Un sistema d'adquisició i interpretació de dades sobre processos industrials

Grzegorz Szymanski

Abril de 2019

Marc Alier, Enginyeria de Serveis i Sistemes d'Informació

Titulació: Màster en Enginyeria Informàtica

Centre: FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

1 RESUM

El projecte consisteix en crear un entorn d'observació d'una màquina industrial on l'usuari d'aquest sistema sigui capaç de veure en temps real l'estat de totes les entrades i sortides i de la memòria de la màquina que està observant mitjançant una aplicació mòbil. D'aquesta manera el projecte consta de 3 parts principals:

- Un dispositiu amb connexió a Internet capaç de comunicar-se amb el PLC de una màquina industrial, utilitzant el protocol Ethernet.
- Un Web Service[12] programat amb la tecnologia .NET, on es connecten les altres dues parts.
- Una aplicació mòbil desenvolupada amb el framework multi plataforma Xamarin.Forms, que fa la funció de la interfície del usuari.

Per tant el projecte del que estem tractant, consisteix, en detall, en preparar un esquelet de comunicació entre una màquina industrial i el món exterior, utilitzant les últimes tecnologies multi-plataforma de .NET Core. Al utilitzar aquest tipus d'eines, podem aconseguir una millor escalabilitat i una millor aproximació del que seria el producte final. Per aconseguir aquesta escalabilitat l'objectiu es utilitzar tecnologies mes sofisticades en cloud (Azure per exemple). Per la part de la interfície de l'usuari s'utilitzarà un framework, com per exemple el de Xamarin.Forms, que permeti crear aplicacions multi-plataforma, reutilitzant així una gran part del codi que podem exportar a diferents sistemes operatius del mercat.
Aquest projecte es una gran oportunitat a les empreses del món industrial, ja que, un sistema d'aquest tipus augmenta considerablement l'eficiència de tots els processos que es duen a terme en una fàbrica, per exemple, eliminant el factor humà en molts d'ells. De fet, treballant en diferents empreses he vist un exemple molt clar on es podria aplicar aquest sistema. Aquesta empresa, fundada en els temps on l'electrònica encara estava al seu inici, no tenia previst utilitzar la tecnologia per augmentar el rendiment dels seus treballadors, provocant així un seguit d'accions repetitives amb molt marge d'optimització que cada empleat havia de fer al llarg de la seva

jornada. Un exemple podria ser el fet de que cada treballador estava obligat a deixar el seu lloc de treball cada cop que canviava de feina/peça, ja que, ho havia de documentar en un full que estava a l'altre punta de la nau. Sabent això vam poder aconseguir fer un seguit de càlculs per calcular les pèrdues que li suposava a l'empresa aquest fet. Vam comptar que cada treballador tardava de mitjana uns 2 minuts per documentar la seva feina i ho feia uns 2 cops al dia (de mitjana), sabent que l'empresa té uns 50 obrers i que a l'any es sol treballar uns 210 dies, vam arribar a la conclusió de que l'automatització d'aquesta acció li suposa un estalvi igual a 7.000€ anuals, una xifra important per una empresa d'aquesta magnitud.

El projecte es desenvoluparà durant els quadrimestres de la tardor de 2018 i la primavera de 2019, matriculant-ho en juliol i presentant-ho en abril del any 2019 permetent així una creació d'una base molt sòlida, d'un sistema que pot marcar la diferencia en moltes àrees del món industrial

2 ABSTRACT

This project consists in providing an observation environment that could be used in industrial machinery, where the user of this system is able to see in real time, all the states and processes occurring inside some remote machine, using a mobile application. The project is structured in 3 parts:

- A device with internet connection, with the ability to communicate with the PLC controller of some machinery, using Ethernet protocol.
- A Web Service programmed using the .NET technologies where the other parts were connected.
- A mobile application developed using the Xamarin.Forms multi-platform framework that had the task of being the user interface.

So finally, the project that I'm describing in this document, has the goal of prividing a skeleton of communication between an industrial machinery controlled by a PLC controller with the exterior world. We will use the latest .NET Core technologies in order to achieve the best scalability and the best approximation of what could be the final product sold to the customer. We will use cloud technologies (Azure i.e.), for storage and hosting purposes, and we will use Xamarin.Forms framework as the basis for our multi-platform mobile app, in order to reuse as much code as possible between all the biggest OS in the market.

This project is a huge opportunity for industrial area companies, because this kind of system can improve the efficiency of all the processes that have place in, for example, a manufacture, removing the human factor from them. With my experience while working in a few companies from the sector I was able to see clear examples where this kind of system could improve some processes. As an example I will give an enterprise founded in times when the electronics were not so popular and the owner hadn't had in mind the usage of technologies in order to improve the workers efficiency, creating, in this way, lots of repetitive processes with a huge margin of improvement. One example could be the fact that every single worker was told to leave his working place every time that he finished his current task in order to go to

the other side of the building to document his work in a common notebook. Knowing this simple task I was able to calculate the looses that the company had to deal with. I calculated that every worker was spending more or less 2 minutes to document his work, and he was doing it twice a day. This enterprise has 50 workers, and knowing that the year has 210 working days, I was able to arrive to the conclusion that the automation of this process would suppose an increase of profit for the company equal to 7.000€ every year. Very important number for enterprise of this magnitude.

This project will be done during the autumn semester of 2018, and spring semester of 2019, with the inscription done in July, and the presentation planned for April of 2019. With this time I will be able to create a solid base, of a system that could mark the difference in many sectors of the industrial world.

ÍNDEX

ÍNDEX DE FIGURES

3 INTRODUCCIÓ

L'objectiu d'aquest projecte es la creació d'un esquelet escalable d'un sistema capaç de monitoritzar en temps real l'estat d'una màquina industrial, o qualsevol altre dispositiu controlat per un PLC que segueix l'estàndard Modbus[16]. Per aconseguir aquest objectiu estic optant per l'entorn donat per l'empresa Microsoft anomenat .NET Core i tots els seus frameworks, que van des de llocs per fer hosting en cloud de tota la lògica web fins a la aplicació mòbil.

La principal motivació que s'amaga a darrera d'aquest projecte es la seva enorme utilitat i la seva capacitat de realment canviar la manera de funcionar dels processos industrials que ara cada cop més, utilitzen la tecnologia per tal d'optimitzar els seus procediments. Aquest industria, anomenada la 4.0, es un sector en un estat de creixement increïble on absolutament totes les empreses han de considerar utilitzar sistemes de informació per tal de poder ser competitius a nivell de eficiència, i amb això vinculat, costos.
En el resum d'aquesta memòria ja està citat el cas d'una empresa que implementant un simple monitor dels processos que realitzen les seves màquines, ja podrien estalviar 7.000€ anuals. Però la cosa no s'acaba aquí, el pas de tenir una base de funcionament i ser capaços d'enviar dades d'una màquina a un dispositiu intel·ligent (que es del que tracta aquest projecte), ens obre moltes portes per una millora molt més important i profunda en els processos realitzats. Aplicant algoritmes de big data i coneixement dels experts, amb uns simples gràfics que podrien mostrar per exemple l'eficiència dels treballadors en algunes condicions específiques, es poden fer moltes prediccions permetent així una enorme millora pel que fa la gestió del capital d'una empresa. Donades unes certes característiques, una empresa així seria capaç de preveure temporades de més bona productivitat, permetent així unes inversions molt més extenses, augmentant així el benefici i el creixement d'aquesta.
Un sistema així podria servir també com una eina d'automatització, on les dades obtingudes sobre el funcionament de les màquines, podrien indicar el temps que s'ha invertit per realitzar una tasca. Per tant establint una connexió entre la base de dades que implementa quest projecte i el ERP del client, podríem automàticament crear pressupostos i despeses per la realització d'alguna tasca, estalviant així molt de temps i/o errors humans.

Pel que fa la tecnologia utilitzada, s'ha optat per utilitzar l'entorn de .NET amb la seva última especificació open source, Core, per la maduresa dels seus frameworks i la seva integració entre totes les parts que s'hauran d'integrar per poder crear aquest sistema.
En primer lloc s'utilitza una aplicació multi-plataforma de línia de comandes, basada en el framework .NET Core, per tal d'implementar el protocol de lectura de dades d'un PLC utilitzant l'estàndard Modbus. Aquest programa pot estar executat en qualsevol ordinador capaç d'executar aplicacions .NET, per tant tots els sistemes operatius més significants actualment. El dispositiu ha d'estar connectat al PLC mitjançant el port de comunicació Ethernet. El procés de lectura de dades es totalment flexible, utilitzant paràmetres d'execució.
En segon lloc s'utilitza una aplicació mòbil multi-plataforma, implementada utilitzant el framework Xamarin, que es la implementació del estàndard de .NET per sistemes operatius que utilitzen la màquina "mono"per tal d'executar aplicacions .NET. Actualment tots els sistemes operatius de dispositius mòbils actuals utilitzen aquest mètode per poder ser executades. En aquest projecte s'utilitza la varietat del framework anomenada Xamarin.Forms, on a part de ser capaços de compartir la lògica, podem compartir la mateixa definició de les vistes, gràcies al estàndard de "XAML".
Per últim, en aquest projecte s'utilitza la plataforma Azure, per tal de fer hosting de la base de dades SQL creada especialment per necessitats d'aquest projecte. Aquesta base de dades es accessible mitjançant una Web Api, creada en entorn de .NET Core que implementa el model de dades mitjançant el Entity Framework Core. Tota la estructura es accessible i funcionant les 24h gràcies a la llicència gratuïta que ens ofereix Azure.

Durant la implementació d'aquest projecte no s'ha tingut en compte el tema de autentificació de usuaris, ja que, aquest projecte es una prova de concepte, i una base per un sistema real que pugui ser venut al client, que segurament voldrà imposar una manera d'autentificació pròpia.

4 OBJECTIUS

L'objectiu d'aquest projecte es crear l'esquelet d'un sistema que vindria a ser una eina capaç d'extreure informació sobre el funcionament de màquines industrials. En detall:

- Crear un programa capaç d'extreure informació d'un PLC, mitjançant el protocol Modbus, i enviar-la de manera controla a un servidor.
- Implementar una base de dades i un Web Service, guardada en una cloud de Azure, capaç d'emmagatzemar i gestionar les dades rebudes.
- Dissenyar i desenvolupar una aplicació mòbil multi-plataforma capaç d'accedir a la informació emmagatzemada en el Web Service mitjançant una API REST, i mostrar-la al usuari.

5 ESTAT DE L'ART

Avui en dia el món de la indústria està vivint un revolució irrevocable a la qual s'han d'unir absolutament totes les empreses par tal de continuar ser competents en el mercat. Es una revolució que dóna lloc a una industria amb un nivell d'eficiència totalment desconegut fins ara, una indústria capaç d'implementar totes les novetats tecnològiques per tal d'augmentar la seva productivitat i els processos amb els que es guia.

Per tant en aquests temps s'ha creat un mercat enorme per empreses IT que es posen com l'objectiu crear solucions el més extensibles, i adaptables, per tal d'abastar el màxim número de sectors de la indústria amb el seu producte, i treure-li el màxim de profit. Aquests productes tenen com el principal objectiu automatitzar els processos que es duen a terme a diari, trencant totalment la manera de treballar coneguda fins aleshores, on l'accés a les dades i la seva anàlisis permet treure conclusions totalment noves sobre molts dels processos.

Aquí es on entren en joc projectes com aquest, capaços de convertir informació de les màquines, en informació fàcilment processable i comprensible per persones que poden extreure conclusions a partir d'elles. Amb aquesta base es poden crear infinites millores a tots els aspectes, seguint les pautes i utilitzant les eines de les tecnologies analítiques de Big Data. Però tot això es possible gracies a l'accés a aquesta informació, que fa uns anys enrere encara no era possible.

A part de poder treure conclusions, un software així, acoplat amb altres programes de l'empresa, com per exemple, un ERP, pot automatitzar i convertir en segons, una feina que feta per una treballador seria, a part de monòtona, extremadament lenta, subjecte al errors humans.

Per l'altre banda en el món de la informàtica cada cop més podem apreciar com amb un dinamisme increïble les tecnologies es van adaptant a les necessitats del món. Com que actualment estem vivint en una era on la informació és la moneda amb més valor les tecnologies van seguint el mateix camí. D'aquesta manera, com es el cas d'aquest projecte, una persona es capaç d'implementar tot un ecosistema d'obtenció, enviament i interpretació de dades, només cal utilitzar les eines adequades i dedicar-li temps. Fet que fa uns anys enrere seria completament impossible d'aconseguir, sense tenir una experiència molt extensa en tots els camps en quins aquest projecte entra.

En el cas d'aquest projecte, s'ha decidit basar-se en el modern entorn .NET en la seva última

especificació Core 2.0, que permet unificar tota la solució sota una mateixa família de tecnologies que comparteixen el llenguatge de programació, el seu entorn i la manera de treball, fent que coneixent una framework que hereta d'aquesta família, la corba d'aprenentatge per dominar un altre framework de la mateixa família es molt menys dolorosa que no pas la d'haver d'aprendre un entorn de nou.
A part de la unificació, l'entorn en sí ens dóna moltes solucions que s'adapten a les necessitats de tots els projectes implicats en aquesta solució. Vegem-ne:

- .NET Core: Pel que fa la implementació de la part de la solució connectada directament a la màquina industrial, aquest es un framework ideal, que permet una flexibilitat enorme pel que fa la implementació ja que té una llibreria bàsica molt poderosa que li permet al programador, totalment desconnectar-se del fet d'estar treballant en un projecte multiplataforma. Si per exemple s'hagués triat fer un programa escrit en C++, com ho fan la gran majoria de competidors, el temps i l'esforç del desenvolupament seria magnituds més gran, pel simple fet de no proporcionar una interfície el suficientment abstracte respecte a la màquina on s'executa el codi (la implementació dels sockets TCP per exemple).

  Tot això ens aporta avantatges que realment no estan pagades amb ningun desavantatge, ja que, tot i ser un entorn de molt alt nivell, aquest no ens penalitza amb un rendiment inferior. De fet passa el contrari on l'entorn es capaç d'extreure el màxim de profit d'un llenguatge interpretat, mantenint al mateix moment les avantatges d'entorns precompilats, gràcies a la madura especificació CLI[17] (Common Language Infrastructure) que aquest entorn implementa.

- ASP.NET Core: Si ens centrem en la part servidora del projecte, podem veure que l'entorn open source proporcionat per Microsoft, és un dels més potents del mercat, en molts aspectes. Es líder per que fa la integració amb altres sistemes necessaris per tal de crear un entorn servidor modern complet, com per exemple la publicació de la nostre feina en un entorn cloud, que en aquest cas es limita a una interactiva configuració de totes les parts.

  En les seves últimes especificacions, aquest entorn s'ha posat com líder pel que fa el rendiment en moltes utilitats (accessos a la base de dades, número de peticions per segons que es capaç de manejar, etc.) fent que la implementació d'un projecte competitiu no requereix d'una experiència extremadament elevada en termes d'implementació de

serveis Web.

- Xamarin: La part clau del sistema, que realment li mostra la potència de tot el sistema al usuari final, que amb una eina que utilitza cada dia, com es el seu telèfon mòbil, es capaç de veure d'una manera senzilla, l'evolució de funcionament d'una màquina que te ubicada a l'altre punta del món, si cal. Al ser un entorn completament multiplataforma, basat en el mateix entorn .NET, permet implementar aplicacions que s'adapten a les necessitats i costums dels clients, que sempre podran utilitzar eines amb les quals són més familiaritzats, sense posar al davant del desenvolupador reptes, com seria el cas d'haver de mantenir 3 aplicacions separades, per tal de suportar totes les plataformes dels clients finals (Android, iOS, UWP).

Dit això, podem resumir que aquest projecte es pot veure com una prova de concepte, ja que, a part de les seves utilitats immediates per les empreses, també ensenya el potencial que tenen els frameworks disponibles de forma totalment gratuïta i la falta de límits que ens proporciona el món informàtic actual.

6 METODOLOGIA

Per realitzar aquest projecte i totes les diferents fases del mateix, s'ha optat per utilitzar un desenvolupament iteratiu, on cada iteració es denomina sprint. Aquesta metodologia es basa en l'adaptabilitat de qualsevol canvi del projecte com a mitjà per augmentar les possibilitats d'èxit d'un projecte, fent de forma iterativa els següents processos: Planificació, desenvolupament, execució i test. La seva utilització ha permès fer profit de les següents capacitats:

- Es basa en l'adaptabilitat de qualsevol canvi com a mitjà per augmentar les possibilitats d'èxit del projecte, intentant minimitzar el risc desenvolupant en iteracions.
- Procura dividir el projecte principal en diferents subprojectes amb l'objectiu de reduir la dificultat de desenvolupament en cadascun d'ells. A més, d'aquesta manera és més fàcil adaptar qualsevol canvi d'implementació havent només de modificar un o alguns dels subprojectes i no haver de tractar el canvi en el context d'un projecte general gran que després podria propagar alteracions en el codi d'altres subprojectes modificant la funcionalitat.
- Utilitza el desenvolupament iteratiu o àgil, amb el qual es realitza la fase de testing de forma paral·lela amb el desenvolupament del codi. Això resulta especialment útil en aquest projecte, ja que cada cop que es fa qualsevol petita implementació es pot provar i validar aquesta petita peça del programari. A més, el fet de tenir a cada iteració del desenvolupament la possibilitat de replantejar el camí que seguirà el programa, permet intentar maximitzar el valor que aquest ofereix.

Com que en el projecte falten figures necessàries (client, cap de projecte, etc...) no podem parlar d'un procés agile, però si que podem adaptar la mateixa forma de treballar, on el qui posa els objectius i valora el progrés de la implementació, es el mateix developer.

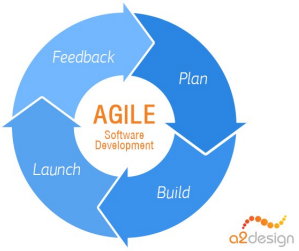


Fig. 1: Visualització d'una metodologia iterativa.

7 PLANIFICACIÓ

Per fer aquest projecte, s'han definit de manera ordenada una sèrie d'activitats i subactivitats a desenvolupar, per tal d'assolir l'objectiu de realitzar un procés d'enginyeria del software. La definició d'aquestes tasques, ha permès tenir un control del progrés del desenvolupament.

7.1 Fases i activitats

- Formació i investigació: Familiaritzar-se amb l'entorn del framework .NET Core i tots els seus components com: Entity Framework (per la gestió de BBDD), ASP.NET Core (per la implementació de la API), Xamarin.Forms (per l'aplicació mòbil) etc.

  Una segona gran tasca d'adaptació seria la d'entendre el funcionament del servei cloud Microsoft Azure on es posarà en marxa la nostre base de dades i la API.

- Disseny: Elaboració dels requeriments del sistema i amb això establir un prototip de la interfície d'usuari, la base de dades del sistema i la API d'accés a aquesta.

- Desenvolupament:

  - Desenvolupament del procés "daemon"monitoritzador que mitjançant els ports Ethernet extregui les dades del PLC de la màquina.
  - Desenvolupament de la base de dades i de la API d'accés, en local, per tal de poder enviar les dades extretes del PLC.
  - Desenvolupament de la aplicació mòbil.
  - Publicar la base de dades i la API en el entorn de Azure, i configurar-lo tot perquè totes les part puguin comunicar-se entre elles.

- Test: Comprovar el correcte funcionament del disseny i dels requisits documentats i corregir errors. Cal destacar que tant el desenvolupament de l'aplicació com el testing s'han fet de manera simultània al desenvolupament, gràcies a la metodologia àgil que s'ha fet servir.

- Escriptura de la memòria: Utilitzant el llenguatge Latex.

Com que tot el desenvolupament s'ha encarat a la utilització de les eines de la companyia Microsoft, el més beneficiós ha estat utilitzar la plataforma Azure DevOps[1] per tal de utilitzar-lo com es gestor de codi font, i com a gestor del projecte, en el qual de forma molt fàcil es poden definir tasques a realitzar, els sprints, i objectius en general. En aquesta plataforma m'he creat projectes separats per a cada part del sistema que aquest implementa.

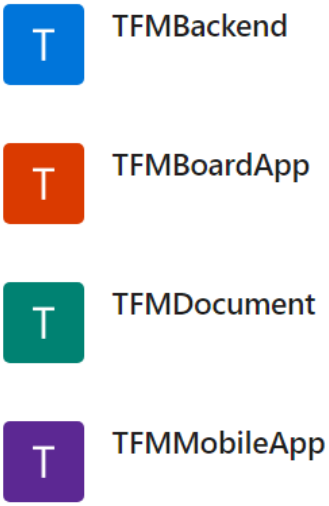


Fig. 2: Projectes creats per cada part del sistema

D'aquesta manera s'han creat els següents objectius amb les seves corresponents tasques:

7.2 Planificació del monitoritzador del PLC

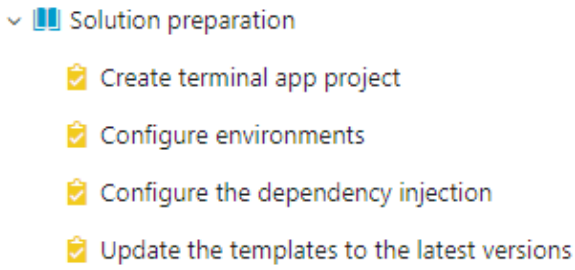


Fig. 3: Creació de la solució

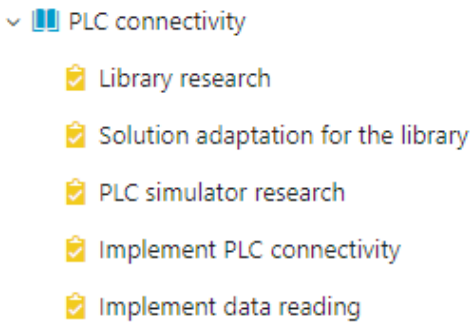


Fig. 4: Implementar la connectivitat amb el PLC

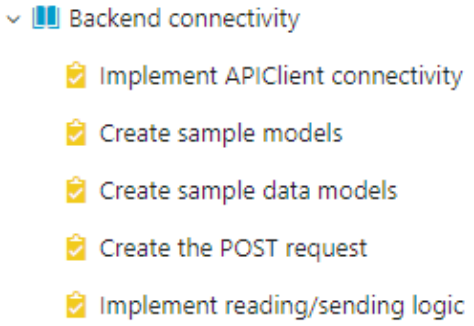


Fig. 5: Implementat la connectivitat amb la API

7.3 Planificació de la part servidora

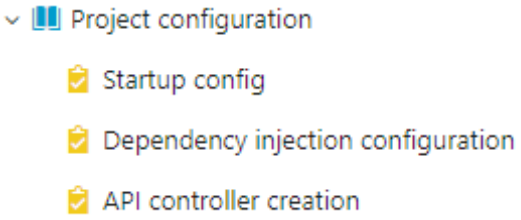


Fig. 6: Creació de la solució

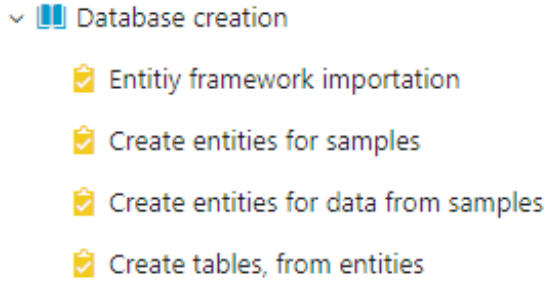


Fig. 7: Creació de la base de dades i les seves taules

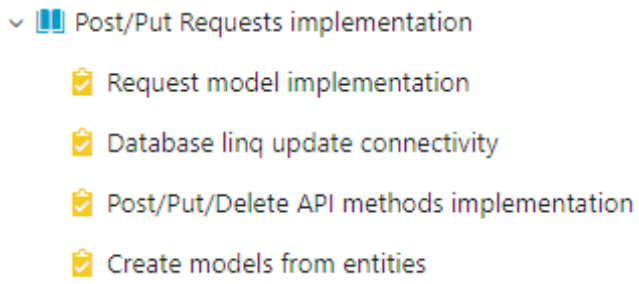


Fig. 8: Implementació dels mètodes d'actualització

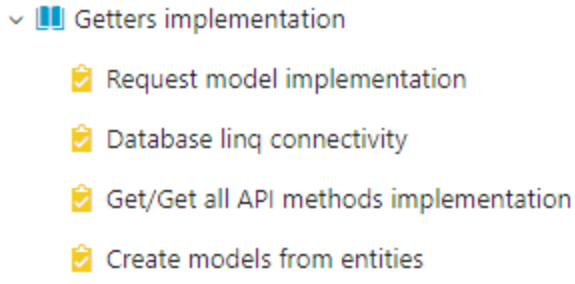


Fig. 9: Implementació dels mètodes de lectura

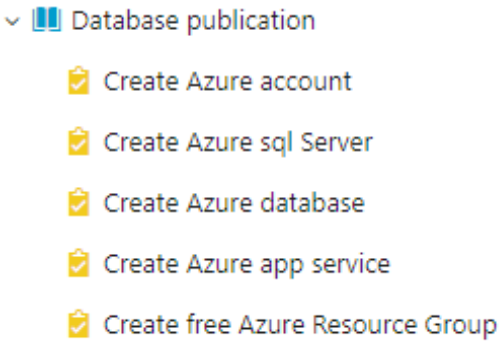


Fig. 10: Publicació de la base de dades i de la API

7.4 Planificació de l'aplicació mòbil

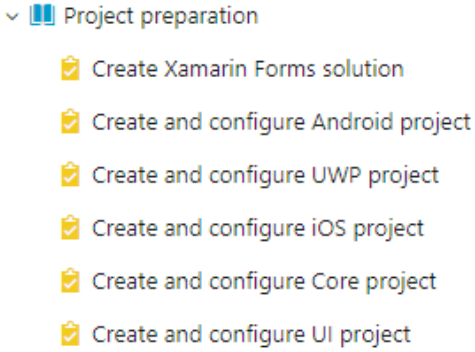


Fig. 11: Creació de la solució

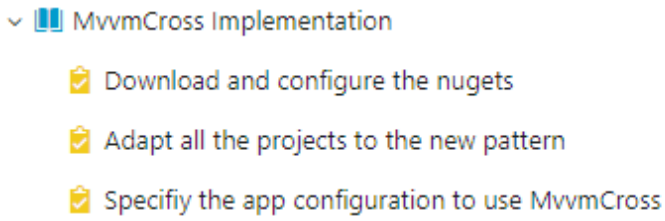


Fig. 12: Integració del framework MvvmCross en la solució

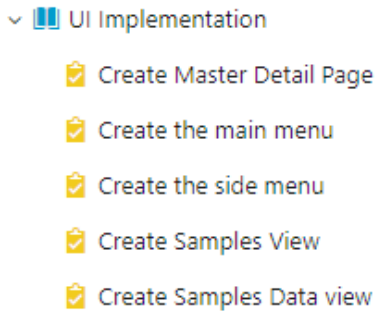


Fig. 13: Creació de les vistes de la aplicació

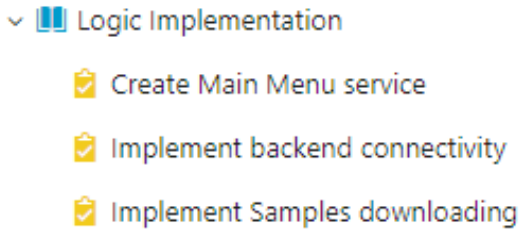


Fig. 14: Implementació de la lògica del negoci

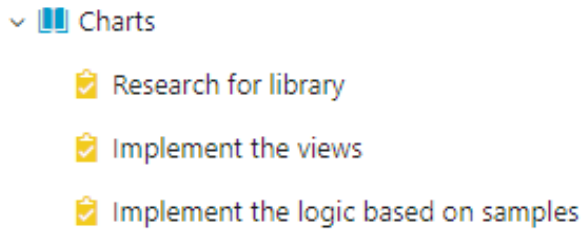


Fig. 15: Implementació de les gràfiques sobre les dades obtingudes

7.5 Planificació de la escriptura de la memòria

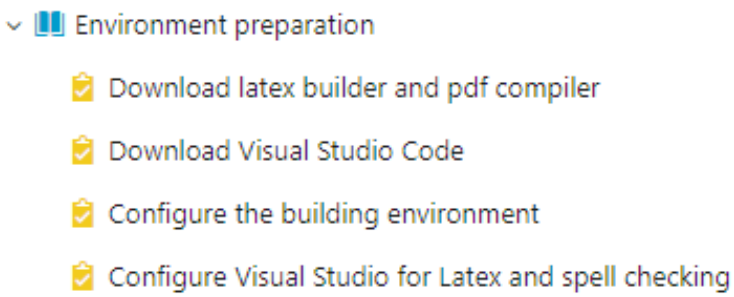


Fig. 16: Preparació del entorn per Latex

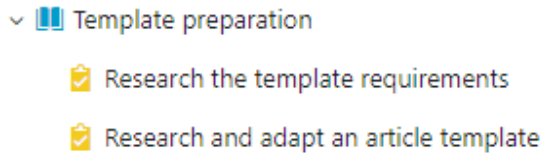


Fig. 17: Configuració del patró a utilitzar per document

7.6 Planificació de la creació de la presentació

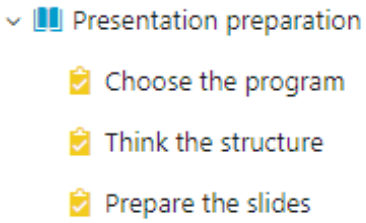
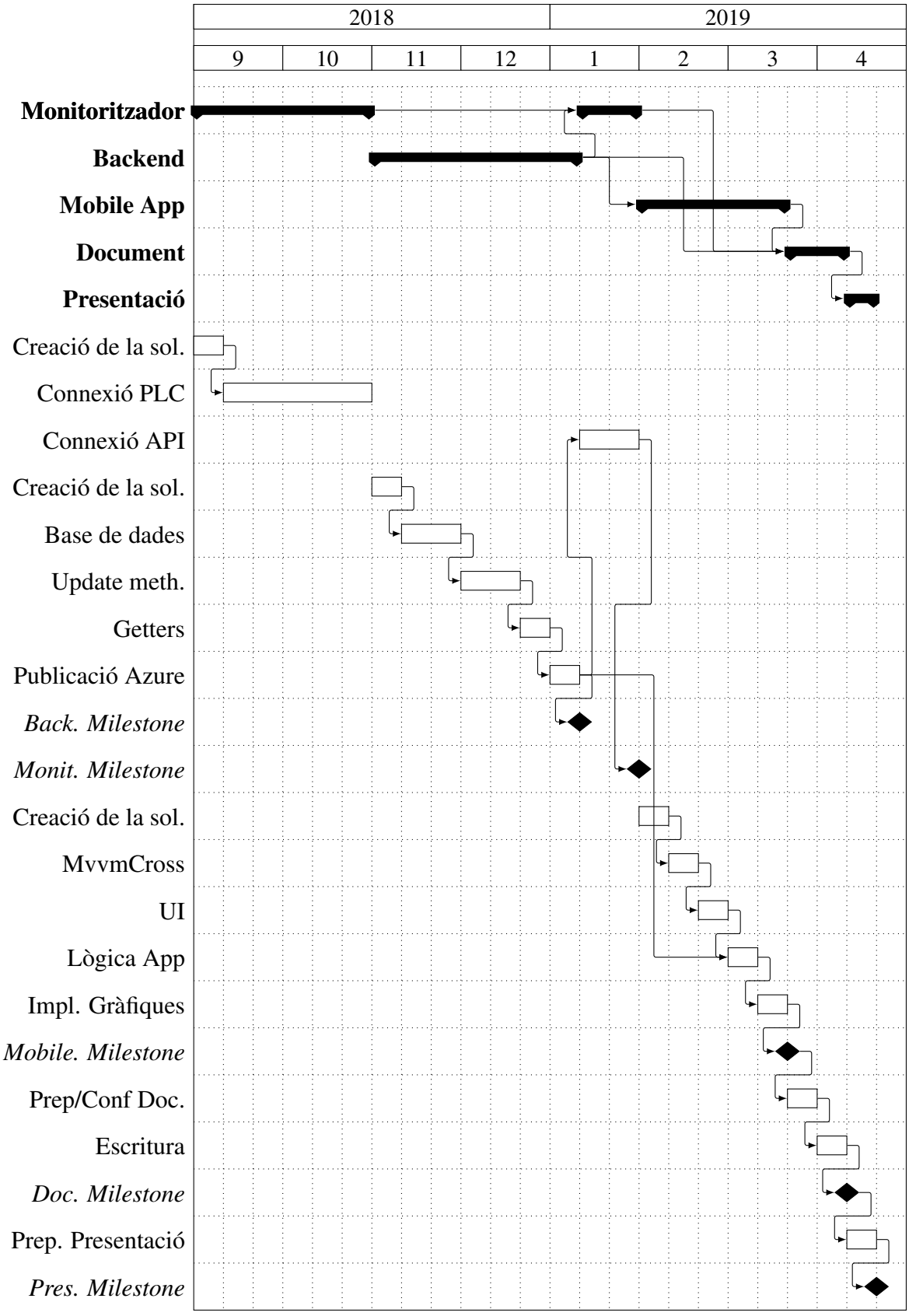


Fig. 18: Preparació de la presentació

7.7 Diagrama de Gantt del projecte



7.8 Eines

Per poder realitzar aquest projecte, ha sigut indispensable l'ús del següents softwares:

- Visual Studio 2017[15]: Plataforma que permet desenvolupar aplicacions utilitzant el entorn .NET, utilitzant tant C#[3], F#[4] com C++[5]. Pot:
  - Compilar *(build)* aplicacions per a diferents Sistemes Operatius com Windows o Android.
  - Desplegar *(deploy)* aplicacions i software de manera rapida i eficient.
  - Executar *(run)* aplicacions de manera optimitzada i debugar-les.
- mod_RSsim: Simulador capaç de crear un servidor HTTP simulant així el funcionament d'un PLC.

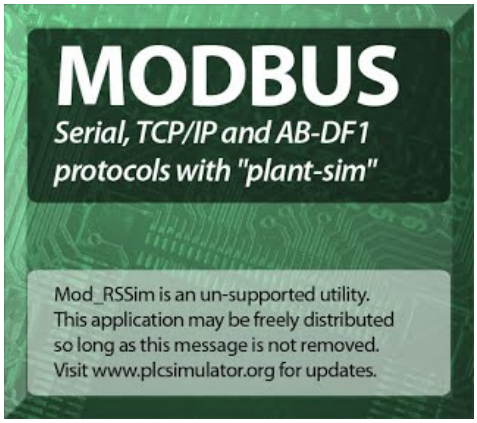


Fig. 19: Modbus PLC Simulator. Obtingut en http://www.plcsimulator.org

- Microsoft Azure: Plataforma Cloud, que permet el hosting de diversos serveis.
- Azure DevOps: Plataforma que permet el hosting de codi font i l'organització de les tasques de cada projecte del sistema.
- Tex Live: Compilador de fitxers en format Latex capaç de convertir-los al format PDF
- Visual Studio Code: Editor de textos de diversos formats.

8 DESENVOLUPAMENT

En aquesta apartat del projecte s'explica amb molt detall la implementació de cadascuna de les parts del sistema.

Totes les parts del projecte han estat desenvolupades utilitzant el entorn .NET i el llenguatge de programació C#. Totes les seves parts s'han dissenyat pensant en la seva màxima portabilitat, escalabilitat i eficiència. Per tal d'aconseguir aquests objectius he optat per la utilització de la variant més moderna del entorn, anomenada, .NET Core. Es una versió oberta del framework, que té com a objectiu la unificació de tots els entorns del món de .NET. Al ser un entorn molt modern i molt nou, una tasca molt important a resoldre es la d'adaptació al nou entorn. TTambé cal esmentar que aquesta es una tecnologia amb molt creixement, així que la ajuda que es pot trobar per internet a vegades pot ser molt escassa.

Això ha sigut visible sobre tot en l'apartat del Web Service i la base de dades, on molt poca gent utilitza la nova modalitat del entorn, ja que, els experts de la matèria, encara han de fer el pas de transició a la nova tecnologia.

8.1 .NET Core

.NET Core (dot net core) es un framework desenvolupat per Microsoft que intenta estandarditzar i unificar l'entorn de programació per diferents sistemes operatius més populars. El framework en sí utilitza una màquina virtual anomenada CLR (Common Language Runtime) capaç d'executar diferents codis escrits en diferents llenguatges sobre diferent Hardware. Tot el codi és executat en un entorn de Software totalment independent del Hardware on aquest està ubicat, permetent al programador un treball molt eficient sense haver de preocupar-se dels detalls de la plataforma física on el programa estarà ubicat.

Fig. 20: Logotip del framework .NET Core

Aquest framework està en un continu desenvolupament on cada cop se li afegeixen més característiques al entorn i la llibreria bàsica i es dóna suport a les novetats dels llenguatges de programació que aquest framework dóna suport. Un dels avantatges més grans pel que fa l'avanç d'aquest framework, ha estat la implementació de les operacions asíncrones, fent possible d'una manera molt simple la divisió de les tasques entre diferents processos, creant així programes capaços d'aprofitar al 100% el multi-threading de les CPU's modernes.
També al ser un entorn totalment open source la comunitat ha influenciat d'una manera extremadament eficient el avanç d'aquest entorn, fent possible la unificació de molts entorns en un únic. Gràcies a la contribució de la comunitat es va crear una versió multi-plataforma de la màquina virtual del .NET anomenada «Mono», fent possible l'execució del codi .NET en totes les plataformes més importants del mercat (UWP, Linux, Android, iOS, etc...) que suposa un gran avanç pel que és la popularitat i la utilitat d'aquest framework. Gràcies a aquesta col·laboració es va crear el framework Xamarin[2] que serà utilitzat per crear l'aplicació mòbil multi-plataforma d'aquest projecte. Cal esmentar que amb l'aparició de .NET Core, l'esforç tant del equip desenvolupador del mateix, com l'equip de Mono, com la comunitat han posat els seus esforços en la unificació dels 2 entorns traient el millor dels dos (qualitat i optimització del .NET Core i multiplataformitat del Mono).
Les aplicacions creades amb quest framework estan caracteritzades pel seu elevat rendiment sobre tot en contextos on es pugui utilitzar al màxim la capacitat del llenguatge de separar la feina en tasques i en diferents processos.

En el cas d'aquest projecte utilitzarem el llenguatge de programació C#.

8.2 Programa monitoritzador

Per tal de optimitzar al màxim el procés del desenvolupament de la solució, he optat per començar-lo per la part del programa monitoritzador capaç de connectar-se per port Ethernet a una PLC i llegir el contingut de la seva memòria. El procés d'implementació d'aquesta part l'he dividit en diferents parts.

8.2.1 Protocol Modbus

El primer pas per implementar aquesta part, es ser capaços d'establir comunicació amb un PLC i per fer-ho hem de seguir el protocol de comunicació Modbus. Inicialment la idea era implementar el protocol des de 0, però al final, he optat per aprofitar les avantatges de treballar en entorn de .NET amb el seu gestor de paquets "nuget". Aquest gestor permet descarregar mòduls fets per altre gent per tal d'augmentar la productivitat del teu codi.
Durant la primera recerca dels paquets disponible m'he adonat del fet que el entorn de programació basat en .NET Core, no té compatibilitat amb les modalitats anteriors del entorn, per tant el grup de mòduls possibles a utilitzar es veu restringit. També cal esmentar que la majoria de mòduls que implementen el protocol Modbus, son una paquets molt vells que no s'han actualitzat durant els últims anys, per tant la cerca d'un mòdul compatible amb la meva solució ha estat una tasca molt consumidora temporalment. També s'ha de dir que no tots els paquets, fins hi tot els actualitzats al entorn .NET Core, no ens donaven les opcions suficients com per poder utilitzar-los en el nostre projecte. Molt dels paquets, implementaven el protocol Modbus sobre la comunicació sèrie, no Ethernet, i els que si que l'implementaven, no ens donaven la API suficient com per poder llegir tota la memòria del PLC.
Finalment després de provar uns quants paquets m'he quedat amb el mòdul anomenat ModbusTcp[6]. Aquest mòdul té totes les característiques que necessito, però m'he trobat amb el problema de la compatibilitat amb els paquets, ja que, la llibreria tot i ser compatible amb .NET Core, no s'ha actualitzat des de fa mig any, per tant els paquets base de la meva solució eren molt més nous que el paquets amb quins s'ha creat la llibreria, fent-los incompatibles. Però com que la llibreria es open source i el seu codi font es accessible des de la plataforma GitHub, m'ho he pogut descarregar i posar-ho a dintre de la meva solució, actualitzant tots els paquets necessaris.

Fig. 21: El paquet utilitzat per implementar el protocol de comunicació

8.2.2 Connexió amb el PLC

Un cop triada i estabilitzada la llibreria que implementa el protocol de connexió, ha arribat el temps de integrar-la en el nostre programa principal. La llibreria funciona en base a un client HTTP que estableix la comunicació amb el PLC, i a part implementa els missatges i trames de Modbus. Per tant la inicialització de la llibreria es fa de la següent manera:

```
_modbusClient = new ModbusClient(_configModel.PLCConnectionIP,
                                 _configModel.PLCConnectionPort,
                                 MODBUS_CALL_TIMEOUT);
_modbusClient.Init();
```

Fig. 22: Inicialització del client de la llibreria ModbusTCP

Com podem veure en la anterior captura (Fig. 22) he creat un objecte anomenat "_configModel"que conté la configuració de la aplicació que més endavant s'utilitzarà per emmagatzemar els paràmetres d'execució definides per l'usuari. Aquesta classe conté les següents propietats:

```
public class ConfigModel
{
    public int ReadPeriod { get; set; }
    public int PLCConnectionPort { get; set; }
    public string PLCConnectionIP { get; set; }
    public string APIConnectionURL { get; set; }
    public int ReadStartPosition { get; set; }
    public int ReadEndPosition { get; set; }
    public int SamplesPerRequest { get; set; }
}
```

Fig. 23: La classe ConfigModel, que conté la configuració inicial de la aplicació

En aquest específic cas, utilitzem l'adreça IP on està connectat el nostre PLC i a quin port

el podem trobar. A part hem d'especificar el temps de timeout de la connexió, que en aquest cas es de 4s.
En el següent pas hem d'establir una rutina de lectura de la memòria del PLC, que es farà cada X especificat temps. La declaració d'aquesta rutina, la he decidit definir amb un timer que executa una funció quan passa el temps especificat.

```
var timer = new
    Timer(TimeSpan.FromSeconds(_configModel.ReadPeriod).TotalMilliseconds)
{
    AutoReset = true
};
timer.Elapsed += Timer_Elapsed;

timer.Start();
```

Fig. 24: Definició del timer que llegirà la memòria del PLC cada X temps

Com podem veure en la figura anterior (Fig. 24) el temps d'execució de la rutina, ve definit pel paràmetre "ReadPeriod"del "_configModel".

Un cop definit el timer, hem de definir la funció que llegeix la memòria. Per fer-ho utilitzem la API donada per la llibreria "modbusTCP"que ens permet la lectura de la següent manera:

```
List<short> newResult =
    await _modbusClient.ReadRegistersAsync(_configModel.ReadStartPosition,
                                           _configModel.ReadEndPosition);
```

Fig. 25: Exemple de lectura de dades de la memòria del PLC

Com podem veure (Fig. 25) a la funció lectora li hem d'indicar el espai que volem llegir, indicant la posició inicial i final de lectura. Els dos paràmetres ens vindran definits pel objecte de configuració. Aquesta funció ens retorna una llista d'elements de tipus double[7] de 16 bits, que es la mida de les cel·les d'un PLC.

8.2.3 Definició dels paràmetres

Per tal de fer el programa el més flexibles possible, hem de fer que el programa es pugui executar utilitzant paràmetres d'execució de la línia de comandes. Per aconseguir-lo utilitzaré una llibreria proveïda per Microsoft que facilita la implementació de funcions bàsiques de un programa de línia de comandes.
La llibreria utilitzada es la CommandLineUtils[8] que ens facilita la implementació de les funcions conegudes en totes les aplicacions de línia de comandes.

```
var app = new CommandLineApplication();

app.Name = "TFMBoardApp";

app.Description = ".NET Core console application created with the purpose of
    reading data from PLC memory.";

app.ExtendedHelpText = "This app can read the indicated range of PLC memory, and
    send to the specified endpoint";

app.HelpOption("-?|-h|--help");

app.VersionOption("-v|--version", () =>
{
return string.Format("Version {0}",
                     Assembly
                     .GetEntryAssembly()
                    .GetCustomAttribute<AssemblyInformationalVersionAttribute>()
                    .InformationalVersion);
});
```

Fig. 26: Implementació de les funcions bàsiques de la nostre aplicació de la línia de comandes

En la figura anterior (Fig. 26) podem veure la implementació de les funcions bàsiques de la aplicació, per donar més informació sobre aquesta al usuari. En el següent pas volem definir quins paràmetres es podran definir en la nostre aplicació. ho podem veure en la següent captura:

```
app.Option("-readPeriod <optionvalue>",
           "Memory read period, specified in seconds",
           CommandOptionType.SingleValue);

app.Option("-startPosition <optionvalue>",
           "First position of the PLC memory to read",
           CommandOptionType.SingleValue);

app.Option("-endPosition <optionvalue>",
           "Final position of the PLC memory to read",
           CommandOptionType.SingleValue);

app.Option("-samplesPerRequest <optionvalue>",
           "The number of samples that have to be read before they are sent to
               the API",
           CommandOptionType.SingleValue);

app.Option("-plcIP <optionvalue>",
           "PLC IP",
           CommandOptionType.SingleValue);

app.Option("-plcPort <optionvalue>",
           "PLC port",
           CommandOptionType.SingleValue);

app.Option("-apiURL <optionvalue>",
           "Base API URL where the samples will be sent, {apiURL}/api/samples",
           CommandOptionType.SingleValue);
```

Fig. 27: Implementació dels paràmetres possibles a definir en la nostre aplicació de la línia de comandes

Com podem veure, en les figures (Fig. 23) i (Fig. 27) podem parametritzar totes les variables del nostre programa. Un cop definides les variables, hem de saber parsejar-les del input del usuari. En la següent captura podem veure com ho faig utilitzant les possibilitats que ens dóna la llibreria:

```
case "readPeriod":
if (option.HasValue())
{
    Console.WriteLine($"{option.ShortName} was selected, value:
        {option.Value()}");
    _configModel.ReadPeriod = int.Parse(option.Value());
}
else
{
    _configModel.ReadPeriod = DEFAULT_READ_PERIOD;
}
break;
case "startPosition":
if (option.HasValue())
{
    Console.WriteLine($"{option.ShortName} was selected, value:
        {option.Value()}");
    _configModel.ReadStartPosition = int.Parse(option.Value());
}
else
{
    _configModel.ReadStartPosition = DEFAULT_START_POSITION;
}
break;
case "endPosition":
if (option.HasValue())
{
    Console.WriteLine($"{option.ShortName} was selected, value:
        {option.Value()}");
    _configModel.ReadEndPosition = int.Parse(option.Value());
}
else
{
    _configModel.ReadEndPosition = DEFAULT_END_POSITION;
}
break;
```

Fig. 28: Exemple del parseig dels paràmetres d'execució donats pel usuari mitjançant la línia de comandes

Com podem també veure en la captura anterior, si l'usuari decideix no definir algun dels

paràmetres, el posem per defecte seguint els valors mostrats en la següent figura:

```
private const int DEFAULT_READ_PERIOD = 5;
private const int DEFAULT_PLC_CONNECTION_PORT = 5002;
private const string DEFAULT_PLC_CONNECTION_IP = "127.0.0.1";
private const string DEFAULT_API_CONNECTION_URL =
    "https://tfmbackend.azurewebsites.net";
private const int DEFAULT_START_POSITION = 0;
private const int DEFAULT_END_POSITION = 64;
private const int DEFAULT_SAMPLES_PER_REQUEST = 1;
```

Fig. 29: Els paràmetres per defecte

Un cop finalitzat el parseig dels paràmetres, ja tenim completat l'objecte que guarda la configuració.

8.2.4 Enviament de dades

Com es pot apreciar en l'anterior figura (Fig. 29), ja tenim definida la URL de connexió al nostre Web Service on podem enviar les dades extretes del PLC.
Per fer-ho hem de crear-nos un altre client de HTTP, igual que en el cas del PLC, però la seva ruta de connexió apunta al nostre Web Service. Ho fem de la següent manera:

```
_httpClient = new HttpClient
{
    BaseAddress = new Uri(_configModel.APIConnectionURL)
};
```

Fig. 30: Creació del HttpClient responsable de la connexió amb el Web Service

Un cop definida la connexió, hem de crear tota la estructura d'enviament de dades, basada en peticions POST que utilitzen el format JSON[14] per enviar informació. Per tal de poder crear aquest tipus de petició hem de crear els models adequats que podrem serialitzar en format de JSON. Els podem veure a continuació:

```
public class SendSamplesRequest
{
    public IList<SampleModel> Samples { get; set; }
}
```

Fig. 31: Model de la petició que enviarem utilitzant el mètode POST, del HttpClient

```
public class SampleModel
{
    public int StartPosition { get; set; }
    public int EndPosition { get; set; }
    public List<SampleDataModel> Data { get; set; }
    public DateTimeOffset DateTime { get; set; }
}
```

Fig. 32: Model de la mostra creada al llegir la memòria del PLC

```
public class SampleDataModel
{
    public short Data { get; set; }
    public int MemoryPosition { get; set; }
}
```

Fig. 33: Model de les dades llegides del PLC

Com podem veure en el model de la petició (Fig. 31) la API que hem creat ens permet enviar més d'una mostra per fer tot el sistema més eficient.
En el model de la mostra (Fig. 32) podem veure que guardem la data exacte de quan s'ha llegit la mostra i les posicions quines s'han obtingut. L'ultima variable són les dades en sí que tenen el seu propi model.
En el model de les dades (Fig. 33) podem veure que a part de la dada guardem també la seva concreta posició per tal de tenir la completa seguretat de quin camp correspon a quina dada.

Un cop definits els models, hem de crear la lògica d'enviament de dades. Per fer la solució més robusta les dades només s'enviaran si aquestes són diferents que les anteriors enviades al servidor, per tal de no col·lapsar la connexió amb dades redundants. Per tal de comprovar si les dades són iguals que les anteriors, utilitzo una llibreria anomenada

AdvancedComparer[9], en la qual era el contribuïdor. Aquesta llibreria utilitza la propietat reflection[10] del entorn .NET que es capaç de treballar amb els tipus de les dades fent possible un comparador genèric, com el citat anteriorment. La lògica d'enviament de dades la podem veure a continuació:

```
List<short> newResult =
    await _modbusClient.ReadRegistersAsync(_configModel.ReadStartPosition,
                                           _configModel.ReadEndPosition);

var result = ExtendedComparer.Compare(newResult, _prevResult);

if (!result)
{
    Console.WriteLine("NEW_DATA_READ");

    WriteSamples(newResult);

    _sampleBuffer.Add(new SampleModel
    {
        Data = newResult.Select((x, i) => new SampleDataModel
        {
            Data = x,
            MemoryPosition = i
        }).ToList(),
        DateTime = DateTimeOffset.Now,
        EndPosition = _configModel.ReadEndPosition,
        StartPosition = _configModel.ReadStartPosition
    });

_prevResult = newResult;
```

Fig. 34: Comparació de les dades prèvies i les presents

```
if ((_configModel.SamplesPerRequest >= _sampleBuffer.Count) ||
    (_sampleBuffer.Count != 0))
{
    var request = new SendSamplesRequest
    {
        Samples = _sampleBuffer
    };

    var serializedRequest = JsonConvert.SerializeObject(request);

    var content = new StringContent(serializedRequest, Encoding.UTF8,
        "application/json");

    var res = await _httpClient.PostAsync(_httpClient.BaseAddress +
                                          "/api/samples", content);

    Console.WriteLine($"HTTP_REPSONSE_CODE: {res.StatusCode.ToString()}");

    if (res.IsSuccessStatusCode)
    {
        _sampleBuffer.Clear();
    }
}
```

Fig. 35: Lògica d'enviament de dades

En la Fig. 34 podem veure com funciona el comparador. El mètode WriteSamples, només escriu per pantalla les dades llegides del PLC. També podem veure que si les dades que hem llegit són diferents que les anteriors, creem el model de les mostres i ho guardem en el buffer. En la Fig. 35 podem veure la comprovació de quantes mostres hem d'enviar en cada petició, i un cop assolit aquest número, creem l'objecte de la petició i el serialitzem utilitzant el JsonConvert[11], una llibreria molt popular en el món de .NET. Un cop la petició serialitzada en forma de string en format de JSON, l'enviem utilitzant el mètode POST del nostre http-Client a la URL que s'ha definit en els paràmetres, afegint-li el sufix /api/samples, que es la direcció exacte on enviem les mostres.

Si la resposta del servidor ha sigut correcte, esborrem el buffer, en cas contrari, el guardem per la següent iteració, on es farà un altre intent d'enviament.

Amb això hem acabat la part de monitorització del PLC. En la següent apartat s'explicarà la part del servidor que gestiona les dades.

8.3 Backend

Aquesta es la part més important del projecte, ja que, es la responsable de connectar les altres parts entre elles i amb el món extern. També la implementació del Web Service suposa un gran repte degut a la varietat de clients que s'hi han de connectar, que tot i seguir un mateix protocol, tenen els seus detalls propis que el WS ha de tenir en ment. També ha sigut un repte personal, ja que, aquest ha estat el primer Web Service implementat utilitzant la tecnologia puntera de ASP.NET Core, que encara està en procés de trobar el seu lloc en el món del desenvolupament i de trobar un grup de seguidors enorme, igual que el seu antecessor ASP.NET Framework. Per tant no hi han moltes guies a seguir, i tampoc hi ha molta gent que hagi tingut problemes amb ell per tal de trobar solucions. Però es un preu molt petit comparat amb la experiència obtinguda durant el desenvolupament d'aquesta part.

Per aclarir, el Web Service és la part visible pel món exterior d'un server. És la API amb la qual podem accedir a les dades que emmagatzema, gestionar-les i fer-ne un ús sense posar en perill la seguretat del propi servidor ni de les dades que conté, ja que, un Web Service creat correctament inclou mesures de seguretat necessàries per tal de que no es pugui fer un ús inadequat del sistema. En el cas d'aquest projecte el WS no té una API molt extensa, ja que, només cal que implementi els mètodes de POST (guardar informació que prové del monitoritzador) i GET (per accedir a aquesta informació guardada) de manera genèrica, sense haver de processar les dades que conté. Addicionalment s'han implementat els mètodes DELETE i PUT però no s'ha fet ús en aquest concret cas.

Fig. 36: Logotip de ASP.NET Core

8.3.1 ASP.NET Core Web API

ASP.NET Core Web Api es un framework capaç d'implementar i exposar al usuari les interfícies necessàries per crear Web Services de tipus REST. Es un framework que forma part de la tecnologia .NET Core, per tant es mantenen totes les característiques explicades en les apartats anteriors.

En el projecte utilitzo la tercera revisió d'aquesta API, que bàsicament s'ha centrat en la simplificació de la feina que ha de fer el programador per tal de configurar/implementar les diferents funcions responsables de donar-li vida al Web Service. Les diferents característiques d'aquesta API i els seus detalls els explico aprofitant l'explicació del codi del Web Service que faig en la següent apartat.

La base del funcionament de l'API es caracteritza per un sistema de Petició-Resposta, on el programador indica quina funció s'ha d'executar quan es rep una petició d'un tipus concret (POST, GET, PUT, DELETE) i el valor de retorn d'aquesta funció, que es convertirà en la resposta que li donarà el servidor al client que ha fet la petició. Això fa que el treball amb aquesta API sigui molt intuïtiu i molt personalitzable pel que fa l'adaptació del Web Service a diferents escenaris de treball.

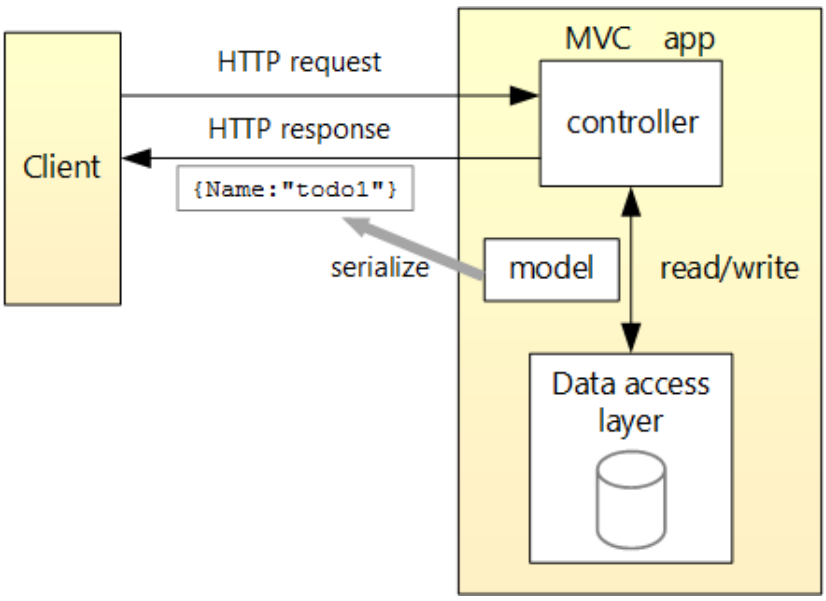


Fig. 37: Diagrama de funcionament d'una solució Core Web API

8.3.2 Entity Framework Core

El entity framework es un eina increïblement útil en el treball amb una base de dades. En aquest cas el framework fa de asignador relacional d'objectes, que permet al desenvolupador

de .NET, treballar amb una base de dades, utilitzant objectes directament de .NET, sense haver d'escriure codi d'accés en la base de dades.

En cas de EF Core el accés a les dades es realitza mitjançant un model (entitat) que definirà la nostre taula en la base de dades.

Per tal de fer consultes a la base dades, únicament hem d'agafar el context d'aquesta i mitjançant la entitat creada abans i el LINQ[13] (Language Integrated Query), podem accedir a les dades com si fos una llista normal i corrent de .NET.

Fig. 38: Logotip de Entity Framework Core

8.3.3 Creació del projecte de Web Service

Com es pot veure per implementar aquest Web Service he utilitzat el IDE Visual Studio 2017, el més ben preparat per suportar el treball amb les tecnologies .NET. Aquest programa dóna moltes plantilles pre-creades de projectes de diferents tipus, entre elles la de la Web Api. Aquest fet es una gran ajuda que li dóna l'IDE al programador, ja que, la configuració del entorn i de les classes bàsiques que s'han de crear obligatòriament, ja estan creades des del inici del projecte. També cal esmentar que el VS2017, ja des del inici, crea una solució correctament estructurada en diferents carpetes, seguint les regles generals del que és la gestió del codi del projecte i l'ordenació d'aquest.

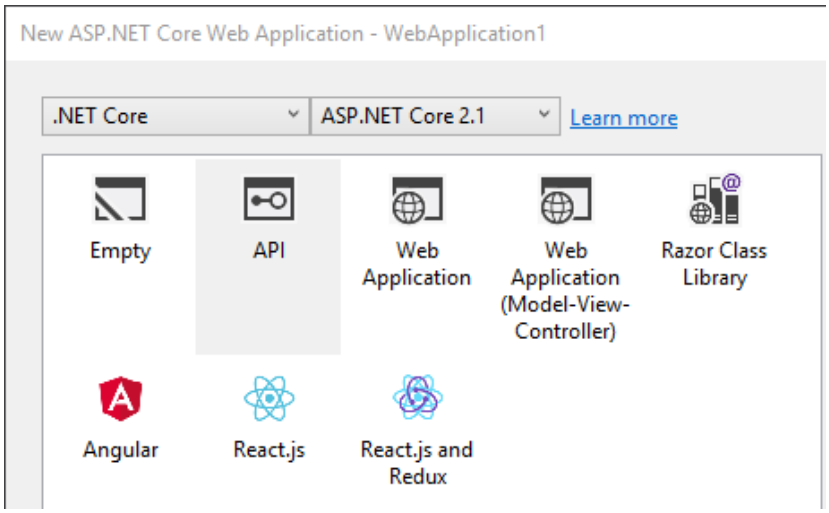


Fig. 39: Creació del projecte de ASP.NET Core Web API

Com podem apreciar en la Fig. 34, el entorn ASP.NET dóna moltes més possibilitats per crear projectes de diferents tipus i aplicacions.

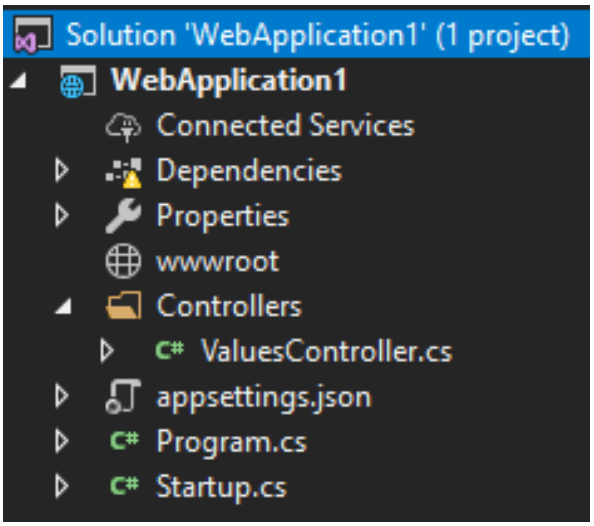


Fig. 40: Estructuració de la solució

8.3.4 Creació de les entitats

Ja sabent l'estructura general del projecte, he pogut passar a implementar les bases del funcionament del nou Web Service, d'aquesta manera he procedit a implementar la base de dades que emmagatzemarà totes les dades enviades pel nostre sistema monitoritzador.

Per fer-ho he utilitzat una de les eines més poderoses que ens dóna a disposició el Visual Studio, anomenada NuGet. Aquesta eina es un gestor de paquets que permet incorporar en el projecte diverses llibreries fent que aquestes s'incorporin directament en l'estructura del projecte. També cal esmentar que si volguéssim executar la solució en un altre ordinador, el

gestor de paquets descarregaria els components automàticament just al carregar la solució. El component que volem descarregar es el citat anteriorment Entity Framework Core, que utilitzarem per facilitar l'accés a la base de dades.

Fig. 41: El paquet de Entity framework Core

Un cop adquirit el component necessari per la base de dades, he pogut crear el model de dades que emmagatzemaré en aquesta. Un model o entitat, es una classe que ens serveix per definir, quina és l'estructura de les taules que ha de contenir la base de dades.

```
public class SampleEntity
{
    [Key]
    public int SampleId { get; set; }
    public int StartPosition { get; set; }
    public int EndPosition { get; set; }
    public IList<SampleDataEntity> Data { get; set; }
    public DateTimeOffset DateTime { get; set; }
}
```

Fig. 42: La entitat que defineix una mostra rebuda del sistema monitoritzador

```
public class SampleDataEntity
{
[Key]
public int DataId { get; set; }
public short Data { get; set; }
public int MemoryPosition { get; set; }

public SampleEntity SampleModel { get; set; }
}
```

Fig. 43: La entitat que defineix les dades rebudes del sistema monitoritzador

En el cas de la nostre base de dades tindrem 2 entitats relacionades entre elles, que es transformaran en 2 taules:

- Una que emmagatzema les mostres rebudes del nostre sistema monitoritzador, que al mateix temps farà de taula pare per la taula de les dades
- La taula de les dades, emmagatzema les dades en sí, on cada posició de la memòria es una posició en la taula. Com podem veure en la Fig. 43 Per tal d'indicar que la classe te una clau forània (clau que indica qui es el pare de la taula) hem d'afegir la classe pare a la entitat.

Els id's dels elements es generen automàticament de forma incremental.
Un cop creades les entitats hem de crear el entorn per utilitzar-les en la nostre base de dades. Per fer-ho hem d'utilitzar una classe anomenada DBContext la qual crea una instància de la nostre base de dades, i en la qual haurem de instanciar les nostres entitats de la següent forma:

```
public class SamplesContext : DbContext
{
    public SamplesContext(DbContextOptions<SamplesContext> options)
        : base(options)
    { }
    public DbSet<SampleEntity> SamplesEntity { get; set; }
    public DbSet<SampleDataEntity> SampleDataEntity { get; set; }
}
```

Fig. 44: Definició del meu propi DBContext

Un cop creat el context només falta indicar-li al Web Service que volem utilitzar aquesta base de dades. Això s'aconsegueix enregistrant aquesta-la, en el fitxer Startup.cs creat per la plantilla, en la configuració del Web Service, de la següent forma:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.
                                                Version_2_1);
    services.AddDbContext<SamplesContext>(options =>
        options.UseSqlServer(Configuration.
                             GetConnectionString("TestConnection")));
}
```

Fig. 45: Exemple de com afegir un DBContext a la API

Com podem veure en la Fig. 45, ja li estem indicant al Web Service quin servei de base de dades utilitzarem com en aquest cas es el "TestConnection", que es un recurs extern on el seu valor es la connection string, que emmagatzema totes les dades necessàries per tal d'establir una connexió remota al SQL Server.
Amb la base de dades configurada d'aquesta manera, a partir d'ara som capaços d'accedir a ella des de qualsevol lloc de la aplicació, mitjançant la injecció de dependències.

```
[Route("api/[controller]")]
[ApiController]
public class SamplesController : ControllerBase
{
    private SamplesContext _samplesContext;

    public SamplesController(SamplesContext samplesContext)
    {
        _samplesContext = samplesContext;
    }
}
```

Fig. 46: Accés a la base de dades, mitjançant la injecció de dependències

En la Fig. 46 podem veure el constructor del controlador de la API, que explicaré en la següent apartat.

8.3.5 Implementació de la API

Com que ja hem configurat correctament la base de dades, ara podem procedir a implementar el Web Service en sí. Per fer-ho necessitem crear un controlador on al seu interior crearem l'API

que podran utilitzar els clients. La plantilla creada automàticament, ja conté un controlador genèric, però jo he decidit crear un controlador propi, ajustat a les meves necessitats. Per això he començat per la implementació del constructor del controlador, que que es pot apreciar en la Fig. 46.

El paràmetre "Route"serveix per indicar-li al controlador quina ruta després de la URL bàsica del entorn haurà de posar el client per accedir a aquest controlador en concret. La variable [Controller] en aquest cas agafaria el nom del mateix controlador sense el sufix Controller, per tant "Samples".

Un cop controlat el constructor, he pogut passar a implementar els mètodes d'una API REST per tant els GET, POST, PUT, DELETE. On el GET serveix per obtenir dades, el POST per crear instàncies noves a la base de dades, PUT per actualitzar-les i DELETE per eliminar-les.

8.3.6 POST

Pel que fan els mètodes que actualitzen l'estat de la base dades, he començat pel més utilitzat d'ells, POST.

```
[HttpPost]
public void PostValue([FromBody] object value)
{
    SendSamplesRequest samplesRequest = null;
    try
    {
        samplesRequest =
            JsonConvert.DeserializeObject<SendSamplesRequest>(value.ToString());
    }
    catch
    {
        samplesRequest = null;
    }
```

Fig. 47: Conversió de les dades

En aquest mètode (Fig. 47), el primer pas a fer es el de obtenció del objecte de petició que es rep del client, per això l'hem de deserialitzar del format JSON al nostre objecte contenidor.

```
if (samplesRequest != null && samplesRequest.Samples != null)
{
    var entities = new List<SampleEntity>();

    entities = samplesRequest.Samples.Select(sample => new SampleEntity
    {
        Data = sample.Data.Select(x =>
        {
            var dataEntity = new SampleDataEntity
            {
                Data = x.Data,
                MemoryPosition = x.MemoryPosition
            };
            _samplesContext.SampleDataEntity.Add(dataEntity);

            return dataEntity;
        }).OrderBy(x => x.DataId).ToList(),
        DateTime = sample.DateTime,
        EndPosition = sample.EndPosition,
        StartPosition = sample.StartPosition
    }).ToList();

    _samplesContext.SamplesEntity.AddRange(entities);

    _samplesContext.SaveChanges();
}
```

Fig. 48: Actualització de la base de dades amb les dades noves

En el següent pas (Fig. 48) hem de crear el llistat de entitats que volem afegir a la nostre base de dades, que creem utilitzant les dades enviades pel client. Com podem veure gràcies a utilitzar LINQ i entity framework, el fet de crear instàncies noves a la taula de una base de dades, es redueix a afegir els nous elements a una llista. Fent la connexió a la base de dades completament transparent al programador.

8.3.7 DELETE

Pel que fa el mètode capaç d'eliminar les entitats de la base de dades, aquest es diferencia dels altres mètodes, ja que, el fet d'eliminar una entitat d'una taula de la base de dades, no es recursiu, per tant hem de vigilar i eliminar la entitat concreta i tots els seus fills que estiguin

referenciats a ella.

```
[HttpDelete("{id}")]
public void Delete(int id)
{
    var element = _samplesContext.SamplesEntity.Skip(id).First();
    var deleteElements = _samplesContext.SampleDataEntity.Where(x =>
        x.SampleModel == element);
    _samplesContext.SampleDataEntity.RemoveRange(deleteElements);
    _samplesContext.SamplesEntity.Remove(element);
    _samplesContext.SaveChanges();
}
```

Fig. 49: Eliminació de les dades de la base de dades

El procediment (Fig. 49) que s'ha seguit es el següent:

- Trobar la entitat pare, donada pel identificador.
- Trobar tots els elements fills que tenen la entitat pare referenciada.
- Esborrar les entitats referenciades.
- Esborrar la entitat pare.

8.3.8 GET

Pel que fa el mètode GET de la especificació REST. He decidit crear 2 endpoints, un genèric que retorna tot el contingut de la base de dades i un específic que permet retornar una sola mostra, amb un ID específic.
Es una manera molt comuna de definir endpoints d'obtenció de dades.

```
[HttpGet]
public ActionResult<string> Get()
{
    var entities = _samplesContext.SamplesEntity;

    var samples = entities.Select(entity => new SampleModel
    {
        Data = entity.Data.Select(x => new SampleDataModel
        {
            Data = x.Data,
            MemoryPosition = x.MemoryPosition
        }).ToList(),
        DateTime = entity.DateTime,
        EndPosition = entity.EndPosition,
        StartPosition = entity.StartPosition
    }).ToList();
```

Fig. 50: Accés a les dades de la base de dades

```
var response = new SendSamplesRequest
{
    Samples = samples
};

var serializedResponse = JsonConvert.
                        SerializeObject(response,
                                    Formatting.None,
                                    new JsonSerializerSettings()
                                    {
                                        ReferenceLoopHandling =
                                            ReferenceLoopHandling.Ignore
                                    });
return serializedResponse;
```

Fig. 51: Enviament de les dades extretes de la base de dades

Com podem veure en la Fig. 50, per tal d'accedir a les dades de la base de dades utilitzem el context creat anteriorment, i seguidament convertim les entitats, als models que s'enviaran en format JSON al client. Aquest pas es necessari per tal de no enviar id's al client.

En el següent pas (Fig. 51), les dades extretes i convertides, s'hauran de serialitzar al

format JSON i enviar mitjançant el model de la petició acordat entre el client i el servidor:

```
public class SendSamplesRequest
{
    public IList<SampleModel> Samples { get; set; }
}
```

Fig. 52: Model de petició

Pel que fa el mètode get que retorna una mostra específica, el seu constructor varia de la següent forma:

```
[HttpGet("{id}")]
public ActionResult<string> Get(int id)
{
    var entity = _samplesContext.SamplesEntity.Where(x => x.SampleId == id)
                                                .FirstOrDefault();

    if (entity == null)
    {
        return null;
    }
```

Fig. 53: Obtenció de la mostra amb el id indicat en la petició

Com podem veure en la Fig. 53, accés a la mostra específica es realitza a traves del mètode LINQ "Where"que donada una condició es capaç de retornar totes les entitats que la compleixen.

En aquest cas (Fig. 54) l'enviament de les dades es realitza d'una manera molt similar que abans, amb la diferència de que ja no necessitem encapsular la resposta en un model de petició, ja que, no cal fer-ho quan a dintre només hi posaríem un sol element.

```
var entityData = _samplesContext.SampleDataEntity.Where(x => x.SampleModel ==
    entity);

return JsonConvert.SerializeObject(new SampleModel
{
    Data = entityData.Select(x => new SampleDataModel
        {
            Data = x.Data,
            MemoryPosition = x.MemoryPosition
        }).ToList(),
        DateTime = entity.DateTime,
        EndPosition = entity.EndPosition,
        StartPosition = entity.StartPosition
},
Formatting.None,
new JsonSerializerSettings()
{
    ReferenceLoopHandling = ReferenceLoopHandling.Ignore
});
```

Fig. 54: Enviament de la mostra al client

Amb aquesta explicació hem terminat la implementació local del la nostre API, que ara, per fer-la accessible, des de qualsevol lloc del món, hem de publicar en un hosting/cloud. Aquest procediment s'explica en la següent apartat.

8.4 Azure

En el món dels serveis de dades i de la infraestructura computacional, cada vegada més les empreses es decideixen a portar els seus serveis al núvol, o bé complementar els seus servidors remots amb altres locals. Azure es a resposta a tot això, ja que, tracta d'una oferta de plataforma i infraestructura com a servei. Azure es podria definir com a centenars de Datacenters, amb una gran quantitat de recursos que poden ser consumits per a per a persones o empreses des de qualsevol part del món, pagant tan sols pel que fan servir.

Azure no representa tan sols emmagatzemar i montar una base de dades al núvol, sinó que a més, pretén oferir serveis avançats de plataforma, i aquesta és potser un dels principals avantatges que el diferencien dels altres productes Cloud que trobem al mercat. En definitiva, proposen una solució integral a problemes que moltes vegades són costosos i complexos per a

una empresa, ja solucionats per a Microsoft, i que es poden oferir com a servei, facilitant la seva integració amb altres eines i frameworks de la família Microsoft.

Fig. 55: Logotip d'Azure

Microsoft Azure utilitza un sistema operatiu especialitzat, que té el mateix nom, per a fer córrer en les seves çapesün clúster localitzat en els servidors de dades de Microsoft que s'encarreguen de gestionar els recursos emmagatzemats.
En diverses ocasions se l'ha descrit com una capa de núvols a la part superior d'una sèrie de sistemes que es basen en Windows Server 2008 i una versió personalitzada de Hyper-V, conegut com el Microsoft Azure Hipervisor de virtualització, per a proporcionar els serveis. La fiabilitat és controlada per Microsoft Azure Fabric Controller, amb l'objectiu de que l'entorn i els serveis no puguin caure o deixar de funcionar en el Microsoft Data Center, que proporciona la gestió de l'aplicació web, així com els recursos de memòria i el balanç de càrrega entre diferents servidors físics.
Azure proporciona una API basada en REST, HTTP i XML que permet als desenvolupadors poder interactuar amb els seus serveis. Microsoft també ofereix una biblioteca de classes pel costat del client gestionat que encapsula les funcions d'interacció amb els serveis.
A més de la interacció amb els serveis a través de l'API, els usuaris poden gestionar els serveis d'Azure mitjançant Azure Portal basat en la web, que va arribar a la disponibilitat general el desembre de 2015. El portal permet als usuaris navegar pels recursos actius, modificar la programació, llençar nous recursos, i veure les dades de monitoratge bàsiques de les màquines virtuals i serveis actius.

8.4.1 Creació del entorn i de la base de dades

Per tal de començar a treballar amb el servei en cloud Azure, primerament hem de crear-nos una compte de Microsoft la qual serà el nostre identificador com a propietari dels recursos vinculats a aquesta compte. Un cop creada la compte podem accedir al panell de control de Azure (Fig. 56), per on podem gestionar absolutament tot el relacionat amb aquest servei.

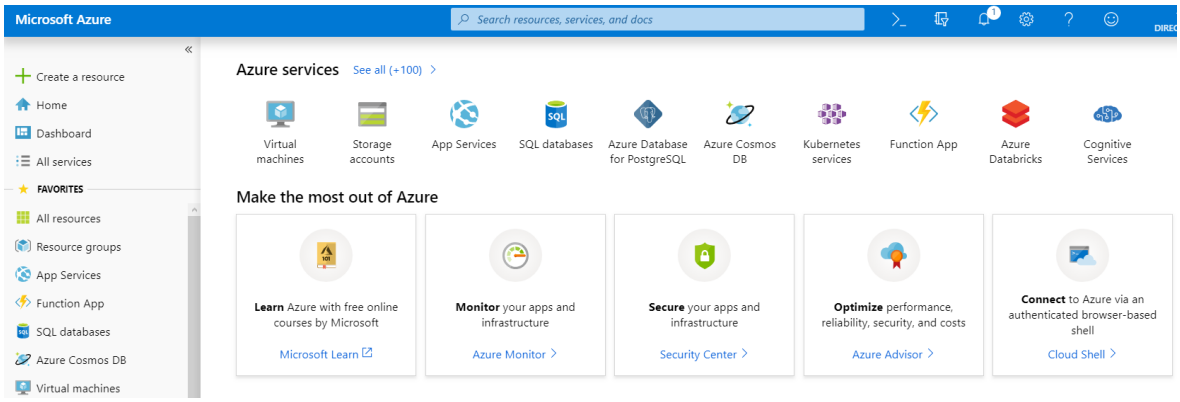


Fig. 56: Una part del panell de control de Azure

En el cas d'aquesta solució he hagut de crear els serveis[19][20] mostrats en la Fig. 57

NAME	TYPE	RESOURCE GROUP	LOCATION
samplessqlserver	SQL server	myResourceGroup	West Europe
SampleDatabase (samplessqlserver/SampleDatabase)	SQL database	myResourceGroup	West Europe
TFMBackend	Application Insights	myResourceGroup	West Europe
TFMBackend	App Service	myResourceGroup	West Europe
TFMBackendPlan	App Service plan	myResourceGroup	West Europe

Fig. 57: Els serveis creats per les necessitats de del projecte

Una breu explicació de tots ells:

- samplessqlserver: Es el servidor SQL que gestiona tota la part de comunicació amb el món exterior de la nostre base de dades
- SampleDatabase: Es la base de dades en sí, localitzada en algun dels servidors de Microsoft d'Europa de l'oest.
- TFMBackend: Es una composició de 3 serveis diferents, vinculats amb el mateix entorn.
  - Application Insights: ens facilita la gestió sobre la informació de funcionament del nostre entorn, fent un resum comprensible del que està passant amb ell al llarg del temps.
  - App Service: Es l'entorn en sí on està ubicada la nostre Web API i on podem gestionar tota la part d'accés a ella.
  - App Service plan: Es la part comptable del servei on podem gestionar totes les despeses que genera el nostre entorn.

8.4.2 Publicació de la API i de la base de dades

Un cop creats tots el recursos i serveis a Azure, podem desplegar la nostre solució Fig. 58.

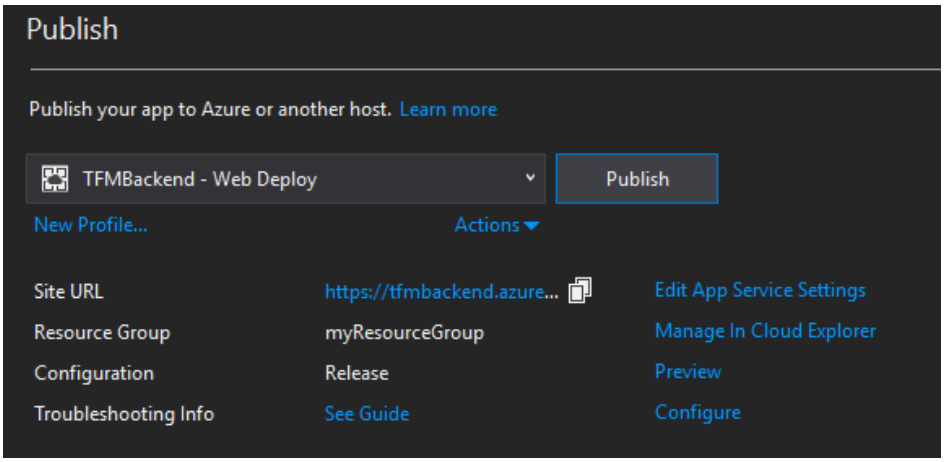


Fig. 58: Publicació del nostre servei a Azure

Pel que fa la base de dades, per desplegar-la en Azure hem d'utilitzar eines proporcionades per Entity Framework, per tal de convertir els nostres models .NET a taules de base de dades[18].

```
dotnet ef migrations add InitialCreate
```

Fig. 59: Creació de la migració inicial

```
dotnet ef database update
```

Fig. 60: Enviament de la migració a la base de dades

El procediment a seguir podem el veure en les figures 59 i 60, on el el primer creem un fitxer auto generat que es un script que crearà les nostres taules en la base de dades, i el segon puja aquest fitxer al entorn de Azure.

8.5 App mòbil

En aquesta part del document explico les meves experiències, del que ha sigut el procés d'implementació de l'aplicació mòbil, capaç de d'agafar informacions emmagatzemades en el Web Service explicat anteriorment i mostrar-les d'una forma comprensible al usuari.
He decidit seguir amb la mentalitat d'un projecte multi-plataforma que no li obliga a l'usuari

l'ús d'alguna plataforma en concret, per tant, per desenvolupar aquesta aplicació, he utilitzat un framework anomenat Xamarin. Es un framework bastant nou, en un continu desenvolupament però tot i així, ja avui en dia dóna l'estabilitat i la fiabilitat necessària com per utilitzar-lo en projectes a gran escala.

Al realitzar aquesta aplicació m'he centrat més en el seu aspecte arquitectònic, proveint una interfície simple amb unes funcionalitats concretes, però preparant la aplicació perquè sigui el més extensible i el més adaptable a funcionalitats noves, creant així un programa molt simple i fàcil d'utilitzar amb molta utilitat i possibilitats d'un posterior millorament, deixant les portes obertes per poder afegir funcionalitats noves a mida que el sistema en general sigui dopat amb noves característiques.

8.5.1 Xamarin

He decidit utilitzar el framework Xamarin per desenvolupar aquesta aplicació, perquè, aquest framework és capaç, utilitzant les tecnologies .NET, de compilar un mateix codi i una mateixa representació gràfica per diferents plataformes, entre altres iOS i Android que són les més interessants avui en dia. Al utilitzar la tecnologia .NET, aquest framework està fonamentat sobre una base sòlida i estable com és el llenguatge C# i tot l'entorn que permet utilitzar.

Fig. 61: Logotip de Xamarin

La possibilitat de poder compartir el mateix codi entre diferents plataformes es un gran avanç pel que fa el desenvolupament d'aplicacions mòbils, ja que, tant els costos de la implementació com el temps necessari per crear-la baixen considerablement eliminant la necessitat de duplicar la mateixa lògica del codi creat en diferents llenguatges per diferents plataformes.

Per descriure el funcionament d'aquest framework aprofitaré l'explicació del codi de l'aplicació creada per aquest projecte d'una manera semblant com ho he fet amb l'explicació de la part del Web Service.

8.5.2 Creació i preparació de la solució

Per desenvolupar l'aplicació he utilitzat el IDE Visual Studio 2017 el qual, igual que amb la implementació del Web Service, aporta moltes facilitats al programador creant un entorn molt agradable que intenta agilitzar al màxim totes les tasques a realitzar.
Per tant en primer lloc he començat creant una plantilla adequada pel meu projecte amb l'ajut de creador de projectes (62) de VS.

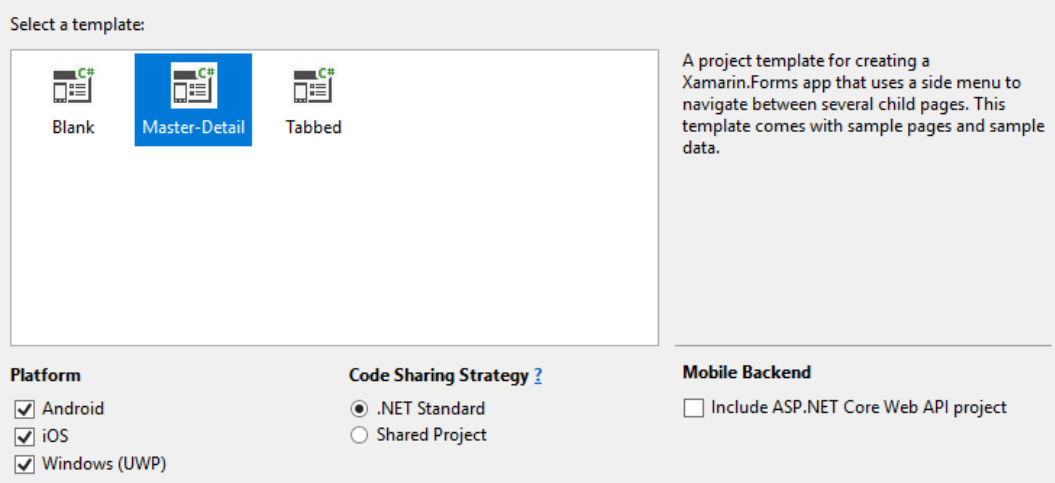


Fig. 62: Creació de la plantilla del projecte de Xamarin

Com es pot observar, estem creant la plantilla d'una aplicació Master-Detail, on la base de la app es divideix en 2 pantalles, la principal i una pantalla secundària desplegable pel costat esquerre.
Com també es pot observar, he utilitzat la tecnologia Xamarin.Forms per la creació de la interfície gràfica de l'aplicació. Es una tecnologia que permet crear una única definició de la GUI utilitzant un fitxer .xaml i compilar-lo per a diferents plataformes sense haver de canviar el codi d'aquest. En segon lloc es pot veure que he decidit crear un codi compilable amb el .NET Standard. Aquesta estratègia permet crear controladors i models compartibles entre totes les plataformes i tots els projectes compatibles amb el .NET Standard, però també deixa la possibilitat d'implementar coses natives en els projectes individuals per a cada plataforma, si fos necessari.

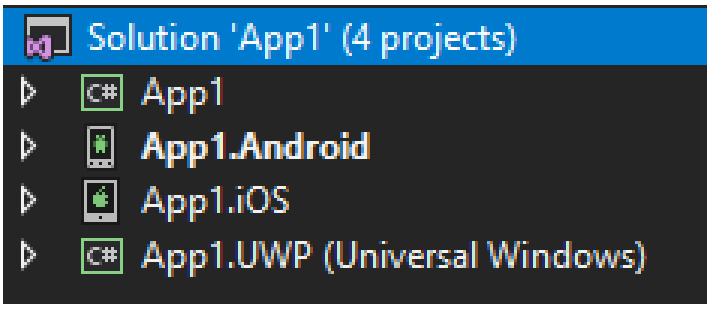


Fig. 63: Projectes creats a la plantilla

Mirant la Fig. 63 es pot apreciar els projectes creats per la plantilla, comú compatible amb .NET Standard, un per Android, un per iOS i un per la plataforma Windows (UWP). En el cas de la meva aplicació he utilitzat només classes del projecte comú, ja que, no ha sigut necessària la implementació de coses natives de cada plataforma.

8.5.3 Implementació del MvvmCross

El funcionament normal d'una aplicació feta en Xamarin, es basa en el model MVC[21] on la capa de la vista i del controlador estan totalment vinculades entre sí, estan ubicats en el mateix projecte. Això té l'avantatge de ser una arquitectura molt simple d'entendre i de desenvolupar, però té certes limitacions pel que fa la reutilització del codi en altres projectes, o la seva flexibilitat, ja que la lògica i la part visual tenen dependències entre si, sense cap separació/abstracció.

Per resoldre aquest problema ens ve com l'ajuda el esquema MVVM[22], que posa una altre capa en el nostre projecte, on la vista i el seu controlador estan totalment separades mitjançant bindings per tal de donar una certa abstracció entre la vista i la lògica, permetent així una reutilització de la lògica, amb altres vistes, sempre i quan implementin els bindings necessaris.

Fig. 64: Logotip del paquet MvvmCross

Per tal d'implementar aquest esquema en la nostre aplicació, hem utilitzat el paquet anomenat MvvmCross[23], que facilita la separació de les peces de la nostre solució, donant solucions preparades per crear bindings i interfícies capaces de abstreure les nostres parts

visuals.

Per tal d'implementar-lo en la solució hem d'instal·lar-nos els paquets (Fig. 65) preparats per MvvmCross i fer unes quantes modificacions en la solució/codi.

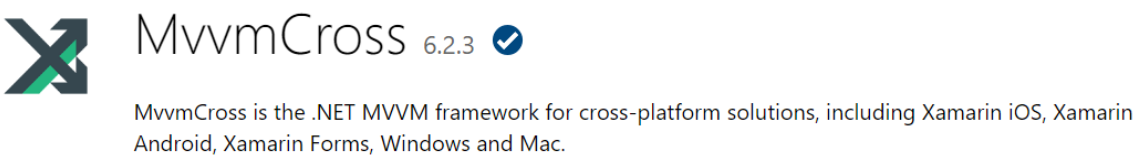


Fig. 65: Paquet nuget de MvvmCross

Per començar hem de modificar la nostre solució per tal d'incloure els nous projectes que ens serviran com capes separadores entre diferents elements.

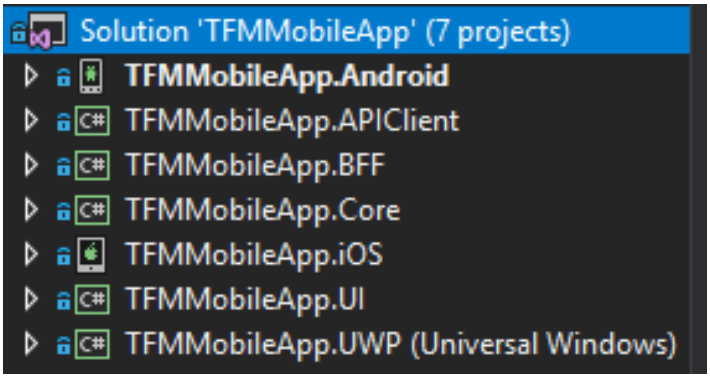


Fig. 66: Projectes creats per implementar l'esquema MVVM

Un cop creats el projectes, hem de començar a vincular els nous projectes amb la app inicial. Primer hem de crear un fitxer anomenat App.cs que serà el nostre punt inicial de la aplicació.

```
public class App : MvxApplication
{
    public override void Initialize()
    {
        //Identity
        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<ITokenSettings,
            TokenSettings>();
        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<ITokenService,
            TokenService>();

        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<IBaseSettings,
            BaseSettings>();
        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<IMenuService,
            MenuService>();
        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<ISamplesService,
            SamplesService>();

        // BFF
        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<ISamplesWebService,
            SamplesWebService>();

        // Essentials
        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<IAppInfo,
            AppInfoImplementation>();
        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<IConnectivity,
            ConnectivityImplementation>();
        Mvx.IoCProvider.LazyConstructAndRegisterSingleton<IPreferences,
            PreferencesImplementation>();

        CreatableTypes()
            .EndingWith("Service")
            .AsInterfaces()
            .RegisterAsLazySingleton();

        RegisterAppStart<MainMasterDetailViewModel>();
    }
}
```

Fig. 67: Implementació de la inicialització de la aplicació utilitzant MvvmCross

Aquest fitxer (Fig. 67) té 3 apartats principals

- Primerament es registren els serveis i les seves interfícies, per tal de poder utilitzar-les al llarg de l'aplicació utilitzant la injecció de dependències.
- Seguidament es registren els serveis bàsics de l'aplicació.
- I per últim es defineix quin es el punt inicial de la nostre aplicació

Amb això ja hem configurat la aplicació per funcionar amb el nou paquet, però com podem apreciar en la figura 66 s'han creat 2 projectes addicionals no relacionats amb l'esquema MVVM:

- APIClient: Es el projecte responsable d'implementar tota la lògica responsable amb la connexió amb el món exterior de la app.
- BFF: Es la capa d'abstracció entre la lògica de negoci, i els endpoints amb quins connectarem la nostre app al Web Service.

Un cop explicada la estructuració del projecte, podem passar a implementar les parts de l'aplicació.

8.5.4 Implementació de la GUI

Ja creada la solució, he pogut començar a dissenyar l'aplicació. En aquest cas he preferit primerament començar per la implementació de la part gràfica de l'aplicació i després implementar els controladors necessaris per donar funcionalitats a les parts gràfiques dissenyades anteriorment.
Primerament he començat amb la implementació d'un menu principal que de moment només tindrà una sola opció, per veure les mostres rebudes del Backend.

```
<views:BaseView x:Class="TFMMobileApp.UI.Views.Menus.MainMenuView"
xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:controls="clr-namespace:TFMMobileApp.UI.Controls;assembly=TFMMobileApp.UI"
xmlns:views="clr-namespace:TFMMobileApp.UI.Views;assembly=TFMMobileApp.UI"
xmlns:menus="clr-namespace:TFMMobileApp.UI.Views.Menus;assembly=TFMMobileApp.UI"
Title="Samples App"
AutomationId="MainMenuPage">
    <Grid>
        <controls:CommandListView x:Name="menuList"
                            Command="{Binding MenuItemSelectedCommand}"
                            ItemsSource="{Binding MenuListItems}">
        <ListView.ItemTemplate>
            <DataTemplate>
                <menus:MenuItemViewCell/>
            </DataTemplate>
        </ListView.ItemTemplate>
        </controls:CommandListView>
    </Grid>
</views:BaseView>
```

Fig. 68: Definició de la vista del menú principal

En la figura 68 podem veure en pràctica la implementació del esquema MVVM, on tots els camps dinàmics (elements de la llista, i la comanda a executar al pitjar un element) es comuniquen a la lògica mitjançant bindings, permetent així la reutilització del la mateixa vista per diferents contextos lògics.

En la mateixa mostra del codi podem veure que per definir la vista del ítem de la llista, utilitzem un fitxer extern (Fig. 69), per tal de mantenir un codi ben estructurat i reutilitzable.

```
<ViewCell x:Class="TFMMobileApp.UI.Views.Menus.MenuItemViewCell"
xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml">
<ViewCell.View>
    <Frame BackgroundColor="White"
           BorderColor="White"
           CornerRadius="4"
           HasShadow="true"
           Padding="0">
        <StackLayout Orientation="Horizontal"
                     Padding="8"
                     Spacing="16">
            <Label FontSize="22"
                     HorizontalTextAlignment="Center"
                     Text="{Binding Name}"
                     TextColor="Blue"
                     VerticalTextAlignment="Center" />
        </StackLayout>
    </Frame>
</ViewCell.View>
</ViewCell>
```

Fig. 69: Definició de la vista del ítem del menú principal

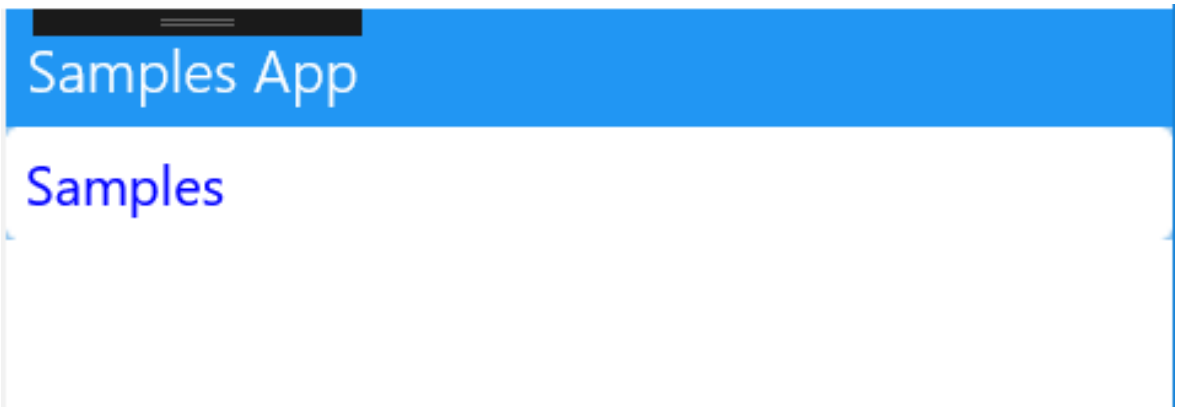


Fig. 70: Vista del menú principal

Les captures que representen les vistes són provinents de la plataforma UWP, però la aplicació sense cap mena de canvi, pot executar-se en android o iOS.

En la figura 70, podem veure com es veu el menú amb un sol element.

El següent pas es el d'implementar la llista de les mostres que rebem del nostre backend. La manera d'implementar-ho es molt similar a la del menú.

```
<views:BaseView xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="TFMMobileApp.UI.Views.Samples.SamplesView"
xmlns:views="clr-namespace:TFMMobileApp.UI.Views;assembly=TFMMobileApp.UI"
xmlns:controls="clr-namespace:TFMMobileApp.UI.Controls;assembly=TFMMobileApp.UI"
Title="Samples">
<views:BaseView.ToolbarItems>
    <ToolbarItem Command="{Binding GetSamplesCommand}"
            Text="Retrieve Samples"
            Icon="{extensions:PlatformImage  SourceImage='ic_refresh'}"
            AutomationId="toolbarGetSamples"/>
</views:BaseView.ToolbarItems>

<controls:CommandListView ItemsSource="{Binding Samples}"
               Command="{Binding SampleSelectedCommand}">
<ListView.ItemTemplate>
<DataTemplate>
    <templates:SampleTemplate/>
</DataTemplate>
</ListView.ItemTemplate>
</controls:CommandListView>
</views:BaseView>
```

Fig. 71: Definició de la vista de les mostres

```
<ViewCell xmlns="http://xamarin.com/schemas/2014/forms"
          xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
          xmlns:converters="clr-namespace:TFMMobileApp.UI.Converters;
          assembly=TFMMobileApp.UI"
          x:Class="TFMMobileApp.UI.Templates.SampleTemplate">
  <ViewCell.View>
        <Grid HorizontalOptions="FillAndExpand"
            VerticalOptions="FillAndExpand">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>

            <Label Grid.Row="0"
                Grid.Column="0"
                Text="{Binding DateTime,
                    Converter={converters:DateTimeConverter}}"/>
            <Label Grid.Row="0"
                Grid.Column="1">
                <Label.FormattedText>
                    <FormattedString>
                        <Span Text="Positions: "/>
                        <Span Text="{Binding StartPosition}"/>
                        <Span Text=" - "/>
                        <Span Text="{Binding EndPosition}"/>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
        </Grid>
    </ViewCell.View>
</ViewCell>
```

Fig. 72: Definició de la vista del ítem de les mostres

La única diferència que podem apreciar en aquesta definició (Fig. 71) es que aquí tenim un toolbar item que ens permetrà refrescar la llista de les mostres (Fig. 73).

Fig. 73: Vista de les mostres rebudes del backend

Per últim ens queda poder veure les dades emmagatzemades en cada mostra, per aquest motiu hem d'implementar una altre vista que contindrà una llista.

```
<views:BaseView xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:views="clr-namespace:TFMMobileApp.UI.Views;assembly=TFMMobileApp.UI"
xmlns:templates="clr-namespace:TFMMobileApp.UI.Templates;
assembly=TFMMobileApp.UI"
xmlns:controls="clr-namespace:TFMMobileApp.UI.Controls;assembly=TFMMobileApp.UI"
x:Class="TFMMobileApp.UI.Views.Samples.SampleDetailView"
Title="Sample Detail">
<controls:CommandListView ItemsSource="{Binding SampleData}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <templates:SampleDetailTemplate/>
        </DataTemplate>
    </ListView.ItemTemplate>
</controls:CommandListView>
</views:BaseView>
```

Fig. 74: Definició de la vista del detall d'una mostra

```
<ViewCell xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            x:Class="TFMMobileApp.UI.Templates.SampleDetailTemplate">
    <ViewCell.View>
        <StackLayout>
            <Label>
                <Label.FormattedText>
                    <FormattedString>
                        <Span Text="Mem Pos "/>
                        <Span Text="{Binding MemoryPosition}"/>
                        <Span Text=": "/>
                        <Span Text="{Binding Data}"/>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
        </StackLayout>
    </ViewCell.View>
</ViewCell>
```

Fig. 75: Definició de la vista del item del detall d'una mostra

La definició de les figures 74 i figures 75 té el resultat que podem veure en la Fig. 76

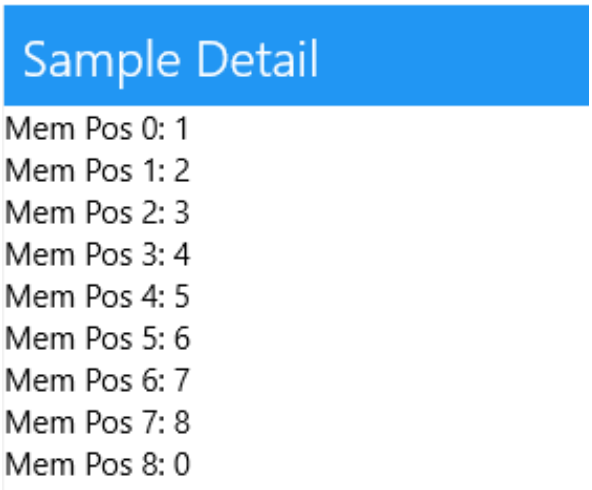


Fig. 76: Vista del detall de les mostres rebudes del backend

8.5.5 Implementació de la lògica

Un cop finalitzada aquesta part de l'aplicació, m'he posat a implementar la seva lògica, per tant la part responsable de donar-li dinamisme i accions a tots els components gràfics.

Primer de tot s'han d'inicialitzar (Fig. 77) totes les estructures necessàries per tal de que

l'aplicació funcioni correctament, que en aquest cas es redueix a la definició de la URL del nostre Web Service, per tal de simplificar la connexió a aquest més endavant. També hem, de definir els ViewModels inicials (menú principal i el lateral) que es posaran a primer pla quan l'aplicació s'executa.

```
public MainMasterDetailViewModel(IMvxLogProvider logProvider,
                        IMvxNavigationService navigationService)
    : base(logProvider, navigationService)
{ }

public override void ViewAppeared()
{
    base.ViewAppeared();

    ApiClient.Initialize("https://tfmbackend.azurewebsites.net/");
}

public override void FirstTimeAppearing()
{
    MvxNotifyTask.Create(async () => await this.InitializeViewModels());
}

private async Task InitializeViewModels()
{
    await NavigationService.Navigate<MasterMenuViewModel>();
    await NavigationService.Navigate<MainMenuViewModel>();
}
```

Fig. 77: Inicialització de les estructures de la aplicació

Seguidament he hagut d'implementar la lògica del menú per poder fer navegació per dintre de l'aplicació. Primerament hem de crear el ViewModel responsable d'implementar el bindings requerits de la vista.

```
public MainMenuViewModel(IMvxLogProvider logProvider,
                         IMvxNavigationService navigationService,
                         IMenuService mainMenuService)
    : base(logProvider, navigationService)
{
    _mainMenuService = mainMenuService;

    MenuItemSelectedCommand = new MvxCommand<MenuListModel>(
        async (item) => await ExecuteMenuItemSelectedCommand(item));
}
```

Fig. 78: Constructor del ViewModel de la vista del menú

```
private async Task ExecuteMenuItemSelectedCommand(MenuListModel item)
{
    await FromBusyContext(async () =>
    {
        await _mainMenuService.NavigateToSelection(item);
    });
}
```

Fig. 79: Funció que s'executarà al pitjar sobre un ítem de la llista

En el constructor d'aquest ViewModel (Fig. 78) hem de definir quina funció (Fig. 79) s'executarà al pitjar un element de la llista.

```
public override async Task Initialize()
{
    MenuListItems = new
        ObservableCollection<MenuListModel>(_mainMenuService.GetItems());

    await base.Initialize();
}
```

Fig. 80: Funció Initialize del ViewModel

Després només ens falta definir la llista de ítems que omplirà la llista del menú (Fig. 80). Com podem veure en aquesta funció s'utilitza el _mainMenuService que té l'estructura definida en la figura 81.

```
public IEnumerable<MenuListModel> GetItems()
{
    var items = new List<MenuItemType>
    {
        MenuItemType.Samples
    };
    return items.OrderBy(s=>s).Select(GetMenuItem);
}

public async Task NavigateToSelection(MenuListModel selectedListItem)
{
    if (selectedListItem.Parameter == null)
    {
        await _navigationService.Navigate(selectedListItem.ViewModel);
    }
    else
    {
        await _navigationService.Navigate(selectedListItem.ViewModel,
            selectedListItem.Parameter);
    }
}
```

Fig. 81: Els mètodes encapsulats a dintre del mainMenuService

Un cop definida la lògica del menú, podem passar a definir la lògica de la vista de les mostres, que s'inicialitza igual que el menú principal amb la diferència que ara hem de definir 2 comandes (Fig. 82) a executar (selecció del element de la llista i refresc de les mostres)

```
public SamplesViewModel(IMvxLogProvider logProvider,
                        IMvxNavigationService navigationService,
                        ISamplesService samplesService)
    : base(logProvider, navigationService)
{
    _samplesService = samplesService;

    GetSamplesCommand = new MvxCommand(async () => await
        ExecuteGetSamplesCommand());
    SampleSelectedCommand = new MvxCommand<SampleModel>(async (sample) => await
        ExecuteSampleSelectedCommand(sample));
}
```

Fig. 82: Constructor del ViewModel de la vista del menú

I a continuació podem veure les funcions en sí:

```
private async Task ExecuteGetSamplesCommand()
{
    await FromBusyContext(async () =>
    {
        Samples = new ObservableCollection<SampleModel>(await
            _samplesService.GetSamples());
    });
}

private async Task ExecuteSampleSelectedCommand(SampleModel sample)
{
    await NavigationService.Navigate<SampleDetailViewModel,
        IList<SampleDataModel>>(sample.Data);
}
```

Fig. 83: Funcions definides en les 2 comandes del constructor

La funció definida en la figura 83 utilitza el _samplesService que s'explicarà en la apartat dedicada a la connexió al Web Service.

Per últim ens falta definir la lògica de la vista del detall d'una mostra que es la més simple de totes. Ja que com podem veure en la figura 83, aquesta vista ja rep totes les dades per paràmetre (sample.Data). Per això hem d'utilitzar el mètode "Prepare"(Fig. 84) utilitzat per rebre dades enviades per un altre ViewModel.

```
public override void Prepare(IList<SampleDataModel> data)
{
    SampleData = new ObservableCollection<SampleDataModel>(data);

    base.Prepare();
}
```

Fig. 84: Funció Prepare que omple la llista de les dades de la mostra

Amb això hem acabat d'explicar la lògica dels ViewModels, en l'apartat següent ens centrarem en la obtenció de les mostres des del Web Service.

8.5.6 Connexió al Web Service

La connexió al Web Service en el nostre projecte hem decidit abstreure-la a un projecte de la solució apartat per tal de facilitar els possibles canvis. Per tant en primer pas hem d'implementar el servei (Fig. 86) al quin cridarà el nostre ViewModel.

```
public async Task<ICollection<SampleModel>> GetSamples()
{
    var samples = await _samplesWebService.GetSamples();

    if (samples?.Samples != null)
        return samples.Samples.Select(x => SampleMapper.Map(x)).ToList();
    else
    {
        return new List<SampleModel>();
    }
}
```

Fig. 85: Funció del servei de les mostres que crida als mètodes responsables de la connexió al Web Service.

```
public async Task<ICollection<SampleModel>> GetSamples()
{
    var samples = await _samplesWebService.GetSamples();

    if (samples?.Samples != null)
        return samples.Samples.Select(x => SampleMapper.Map(x)).ToList();
    else
    {
        return new List<SampleModel>();
    }
}
```

Fig. 86: Funció del servei de les mostres que crida als mètodes responsables de la connexió al Web Service.

En aquest servei s'utilitza un mapper responsable de mapejar els objectes DTO[24] a objectes que utilitzarem en els ViewModels.
Per l'altre banda la implementació de la connexió en sí s'implementa com ho podem veure en la següent figura:

```
public async Task<SamplesRequestResponse> GetSamples()
{
    return await ApiClient.Get<SamplesRequestResponse>("api/samples");
}
```

Fig. 87: Funció que crida al Web Service

En la crida d'aquesta funció només hem d'especificar la ruta que hem d'afegir a la URL base definida anteriorment.

8.5.7 Interpretació de les dades

Com l'últim punt de la part mòbil he hagut implementar una bàsica interpretació de les dades rebudes del servidor. Per utilitzar com l'exemple he decidit que la posició de la memòria del PLC nr.0 ens indica si la màquina està apagada (0) o en funcionament (1).
Per tal de mostrar gràfiques de les dades rebudes, he utilitzat la llibreria Microcharts[25] (Fig. 88), que facilita d'una manera molt considerable el treball amb dades i figures.

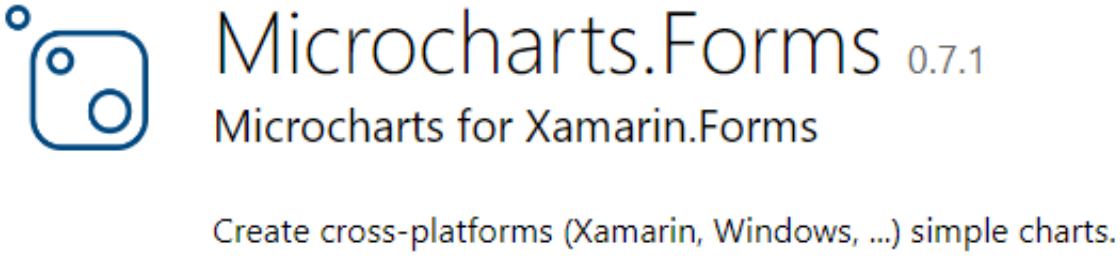


Fig. 88: La llibreria Microcharts

Un cop instal·lat el nuget de la llibreria, he hagut d'implementar una nou ViewModel i la seva vista per tal de poder mostrar els gràfics.
Per la part visual (Fig. 89), la he creat seguint el mateix patró que les altres vistes del projecte.

```
<views:BaseView xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:views="clr-namespace:TFMMobileApp.UI.Views;assembly=TFMMobileApp.UI"
xmlns:microcharts="clr-namespace:Microcharts.Forms;assembly=Microcharts.Forms"
x:Class="TFMMobileApp.UI.Views.ChartView"
Title="Chart">
    <Grid>
        <microcharts:ChartView x:Name="chartView"  />
    </Grid>
</views:BaseView>
```

Fig. 89: Definició de la vista del gràfics

```
public partial class ChartView
{
    public ChartView()
    {
        InitializeComponent();
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();

        var vm = (ChartViewModel)ViewModel;

        var chart = new PieChart() { Entries = vm.Entries };

        this.chartView.Chart = chart;
    }
}
```

Fig. 90: El controlador de la vista dels gràfics

Com podem apreciar en les figura 90, quan la vista apareix agafem la llista de mostres des del ViewModel (Fig. 91) que volem mostrar en el gràfic.

```
public override void Prepare(IList<SampleModel> data)
{
    var samples = new List<SampleModel>(data);

    var onSamples = data.Where(x => x.Data[0]?.Data == 1).Count();
    var offSamples = data.Where(x => x.Data[0]?.Data == 0).Count();

    var entries = new List<ChartEntry>
    {
        new ChartEntry(onSamples)
        {
            Label = "On Samples",
            ValueLabel = onSamples.ToString(),
            Color = SKColor.Parse("#266489")
        },
        new ChartEntry(offSamples)
        {
            Label = "Off Samples",
            ValueLabel = offSamples.ToString(),
            Color = SKColor.Parse("#68B9C0")
        }
    };

    Entries = entries;

    base.Prepare();
}
```

Fig. 91: El controlador de la vista dels gràfics

Com podem veure, quan rebem les mostres del ViewModel de les mostres, el que fem es agafar i fer un recompte de totes les mostres on la dada en la primera posició de la memòria es igual a 1 o a 0.

El resultat de la execució d'aquest codi, amb dades reals de la base de dades es el següent:

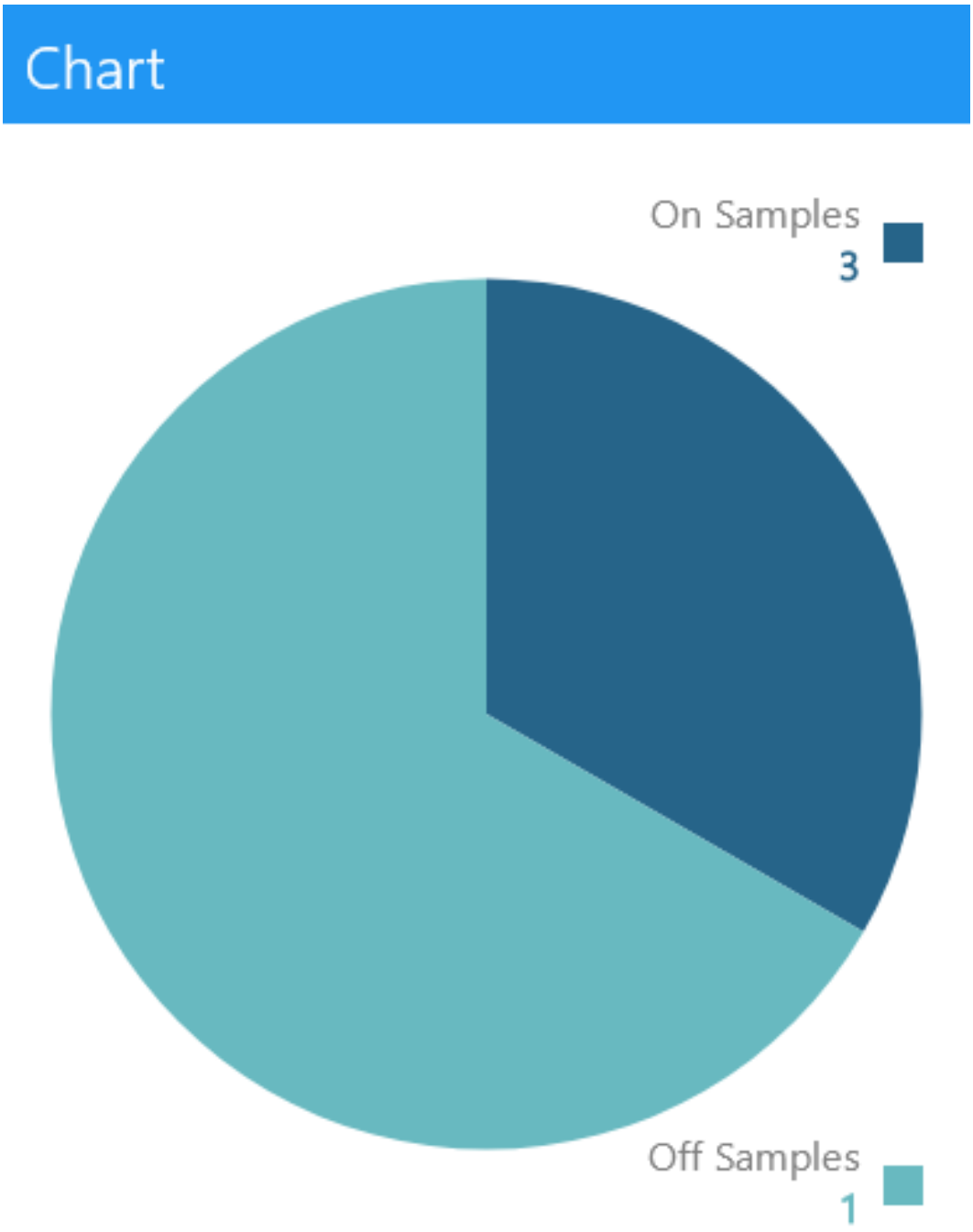


Fig. 92: La gràfica resultant

9 CREACIÓ DE LA MEMÒRIA

Per tal d'escriure l'entregable he decidit donar-li oportunitat a un entorn més avançat d'edició de documents anomenat Latex[26], que permet una sofisticació molt més extensa sobre els documents editats que les eines que he tingut l'oportunitat de treballar en el passat.
En la primera impressió semblava un repte molt important, però després d'escriure aquest document puc afirmar amb tota la confidència que es un editor molt agradable que al final facilita molt la vida del autor, automatitzant molts processos en la escriptura d'un document oficial com aquest.
Per poder escriure aquest document, en aquest format, he hagut d'utilitzar vàries eines que explico a continuació.

9.0.1 Visual Studio Code

Visual Studio Code[27] és un editor de codi font desenvolupat per Microsoft per a Windows, Linux i macOS. Inclou suport per a la depuració, control integrat de Git, ressaltat de sintaxi, finalització intel·ligent codi, fragments i refactorització de codi. També és personalitzable, de manera que els usuaris poden canviar el tema de l'editor, les adreces de teclat i les preferències. És gratuït i de codi obert, encara que la descàrrega oficial està sota programari propietari. Visual Studio Code es basa en Electron, un framework que s'utilitza per implementar aplicacions NODE.JS per a l'escriptori, que s'executa en el motor de disseny Blink.

Fig. 93: Logotip del Visual Studio Code

En el context d'aquest document, el VSC m'ha servit com editor de text, ja que, gràcies a la multitud de components que se li poden instal·lar, he pogut crear un entorn d'edició de documents molt agradable, integrant en ell la gestió del codi .tex amb Git, el compilador de Latex i la correcció de l'escriptura en català.

9.0.2 TexLive

TexLive[28] es una distribució molt poderosa de TeX/LaTeX que integra en sí mateix tot un entorn de treball amb fitxers de tipus .tex.

Fig. 94: Logotip del TexLive

En el cas d'aquest projecte, l'he utilitzat per les eines que implementa com per exemple el compilador i el exportador a PDF que després són utilitzades en el Visual Studio Code.

9.0.3 Preparació del entorn

Per tal de preparar l'entorn per un treball eficient amb arxius .tex he hagut de modificar el Visual Studio Code, comentat abans, perquè aquest integri el suport d'aquest sistema.
Per tant en primer lloc he hagut de descarregar-me la extensió Azure Repos[29] (Fig. 95), que me permet la connexió al meu repositori DevOps, on podré tenir un seguiment sobre el codi font del document.

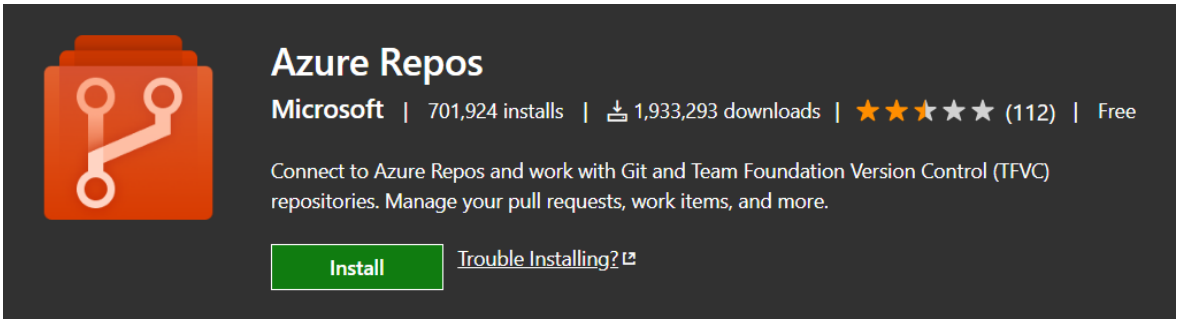


Fig. 95: La extensió Azure Repos

Una altre extensió molt important és la LaTeX Workshop[30] (Fig. 96), que implementa una GUI per sobre de la interfície de Visual Studio Code que facilita d'una manera molt significativa totes les operacions que s'han de fer amb aquest tipus de sistema.

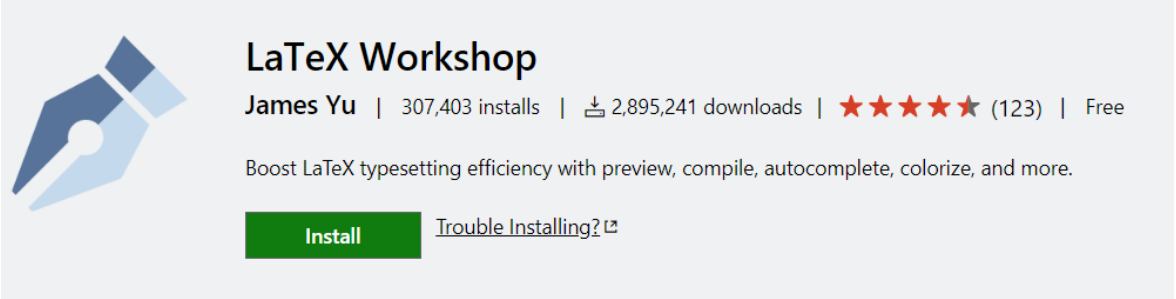


Fig. 96: La extensió LaTeX Workshop

Per últim l'ultima extensió a instal·lar és la de la correcció ortogràfica Spell Right[31] (Fig. 96)

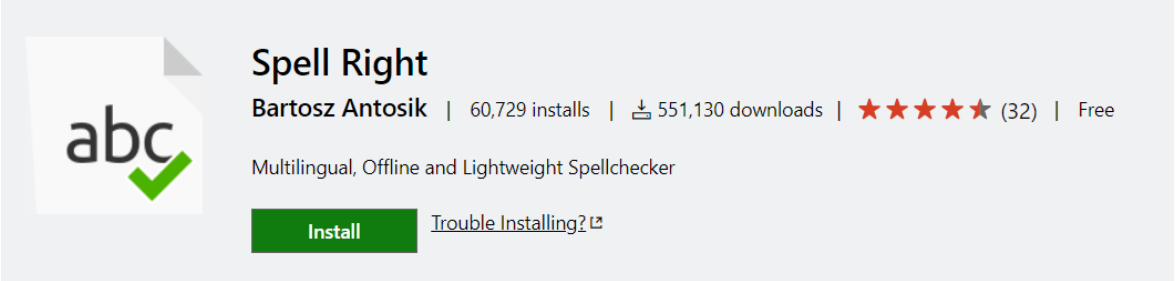


Fig. 97: La extensió Spell Right

Un cop finalitzada la preparació del Visual Studio Code, ha arribat el moment de la configuració del TexLive, perquè les seves eines estiguin disponibles des de la línia de comandes de Windows. Per fer-ho hem de modificar la variable PATH del sistema operatiu, afegint-hi els directoris on es troben el executables del TexLive.
Aquest pas es necessari perquè la extensió LaTeX Workshop pugui trobar totes les eines que necessita.

10 CONCLUSIONS

Pel que fan els propòsits inicials del projecte, puc afirmar que s'han completat tots d'ells amb la seva totalitat creant un sistema del qual em puc sentir orgullós, sense por de poder presentar-lo davant d'un públic. La manera de com està fet el projecte li permet una llarga vida, on de forma fàcil se li poden implementar funcionalitats noves, adaptant-lo a les noves necessitats i/o àrees del món de la indústria actual, fent possible l'aparició d'aquest sistema en una infinitat de sectors i dispositius monitoritzadors del món.

Considero que les idees inicials pel que fa la forma del projecte, per tant, la seva flexibilitat i escalabilitat han estat les correctes i són els punts clau que destaquen d'aquest, fent-lo especial i competitiu, amb un possible futur com un implementació real en el món industrial. El fet d'haver lo implementat amb eines multiplataforma li ha donat la possibilitat d'encaixar en un ampli ventall de gustos del possibles clients, sense haver de lligar-los a una plataforma predeterminada, com pot ser el as d'aplicació mòbil que pot ser utilitzada en totes les plataformes punteres d'avui en dia.

El fet d'utilitzar eines punteres també li dóna una avantatge sobre la competició en tots els aspectes, ja sigui el de manteniment del codi font, com el rendiment d'aquest, com l'adaptació a altres eines del mercat que probablement deixaran de suportar en algun moment donat, les solucions obsoletes creades per la competència.

Fent unes últimes paraules en aquest document m'agradaria destacar el fet de que la tecnologia d'avui en dia permet que una persona jove, sense molta experiència en els camps que tracta, que no pot dedicar tot el seu temps al desenvolupament del projecte, es capaç de crear i posar en funcionament un sistema tan complex que involucra els coneixements de moltes àrees del món informàtic. Es capaç de fer-ho en un temps raonable que es el temps de la implementació d'un Treball de Final de Màster, augmentant així els seus propis coneixements de forma autodidacte. Considero que es un fet molt important que pot servir com a exemple de les possibilitats que s'obren davant d'un programador actual que, si es capaç de fer el salt a utilitzar les tecnologies punteres, sense estancar-se a les solucions clàssiques, pot implementar coses sense cap mena de frontera, limitant-se només amb la seva imaginació.

11 REFERÈNCIES

[1] azure.microsoft.com. (2019). Azure DevOps. [online] Disponible a: https://azure.microsoft.com/es-es/services/devops/

[2] visualstudio.microsoft.com. (2019). Visual Studio Tools for Xamarin. [online] Disponible a: https://visualstudio.microsoft.com/xamarin/

[3] docs.microsoft.com. (2019). C# Guide. [online] Disponible a: https://docs.microsoft.com/en-us/dotnet/csharp/

[4] dotnet.microsoft.com. (2019). F#. [online] Disponible a: https://dotnet.microsoft.com/languages/fsharp/

[5] docs.microsoft.com. (2019). Visual C++ Documentation. [online] Disponible a: https://docs.microsoft.com/en-us/cpp/?view=vs-2019

[6] www.nuget.org. (2019). ModbusTcp. [online] Disponible a: https://www.nuget.org/packages/ModbusTcp/

[7] docs.microsoft.com. (2019). double (C# Reference). [online] Disponible a: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/double

[8] www.nuget.org. (2019). CommandLineUtils. [online] Disponible a: https://www.nuget.org/packages/Microsoft.Extensions.CommandLineUtils/

[9] www.nuget.org. (2019). AdvancedComparer. [online] Disponible a: https://www.nuget.org/packages/AdvancedComparer/

[10] docs.microsoft.com. (2019). Reflection (C#). [online] Disponible a: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection

[11] www.newtonsoft.com. (2019). JsonConvert Class. [online] Disponible a: https://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_JsonConvert.htm

[12] en.wikipedia.org. (2019). Web Service. [online] Disponible a: https://en.wikipedia.org/wiki/Web_service

[13] docs.microsoft.com. (2019). Introduction to LINQ Queries (C#). [online] Disponible a: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries

[14] en.wikipedia.org. (2019). JSON. [online] Disponible a: https://en.wikipedia.org/wiki/JSON

[15] visualstudio.microsoft.com. (2019). Visual Studio. [online] Disponible a: https://visualstudio.microsoft.com

[16] en.wikipedia.org. (2019). Modbus. [online] Disponible a: https://en.wikipedia.org/wiki/Modbus

[17] en.wikipedia.org. (2019). Common Language Infrastructure. [online] Disponible a: https://en.wikipedia.org/wiki/Common_Language_Infrastructure

[18] docs.microsoft.com. (2019). Migrations. [online] Disponible a: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/

[19] docs.microsoft.com. (2019). Create an ASP.NET Core web app in Azure. [online] Disponible a: https://docs.microsoft.com/en-gb/azure/app-service/app-service-web-get-started-dotnet#launch-the-publish-wizard

[20] docs.microsoft.com. (2019). Quickstart: Create a single database in Azure SQL Database using the Azure portal. [online] Disponible a: https://docs.microsoft.com/en-gb/azure/sql-database/sql-database-single-database-get-started

[21] ca.wikipedia.org. (2019). Model-Vista-Controlador. [online] Disponible a: https://ca.wikipedia.org/wiki/Model-Vista-Controlador

[22] ca.wikipedia.org. (2019). Model-vista-vista model. [online] Disponible a: https://ca.wikipedia.org/wiki/Model-vista-vista_model

[23] https://www.mvvmcross.com. (2019). MVVM CROSS. [online] Disponible a: https://www.mvvmcross.com

[24] en.wikipedia.org. (2019). Data transfer object. [online] Disponible a: https://en.wikipedia.org/wiki/Data_transfer_object

[25] www.nuget.org. (2019). Microcharts for Xamarin.Forms. [online] Disponible a: https://www.nuget.org/packages/Microcharts.Forms/

[26] www.latex-project.org. (2019). An introduction to LaTeX. [online] Disponible a: https://www.latex-project.org/about/

[27] visualstudio.com. (2019). Visual Studio Code. [online] Disponible a: https://code.visualstudio.com

[28] www.tug.org. (2019). TeX Live. [online] Disponible a: https://www.tug.org/texlive/

[29] marketplace.visualstudio.com. (2019). Azure Repos. [online] Disponible a: https://marketplace.visualstudio.com/items?itemName=ms-vsts.team

[30] marketplace.visualstudio.com. (2019). LaTeX Workshop. [online] Disponible a: https://marketplace.visualstudio.com/items?itemName=James-Yu.latex-workshop

[31] marketplace.visualstudio.com. (2019). Spell Right. [online] Disponible a: https://marketplace.visualstudio.com/items?itemName=ban.spellright