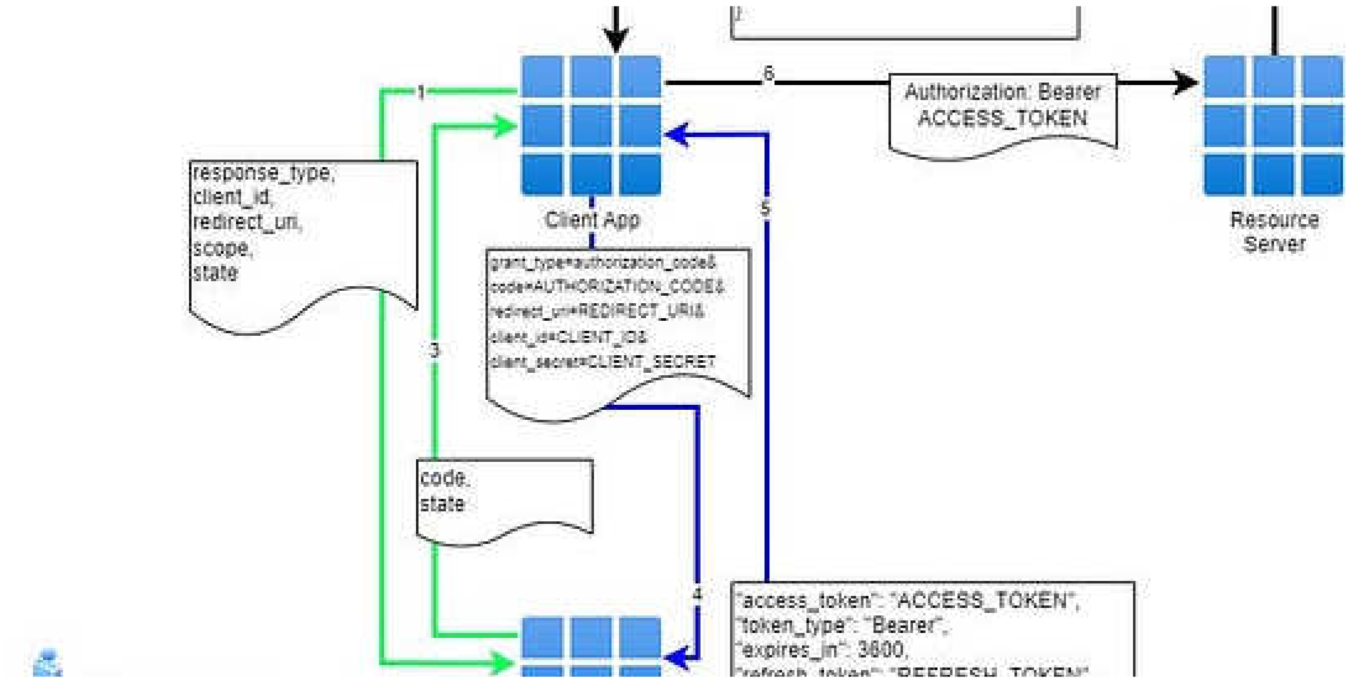
Dale Bingham

Integrate Keycloak and Google Workspace for Authentication

If you use Google Workspace for your corporate email, shared drive, calendar and such you can easily setup Keycloak to use THAT...

5 min read · Aug 22

1



Anirban Bhattacherji

Understanding OAuth 2.0: Architecture, Use Cases, Benefits, and Limitations (Part 3 — PKCE)

During our previous discussion in OAuth 2.0 Part 1, we focused on comprehending the distinctions between SAML and OAuth 2.0, as well as...

7 min read · Jul 11

9

Pranavk

Python Loading Dataframe in SQL Server using BCP

Introduction

3 min read · Oct 11

5

Maharoz Alam Mugdho

Step-by-Step Guide: Implementing Single Sign-On with Keycloak and Azure AD in Blazor WebAssembly...

Authentication is very important module for building a web application. Today I will show you how to implement Single Sign-On(SSO) with...

6 min read · Sep 2

5

See more recommendations