IdentityServer in Production

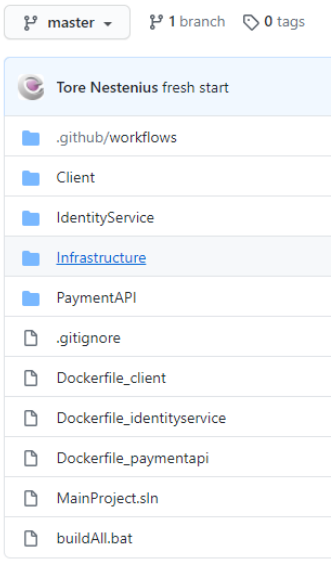
T339

Main Exercises

Main exercise 1 – Deployment setup

Exercise 1.1 – Deploying the main project

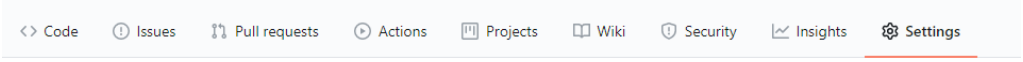
1. This exercise assumes you have followed the course preparation as stated in the **Student installation guide.pdf** document, including:
    a. Created your own **GitHub repository**
    b. Added the **start-project** source code to it and pushed it to your GitHub account
    c. In GitHub it should look something like this:



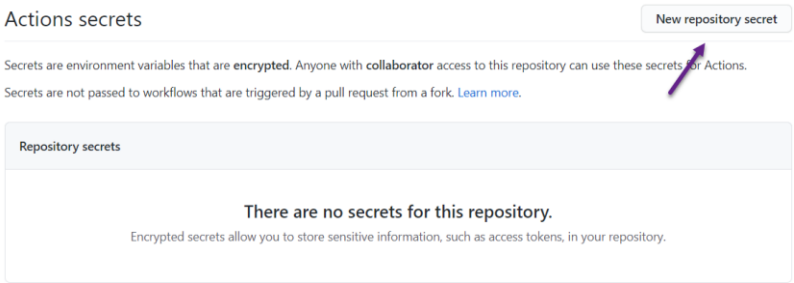
2. To be able to deploy these projects to the cloud, we need to first add a few **secrets** to your GitHub repository.

    a. Open **GitHub** in your browser and navigate to your repository

    b. Click on **Settings**



and then click on **Secrets** in the left column



c. Open your personal **\Config\StudentX\Infrastructure.txt** file and add the following **secrets** (one by one):

Name	Value
REGISTRY_PASSWORD	The container registry password
USERNAME	Your unique username, like studentx (must be all **lowercase**)

It is important that the names are exactly as listed above. You find the values to add at the top of the **Infrastructure.txt** file)

d. Open the file **azure_credentials.txt** and create the following secret in GitHub:

Name	Value

AZURE_CREDENTIALS	Copy the JSON object including the { }
	{
	"clientId": "xxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx",
	"clientSecret": " xxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx",
	"subscriptionId": " xxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx",
	...
	}

e. Add the following secrets that we will use later. For now, just set some dummy value.

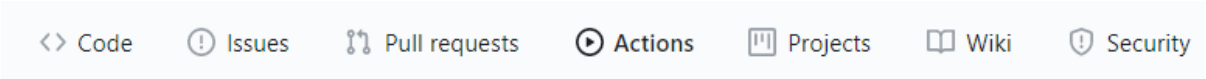
Name	Value
AZURE_CLIENTID	x
AZURE_SECRET	x
AZURE_TENANTID	x
AZURE_VAULTURL	x

f. When done you should have the following secrets added:

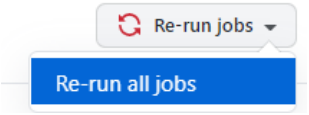
AZURE_CLIENTID
AZURE_CREDENTIALS
AZURE_SECRET
AZURE_TENANTID
AZURE_VAULTURL
REGISTRY_PASSWORD
USERNAME

3. Let's try to build and deploy the solution.

   Make sure the code is committed to GitHub and then click on the **Actions** tab in GitHub.



   Click on the first failing job in the **All Workflows list** and in the upper right corner **re-run** the deployment work by clicking on the **Re-run jobs** button.



4. Go back to the **Actions view** and watch the deployment work progress. After a few minutes everything should be green.

   The action will automatically run each time you make a **push** to GitHub.

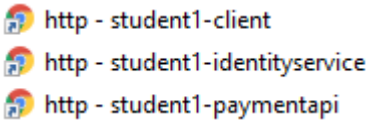
   The GitHub action will start the build-script found in the **\.github\workflows** folder in your repository. This script will be executed each time you push to GitHub. Feel free to look what that script contains.

   Wait for the work to complete successfully before you continue.
   (Contact the teacher if this deployment step does not work.)

Exercise 1.2 – Accessing the deployed services

1. Now, let's try to access the three services.

   In your **\Config\StudentX** folder click on the link to **Client**, **IdentityService** and **PaymentAPI** and if all is set up correctly you should see the three services up and running.



   (The ***-secure.nu** links will not work at this moment.)

   Now, each time you push your changes, GitHub will automatically deploy the changes to the cloud. Awesome! The deploy time should be around 2 minutes.

Exercise 1.3 – Fixing the culture

Let's warm up with a simple change to the project:

1. The start project prints out the time the application started, just so that we can figure out when the application was last deployed.

   As you notice the culture is not correct for the application because the dates are in the format:

   ```
   Hello Identity Service!, Deployed 08/17/2020 14:36:48
   ```

   We would prefer to have it in the **yyyy-mm-dd** format instead.

2. To change this, add the following middleware in **Startup.Configure** method after **UseRouting():**

   ```
   app.UseRouting();

   app.UseRequestLocalization(
       new RequestLocalizationOptions()
           .SetDefaultCulture("se-SE"));
   ```

   (Place it after UseRouting())

   Add this to all **three web projects**.

   A list of .NET culture and country codes can be found here:
   https://lonewolfonline.net/list-net-culture-country-codes/

   Push the code to GitHub to deploy it and you should see that the date is now displayed in the correct format. The total build/deploy time is usually around 2-3 minutes.

   Having a build-date/version on your production site will help you to determine if the version is what you expect.

Exercise 1.4 – Profiles and the environment

1. We want to run our application in three different environments:
   - **Development**
     - Development mode
     - External database and dependencies
   - **Offline**
     - Development mode
     - Local database and no external dependencies
   - **Production**
     - Production release mode
     - External database and external dependencies

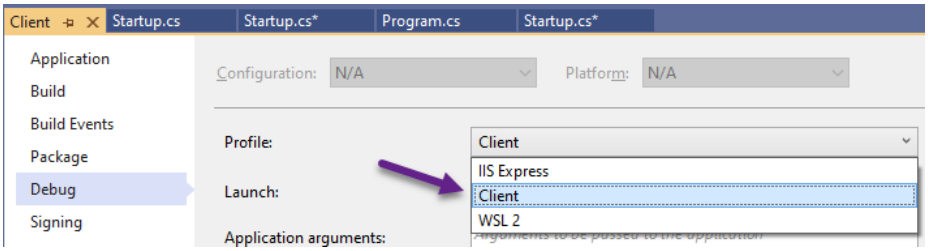
The **ASPNETCORE_ENVIRONMENT** environment variable is used to select what environment to run out application under.

Visual Studio 2019

2. We will use the built-in **debug profiles** to allow us to choose environment to use during development.

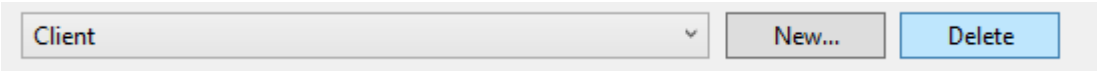
   Let's create the necessary profiles!

   First open the Client **project properties** and locate the **Debug** profiles here:



3. Choose the **Client profile** and then **copy** the following strings in that profile and save them in a plain text editor:
   - ASPNETCORE_ENVIRONMENT=Development
   - https://localhost:5001;http://localhost:5000

4. **Delete** all existing **profiles**



5. Let's create three new **profiles** for the three supported environments.

   Click on the **New** button and create the following three profiles:

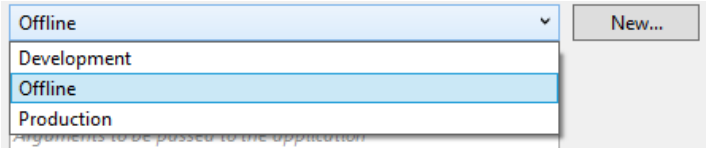
   - Development
   - Offline
   - Production

   With the following settings

   - Launch type: **Project**
   - Check **Launch browser**
   - Add the **ASPNETCORE_ENVIRONMENT** environment variable and set it to **Development**, **Offline** and **Production** respectively.

- **App URL** https://localhost:5001;http://localhost:5000

As a result, you should now have three profiles, with the same properties except for the environment variable:

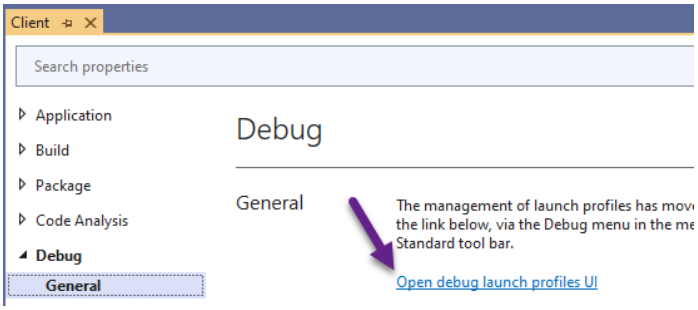


Visual Studio 2022

2. We will use the built-in **debug profiles** to allow us to choose environment to use during development.

   Let's create the necessary profiles!

   First open the Client **project properties** and click on **Open the debug launch profiles UI**:

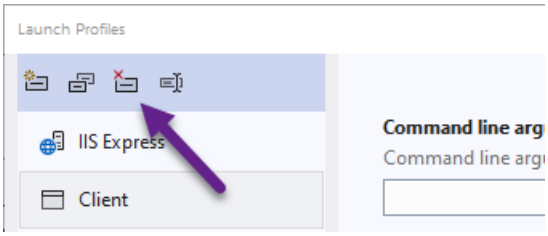


3. Choose the **Client profile:**

   Then **copy** the existing **Environment variables** and **App URL** values in that profile and save them in a plain text editor. Meaning these two values:

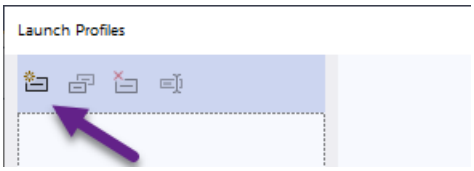
ASPNETCORE_ENVIRONMENT=Development https://localhost:5001;http://localhost:5000

4. **Delete** all existing **profiles** using the delete button:

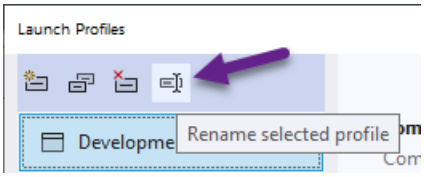


5. Let's create a new **Client project profile** for the development environment:

   Click on the **New** button

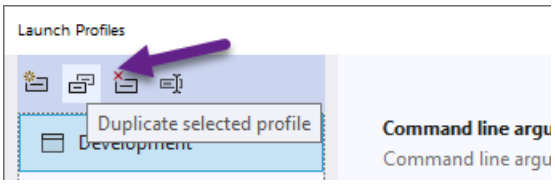
   Use the rename button to rename the project to **Development**:

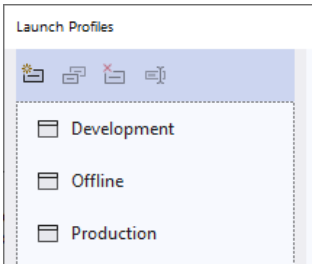
   Set the following settings

   - Copy the value you saved earlier to the **Environment variables**
   - Check **Launch browser**
   - Copy the **App URL** value.

   Then click on the duplicate button two times to create two new copies of the existing profile:

   rename them as follows:

   In the Offline profile set the environment to:
   ASPNETCORE_ENVIRONMENT=Offline

   In the Production profile set the environment to:
   ASPNETCORE_ENVIRONMENT=Production

   As a result, you should now have three profiles, with the same properties except for the environment variable.

6. Close the profiles window and rebuild the application.

7. Open the **Properties\launchSettings.json** in the **Client** project and delete the **iisSettings** object. We don't need it.

   In this file you should see your three profiles listed and the settings should be the same except the environment variable.

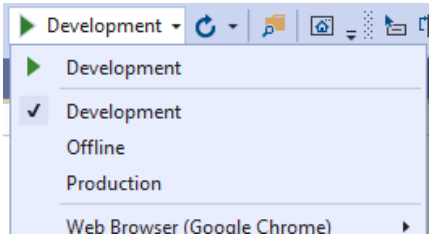
   Some rules and facts about this file are:

   - Never store secrets in this file
   - The file is only used during development
   - When publishing your application for production, the file is not used or included.

8. Save the **launchSettings.json** file and then copy it to the two other **web projects** with the following changes:

   - **IdentityService**
     - Set the ports to 6001 and 6000
   - **PaymentAPI**
     - Set the ports to 7001 and 7000

   Now you can choose the profile to run under by selecting it next to the run button:



   Give it a try, and verify that in the client you will see the current environment at the bottom of the client page:

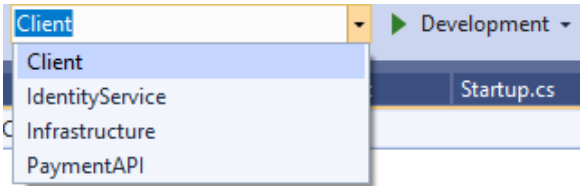
9. Open the **Startup** class and you will find that we have tweaked the following **if-statement** for you in the starter projects:

```
if (!env.IsProduction())
{
    app.UseDeveloperExceptionPage();
}
```

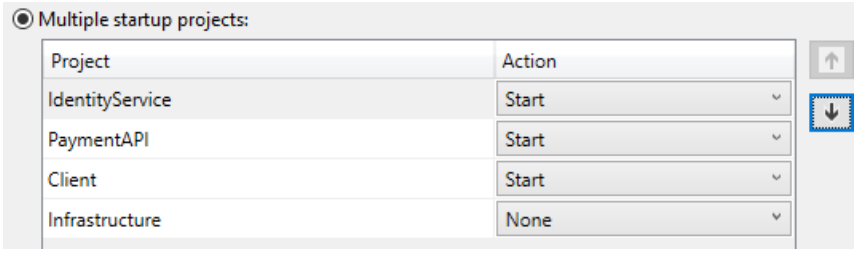
   (We only show the developer exception page when we are not in production)

Exercise 1.5 – Solution startup

1. First, make sure **all** the three web projects are using the **Development** profile. You can use the start project selector to quickly iterate through the projects:



2. When we start our solution, we want all the applications to start at the same time. Open the **properties** for the Solution and set all the web projects to **Start**:

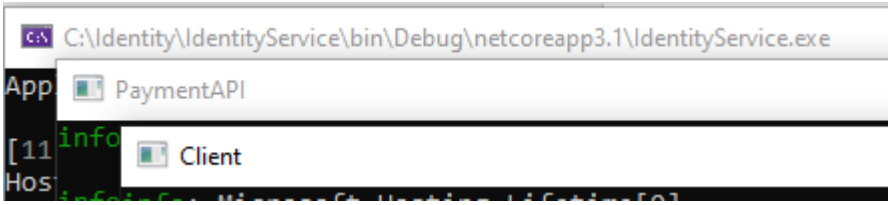


Also rearrange so the order is as shown above.

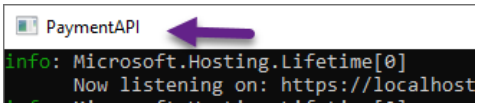
3. The question, under what profile does the project now startup under? When we use multiple startup projects, the last profile used is used. So that is why we made sure all projects are under the development profile before we made the switch to multiple projects.

4. Start the projects and you should see that all three sites start and that the ports are correct as we configured them.

5. We also see that one **console window** is opened for each application, as below:



6. To better allow us to tell the console windows apart, it can be a good idea to set the title of the console windows:



To do this, add the following to the top of your **Program.Main** method in each web project:

```
public static void Main(string[] args)
{
    Console.Title = _applicationName;
```

Run the projects again and check the title of the console windows.

7. Push the code to GitHub and make sure everything still works as expected and that the date is now displayed in the correct format!

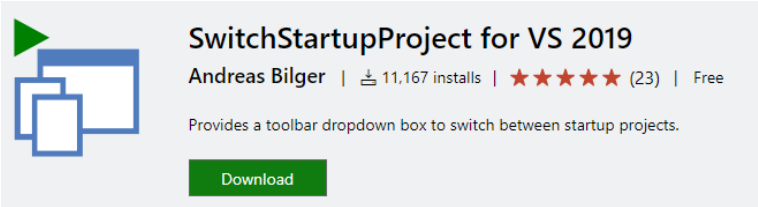
Payment API

Service started: 2020-08-29 13:47:32 (Production)

Congratulations, you have now pushed your first changes to production!

If you have time

- There's a Visual Studio extension that can help if you need to switch projects and profiles a lot:



Read more about it here:
https://marketplace.visualstudio.com/items?itemName=vs-publisher-141975.SwitchStartupProjectForVS2019

It also exists for Visual Studio 2022.

Further reading

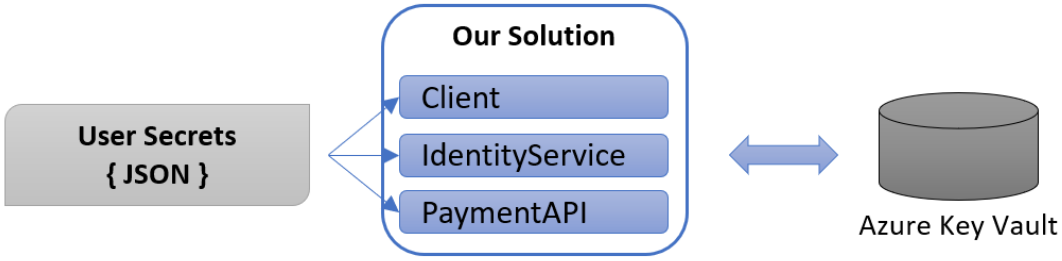
- GitHub Actions Documentation
  https://docs.github.com/en/actions
- GitHub Actions for .NET Core NuGet packages
  https://dusted.codes/github-actions-for-dotnet-core-nuget-packages
- Building .NET Framework Applications with GitHub Actions
  https://www.codingwithcalvin.net/building-net-framework-applications-with-github-actions/

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise 2 – Configuration

Let's add support for **User Secrets** and **Azure Key Vault** to our main application with these core requirements:

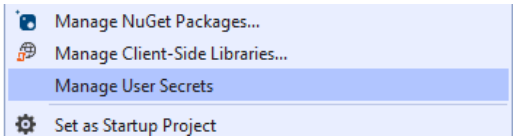
- We share the same **user secrets file** for all the web projects
- We use one shared **key vault** for development and production.
- In offline mode we don't use Azure Key Vault.



Exercise 2.1 – Adding support for User Secrets and Azure Key Vault

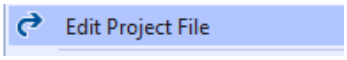
1. Open the **Main project** again.

2. Right click on the **Client** project and select **Manage User Secrets**. A **secrets.json** file is created for you.



3. In our solution we want to share the same secrets file for all three web projects. To achieve this, do the following:

   a. Right click on the Client project and click on **Edit Project File** (or double click on it)



   b. Copy the **UserSecretsId** tag:

```
<UserSecretsId>XXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX</UserSecretsId>
```

   Then paste it into the **IdentityService** and **PaymentAPI** project files. Place it within the **PropertyGroup** tag.

   Now all three projects are referring to the same **secrets.json** file.

4. Copy the content of the user secrets file you used in the module.

   Now all the **web projects** will use the same **secrets.json** file. That's convenient!

Exercise 2.2 – Adding Azure Key Vault support

1. First add following NuGet packages to the **Infrastructure** project:

   - Azure.Identity
   - Azure.Security.KeyVault.Certificates
   - Azure.Security.KeyVault.Secrets
   - Azure.Security.KeyVault.Keys
   - Azure.Extensions.AspNetCore.Configuration.Secrets

2. Add the class **KeyVaultExtensions.cs** found in **\Starter-kit\Exercise 2** to the shared **Infrastructure** project.

   Review the code and see what it does. It is basically a custom **extension method** to add **Azure Key Vault** support to our application.

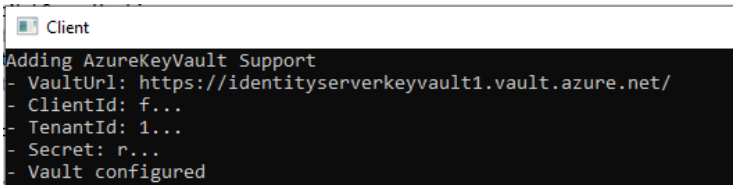
3. Compile the code and make sure it builds.

4. To add **Azure Key Vault** as a configuration provider, add the following in the **Program.cs** class in all **three** web projects:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            if(context.HostingEnvironment.EnvironmentName != "Offline")
                config.AddAzureKeyVaultSupport();
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

   (We add Azure Key Vault support to all environments except the Offline one)
   (The entire method is also found in the **\Starter-kit\Exercise 2\Exercise 2.2.txt** file)

5. Start all the projects and you should see this output in all three console windows:



   (To assist in debugging, we print out the first character of each secret)

6. Let's see if we can access the secret, you added earlier in **Azure Key Vault**.

7. Modify the **Client\Home\Index** view and add the following at the top:

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration config
```

Then further down the page display it using:

```
mysecret: @config["mysecret"]
```

(If you don't see any, then make sure you added the secret in the module exercise we just did)

Start the application and verify that the secret is displayed.

Exercise 2.3 – Adding support for Azure Key Vault in production

1. Our application now works fine on our machine, but we have a few things to do to get it to work in production.

2. How should we pass the **Vault credentials** to the application in production?

   Obviously we can't use the **secrets.json** file in production.

   In our setup we will add the **Azure Key Vault** credentials to **GitHub** and then make sure that the **GitHub action** injects these credentials into each application container at deployment time using environment variables.

   Open **Github.com** and navigate to the **repository** for this project.

   Under **secrets** update the following secrets with the values from your user secrets file.

   - AZURE_VAULTURL
   - AZURE_CLIENTID
   - AZURE_TENANTID
   - AZURE_SECRET

   Make sure you name them correctly and add the corresponding value from your **secrets.json** file. Make sure you don't include the quotes in the strings or empty lines.

3. Save all the files and then **commit** the code to GitHub. The code should now deploy to the cloud. Make sure the GitHub action deployment passes.

   The secrets are passed as environment variables to the containers in the **.github\workflows\main.yml** GitHub Action file. Look if you are interested in how they are passed to the application in production.

The funny looking **Vault__Url** environment variable points to the configuration string **Vault:Url** that points to the **Url** value in the **Vault** configuration object.

When done, visit the three sites and they should all be up and running. If the vault initialization fails, then the sites will not start up.

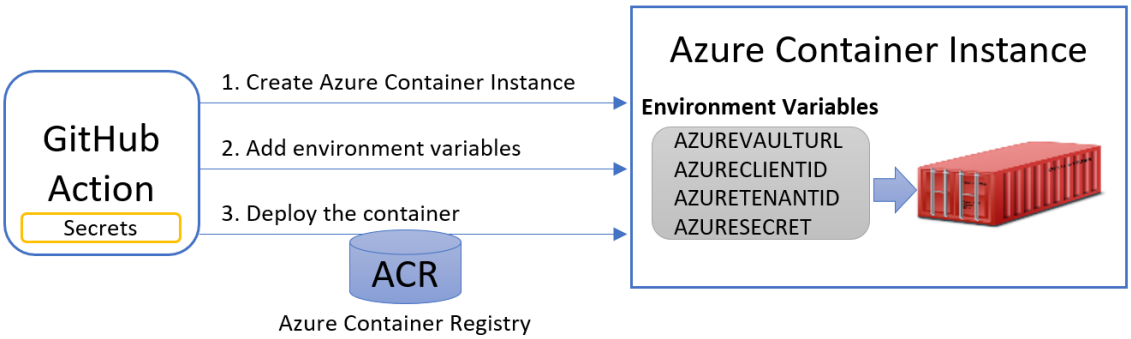
The secret from Vault should be displayed on the client start page as well.

If it fails, then ask your teacher.

4. Where are the secrets stored in Azure?

   The secrets are not stored inside the container. Instead, they are stored in the Azure infrastructure and passed to the container at startup as environment variables.

   This means that the secrets are not stored in the container image in the registry or inside the container itself. If a container image is "stolen", they don't get the secrets.



To do if you have time:

- Currently the **Production** environment can't be executed locally. The problem is that the the four environment variables as used above are missing locally.

  How can you pass them to the application?
  - You can't use the User Secrets in **secrets.json** because they are only available in the development environment.
  - Storing them in the **offline debug profile** could work but is not recommended. Because then they would eventually be stored in GitHub when you push your code.

  One option is to set them manually command line, and if you decide to try it out, you must name them as follows:
  - `Vault__Url`
  - `Vault__ClientId`
  - `Vault__ClientSecret`
  - `Vault__TenantID`
    (the double underscore __ represents colon ':' inside .NET)

You can see them being set in the **\.github\workflows\main.yml** file.

Further reading

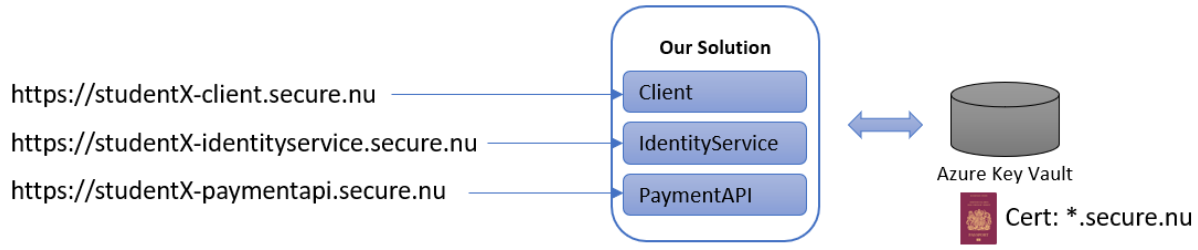
- Configuration in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration
- Use multiple environments in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments
- The danger of using config.AddAzureKeyVault() in .NET Core
  http://www.azureintegrationgurus.com/blogs/azure-integration-community/the-danger-of-using-config-addazurekeyvault-in-net-core/
- Azure Low Lands 2019 - Building secure cloud applications with Azure Key Vault
  https://www.slideshare.net/TomKerkhove/azure-low-lands-2019-building-secure-cloud-applications-with-azure-key-vault
- User Secrets, the human-readable version
  https://weblogs.asp.net/sfeldman/user-secrets-the-human-readable-version
- How can I list all of the configuration sources or properties in ASP.NET Core?
  https://stackoverflow.com/questions/37688027

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise 3 – Adding HTTPS and securing our services

Let's add HTTPS support and security headers to our application.

The goal is to have a **HTTPS endpoint** for each service with a wildcard ***.secure.nu** certificate stored in **Azure Key Vault**.



Exercise 3.1 – Adding the certificate to Key Vault

1. To support HTTPS, we have provided a wildcard ***.secure.nu** certificate that you will find in the **\Config** folder.

   This **wildcard certificate** will allow you to issue certificates for **any sub-domain** under the **secure.nu** domain.

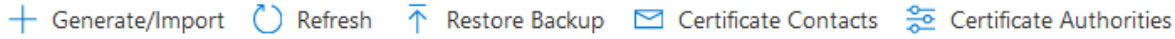
   In the **DNS** settings for **secure.nu**, we have added the following CNAME redirect mapping:

*.webapi.se domain	Domain in Azure
https://student1-client.secure.nu	student1-client.northeurope.azurecontainer.io
https://student1-identityservice.secure.nu	student1-identityservice.northeurope.azurecontainer.io
https://student1-paymentapi.secure.nu	student1-paymentapi.northeurope.azurecontainer.io

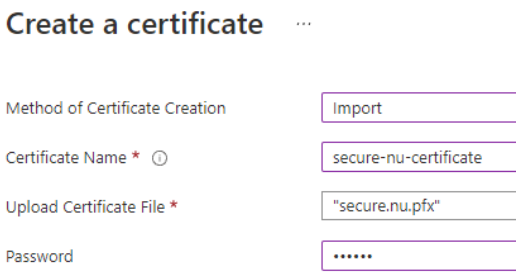
   When your browser makes a request to **student1-client.secure.nu**, the DNS system will redirect you to the **IP** that **student1-client.northeurope.azurecontainer.io** points to.

2. First, we need to import the **certificate** into **Azure Key Vault**. Open and login to the Azure Portal at https://portal.azure.com and then go to your **Azure Key Vault** instance.

   Under **Settings -> Certificates** click on **Generate/Import**.

   Select Import and name the certificate **secure-nu-certificate.** The password is **tndata**.

Exercise 3.2 – Adding HTTPS support to our application

1. To help us add the support for HTTPS to our applications, add the file **\Starter-kit\Exercise 3\HttpsExtensions.cs** to the **Infrastructure** project.

   Review the code and see what it does.

2. To call this extension method we need to do a small refactoring of the **CreateHostBuilder** method in **Program.cs**.

   Open the file **\Starter-kit\Exercise 3\Program.txt** and follow the instructions and apply the changes to all three projects.

   Review the refactoring and make sure you understand it. The core difference is that we call the **AddHttpsSupport** method in **production**. We also handle the **offline** case where we just run with the default settings.

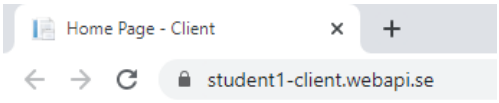
3. Run the program and everything should work as before on your machine.

4. **Push** the application to GitHub and then after it is deployed, first make sure the services work fine over HTTP.

   Then try the **HTTPS links** and see if they work:

   https - student1-client.secure.nu
   https - student1-identityservice.secure.nu
   https - student1-paymentapi.secure.nu

5. If it works, you should see that the page is secure with the **padlock**:



6. Click on the padlock and explore the certificate details.

   Notice the **short lifetime** of this certificate.

Exercise 3.3 – Verifying our HTTPS configuration

1. Just adding HTTPS is of course a good start, but how good is our HTTPS configuration?

   Let's find out!

   Visit https://www.ssllabs.com/ssltest/ and then enter one of your ***.secure.nu** domains in the form, as shown below:

   Examine the result and we typically get a **grade A** as our overall rating by default when using **ASP.NET Core 5**. If you were using **ASP.NET Core 3.x** you would get a **grade B** by default.

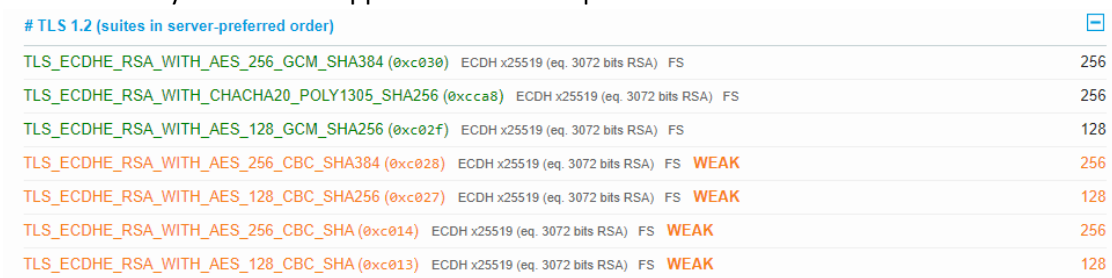
   If you need to tweak the protocol support, then you can do that in the **HttpsExtensions** class and set the **SslProtocols** using the following:

```
options.UseHttps(serverCertificate: cert, configureOptions: httpsOptions =>
{

   //Add HTTPS configuration here
   httpsOptions.SslProtocols = SslProtocols.Tls12 | SslProtocols.Tls13;
});
```

   (You don't need to do this in **.NET 5** because we are secure by default)

2. The second thing we need to look at is the supported **TLS ciphers** used.

   We see that by default we support some weak ciphers:



3. For ASP.NET Core 3.x review this GitHub issue for details about this.

   - Kestrel server does not get grade A in SSL tests because it has no cipher suit preference https://github.com/dotnet/runtime/issues/30767

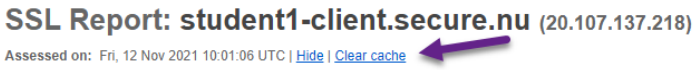
   To address this in our project, this open **\Starter-kit\Exercise 3\Cipher.txt** and follow the instructions.

   ALERT!

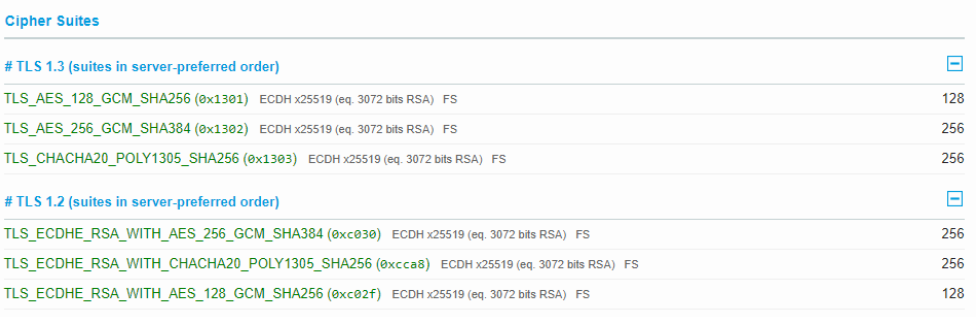
   Be aware that restricting the ciphers and protocol might result in older clients (for example on Windows XP or Windows 7) not being able to connect. Also, older browsers, like Internet Explorer 6.x, might have issues.

   In the report you will see under the **handshake simulation** which clients can't connect.

4. Push the code again and force a recheck in SSL Labs. You force a recheck by clicking on the **Clear cache** link at the top of the page:



   This time we have no weak ciphers in use:



   Great, Now HTTPS is good!

Exercise 3.4 – Securing our server

1. Now we have secured the HTTPS connection and we get good grades! Awesome!

   Let's run a scan using this tool **https://observatory.mozilla.org/** against any of our sites.

   The Observatory is an online service that can scan your website for security vulnerabilities to help you configure and secure your web application.

   As you see the result is a bit disappointing - let's fix this!

2. **Remove the server header**

Open Fiddler and navigate to our server and you will see that the server header is included in the response. This is unnecessary information to leak out:

```
HTTP/1.1 200 OK
Date: Fri, 21 Aug 2020 11:12:17 GMT
Server: Kestrel
Content-Length: 53
```

Let's remove it by adding the following to the **AddHttpsSupport** method in **HttpsExtensions.cs**:

```
webBuilder.UseKestrel((context, serverOptions) =>
            {
                 serverOptions.AddServerHeader = false;
```

3. **Redirect HTTP -> HTTPS**

We should automatically redirect all **HTTP** traffic over to **HTTPS** and we do that by adding the following statement in **Startup.cs** (in all three projects):

```
if (!env.IsProduction())
{
    app.UseDeveloperExceptionPage();
}
app.UseHttpsRedirection();
```

Add it directly after the `if (!env.IsProduction())` if-statement.

4. **Enable HSTS**

Adding the **HTTP Strict Transport Security** header informs the browser that all HTTP Requests should be upgraded to use HTTPS instead.

In **Startup.cs**, add the following in the **ConfigureServices** method:

```
services.AddHsts(opts =>
{
    opts.IncludeSubDomains = true;
    opts.MaxAge = TimeSpan. FromSeconds(15768000);
});
```

In the configure method, add it by modifying the if-statement so it becomes (in all three projects):

```
if (!env.IsProduction())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseHsts();
}
```

Read more about HSTS here:
https://developer.mozilla.org/en-US/docs/Glossary/HSTS

5. **Fix the response headers**

   Add the file **SecurityExtensions.cs** to the Infrastructure project. Review what the class does.

   Add the following after the existing **UseHttpsRedirection()** method:

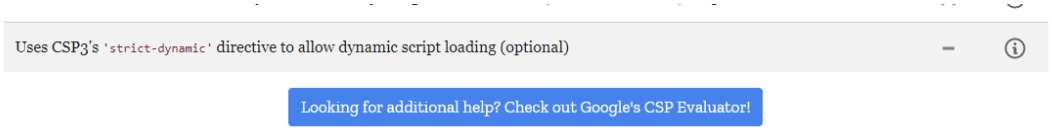
```
app.UseHttpsRedirection();
app.UseSecurityHeaders();
```

> **WARNING!**
>
> Depending on your site you might need to tweak the security headers, especially the **content security policy**, because it might break JavaScript and other things.

6. Make sure the solution compiles and then push the code to GitHub.

   After the deployment is done, do a new scan using https://observatory.mozilla.org/ and verify that we now get good grades!

7. For more detailed **Content Security Policy** checks, try to click on the **Google CSP Evaluator** button at the end of the page:



8. Additional scanners that you can try out are:

   - Webhint
     https://webhint.io/scanner/
   - Internet.nl (Test your site)
     https://en.internet.nl/
   - Security Headers
     https://securityheaders.com/

9. Congratulations! You have now set the security foundation for the rest of this course!

Future ideas:

- Should we close port 80 permanently?

- We could further restrict what hosts we accept by setting the AllowedHosts value
  https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel

  - Adding host filtering to Kestrel in ASP.NET Core
    https://andrewlock.net/adding-host-filtering-to-kestrel-in-aspnetcore/

- ASP.NET Core supports **HTTP/2**, should we support that as well?

- Could you run the site using the **HTTPS** and the ***.webapi** certificate locally? How would you go about to do that?

  Https won't work unless you locally map **web.api** to point towards **127.0.0.1** in your **hosts** file located in **C:\Windows\System32\drivers\etc**.
  For example, you could try to add this mapping entry to that file:

  ```
  127.0.0.1 secure.nu

  # Alternatively

  127.0.0.1 student1-client.secure.nu
  127.0.0.1 student1-paymentapi.secure.nu
  127.0.0.1 student1-identityservice.secure.nu
  ```

- Currently we listen on all available IP-addresses using:
  `serverOptions.Listen(IPAddress.Any, 443);`
  This could be further restricted to only accept traffic on certain addresses. Accepting traffic from all IP-addresses (0.0.0.0) is a potential issue.
  - What is the difference between 0.0.0.0, 127.0.0.1 and localhost?
    https://stackoverflow.com/questions/20778771

Further reading – Azure Key Vault and certificates

- Azure Key Vault
  https://docs.microsoft.com/en-us/azure/key-vault/
- Using certificates from Azure Key Vault in ASP.NET Core
  https://damienbod.com/2020/04/09/using-certificates-from-azure-key-vault-in-asp-net-core/
- Azure Key Vault Certificate client library for .NET
  https://azuresdkdocs.blob.core.windows.net/$web/dotnet/Azure.Security.KeyVault.Certificates/4.1.0/api/index.html
- Azure SDK for .NET
  https://azure.github.io/azure-sdk-for-net/index.html
- Azure Key Vault Certificate client library for .NET
  https://docs.microsoft.com/en-us/dotnet/api/overview/azure/security.keyvault.certificates-readme
- Azure Key Vault SDK source code
  https://github.com/Azure/azure-sdk-for-net/tree/master/sdk/keyvault
- Sample - Get the private key for Key Vault certificate
  https://github.com/heaths/azsdk-sample-getcert
- Using certificates from Azure Key Vault in ASP.NET Core
  https://damienbod.com/2020/04/09/using-certificates-from-azure-key-vault-in-asp-net-core/

Further reading – Kestrel

- Kestrel web server implementation in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel
- ASP.NET Core Web Host
  https://docs.microsoft.com/en-US/aspnet/core/fundamentals/host/web-host
- Kestrel web server implementation in ASP.NET Core
  https://docs.microsoft.com/en-US/aspnet/core/fundamentals/servers/kestrel
- Understanding .NET Generic Host Model
  https://sahansera.dev/dotnet-core-generic-host/
- Enforce HTTPS in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl
- ASP.NET Core Web Host
  https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/web-host
- How to use TLS 1.2 in ASP.NET Core 2.0
  https://karthiktechblog.com/aspnetcore/how-to-use-tls-1-2-in-asp-net-core-2-0-and-above

Further reading – Security

- How HTTPS works (in a comic format)
  https://howhttps.works/
- HTTP Strict Transport Security Protocol (HSTS)
  https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl

- How to apply various Security Feature in ASP.NET Core application https://karthiktechblog.com/aspnetcore/how-to-apply-various-security-feature-in-asp-net-core-application
- Transport Layer Security (TLS) best practices with the .NET Framework https://docs.microsoft.com/en-us/dotnet/framework/network-programming/tls
- Configuring HTTPS in ASP.NET Core across different platforms https://devblogs.microsoft.com/aspnet/configuring-https-in-asp-net-core-across-different-platforms/
- Maximum Lifespan of SSL/TLS Certificates is 398 Days Starting Today https://thehackernews.com/2020/09/ssl-tls-certificate-validity-398.html
- Enforce HTTPS in ASP.NET Core https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise 4 – Adding IdentityServer

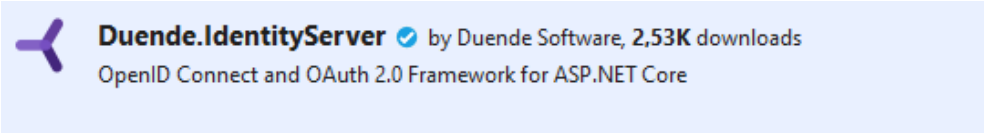
Let's focus on the IdentityService in this exercise.

Exercise 4.1 – Adding IdentityServer to our IdentityService project

1. The team behind IdentityServer provides various templates that are a great starting point for any IdentityServer project. Details about these templates can be found here: https://github.com/IdentityServer/IdentityServer4.Templates

   Because we're starting from our own starter project, we need to manually add IdentityServer.

   First add the **Duende.IdentityServer** NuGet package to the **IdentityService** project:

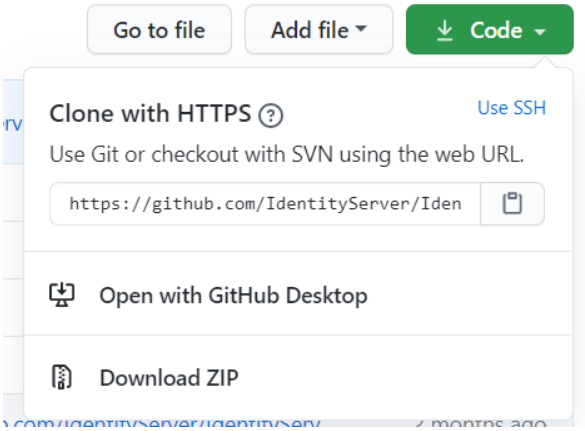
   

   (Make sure you install version 5.x.x)

2. To add the user interface, open this folder **\Starter-Kit\IdentityServer** and unzip the **IdentityServer.Templates-main.zip** file.

   Alternatively, visit https://github.com/IdentityServer/IdentityServer4.Templates

   Then click to download the repository as a **.ZIP** file.

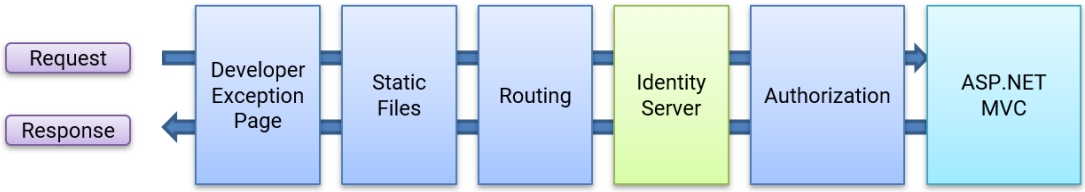
   

3. Unzip the ZIP file and go to the following folder **\src\IdentityServer4InMem**

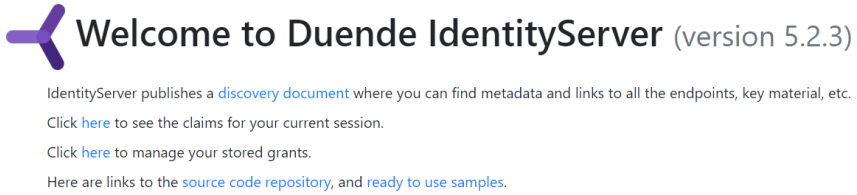
   Copy the following items into the **IdentityService** project folder:
   - Quickstart
   - Views
   - Wwwroot
   - Config.cs

4. Open the file **\Starter-kit\Exercise 4\Startup.cs** and follow the instructions in it.

   IdentityServer is provided as a **middleware** and in our application the request pipeline looks something like this:



5. The project should now compile and if you run it, you should be presented with this page:



6. Explore the various files and folders that we just added and make sure you understand what they contain.

7. The file **\Quickstart\SecurityHeadersAttribute.cs** contains the same functionality that we added in the previous module, so this file should be removed.

   Delete this file and all the **[SecurityHeaders]** attributes that references it.

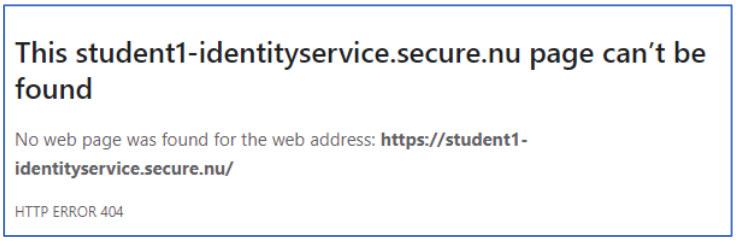
   Make sure the code still compiles.

8. Push the code to **GitHub** and then wait for it to be deployed.

9. Visit the **IdentityService** using https://student**X**-identityservice.secure.nu/

   Does that work?

   We now get a **404 Not Found** error:



   Weird, it worked locally!

10. Open the **\Quickstart\Home\HomeController** class and examine the code.

    As you see it just returns a **404** error if we are in production. This is of course a good security precaution to not identify what kind of service we are running.

    To verify that it is actually working, visit
    https://student**X**-identityservice.secure.nu/grants

    You should be presented with the IdentityServer login screen. You can try to login as username **bob** and password **bob** and that should work.

11. Showing a **404** by default is good, but at the same time, perhaps just a plain OK message could be useful, just to know that the service is running.

    Let's change so that the output in production is the following:

    ```
    Service started: 2020-10-23 16:23:12 (Production, Release build 2f08d02c)
    ```

    To do this, do the following:

    1. In the **HomeController** constructor, add and inject an instance of **IConfiguration** and save the result in a private field name **configuration**.

    2. Open the **\Starter-kit\Exercise 4\HomeController.txt** and follow the instructions.

12. Redeploy the application and verify the output when you visit the **IdentityService** homepage.

13. Visit https://student**X**-identityservice.secure.nu/.well-known/openid-configuration (replace X with your personal number)

    This page is generated by IdentityServer and contains various facts that the client can use to configure itself.

14. The JSON is a bit unreadable when it is not properly formatted, so copy the JSON document and then paste it into this page: https://jsonformatter.org/ (Then click on Beautify).

    Now you can more easily see the various endpoints and configuration provided by this service.

    Great! We now have a basic IdentityServer up and running!

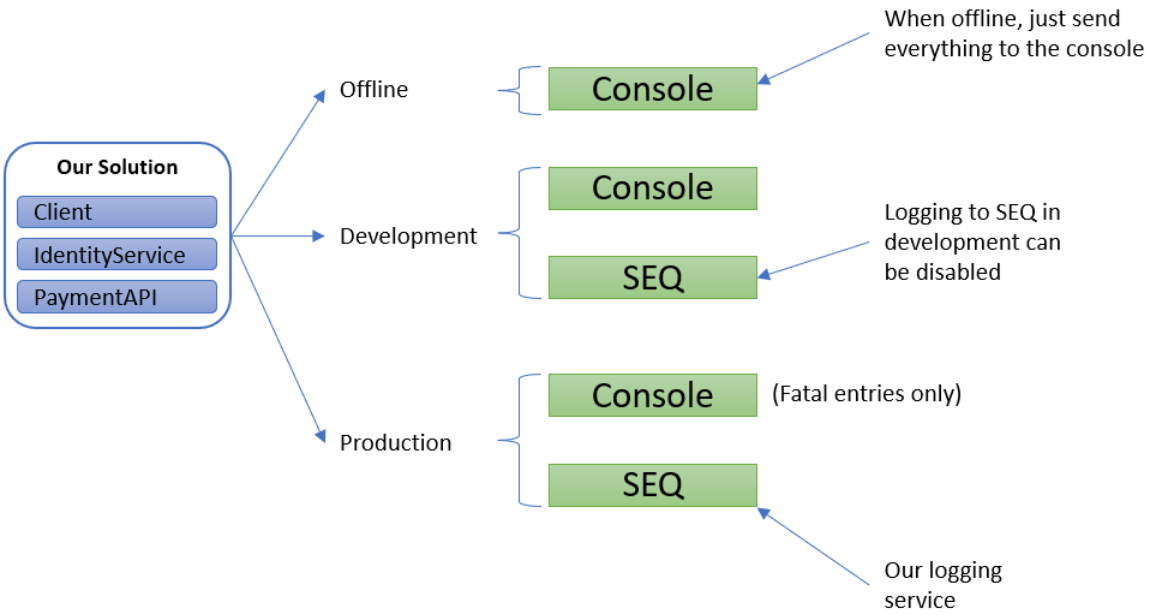
Further reading

- IdentityServer documentation
  https://docs.duendesoftware.com
- IdentityServer source code
  https://github.com/duendesoftware
- Quickstart UI for IdentityServer
  https://github.com/DuendeSoftware/IdentityServer.Quickstart.UI
- IdentityServer Templates
  https://github.com/DuendeSoftware/IdentityServer.Templates

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

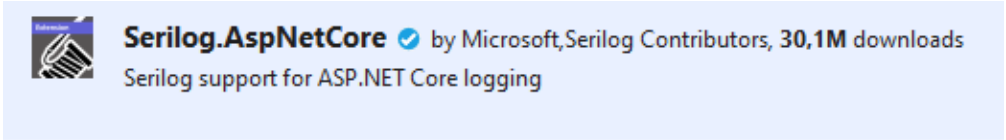
Main Exercise 5 – Adding Logging

Let's add logging to our three services and introduce **Seq** – a convenient logging service. The goal is to have a logging configuration that looks like this setup:



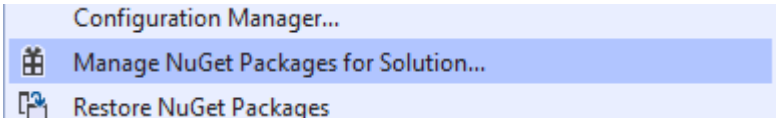
Exercise 5.1 – Adding Serilog

1. First, we need to add the **Serilog.AspNetCore** NuGet package to **all projects** in the solution. SeriLog is the most popular logging library for .NET Core.

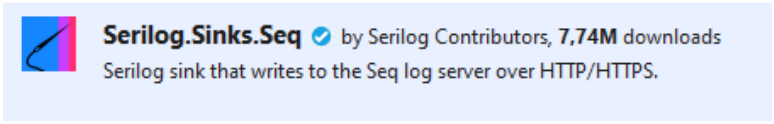


(It will also bring in the main Serilog package)

Hint: use **Manage NuGet Packages for Solution** to add it to all projects.



2. Then add the **serilog.sink.seq** NuGet Package to the **Infrastructure** project.



We will not use the default **LogLevel** configuration in **appsettings.json**, so go to all the **appsettings.json** files and remove the all the **Logging** nodes:

```
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
```

(We will configure the logging levels in code in this implementation.)

Your **appsettings.json** will probably look something like this:

```
{
  "AllowedHosts": "*"
}
```

3. Add the file **\Starter-kit\Exercise 5\LoggingExtensions.cs** to the Infrastructure project. Do review what the class do.

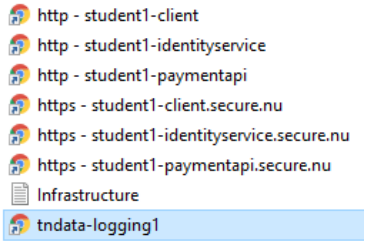
4. Open **\Starter-kit\Exercise 5\Program.txt** and follow the instructions for each of the three web projects.

5. Add the following entry to your **appsettings.json** file in all three web projects:

```
{
  "AllowedHosts": "*",
  "SeqServerUrl": "[Your seq-logging server URL]"
}
```

Do copy the seq logging url from your personal **Infrastructure.txt** file)

6. Launch **Seq** using the **tndata-logging** link or the direct URL of http://tndata-loggingX.northeurope.azurecontainer.io/



7. Start the application on your computer and then go to **Seq** and explore the incoming logs that are sent to **Seq** in real-time.

8. When it all works, feel free to comment out the **.WriteTo.Seq(…)** code block in the **LoggingExtensions** class to stop sending your local logs entries to **Seq**.

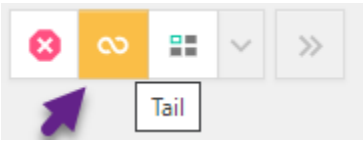
(In the default switch block)

9. The logging levels that are set in the **ConfigureLogLevels** method might need tweaking to match what your own requirements are:

```
.MinimumLevel.Override("IdentityServer4", LogEventLevel.Debug)
.MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
.MinimumLevel.Override("Microsoft.Hosting.Lifetime", LogEventLevel.Information)
.MinimumLevel.Override("System", LogEventLevel.Warning)
.MinimumLevel.Override("Microsoft.AspNetCore.Authentication", LogEventLevel.Information)
.MinimumLevel.Override("Microsoft.AspNetCore.Authorization", LogEventLevel.Information);
```

10. Push the code to **GitHub** and verify that you do see the logs from the production instances in **Seq**. Do visit the sites to trigger log entries to Seq.

11. In **Seq**, enable the **Tail** function so that the log is continuously updated with new log entries.



Exercise 5.2 – Improving the log entries

1. Right now, we see log entries from our application in **Seq** - that's great! The log entries that we see are useful, but they are a bit noisy, with sometimes multiple log entries per request.

2. To improve this, Serilog includes a **RequestLogging** middleware that will issue a log entry for each request, including information like timing and other vital details.

   Let's add the **UseSerilogRequestLogging** in each project Startup class, just before **UseHttpsRedirection**.

```
app.UseSerilogRequestLogging();

app.UseHttpsRedirection();
app.UseSecurityHeaders();
```

   (Add it to all web projects)

   See https://github.com/serilog/serilog-aspnetcore for details on how you can further customize `UseSerilogRequestLogging`.

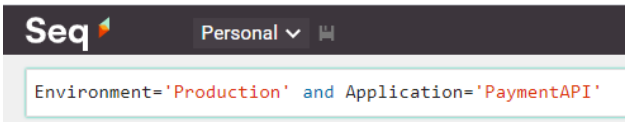
3. Start the application locally and explore the new log entries generated by this middleware. You should, for example, see entries like this:

Exercise 5.3 – Filtering in Seq

1. In **Seq** you see one long list of log entries from all applications, both in production and development.

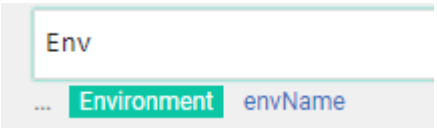
   For example, to only see entries from the **PaymentAPI** in **Production**, then you can type the following in the **Seq search bar**:

   ```
   Environment='Production' and Application='PaymentAPI'
   ```

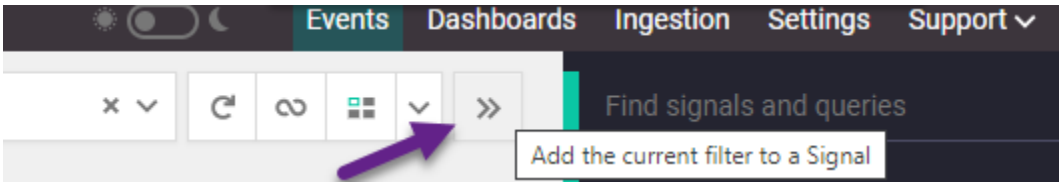
   Give it a try!

   You should also notice that you get statement completion when you start to type field name, like:

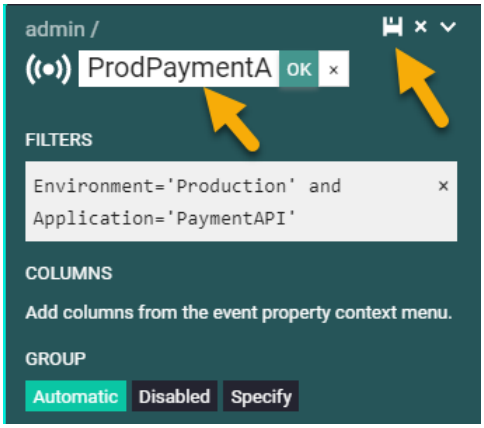
   

   Press TAB complete the name.

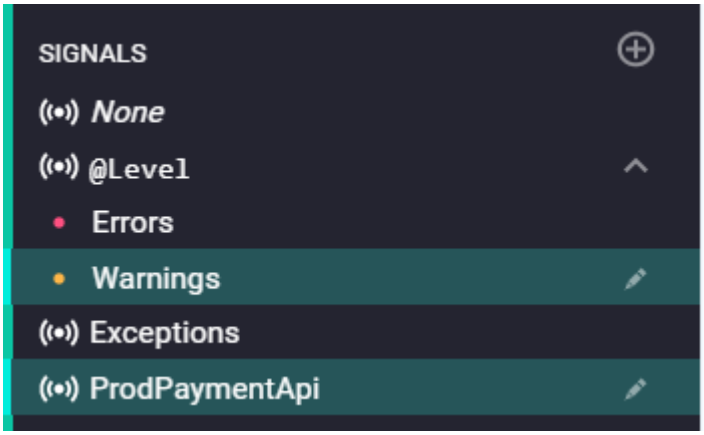
2. If you have found a filter that you want to save, then you can press the **»** button:

Then you can give it a name, like **ProdPaymentAPI** and then press the save icon:



3. Be aware that you can select multiple filters like this to select all **warnings** from the **PaymentAPI** in **production**.



Exercise 5.4 – Events in IdentityServer

1. In IdentityServer we can see the log entries from **IdentityService** and that is great! But if we open the **\QuickStart\Account\AccountController.cs** class, we see statements like these ones all over the place:

```
await _events.RaiseAsync(new UserLoginSuccessEvent(…));
await _events.RaiseAsync(new UserLoginFailureEvent(…));
await _events.RaiseAsync(new UserLogoutSuccessEvent(…));
await _events.RaiseAsync(new ConsentDeniedEvent(…));
…
```

These events represent higher level operations that occurs inside **IdentityServer** and the great thing is that we can also send these statements to **Seq**!

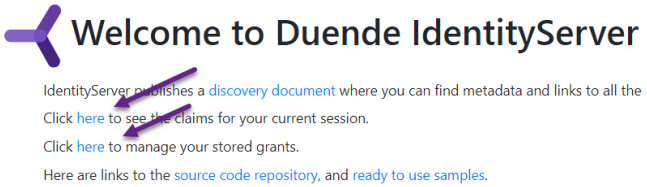
2. To enable these events, then add the following in your IdentityService **Startup** class:

```
var builder = services.AddIdentityServer(options =>
    {
        options.Events.RaiseErrorEvents = true;
        options.Events.RaiseFailureEvents = true;
        options.Events.RaiseInformationEvents = true;
        options.Events.RaiseSuccessEvents = true;
```

3. By default, all the events will now be sent to the **ASP.NET core logging** system. If you want to handle these events separately, then you need to implement the **IEventSink** interface. You can read more about the event system here: https://docs.duendesoftware.com/identityserver/v5/diagnostics/events/

   You can also create your own event types.

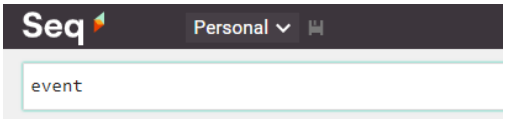
4. Start the application locally and click on one of these links in **IdentityService** to trigger a login:



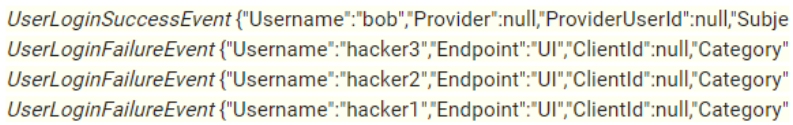
   Do a few failed login attempts and then login with **alice/alice** or **bob/bob**.

5. Now open **Seq** and to only see the events, you can search for the following in the **Seq** search toolbar:

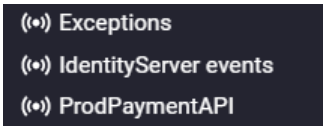
   - event



   Doing so, should show only the IdentityServer events, like:



6. Create a new signal (filter) in **Seq** named **IdentityService events** that only shows the events:



   An alternative is to make a search filter, like this one:
   `SourceContext='Duende.IdentityServer.Events.DefaultEventService'`

7. A handy **cheat sheet** for **Seq** is available here:

   Seq expression and query cheat sheet
   https://github.com/datalust/seq-cheat-sheets/releases

If you have time:

- **Secure Seq using API Keys.**
  By default Seq will accept all events sent to the ingestion endpoint. By requiring that each client provides an API key, you can get better control over the data that is sent to Seq. Each API key can be associated with a set of properties that will be applied to events written with it.
- **Explore the Seq third party applications**
  Visit https://www.nuget.org/ and search for **Tags:"seq-app"** (including the quotes) and see what applications are available.

  Interesting applications are:
  a. **Seq.App.Slack**
     An app for Seq that forwards events to Slack.
  b. **Seq.App.FirstOfType**
     Raises an event whenever it detects that a raised event type has not been seen previously.
  c. **Seq.App.Thresholds**
     Counts events in a sliding time window, writing an event back to the stream when a set threshold is reached.

Further reading

- Serilog in ASP.NET Core 3.1 – Structured Logging Made Easy
  https://www.codewithmukesh.com/blog/serilog-in-aspnet-core-3-1/
- Seq
  https://datalust.co/seq
- Seq – Signals
  https://docs.datalust.co/docs/signals
- Serilog.AspNetCore NuGet package source
  https://github.com/serilog/serilog-aspnetcore
- Using Serilog.AspNetCore in ASP.NET Core 3.0 - Part 1
  https://andrewlock.net/using-serilog-aspnetcore-in-asp-net-core-3-reducing-log-verbosity/
- Seq Cheat Sheets
  https://github.com/datalust/seq-cheat-sheets

Main Exercise 6 – Error handling

Exercise 6.1 – Error and exception handling

1. Error and exception handling are two important things that we need to configure and get control over.

   To help us keep this under control, it is a good practice to be able to trigger errors on the backend so that we can validate that the error actually works as we expect.

2. To do this we have created a few controllers for you.

   Open **\Starter-Kit\Exercise 6** and copy the **TestErrorController.cs** to the following locations:

   - **\Client\Controllers**
   - **\IdentityService\Quickstart\TestError**
     (Create the error folder)
   - **\PaymentAPI\Controllers**

3. Review what the controllers do:

   To summarize, the following URL's will now trigger various errors conditions:

   - **/testerror/error1**
     Triggers an InvalidOperationException
   - **/ testerror /error2**
     Triggers an InvalidOperationException
   - **/ testerror /error3**
     Throws an exception inside the action method
   - **/ testerror /servererror**
     Returns a 500 server error
   - **/ testerror /notfound404**
     Returns a 404 not found error
   - **/ testerror /noaccess**
     Returns a 401 unauthorized
   - **/api/error1**
     Throws InvalidOperationException
   - **/api/error2**
     Throws an exception inside the action method
   - **/api/error3**
     Returns a problem details response

4. Commit the code and push it to production.

   After it is deployed, do try some of various error URL's against one of the sites in production. All sites should behave the same.

5. Right now, all we get is this error (if we look at the response in Fiddler):

```
HTTP/1.1 500 Internal Server Error
Date: Mon, 26 Oct 2020 12:16:37 GMT
Content-Length: 0
```

Let's improve this!

6. Add the **UseExceptionHandler** as shown below to all **three Startup** classes:

```
if (!env.IsProduction())
{
     app.UseDeveloperExceptionPage();
}
else
{
     app.UseHsts();
     app.UseExceptionHandler("/Home/Error");
}
```

This middleware will catch exceptions further down the pipeline, log them, and re-execute the request to **/Home/error** instead.

Let's look at the different projects and how the **/Home/Error** is implemented:

- **Client**
  The default MVC template that this project is based on, already contains an action for **/Home/Error** and it will return the **Error** view found in **\Views\Shared**

  It's up to you to customize this view. So, we don't need to do anything here.

- **IdentityService**
  The default IdentityServer **QuickStart UI** also includes a similar predefined **/Home/Error** action. Do explore what the action and view contain.

- **PaymentAPI**
  This is an API, and not a website. So, the error that we should return here should probably be a more API-friendly response instead o a HTML page.

  But we have no **\Home\Error** controller here, so let's fix that.

7. First, copy the **Error action** from the **Client\HomeController** and paste it into the **PaymentAPI\HomeController**.

8. Then copy the **Client\Models** folder with the **ErrorViewModel** class in it, to the **Payment** Project. Do fix the name space to namespace **PaymentAPI.Models**.

   Copy the **Client\Views\Shared\Error.cshtml** to the **PaymentAPI\Views\Home** folder.

9. Fix the namespace at the top of the **\PaymentAPI\Views\Home\Error** view. Simplest is to add the fully qualified name to the type:
```
@model PaymentAPI.Models.ErrorViewModel
```

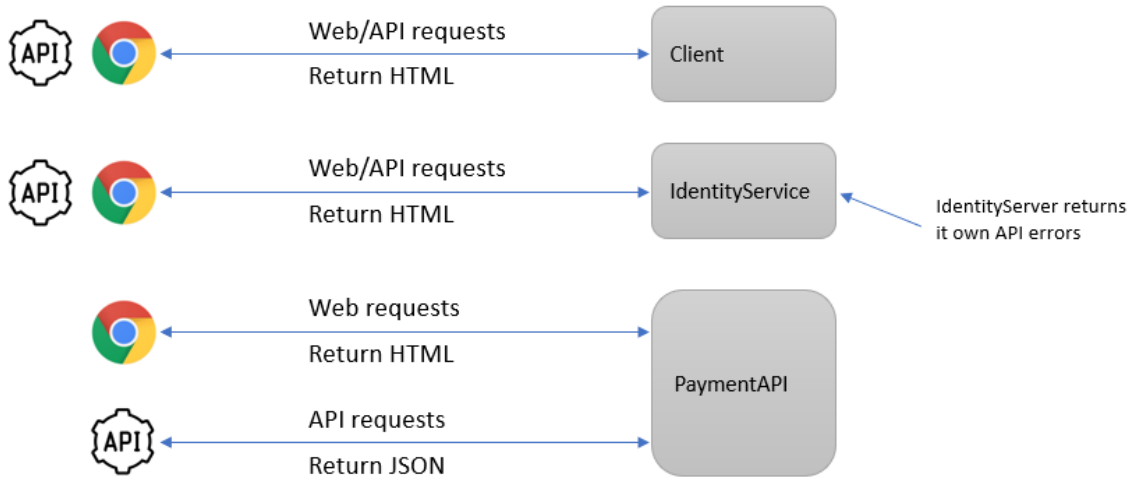
10. Fix any missing using statements and make sure the code compiles.

11. Great! But we have a problem here!

    Right now, the **PaymentAPI** will return the error view for all requests! Is that good?

    Do API requests (using JSON/XML) really want HTML back? Probably not, we typically want to return a JSON error for API requests and HTML for web-requests.

    The desired setup will then be as follows:



    Let's modify the **Error** action in the **Payment.HomeController** class to achieve this:

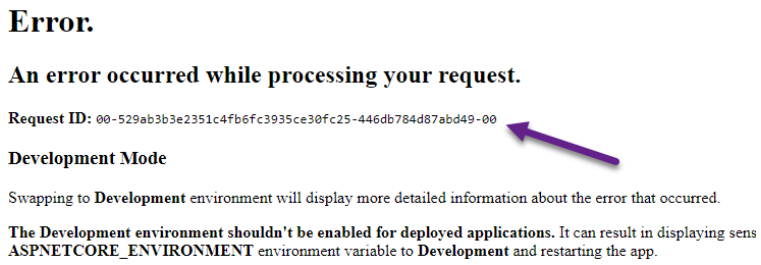
```
public IActionResult Error()
{
    if (Request.Headers["accept"] == "application/json")
    {
        //Return JSON response
        return Problem(title: "Server error",
                       detail: "Please contact support");
    }
    else
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
                    HttpContext.TraceIdentifier });
    }
}
```

    (The code is found in Starter-kit\Exercise 6\PaymentErrorAction.txt)

    The **Problem(…)** method returns an API friendly response using the **ProblemDetails** standard as defined here: https://tools.ietf.org/html/rfc7807

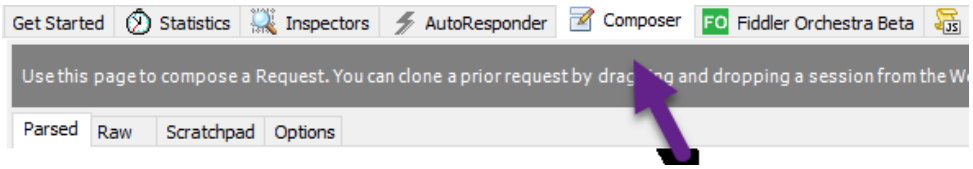
12. Do push the code to GitHub and let's verify that this works for the **PaymentAPI**.

- **Web requests**
  Visit https://studentX-paymentapi.secure.nu/testerror/error1 in your browser and review the response. As the view does not have any layout, it is just a plain text page.

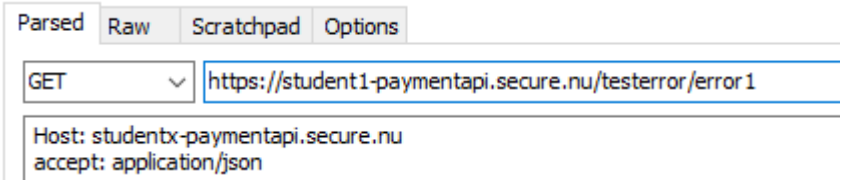


  Do notice the RequestId that is unique for each request.

- **API Requests**
  Open Fiddler and click on the **Composer** tab:



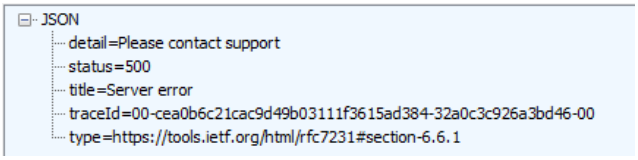
  Then enter the **URL** to your API and add the **Accept** header as shown below:



  Click on the **Execute** button and now you should see the **problem detail** response returned instead:

  This is clearly a more API friendly response!

  Do also view the response using the JSON view



13. To wrap this up, do add the **[AllowAnonymous]** attribute to the **/Home/Error** action in all your web projects. Just to make sure that non-authenticated users can view this page as well.

Future to-do:

- Add support for custom error pages for the different http error codes, like **401,403, 404,500,503** using the **UseStatusCodePages** middleware to do this.
- Should we make the **/TestError** controller a bit more secure?
- Should we add some extra log entries if we reach the **/Home/Error** page?
- You can add this code to get details about the exception inside the **Error** action method:

```
public IActionResult Error()
{
    var error = HttpContext.Features.Get<IExceptionHandlerPathFeature>();
    ...
}
```

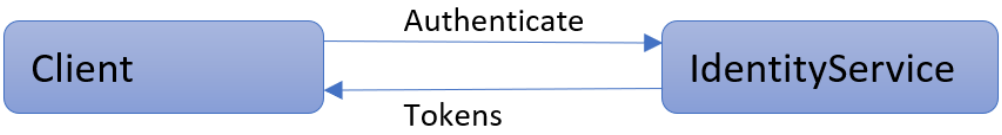
Further reading

- Handle errors in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling
- Handle errors in ASP.NET Core web APIs
  https://docs.microsoft.com/en-us/aspnet/core/web-api/handle-errors
- Problem Details for HTTP APIs
  https://tools.ietf.org/html/rfc7807
- Global Error Handling in ASP.NET Core MVC
  https://chrissainty.com/global-error-handling-aspnet-core-mvc/

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise 7 – Securing the client

Let's focus and add initial support for **OpenIDConnect** to our **Client** application and try to successfully authenticate against our **IdentityService**.



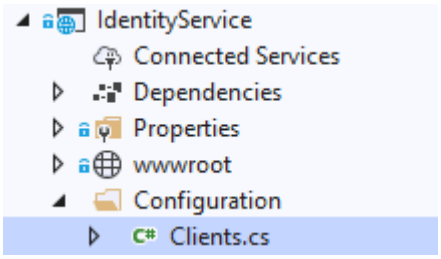
Exercise 7.1 – Defining the client in IdentityService

1. First, we need to define the client in **IdentityService** and by default the clients are defined in the **Config** class.

   Open the **\IdentityService\Config.cs** class and by default there are two clients predefined. Delete the entire **Clients** definition, including the **Clients** method:

   ```
   //Delete
   public static IEnumerable<Client> Clients =>
   …
   …
   …
   ```

2. Create a new folder in the **IdentityService** project named **Configuration** and add the class **\Starter-kit\Exercise 7\Clients.cs** to this newly created folder.

3. To get the code to compile, open **Startup.cs** and replace

   ```
   .AddInMemoryClients(Config.Clients);
   ```

   with:

   ```
   .AddInMemoryClients(Clients.GetClients());
   ```

   The code should now compile.

   Do also review what the client configuration contains.

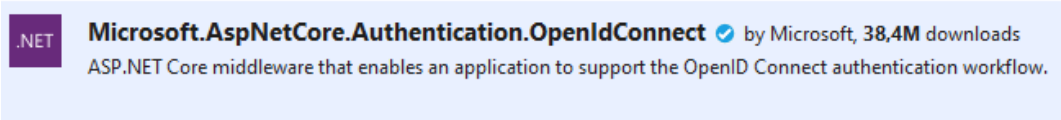
4. In **Config.cs** update the **IdentityResources** so that the client is also allowed to request the **Email** scope. (Add the Email scope)

Exercise 7.2 – Preparing for OpenIDConnect

Now let's configure the **Client** project to authenticate against our **IdentityServer**.

1. Add the following **NuGet** packages to the **Client** project:

   Microsoft.AspNetCore.Authentication.OpenIdConnect

   (important, use version 5.x.x because we are running on .NET 5)

   IdentityModel

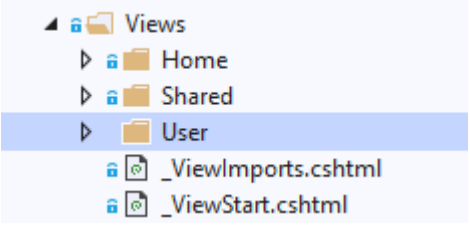
   

2. To support authentication in the client, we need to add the following **endpoints/pages**:

`/User/Login`	Triggers a challenge of the current user
`/User/Logout`	Triggers a sign-out of the current user
`/Users/Info`	Displays various details about the current logged in user
`/User/AccessDenied`	Shows the access denied page
`/User/Register`	Shows a future "signup" page

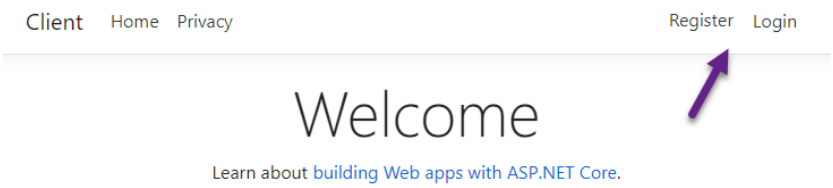
   To get started here, add the **\Starter-kit\Exercise 7\UserController.cs** to the client **Controllers** folder and review what it contains.

   Also add the entire folder in the **\Starter-kit\Exercise 7\User** to the client **\Views** folder:

   Review the code/views and make sure you understand what they do.

3. To further integrate it in the application we want to add an area at the top right with the **Login/Register** links.



To get it into the page layout file, do the following:

1. Copy the file **\Starter-kit\Exercise 7_LoginPartial.cshtml** to the **\Views\Shared** folder.

2. Open the file **\Views\Shared_Layout.cshtml** and replace the **<header>...</header>** tag with the content found in the **\Starter-kit\Layout.txt** file.

3. Review the content we just added and make sure you understand what it does.

   In particular, look at how we link to the **Login** and **Logout** pages in the **_LoginPartial.cshtml** file:

   For security reasons we require that you do a **HTTP POST** and provide an **antiforgery** token to trigger the login and logout action method.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task Login() { ... }
```

   For details, see:

   - Should login and logout action have CSRF protection? https://security.stackexchange.com/questions/62769

Exercise 7.3 –Configuring OpenIDConnect

1. Add the **authentication** middleware to the **Startup.Configure** method:

```
app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints => ...
```

(Should always be placed before UseAuthorization);

2. Review the presentation and add the necessary code to successfully authenticate against IdentityServer.

   Make sure you set these properties:

   Authentication
   Set the default scheme to use **Cookie** and **OpenIDConnect** for the challenge.

   Cookie handler

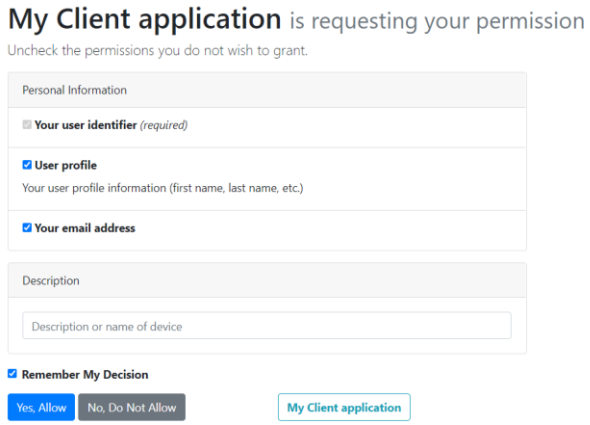
LogoutPath	/User/Logout
AccessDeniedPath	/User/AccessDenied

   OpenIDConnect handler

Authority	https://localhost:6001
clientId	See the client definition in IdentityServer **Clients** class
ClientSecret	See the client definition in IdentityServer **Clients** class
Scope	openid, profile and email
SaveTokens	true
GetClaimsFromUserInfoEndpoint	true
Prompt	consent

3. Start **Fiddler** and then start the applications and try to login in the **Client** as bob (username **bob**, password **bob**).

   You should be able to **login** and get to the **consent** screen:



   But after you give consent to the scopes, you end up with an empty page.

   Can you try to figure out why it does not work without reading ahead?

   Check in Fiddler and see what the last request is all about and what that response tries to do.

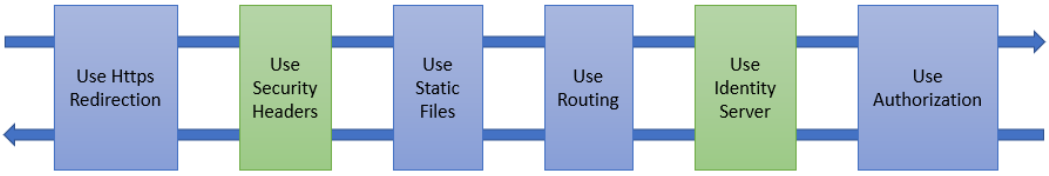
   Why does the JavaScript in the response not work?

4. Switch to the **client browser** window and press **F12**.

   In the browser **console** you should see this error:

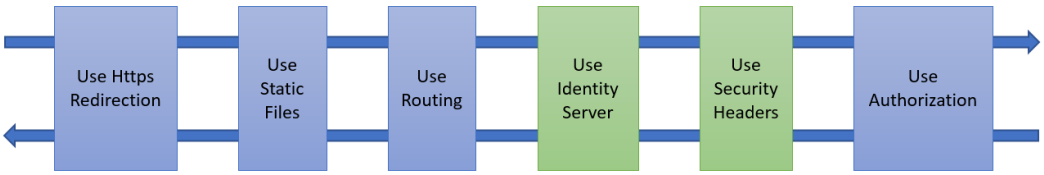
   > Refused to execute inline script because it violates the following Content Security Policy directive: "script-src 'self'". Either the 'unsafe-inline' keyword, a hash ('sha256-orD0/VhH8hLqrLxKHD/HUEMdwqX6/0ve7c5hspX5VJ8='), or a nonce ('nonce-...') is required to enable inline execution.

5. The problem is that our request pipeline currently looks something like this:



   The issue is that for some requests **IdentityServer** wants to set its own **C**ontent **S**ecurity **P**olicy header. This **CSP-policy** is never set because we have already added one earlier in the pipeline.

   To fix this, move the **UseSecurityHeaders** to after the **UseIdentityServer** middleware.



   Try it again!

   If you get this error:

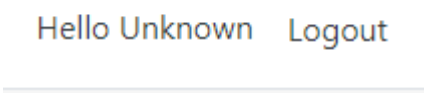
   > This localhost page can't be found
   >
   > No webpage was found for the web address: **https://localhost:5001/signin-oidc**
   >
   > HTTP ERROR 404

   then you have forgotten to add the **Authentication** middleware. Place it before the Authorization middleware.

6. Verify that you now can login in the client application:
   - User can **login** to the site (using bob/bob)
   - User can **logout** (it's ok if the logout stops at IdentityServer, we deal with signout in a later module)

- As a non-logged in user, trying to visit **/User/info** page should redirect to IdentityServer.
- Clicking on the username takes you to the **/User/Info** page

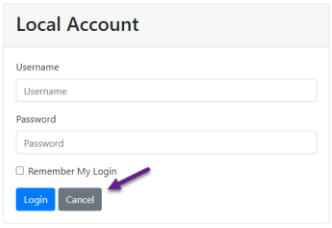
7. If the name of the user is Unknown like this:



Then you need to see the **presentation** about how to remap the **Name claim** type and set it in the OpenID connect options.

Exercise 7.4 – Access denied

1. Try to login, but on the consent screen, click on the **Cancel** button

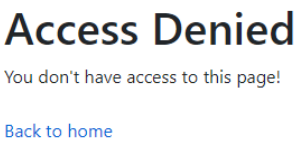


You will probably get an exception from the authentication handler in your client.

2. To fix this you need to set the following **OpenIDConnect** option:

```
options.AccessDeniedPath = "/User/AccessDenied";
```

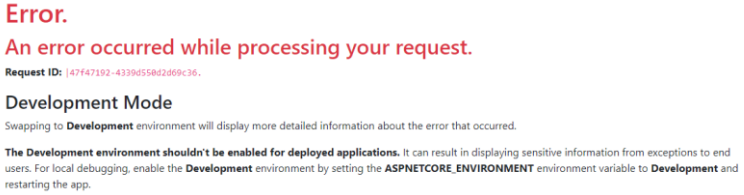
Try again and now you should be presented with the following after you cancel:



Exercise 7.5 – Deploying the project

1. Push the code to **GitHub** to deploy the project and then try to login to the client.

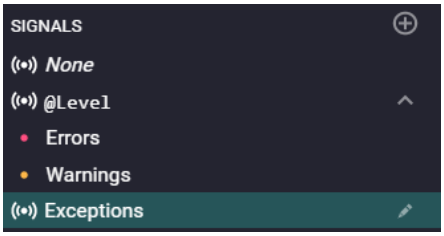
If you try to login, you are presented with the following exception:



2. So, how do we troubleshoot this error?

   Fortunately, we have logging in place. Open **Seq** and try to find the log entry for the exception.

   Click on the Exceptions signals filter in the right column to only show the exceptions:



3. The exception that you see should be:

```
System.InvalidOperationException: IDX20803: Unable to obtain configuration from:
'[PII is hidden. For more details, see https://aka.ms/IdentityModel/PII.]'. --->
System.IO.IOException: IDX20804: Unable to retrieve document from: '[PII is
hidden. For more details, see https://aka.ms/IdentityModel/PII.]'. --->
System.Net.Http.HttpRequestException: Connection refused
```

4. The problem is that the client can't download the configuration from IdentityServer from the **https://localhost:6001/.well-known/openid-configuration** URL.

   Clearly, **localhost** does not work in production, so we are faced with a few issues here:

   1. The **Client** application needs to point to our **production** IdentityServer when running in production.
   2. We need a separate **Client definition** in IdentityServer for production, as things like the **RedirectUrl** will differ.

   Let's fix this!

5. The easiest way to fix this is to have one client definition for **development** and one client definition for **production** in IdentityServer.

   1. Duplicate the client definition in **\IdentityServer\Configuration\Client.cs**
   2. Name the client variables **clientDev**, and **clientProd**

3. Add them to the client list at the end of the method:

```
return new List<Client>()
{
    clientDev,
    clientProd
};
```

4. Rename the **clientId** field in the two clients as follows:

clientDev	authcodeflowclient_dev
clientProd	authcodeflowclient_prod

5. In **clientProd** client definition, replace all the **https://localhost:5001** URLs so that they point to your production client instead. (https://studentX-client.secure.nu)

   Also be sure to remove any port numbers, like 5001. We use the standard ports in production.

   Now IdentityServer is fixed!

6. To fix the **client** we need to do the following:

   1. Depending on the environment, we need to provide different values for these two parameters in the OpenIDConnect configuration:

```
options.Authority = "https://localhost:6001";
options.ClientId = "authcodeflowclient";
```

      The easiest way to fix this for now is to put these configuration values in your **appsetting** files. You could also store them in **Azure Key Vault** for improving the security further.

   2. In your Client **Startup.cs** replace the options above with:

```
options.Authority = _configuration["openid:authority"];
options.ClientId = _configuration["openid:clientid"];
```

   3. In your **appsettings.Development.json** add the following entry:

```
{
  "openid": {
    "clientid": "authcodeflowclient_dev",
    "authority": "https://localhost:6001"
  }
}
```

   4. In your **appsettings.Offline.json** file add the same as above.

5. In **appsettings.Production.json** copy the above and change the values to use the production values instead. (Both **clientid** and **authority**).

7. Start the application locally and you should be able to login as usual.

   Push the code to **GitHub** and try to login in production. This should work as well.

   Make sure that when you login on your machine in **development** that you end up on your local IdentityService at https://localhost:6001/ and in **production** that you end up on your production instance of the same service.

Things to do and explore if you have time:

- Do make sure the client secret used is something stronger than the one used in this example.
- Where should we store the **client secret**? In Azure Key Vault? User Secrets locally?
  - Client IP-address restrictions
    In the client definition in IdentityServer you can set the **IdentityProviderRestrictions** parameter to restrict what IP-addresses that are allowed for a given client. It's defined as:

```
/// <summary>
/// Specifies which external IPs can be used with this client (if list is
empty all IPs are allowed). Defaults to empty.
/// </summary>
public ICollection<string> IdentityProviderRestrictions { get; set; } =
new HashSet<string>();
```

Further reading

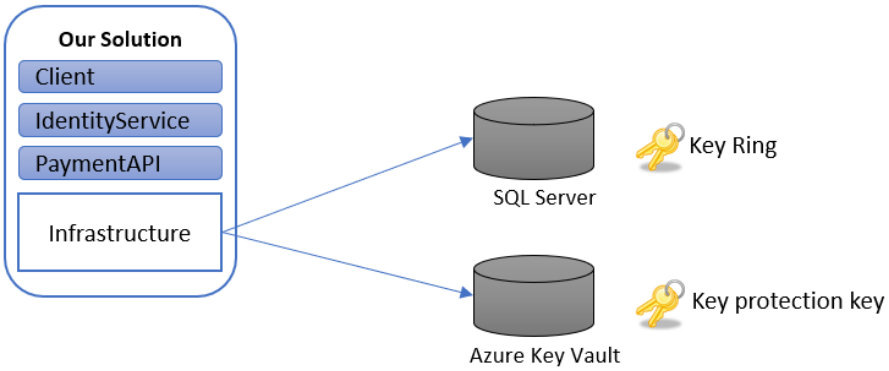
- ASP.NET Core Identity source code
  https://github.com/dotnet/aspnetcore/tree/master/src/Identity
- Interactive Applications with ASP.NET Core
  https://identityserver4.readthedocs.io/en/latest/quickstarts/2_interactive_aspnetcore.html
- Cookies, tokens and session lifetime with Identity Server
  https://vxcompany.com/2018/12/13/cookies-tokens-and-session-lifetime-with-identity-server/
- Use cookie authentication without ASP.NET Core Identity
  https://docs.microsoft.com/en-us/aspnet/core/security/authentication/cookie

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise - 8 – Configuring ASP.NET Core Data Protection

The goal in this exercise is to configure the **Data Protection API** to store the **key ring** in a database and protect each key in it using a crypto key stored in **Azure Key Vault**.

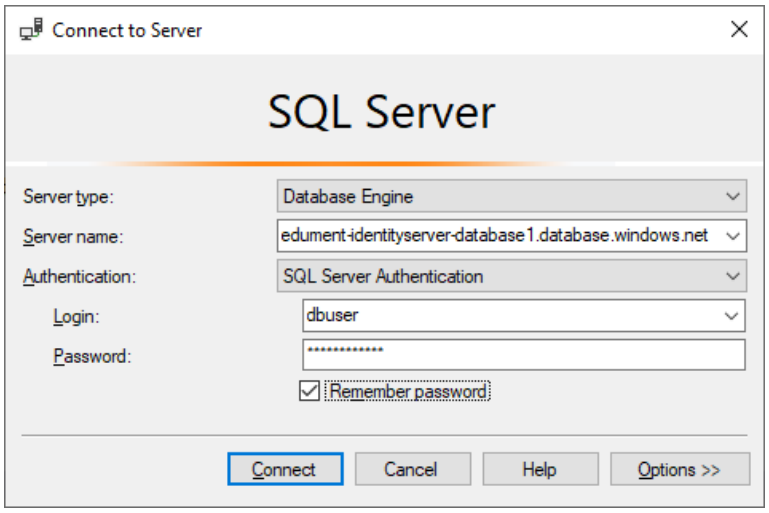
The setup will look like this:



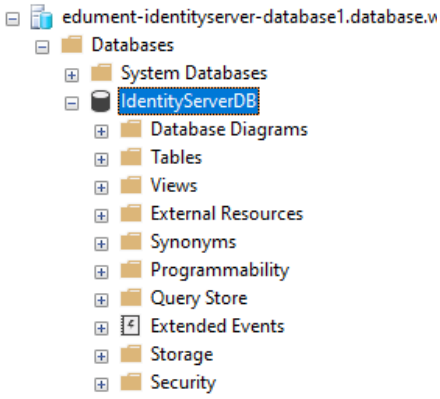
Exercise 8.1 – The database

1. We will use an **Azure SQL Server** database and the login details is detailed in your personal **Infrastructure.txt** file. Do open the file and locate your database details.

2. Open **SQL Server Management Studio** and connect to your database server instance.

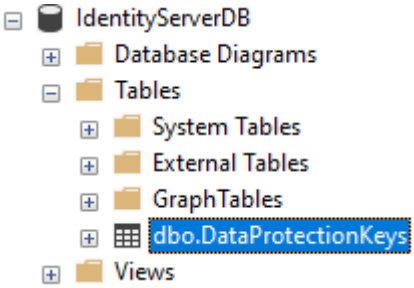


Once connected, you will find one empty database named **IdentityServerDB**



3. Now we need to create a **table** for the **key ring** and we could of course use the Entity Framework migrations to create it for us. But, because it is just one table, its simpler to just create it manually.

4. Open **\Starter-kit\Exercise 8** and double-click on the **SqlScript.sql** to open it in **SQL Server Management Studio**. Run the script to create the table:

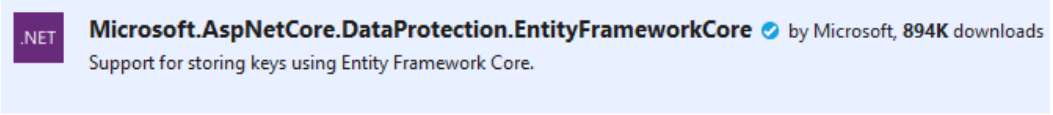


Exercise 8.2 – Adding the Data Protection code

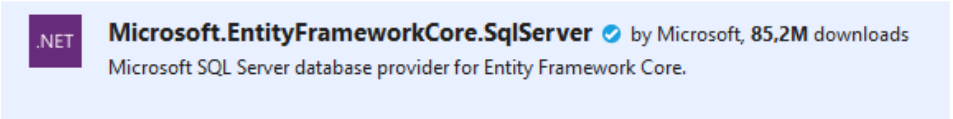
1. Add the following NuGet package to the **Infrastructure** project:

!!! Important, do install version 5.x.x of the packages!!!

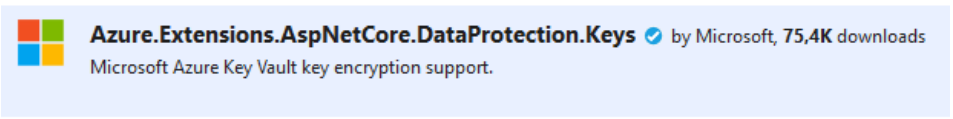
- **Microsoft.AspNetCore.DataProtection.EntityFrameworkCore**



- **Microsoft.EntityFrameworkCore.SqlServer**



- **Azure.Extensions.AspNetCore.DataProtection.Keys**



2. Create a folder named **DataProtection** in the **Infrastructure** project

   Then open **\Starter-kit\Exercise 8** and add the following files to the newly created folder:

3. Open **\Starter-kit\Exercise 8\Startup.txt** and add the content to **all three startup** classes in the solution.

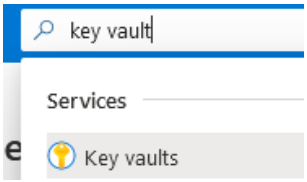
   Make sure you add it <u>at the top</u> of the method!

4. Make sure it builds.

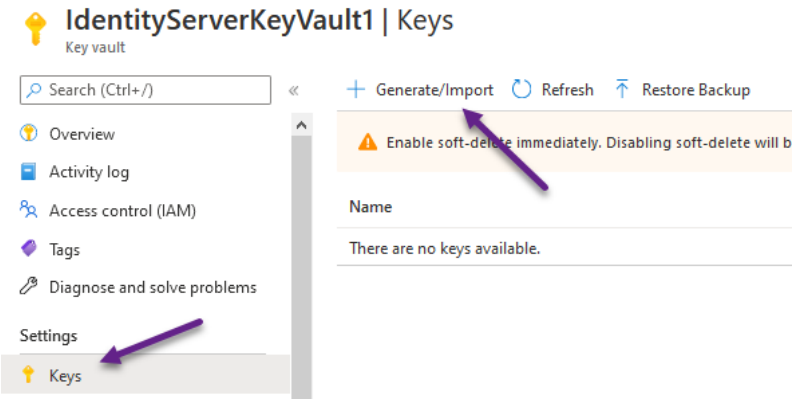
Exercise 8.3 – The key protection key

1. We are using an **RSA-key** in Azure Key Vault to protect the keys in the database. If the database is lost/hacked, then the keys stored there can't be used. This is a good security practice to encrypt sensitive material in the database. (We will learn about RSA-keys in the next module).

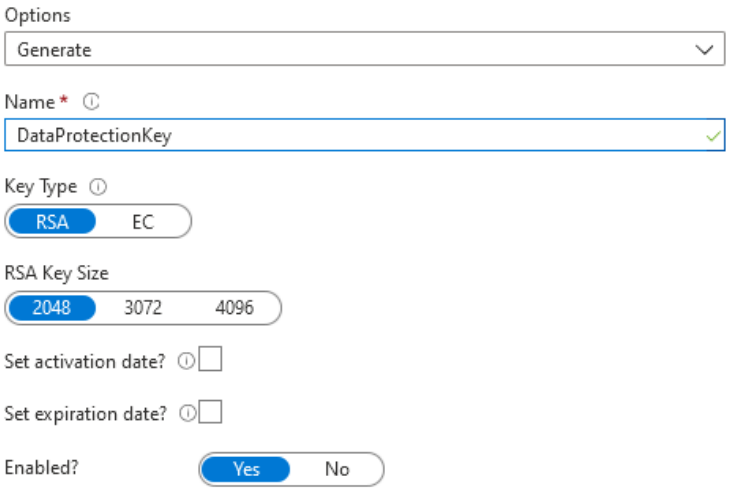
2. Login to the **Azure Portal** and then locate your **Azure Key Vault** service. The easiest is to search for **key vault**:



3. Select **Keys** and then click on **Generate/Import**



4. Then create a key named **DataProtectionKey**:



5. After the key is created, click on the key (and its current version). Then copy the **key identifier URL** (without the version number) and store it somewhere:

6. We will store the URL to this key in **appsettings.json.**

   In all **appsettings.json**, add the following entry:

```
  "ConnectionStrings": {
    "ConnectionString": "[Your database connection string]" },
  "DataProtection": {
    "EncryptionKeyUrl": "[The Key identifier Url from the previous step]"
  }
```

- The exact string is found in **\Starter-kit\Exercise 8\Appsettings.txt**
- The full connection string is found in your **Infrastructure.txt** file
- Don't include the **version number in the vault url**

Your **appsettings.json** file should look something like this:

```
{
  "AllowedHosts": "*",
  "SeqServerUrl": "http://tndata-logging1.northeurope.azurecontainer.io",
  "ConnectionStrings": {
    "ConnectionString": "Server=tndata-identityserver-database1.database.windows.net;Database=Ide
  },
  "DataProtection": {
    "EncryptionKeyUrl": "https://identityserverkeyvault1.vault.azure.net/keys/DataProtectionKey"
  }
}
```

7. Start the projects locally and make sure that it all works, and you don't get any weird exceptions in the console windows.

   In the logs you should now see various entries related to the **Data Protection** services.

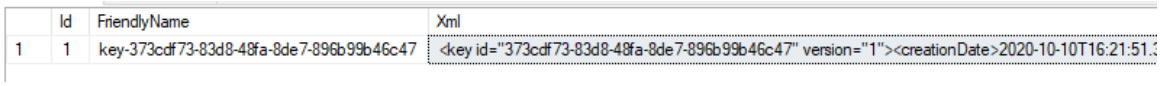
8. Push the code to production and wait for it to deploy.

   In **Seq**, you should also see the corresponding **Data Protection** entries from each service.

Exercise 8.4 – Viewing your key ring

1. Open your **SQL Management Studio** and query the **DataProtectionKeys** table and look at the entry in the table.

2. Then copy the XML value and paste it into http://www.xmlformatter.net/ or http://prettyprint.io/ to make it more readable:



	Id	FriendlyName	Xml
1	1	key-373cdf73-83d8-48fa-8de7-896b99b46c47	<key id="373cdf73-83d8-48fa-8de7-896b99b46c47" version="1"><creationDate>2020-10-10T16:21:51.3

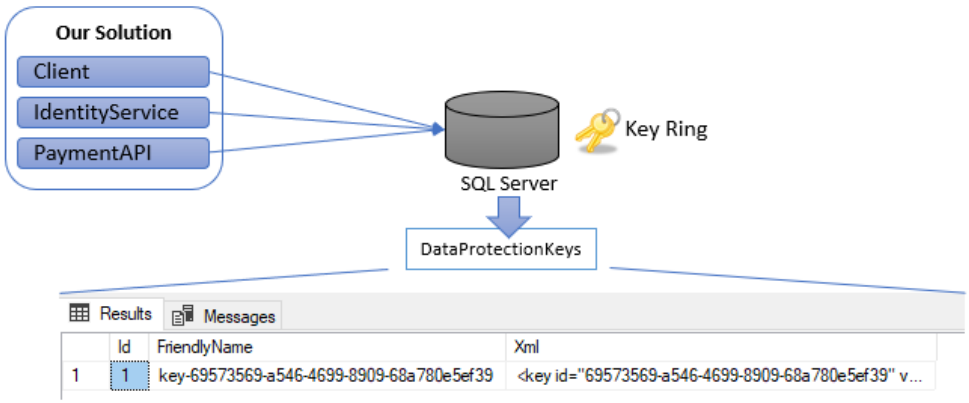
3. Is this not a security issue to paste the key ring here?

   No, the keys are encrypted using the RSA-key located in your Azure Key Vault. The key ring is unusable for anyone without that key. You can see that it is encrypted because inside the key ring it says:

```
<!-- This key is encrypted with Azure KeyVault. -->
```

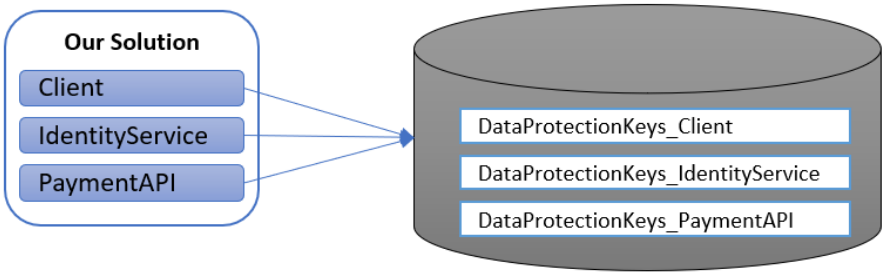
Exercise 8.5 – Problems!

1. We have a problem! What we have done above will probably work, but we will end up with various race conditions because now we have three services all reading and writing to one or multiple key-rings in the **DataProtectionKeys** table.



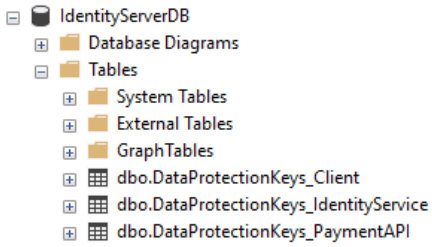
   What we need to do is to keep them apart and unfortunately, **PersistKeysToDbContext** is hardcoded to only write to this specific table and it always uses the first entry found in the table.

2. The solution that we will do next is to create **one table for each application**, like:



3. First, we need to create the new database tables:
   a. Run the SQL-script in **\Starter-Kit\Exercise 8\Exercise 8.5** that will create the new tables
   b. Delete the existing **DataProtectionKeys** table in the database

You should now have the following tables in your database:



4. Replace the **DataProtectionContext.cs** that we added earlier with the one found in **\Starter-Kit\Exercise 8\Exercise 8.5**

   It contains one context for each application. In the **OnModelCreating** method we added code to change the name of the table.

5. Replace the **DataProtectionExtensions.cs** that we added earlier with the one found in **\Starter-Kit\Exercise 8\Exercise 8.5**

   It now contains one **extension method** for each service, and we added some refactoring to make the code clean and more production ready.
   - AddDataProtectionWithSqlServerForClient
   - AddDataProtectionWithSqlServerForIdentityService
   - AddDataProtectionWithSqlServerForPaymentApi

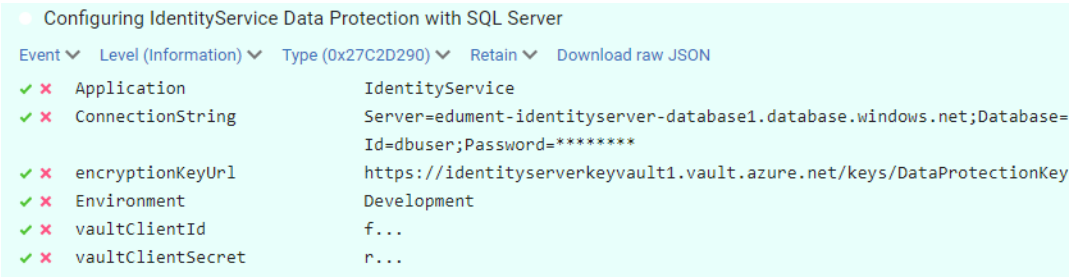
6. Go to each **Startup** class so that each class calls the correct extension method. Make sure the code compiles.

7. Now run the application locally and you should now see that a new **key ring** has been created in each database table.

8. Deploy the code to **production** and make sure everything works as expect and that you have no weird exceptions in Seq.

9. In **Seq**, search for "**sql**" and click on one of the "Configuring XXX Data Protection with SQL Server" entries, like this one:

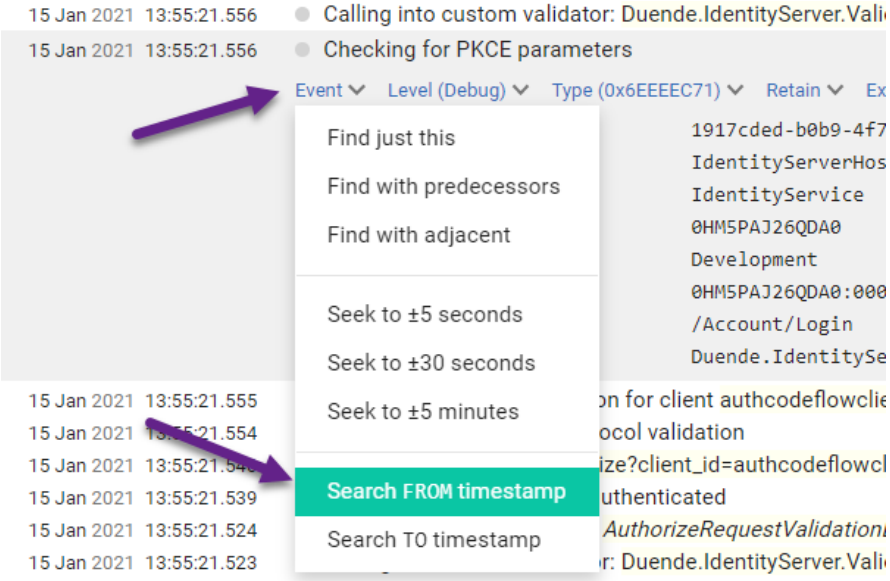
As you see, the **Data Protection API** configuration details are included as extra properties in the log entry. Some of the fields are masked to prevent secrets from leaking out.



Do review in the code how this is done and how we added these extra properties to this log entry. (Hint, look at the **AddLogEntry** method)

10. A useful hint when you use **Seq** is that it is easy to just show all events after a given event.

    You can do that easily by choosing the **Search FROM timetamp** feature, give it a try!



If you have time:

- Right now, we share the same key ring between development and production, this should of course be separated. In theory we can have a race condition here as well (when the Data Protection system issues new keys).

- Did you know that you can store the key ring in Azure Key Vault?

  Free to explore our own implementation that you can read more about here and that is pretty easy to plug into this solution:

  AzureKeyVaultKeyRingRepository
  https://github.com/edumentab/AzureKeyVaultKeyRingRepository

- Should we hardcode in application that we can only connect to local databases when we are in development? Just to make sure we never can connect to production databases during development?

  This is an interesting read:

  - We deleted the production database by accident ✹ https://keepthescore.co/blog/posts/deleting_the_production_database/

Further reading

- Configure ASP.NET Core Data Protection https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/overview
- Data Protection key management and lifetime in ASP.NET Core https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/default-settings
- ProtectKeysWithAzureKeyVault deserves more explanation https://github.com/dotnet/AspNetCore.Docs/issues/16422
- Storing the ASP.NET Core Data Protection Key Ring in Azure Key Vault https://www.edument.se/en/blog/post/storing-the-asp-net-core-data-protection-key-ring-in-azure-key-vault
- An introduction to the Data Protection system in ASP.NET Core https://andrewlock.net/an-introduction-to-the-data-protection-system-in-asp-net-core/

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise -11 - Adding signing keys

Exercise 11.1 – The problem with the Developer Signing Credential

1. In our **IdentityServer Startup** class we use this method call to automatically create a token signing key for us:

```
builder.AddDeveloperSigningCredential();
```

This works great in **development**, but less so in **production**.

Why is this a problem?

Let's explore the problem.

2. When you call the method above, a file named **tempkey.jwk** is created in the **IdentityService** project.

Open this file and explore the content.

3. To better visualize the content of this file, copy the JSON content and paste it into https://jsonprettyprint.org/

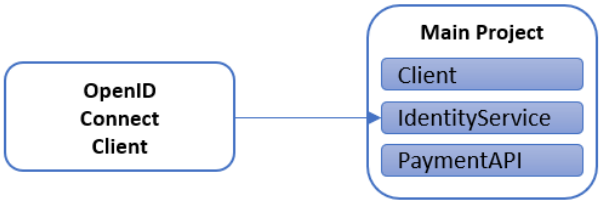
Now you see a private RSA key described using its parameters:

Parameter	Description
alg	Algorithm intended for use with the key
d, dp, dq, e, n, p, q, qi	RSA key parameters
kid	Key Identifier (Key ID)
kty	Key type

The JWK format is specified in RFC 7517 (JSON Web Key) https://tools.ietf.org/html/rfc7517.

4. Navigate to the **Starter-Kit\Exercise 11\OpenIdConnectClient** folder and open the project in a new instance of Visual Studio.

   This project contains a minimal **test client** that we will use to authenticate against our local **IdentityService**.



5. This client runs on port **5002** and we need to make a tweak to our client definition so that it also accepts the redirect URL to this port:

   Open the IdentityService **Configuration\Clients.cs** file and modify the **RedirectUris** for the **authcodeflowclient_dev** client, so it becomes:

```
RedirectUris =
{
    "https://localhost:5001/signin-oidc",
    "https://localhost:5002/signin-oidc"
},
```

6. Start **Fiddler**, then start the **main project** and then the **OpenIDConnectClient**.

7. In the **OpenIDConnectClient** do the following:

   a. Login using **bob/bob**.
   b. Then click on the link **Use the access token** that is found on the start page.
   c. This page will use the **access token** and try to request and display the user details from the **UserInfo** endpoint.
   d. Explore the code in the controller that is used to access the information. It uses the **IdentityModel** library to access IdentityServer (https://identitymodel.readthedocs.io/).

e. If it all works; you should be presented with the claims about the user Bob.

Status	OK
UserInfo claims	

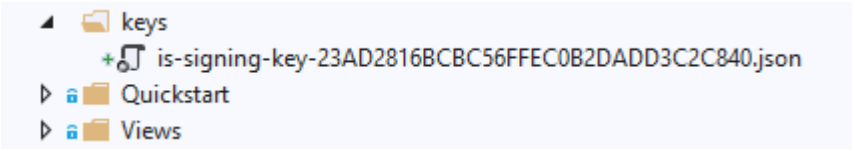
name	Bob Smith
given_name	Bob
family_name	Smith
email	BobSmith@email.com
email_verified	true
website	http://bob.com
sub	88421113

f. Do locate the request to the **/Connect/UserInfo** endpoint in Fiddler and see how the access token is included in the request.

8. Now, let's stop the **main project** without stopping the **OpenIdConnectClient** project.

   Next, delete the **tempkey.jwk** file from the IdentityService project and then start the projects again. A new **tempkey.jwk** file will automatically be generated by IdentityServer at startup.

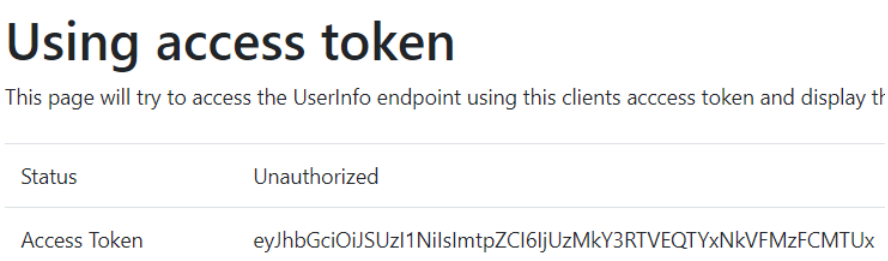
   Do also delete the **\Keys** folder that contains the created signing keys:



   (This folder is a new feature in Duende IdentityServer v5.x)

9. Now go back to the **OpenIdConnectClient** web page and reload the **/token/useaccesstoken** page.

   You should now be **unauthorized** because the token you have is signed with the previous key. This means that all issued tokens are now invalid.



10. How can we troubleshoot this? Let's look at a few steps we can do to explore the problem.

    a. Open **Fiddler** and locate the failing request to the **/connect/userinfo** endpoint.

    In the response you should see the **WWW-Authenticate** header that can give you some clues about the problem.

```
HTTP/1.1 401 Unauthorized
Date: Tue, 22 Sep 2020 12:33:11 GMT
Server: Kestrel
Content-Length: 0
Cache-Control: no-store, no-cache, max-age=0
Pragma: no-cache
WWW-Authenticate: Bearer realm="IdentityServer",error="invalid_token"
```

    b. Open https://jwt.io and paste in the access token. In the token header, you should see something like this:

```
{
  "alg": "RS256",
  "kid": "532F7E5DA616EE31B151A41D1CFD32D3",
  "typ": "at+jwt"
}
```

    The interesting fact here is the **key id** (kid) value. This is the ID of the key that was used to sign the token.

    Now, go back to Fiddler and you should see a request to the **jwks** endpoint at **/.well-known/openid-configuration/jwks**

    This endpoint returns the **JSON Web Key Set** (JWKS) document and it contains all the **public keys** that can be used to verify tokens. This endpoint might contain multiple public keys!

    Explore the content of the response in **Fiddler** from the **JWKS** request.

    A direct link to the JWKS endpoint can be found in the test client here:

    - Use the access token
    - View the JWKS endpoint

    Now notice the **kid** field found in the access token is not the same as the **kid** found in the **jwks** endpoint. Clearly, the key referenced by the token no longer exists and hence the token can't be verified.

    Under normal circumstances, the **kid** found in the token should exist in the **jwks** endpoint.

Exercise 11.2 – Adding permanent signing keys

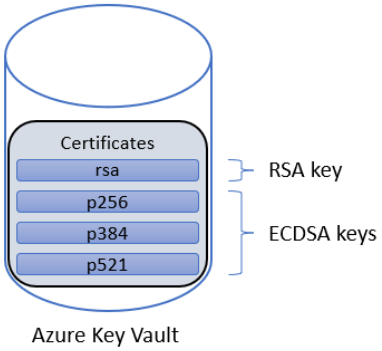
1. In the previous exercise we learned that tokens can no longer be verified if we lose the signing key.

   Now let's fix this problem!

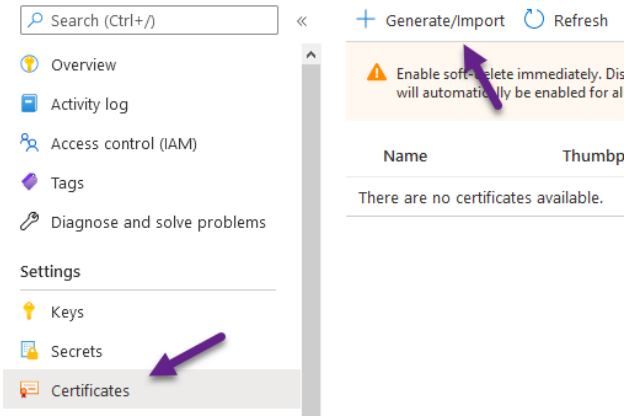
2. In the previous modules we created a set of **RSA** and **ECDSA** keys, now let's apply them to the **IdentityService** project.

3. If you haven't done it already, copy the keys from the remote machine to your local computer. Perhaps place them in the **c:\Keys** folder as a starting point.

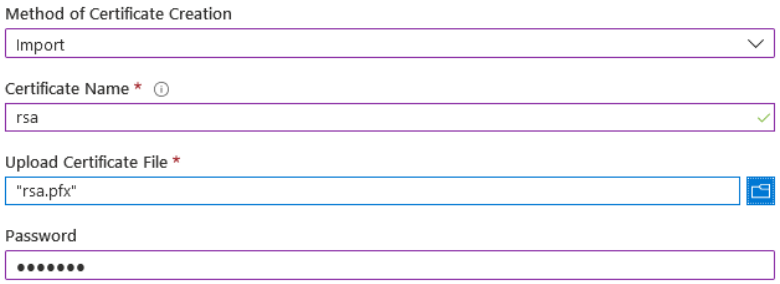
4. Now, we want to add all the **PKCS #12 files** (the **.pfx** files) that we created earlier to Azure Key Vault, as this picture shows:



   Open **Azure Key Vault** in the Azure portal and navigate to your key vault. Select **Certificates** and then select **Generate/Import**:



5. Now, import the four **PKCS #12** files (the **.pfx** files) to the key vault. Name them as the picture above shows (**rsa, p256, p384, p521**). Don't forget to enter the **password** you used when you created them earlier.



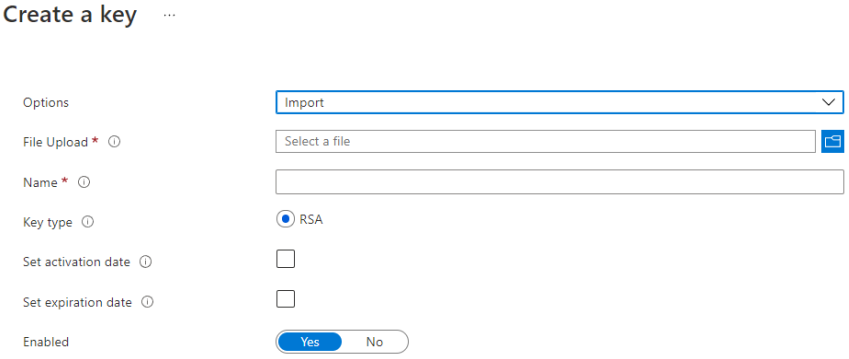
After the import, you should have four certificates in the vault with these names:

Name	Thumbprint	Status	Expiration Date
Completed			
p521	F241565A5692252586E2...	✓ Enabled	2021-09-13
p384	5FEB3C12109DE981BE7F...	✓ Enabled	2021-09-13
p256	8A283EA476DAEBD2AB9...	✓ Enabled	2021-09-13
rsa	ED9E67146F61115B6AD2...	✓ Enabled	2021-09-13

Why do we take the detour via certificates? Why not upload the keys directly to vault?

- We want to use the keys we carefully created ourselves in the previous modules. We could generate them directly in Azure Key Vault, but that would not be as fun ☺

- It seems that Azure Key Vault doesn't support importing **ECDSA** keys (only RSA keys). If we want to, we can create private ECDSA keys in Azure Key Vault directly. But we want to use our own keys!



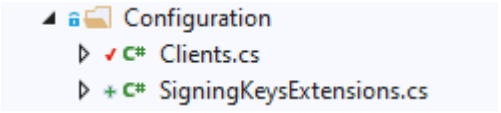
- .NET core seems to deal better with **X509 certificates** than raw keys. This will improve in .NET 5 and in the next version of IdentityServer.

Basically, we have many options here.

6. Now let's access these keys from **IdentityService**.

   To help us out, we have provided a helper class in the form of an extension method in the **\Starter-kit\Exercise 11\SigningKeysExtensions.cs** file.

   Add this file to the **Configuration** folder in **IdentityService**.



7. Review the class and try to understand what it does.

   It tries to add every possible **RSA/ECDSA**-based signature algorithm as defined in **RFC 7518** (JSON Web Algorithms) section 3.1.

   We don't support the **HS256**, **HS384** and **HS512** signature algorithms because IdentityServer does not support them.

8. Then, open your **IdentityService Startup class** and replace these two lines:

```
// not recommended for production - you need to store your key material
somewhere secure
builder.AddDeveloperSigningCredential();
```

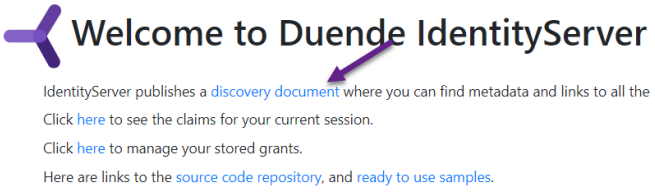
   with:

```
if (_environment.EnvironmentName != "Offline")
    builder.AddProductionSigningCredential(_configuration);
else
    builder.AddDeveloperSigningCredential();
```

9. Delete the existing **tempkey.jwk** file and **\Keys** folder if it exists. We don't need it anymore.

   In the **.gitignore** file in our repository we also have a rule to not commit this file to GitHub.

10. Start the **main project** again and then go to your IdentityServer browser window. Click on the **discovery document** link and then locate the **jwks_uri** link.



   Copy the link (https://localhost:6001/.well-known/openid-configuration/jwks) to a new browser tab.

Copy the content and paste it into https://jsonprettyprint.org/ to get a prettier view of the JSON document.

In the result, you should have seen that we now have added support for every possible signing key supported in IdentityServer.

Now you know how to add support for every possible signature algorithm.
(RSxxx, PSxxx and ESxxx)

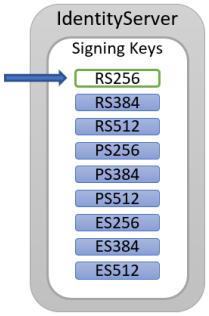
However you will probably only need to support one or two signature algorithms.

Bonus Exercise 11.3 – Controlling which signing algorithm to use (If you have time)

1. Now we have added the keys to IdentityServer, but which one will IdentityServer use to sign the tokens?

   The documentation says:
   *You can use multiple signing keys simultaneously, but only one signing key per algorithm is supported. The first signing key you register is considered the **default signing key.***

   This mean that the first key added will be used by default:



   But how can we choose other keys?

   Let's look at the documentation, that says:

   *Both clients and API resources can express **preferences** on the signing algorithm. If you request a single token for multiple API resources, all resources need to agree on at least one allowed signing algorithm.*
   https://identityserver4.readthedocs.io/en/latest/topics/crypto.html

   This means that when we define our **Clients** and **ApiResources**, we can also control what signing key to use.

For example:

Clients (Controls the ID-token):

```
var client = new Client
{
    ...
    //List of allowed signing algorithms for identity token.
    //If empty, will use the server default signing algorithm.
    AllowedIdentityTokenSigningAlgorithms = new List<string> {"PS384", "PS512"},
};
```

ApiResources (Controls the access token):

```
var api = new ApiResource()
{
    ...
    //Signing algorithm for access token.
    //If empty, will use the server default signing algorithm.
    AllowedAccessTokenSigningAlgorithms = new List<string> {"ES512"},
};
```

The first signing key match found will be used to sign the token. The IdentityServer source code for the selection of signing key can be found at: https://github.com/IdentityServer/IdentityServer4/blob/main/src/IdentityServer4/src/Services/Default/DefaultKeyMaterialService.cs

For example:

Due to lack of time, we won't try this out in this exercise.

2. Push the source code to GitHub and make sure that you can see the same keys in the IdentityServer **JWKS endpoint** in production.

Bonus exercise 11.4 – mkjwk - simple JSON Web Key generator (if you have time)

1. If you need to quickly generate a key in the JWK format, then you can use a tool at https://mkjwk.org/

   Go to this site and explore what it does. It supports both **RSA** and **ECDSA** keys. Explore the keys that it generates.

2. Try to create a few keys, for example:
   - 2048 bits
   - Key usage: signature
   - Algorithm: RS256
   - KeyID: Specify: TestKey
   - Show X.509: Yes

   As you notice, the tool generates the keys in both JWK and PEM format.

3. Open this page in a new browser tab
   https://redkestrel.co.uk/products/decoder/

   Then, copy a sample self-signed certificate from **mkwk** to the new site and decode it:
   (If you don't see this box, make sure the show X.509 dropdown is set to yes)

   Other tools that can decode certificates online are:
   - https://www.sslchecker.com/certdecoder
   - https://certlogik.com/decoder/

If you have time:

1. To view and decode the HTTPS certificate for a given website, you can use **openssl** to extract it.

   Go back to your remote machine and then run the following command to view the certificate for a given domain:

```
openssl s_client -showcerts -connect www.google.com:443
```

It will print out the certificate in PEM format. Copy the certificate and then paste it into one of the online tools above to view the certificate details on a low level.

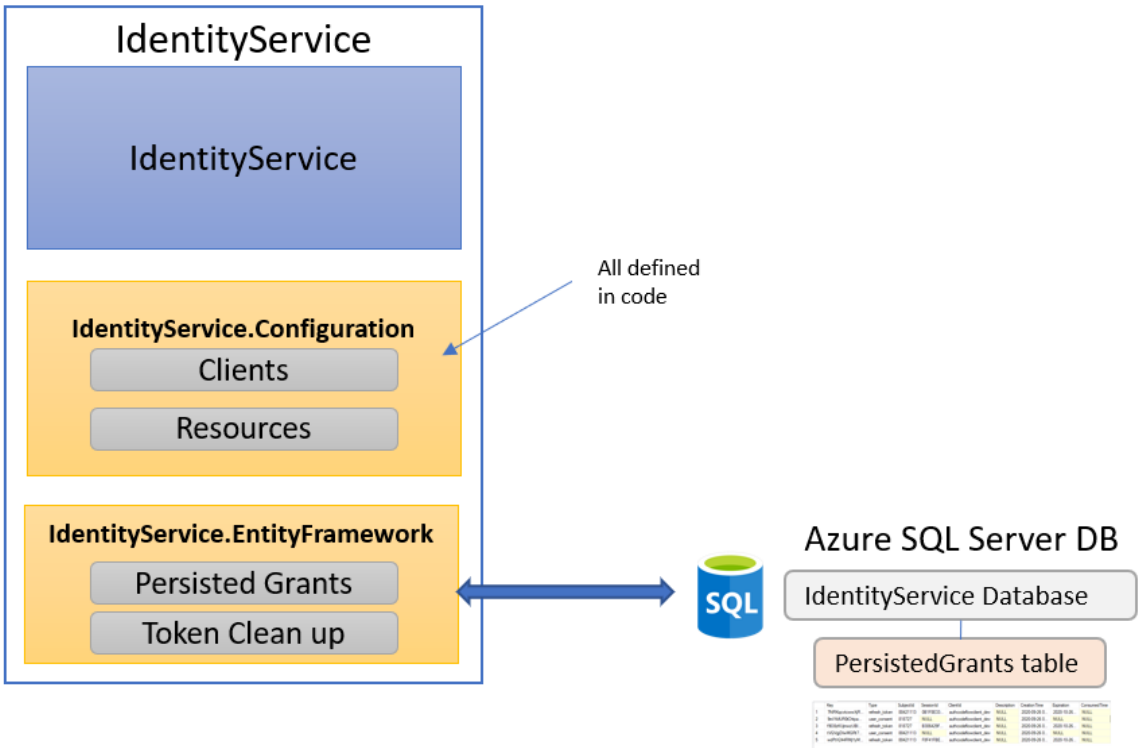
Further reading:

- RFC 7517 - JSON Web Key (JWK)
  https://tools.ietf.org/html/rfc7517
- RFC 7518 - JSON Web Algorithms (JWA)
  https://tools.ietf.org/html/rfc7518
- Self-signed certificate and configuring IdentityServer 4 with certificate
  https://www.teilin.net/self-signed-certificate-and-configuring-identityserver-4-with-certificate/
- JWT Signing using RSASSA-PSS in .NET Core
  https://www.scottbrady91.com/C-Sharp/JWT-Signing-using-RSASSA-PSS-in-dotnet-Core
- JWT Signing using ECDSA in .NET Core
  https://www.scottbrady91.com/C-Sharp/JWT-Signing-using-ECDSA-in-dotnet-Core
- Which elliptic curve should I use?
  https://security.stackexchange.com/questions/78621
- Cryptography, Keys and HTTPS
  https://identityserver4.readthedocs.io/en/latest/topics/crypto.html
- IdentityServer and Signing Key Rotation
  https://brockallen.com/2019/08/09/identityserver-and-signing-key-rotation/
- How to create a signing certificate and use it in IdentityServer4 in production?
  https://stackoverflow.com/questions/58136779/how-to-create-a-signing-certificate-and-use-it-in-identityserver4-in-production
- .NET Core X509Certificate2 usage (under Windows/IIS, Docker, Linux)
  https://stackoverflow.com/questions/53334142/net-core-x509certificate2-usage-under-windows-iis-docker-linux
- Is RSASSA-PKCS1-v1_5 a good signature scheme for new systems?
  https://crypto.stackexchange.com/questions/3850
- Using certificates from Azure Key Vault in ASP.NET Core
  https://damienbod.com/2020/04/09/using-certificates-from-azure-key-vault-in-asp-net-core/
- JWTs: Which Signing Algorithm Should I Use?
  https://www.scottbrady91.com/JOSE/JWTs-Which-Signing-Algorithm-Should-I-Use

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise 12 – Database

The goal with this exercise is to add two new **class libraries** to our main project, one for dealing with **clients/resources** and one for the **persisted grants**.



Exercise 12.1 – The database

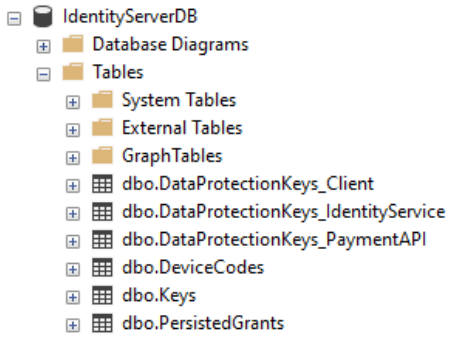
Let's first fix the database.

1. Open **SQL Server Management Studio** and connect to your database server.

2. Now we need to create the table for the **persisted grants.** We could, of course, use the Entity Framework migrations to create it for us. But, because it is just one table, it's simpler to just create it manually.

   Double-click on the file **\Starter-Kit\Exercise 12\PersistedGrants.sql** to open it in **Management Studio** and then run it against the **IdentityServerDB**.

Then refresh the tables to access the newly created tables:



The **DeviceCodes** table is not used in this course but is needed by the **TokenCleanup** service that we use later in this exercise. The **Keys** table is used by the new key management system in version 5.

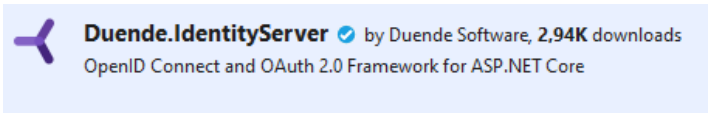
Exercise 12.2 – Creating the IdentityService.Configuration project

1. Open your main project and add a new **class library** (.NET Core) named **IdentityService.Configuration**

2. Make sure it targets **.NET 5**

3. Add a project reference to it from the main **IdentityService** project.

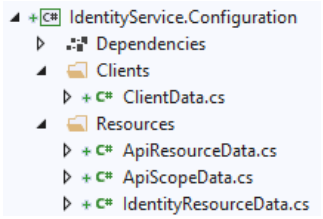
4. Add the **Duende.IdentityServer** NuGet package to the new project:



(Make sure you install version 5.x.x)

5. Let's refactor the IdentityServer configuration that is currently located in the **IdentityService\Configuration\Clients.cs** file. Locate this file in the project.

6. Open the **\Starter-Kit\Exercise 12\IdentityService.Configuration** folder and then copy the **Clients** and **Resources** folder to the newly created project.



Explore what the code contains.

7. Open the **IdentityService\Configuration\Clients.cs** file and move your existing clients to the new **\Clients\ClientData** class in the data project.

8. Delete the **IdentityService\Configuration\Clients.cs** file.

9. Locate and delete the **IdentityService\Config** class in the root of the **IdentityService** project and delete it.

10. Open your **IdentityService\Startup** class and modify the code so it becomes:

```
var builder = services.AddIdentityServer(options =>
    {
        options.EmitStaticAudienceClaim = true;
    }).AddTestUsers(TestUsers.Users)
    .AddInMemoryIdentityResources(IdentityResourceData.Resources())
    .AddInMemoryApiResources(ApiResourceData.Resources())
    .AddInMemoryApiScopes(ApiScopeData.Resources())
    .AddInMemoryClients(ClientData.GetClients());
```

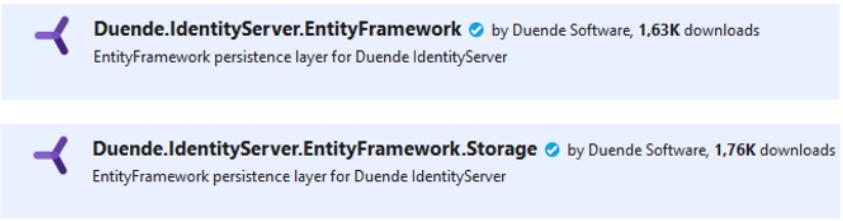
(The exact code is found in the **Startup.txt** file)

11. Make sure your application still compiles and then run your application and make sure everything works as before.

    We have only done a simple refactoring of the code so far.

Exercise 12.3 – Persisted grants

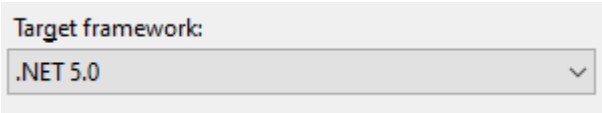
1. To implement the support for storing the **persisted grants** in our **SQL Server database**, we could very quickly just add these two projects and be done with it:



   But at the same time, hiding the logic inside these packages will prevent us from understanding how it works. To be on top of this, we will instead in this exercise add the **source code** for these two packages into a separate class library project that we can control and reason about.

2. Add a new class library (.NET Core) project named **IdentityService.EntityFramework**.

3. Make sure it targets **.NET 5**



4. Add a reference to this project dependency from the **IdentityService** project.

5. Right click on the **IdentityService.EntityFramework** project and edit the project file:

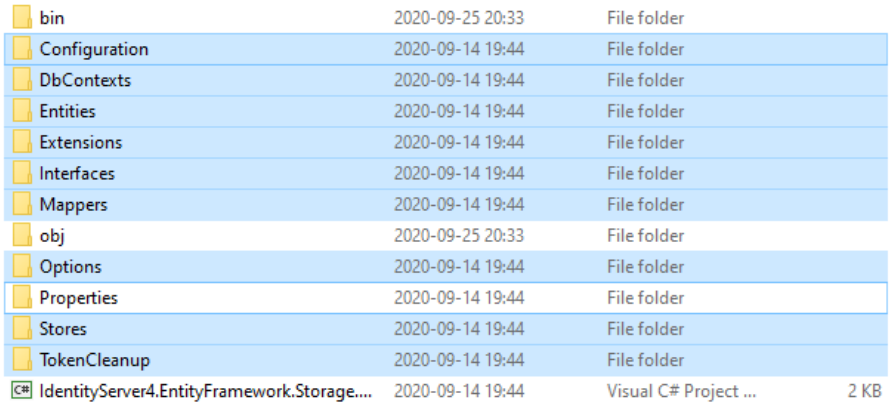
6. Open the **\Starter-Kit\Exercise 12\Projectfile.txt** and add the **<ItemGroup>** to the project file.

   Save the project file and rebuild the solution to add the pages to the project. Make sure it builds!

7. Unzip the **Duende IdentityServer** source code found in the **\Starter-Kit\IdentityServer** folder (IdentityServer-5.x.x.zip). Then copy the following folders and files to the **IdentityService.EntityFramework** project.

   1. \src\EntityFramework
      - **Services**
      - **IdentityServerEntityFrameworkBuilderExtensions.cs**
      - **TokenCleanupHost.cs**

   2. \src\EntityFramework.Storage
      - All folders except (**bin**, **Properties** and **obj** folders if present)



8. Rebuild the solution and everything should build without any errors.

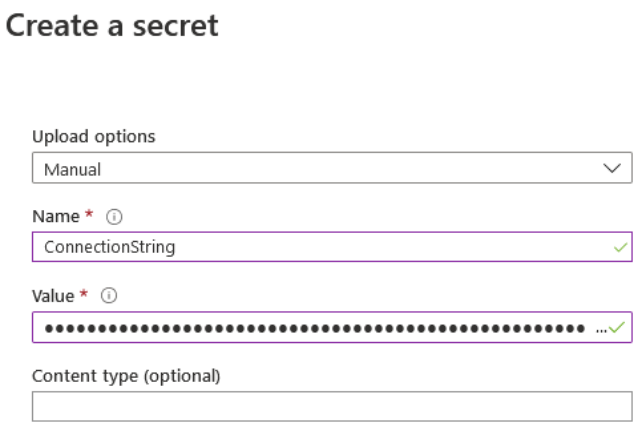
9. We need to configure **IdentityServer** to use the **EntityFramework** code we just added.

   Open your **Startup** class and add the following in bold:

```
var builder = services.AddIdentityServer(options =>
{
    options.Events.RaiseErrorEvents = true;
    options.Events.RaiseFailureEvents = true;
    options.Events.RaiseInformationEvents = true;
    options.Events.RaiseSuccessEvents = true;

}).AddTestUsers(TestUsers.Users)
    .AddInMemoryIdentityResources(IdentityResourceData.Resources())
    .AddInMemoryApiResources(ApiResourceData.Resources())
    .AddInMemoryApiScopes(ApiScopeData.Resources())
    .AddInMemoryClients(ClientData.GetClients())
    .AddOperationalStore(options =>
    {
        options.ConfigureDbContext = options =>
        {
            options.UseSqlServer(_configuration["ConnectionString"]);
        };
    })
```

   (The code is also found in the **\Starter-Kit\Exercise 12\Startup.txt** file)

10. We will store the connection string in **Azure Key Vault.** To add it do the following:
    1. Open the Azure portal
    2. Add a new **secret** named **ConnectionString** and set the value to the connection string found in the **Infrastructure.txt** file.



11. Before we run our application, we will also add support for **refresh tokens,** and we do that by:
    1. In the **IdentityService.Configuration\Clients\ClientData.cs** file add **OfflineAccess** to the list of allowed scopes:

```
AllowedScopes =
{
    //Standard scopes
    IdentityServerConstants.StandardScopes.OpenId,
    IdentityServerConstants.StandardScopes.Email,
    IdentityServerConstants.StandardScopes.Profile,
    IdentityServerConstants.StandardScopes.Phone,
    IdentityServerConstants.StandardScopes.OfflineAccess,
},
```

(Do add it to both client definitions)

    2. In the **Client\Startup** class, ask for the **offline_access** scope as well.

```
options.Scope.Clear();
options.Scope.Add("openid");
options.Scope.Add("profile");
options.Scope.Add("email");
options.Scope.Add("offline_access");
```

We will talk more about **refresh tokens** later.

Exercise 12.4 – Testing it out!

1. In SQL Server Management Studio query the **PersistedGrants** table and verify that it is empty.

2. Start the application and login using **bob/bob** or **alice/alice**

3. Look in the table again and you should now see two entries added to the table:

Results | Messages

	Key	Type	SubjectId	SessionId	ClientId
1	MWozCBKwVUJWPVcWj10Y6o7yBO07i5aC2Lv...	refresh_token	88421113	019FA1C6EDC28DF7C430CDF...	authcodeflowclient_dev
2	tVQVgjDIwMGRt7v1LeBKLVhzuPRzxV4hEsubiMj...	user_consent	88421113	NULL	authcodeflowclient_dev

4. But what about the **authorization codes**? Should they not also be present in this table?

   Let's find out!

5. Locate the **PersistedGrantStore** class in the **IdentityService.EntityFramework\Stores** folder.

6. Put a breakpoint in the **StoreAsync** and **RemoveAsync** methods.

7. Run the application and **logout** and then **login** again.

8. You will see that the **StoreAsync** is hit two or three times, once for the **consent** and once for the **authorization code**.

   Look what the provided token object contains.

9. When the **RemoveAsync** breakpoint is hit, go back to **SQL Management Studio**, and look in the **PersistedGrants** table.

	Key	Type	SubjectId
1	bUv2lAOxzd73b214bla73ztXcoEoBU2bPkuUuoJZ+Ys=	authorization_code	88421113
2	tVQVgjDlwMGRt7v1LeBKLVhzuPRzxV4hEsubiMjWzvI=	user_consent	88421113
3	ZTx4P95qFrzcJ6S7j6YAN3pYP6HMpyGS+Wsy0CaDUk0=	refresh_token	88421113

   You should now see that the **authorization code** is present, but it will be removed when the client uses it to get the real tokens. This means that the code is only present for a very short time in the table. The **Refresh token** is then added after that.

   This is the reason why we didn't see it before!

Exercise 12.5 – The token clean-up service

1. Included in the **IdentityService.EntityFramework project,** there is a **TokenCleanup** service that will remove old entries in the **PersistedGrants** table.

2. This service is disabled by default. To enable it, add the following to your **IdentityService Startup** class:

```
.AddOperationalStore(options =>
{
    options.EnableTokenCleanup = true;
    //The number of records to remove at a time. Defaults to 100.
    options.TokenCleanupBatchSize = 100;
    options.TokenCleanupInterval = 30;    //Seconds

    options.ConfigureDbContext = b =>
    {
        options.ConfigureDbContext = c =>
              c.UseSqlServer(_configuration["ConnectionString"]);
    };
});
```

3. Then locate the **TokenCleanupService** class in the **IdentityService.EntityFramework\TokenCleanup** folder.

   Put a breakpoint in the **RemoveExpiredGrantsAsync** method and then start the application again.

4. After **30 seconds**, the method will be called, and any expired items will be removed.

   You can also see entries in the log when the token service is executed:

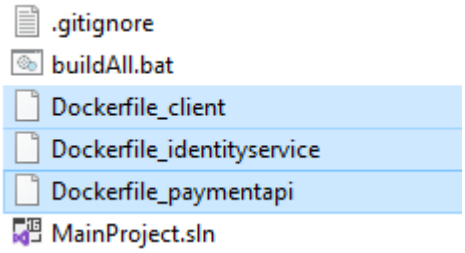
```
[17:42:39 Information] Duende.IdentityServer.EntityFramework.TokenCleanupService
Removing 2 grants

[17:42:40 Information] Duende.IdentityServer.EntityFramework.TokenCleanupService
Removing 0 device flow codes
```

5. Remove the breakpoints in the **TokenCleanup** service and then set the **TokenCleanupInterval** to something like **3600** to avoid having it run too often.

Exercise 12.6 – Deploying to production

1. We need to do a small tweak before we commit our code to production.

   Locate the three **Docker_*** files in the root of the GitHub repository and open them in **Notepad++:**



   In each one, we need to **uncomment** the two commented COPY lines:

```
COPY *.sln .
COPY Client/*.csproj ./Client/
COPY Infrastructure/*.csproj ./Infrastructure/
COPY PaymentAPI/*.csproj ./PaymentAPI/
COPY IdentityService/*.csproj ./IdentityService/
#COPY IdentityServer.EntityFramework/*.csproj ./IdentityServer.EntityFramework/
#COPY IdentityService.Configuration/*.csproj ./IdentityService.Configuration/
```

   Now commit the code to **GitHub** and deploy the application to production.

   Then run it in production and verify that everything still works, and that the new grants are added to the database when you login.

Things to do and explore

- Secure the connection against the database and who can access the database.
- Create separate development and production databases and use separate connection strings.

Further reading

- Creating Your Own IdentityServer4 Storage Library
  https://www.scottbrady91.com/Identity-Server/Creating-Your-Own-IdentityServer4-Storage-Library
- IdentityServer4.Dapper
  https://github.com/DarinHan/IdentityServer4.Dapper
- identityserver4-azurestorage
  Uses Azure Blob and Table Storage services as an alternative to Entity Framework/SQL data access for IdentityServer4.
  https://github.com/dlmelendez/identityserver4-azurestorage
- Using EntityFramework Core for configuration and operational data
  https://docs.duendesoftware.com/identityserver/v5/quickstarts/4_ef/
- Latest IdentityServer database scripts:
  https://github.com/DuendeSoftware/IdentityServer/tree/main/migrations/IdentityServerDb/Migrations

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

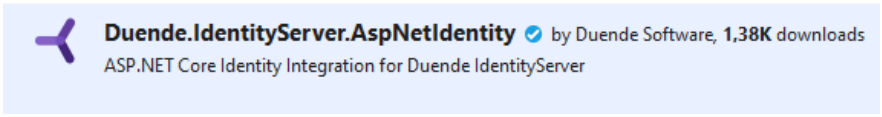
Main Exercise 13 – Users

Let's add support for the **user tables** that we created in the previous exercise.

Exercise 13.1 – Adding ASP.NET Identity Support to IdentityServer.

1. Add the following NuGet packages to the **IdentityService** project:

   Duende.IdentityServer.AspNetIdentity

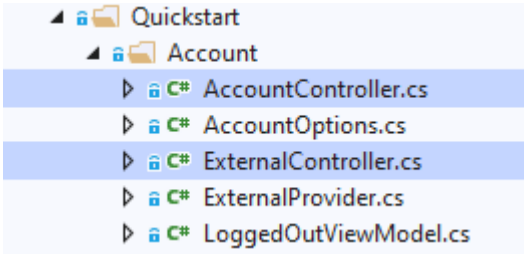
   (Version 5.x.x, do not install version 6.x)

   Microsoft.AspNetCore.Identity.EntityFrameworkCore

   (Version 5.x.x, do not install version 6.x)

2. Next, we need to update the **AccountController** and **ExternalController** to support **ASP.NET Identity**.

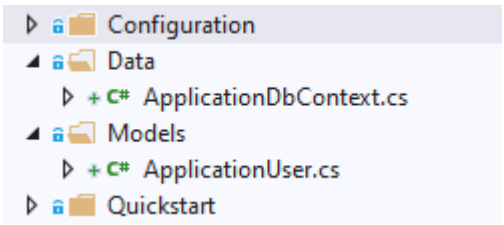
   

   Replace the existing two files with the ones found in **\Starter-Kit\Exercise 13\Main exercise.**

   The main difference here is that instead of using the **TestUserStore**, we use the **ASP.NET Identity UserManager** class. It provides the APIs for managing users in our database.

3. Add the **Data** and **Models** folders from the **\Starter-Kit\Exercise 13\Main exercise** folder to the root of the **IdentityService** project.

   You should now have these two files added to the project:

4. Next, we need to fix the **Startup** class and wire up ASP.NET Identity to work with IdentityServer.

   Open the **\Starter-Kit\Exercise 13\Main exercise\Startup.txt** file and follow the instructions.

5. Add this statement **at the top** of the **Startup.ConfigureServices** method:

```
services.AddDbContext<ApplicationDbContext>(options =>
{
    options.UseSqlServer(Configuration["ConnectionString"]);
});

services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

   (The code is found in the **Startup.txt file** in the starter kit)

   !!!Make sure it is placed before AddIdentityServer!!!

6. As the final step, add the **AddAspNetIdentity** statement at the end of this code block as shown below:

```
var builder = services.AddIdentityServer(options =>
        {
            options.EmitStaticAudienceClaim = true;
        })
    .AddTestUsers(TestUsers.Users)
    .AddInMemoryIdentityResources(IdentityResourceData.Resources())
    .AddInMemoryApiResources(ApiResourceData.Resources())
    .AddInMemoryApiScopes(ApiScopeData.Resources())
    .AddInMemoryClients(ClientData.GetClients())
    .AddOperationalStore(options =>
        {
            options.EnableTokenCleanup = true;
            options.TokenCleanupBatchSize = 100;
            options.TokenCleanupInterval = 3600;

            options.ConfigureDbContext = b =>
                {
                    options.ConfigureDbContext = c =>
                      c.UseSqlServer(Configuration["ConnectionString"]);
                };
        })
    .AddAspNetIdentity<ApplicationUser>();
```

7. Finally, remove the **TestUsers** class found in the **\Quickstart** folder and the **.AddTestUsers(TestUsers.Users)** call in your **Startup** class.

Exercise 13.2 – Testing the setup

1. To see in the queries sent to the database in the logs, then add the following setting to **IdentityService\Program.cs**: (under development)

   ```
   .MinimumLevel.Override("Microsoft.EntityFrameworkCore", LogEventLevel.Debug);
   ```

2. Start the application and try to login as **alice/alice** or **bob/bob.**

   You should find a lot of SQL queries in the IdentityService log window:

   ```
   [20:33:46 Debug] Microsoft.EntityFrameworkCore.Database.Command
   Executing DbCommand [Parameters=[@__userId_0='?' (Size = 450)],
   SELECT [a0].[Name]
   FROM [AspNetUserRoles] AS [a]
   INNER JOIN [AspNetRoles] AS [a0] ON [a].[RoleId] = [a0].[Id]
   WHERE [a].[UserId] = @__userId_0
   ```

   You can also find database related log entries in Seq! Do check it out!

3. When that works, push the code to **production** and verify that you can login there too.

   We have now successfully added support for our user database to IdentityServer!

Things to do and explore

- Secure the connection against the database and who can access the database.
- Create separate development and production databases and use separate connection strings.

User and profile related projects

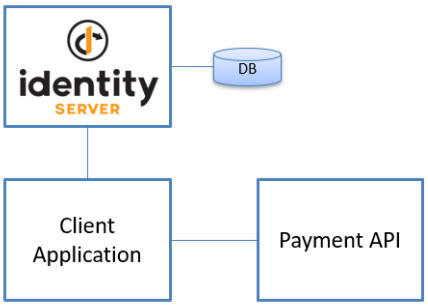
- IdentityServer4.Contrib.RedisStore
  https://github.com/AliBazzi/IdentityServer4.Contrib.RedisStore
- Skoruba.IdentityServer4.Admin
  https://github.com/skoruba/IdentityServer4.Admin
- An ASP.NET Core IdentityServer4 Identity Template with Bootstrap 4 and Localization
  https://github.com/damienbod/IdentityServer4AspNetCoreIdentityTemplate
- ASP.NET Core Identity with FIDO2 WebAuthn MFA
  https://github.com/damienbod/AspNetCoreIdentityFido2Mfa
- IdentityServer4 using MongoDB with Redis cache
  https://github.com/markglibres/identityserver4-mongodb-redis/tree/master
- Duende.IdentityServer.AspNetIdentity NuGet source code
  https://github.com/DuendeSoftware/IdentityServer/tree/main/src/AspNetIdentity

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

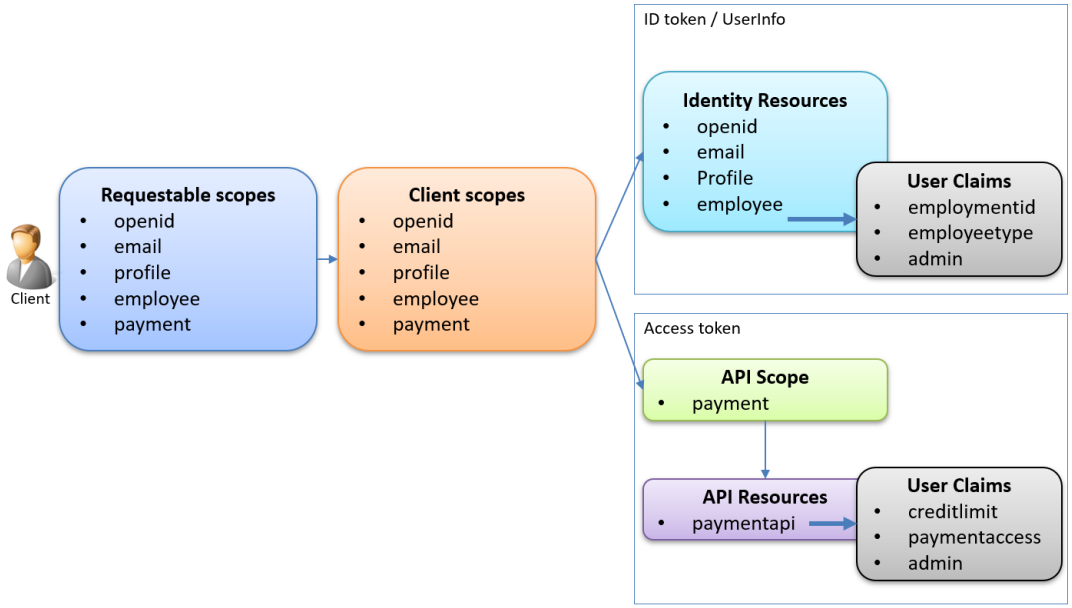
Main Exercise 14 – Tokens and claims

Exercise 14.1 – Configuring the clients

1. To be able to complete this picture below, we need to configure IdentityServer so that we can properly access the payment API from the client.



Open the **Configuration** folder in the **IdentityService.Configuration** project and add the necessary configuration so that the following requirements are true:



2. Make sure the **development** and **production client** has the same scopes.

3. Don't include the user claims in the **ID-token** to save space, instead we access it via the **UserInfo** endpoint. (Hint: set the correct flag in the client definition)

4. Set the **paymentapi**.**ApiSecret** to **myaipsecret** (hash it with SHA-256)

Exercise 14.2 – Verifying the setup

1. Open the **Claims debugger** that we used in the previous exercise and change the scope in the **AuthCodeController** to:

   ```
   openid email profile employee payment
   ```

2. Set the **tokenIntrospectionClientId** to **paymentapi** and make sure the **ApiResource** secret is the same as you provide in the previous exercise.

   The introspection endpoint is for API's to check the status of a token.

3. Then start the debugger and login as **alice**/**alice** using the **authorization code flow.**

   Do verify the following:

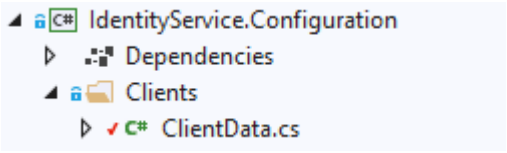
   - The **ID token**:
     - Does contain any user claims

   - The **access token** contains:
     - The **paymentapi** audience
     - The **admin**, **creditlimit** and **paymentaccess** claims
     - It contains the user claims as defined by the API resource (**creditlimit**, **paymentaccess**, **admin**)
     - The scope list contains the employee and payment scope
   - **User info** endpoint
     - Contains the user **profile** and **email** claims.
     - Contains the **Employmentid**, **Employeetype** and **admin** claims
   - **Token introspection** endpoint
     (Data that the API can request using the received access token)
     - Contains the **creditlimit**, **paymentaccess**, **admin** claims
     - The access token is **active** (the active claim is true)

   If you don't see the admin claim, then you are probably logged in with the wrong user. Not all users are admin.

Exercise 14.3 – Refactoring the client

1. Open the **ClientData** class (that contains the client definitions) and explore what it contains.

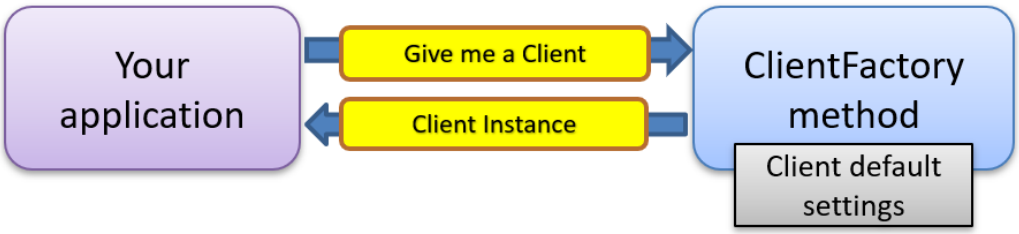


   Right now, this class is a bit hard to work with, because the classes are quite big and it is hard to keep them both client definitions in sync.

   We also notice that many of the configuration options are **the same** between the **development** and **production** client.

   Can we improve this?

2. We could create a simple client **factory method** that can create a basic client with a common base configuration:



3. Open the **\Starter-Kit\Exercise 14\ClientData.txt** file and explore what it does.

   Then replace the existing class in your project with this new one. If you have any custom settings, then you need to update them manually.

   Doesn't this feel like an improvement?

Exercise 14.4 – Wrapping it up

1. Start the application locally and verify that you can login as Alice.

2. When all works, do push the code to production and verify that it works there too.

Further reading

- Reference Tokens and Introspection
  https://leastprivilege.com/2015/11/25/reference-tokens-and-introspection/
- OAuth 2.0 Token Introspection
  https://tools.ietf.org/html/rfc7662

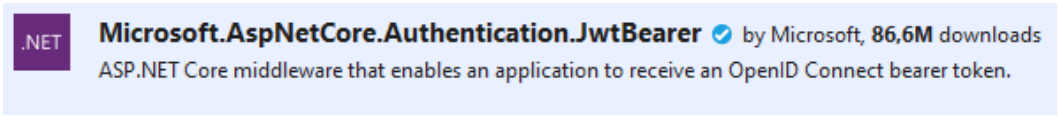
- Introspection Endpoint
  https://identityserver4.readthedocs.io/en/latest/endpoints/introspection.html
- Reference Tokens
  https://identityserver4.readthedocs.io/en/latest/topics/reference_tokens.html
- Flexible Access Token Validation in ASP.NET Core
  https://leastprivilege.com/2020/07/06/flexible-access-token-validation-in-asp-net-core/
- Automatic OAuth 2.0 Token Management in ASP.NET Core
  https://leastprivilege.com/2019/01/14/automatic-oauth-2-0-token-management-in-asp-net-core/

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise 15 – Securing the API

Exercise 15.1 – Protecting the API

1. Add the **Microsoft.AspNetCore.Authentication.JwtBearer** NuGet package to the **API** Project



2. Add the necessary protection in **Startup.cs** so that we only accept tokens with the **audience** claim set to **paymentapi** (see the presentation for details)

3. Do set the authority in the same way as we do it in the client, like:

```
opt.Authority = _configuration["openid:authority"];
```

4. Make sure **appsettings.Development.json** and **appsettings.Offline.json** contains the following setting:

```
{
  "openid": {
    "authority": "https://localhost:6001"
  }
}
```

   Then make sure the **appsettings.Production.json** contains the URL to your IdentityServer in production, like:

```
{
  "openid": {
    "authority": "https://studentXX-identityservice.webapi.se"
  }
}
```

5. Disable any **claims mapping** when the incoming token is converted into a ClaimsPrincipal Identity object.

6. Add mapping so that the **role** and **name** claim is mapped correctly.

7. Do set the **IncludeErrorDetails** JwtBearer option to **true**.

   What does it do?

> Defines whether the token validation errors should be returned to the caller.
> **Enabled by default**, this option can be disabled to prevent the JWT handler
> from returning an error and an **error_description** in the WWW-Authenticate header.

You should consider to set this flag to **false** once you are in real production to further strengthen the security.

8. Make the request pipeline contains these two modules:

```
app.UseAuthentication();
app.UseAuthorization();
```

Exercise 15.2 – Adding the API controller

1. Add a new API controller named **PaymentsController**

2. Open the **\Starter-Kit\Exercise15\PaymentsController.txt** file and follow the instructions.

3. We will consume this API in a later module

If you have time

- Check out the **Fake Authentication JWT Bearer for ASP.NET Core** project
  https://github.com/GestionSystemesTelecom/fake-authentication-jwtbearer
  This code allows you to fake a JWT Bearer and build integration test for ASP.Net Core application. By this way we can fake any authentication we need, without the need to really authenticate a user.
- Explore the source code for the **JwtBearerOptions** class
  https://github.com/dotnet/aspnetcore/blob/main/src/Security/Authentication/JwtBearer/src/JwtBearerOptions.cs
- Explore the source code for the **TokenValidationParameters** class
  https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/blob/dev/src/Microsoft.IdentityModel.Tokens/TokenValidationParameters.cs

Further reading

- Protecting an API using Client Credentials
  https://identityserver4.readthedocs.io/en/latest/quickstarts/1_client_credentials.html
- Protecting APIs
  https://identityserver4.readthedocs.io/en/latest/topics/apis.html
- **JwtBearer** source code at GitHub:
  https://github.com/dotnet/aspnetcore/tree/master/src/Security/Authentication/JwtBearer/src
- ConfigurationManager source code
  https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/blob/dev/src/Microsoft.IdentityModel.Protocols/Configuration/ConfigurationManager.cs
- Why we have two classes for JWT tokens?
  JwtSecurityTokenHandler vs JsonWebTokenHandler?
  https://stackoverflow.com/questions/60455167

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise - 16 - Securing the API – Advanced

Exercise 16.1 – Exploring the ETW logging infrastructure

1. Add the **\Starter-kit\Exercise 16\IdentityModelEventListener.cs** class to the **Infrastructure** project.

   This file contains an event lister that will listen for events from the **Microsoft.IdentityModel** classes. Do study what the code does.

2. But, how do we enable it?

   It's very easy! Just create an instance of the class in each **Startup. ConfigureServices** methods, like this:

   ```
   //Add the listener to the ETW system
   IdentityModelEventSource.Logger.LogLevel =
       System.Diagnostics.Tracing.EventLevel.Verbose;

   var listener = new IdentityModelEventListener();
   ```

   (The exact text is found in **\Starter-kit\Exercise 16\Startup.txt)**

   Just add it to your API project for now (You can add it to the other projects if you have time)

3. Now run the applications and at the top of the PaymentAPI console window, you will also find a list of all the potential **event sources** that you can subscribe to.

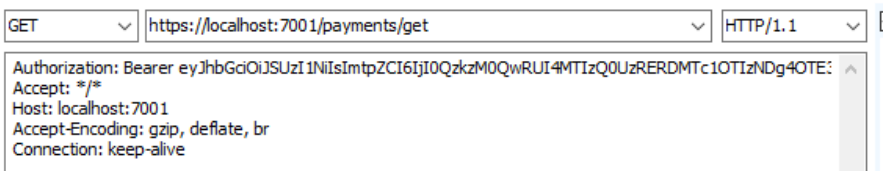
   ```
   EventSource found: Microsoft-Windows-DotNETRuntime
   EventSource found: System.Runtime
   EventSource found: System.Buffers.ArrayPoolEventSource
   EventSource found: Private.InternalDiagnostics.System.Net.Http
   EventSource found: Microsoft.IdentityModel.EventSource
   EventSource found: System.Diagnostics.Eventing.FrameworkEventSource
   …
   ```

   In this exercise we are only interested in events from **Microsoft.IdentityModel.EventSource.**

   You can comment out the **Console.Writeline** in the **OnEventSourceCreated** method to remove this printout if you like.

4. To see real events from this system, we need to send real HTTP requests to the API. To do this, open the file **\Starter-kit\Exercise 16\Fiddler.txt** and follow the instructions.

   It should look like this in Fiddler:

   (The token we use here is not valid and that's fine. We just want to have something that triggers token validation)

5. Send a request to the API and you should see that the **Debugger.Break()** statement is hit multiple times.

   ```
   //Send the event to the console
   Console.WriteLine("#### Event:" + eventData?.Payload[0]?.ToString() ?? "Unknown");

   Debugger.Break(); ≤ 6 719ms elapsed
   ```

   Just press **F5** and continue each time you hit this statement. This allows you to go to your console windows and see what the printout was.

   You should see events like these ones:

   ```
   IDX20805: Obtaining information from metadata endpoint: 'System.String'.
   IDX21811: Deserializing the string: 'System.String' obtained from metadata endpo
   ect.
   IDX21812: Retrieving json web keys from: 'System.String'.
   IDX20805: Obtaining information from metadata endpoint: 'System.String'.
   IDX21813: Deserializing json web keys: 'System.String'.
   :53:48 IDX12716: Decoding token: 'System.String' into header, payload and signatu

   48 Library version: 6.7.1.0.
   49 Date: 01/27/2021 13:53:49.
   ```

   (IDXxxxxx are internal Microsoft error codes)

6. However, it looks like we have a bug here. When we see statements like

   ```
   Retrieving json web keys from: 'System.String'.
   ```

   Did Microsoft do some mistake here?

   No, further down the log you will find a statement that says:

   ```
   PII (personally identifiable information) logging is currently
   turned off. Set IdentityModelEventSource.ShowPII to 'true' to view
   the full details of exceptions.
   ```

   So, to prevent secrets from ending up in the logs, they replace the secrets with the type names instead.

7. How can we see the real data?

   Set this flag to true in the PaymentAPI **startup.cs** class:

```
IdentityModelEventSource.ShowPII = true;
```

So, that the code in your startup should looks like this:

```
//Add the listener to the ETW system
var listener = new IdentityModelEventListener();
IdentityModelEventSource.Logger.LogLevel =
    System.Diagnostics.Tracing.EventLevel.Verbose;
IdentityModelEventSource.ShowPII = true;
```

8. Comment out the **Debugger.Break();** statement and then start the Main project again.

   Then send a new API request from Fiddler and explore the event data in the console window without the PII filter.

   As you see it is **very detailed** and also contains **sensitive information** like the actual access tokens.

   You could raise the logging level from verbose to the **warning** level to avoid exposing the tokens in the logs.

9. Should you use it?

   In general no, but it is good to know that this feature exists and now you know how to get extract this information from the **Microsoft.IdentityModel** or other event sources.

   One option is to just use it in development, but not in production.

   Don't forget that you can also apply this kind of logging in the Client and IdentityServer as well.

Exercise 16.2 – BackchannelHttpHandler

1. Let's add so that we also get logging when the **JwtBearer handler** makes request to our IdentityServer to get new configuration data.

2. Add the file **\Starter-kit\Exercise 16\BackChannelListener.cs** to the **Infrastructure** project.

   It contains a **HttpMessageHandler** that will log each time we make a request to get the IdentityServer configuration.

   Explore the code and make sure you understand what it contains.

3. To enable it, add the following to the PaymentAPI **AddJwtBearer** method:

```
.AddJwtBearer(opt =>
{
    ...
    opt.BackchannelHttpHandler = new BackChannelListener();
    opt.BackchannelTimeout = TimeSpan.FromSeconds(5);
});
```

4. We can also add it to the client project, by adding the following to the **AddOpenIdConnect options**:

```
.AddOpenIdConnect(options =>
{
    ...
    options.BackchannelHttpHandler = new BackChannelListener();
    options.BackchannelTimeout = TimeSpan.FromSeconds(5);
});
```

5. Start the project and either login as a user in the **client** or send a request to the API using **Fiddler Composer** as we did in the previous exercise.

   Then open **SEQ** and see if you can find the requests produced by the back channel handler.

   The title of the log entries should be **Loading IdentityServer configuration** and they should look something like this:

   Do you find them?

   You can search for them using the **SourceContext='BackChannelListener'** search query.

Exercise 16.3 – Improving resiliency by waiting for IdentityServer

Let's try to improve our API by waiting for IdentityServer to be available at startup. No traffic will be allowed through the API until its reachable.

1. Add the **Middleware** folder found in **\Starter-kit\Exercise 16\Exercise 16.3** to the **PaymentAPI** project.

   Do review what the code contains.

2. To enable it, add the following to your **request pipeline** in your PaymentAPI Startup class:

```
app.UseExceptionHandler("/Home/Error");
app.UseSerilogRequestLogging();

//Wait for IdentityServer to startup
app.UseWaitForIdentityServer(new WaitForIdentityServerOptions()
    { Authority = _configuration["openid:authority"] });

app.UseHttpsRedirection();
app.UseSecurityHeaders();
```

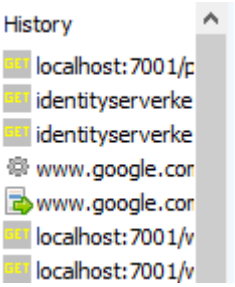
(The code is found in the **\Starter-kit\Exercise 16\Exercise 16.3\Startup.txt** file)

3. Start **Fiddler**

4. Start the **PaymentAPI** without the other projects by choosing **Debug -> Start new instance**

5. In **Fiddler composer** Issue API requests to the API and verify that all requests return a **503 error** back to you. (Because IdentityServer is not running)

   If you don't have the previous API request in Composer, then you can find it in the **composer history log**:

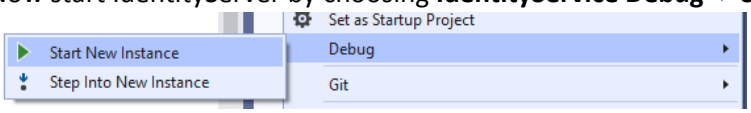


6. In **Seq**, you should find requests like these ones:

   - HTTP GET /payments/get responded 503 in 0.0096 ms
   - HTTP GET /payments/get responded 503 in 2003.2519 ms
     Failed to reach IdentityServer at startup
   - HTTP GET /payments/get responded 503 in 0.0229 ms
   - HTTP GET /payments/get responded 503 in 0.0178 ms
   - HTTP GET /payments/get responded 503 in 2018.2805 ms
     Failed to reach IdentityServer at startup
   - HTTP GET /payments/get responded 503 in 0.0099 ms
   - HTTP GET /payments/get responded 503 in 0.0186 ms

   It will periodically try to reach IdentityServer.

7. Now start IdentityServer by choosing **IdentityService Debug -> Start new instance**



   After it starts up, all the requests will pass through the middleware as usual. Probably returning a **401 Unauthorized** response instead.

8. What do you think? Is this a good idea to have this approach at startup of the API?

If you have time

Visit the GitHub Advisory Database and explore what it contains at:
https://github.com/advisories

Click on the **NuGet category** and see what the latest security vulnerabilities are.

Further reading

Azure Active Directory IdentityModel Extensions for .NET
https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet

The source for parts of ASP.NET authentication and authorization stack, including:

- ConfiguratioManager to download the IdentityServer configuration parameters
- TokenValidator
- JWT handling
- …

- .NET Core logging and tracing
  https://docs.microsoft.com/en-us/dotnet/core/diagnostics/logging-tracing
- Microsoft.IdentityModel Wiki
  https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/wiki
- Why the Personally Identifiable Information (PII) replaces facts with the type name instead.
  https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/issues/1589

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise 17 - Consuming the API

Exercise 17.1 – Preparations

1. In a previous exercise we configure IdentityServer client and resources to support the following scopes:

   `openid email profile employee payment`

   Update the client to ask for these scopes.

2. We need to update our **Appsettings** files add the URL to the Payment API.

   For **appsettings.Development.json** and **appsettings.Offline.json** add the **paymentapiurl** entry:

```
{
  "openid": {
    "clientid": "authcodeflowclient_dev",
    "authority": "https://localhost:6001"
  },
  "paymentApiUrl": "https://localhost:7001"
}
```

   For **appsettings.Production.json** add the following entry:

```
{
  "openid": {
    "clientid": "authcodeflowclient_dev",
    "authority": "https://localhost:6001"
  },
  "paymentApiUrl": "https://studentXX-paymentapi.webapi.se"
}
```

   (where XX is your personal ID)

Exercise 17.2 – Registering a HttpClient

1. In your Startup class register a new **HttpClient** named **paymentapi** that is configured with the following settings:

   a. **BaseAddress** should be set from the **paymentApiUrl** configuration parameter.
   b. Add the **Accept** header with this value "**application/json**".
   c. Set timeout to **5 seconds**

Exercise 17.3 – Calling the API

Let's add so that we do a request to the API when we go to the **Privacy** page.

1. First, we need to inject and instance of **IHttpClientFactory** to our **HomeController** constructor. Save it to a private field.

2. Then in the Privacy action method, add the code to make a request to the API.
   a. Get the **access token**
   b. Then ask for the **paymentapi** client
   c. Set the **bearer token**
   d. Do the request to the **/payments/get** url
   e. Parse the JSON response and store it in the ViewBag using:

   ```
   ViewBag.Json = JObject.Parse(content).ToString();
   ```

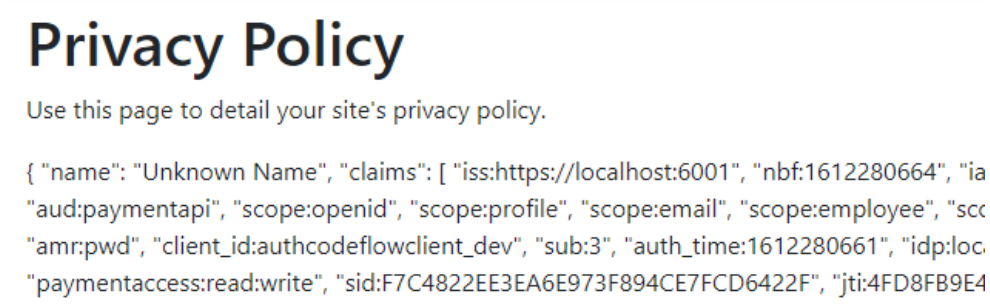
3. In the privacy view, add the following to print out the result to the page:

   ```
   Payment data result:<br>
   @ViewBag.Json
   ```

4. Start **Fiddler**

5. Run all three applications and then login as **joe/joe**.

6. Then go to the Privacy page in the client and if all is successful you should get back something that looks like this:

   The API just returns the user details as seen by the **PaymentController** in the API.

7. Do locate the request to the API in Fiddler and examine the request and response.

   Congratulations! We finally made a successful request to our API! ☺

Exercise 17.4 – Logging calls to the API

Let's add so that we can log every call to the API.

1. To do that we first add a custom logging Http message Handler to the **HttpClient** we registered earlier.

   Add the **\Starter-kit\Exercise 17\ SerilogHttpMessageHandler.cs** to the **Infrastructure** project.

2. Register the **SerilogHttpMessageHandler** in the **Startup.ConfigureServices** method:

```
services.AddTransient<SerilogHttpMessageHandler>();
```

3. Add the message hander to the **paymentapi** HttpClient using the **AddHttpMessageHandler** method:

```
services.AddHttpClient("paymentapi", client =>
{
    client.BaseAddress = new Uri(_configuration["paymentApiUrl"]);
    client.Timeout = TimeSpan.FromSeconds(5);
    client.DefaultRequestHeaders.Add("Accept", "application/json");
}).AddHttpMessageHandler<SerilogHttpMessageHandler>();
```

4. Start the application again and do a new request to the API.

   This time the execution breaks when it hits the **Debugger.Break**() method in the handler:

```
.Debug("API Request");

Debugger.Break();

return response;
```

   Do examine what was printed to the **console** and the log entry in **Seq**.
   (The request will not end up in **Seq** until you resume the program execution)

5. Stop the application and comment out the **Debugger.Break()** statement.

6. Push the code to production and make sure the API request works there too.

7. In production you will of course not see the request to the API in Fiddler and that's why logging of these requests is important to have.

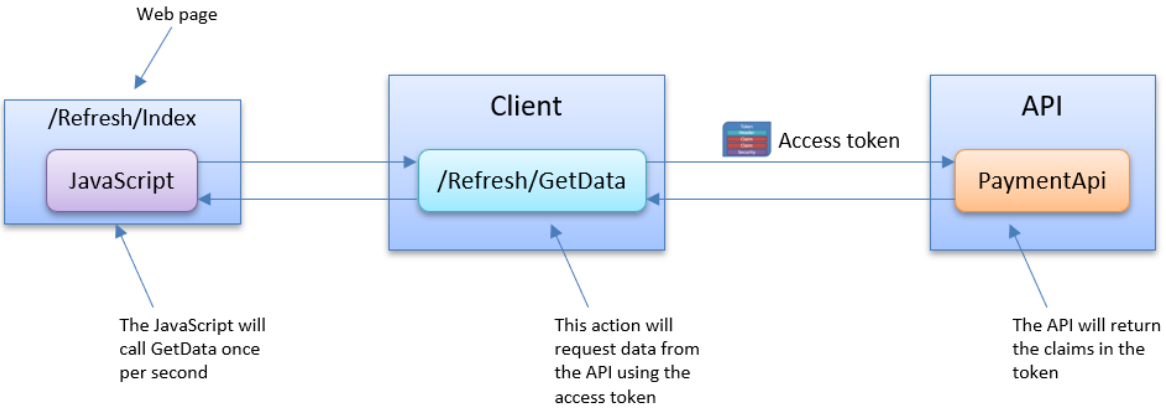
Further reading

- Make HTTP requests using IHttpClientFactory in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/fundamentals/http-requests
- You're using HttpClient wrong and it is destabilizing your software
  https://aspnetmonsters.com/2016/08/2016-08-27-httpclientwrong/
- Configure HttpClientFactory to use data from the current request context
  https://stackoverflow.com/questions/51358870

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`

Main Exercise - 18 - Refresh tokens

In this exercise we create the following setup:



Exercise 18.1 – Adding support for refresh tokens in IdentityService

1. Make sure the **offline access** is enabled in the Client definition.

   Do enable it in the **ClientData. ClientFactory(…)** method in the IdentityService.Configuration project so that both the development and production client gets this setting.

   Optionally, you can set the **RefreshTokenUsage** to **TokenUsage.OneTimeOnly** to make this setting explicit in your code.

2. Next set the **access token** life time in the **ClientFactory** to **45 seconds**.

Exercise 18.2 – Tweaking the PaymentAPI

1. To make sure the API does not accept slightly expired tokens, we need to set the **ClockSkew** to Zero.

   ```
   opt.TokenValidationParameters.ClockSkew = TimeSpan.FromSeconds(0);
   ```

   Set this in the **JwtBearer** options.

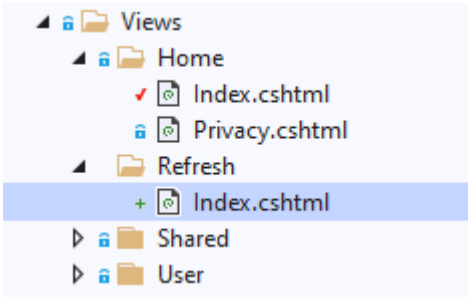
In production it can be wise to accept some **clockskew** due to clock drift between services and clients.

Exercise 18.3 – Updating the client

1. Make sure you ask for the **offline_access** scope in the Startup class.

2. Add the **\Starter-kit\Exercise 18\RefreshController.cs** to the Client **Controllers** folder.

3. Add the **Refresh** folder found in the starter-kit to the **Client\Views** folder.



4. Add a link to the **/refresh/Index** page from your **/Home/Index** view.

5. Add the file **refresh.js** to the **\wwwroot\js** folder.

   This script will once a second call the **GetData** action method and present the result to the page.

Exercise 18.4 – Running the program

1. Start the application and login as **joe/joe**. If you are already logged in, do logout and login again. The access token only has a 45 second lifetime.

2. Visit the **Refresh/GetData** page and if all is working you should see the following result:

   # Refresh token demo page

   - 19:37:44 Got Data name='Unknown Name' (Access token exires in 42 sec
   - 19:37:46 Got Data name='Unknown Name' (Access token exires in 40 sec
   - 19:37:46 Got Data name='Unknown Name' (Access token exires in 38 sec
   - 19:37:46 Got Data name='Unknown Name' (Access token exires in 38 sec
   - 19:37:46 Got Data name='Unknown Name' (Access token exires in 38 sec
   - 19:37:47 Got Data name='Unknown Name' (Access token exires in 37 sec

   The name is unknown because the access token does not contain a name claim. The "Unknown Name" is set in the Payment controller in the API when the claim is missing.

   And eventually, the token will expire and you can no longer access the API

   - 19:38:23 Got Data name='Unknown Name' (Access token exires in 1 sec
   - 19:38:24 Got Data name='Unknown Name' (Access token exires in 0 sec
   - 19:38:25 Response status code does not indicate success: 401 (Unauthorized).
   - 19:38:26 Response status code does not indicate success: 401 (Unauthorized).

   If you logout and login again, you can access the API for up to 45 seconds.

Exercise 18.5 – Introducing refresh tokens

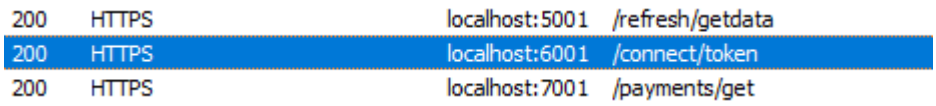
1. Open the file **\Starter-Kit\Exercise 18\Execise 18.5\RefreshController.txt** and follow the instructions.

   Review the code and try to understand what it does.

2. Start Fiddler and then start the application. Do login again and then access the refresh controller again.

3. When the access token is a about to expire, it will automatically ask for a new set of tokens.
4. Do check in Fiddler how the refresh token is used to get new access and refresh tokens.

   Locate the request to the token endpoint and explore the **request** and **response**:



   In the next exercise we will explore how can streamline the refresh token management.

If you have time

- Can you figure out how to get the users real name into the refresh page?

Further reading

- Refresh Tokens in IdentityServer4 v4
  https://leastprivilege.com/2020/06/29/refresh-tokens-in-identityserver4-v4/
- Hardening Refresh Tokens
  https://leastprivilege.com/2020/01/21/hardening-refresh-tokens/
- Use bearer tokens in client applications
  https://doc.sitecore.com/developers/91/sitecore-experience-manager/en/use-bearer-tokens-in-client-applications.html
- Making API calls using the access token and refresh token from an ASP.NET Core authentication handler
  https://blog.maartenballiauw.be/post/2020/01/13/making-api-calls-using-the-access-token-and-refresh-token-from-an-aspnet-core-authentication-handler.html

Main Exercise - 19 - Consuming the API – Advanced

Exercise 19.1 – Adding a session store

1. Before we optimize the cookie size, let's look at the current cookie size in your browser.

   In some cases, the cookie is broken up into several chunks:

Name	Value	Domain	Path	Expires / Max-Age	Size	H
.AspNetCore.CookiesC2	ALdMxbblLiu...	localhost	/	Session	776	
.AspNetCore.CookiesC1	CfDJ8D1HSNT...	localhost	/	Session	4008	
.AspNetCore.Cookies	chunks-2	localhost	/	Session	27	

   (in this example the session cookie is about 4800 bytes)

2. Let's improve this by adding a custom session store to the client.

   Open the **\Starter-Kit\Exercise 19** folder and add the file **MySessionStore.cs** file to the **Client** project.

3. Add the session store to the cookie authentication handler in your Startup class. See the presentation for details.

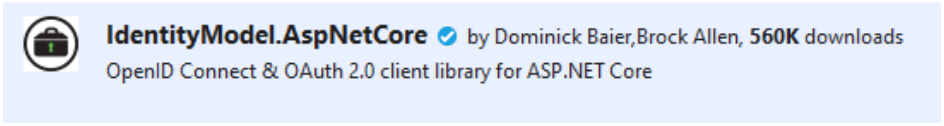
4. **Run the application again and login as joe/joe.** Then examine the new size of the cookie in your browser. You should see that it is very small:

Name	Value	Domain	Path	Expires / Max-Age	Size
.AspNetCore.Cookies	CfDJ8D1HSNT...	localhost	/	Session	323

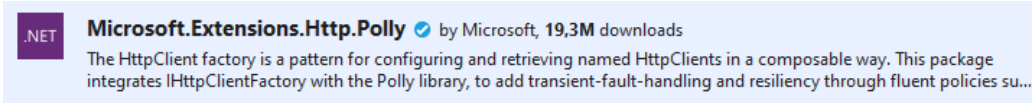
Exercise 19.2 – Improving the client using IdentityModel.AspNetCore

1. In this exercise we will introduce the **IdentityModel.AspNetCore** to improve the code we have in the **RefreshController**.

2. First, add the **IdentityModel.AspNetCore NuGet** package.



3. Then add the **Microsoft.Extensions.Http.Polly** NuGet package



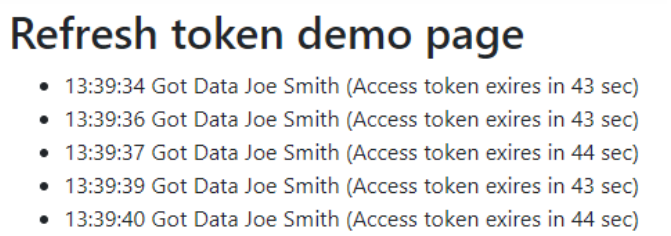
4. Now we need to activate it in our **Startup** class. Open **Startup.txt** in your **\Starter-Kit\Exercise 19** and follow the instructions.

5. Open the file **RefreshController.txt** in your starter kit folder and follow the instructions.

6. Start **Fiddler** and then run the application and login as **joe/joe.**

7. Now try to access the API and we successfully get back the data.

   But we have one problem:



   If you look in Fiddler, you see that re request new access tokens all the time.

8. The problem is that the Access Token Management library will ask for new tokens **1 minute** before it expires. And in our setup the access token lifetime is only **45 seconds**.

9. Let's change this to just 5 seconds by setting this property:

```
// adds user and client access token management
services.AddAccessTokenManagement(options =>
{
    options.User.RefreshBeforeExpiration = TimeSpan.FromSeconds(5);
})
```

10. Start the application and try to get the API data again.

    Now you should see that the token is refreshed when it is about to expire:

- 13:56:35 Got Data Joe Smith (Access token exires in 4 sec)
- 13:56:36 Got Data Joe Smith (Access token exires in 3 sec)
- 13:56:38 Got Data Joe Smith (Access token exires in 44 sec)
- 13:56:38 Got Data Joe Smith (Access token exires in 44 sec)

Congratulations! You made it to the end of this course!

Further reading

- IdentityModel.AspNetCore
  https://identitymodel.readthedocs.io/en/latest/aspnetcore/overview.html
  Distributed Cache sample
  https://github.com/aspnet/Security/issues/1034
- ITicketStore Interface
  https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.cookies.iticketstore
- Cache in-memory in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory
- MemoryCache
  https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.caching.memory.memorycache

Material feedback. We continuously strive to improve our course material. If you have found typos, bugs, found the exercises unclear or if you have other suggestions on how we could improve our course material, please visit the following:
`https://tinyurl.com/tndatafeedback`