IdentityServer in Production

T339

Module Exercises

Module Exercise 2 – Configuration

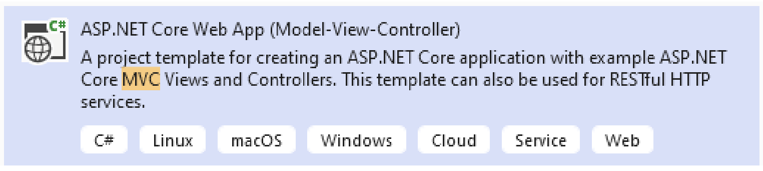
Exercise 2.1 – Configuration requirements

1. Discuss configuration with a fellow student (or in your group) and consider the following questions:
   - How do you handle configuration in your work today?
   - How do you handle secrets, certificates, and passwords today?
   - What are your current configuration pain points?

Exercise 2.2 – Hello environments

1. In the previous module we introduced the **ASPNETCORE_ENVIRONMENT** environment variable. In this exercise we will get more familiar with this variable.

   Close the main project and create a new **ASP.NET Core Web Application** project using the **MVC** template.

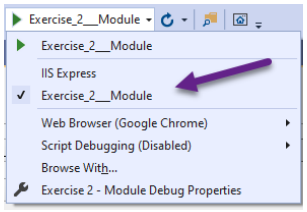
   

   ASP.NET Core Web App (Model-View-Controller)

   A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

   C# Linux macOS Windows Cloud Service Web

2. Make sure the project is selected as start-up:

   
   Exercise_2__Module
Exercise_2__Module
IIS Express
✓ Exercise_2__Module
Web Browser (Google Chrome)
Script Debugging (Disabled)
Browse With...
Exercise 2 - Module Debug Properties

3. Open **\Starter-kit\Exercise 2\Index.txt** and replace the content in the **\Views\Home\Index.cshtml** file.

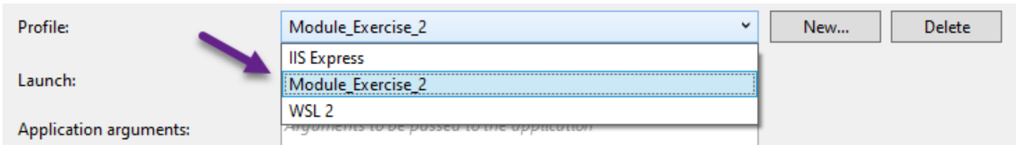
4. The view shows the current environment and what the various **IsXXXXXXX()** extension methods returns when called.

5. Run the application and you should now see that you are in the **Development** environment.

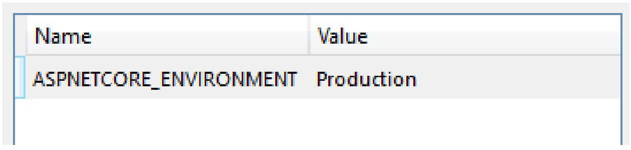
6. To set the application in a **Production** environment, go to the **project properties** and click on the **Debug** tab.

   for **Visual Studio 2019**

   - Make sure the correct profile is selected (the project, not IIS Express)



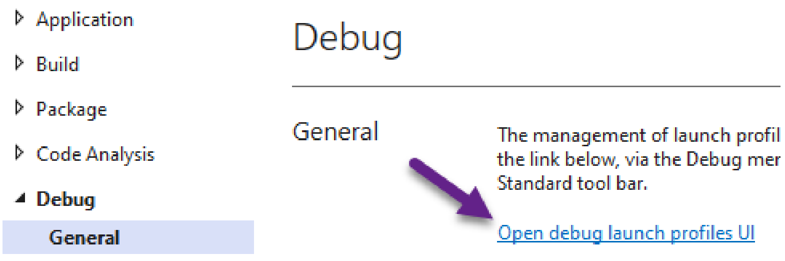
   - Then for the current profile set the environment variable to **Production**.



Name	Value
ASPNETCORE_ENVIRONMENT	Production

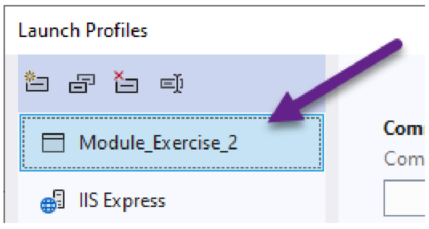
   For **Visual studio 2022**

   - Click on the **Open debug launch profiles UI**

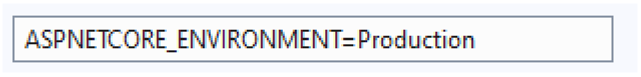


The management of launch profil
the link below, via the Debug mer
Standard tool bar.

   - Make sure the **correct profile** is selected (your project, not IIS Express)



   - Then for the current profile set the environment variable to **Production**.



ASPNETCORE_ENVIRONMENT=Production

7. Run the site again and see the change in output.

8. What happens if the **ASPNETCORE_ENVIRONMENT** variable is empty? Or deleted?

   Give it a try and see what environment you end up with.

9. When you run the application, you see that there's support for a **Staging** environment as well. Set the variable to **Staging** and see the result.

   We will not use the **staging** environment in this course.

10. Stop the project and then open a **command prompt** in your project directory and then execute:

```
dotnet run
```

Then open a browser and navigate to one of the URL's displayed, for example:

```
info: Microsoft.Hosting.Lifetime[14]
     Now listening on: https://localhost:7147
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5147
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
```

(The port number might vary...)

You should now see that you are in the environment as defined in **launchSettings.json.**

Stop the server by pressing **CTRL + C**

11. To control the environment, you can pass it in via the **ASPNETCORE_ENVIRONMENT** variable. Let's try that by setting it using:

```
Set ASPNETCORE_ENVIRONMENT=Production
dotnet run --no-launch-profile
```

(We add **--no-launch-profile** to ignore the settings in **launchSettings.json**.)

If you now go to your browser, you should see that we are in the **Production** environment.

Beware, the port number might change, so you need to check again in the console output what port Kestrel is listening on.

Try to set it to values like **Offline** and see what the result is on the page.

Exercise 2.3 – Hello configuration

1. Let's explore the **appsettings.json** files.

   first stop the server in the command window (**CTRL + C**) and return to Visual Studio.

   In Visual Studio set the **ASPNETCORE_ENVIRONMENT** variable to **Development**.

2. Open **appsettings.json** and add a new **Settings** entry as follows:

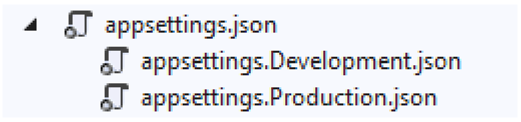
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",

  "Settings": {
    "Name": "Joe",
    "Email": "joe@tn-data.se"
  }
}
```

3. To display the values, do the following:
   1. Open **\Views\Home\Index.cshtml** and add the following at the end of the page to display the name and email settings:

```
@{
    var mysettings = config.GetSection("Settings");
}
<br />
Name: @mysettings["Name"]<br />
Email: @mysettings["Email"]<br />
```

4. Run the application and you should now see the name and email presented on the page.

5. Now add a new **appsettings.Production.json** file (be careful with the casing).



Then replace its content with:

```
{
  "Settings": {
    "Name": "Joe in production"
  }
}
```

6. Run the application again and you see that the name is still **Joe**.

7. Change the environment to **Production** and you should see that the name in **appsettings.Production.json** overrides the value in **appsettings.json**

8. Currently in the **Index** view, we first get the **Settings** section in **Appsettings** and then within that section we get the individual items.

   An alternative is to reference them directly and you do that using this syntax:

```
<br />
Name: @config["Settings:Name"]
<br />
Email: @config["Settings:Email"]
<br />
```

(Do give it a try!)

The colons express the hierarchy in a way that it is possible to use this syntax in JSON-objects as well as command line arguments and environment variables.

Exercise 2.4 – User Secrets

1. To authenticate against **Azure Key Vault**, we need to add a few secrets to our application.

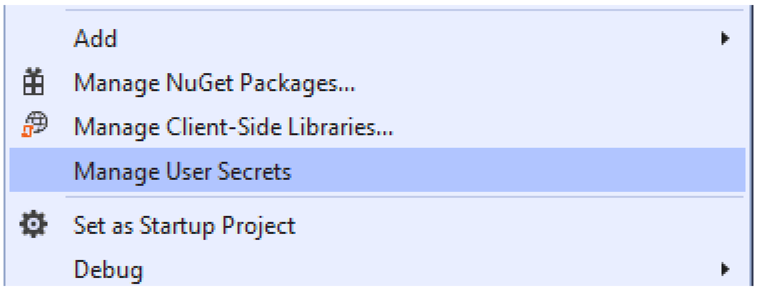
   The secrets that we need to add are found in your **\Config\azure_credentials.txt** file.

   Clearly, these are sensitive secrets, and we should not store them in our source-code, just to make sure that they don't end up on GitHub.

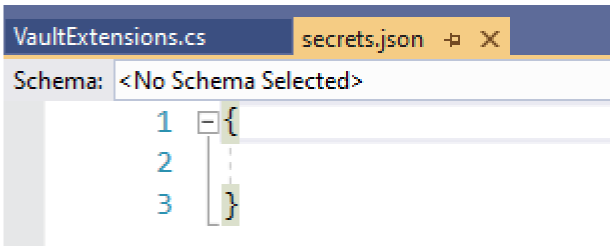
   So, where can we store them instead?

   Let's introduce **User Secrets**!

1. Right-click on the project in the **solution explorer** and select **Manage User Secrets**:

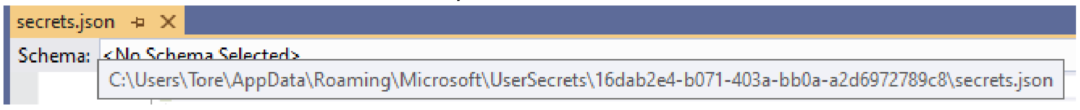


2. A new **secrets.json** file is then created for you:



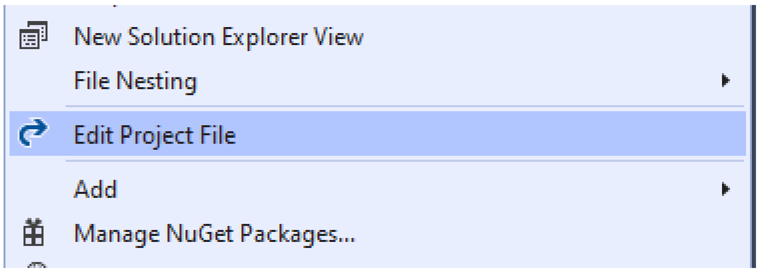
```
{
}
```

   Hover your mouse pointer over the filename and you should notice that the path to the file is located in the **c:\Users\.....** directory.



```
C:\Users\Tore\AppData\Roaming\Microsoft\UserSecrets\16dab2e4-b071-403a-bb0a-a2d6972789c8\secrets.json
```

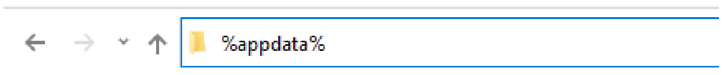
3. Open your project file by selecting **Edit Project File: (Or double-click on the project name)**



As you notice, there is now a new **UserSecretsId** entry, with an ID. This is the same ID as found in the full path to the **secrets.json** file.

`C:\Users\[Name]\AppData\Roaming\Microsoft\UserSecrets\[UserSecretsId]\secrets.json`

Open a file explorer and try to locate the file on your hard drive. Beware, some of the folders might be hidden. Or enter the **%appdata%** shortcut in Windows explorer to get to the `C:\Users\[Name]\AppData\Roaming` folder.



Open **secrets.json** it in a text editor to see what it contains. It should be the same as the one you see in Visual Studio.

4. Go back to **Visual Studio** and then open **\Starter-Kit\Exercise 2\secrets.txt** in a text editor. It contains the following:

```
{
  "Vault": {
    "Url": "https://XXXXXXX.vault.azure.net/",
    "ClientId": "xxxxxxxxxxxxxxxxxxxx",
    "TenantId": "yyyyyyyyyyyyyyyyyyyy",
    "ClientSecret": "zzzzzzzzzzzzzzzzzzzzz"
  }
}
```

5. Copy the content of that file into your **secrets.json** file in Visual Studio.

6. Do update the **Url** with the **Azure Key Vault Url** found in **\config\StudentX\Infrastructure.txt**

7. Update the **ClientId**, **TenantId** and **ClientSecret** with the data found in you **\Config\azure_credentials.txt** file.

8. To test that we can access these secrets, add the following to the **\Views\Home\Index** view:

```
<br />
Url: @config["Vault:Url"]
<br />
ClientId: @config["Vault:ClientId"]
<br />
```

You should see the secrets on the screen if all is correct. If you don't see them, you are probably in **production** mode. Do notice how we access the entries under the Vault node in **secrets.json**.

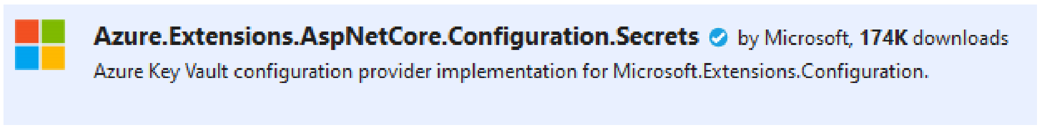
Verify that these settings are only visible when we are using the **Development** environment**.**

For all other environments (**staging**, **production…)** then settings from the **User Settings** file are not available.

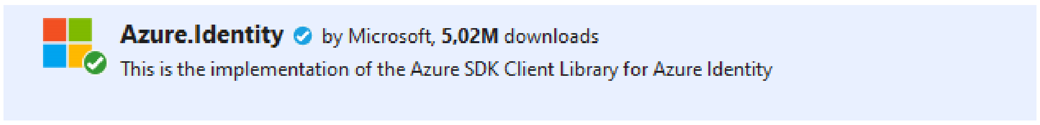
Do verify this!

Exercise 2.5 – Accessing Azure Key Vault

1. To add support for **Azure Key Vault**, we first need to add the **Azure.Extensions.AspNetCore.Configuration.Secrets** and **Azure.Identity** NuGet packages:



Azure.Extensions.AspNetCore.Configuration.Secrets by Microsoft, **174K** downloads
Azure Key Vault configuration provider implementation for Microsoft.Extensions.Configuration.

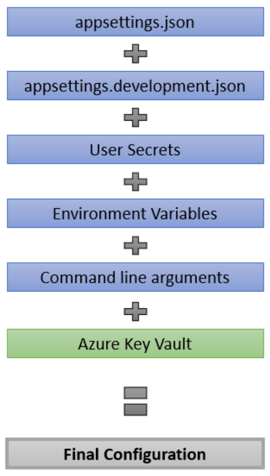


Azure.Identity by Microsoft, **5.02M** downloads
This is the implementation of the Azure SDK Client Library for Azure Identity

2. Add support for **Azure Key Vault** to the configuration system in ASP.NET Core by:
   a. Open **\Starter-kit\Exercise 2\Program.txt** and follow the instructions.

3. Start the application and the program should not crash.

   What we have done here is to inject Azure Key Vault into the list of configuration providers, like:



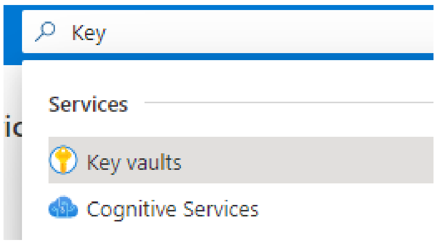
   This means that when we ask for configuration values, it will also check if the configuration is available from Azure Key Vault.

   Let's give it a try!

4. Login to Azure portal https://portal.azure.com/ using the personal login you were given in your **\config\StudentX\Infrastructure.txt** file.

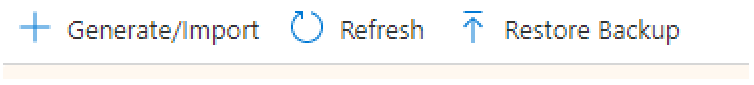
   Use a browser **incognito mode** in case you are already logged in with your work account.

5. Then search for **Key** and click on **Key Vaults**.

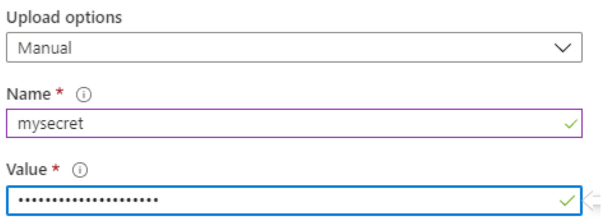


6. Then click on your personal vault, named **IdentityServerKeyVaultX**.

7. Click on **Secrets** (under Settings) and then click on **Generate/Import** to create a new secret.



   Enter a secret named **mysecret** and value of your choice:



Upload options
Manual

Name *
mysecret

Value *
••••••••••••••••••••

   Then press the **Create** button.

8. Now let’s see if we can access this secret from our test application.

   Open the **\Home\Index** view and add the following to access it:

```
mysecret: @config["mysecret"]
<br />
```

   Run the application and the secret value should be displayed!

   Awesome! We have now successfully added support for **Azure Key Vault** and we can access its stored secrets.

Beware
Secrets are cached until IConfigurationRoot.Reload() is called. Expired, disabled, and updated secrets in the key vault are not respected by the app until **Reload** is executed.

Further reading

- How to set the hosting environment in ASP.NET Core
  https://andrewlock.net/how-to-set-the-hosting-environment-in-asp-net-core/
- ASP.NET Core launchSettings.json file in Detail
  https://dotnettutorials.net/lesson/asp-net-core-launchsettings-json-file/
  Why isn't my ASP.NET Core environment-specific configuration loading?
  https://andrewlock.net/why-isnt-my-asp-net-core-environment-specific-configuration-loading/
- Safe storage of app secrets in development in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets
- Azure Key Vault Configuration Provider in ASP.NET Core
  https://docs.microsoft.com/en-us/aspnet/core/security/key-vault-configuration
- The danger of using config.AddAzureKeyVault() in .NET Core
  https://www.codit.eu/blog/the-danger-of-using-config-addazurekeyvault-in-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`

Module Exercise 8 –ASP.NET Core Data Protection

Exercise 8.1 – Exploring ASP.NET Core Data Protection

1. Start the main project.

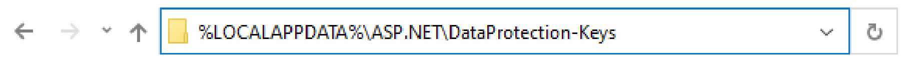
2. Open the **Client console window** and you will see various log entries from the **Microsoft.AspNetCore.DataProtection** namespace. Examine what they contain.

3. Locate the entry that says:

```
Considering key {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx} with expiration date yyyy-
yy-yy 09:08:59Z as default key.
```

4. Open a new **Windows Explorer** window and navigate to the following path:

   `%LOCALAPPDATA%\ASP.NET\DataProtection-Keys`



```
%LOCALAPPDATA%\ASP.NET\DataProtection-Keys
```

   In this directory you will find the various key rings that have been created on your computer.

5. Locate the key that was mentioned in the log, open it in **Notepad++** and try to verify the following facts:
   a. The key has a lifetime of **3 months**
   b. The key is **encrypted** using the Windows built-in DPAPI crypto API

6. Open a new **browser instance** (not a new tab) and navigate to the client application at https://localhost:5001/

7. Login as one of the users (like bob/bob).

8. Now **stop** the main project and then **delete all the key files** in the directory.

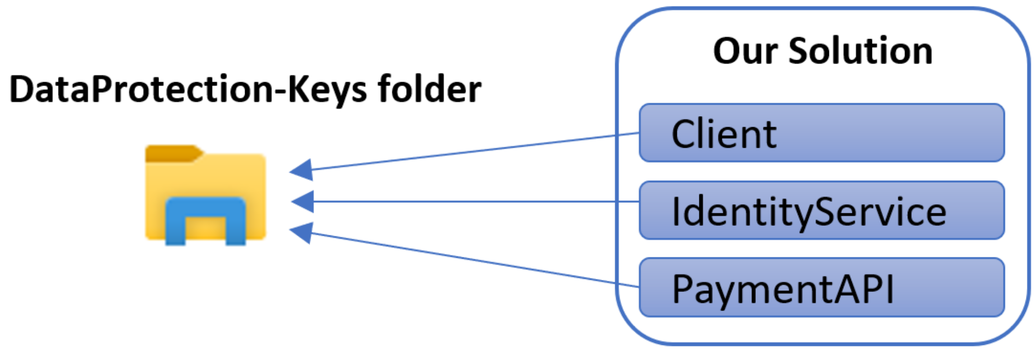
9. Start the **main project** again and you should see that new keys are created in this folder.

   But we have problems!

Exercise 8.2 – Trouble!

1. We have several issues here:
    - Which one of the three services created the key?
        - Try to locate which one of the services created the key
    - If you repeat the process, you will see that sometimes two keys are created and other times only one key is created. We have a race condition issue here!

    At startup, all three services will look at the data protection folder, and the service that happens to start up first will create the key. If two services happen to start up at the same time, then **two keys** might be created.



2. Open the browser window that you opened earlier (that you are logged in to) and then reload the page.

    First, you should see that you are logged out. Your current session cookie can't be decrypted anymore and is seen as invalid.

    In the client log window, you should also see this error:

```
Microsoft.AspNetCore.Antiforgery.DefaultAntiforgery
An exception was thrown while deserializing the token.
Microsoft.AspNetCore.Antiforgery.AntiforgeryValidationException:
    The antiforgery token could not be decrypted.
System.Security.Cryptography.CryptographicException:
    The key {89513e82-daf3-4a7d-84bd-cb96e5870bf2} was not found in the key ring.
```

    Locate it in your log file.

3. What is the solution?

    There are no good solutions here, especially if you want to share the keys across different services.

    In the main exercise that follows, we will look at one solution.

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

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`

Module Exercise 9 – Private-Public cryptography

For these exercises, you need to login to the remote machine that you find in the **\Config\Remote Desktop** folder. Choose the file with your name.

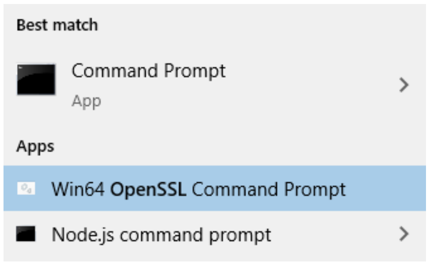
The Username is **Edument** and the password to the machine is **Edumentor123**

Exercise 9.1 – Creating RSA keys

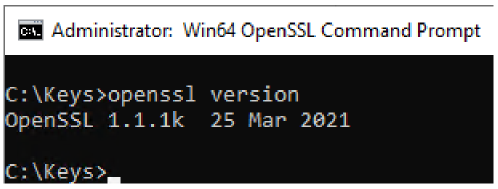
1. Login to the remote machine where we have installed **OpenSSL** for you. If you want to download and install OpenSSL yourself, then the easiest way is to use **Chocolatey** https://chocolatey.org/packages/openssl

2. Create a folder named **c:\Keys** and then open a **Win64 OpenSSL Command Prompt** and navigate to that folder.

   You can find it from the Windows start menu if you search for **openssl**:

   Then type **openssl version** to view the version details and to make sure OpenSSL is installed and working.

   ```
   C:\Keys>openssl version
   OpenSSL 1.1.1k  25 Mar 2021

   C:\Keys>_
   ```

3. Use **openssl** to create a **private RSA key** with the following specifications:
   a. **2048** bits
   b. **RSA**
   c. Name the file **rsa-private.pem**

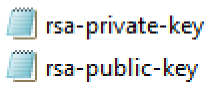
   See the presentation for details. If it asks you to "**Enter PEM pass phrase**", then enter a password that you can remember, like **tndata**.

4. Open the generated **pem** file in **Notepad++** and explore its content. This is your **private key file,** and you should keep this safe.

   A **pem** file is a base-64 encoded version of the binary **DER** file.

   a. Now, from this private key, use **openssl** to extract the **public** key from the private key.
   Name it **rsa-public.pem**

   See the presentation for details. The **public key** can always be extracted from the **private key**. You should have now created two files:


rsa-private-key
rsa-public-key

   Open the new file in **Notepad++** as well. As you notice, the **public key** is much smaller than the **private key**.

5. To print out the various parts of the private key, you can use the text option. Give it a try:

```
openssl rsa -text -noout -in rsa-private-key.pem
```

The result is the fundamental parameters that the key consists of. We won't go into the details of RSA keys here ☺

6. To view the parameters of the **public** key, run the following command:

```
openssl rsa -text -noout -pubin -in rsa-public-key.pem
```

(**pubin** is added to indicate that we are reading a public key instead of the private key).

The conclusion is that the two files contain these parts, and the shared parts are the same:

Private key	Public key
Modulus	Modulus
publicExponent	Exponent
privateExponent	
prime1	
prime2	
exponent1	
exponent2	
coefficient	

We will get back to these keys in the following exercise modules.

Exercise 9.2 – Creating ECDSA keys

1. When dealing with **elliptic cryptography** we work with various "curves".

   To see the supported curves in **openssl**, run the following command:

   ```
   openssl ecparam -list_curves
   ```

   For **OAuth2** and **OpenID-Connect** we only deal with the following set of standard curves to sign our JWT tokens:

NIST Curve	Curve name
P-256	prime256v1
P-384	secp384r1
P-521	secp521r1

   (The P-xxx refers to the standard curve notation from https://www.nist.gov/)

2. Create the following ECDSA private keys
   a. Name it **p256-private.pem**, using the **P-256** curve
   b. Name it **p384-private.pem**, using the **P-384** curve
   c. Name it **p521-private.pem**, using the **P-521** curve

   See the presentation for details or copy the command line statements from
   \Starter-kit\Exercise 9\Create ECDSA Private keys.txt

   Then open the three keys in **Notepad++** and if you compare them with your RSA keys you will notice that they are much smaller.

   Create the corresponding public keys based on the private keys. Name the files **p256-public.pem**, **p384-public.pem** and **p521-public.pem.**

   See the presentation for details or copy the command line statements from
   \Starter-kit\Exercise 9\Create ECDSA Public keys.txt.

3. Open the **public keys** in **Notepad++** and you should notice that the public keys are even smaller.

4. To print out the inner details of the private and public keys, run the following commands:

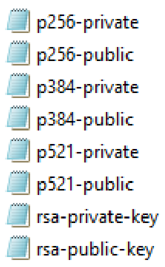
   ```
   openssl ec -text -noout -in p256-private.pem
   openssl ec -text -noout -pubin -in p256-public.pem
   ```

   As you notice, they contain different parameters compared to RSA-keys. You also notice that the **private key** also contains the **public key**.

At the end of the printout, you see the two alternative names for the curve used:

```
Public-Key: (256 bit)
pub:
    04:bc:8e:9d:a9:5d:d8:65:ed:04:01:09:92:8c:78:
    59:bf:33:a5:e1:b3:78:a8:3f:b8:bc:4a:dc:d8:64:
    28:07:b4:28:63:bc:02:58:c7:87:2b:28:91:b1:4a:
    23:d9:2f:35:3b:65:a2:1e:11:ef:af:9f:29:86:29:
    5e:2e:05:3e:44
ASN1 OID: prime256v1
NIST CURVE: P-256
```

5. You should now have created the following files:



p256-private
p256-public
p384-private
p384-public
p521-private
p521-public
rsa-private-key
rsa-public-key

Keep these files because we will use them in the following modules and, eventually, we will deploy them to IdentityServer.

Further reading

- OpenSSL Cookbook (free ebook)
  https://www.feistyduck.com/books/openssl-cookbook/
- OpenSSL - RSA key processing tool manual
  https://www.openssl.org/docs/manmaster/man1/openssl-rsa.html
- OpenSSL – EC key processing tool manual
  https://www.openssl.org/docs/manmaster/man1/openssl-ec.html
- OpenSSL – ECPARAM - EC parameter manipulation and generation tool
  https://www.openssl.org/docs/manmaster/man1/openssl-ecparam.html
- OpenSSL command cheat sheet
  https://www.freecodecamp.org/news/openssl-command-cheatsheet-b441be1e8c4a
- How to generate RSA and EC keys with OpenSSL
  https://connect2id.com/products/nimbus-jose-jwt/openssl-key-generation
- Which elliptic curve should I use?
  https://security.stackexchange.com/questions/78621
- OpenSSL – Wiki
  https://wiki.openssl.org/
- A Beginner's Guide: Private and Public Key Cryptography Deciphered
  https://medium.com/coinmonks/private-and-public-key-cryptography-explained-simply-4c374d371736
- Understanding Public Key Cryptography with Paint
  http://maths.straylight.co.uk/archives/108
- Public Keys and Private Keys in Public Key Cryptography
  https://sectigo.com/public-key-vs-private-key
- (Very) Basic Intro To Elliptic Curve Cryptography
  https://qvault.io/2020/07/21/very-basic-intro-to-elliptic-curve-cryptography/

- Symmetric and asymmetric encryption in .NET Core
  https://damienbod.com/2020/08/19/symmetric-and-asymmetric-encryption-in-net-core/
- Alice and Bob
  https://en.wikipedia.org/wiki/Alice_and_Bob
- Creating RSA Keys using OpenSSL
  https://www.scottbrady91.com/OpenSSL/Creating-RSA-Keys-using-OpenSSL

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`

Module Exercise 10 - Keys, certificates and PKCS #12

For these exercises, you need to login to the remote machine.
The Username is **Edument** and the password to the machine is **Edumentor123.**

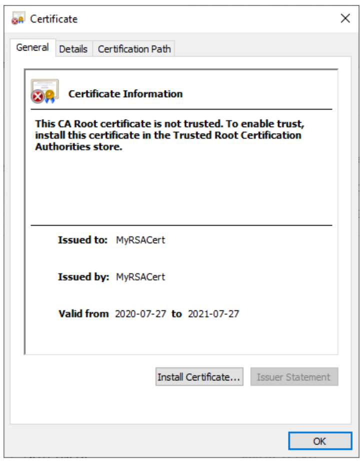
Exercise 10.1 – Creating a self-signed RSA certificate

1. Open a **command-prompt** and navigate to the **c:\Keys** folder used in the previous exercise.

2. Create a **certificate** file based on the **RSA private key** created in the previous exercise with the following specifications:
    a. Expires after **365** days
    b. Give it a **subject** in the form of **"/CN=XXXXX",** where XXXXX is a custom string of your choice. CN stands for common name. The name is not that important because we are not using it as a HTTPS certificate.
    c. Name the certificate file **rsa-cert.crt**

    See the presentation for details or copy the command line statements from:
    \Starter-kit\Exercise 10\Create RSA X509 certificate.txt.

3. Open your file explorer in Windows and **double-click** on the newly created certificate file. It should open this window:



Certificate Information

This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store.

Issued to: MyRSACert

Issued by: MyRSACert

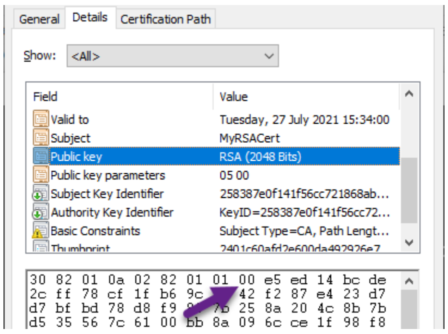
Valid from 2020-07-27 to 2021-07-27

Explore the various parameters under the **Details** tab. For this exercise we don't care so much about the details in this file.

Exercise 10.2 – Verifying the public key

1. Let's verify if the public key in the certificate window matches the public key.

2. Locate the **Public key** field in the certificate window.



General | Details | Certification Path

Show: <All>

Field	Value
Valid to	Tuesday, 27 July 2021 15:34:00
Subject	MyRSACert
Public key	RSA (2048 Bits)
Public key parameters	05 00
Subject Key Identifier	258387e0f141f56cc721868ab...
Authority Key Identifier	KeyID=258387e0f141f56cc72...
Basic Constraints	Subject Type=CA, Path Lengt...
Thumbprint	2401c60afd2e600da492926e7

```
30 82 01 0a 02 82 01 01 00 e5 ed 14 bc de
2c ff 78 cf 1f b6 9c [illegible] 42 f2 87 e4 23 d7
d7 bf bd 78 d8 f9 [illegible] 7b 25 8a 20 4c 8b 7b
d5 35 56 7c 61 00 bb 8a 09 6c ce 1f 98 f8
```

3. View the parameters of the public key using:

```
openssl rsa -text -noout -pubin -in rsa-public-key.pem
```

(We did this in the previous module.)

You should see that after a few header bytes, the numbers do match:

```
Modulus:
00:e5:ed:14:bc:de:2c:ff:78:cf:1f:b6:9c:8d:42:
f2:87:e4:23:d7:d7:bf:bd:78:d8:f9:88:7b:25:8a:
20:4c:8b:7b:d5:35:56:7c:61:00:bb:8a:09:6c:ce:
1f:98:f8:8b:67:8c:4f:cb:75:d4:cd:cb:ec:9e:6d:
...
```

Exercise 10.3 – Creating a self-signed ECDSA certificate

1. Create a certificate for the three different ECDSA private keys that we created in the previous exercise. Name them:

Private key name	Certificate name	Subject
p256-private.pem	p256-cert.crt	MyP256Cert
p384-private.pem	p384-cert.crt	MyP384Cert
p521-private.pem	p521-cert.crt	MyP521Cert

(Certificate files can also have the **.cer** extension.)

See the presentation for details or copy the command line statements from **\Starter-kit\Exercise 10\Create ECDSA X509 certificate.txt**

2. Double click on the created certificates and verify that the **public key** in the certificate file matches the **public key**.

   Use this command to view the **public** and **private** key parts of a given private key:

```
openssl ec -text -noout -in p256-private.pem
```

Exercise 10.4 – Creating the PKCS #12 file

1. Package the private key and the corresponding certificate into a PKCS #12 file, and name them as follows:

Private key name	Certificate name	PKCS #12 name
rsa-private.pem	rsa-cert.crt	rsa.pfx
p256-private.pem	p256-cert.crt	p256.pfx
p384-private.pem	p384-cert.crt	p384.pfx
p521-private.pem	p521-cert.crt	p521.pfx

   (PKCS #12 files can also have the **.p12** extension.)

   See the presentation for details or copy the command line statements from **\Starter-kit\Exercise 10\Create PKCS12 files.txt.**

2. What happens if you open the PFX file in **Notepad++**?

3. What happens if you double-click on a **.PFX** file?

4. **Copy all the keys from the remote machine to your local computer**.
   Perhaps put them in **c:\Keys** as a starting point.

   We will not use the virtual machine any more in this course.

If you have time

- Visit http://bouncycastle.org/ and read about **Bouncy Castle.**
  This is a very popular C# and Java library for working with cryptography related concepts.

Further reading:

- The PKCS Standards
  https://en.wikipedia.org/wiki/PKCS
- Openssl **pkcs12** utility guide
  https://www.openssl.org/docs/manmaster/man1/pkcs12.html
- Openssl **req** utility guide
  https://www.openssl.org/docs/manmaster/man1/openssl-req.html

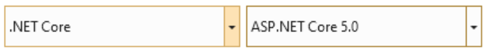
- .NET Core Cryptography breaking changes
  https://docs.microsoft.com/en-us/dotnet/core/compatibility/cryptography
- What is a Pem file and how does it differ from other OpenSSL Generated Key File Formats?
  https://serverfault.com/questions/9708
- How to create a signing certificate and use it in IdentityServer4 in production?
  https://stackoverflow.com/questions/58136779/how-to-create-a-signing-certificate-and-use-it-in-identityserver4-in-production
- What is the difference between X509Certificate2 and X509Certificate in .NET?
  https://stackoverflow.com/questions/1182612
- ASP.NET Core Certificate Manager
  A package which makes it easy to create certificates which can be used in client server authentication and IoT Devices like Azure IoT Hub.
  https://github.com/damienbod/AspNetCoreCertificates
- Cryptography Improvements in .NET 5 - Support for PEM
  https://www.tpeczek.com/2020/12/cryptography-improvements-in-net-5.html

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`

Module Exercise 13 – Users

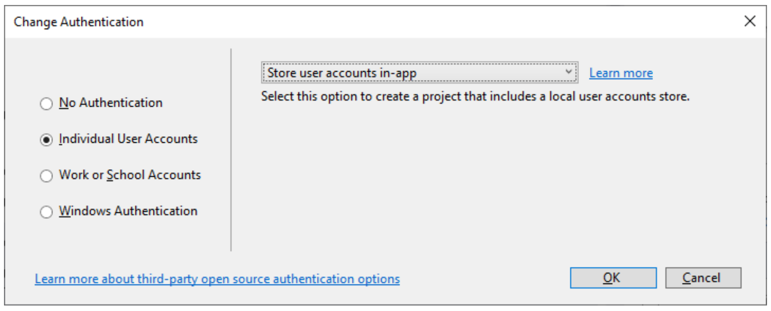
Exercise 13.1 – Creating the User Management Service

1. **If you are using Visual Studio 2019**
   a. Create a new Visual Studio Project with the following properties:
      i. Template: **ASP.NET Core Web Application**
      ii. Name: **User Management Service**
      iii. Framework: **ASP.NET Core 5.0**



.NET Core | ASP.NET Core 5.0

      iv. Template: **Web Application MVC**
      v. Authentication: **Individual User Accounts** (Store user accounts in-app)



Change Authentication

No Authentication
[✓ Individual User Accounts]
Work or School Accounts
Windows Authentication

Store user accounts in-app | Learn more

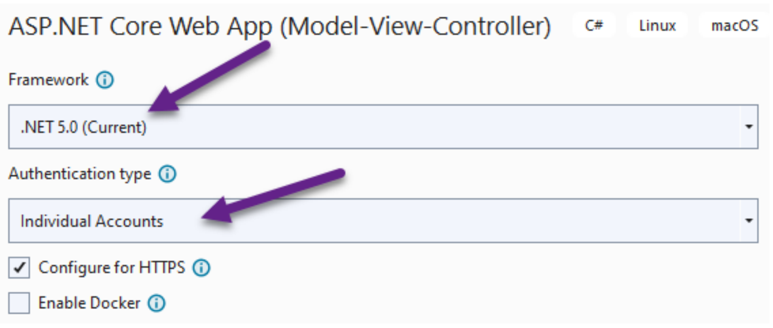
Select this option to create a project that includes a local user accounts store.

Learn more about third-party open source authentication options

OK | Cancel

If you are using Visual Studio 2022

   a. Create a new Visual Studio Project with the following properties:
      i. Project type: **ASP.NET Core App (MVC)**
      ii. Name: **User Management Service**
      iii. Framework: **.NET 5**
      iv. Authentication: **Individual User Accounts**



ASP.NET Core Web App (Model-View-Controller) C# Linux macOS

Framework: .NET 5.0 (Current)

Authentication type: Individual Accounts

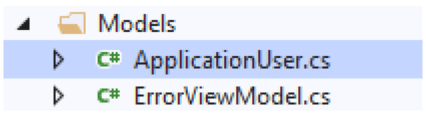
[✓] Configure for HTTPS

[ ] Enable Docker

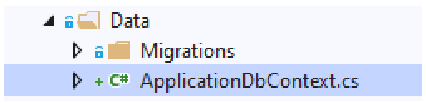
      v. Click on **Create**

2. Open the **appsettings.json** file and update the **DefaultConnection** with the connection string to your Azure SQL Server.

3. To seed the database with initial users, do the following:
   a. Add the folder **\Starter-Kit\Exercise 13\Module exercise\ApplicationUser.cs** to the **Models** folder.



b. Replace the **\Data\ApplicationDbContext** class with the one found in **\Starter-Kit\Exercise 13\ApplicationDbContext.cs**



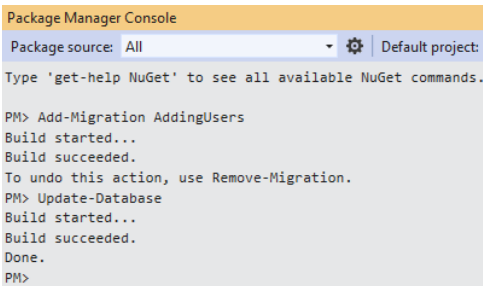
4. Fix the missing using statements so that the code compiles.

   Then we need to relax the password requirements by disabling most of the requirements. Open **\Starter-Kit\Exercise 13\Module exercise\Startup.txt** and follow the instructions.

5. To create the ASP.NET Identity tables in our database and seed it with initial user data, run the following command from the **NuGet Package Manager Console:** (Found under **Tools -> NuGet Package Manager**)

```
Add-Migration AddingUsers

Update-Database
```

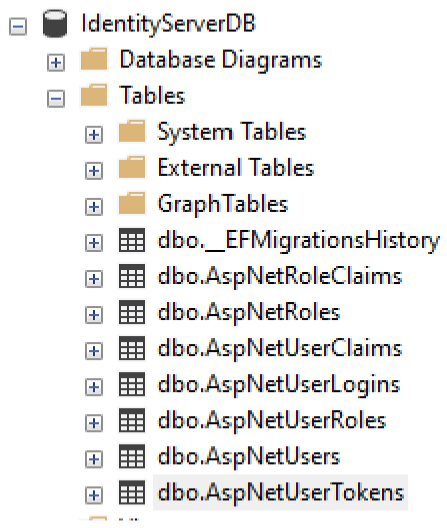
It should look like this:



```
Type 'get-help NuGet' to see all available NuGet commands.

PM> Add-Migration AddingUsers
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
Done.
PM>
```

6. Now the migration should have created the necessary **tables** and added the **users** as described in the **UsersSeed** method.

   Open **SQL Server Management Studio** and verify this. Also verify that you find the **users** and **claims** in the relevant tables.



```
⊟ IdentityServerDB
   ⊞ Database Diagrams
   ⊟ Tables
      ⊞ System Tables
      ⊞ External Tables
      ⊞ GraphTables
      ⊞ dbo.__EFMigrationsHistory
      ⊞ dbo.AspNetRoleClaims
      ⊞ dbo.AspNetRoles
      ⊞ dbo.AspNetUserClaims
      ⊞ dbo.AspNetUserLogins
      ⊞ dbo.AspNetUserRoles
      ⊞ dbo.AspNetUsers
      ⊞ dbo.AspNetUserTokens
```

Important
Trying to login with the two users we just created will not work in this application due to the default setup of ASP.NET Identity in this project.

7. **Close the project and then continue with the main exercise where you will add support for the database and ASP.NET Identity.**

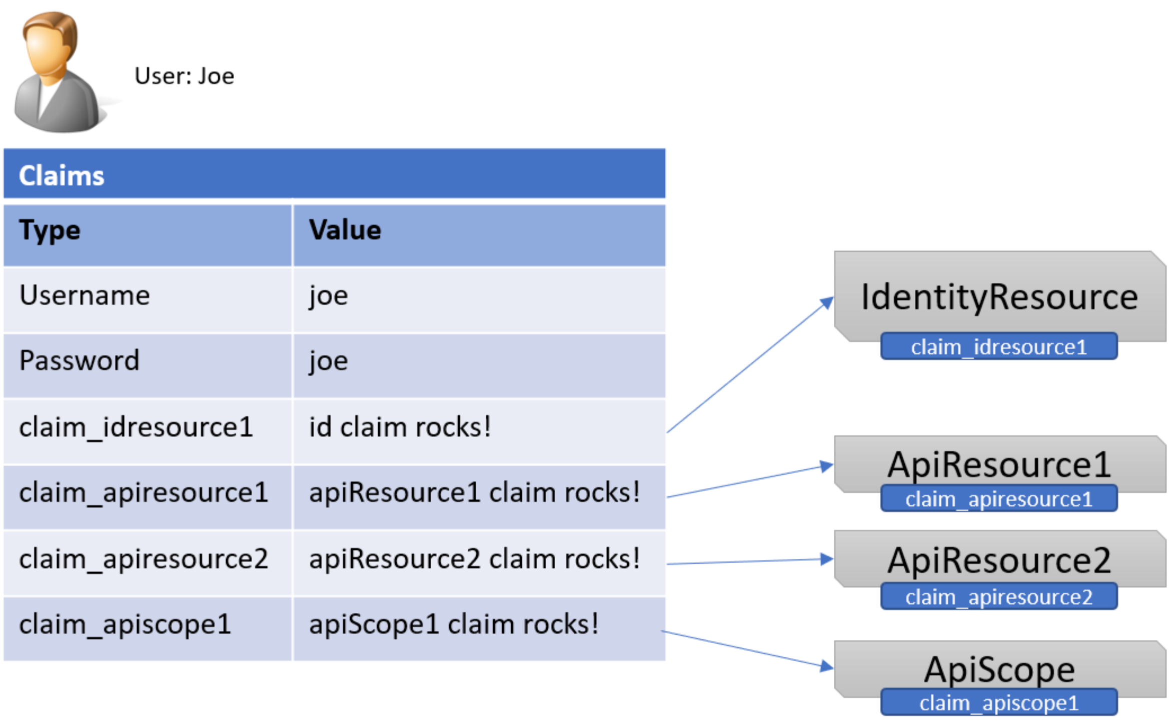
   We now have our ASP.NET Core identity tables created and seeded with some some data.

Module Exercise 14 – Tokens and claims

Exercise 14.1 – Exploring the claims

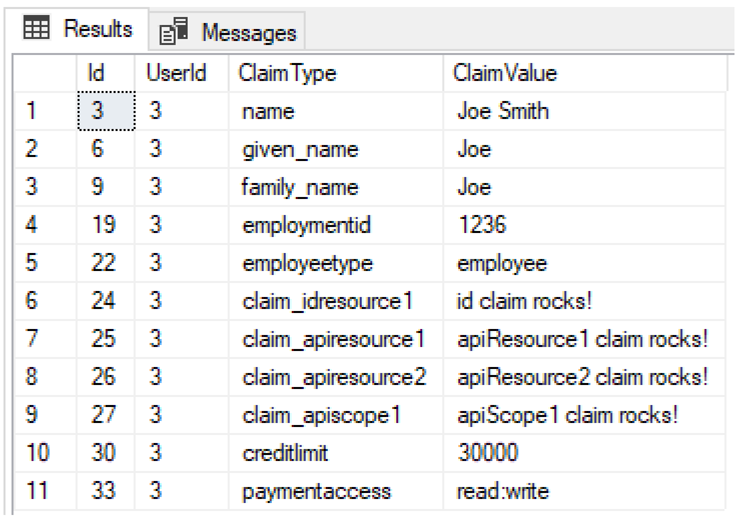
In this exercise we will explore which claims go where, by setting up the following scenario:

User **Joe** currently has the following claims (and more) associated with it in the SQL database and we have defined the resources to include one claim each:



Claims	
Type	**Value**
Username	joe
Password	joe
claim_idresource1	id claim rocks!
claim_apiresource1	apiResource1 claim rocks!
claim_apiresource2	apiResource2 claim rocks!
claim_apiscope1	apiScope1 claim rocks!

The claims can be found in the database:



	Id	UserId	ClaimType	ClaimValue
1	3	3	name	Joe Smith
2	6	3	given_name	Joe
3	9	3	family_name	Joe
4	19	3	employmentid	1236
5	22	3	employeetype	employee
6	24	3	claim_idresource1	id claim rocks!
7	25	3	claim_apiresource1	apiResource1 claim rocks!
8	26	3	claim_apiresource2	apiResource2 claim rocks!
9	27	3	claim_apiscope1	apiScope1 claim rocks!
10	30	3	creditlimit	30000
11	33	3	paymentaccess	read:write

1. First, we will make a few changes to the **authcodeflowclient_dev** client in the **IdentityService.Configuration** project:

    a. Change the **AllowedGrantTypes** so that this client supports both the **authorization code flow** and the **client credentials flow** by setting it to:

    ```
    AllowedGrantTypes = GrantTypes.CodeAndClientCredentials,
    ```

    b. Add the following **AllowedScopes**:

    ```
          "idresource1",
          "apiscope1"
    ```

    c. Add the following client settings:

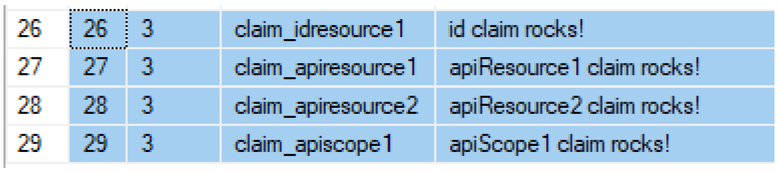
    ```
    AlwaysSendClientClaims = false,
    ClientClaimsPrefix = "client_",

    Claims = new List<ClientClaim>()
    {
        new ClientClaim("client-claim1", "clientvalue")
    },
    ```

    d. Add the following **RedirectUris**:

    ```
    https://localhost:8001/authcode/callback
    ```

2. Locate the **\IdentityService.Configuration\Resource** folder and then open the **\Starter-Kit\Exercise 14\Resources.txt** file and apply the changes to the resources.

3. If you look at the end of the **AspNetUserClaims** table in the database, you should see that user Joe (UserID=3) has the following claims (and more) defined:



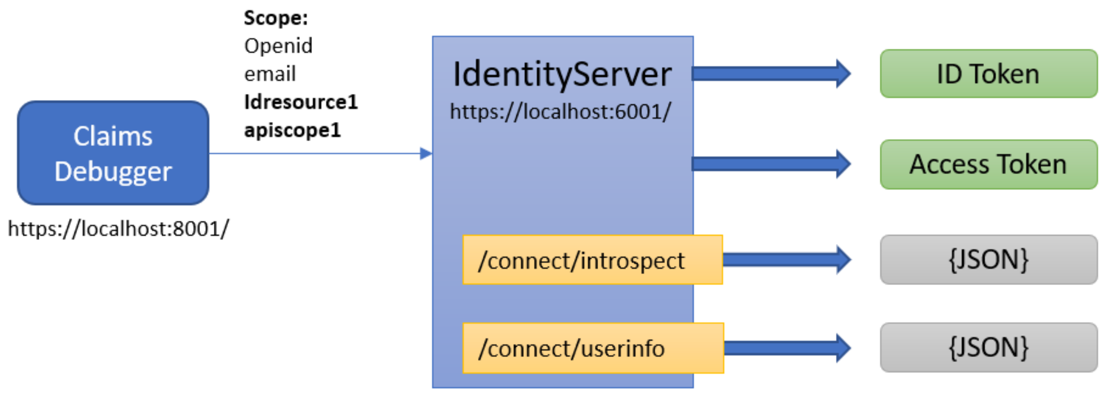
26	26	3	claim_idresource1	id claim rocks!
27	27	3	claim_apiresource1	apiResource1 claim rocks!
28	28	3	claim_apiresource2	apiResource2 claim rocks!
29	29	3	claim_apiscope1	apiScope1 claim rocks!

4. Start the main project.

Exercise 14.2 – The claims debugger

1. Now locate and open the **ClaimsDebugger** found in the **\Starter-Kit\Exercise 14** folder.

   The project will "manually" authenticate against **IdentityServer** and display what is available in the tokens and from the two endpoints.

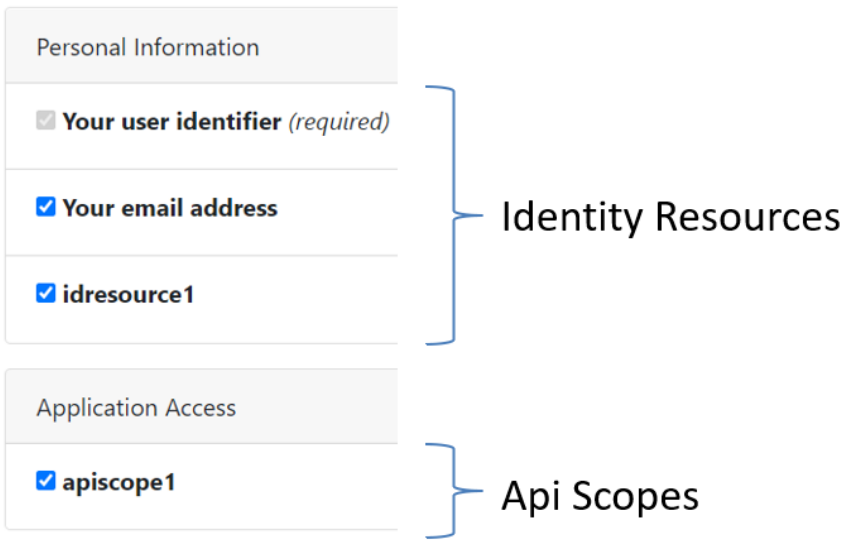


2. Locate the **AuthCodeController** and look at what **scopes** it will ask for.

3. Start the project and login with the **Login with authorization code flow.**

   Login with the user **joe** and password **joe**.

   When you consent to the permissions, you see that it asks for the **openid**, **email**, **idresource1** and the **apiscope1**:



4. Give consent and then explore the result that is displayed.

   ID-Token

   - This token does not contain any **user-claims** at the moment. This is because this setting is set to false in the client definition:

     ```
     AlwaysIncludeUserClaimsInIdToken = false,
     ```

   Access-token

   - Explore the **audience** (aud) and the **user-claims** that are included.
   - The **claim_idresource1** claim is not included in this token.

   User-info

   - This endpoint contains the identity-related claims, including the **claim_idresource1** claim.
   - **Email** is also included because we asked for that scope.

   Token Introspection

   - This endpoint is only called by API resources, and you can see how that is done in the **GetTokenIntrospectionData()** method in the debugger.

     The API must authenticate using a separate API **clientId/secret** defined in the **ApiResource**.
   - The **access token** is also included in this request.
   - It contains the same **user claims** as the **access token.**

5. Open the **client** definition in the main project. Currently these two settings are set to false:

   ```
   AlwaysIncludeUserClaimsInIdToken = false,
   AlwaysSendClientClaims = false,
   ```

   Turn them on (both, or just one at the time) and restart both solutions.

   AlwaysSendClientClaims
   - Setting this to true will include the client claim in both the **access token** and **userinfo.**

     Notice that the client claim is prefixed with **client_** to better separate them. This can be controlled using the **ClientClaimsPrefix** client setting.

   AlwaysIncludeUserClaimsInIdToken

   - When true, the **user claims** that are normally included in the **userinfo** endpoint are also included in the **ID-token** (email, claim_idresource1, email_verified).

Exercise 14.3 – The client credentials flow

1. Restart the claims debugger and login using the **Login using client credentials** flow instead.

   Some of the things to observe are:

   - The **ID-Token** and **userinfo** endpoint are not used in this flow, because there is no user involved.
   - The **claim_apiresource1, claim_apiresource2, claim_apiscope1** are not included. Because still we have no **user** involved here!
   - The client claim **client_client-claim1** is included in the access token and from the token introspection endpoint.
   - The **AlwaysSendClientClaims** and **AlwaysIncludeUserClaimsInIdToken** client settings does not have any purpose in this flow.
   - Even though we asked for the **apiscope1**, we get the audience to be **apiresource1** and **apiresource2** as expected. This is the same as we got in the previous exercise.

2. Hopefully, this tool can give some clarity to what claim goes where and how the various configurations affects this.

3. One last thing, in this exercise we only make a request for **apiscope1**. Can you make a request for these scopes: "**openid email idresource1 apiscope1**"?

   Update the **ClientCredentialsController** class in the **claims debugger** and give it a try.

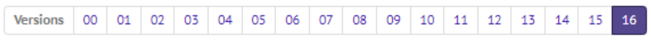
Exercise 14.4 – Security best practice

1. How should we create a secure OpenID Connect solution?

   A good starting point is to review the **OAuth 2.0 Security Best Current Practice** document at:

   https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/

   This document is regularly updated with the latest best practices and you can see the version history on this page:

   Versions 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16

   Review the document and see what it contains.

2. Other related best practice documents are:

   - JSON Web Token Best Current Practices
     https://datatracker.ietf.org/doc/rfc8725/
   - OAuth 2.0 for Browser-Based Apps
     https://datatracker.ietf.org/doc/draft-ietf-oauth-browser-based-apps/
   - The OAuth 2.1 Authorization Framework
     https://tools.ietf.org/html/draft-ietf-oauth-v2-1-00

   Feel free to review and see what they contain and cover!