User Authentication and Identity with Angular, Asp.Net Core and IdentityServer

April 28, 2019

User Authentication and Identity with Angular, ASP.NET Core and IdentityServer
FullStack Mark
ASP.NET Core

Building a robust security model within our applications is a critical step toward shipping the type of high-quality, high-value software solutions we strive to deliver to our customers and organizations.

Despite its importance, the topic of application security is met, by many of us with less enthusiasm and priority than other more exciting or seemingly higher-value areas of our project. Every project is different, but a couple of general factors that I believe contribute to this perception are:

Modern application security is a complex problem

The natural inertia of innovation is causing applications to expand in size and complexity.

To maximize market reach, we often want to make our apps available to users coming from just about anywhere: corporate networks, social sites like facebook and twitter along with anonymous guests from the internet.

On top of that, many apps must also be accessible on a variety of interfaces, the usual suspects being web browsers and mobile devices but this list is expanding to include a growing number of IoT-enabled devices and things like Smart TVs.

Further complicating all of this is the rise of distributed and microservice architectures on the backend which results in a much more complex software environment with many moving pieces.

Designing, implementing and testing a security approach that encompasses all of these different pieces and their interactions is no small task!

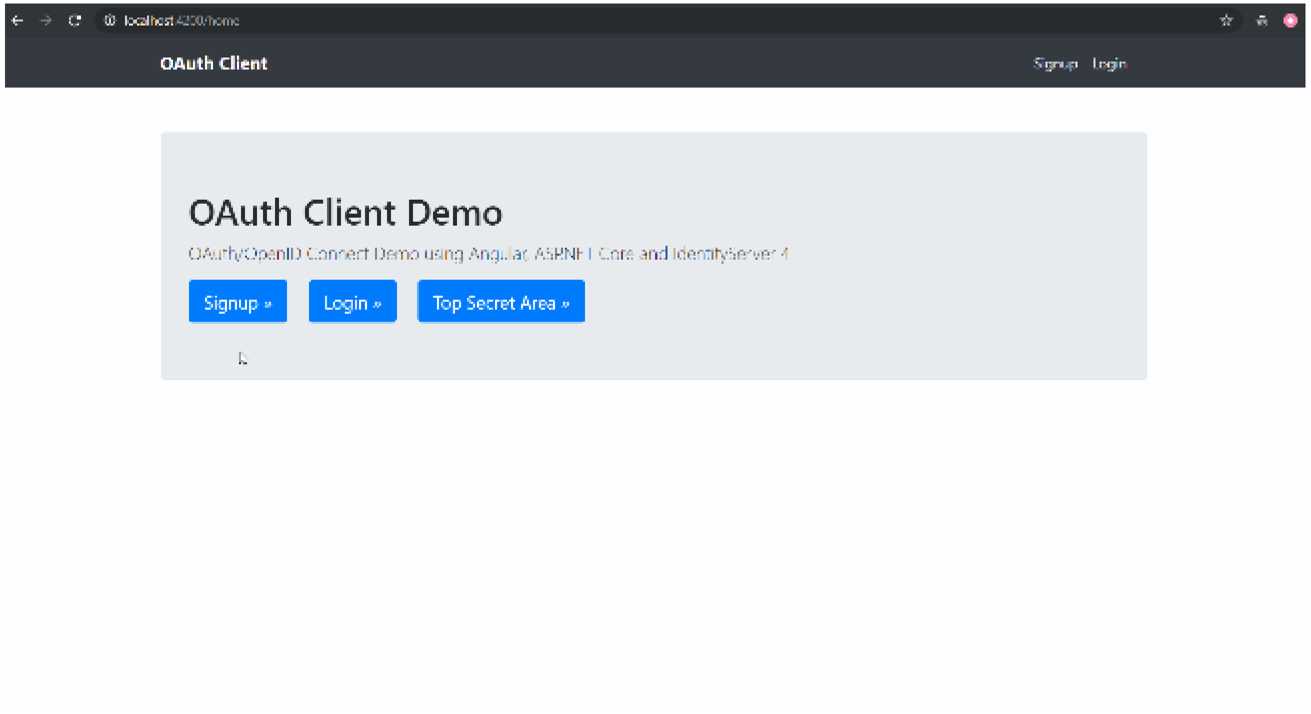
Security shouldn't be a specialty

Security protocols, principles and best practices are not as widely shared and commonly understood by most of us as they should be.

Security is a vast and deep topic. This may be an education or awareness thing but I believe that even if we don't identify as a security specialist within our primary developer roles there are some small steps we can begin to take to gain a better understanding of real-world security threats and the defensive coding techniques and practices we can use to protect against them. The OWASP Checklist is a great place to start with this.

Our Goal

In this guide, we'll see how to work through some of these problem areas while learning about identity and access control using OpenID Connect and OAuth. We'll implement the Implicit Grant OAuth flow and utilize these identity and authorization protocols by implementing IdentityServer4 as our OpenID Connect Provider and then using it to authenticate our Angular SPA client to authorize access to an independent ASP.NET Core Web API.



OIDC/OAuth authentication and authorization flow with Angular, ASP.NET Core and IdentityServer4.

Note: Security specs and standards evolve over time and OAuth is no different. There is currently a proposal in place to move away from the Implicit Flow we're utilizing in this guide in favor of the authorization code grant with Proof Key for Code Exchange (PKCE) to request access tokens from SPAs, as opposed to the original spec proposing the implicit grant for this scenario. This new recommendation is not the result of some gaping security hole in the implicit flow model but rather an improvement with the major difference being the exchange of the access token is removed from the browser URL and instead redeemed by the client application using an authorization code.

Here's a some great information from Auth0 on the topic and a quote to consider from the article.

> The new guidance does not stem from any newly discovered vulnerability: if you are satisfied with the threat model of your SPAs based on current guidance, you have no new reasons to update. That said, the new guidance does confer significant advantages: it is strongly recommended that you consider it, especially for brand new applications.

Before we dive into the code, let's review some of the basics behind the main concepts and terms we'll discuss as part of this solution.

Identity and Access Control

Identity as the name might suggest, means some set of attributes that a computer system can use to represent a person, organization, application or device. *Access control*, on the other hand, refers to a security technique used to regulate who or what can access and use resources (data and operations) in a computing environment.

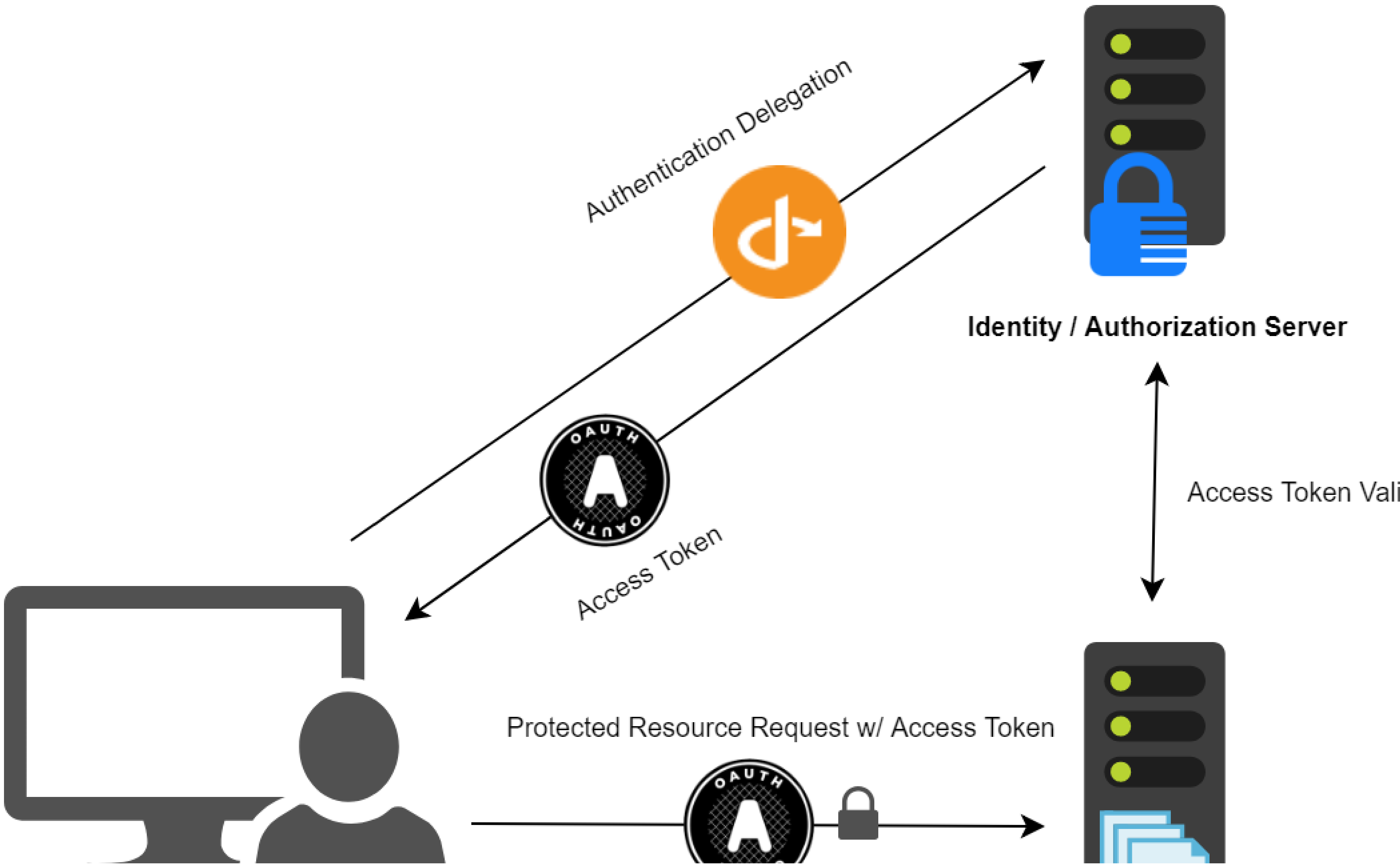
Authentication and Authorization

Authentication means the process used to determine whether a user is who they claim to be. Once authenticated, *authorization* determines which resources a given user should be able to access, and what they're allowed to do with those resources.

OpenID Connect and OAuth

OpenID Connect (OIDC) is a simple identity and authentication protocol layer built on top of the OAuth protocol that allows applications (typically referred to as clients) to verify the identity of end-users. *OAuth* is an open standard for authorization that provides secure, delegated access, meaning that an application (or client) can take actions or access resources on a resource server on behalf of a user, without the user ever sharing their credentials with the application - the key being *delegated access*.

Here's a diagram showing how these concepts and protocols fit together in a basic authentication and authorization scenario.


Authentication Delegation
Identity / Authorization Server
Access Token
Access Token Vali
Protected Resource Request w/ Access Token
OAUTH

OIDC/OAuth flow.

To summarize this workflow:

1. The user (resource owner) initiates an authentication request with the authorization server.
2. If the credentials are valid and everything checks out the authorization server obtains end-user consent and grants the client application an access token.
3. The access token is attached to subsequent requests made to the protected resource server.
4. The authorization server validates the access token; if successful the request for protected resources is granted, and a response sent back to the client application.

Ok, we've covered off some theory behind identity, access control, OpenID Connect, and OAuth. Now we'll look at implementing a similar workflow using Angular, ASP.NET Core and IdentityServer4.

Development Environment

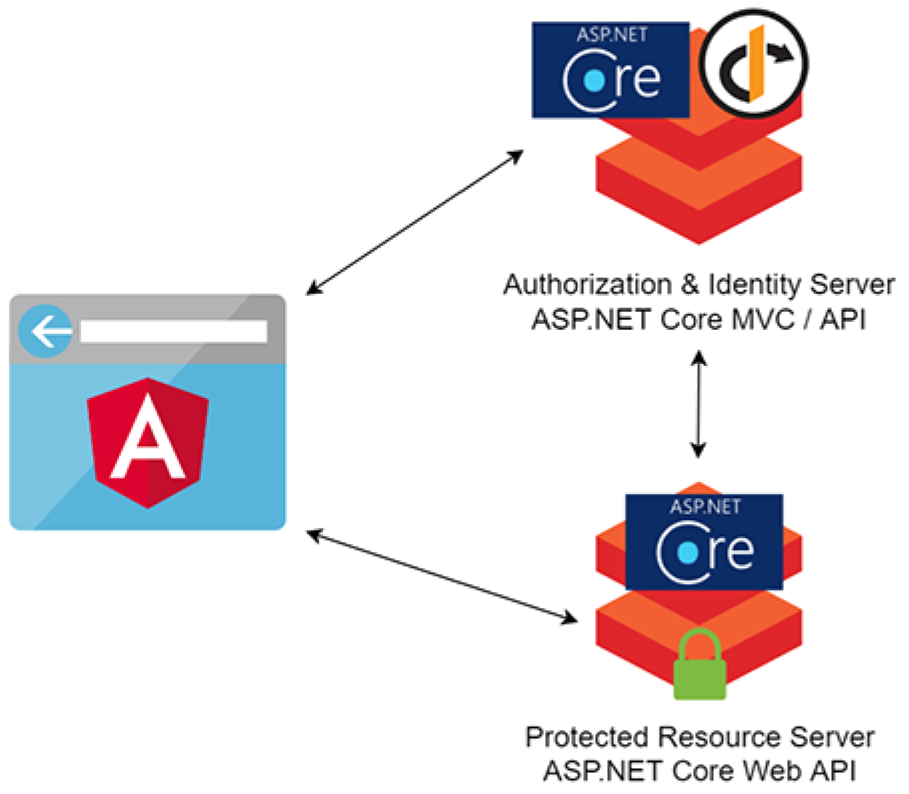
As of March 31, 2019, the demo solution builds and runs successfully with the following tools and SDKs:

- Visual Studio 2019 Community
- Visual Studio Code 1.32.3
- .NET Core SDK 2.2.104
- Angular 7.2.9
- IdentityServer4 2.4.0
- SQL Server Express 2016 LocalDB

Architecture

Our solution architecture has three main components:

- SPA client application - Angular
- Authorization/Identity server - ASP.NET Core MVC and IdentityServer4
- Resource server - ASP.NET Core Web API



Now, we'll step through the process to build out and integrate these components to create the demo solution.

AuthServer

The AuthServer is our OpenID Connect and OAuth provider so it's a great place to start as it is essentially the hub of our solution.

I created a new project using the out of the box ASP.NET Core MVC template. The next step is to bootstrap it with the required IdentityServer packages. I nugetted the main *IdentityServer4* package along with *IdentityServer4.AspNetIdentity*. The second package provides integration between IdentityServer and ASP.NET Core's Identity system which we'll explore shortly. You can use your preferred NuGet method; I use the package manager console in Visual Studio.

```
PM> Install-Package IdentityServer4

PM> Install-Package IdentityServer4.AspNetIdentity
```

Configuring IdentityServer

With the packages installed, we're ready to turn on IdentityServer in our project. IdentityServer uses the standard pattern to configure and add its services to your ASP.NET Core application via the middleware configuration in Startup.cs. In ConfigureServices() the required services are configured and added to the DI system. In Configure() the middleware is added to the HTTP pipeline.

Here are the relevant bits from the Startup class.

```
public void ConfigureServices(IServiceCollection services)
{
...
   services.AddDbContext<AppIdentityDbContext> (options => options.UseSqlServer(Configuration.GetConnectionString("Default")));

   services.AddIdentity<AppUser, IdentityRole>()
     .AddEntityFrameworkStores<AppIdentityDbContext>()
     .AddDefaultTokenProviders();

   services.AddIdentityServer().AddDeveloperSigningCredential()
       // this adds the operational data from DB (codes, tokens, consents)
      .AddOperationalStore(options =>
      {
        options.ConfigureDbContext = builder => builder.UseSqlServer(Configuration.GetConnectionString("Default"));
        // this enables automatic token cleanup. this is optional.
        options.EnableTokenCleanup = true;
        options.TokenCleanupInterval = 30; // interval in seconds
      })
      .AddInMemoryIdentityResources(Config.GetIdentityResources())
      .AddInMemoryApiResources(Config.GetApiResources())
      .AddInMemoryClients(Config.GetClients())
      .AddAspNetIdentity<AppUser>();
...
}
```

Our demo uses the in-memory style for clients and resources, and those are sourced from Config.cs. The framework thoughtfully provides a couple of helpers and in-memory stores, so we don’t have to worry about persistence right from the start for that stuff.

Notice also, the necessary AddDbContext<AppIdentityDbContext> and AddIdentity<AppUser, IdentityRole> calls are done to configure ASP.NET Identity - we'll touch on this next.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
...
  app.UseIdentityServer();
...
}
```

Finally, we add IdentityServer to our request processing pipeline in Configure(), and that's it for now!

User Data and ASP.NET Core Identity

We need somewhere to store our application's user data - things like their credentials, profile information, etc. For this, we can leverage ASP.NET Core's Identity system and use the default data access approach with Sql Server and Entity Framework Core handling persistence. A further benefit of this setup is that the Identity system plugs nicely into IdentityServer to provide user profile and claims data which we'll see shortly.

Create the Database Schema with Entity Framework Core Migrations

Using code-first migrations, we can generate our Sql Server database schema based on the model we define in code. For this demo, that model is pretty tiny and consists of only one custom class AppUser which simply extends the out of the box IdentityUser by adding a Name property.

```
public class AppUser : IdentityUser
{
  // Add additional profile data for application users by adding properties to this class
  public string Name { get; set; }
}
```

Next up, we need a new DbContext class that is aware of our Identity model. IdentityDbContext is the base database context used by Identity so I extended it by creating AppIdentityDbContext. This step isn't required but is valuable if you need to customize or override any defaults in the identity model.

I decoupled things slightly by putting the data access and Entity Framework Core bits in a separate AuthServer.Infrastructure class library project.

To generate and apply the migrations I ran the following on the command line from within that project's folder.

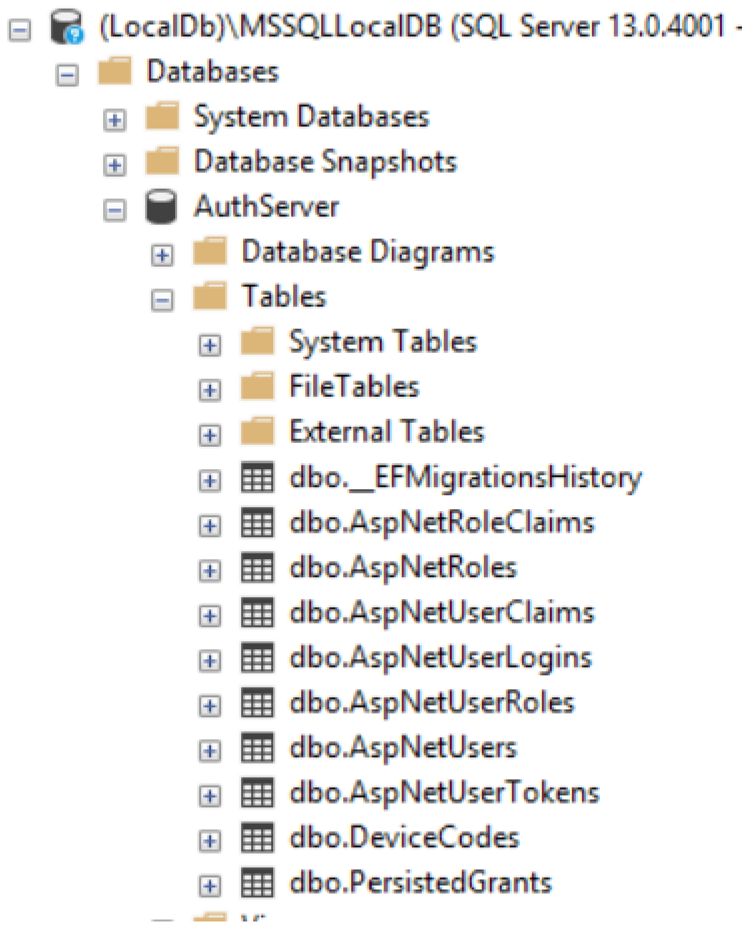
```
AuthServer.Infrastructure> dotnet ef migrations add initial --context AppIdentityDbContext
AuthServer.Infrastructure> dotnet ef migrations add initial --context PersistedGrantDbContext


AuthServer.Infrastructure> dotnet ef database update --context AppIdentityDbContext
AuthServer.Infrastructure> dotnet ef database update --context PersistedGrantDbContext
```

You'll notice a second DbContext here in PersistedGrantDbContext. It belongs to IdentityServer and can be used to store things like authorization grants, consents, and tokens (refresh and reference) in an EF-supported database. So, we must include this during the migration process to model those IdentityServer-related tables in the database.

Note, if you're using some other flavor of Sql Server other than LocalDB you'll need to change the connection strings in these spots:

- AuthServer\appsettings.json
- AuthServer.Infrastructure\appsettings.json

After creating and applying the migrations, I have a shiny new *AuthServer* database and tables in my LocalDB instance.


```
(LocalDb)\MSSQLLocalDB (SQL Server 13.0.4001 -
  Databases
    System Databases
    Database Snapshots
    AuthServer
      Database Diagrams
      Tables
        System Tables
        FileTables
        External Tables
        dbo.__EFMigrationsHistory
        dbo.AspNetRoleClaims
        dbo.AspNetRoles
        dbo.AspNetUserClaims
        dbo.AspNetUserLogins
        dbo.AspNetUserRoles
        dbo.AspNetUsers
        dbo.AspNetUserTokens
        dbo.DeviceCodes
        dbo.PersistedGrants
```

Ok, we've got a good start on the AuthServer, there's more work to do here, and we'll revisit that shortly, but for now, let's shift gears and take a look at building out our Angular SPA client.

Angular SPA

Frontend time! Our angular client is where the rubber meets the road so to speak and is where we'll see the OAuth dance take place.

Angular CLI

The Angular CLI allows us to quickly spin up a brand new Angular app bootstrapped with the configuration and foundational bits required to get our app up and running in no time.

It's not required to work with Angular, but I prefer it as it comes with best practices baked in and provides a friendly, consistent and automated approach to working with your Angular project.

First, I installed the CLI globally using npm.

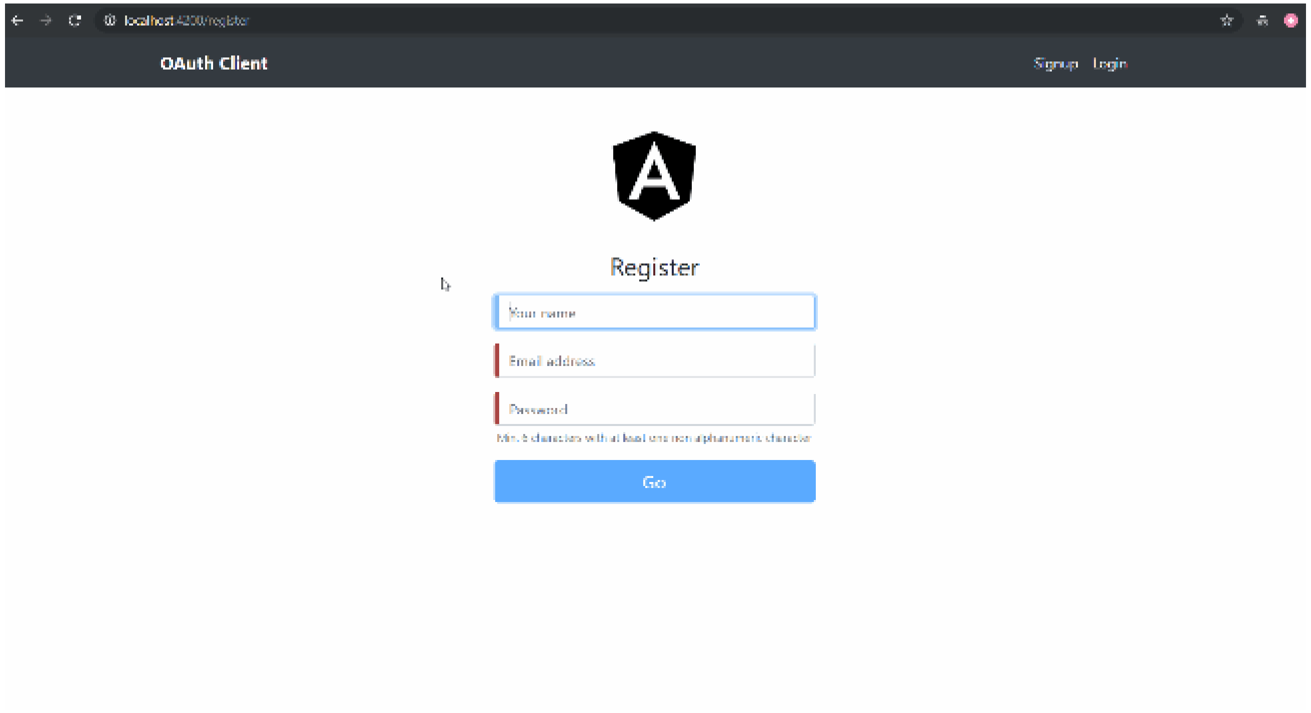
```
Spa> npm install -g @angular/cli
```

After that, I created my app using ng new.

```
Spa> ng new oauth-client
```

User Registration Flow

We need a mechanism to populate our Identity database with some actual user data, i.e., credentials, profile data, etc. For demo purposes, we could seed the database directly, but I thought in the spirit of creating a more real-world demo we should add this responsibility to our Angular client by adding a new registration form/flow.



Organizing Components with Modules

Rather than create the new registration component directly in the root of the project, I want to add a little more structure to things.

Modules do just that by allowing us to group related features and partition our code into more focused areas.

Using the CLI, I created a new account module.

```
oauth-client\src\app> ng generate module account
```

Next, I returned to the CLI and spun up a new registration form component within the account module.

```
oauth-client\src\app\account> ng generate component register
```

Finally, I got down to business by designing and implementing the component template markup and styling and its backing code.

I won't go into great depth on the template details but encourage you to explore the code yourself if you're interested in learning more.

The component code itself is pretty straightforward. Aside from boilerplate, the only slightly interesting aspect here is the onSubmit() method that is triggered when the form is submitted.

```
...
onSubmit() {

    this.spinner.show();

    this.authService.register(this.userRegistration)
      .pipe(finalize(() => {
        this.spinner.hide();
      }))
      .subscribe(
      result => {
         if(result) {
           this.success = true;
         }
      },
      error => {
        this.error = error;
      });
}
...
```

This simply calls the register() method on authService passing it the form data. We'll take a closer look at what the code in authService is doing shortly but from the point of view of our component here we simply subscribe and handle the API call and resulting observable by setting either the success or error properties on our component which are reflected automatically in our template thanks to Angular's data binding.

Register API Endpoint

Back on the AuthServer, we have the associated API endpoint our registration component and authService utilize to create new users on the backend.

The Register() action lives in the AccountController where we simply call on the Identity framework's _userManager to create the new user account based on the provided RegisterRequestViewModel data.

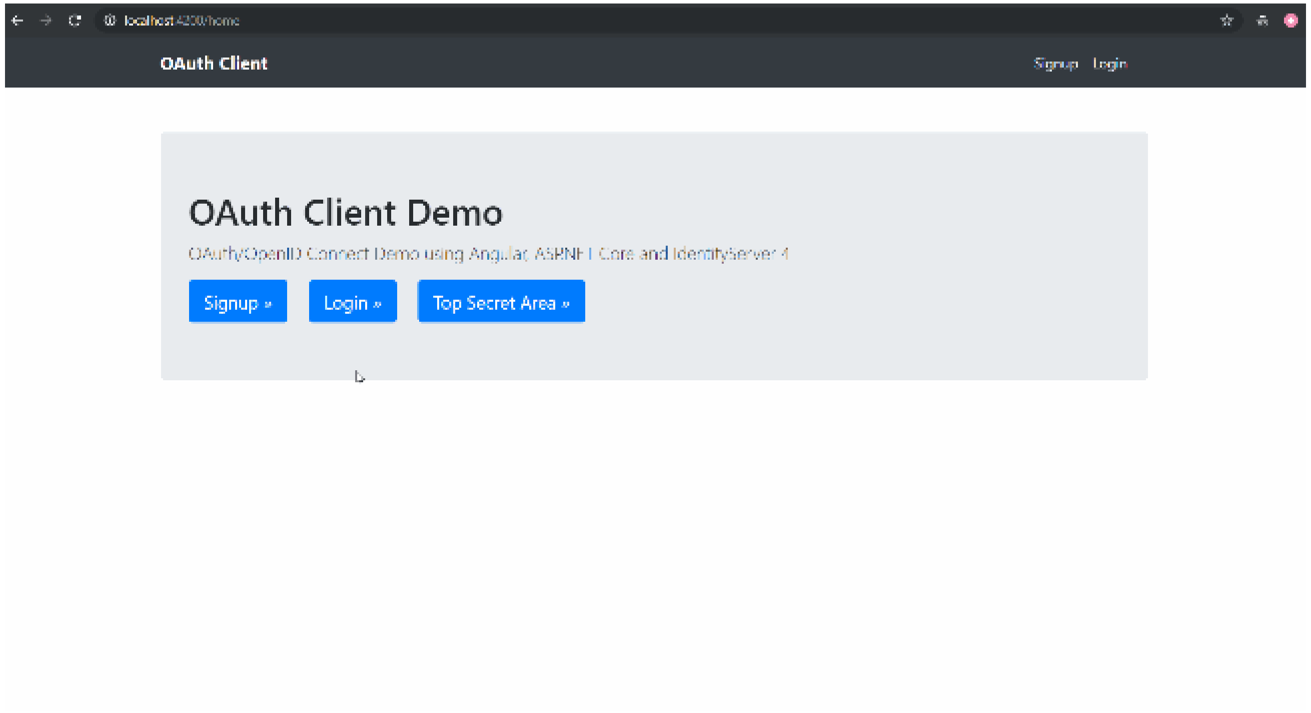
Following the successful creation of our user, we attach and store a few claims via _userManager.AddClaimAsync(). These claims get embedded in the tokens returned to our SPA client during the authentication phase.

To verify this is working you can check the AspNetUsers and AspNetUserClaims tables in the database.

```
[HttpPost]
[Route("api/[controller]")]
public async Task<IActionResult> Register([FromBody]RegisterRequestViewModel model)
{
  if (!ModelState.IsValid)
  {
     return BadRequest(ModelState);
  }

  var user = new AppUser { UserName = model.Email, Name = model.Name, Email = model.Email };

  var result = await _userManager.CreateAsync(user, model.Password);

  if (!result.Succeeded) return BadRequest(result.Errors);

  await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("userName", user.UserName));
  await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("name", user.Name));
  await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("email", user.Email));
  await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("role", Roles.Consumer));

  return Ok(new RegisterResponseViewModel(user));
}
```

Login Flow

With the functionality laid down to register and persist new users, we can now turn our attention to authorizing them and exercising the OIDC and OAuth interaction between our Angular app and authorization server.



OAuth Client Demo

OAuth/OpenID Connect Demo using Angular, ASP.NET Core and IdentityServer 4

The client application initiates the authorization interaction, so the first step is to create a new [login component](#) for that functionality to live in.

I followed the identical process we used to create the registration component so I'll save some keystrokes and omit those CLI steps here. The login component template and code are dead simple. Similarly to the register component, we're calling a method on the authService to do the heavy lifting - let's dig more into that right now!

```
...
login() {
  this.spinner.show();
  this.authService.login();
}
...
```

Auth Service

The [auth.service](#) is responsible for authentication, managing user data and sesssions in our Angular app primarily through the [oidc-client](#) library which handles all the interactions with our OpenID Connect Provider.

oidc-client

To bring the oidc-client library into the project, I first installed it via npm and confirmed it was added as a dependency in package.json.

```
oauth-client> npm install oidc-client --save
```

Next, I imported UserManager, UserManagerSettings, and User into the auth.service from the oidc-client library.

```
import { UserManager, UserManagerSettings, User } from 'oidc-client';
```

The UserManager class is the main interaction point with the oidc-client library. It provides a high-level API for signing a user in, signing out, managing the user's claims returned from the OIDC provider, and managing an access token returned from the OIDC/OAuth2 provider. The UserManager is the primary feature of the library.

The UserManager's constructor requires a settings object of UserManagerSettings so for this demo, these are hardcoded in the auth service but in a production setting these should be pulled out into a more robust configuration.

```
export function getClientSettings(): UserManagerSettings {
  return {
      authority: 'http://localhost:5000',
      client_id: 'angular_spa',
      redirect_uri: 'http://localhost:4200/auth-callback',
      response_type:"id_token token",
      scope:"openid profile email api.read"
  };
}
```

The required settings have these properties:

- **authority**: The URL of the OIDC/OAuth2 provider. The AuthServer URL in our demo ("http://localhost:5000")
- **client_id**: Your client application's identifier as registered with the OIDC/OAuth2 provider. ("angular_spa")
- **redirect_uri**: The redirect URI of your client application to receive a response from the OIDC/OAuth2 provider. ("http://localhost:4200/auth-callback")
- **response_type**: The type of response desired from the OIDC/OAuth2 provider. ("id_token token")
- **scope** The scope(s) being requested from the OIDC/OAuth2 provider. ("openid profile email api.read")

Note, on the server side we have a similar client configuration defined to represent applications that can request tokens from our IdentityServer. If you must change the URL locations in the demo of the SPA client or AuthServer you'll need to change them accordingly to ensure they match your environment.

```
 public static IEnumerable<Client> GetClients()
 {
    return new[]
    {
          new Client {
                 ClientId = "angular_spa",
                 ClientName = "Angular SPA",
                 AllowedGrantTypes = GrantTypes.Implicit,
                 AllowedScopes = { "openid", "profile", "email", "api.read" },
                 RedirectUris = {"http://localhost:4200/auth-callback"},
                 PostLogoutRedirectUris = {"http://localhost:4200/"},
                 AllowedCorsOrigins = {"http://localhost:4200"},
                 AllowAccessTokensViaBrowser = true,
                 AccessTokenLifetime = 3600
         }
     };
}
```

With the settings defined, we can initialize a UserManager instance by supplying the required settings object as a parameter.

```
private manager = new UserManager(getClientSettings());
```

Next, we can initialize and save off a local variable to hold our user (if present in session storage) in the auth service constructor. We'll rely on the user object to flesh out most of the essential methods in the service.

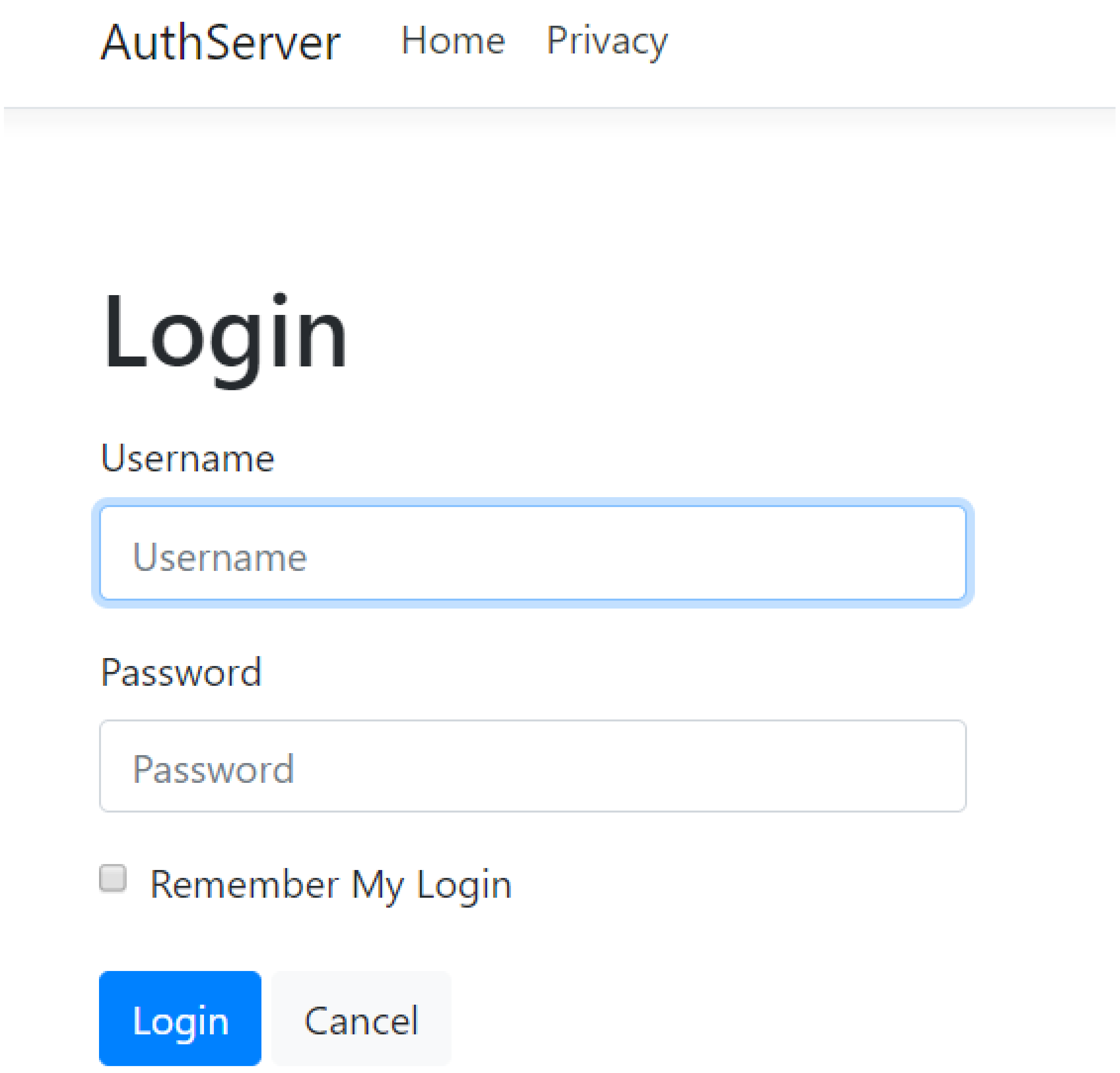
```
constructor(private http: HttpClient, private configService: ConfigService) {
 super();

 this.manager.getUser().then(user => {
    this.user = user;
    this._authNavStatusSource.next(this.isAuthenticated());
 });
}
```

The login() method triggers the authentication flow; it directs out us out of the Angular client and into the authorization server based on the authority setting we defined above.

```
login() {
  return this.manager.signinRedirect();
}
```

The login view we are brought to on the authorization server is driven off the AccountController.

I borrowed a lot of the code for the razor views, controllers and models from the Quickstart.UI repository which is excellent for quickly bootstrapping your ASP.NET Core MVC host project with the necessary code and UI bits for getting IdentityServer up and running fast - this is entirely optional of course.



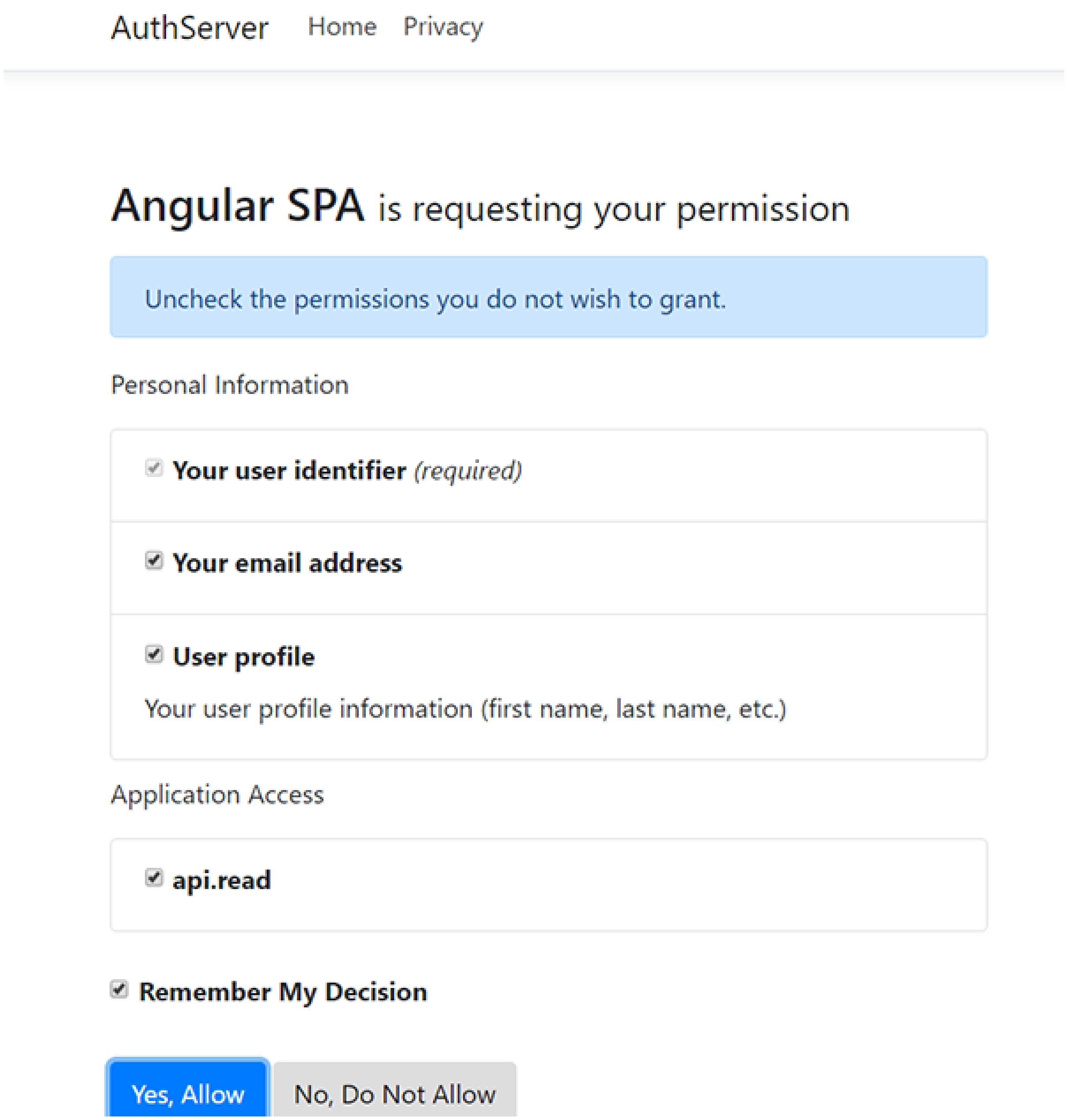
The critical functionality inside the controller action that handles the postback of the credentials is where we authenticate the user.

For this, once again we're relying on the Identity system's UserManager APIs to validate the username and password and then delegating control back to IdentityServer to process the remainder of the request based on the current context.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginInputModel model, string button)
{
...
 if (ModelState.IsValid)
 {
    // validate username/password
    var user = await _userManager.FindByNameAsync(model.Username);
    if (user != null && await _userManager.CheckPasswordAsync(user, model.Password))
    {
        await _events.RaiseAsync(new UserLoginSuccessEvent(user.UserName, user.Id, user.Name));
...
}
```

Consent

The user is presented with a consent page if the authentication check succeeds. This page allows them to grant the client access to resources (identity or API). This is optional and can be disabled in the client settings configuration by adding the following property: RequireConsent = false.



Angular SPA is requesting your permission

Uncheck the permissions you do not wish to grant.

Personal Information

✓ **Your user identifier** *(required)*

✓ **Your email address**

✓ **User profile**

Your user profile information (first name, last name, etc.)

Application Access

✓ **api.read**

✓ **Remember My Decision**

Auth Callback

Once consent is granted, and the authentication process is complete on the IdentityServer the final leg of the journey is a redirect callback to the client to pass along the newly minted identity, and access tokens returned from the OpenID Connect Provider.

Note, this is the component/route we specified as the redirect_uri in our client configuration: *http://localhost:4200/auth-callback*.

For this, I generated a new auth-callback component in the Angular app which performs one of two possible actions based on the status of the redirect URL returned from the authentication step.

```
async ngOnInit() {

// check for error
if (this.route.snapshot.fragment.indexOf('error') >= 0) {
    this.error=true;
    return;
}

 await this.authService.completeAuthentication();
 this.router.navigate(['/home']);
}
```

We're just checking the URL during onInit() of the component and in the event of an error, processing halts, and we display a message in the UI template informing the user of the failure.

The happy path calls the completeAuthentication() method on the authService which is another wrapped oidc-client call to signinRedirectCallback() that receives and handles incoming tokens, including token validation. At this point, we've effectively closed the loop and completed the authentication process.

If loadUserInfo is enabled in the client configuration, it also calls the user info endpoint to get any extra identity data it has been authorized to access. This method returns a promise of the authenticated user, which we can then assign locally for convenient access in a few other places in the service.

```
async completeAuthentication() {
   this.user = await this.manager.signinRedirectCallback();
   this._authNavStatusSource.next(this.isAuthenticated());
}
```

The second line this._authNavStatusSource.next(this.isAuthenticated()) is using an observable to emit the current authentication status returned by isAuthenticated().

This value is used to update the header component in the UI with the appropriate state based on whether or not the user is logged in. The same call occurs in the constructor after loading the user object.

isAuthenticated() checks if we have a user and if so, determines if they are still valid. This check uses the expired property, which calculates if the user’s access token has expired or not.

```
isAuthenticated(): boolean {
    return this.user != null && !this.user.expired;
}
```

Guarding Protected Routes

Now that we can determine whether or not a user is logged in, we can set up a route guard to protect any sensitive or super-secret areas of our client application.

I created a new auth.guard that has a dependency on auth.service and simply allows access to the activated route when the user isAuthenticated(). Otherwise, navigate them to the login page to begin the authentication process.

```
export class AuthGuard implements CanActivate {

  constructor(private router: Router, private authService: AuthService) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    if (this.authService.isAuthenticated()) { return true; }
    this.router.navigate(['/login'], { queryParams: { redirect: state.url }, replaceUrl: true });
    return false;
  }
}
```

To test the guard, I generated a new top-secret module and applied the guard to its lone child route in the routing module.

```
const routes: Routes = [
Shell.childRoutes([
    { path: 'topsecret', component: IndexComponent, canActivate: [AuthGuard] }
  ])
];
```

Accessing Protected API Resources

It's great that we can protect areas of our local angular application from unauthorized access. But now we can realize the full power of the OAuth/OIDC authorization model by accessing protected resources hosted on an independent server that happens to be protected by our OpenID Connect Provider. To enable this I created a new ASP.NET Core Web API project called ResourceApi.

This API project is dead simple, but the key is in the Startup.cs class where we configure the JWT bearer authentication handler in the DI configuration by pointing the Authority to our IdentityServer/OpenID Connect Provider.

Just below that, we define two authorization policies that use claim values stored in the JWT access token to enforce some more granular protection on our controllers and actions. The api.read claim is one of our allowed client scopes, and consumer is a user-level role claim we added during the registration process.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(options =>
    {
        options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    }).AddJwtBearer(o =>
    {
        o.Authority = "http://localhost:5000";
        o.Audience = "resourceapi";
        o.RequireHttpsMetadata = false;
    });

    services.AddAuthorization(options =>
    {
       options.AddPolicy("ApiReader", policy => policy.RequireClaim("scope", "api.read"));
       options.AddPolicy("Consumer", policy => policy.RequireClaim(ClaimTypes.Role, "consumer"));
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Now, we need to put our authorization policies to work.

In the ValuesController we apply both policies by decorating the controller and Get() action method with the Authorize attribute and specify the desired policy name for each. With this in place, the only requests to be granted access to the */api/values* endpoint will be those containing JWT tokens with both of these claims, satisfying both policies. The action method returns a collection of the user's claims as a response to prove we were there.

```
namespace Resource.Api.Controllers
{
  [Authorize(Policy = "ApiReader")]
  [Route("api/[controller]")]
  [ApiController]
  public class ValuesController : ControllerBase
  {
    // GET api/values
    [Authorize(Policy = "Consumer")]
    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
      return new JsonResult(User.Claims.Select(c => new { c.Type, c.Value }));
    }
  }
}
```

Making Protected API Requests

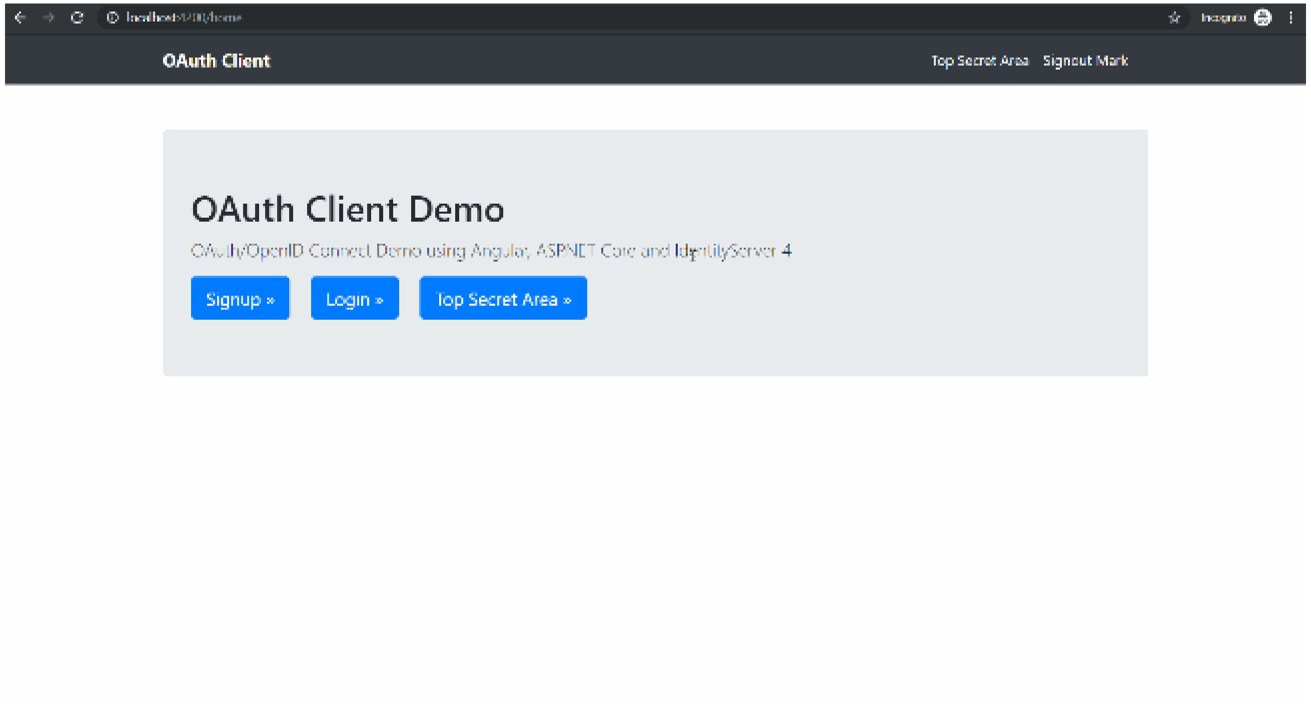
To reach our secured ASP.NET Core API endpoint from the Angular client, I created a new top-secret-service with a fetchTopSecretData() method to issue a GET request to the protected endpoint. This method accepts a token parameter that gets added to the authorization header which is passed along with the request.

```
export class TopSecretService extends BaseService {

 constructor(private http: HttpClient, private configService: ConfigService) {
   super();
 }

 fetchTopSecretData(token: string) {

   const httpOptions = {
     headers: new HttpHeaders({
       'Content-Type':  'application/json',
       'Authorization': token
     })
   };

  return this.http.get(this.configService.resourceApiURI + '/values', httpOptions).pipe(catchError(this.handleError));
 }
}
```

Next, to utilize the new top-secret-service I injected it into the index component within the top-secret module and issued a call to fetchTopSecretData() during the component's onInit() lifecycle method. If the call succeeds, it just sets the claims property to the response and displays the claims data received from the protected API.

```
export class IndexComponent implements OnInit {

  claims=null;
  busy: boolean;

  constructor(private authService: AuthService, private topSecretService: TopSecretService, private spinner: NgxSpinnerService) {
  }

  ngOnInit() {

    this.busy = true;
    this.spinner.show();

    this.topSecretService.fetchTopSecretData(this.authService.authorizationHeaderValue)
    .pipe(finalize(() => {
      this.spinner.hide();
      this.busy = false;
    })).subscribe(
    result => {
      this.claims = result;
   });
  }
}
```

With everything in place, our protected request succeeds, and we can fetch the claims data! 💥💥💥



Wrapping Up

I hope you've found some value in this guide. If you have any feedback, I'd love to hear from you in the comments below.

Source code is here and detailed instructions on running the demo solution can be found in the repository readme.

Special thanks and shout out to Scott Brady for his excellent article on integrating Angular with the oidc-client library which helped immensely in composing the related bits of this guide.

Hi, I'm Mark Macneil — a .net guy who spends most days making software in friendly Halifax, NS.

Elsewhere

1. GitHub
2. LinkedIn
3. Email

Support

Buy me a coffee

Recent

Archive

1.7k Shares
Share
Share
Tweet
Share
Email