Cross-platform user authentication and authorization through a custom identity server

Eduard Campo Raurich

Resum - Les aplicacions mòbil en entorns industrials normalment es veuen velles i desactualitzades, això és normalment per culpa de que l'objectiu és que la aplicació sigui funcional i prou. Aquest projecte s'ha desenvolupat amb la col·laboració d'una companyia que ofereix solucions de software per la optimització de la cadena de subministrament d'empreses de tot tipus. El meu objectiu de millorar el component d'identificació dels usuaris de les noves solucions, reduint significantment el nombre de vegades que l'usuari ha d'introduir les seves credencials a l'aplicació. La meva feina és trobar la millor manera de muntar un Identity Server que serveixi com a únic punt d'inici de sessió per a totes les aplicacions de la companyia i que segueixi un protocol de seguretat ja establert. Això serà provat en una aplicació mòbil multiplataforma desenvolupada amb Xamarin, amb ajuda també de Xamarin.Forms i la novedosa manera de crear interfícies que proporciona.

Paraules Clau - Identity Server, Multiplataforma, App Mòbil, Desenvolupament del Software, Xamarin, Xamarin Forms, ASP.NET, OAuth 2.0, Token

Abstract - Mobile applications in industrial environments usually look and feel outdated, and that's usually because all the effort is put into making it be straight up functional. This project has been developed in collaboration with a company that provides mobile applications that improve the supply chains of many industries. I have been tasked to enhance the user experience around the authentication component of the new solutions. The main goal is to reduce the times a user is requested his credentials significantly. My job is to research the best way to set up an identity server that allows for single sign on throughout all the company's applications and follows an established security protocol. This will be tested on a cross-platform mobile application developed with Xamarin, which will take advantage of the great new interface approach that Xamarin Forms offers.

Index terms - Identity Server, Cross-Platform, Mobile App, Software Development, Xamarin, Xamarin Forms, ASP.NET, OAuth 2.0, Token

———————————— o ————————————

1 Introduction

This project consists on the development of a customized cross-platform[1] authentication system that follows the OAuth2.0 standard security protocol. I will take a look into this protocol and make sure we do not miss any key components that the system need. It is mandatory that I standardize the system to be reused in each and every upcoming software application the company develops.

The current solutions that my company distributes do in fact have a user authentication system. However, any user is now able to access each and every feature that the app offers. This is where the authorization part of the module should come in and set up some kind of feature access control. Furthermore, each and every platform that executes such login has been developed separately with different programming languages and frameworks. And last but not least, every product has a different server where the user authenticates, and the idea would be to have a single server that handles each and every login. This current approach surely makes maintainability and upgradeability of the line of products pretty difficult.

Additionally, a cross-platform approach would increase the amount of shared code in the client application significantly. Likewise the user login service should be reusable on every new and upcoming solution. Also, this could greatly reduce development costs from this point forward, which could allow the company to potentially reduce the pricing of the sellable final products.

Further in this article we will firstly take a look at the OAuth2.0 security protocol together with Identity Server 4. Secondly, we are going to define the project's objectives, a working methodology and the tools to be used throughout. We will then take a look at the requirements and the main development section will follow. Finally, a conclusion of the project is in order.

2 The OAuth 2.0 Protocol

OAuth 2.0 is the industry-standard protocol for authorization. OAuth 2.0 supersedes the work done on the original OAuth protocol created in 2006. This protocol focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices. OAuth defines four main roles which will be heavily referenced in this document:

- Resource owner: an entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user. The resource owner is meant to communicate exclusively with the Client.
- Resource server: The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
- Client: an application making protected resource requests on behalf of the resource owner and with its authorization. This application could be executing on a server, desktop, mobile device...
- Authorization server: the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization. Commonly called Identity Server.

2.1 Tokens

The main idea behind OAuth 2.0 is the use of tokens. A token is an unintelligible array of characters that contain some kind of information. There are many kinds of tokens, but this project requires only the usage of access and refresh tokens.

2.1.1 Access Token

Instead of using stored credentials (user, password) every time the Client needs to access a resource, this token allows for a more secure authorization. Even though this kind of token is encrypted as a JWT[2], the Authorization Server must sign it using a certificate. An access token is really just a list of claims, which contain 2 strings each: "type" and "value". A claim should not be confused with a programming dictionary, it is a misconception to think that an access token can't have claims with repeated types or values, but that is definitely not the case. A token contains lots of claims, but to give you an idea of how the claims look here's an example list:

{ "Role", "UserManager" }
{ "Role", "Driver" }
{ "Access1", "Granted"}
{ "Access3", "Granted"}
{ "Expiration", "1524584430"}

2.1.2 Refresh Token

It is common practice to send a refresh token together with an access token when a request is received. A refresh token allows the application to request the authorization server to issue a new access token directly, without having to re-authenticate the user. This works as long as the refresh token has not been revoked or hasn't expired. The refresh token does not contain claims, and can only be read by the Authorization Server who issued it.

Table1: Differences between tokens

Token type	Access	Refresh
Expiration	1 hour	15 days
Contains claims	Yes	No
Used on	Resource Server	Auth Server
Stored on	Client	Client and Auth Server

2.2 Main Flow

The basic OAuth2.0 flows work like this:

- The Client asks permissions to request tokens to the Authorization Server (client authorization).
- The Client to request tokens to the IS on behalf of the User (login).
- The Client requests resources to the Resource Server with the User's token (Api call).
- The Client requests and extension of his session to the Authorization Server.
- The Client requests the Authorization Server to revoke all the User's tokens (logout).

The figure **A1** in the appendix displays a basic scheme of all the components and their respective connections. These and more OAuth2.0 flows are detailed further below in this article.

3 State of the Art

Many internet giants offer their own identity server implementations. Each and everyone of them has its ups and downs. Should you not be happy with the premade OAuth2.0 compliant options around, some tools are available for free to setup and customize your own identity server.

3.1 Google's Identity Server

Google, for instance, does in fact have an identity provider. It allows for access and refresh tokens to be requested by the users. If Google grants you an access token, you may access any available resource with it instead of having to use your credentials every time. Due to its popularity, Google allows 3rd party applications to use his login identity server. As long as you have a Google account, you do not need to share your username and password to those 3rd party services: you can just send them your access token.

Even though what Google offers is pretty appealing, you still need an account on their site in order to use its service. Should you wish to use their identity server and customize it for your business, you would need each and every user to own a GSuite[3] account, which needs to be paid monthly. Forcing every user to create a Google account under your domain can be a bit of an inconvenience.

3.2 Amazon's Identity Server

Another identity provider you can find around is Amazon. Its biggest downside is that you can only grant access and setup authorization parameters for applications deployed on Amazon Web Services. You cannot deploy any application elsewhere or you will not be able to control its access with Amazon's identity server. This is perfect if you were already planning to use AWS exclusively, otherwise this option will not be of any use to you. This is exactly what the company was willing to avoid, it is mandatory to be able to use any service provider that fits every single application's needs.

3.3 IdentityServer4 (IS4)

IdentityServer4 is a free, open source OAuth 2.0 framework for ASP.NET Core under the permissive Apache 2 license [4]. It provides another layer of functionality that will allow for an easier identity server setup. In OAuth 2.0 general terms, IS4 plays the role of authorization server.

IS4 incorporates all the protocol implementations and extensibility points needed to integrate token-based authentication, single-sign-on and API access control in your applications. IdentityServer4 is officially certified by the OpenID Foundation and thus spec-compliant and interoperable. This framework allows for the development of a totally custom identity server that accommodates the needs of the company.

4 Objectives

The main goal of this project is to develop a proof of concept that shows how the user experience around the authentication component of industrial mobile applications can be improved.

· OBJ-00 - Get to know the OAuth 2.0 security protocol and Identity Server 4

- Read the OAuth 2.0 documentation
- Try the Identity Server 4 demos

· OBJ-01 - Customize and setup an Authorization Server capable of providing access and refresh token to multiple Clients

- Set up a new Identity Server 4 instance
- Customize its features
- Design a database to store Users and Claims
- Feed the database to the server

· OBJ-02 - Develop a cross-platform Client with an interface that eases the testing process.

- Learn Xamarin, its design patterns and best practices
- Create connections to the identity server
- Mock connections to the resource server
- Design and develop a testing interface

· OBJ-03 - Create and deploy a Resource Server that requires signed access tokens to serve resources.

- Set up a new Web Api
- Add a custom authorization validation
- Develop simple calls that validate claims

· OBJ-04 - Test all the components and ensure they are all working as intended.

A more detailed Gantt Diagram can be found at the end of this document, in the Appendix section.

5 Methodology and Tools

The traditional Waterfall software development methodology may seem like a bold approach but in my case it was the perfect option. After defining the requirements there was not much room for change. After all, there's an established protocol to follow if you want to set up OAuth2.0 - and this proof of concept did not expect any additional features. A little diagram of the waterfall methodology steps can be found below in Figure 1.

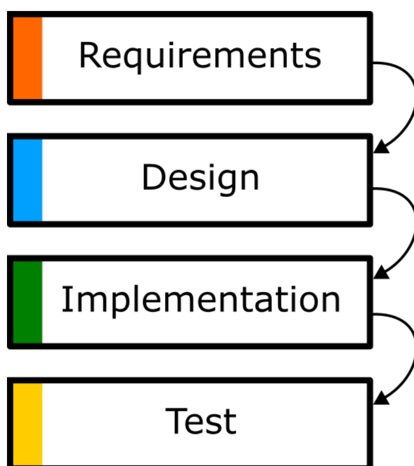


Figure 1: Waterfall development methodology

However there is a small little twist in my methods that may be worth noting. Each and every component developed (Client, IS4 and Api) is useless by itself. For the entire flow to work, I must have everything up and running simultaneously. This meant that for every component I first had a design phase and immediately after the implementation for it. Designing everything in one step would have been overwhelming for me, the latest implementations were bound to have too many problems.

5.1 Design principles

In college we have seen software engineering design principles and evaluative patterns now and again. Here's a list of the ones I consider more important and I set out to follow during the development of this project:

- **Single responsibility principle:** a class should have only a single responsibility and keep operating while the rest of the software changes.
- **Open/closed principle:** software entities should be open for extension but closed for modification.
- **Interface segregation principle:** many client specific interfaces are better than one general purpose interface
- **Dependency inversion principle:** one should depend upon abstractions (interfaces), and not concretions (classes).
- **High cohesion evaluative pattern:** attempts to keep objects appropriately focused, manageable and understandable. Usually paired with low coupling.
- **Low coupling evaluative pattern:** dictates how to assign responsibilities to support lower dependency between the classes, change in one class having lower impact on other classes and higher reuse potential - with focus on that last one.

5.2 Tools

A great amount of tools are needed for a project of this size. Both developing tools and specific frameworks need to be carefully selected to ensure an agile and nimble development.

5.2.1 Frameworks

A software framework is an abstraction that helps to develop software. It can be extended but should not be modified. Below are the ones I planned to use during my development.

- Xamarin

The Client side of the module was programmed from scratch using Microsoft's cross platform development framework Xamarin. The company encouraged that I use Xamarin to develop the first version of the Client.

Based on the .net framework, Xamarin makes the business logic C# code reusable for all platforms. In addition, the new Xamarin Forms library provides the perfect approach possible for building cross-platform interfaces. That is to say, this recent addition to Xamarin will allow this solution to be almost entirely made out of shared code. Only hardware interactions from the app will need to be written individually for each target platform, which might not even be needed in this particular development. Below you can find a simple graphical explanation showing what Xamarin Forms is capable of.

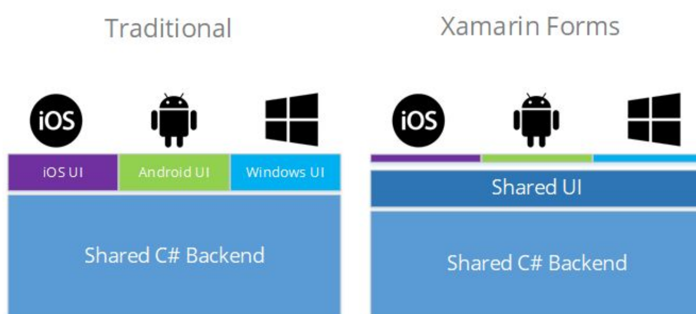


Figure 2: Xamarin Forms and the old UI approach

- IdentityServer 4

IdentityServer 4 is definitely the best option for the company's needs. Not only will this implementation avoid an unnecessary amount of accounts on an external user database but we also won't be restrained to deploy our apps and services anywhere.

- ASP.NET

ASP.NET is an open-source server-side web application framework designed for web development to produce dynamic web pages. It was developed by Microsoft to allow programmers to build dynamic web sites, web applications and web services. ASP.NET will be the framework used to develop the web Api that needs to verify and control received access tokens. After all, even IS4 runs on ASP.NET.

5.2.2 Development tools

The software I chose to develop the project with are pretty commonly used nowadays. Luckily, I only needed 2 even though this project was pretty big.

- Visual Studio Community 2017

The free version of Microsoft's Visual Studio for both individuals and organizations. Visual Studio is the integrated development environment that works fantastically with Xamarin and .net standard. All projects could be edited and deployed with Visual Studio, with the exception of the database.

- SQL Server Management Studio

Microsoft's solution to design, create and manage databases is insanely powerful. I am familiar with its interface, so I should be able to get done what little I need quickly.

5.2.3 Management tools

No software engineering process can be complete without management tools, so this was no exception. It is my goal, after all, to enhance the skills I have acquired throughout these past years studying.

- Trello

Trello[5] helped me a lot with my organization. It proves pretty useful even while working alone, allowing you to have your requirements in an online board accessible from any device. Trello ensured I never missed a thing while both designing and implementing the necessary components.

- Github and Sourcetree

Github is a web-based hosting service for version control using git[6]. It allowed me to save my development progress somewhere I can always access it should I ever encounter a problem with my computer. Sourcetree is just a git client that offers an easy to use interface to control your repositories.

6. Requirements

I compiled a list of both functional and nonfunctional requirements. Some nonfunctional requirements may seem lowballed, but this proof of concept's were not too strict. The four systems referred here are the Client, the Resource Server, the Authorization Server and the SQL Database. See Table 2 for functional requirements and Table 3 for non-functional ones.

Table 2: Functional Requirements

OBJ	FR	Functional Requirements
OBJ 01	FR 101	The Authorization Server must retrieve the User and claims list from a local database
	FR 102	The Authorization Server must send refresh tokens along with access tokens
	FR 103	The Authorization Server should sign access tokens with a test development certificate
	FR 104	The SQL Database must store the user credentials along with his claims
OBJ 02	FR 201	The Client must implement all the required IdentityServer4 connection flows (ROWP)
	FR 202	The Client must securely store the tokens and save them even if the device is rebooted
	FR 203	The Client must have a test interface that allows to check all the app's features
	FR 204	The Client should display text feedback to easily track every kind of error found
OBJ 03	FR 301	The Resource Server should require a signed access token proceed to its validation functions
	FR 302	The Resource Server must be able to decrypt and read the access' token claims
	FR 303	The Resource Server should validate at least if the token received has expired or not

Table 3: Non-Functional Requirements

OBJ	NFR	Non-Functional Requirements
OBJ 00	NFR 001	The Authorization Server must be customized from scratch using IdentityServer4
	NFR 002	The Authorization Server must follow the OAuth 2.0 security protocol
OBJ 01	NFR 101	The Authorization Server should handle at least 10 Clients requesting, refreshing or revoking tokens simultaneously
	NFR 102	The Authorization Server must not take more than 10 seconds to answer the Client
OBJ 02	NFR 201	The Client has to be executed on any modern UWP[7], iOS and Android using the Xamarin framework for development
	NFR 202	The Client module must be easily copied and reused on any other application
	NFR 203	The Client must have a low timeout on its server requests, no greater than 15 seconds
	FR 204	The Client must follow the OAuth 2.0 security protocol
	FR 205	The Client must use the Resource Owner Password grant type to request tokens
OBJ 03	NFR 301	The Resource Server must handle at least 5 Clients requesting resources simultaneously

6.1 The ResourceOwnerPassword grant (ROWP)

IS4 supports numerous grant types[8], which are a way to specify how a client wants to interact with IdentityServer. In this project only the ResourceOwnerPassword grant type is used, being this the best approach considering the needs of the company's new and upcoming applications. Bear in mind that this type of grant is not recommended if you are planning for 3rd party apps to connect to the IS, nor if your login is on a web app. In those cases implementing the hybrid grant type would be wiser.
As the name implies, the ResourceOwnerPassword grant requires the name and a password of the Resource Owner (end user) in order to request tokens to the Authorization Server. By tokens, I mean both an access and a refresh token.

Special pre-defined IS4 flows must be followed to request, use and revoke tokens. A detailed list of the implementations needed follows in this section. Check out A1 in the appendix section to familiarize yourself with the actors and objects used for these flows (already detailed on section 2. The OAuth 2.0 Protocol).

6.1.1 Client authorization request flow

This flow covers the token authorization request from the Client (on behalf of the User) to the Authorization Server. Accessing the Requesting tokens with any grant type requires an authorized Client. Otherwise a

malicious Client with the server's URL would easily be able to brute force a token request.

The IS has a list of authorized Clients which can interact with its token endpoint. This list contains a ClientName and ClientSecret for each Client. They are strings that should be kept secure on both ends, and they act like a "User/Pass" combination but for Clients instead of the Resource Owners. When an IS4 is asked for Token Authorization with the right Client credentials, it allows for the request, refresh and revocation of tokens for a limited time. Check out image A2 in the appendix sketches how this first flow works.

In the case that the wrong Client credentials are provided, the IS simply won't allow any further connections made from that Client.

6.1.2 Login flow

This flow covers the access and refresh token request from the Client (on behalf of the User) to the Authorization Server. After successfully being granted access to the IS, the ROWP grant type itself begins.

The Resource Owner (in this case a User) provides his Username and Password to the Client, who follows up and requests the tokens to the IS. Should those credentials coincide with those stored in the IS database, an access token is signed with a corporate certificate and then issued together with a refresh token (unsigned) to the Client who securely stores them ready to be used when needed. The issued refresh token is also stored in the IS4 cache memory. This enables for the access token to be refreshed should it expire before logout. The figure A3 in the appendix sketches the flow.

If invalid credentials are being provided to the IS, an "Invalid Grant" 400 error is returned to the Client. This ends the current connection but does not revoke the Client token authorization received in the previous flow, allowing it to send the credentials again.

6.1.3 Resource access request flow

This flow covers the resource access request from the Client (on behalf of the User) to the Resource Server. The Resource Server will refuse any connection done by a tokenless HttpClient. It is mandatory to send an access token with every request for the server to validate. A Resource Server has lots of options to ensure the token received has not been forged and is valid. When decrypted, all its information can be read and checked:

- Issuer: the address of the IS which issued the token.
- Scope: the Client's allowed api access. Every User that gets an access token through a specific Client will have the same scope.
- Expiration: the token expiration date and time must be compared with the current and time to validate the token.
- Audiences: list containing, at least, the name of the Resource Server trying to access. Can contain the name of other Resource Servers the User is allowed to request resources from.
- Custom Claims: the controllers have access to the entire claim list which can be customized and controlled.

If the token makes it through all these validations, the Resource Server proceeds to fulfill the request received (GET, POST…) and serve the necessary resources back to the Client. This validation is done every time the Resource Server receives a request.

There is also the case where the token does not go through this validation successfully. The Resource Server simply replies with a response 401 Unauthorized immediately when something is wrong with the token. In order to avoid giving out hints to the Client, this generic 401 status is sent instead of a specific error message. The Client has the option to try and refresh the access token with a refresh token, in order to perform a second attempt to retrieve the request successfully. This is logical when the token has expired, but might not be useful if that was not the validation step it failed to go through. The access token should not be deleted in this last case, for it might be usable on another Resource Server that you would have access to.

6.1.4 Token refresh flow

This flow covers the access token refresh request from the Client (on behalf of the User) to the Authorization Server through a refresh token. Refresh token allows for new access tokens to be obtained without having to provide User credentials again. This flow is pretty straightforward.

The Client sends his stored refresh token to the IS. Remember that to connect to its token endpoint, the Client must have the so called token authorization. Because the IS stores all the issued refresh tokens, it check whether the one received does or does not exist in its temporary database. If the token coincides with the one stored, its expiration date must be checked right after. If everything is correct, then the IS stores the same refresh token but not without first refreshing its expiration date. On second place, both a new access token and a refreshed refresh token are issued and returned back to the client, who must delete the old tokens and store the newly issued ones.

In the rare case that the refresh token has expired (being unused for 15 days straight), the IS will refuse a refresh request. This will force the User to login again, requesting new tokens through his credentials. This can also happen if the IS restarts and loses the cached refresh tokens - the received refresh tokens will not be recognised.

6.1.5 Logout flow

This flow covers the token revocation request from the Client (on behalf of the User) to the Authorization Server. The goal is to, in addition to removing the stored tokens, render the tokens useless. This ensures that if a token has been stolen (which is very unlikely) it will be rendered useless anyways, preventing further usage.

Again, in order to connect to the token endpoint the Client must complete the token authorization flow. This might seem unimportant, but without this layer of security someone could start disabling stolen tokens without permission. After receiving the token authorization request successfully, the Client connects to the revocation token endpoint. He sends both the access and refresh tokens individually in separate requests. The access token need not be revoked, but it is good practice to do so. Revoking a refresh token however, is much more important.

The logout flow should not be allowed without internet connection. If the Client just deletes the tokens, we have already seen what an attacker with a stolen active refresh token can do.

7. Development

Most of the time was spent developing all the modules for the final solution. In this section I will try to cover my development process.

7.1 Identity Server 4 Customization

After checking out multiple startup tutorials and reading through the entire IS4 documentation, I was set out to mount a custom authorization server.

To get the core IS4 structure and functionality, you only need to start a new ASP.NET Core project with Visual Studio and proceed to install the IdentityServer4 NuGet[9] package. This gives you a basic implementation which you can start working on.

7.1.1 Defining resources

Identity resources are data like user ID, name, or email address of a user. An identity resource has a unique name, and you can assign arbitrary claim types to it. These claims will then be included in the identity token for the user. You can also define custom identity resources. Create a new IdentityResource class, give it a name and optionally a display name and description and define which user claims should be included in the identity token when this resource gets requested.

To allow clients to request access tokens for APIs, you need to define API resources. To get access tokens for APIs, you also need to register them as a scope. This time the scope type is of type Resource, and Clients that have this claim included will be able to access the resources from such API. See figure 3 below for a small example of the scopes definition.

```
Scopes =
{
    new Scope()
    {
        Name = "api2.full_access",
        DisplayName = "Full access to API 2",
    },
    new Scope
    {
        Name = "api2.read_only",
        DisplayName = "Read only access to API 2"
    }
}
```

Figure 3: ApiResource scope definition

7.1.2 Defining Clients

Clients represent applications that can request tokens from your identityserver. In this case, we only need to define one client: our cross-platform Xamarin app. The details vary, but you typically define the following common settings for a client:

- A unique client ID
- A secret (the password for connecting clients)
- The allowed interactions with the token service (grant type, ROWP seen in section 7 in this case)
- A list of scopes (resources) the client is allowed to access

```
new Client
{
    ClientId = "service.client",
    ClientSecrets = { new Secret("secret".Sha256()) },

    AllowedGrantTypes = GrantTypes.ClientCredentials,
    AllowedScopes = { "api1", "api2.read_only" }
}
```

Figure 4: Basic Client definition

In order to allow a Client to request refresh tokens together with access tokens, the client setting AllowOfflineAccess must be set to "true". All successful token requests made from this client will automatically be responded with a refresh token. You can also customize the maximum expiration time of a refresh token, which is 15 days by default.

7.1.3 Storing Users

Even though you can straight up store your users in the configuration file of the IS, that would be static and require a restart every time you modify the User list. A nice solution is to seed the User list dynamically from a SQL database. The database must be able to store at least a Username, a Password, and a list of Claims for every user.

Every time a token request is received, the IS checks if the external database's content has changed and proceeds to update the User list if so.

7.2 Xamarin Client

Xamarin is not an easy framework to develop with. Luckily my goal was to end up with a simple interface made solely for testing purposes.

7.2.1 Xamarin Design patterns

Design patterns might probably be my favorite thing about software engineering. Designing a complex architecture that follows all the principles and patterns your project needs brings a great sense of accomplishment in the end. Patterns ensure the code is never duplicated, functionalities are easy to extend and extras like allowing other developers to easier understand the code.

7.2.2 Model - View - ViewModel (MVVM)

MVVM is a software architectural pattern. MVVM facilitates a separation of development of the graphical user interface – be it via a markup language or GUI code – from development of the business logic or back-end logic (the data model).

- Model: refers either to a domain model, which represents real state content (an object-oriented approach), or to the data access layer, which represents content (a data-centric approach).

- View: the view is the structure, layout, and appearance of what a user sees on the screen. All this is usually filled with data prepared from the ViewModel. The components notify the ViewModel when the end-user interacts with them.

- View model: the view model is an abstraction of the view exposing public properties and commands. Instead of the controller of the MVC[10] pattern, or the presenter of the MVP pattern, MVVM has a binder. In the view model, the binder mediates communication between the view and the data binder. The ViewModel has been described as a state of the data in the Model.
- Binder: declarative data and command-binding are implicit in the MVVM pattern. In the Microsoft solution stack (Xamarin included), the binder is a markup language called XAML. Commands are sent from the View to the ViewModel and data is sent from the ViewModel to the View - all through bindings.

The ViewModel of MVVM can be seen as a value converter, meaning the ViewModel is responsible for converting the data objects from the model in such a way that objects are easily managed and presented. In this respect, the ViewModel is more Model than View, and handles most if not all of the View's display logic. The ViewModel may implement a mediator

pattern, organizing access to the back-end logic around the set of use cases supported by the View.

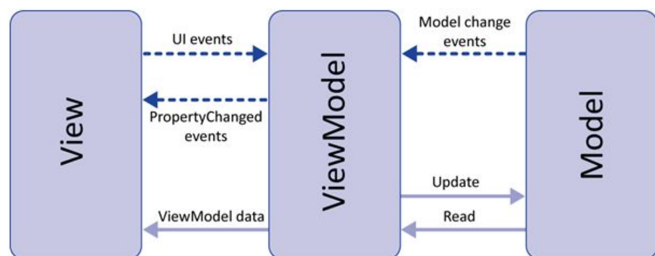


Figure 5: MVVM Pattern Structure

7.2.3 Dependency injection

Dependency injection is a standardized pattern which results very useful in the development of .net applications. At startup, interfaces and classes are registered in a global container. This container can be later requested to serve a new instance of a specified class. The container can also serve singleton instances, meaning their properties are stored and shared throughout the components that use them at runtime.

This avoids having to create massive lines of code when creating new instances of a class. The container resolves all the class' dependencies and serves you a final instance ready work with.

7.2.4 Persisting tokens

Both the access and refresh tokens must be stored by the Client in order to send them to the corresponding server when needed.

As we all know, when you close a mobile app all its data is cleared and deemed unrecoverable. The operating system may also forcefully close the app to free up resources, including RAM where our tokens might be stored. Luckily there are plenty of ways of allowing these tokens to persist throughout crashes and restarts, without the need for a complex database - after all, we just need to store 2 strings.

Xamarin has something that allows us to easily accomplish this goal, and it's called Properties. Properties creates a persisting programming dictionary where you can safely store your tokens. Should you wish to store a new persisting property for the application, all you need to do is tell the dictionary in which key you want it stored on and exactly what do you want to store. Keep in mind that uninstalling the application will delete this dictionary with all of its contents, just like with any mobile database. For example, you would store the tokens in the dictionary like so:

{ "AccessToken", (The access token string) }
{ "RefreshToken", (The refresh token string) }

7.2.5 Test interface

A simple interface was needed to check and validate connections between the Client, the Authorization Server and the Resource Server. This interface consists on a great amount of buttons, each triggering a different process. At the bottom, an output text displays the result of the triggered process.

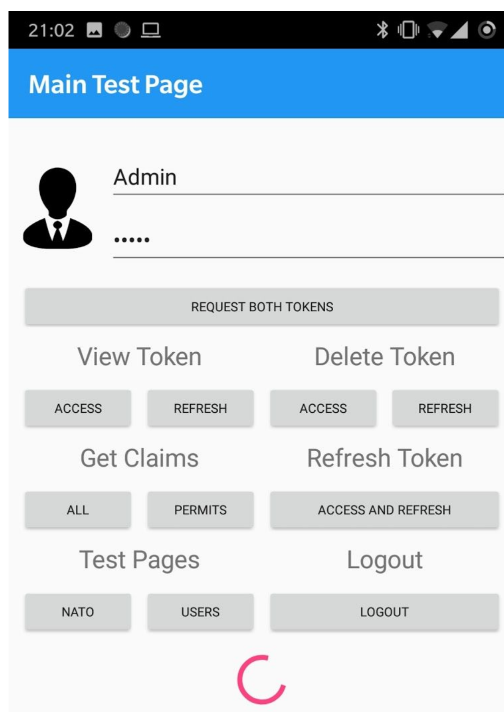


Figure 6: Xamarin Client test interface

7.3 Web Api

My Resource Server needs to be a simple API that authorizes HTTP / HTTPS requests based on the bearer token sent as a request header.

Using ASP.NET Core, a library eases the setup of requests authorization. I only needed to configure some parameters that the API should check. This included the base server of the token issuer (my Identity Server's base address), the token expiration date and the allowed scopes, amongst others.

The library also allows the API controllers to read all the claims the token contains. This proves very useful in development environments like mine - I can send all the claims or a set of specific claims to the Client, and check if everything is working properly throughout all the components.

7.4 Tests

Having designed a user interface that executes all the processes that needed to be tested, this last step was a breeze. All tests were taken from the functional requirements specification and by contrast, stress tests were not meaningful at this time so these non-functional requirements were not tested.

Thinking with use cases, the user must be able to:
- Login into the system with his credentials
- Extend his session time without logging in
- Request resources
- Logout

Having ensured that the user was capable of all this with the current system, this project was concluded.

8. Conclusion and further development

Having concluded this software engineering project, I finally see the value of studying a software engineer's degree at university. This, by no means, can be achieved without the knowledge I've acquired these past 4 years.

OAuth2.0 has proven to be one hell of a protocol, but the things I learned including research and development will serve me well in my professional career as a software engineer. I will gladly venture in another project of similar complexity should I get the chance.

After all this work, I can safely say that this project has been a great success. I managed to get everything working in no time, which I honestly did not expect. The company is moving forward and expects me to develop a fully working system by the end of this summer.

Special thanks

This project would have not been possible with the grandiose help from my university tutor Debora Gil, my company tutor Jordi Soler and my colleague Daniel Bautista - I doubt I would have accomplished all this by myself.

I also would like to take a moment and appreciate all the efforts my parents have made to ensure I could study software engineering at university and start my dream professional career as young as I am.

9 Bibliography

[1] Cross-Platform computer software can be implemented on multiple platforms. More: https://en.wikipedia.org/wiki/Cross-platform

[2] JWT is a JSON Web Token, a means of representing claims to be transferred between 2 parties. More: https://openid.net/specs/draft-jones-json-web-token-07.html

[3] GSuite is the business account type that Google offers. More : https://gsuite.google.com

[4] Apache is a a standard permissive software license. More: https://www.apache.org/licenses/LICENSE-2.0

[5] Trello is a web-based project management application. More: https://www.trello.com/

[6] Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.
More: https://git-scm.com/

[7] Universal Windows Platform (UWP) is a common app platform on every device that runs Windows 10: a PC, Windows Phone, an Xbox... More: https://docs.microsoft.com/en-us/windows/uwp/

[8] All the grant types that IdentityServer4 supports can be found here: http://docs.identityserver.io/en/release/topics/grant_types.html

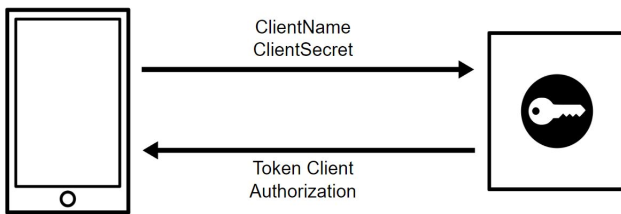
[9] NuGet is a free and open-source package manager for Visual Studio. More info: https://www.nuget.org/

[10] Model-view-controller (MVC) is commonly used for developing software that divides an application into three interconnected parts. More:

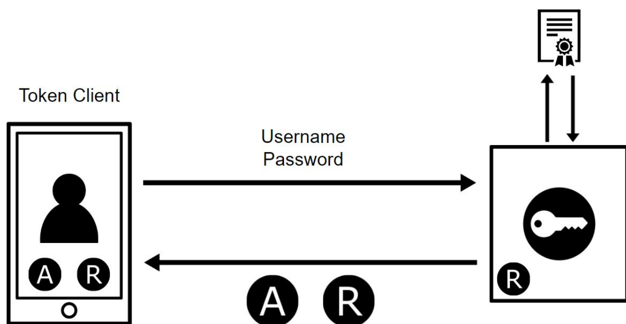
11 Appendix

Resource Server
API +
Certificate
Resource Owner
A
Requests Resources
Signs Access Tokens
Uses
Client
Authorization Server
Xamarin
Identity Server Flows
A R
Tokens

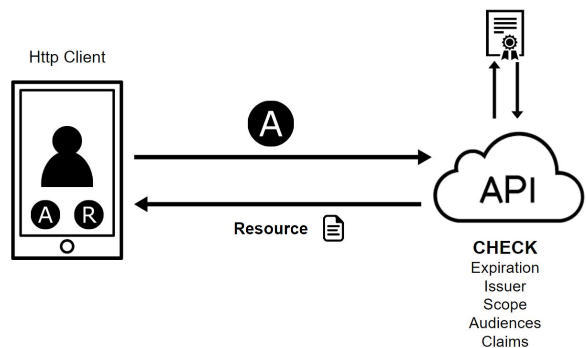
A1 - All roles and and flows



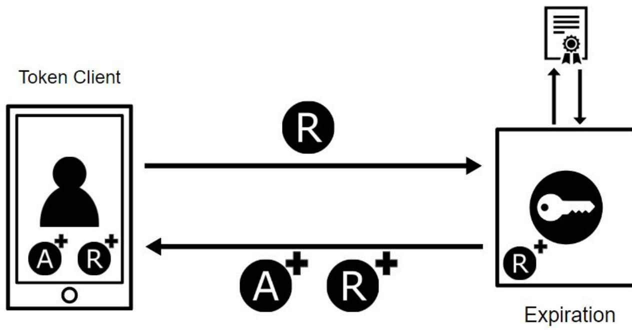
A2 - Client token authorization request flow



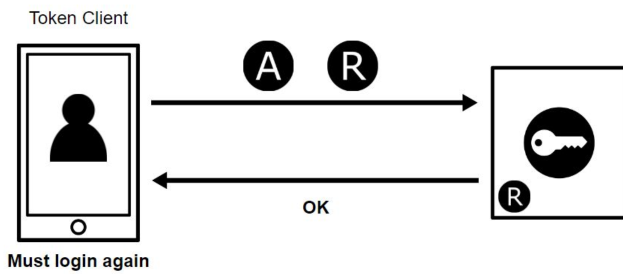
A3 - Access and Refresh token request



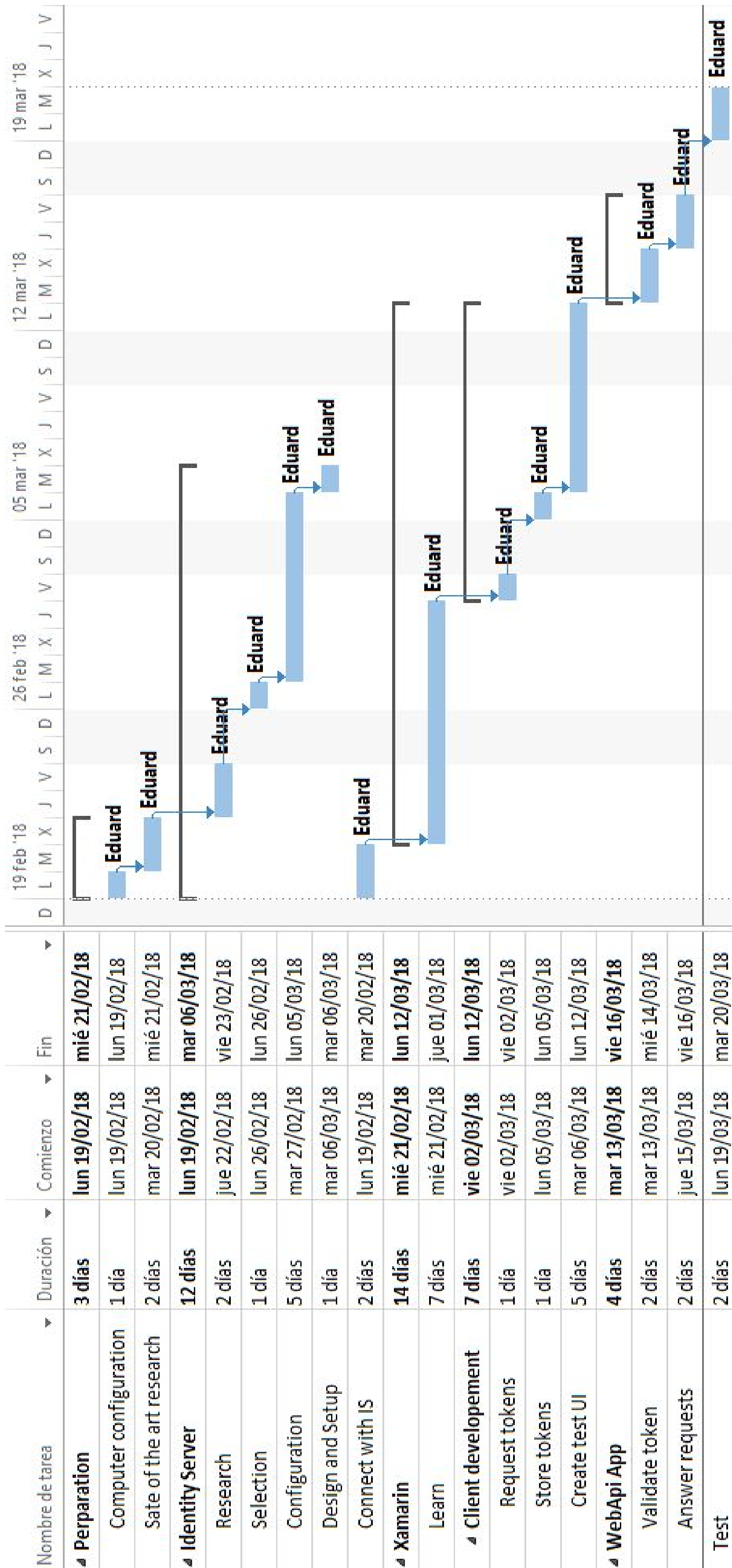
A4 - Resource access request flow with a valid access token



A5 - Access token refresh flow with a valid refresh token



A6 - Token revocation flow and logout



Nombre de tarea	Duración	Comienzo	Fin
Perparation	**3 días**	**lun 19/02/18**	**mié 21/02/18**
Computer configuration	1 día	lun 19/02/18	lun 19/02/18
Sate of the art research	2 días	mar 20/02/18	mié 21/02/18
Identity Server	**12 días**	**lun 19/02/18**	**mar 06/03/18**
Research	2 días	jue 22/02/18	vie 23/02/18
Selection	1 día	lun 26/02/18	lun 26/02/18
Configuration	5 días	mar 27/02/18	lun 05/03/18
Design and Setup	1 día	mar 06/03/18	mar 06/03/18
Connect with IS	2 días	lun 19/02/18	mar 20/02/18
Xamarin	**14 días**	**mié 21/02/18**	**lun 12/03/18**
Learn	7 días	mié 21/02/18	jue 01/03/18
Client developement	**7 días**	**vie 02/03/18**	**lun 12/03/18**
Request tokens	1 día	vie 02/03/18	vie 02/03/18
Store tokens	1 día	lun 05/03/18	lun 05/03/18
Create test UI	5 días	mar 06/03/18	lun 12/03/18
WebApi App	**4 días**	**mar 13/03/18**	**vie 16/03/18**
Validate token	2 días	mar 13/03/18	mié 14/03/18
Answer requests	2 días	jue 15/03/18	vie 16/03/18
Test	2 días	lun 19/03/18	mar 20/03/18

A7 - Gantt Diagram