

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

CORSO DI LAUREA IN INFORMATICA



**Progettazione e sviluppo di un servizio di
Identity Access Managment basato su
blockchain**

Tesi di laurea triennale

Relatore

Prof. Gilberto Filè

Laureando

Simone Ballarin

Dedicato alla mia famiglia.

Sommario

Il presente documento riassume il lavoro svolto durante il periodo di stage della durata di 320 ore presso l'azienda iVoxIT S.r.l. di Padova.

Lo scopo principale del prodotto sviluppato è quello di integrare all'interno dell'applicativo Monokee, un sistema di creazione e verifica dell'identità basato su tecnologia blockchain compatibile con lo standard SAML. Nel corso del documento verranno anche esposte le basi teoriche del prodotto come la gestione delle identità e il Single Sign-On (SSO).

Inizialmente mi sono concentrato sullo studio di vari documenti forniti dall'azienda inerenti al progetto e a come quest'ultimo si doveva integrare con l'attuale sistema e una volta capiti gli aspetti fondamentali mi sono dedicato alla ricerca e all'apprendimento di vari strumenti tecnologici confacenti ad un corretto sviluppo. Dopo aver identificato le principali funzionalità richieste e compreso il funzionamento di come erano definiti i flussi di lavoro, ho iniziato la progettazione del servizio. Il risultato ottenuto, seppure all'altezza della aspettative dell'azienda, ha fatto emergere come un'approccio basato su blockchain risulti essere non adatto nella maggior parte dei contesti.

“Quello che conta è non dare fastidio agli altri: ma chi ci riesce?”

— Enzo Ferrari

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Gilberto Filè, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ho desiderio di ringraziare poi i miei amici per le tante avventure vissute assieme e per questi tre anni indimenticabili.

Inoltre vorrei ringraziare la mia ragazza Cristina per tutti i bellissimi momenti condivisi e per il supporto che mi ha dato. Grazie per avermi sempre motivato, dato conforto e, non da ultimo, per avermi sempre preparato da mangiare la sera.

Desidero infine ringraziare anche tutti i membri di Athesys per il supporto datomi. In particolare Sara, che è stata la mia tutor; Enrico, che mi ha sempre aiutato per qualsiasi problema e Valentina per il suo aiuto con Xamarin. Vividi ringraziamenti vanno anche ad Andrea, Silvia, Roberto e Simone, tutti ottimi giocatori di calcetto con i quali ho svolto partite epiche.

Padova, Agosto 2018

Simone Ballarin

Indice

1	Introduzione	1
1.1	L'azienda	1
1.2	L'idea	1
1.3	Organizzazione del testo	2
2	Processi e metodologie	3
2.1	Processo sviluppo prodotto	3
2.1.1	Metodologie di sviluppo Agile	3
2.1.2	Scrum	4
3	Descrizione dello stage	5
3.1	Introduzione al progetto	5
3.1.1	Descrizione del prodotto	5
3.2	Studio Tecnologico Identity Trust Fabric	5
3.2.1	Sintesi dello studio tecnologico	5
3.2.2	ITF – Identity Trust Fabric	6
3.2.3	Introduzione alla tecnologia blockchain	7
3.2.4	Permissionless e Permissioned blockchain	10
3.2.5	Conclusioni	15
3.3	Studio di fattibilità Identity Wallet	16
3.3.1	Sintesi dello studio di fattibilità	16
3.3.2	Descrizione componente Identity Wallet	16
3.3.3	Studio del dominio	17
3.3.4	Motivazioni	20
3.3.5	Conclusione Studio di fattibilità IW	20
3.4	Studio di fattibilità Service Provider	21
3.4.1	Sintesi dello studio di fattibilità	21
3.4.2	Descrizione Service Provider	21
3.4.3	Studio del dominio	22
3.4.4	Dominio tecnologico	22
3.4.5	Conclusioni scelta sviluppo	26
3.4.6	Motivazioni	26
3.4.7	Conclusioni	26
3.5	Obiettivi	26
3.6	Pianificazione	27
4	Analisi dei requisiti	29
4.1	Specifiche in Linguaggio Naturale	29

4.2	Specifiche in Linguaggio Strutturato	29
4.3	Specifiche in Linguaggio UML Use Case	30
4.4	Analisi dei requisiti IW	30
4.4.1	Casi d'uso	30
4.4.2	Tracciamento dei requisiti	48
4.5	Analisi dei requisiti SP	53
4.5.1	Casi d'uso	53
4.5.2	Diagramma delle attività	56
4.5.3	Tracciamento dei requisiti	57
4.6	Riepilogo requisiti	61
4.6.1	Riepilogo requisiti IW	61
4.6.2	Riepilogo requisiti SP	61
5	Progettazione e codifica	63
5.1	Componente Identity Wallet	63
5.1.1	Tecnologie e strumenti	63
5.1.2	Overview	64
5.1.3	Progettazione	64
5.1.4	Design Pattern utilizzati	69
5.2	Componente Service Provider	71
5.2.1	Tecnologie e strumenti	71
5.2.2	Overview	74
5.2.3	Progettazione	75
5.2.4	Design Pattern utilizzati	82
5.3	Implementazione	82
5.3.1	Procedura di login	82
5.3.2	Uso del database SQLite	84
5.3.3	Implementazione del databinding	86
5.3.4	Instaurazione della rete di messaggi	87
5.3.5	Gestione delle code e dei messaggi	88
5.3.6	Interazioni con la <i>blockchain</i>	89
5.3.7	Procedura di accesso a servizio	92
6	Verifica e validazione	95
6.1	Verifica	95
6.1.1	Attività di verifica statica	96
6.1.2	Realizzazione dei test	96
6.2	Validazione	96
6.2.1	Validazione requisiti componente IW	97
6.2.2	Validazione requisiti componente SP	98
7	Conclusioni	101
7.1	Conoscenze acquisite	101
7.2	Valutazione personale	102
A	Appendice A	105
A.1	Strumenti utili per lavorare su Ethereum	105
	Bibliografia	109

Elenco delle figure

1.1	Logo aziendale iVoIT	1
1.2	Diagramma Moduli	2
2.1	Diagramma flusso Scrum	4
3.1	Diagramma Moduli	6
3.2	Lista concatenato con <i>hash pointer</i>	9
3.3	Albero di Merkle	9
3.4	Diagramma flussi tra i vari componenti	23
4.1	Gerarchia utenti user case	30
4.2	Use Case - UC1: Azioni utente generico	31
4.3	Use Case - UC1.1 – Visualizza info applicazione	32
4.4	Use Case - UC2: Azioni utente non registrato	34
4.5	Use Case - UC2.1: Registrazione	35
4.6	Use Case - UC2.1: Accesso MonoKee	37
4.7	Use Case - UC3: Azioni utente autenticato	40
4.8	Use Case - UC3.4: Inserimento informazione personale	42
4.9	Use Case - UC3.5: Visualizza lista certificazioni	44
4.10	Use Case - UC3.5.1: Visualizza singola certificazione	45
4.11	Gerarchia utenti user case	53
4.12	Use Case - UC1: Azioni servizio convenzionato	54
4.13	Use Case - UC2: Azioni utente IW	55
4.14	Diagramma attività procedura di accesso	62
5.1	Architettura PCL	64
5.2	Architettura PCWL	65
5.3	Architettura IW	66
5.4	Diagramma BusinessLogic Layer IW	67
5.5	Diagramma DataAccess Layer IW	68
5.6	Diagramma UML VM Layer	69
5.7	Schema Mediator Topology	72
5.8	Schema Broker Topology	73
5.9	Flusso eventi SP	75
5.10	Diagramma delle classi del modulo SP	76
5.11	Rappresentazione UML di <i>Starter</i>	77
5.12	Rappresentazione UML di <i>RetriveInfo</i>	77
5.13	Rappresentazione UML di <i>PageGenerator</i>	79
5.14	Rappresentazione UML di <i>PIIDataHandler</i>	80

5.15	Rappresentazione UML di <i>SendAccessInfo</i>	81
5.16	Schermata di login di Salesforce	92
5.17	Schermata di scelta dominio	92

Elenco delle tabelle

3.1	Tabella comparivi framework sviluppo applicazioni mobili	19
3.2	Tabella comparivi linguaggio per sviluppo SP	24
3.3	Tabella comparivi client Ethereum	25
3.4	Tabella comparivi client Ethereum	25
4.1	Tabella del tracciamento dei requisiti funzionali	49
4.2	Tabella del tracciamento dei requisiti di vincolo	50
4.3	Tabella del tracciamento dei requisiti qualitativi	50
4.4	Tabella del tracciamento dei requisiti con le fonti	50
4.5	Tabella del tracciamento delle fonti con i requisiti	51
4.6	Tabella del tracciamento dei requisiti funzionali	58
4.7	Tabella del tracciamento dei requisiti di vincolo	58
4.8	Tabella del tracciamento dei requisiti qualitativi	59
4.9	Tabella del tracciamento dei requisiti con le fonti	59
4.10	Tabella del tracciamento dei fonte con requisiti	60
4.11	Riepilogo requisiti IW	61
4.12	Riepilogo requisiti SP	61
6.1	Tabella validazione IW	97
6.2	Tabella di validazione SP	99

Capitolo 1

Introduzione

1.1 L'azienda

L'attività di stage è stata svolta presso l'azienda iVoIT S.r.l. (logo in figura 1.1) con sede a Pavoda presso il centro direzionale La Cittadella. *iVoxIT S.r.l.* con *Athesys*



Figura 1.1: Logo aziendale iVoIT

S.r.l. e *Monokee S.r.l.* fa parte di un gruppo di aziende fondato nel 2010 dall'unione di professionisti dell'[Information Technology](#)^[g] (IT) con l'obiettivo di fornire consulenza ad alto livello tecnologico e progettuale. Tra le altre cose, Athesys S.r.l. fornisce supporto nell'istanziamento del processo di [Identity Access Management](#)^[g] (IAM), con particolare attenzione alla sicurezza nella conservazione e nell'esposizione dei dati sensibili gestiti. L'azienda opera in tutto il territorio nazionale, prevalentemente nel Nord Italia e vanta esperienze a livello europeo in paesi quali Olanda, Regno Unito e Svizzera. Grazie all'adozione delle [best practise](#)^[g] definite dalle linee guida [Information Technology Infrastructure Library](#)^[g] (ITIL) e alla certificazione ISO 9001 il gruppo è in grado di assicurare un'alta qualità professionale.

1.2 L'idea

Nell'ottica di estendere le funzionalità del prodotto *Monokee* di [Identity Access Management](#) basato su *cloud*, lo stage ha visto lo sviluppo di due moduli applicativi in ambito [blockchain](#)^[g]. Il primo modulo è un'applicazione mobile (**Wallet**) contenente

l'identità digitale dell'utente finale mentre il secondo modulo è un *layer* applicativo (**Service Provider**) per gestire gli accessi alle applicazioni di terze parti. In figura 3.1 un'immagine dei moduli da implementare e il loro posizionamento in un tipico scenario di accesso ai servizi. La tipologia di **blockchain** da integrare è stata individuata in una

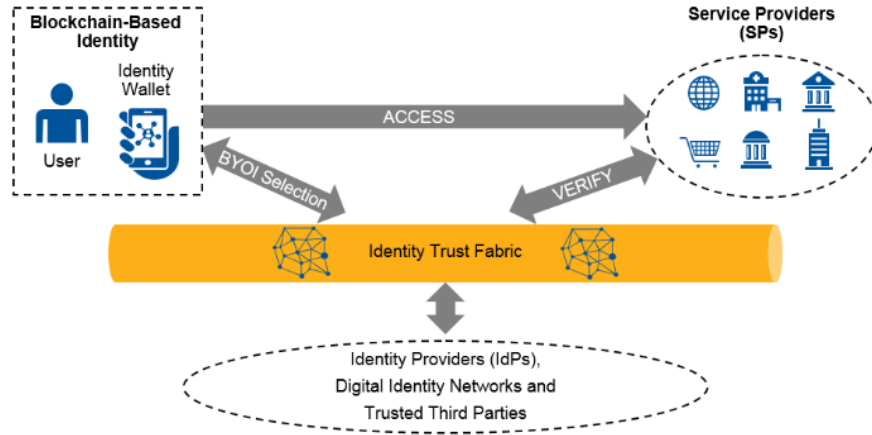


Figura 1.2: Diagramma Moduli

prima fase di analisi.

1.3 Organizzazione del testo

Il secondo capitolo fornisce una breve introduzione alle metodologie adottate durante lo svolgimento delle attività.

Il terzo capitolo presenta una serie di approfondimenti relativi al dominio applicativo e tecnologico in cui si colloca il progetto.

Il quarto capitolo espone lo studio dei requisiti svolto.

Il quinto capitolo descrive la progettazione dei vari componenti presenti nel progetto. Inoltre per alcuni di essi ne approfondisce alcuni aspetti relativi all'implementazione.

Il sesto capitolo approfondisce le varie attività svolte nel contesto della verifica e della validazione.

Nel settimo capitolo fornisce alcune considerazioni personale relative allo stage.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- * gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- * per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*^[§];
- * i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Processi e metodologie

2.1 Processo sviluppo prodotto

Durante ogni attività è stata seguita una metodologia di sviluppo [agile](#). L'azienda iVoxIT S.r.l. per la precisione attua in ogni suo progetto il metodo [Scrum](#). Le attività sono state descritte in task secondo la modalità [Scrum](#); ogni task veniva esposto e discusso in riunioni giornaliere con il tutor aziendale Dott. Sara Meneghetti. Inoltre erano previste riunioni settimanali con il responsabile del progetto Ing. Roberto Griggio.

2.1.1 Metodologie di sviluppo Agile

Le metodologie di sviluppo agile si basano su quattro principi cardine:

- * il software funzionante prima dei documenti;
- * il rapporto con il cliente;
- * i rapporti interni al team;
- * rispondere al cambiamento;

Dal momento che i processi di pianificazione necessari ad uno sviluppo a cascata sono molto costosi e che se questi non vengono rispettati tutta la pianificazione deve essere completamente rivista, queste metodologie si basano su modelli incrementali. Gli approcci agile tendono a progettare il minimo indispensabile in modo tale da essere sempre più reattivi e proattivi possibili agli inevitabili cambiamenti. Inoltre scrivere del *software* incrementale permette una progettazione iniziale molto snella, la quale con l'accrescere della conoscenza sul dominio può diventare sempre più raffinata. Tutto ciò rende meno costoso il *refactoring* del codice e anche l'inserimento di nuove funzionalità. Questi metodi sono adatti per piccoli team, molto coesi e uniti, e non dislocati in regioni diverse questo perché la comunicazione di persona è fondamentale. A tal proposito il team in cui ero inserito si componeva di tre membri. Un altro punto critico è che, essendo l'approccio alla comprensione dei requisiti e alla progettazione è poco concentrato all'inizio e molto diluito in tutto il progetto, è necessario un costante interesse da parte degli [stakeholders](#), cosa che in un nuovo progetto è ipotizzabile, mentre in un progetto in fase manutentiva no.

2.1.2 Scrum

Scrum è un metodo iterativo che divide il progetto in blocchi rapidi di lavoro chiamati *Sprint*, della durata massima di quattro settimane. Alla fine di ogni *Sprint* si ottiene un incremento del prodotto e durante ogni fase dello stage si è seguito questo modello. Scrum prevede una prima fase di analisi e progettazione di massima e una successiva suddivisione del lavoro in unità' creabili e implementabili in un unico sprint che poi vengono messi in un *backlog*. Prima di ogni sprint si sceglie una selezione di lavori in base alle priorità e si inseriscono nel backlog dello sprint. Gli sprint durano da 1 a 4 settimane e se il lavoro non viene portato a termine non viene prolungato lo sprint, ma rimesso nel *backlog* principale. Da ciò emerge come sia importante dare il giusto quantitativo di lavoro. È compito dello Scrum Master assicurarsi del rispetto dei tempi, infatti quest'ultimo effettua quotidianamente una breve riunione, della durata di circa quindici minuti, per valutare i progressi fatti (Daily Scrum). I rapporti tra i membri del progetto sono stati importanti e per questo Scrum prevede riunioni giornaliere per rimanere aggiornati sullo stato dei lavori di ogni componente. In Figura 2.1 è mostrato un tipico ciclo Scrum.

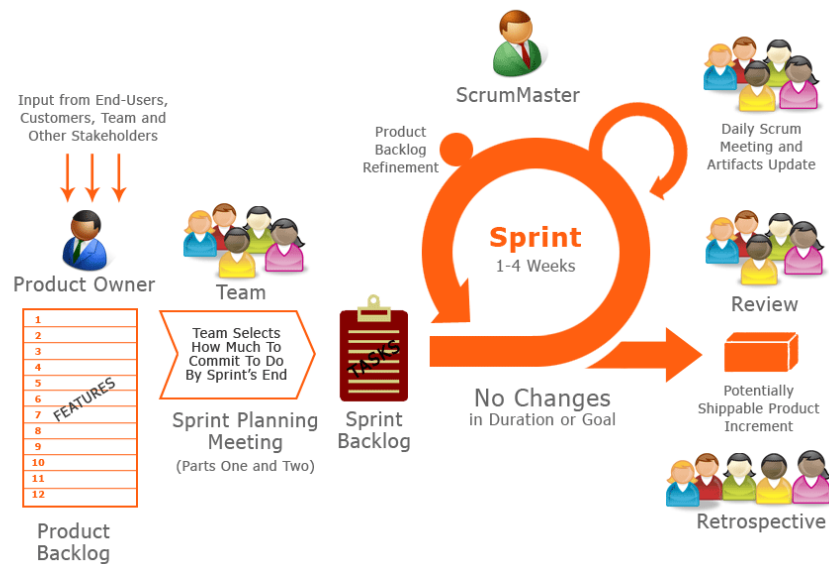


Figura 2.1: Diagramma flusso Scrum

Capitolo 3

Descrizione dello stage

3.1 Introduzione al progetto

3.1.1 Descrizione del prodotto

Il progetto ha come scopo la creazione di un'estensione del servizio *MonoKee* basato su [blockchain](#). L'estensione offre un sistema di [Identity Access Management](#) (IAM) composto da quattro principali fattori:

- * **Identity Wallet** (IW);
- * **Service Provider** (SP);
- * **Identity Trust Fabric** (ITF);
- * **Trusted Third Party** (TTP);

In sintesi l'estensione dovrà operare al fine di fornire la possibilità ad un utente di registrare e gestire la propria identità autonomamente tramite l'IW e mandare i propri dati (*personally identifiable information*, PII) all'ITF, il quale custodirà la sua identità e farà da garante per le asserzioni provenienti dai TTP. Inoltre il SP dovrà essere in grado con le informazioni provenienti da IW e ITF di verificare o meno l'accesso ai propri servizi. L'immagine in figura [3.1](#) dovrebbe chiarificare i vari componenti in gioco.

3.2 Studio Tecnologico Identity Trust Fabric

3.2.1 Sintesi dello studio tecnologico

Il capitolo procede descrivendo le caratteristiche del prodotto *MonoKee* e di come la tecnologia [blockchain](#) si possa collocare in tale contesto; vengono poi trattati i principali strumenti e librerie disponibili per sviluppare in *Ethereum*. L'analisi si conclude facendo emergere come un utilizzo di *Ethereum* sia possibile, ma non consigliato; le ragioni sono prettamente legate alla scalabilità del sistema. Per ragioni di facilità di sviluppo e di *time to market* si è ritenuto ad ogni modo di adottare la scelta di *Ethereum* come base del componente ITF.

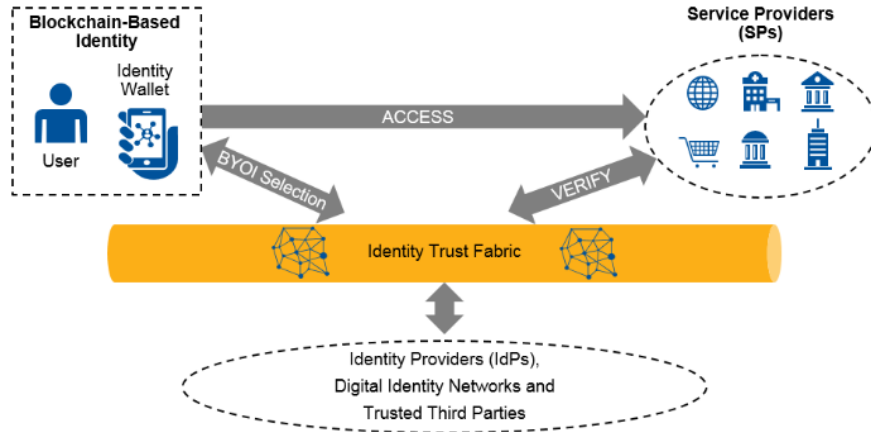


Figura 3.1: Diagramma Moduli

3.2.2 ITF – Identity Trust Fabric

Sulla base di un primo studio di fattibilità l'unico componente coinvolto nell'uso [blockchain](#) è l'*Identity Trust Fabric*. La sua principale funzione è quella di poter permettere ai vari Service Provider aderenti al servizio di poter verificare le informazioni rilasciate dai vari utenti tramite l'utilizzo del loro personale *Identity Wallet* (IW). Il componente mantiene al suo interno l'[hash](#)^[8] della chiave pubblica degli utenti (che rappresenta la loro identità) e le asserzioni fornite dai vari IW che possono essere potenzialmente certificate da una TTP (tramite una firma con la loro chiave privata). Le asserzioni devono poter essere modificate o eliminate in ogni momento, naturalmente ogni alterazione deve essere di volta in volta certificata nuovamente. Anche da parte del TTP ci dev'essere la possibilità di revocare la certificazione di un'asserzione. Secondo lo studio Gartner in nota¹ una buona implementazione di una ITF deve possedere le seguenti caratteristiche:

- * **Fiducia:** il contenuto presente nella ITF deve essere solo quello autorizzato e non ci devono poter essere manomissioni malevoli da parte degli utilizzatori della rete. Ogni componente deve potere aver fiducia nella veridicità dei dati.
- * **Garanzia:** le regole logiche della ITF non devono poter essere manomesse. Deve essere possibile applicare le varie *policy* aziendali in ambito di gestione dei rischi.
- * **Tracciabilità:** ogni informazione e cambio di stato relativo alle identità e alle asserzioni deve poter essere tracciato e verificabile sia in termini cronologici sia in termini di provenienza.
- * **Sicurezza:** intesa come CIA. L'ITF deve rispettare i vincoli di confidenzialità, inalterabilità e disponibilità delle informazioni dentro lei contenute.
- * **Scalabilità:** l'ITF deve fornire un elevato grado di scalabilità soprattutto in un'ottica in cui il prodotto potrebbe essere reso disponibile ad un uso Consumer.
- * **Efficienza:** il funzionamento dell'ITF deve richiedere la minima quantità di risorse possibili.

¹farah:The-Dawn-of-Decentralized-Identity.

3.2.3 Introduzione alla tecnologia **blockchain**

Al fine di rendere più consapevoli ai lettori le seguenti trattazioni si procede ad una esposizione ad alto livello dei principali concetti inerenti alla **blockchain**. *Blockchain* è comunemente definita come una base di dati distribuita composta da una serie di blocchi i quali possono contenere insiemi di dati o codice. In entrambi i casi questi sono temporalmente firmati. Ogni blocco contiene l'**hash** del blocco precedente. In questo modo si crea una sorta di collegamento fra i blocchi che forma una catena, la cosiddetta *blockchain*. In questa catena solo il blocco successore può collegarsi al predecessore.

Consideriamo, quindi, *blockchain* una qualsiasi piattaforma informatica **distribuita** che possiede almeno le seguenti tre caratteristiche tecniche al fine di fornire un registro a prova di manomissioni:

- * uso di funzioni crittografiche di **hash**;
- * uso di strutture dati con *hash pointer*;
- * uso di protocolli di consenso distribuito.

L'uso di funzioni di *hash* e di strutture dati basate su *hash pointers* conferisce a questa particolare tecnologia le sue peculiari caratteristiche di immutabilità dei dati e di conseguenza la rendono abile a fornire un registro a prova di manomissioni. Un registro con queste caratteristiche costituisce a sua volta un'affidabile struttura dati in grado di immagazzinare qualsiasi tipo di dato. Questo rende anche possibile l'aggiungere di nuovi dati in coda al registro.

Chiunque tenti di modificare un qualsiasi punto della *blockchain*, renderà in questo modo l'*hash pointers* al nodo successore errato. Se studiamo la testa della catena, anche se l'attaccante modifica i nodi successivi della catena in modo da renderli consistenti all'alterazione che ha cercato di intrinse, saremmo comunque in grado di identificare la manomissione verificando la correttezza del puntatore di testa.

Altra caratteristica fondamentale è l'uso di un algoritmo di consenso distribuito, questo rende possibile lo sviluppo di applicazioni distribuite; le cosiddette *dapp*. Una *dapp* immagazzina i propri dati e i propri registri delle operazioni in una *blockchain* in modo di evitare qualsiasi punto di vulnerabilità singolo (*single point of failure*). Un ottimo esempio di *dapp* può essere una qualsiasi cripto-moneta.

Funzione crittografica di hash

Una funzione di *hash* è una funzione matematica che converte un input di qualsiasi lunghezza in un output di lunghezza definita efficacemente. Una funzione di *hash* per essere considerata sicura deve possedere almeno le seguenti caratteristiche:

- * essere invertibile;
- * resistente alle collisioni;
- * essere *puzzle-friendly*.

Essere invertibile significa che non ci dev'essere nessun algoritmo tecnicamente fattibile in grado di identificare l'input dall'output. Essere resistente alle collisioni significa che non ci dev'essere nessuna tecnica in grado di individuare casi in cui a due input differenti corrisponda uno stesso output. Questo non implica che la funzione debba essere iniettiva. Essere *puzzle-friendly* significa che la tecnica più facile per invertire una funzione di *hash* è quella del *brute forcing*. Questa ultima caratteristica risulta fondamentale nel contesto dell'algoritmo di consenso.

Puzzle di ricerca

Il puzzle di ricerca consiste in un problema matematico che per essere risolto necessita di una ricerca della soluzione in un vasto dominio. Si dice che un puzzle di ricerca è *puzzle-friendly* quando non è presente nessuna strategia migliore rispetto al provare valori casuali. Questo tipo di problemi computazionali vengono adottati da alcune *blockchain* come base delle cosiddette [proof of work](#), particolari algoritmi di consenso nei quali i vari nodi della rete competono per la risoluzione del problema. L'attuazione di questo algoritmo rende possibile la decentralizzazione. Nelle *blockchain* basate su [proof of work](#) il nodo che riesce a risolvere il puzzle per primo viene riconosciuto come prossimo nodo che inserirà un blocco nella catena

Commitment

Un *commitment* è l'equivalente digitale di leggere un valore, sigillarlo in una busta e poi metterla in un luogo sicuro. Il valore prelevato rimane segreto agli altri nodi fino a che il nodo che ha effettuato il *commitment* non decida di rivelarlo. Le due funzioni che rendono questo possibile sono il **commit** e la **verify**.

Il *commit* è una funzione che prende un messaggio e un numero randomico segreto ([nonce](#)^[6]) come input e ritorna un *commitment*.

La *verify* è una funzione che prende *commitment*, [nonce](#) e messaggio come input. Restituisce *true* in caso la funzione di *commit* con input il messaggio e il nonce passati abbia un output corrispondente al commitment fornito. In tutti gli altri casi ritorna *false*.

Per usare questo schema abbiamo come primo passo quello di generare un numero randomico in qualità di *nonce*. Poi ci usando la funzione *commit* pubblichiamo un *commitment* usando il nonce appena generato. In un secondo momento, quando vogliamo rivelare il valore commitato in precedenza, pubblicheremo il *nonce* e il messaggio usati con la funzione di *commit*. Una terza parte potrà poi essere in grado di verificare il messaggio tramite l'uso della funzione *verify*. Questa particolare tecnica risulta particolarmente utile al fine di immagazzinare le identità o attributi su di essa all'interno della *blockchain*. Un *commitment* per essere tale deve nascondere le informazioni. Questa significa che non potranno più essere alterate.

Strutture dati con hash pointer

Un *hash pointer* è un puntatore verso il posto dove un dato è contenuto, unito al valore del dato sotto forma di [hash](#). A differenza di un normale puntatore che è solo in grado di fornirti un modo per ottenere l'informazione, un *hash pointer* permette anche di essere in grado di capire se il particolare dato è stato alterato dalla creazione del puntatore. Questo particolare puntatore può essere usato per la creazione di tipiche strutture dati concatenate, quali liste e alberi. La *blockchain* è una lista concatenata (*linked list*) costruita usando *hash pointer* in vece ai classici puntatori. In una *blockchain*, ogni blocco non solo ci dice dove si trova il blocco precedente, ma ci fornisce anche il valore dell'*hash* di questo permettendoci quindi di poter verificare qualsiasi alterazione. Le particolari caratteristiche degli *hash pointer* usati ci permette di verificare l'intera catena a partire dal blocco di testa del quale ci dobbiamo fidare. Un'immagine esplicativa di quanto appena detto è presente in figura [3.2](#)

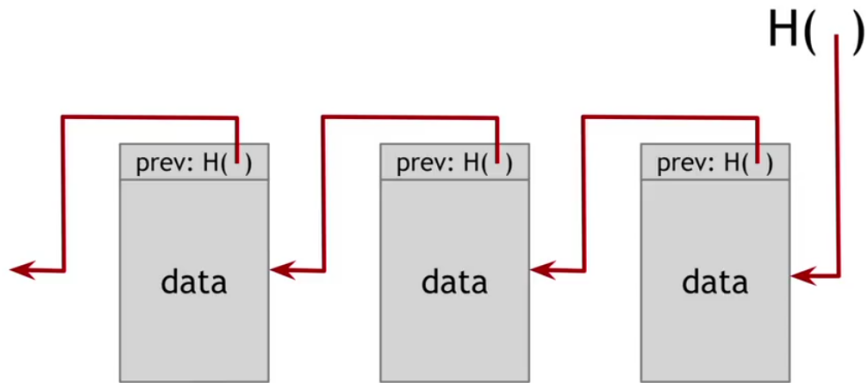


Figura 3.2: Lista concatenato con *hash pointer*

Albero di Merkle

Comunamente le principali *blockchain* fanno uso di una particolare struttura dati per immagazzinare la loro componente di dati, l'albero di Merkle. L'albero di Merkle è un albero binario che usa *hash pointer* in sostituzione dei classici puntatori. I record in quest'albero sono strutturati in coppie, e i loro *hash* sono conservati al livello superiore. Questa definizione si applica ad ogni livello del nodo fino al raggiungimento della radice. Le foglie rappresentano i dati. La figura ?? dovrebbe esporre con chiarezza il concetto.

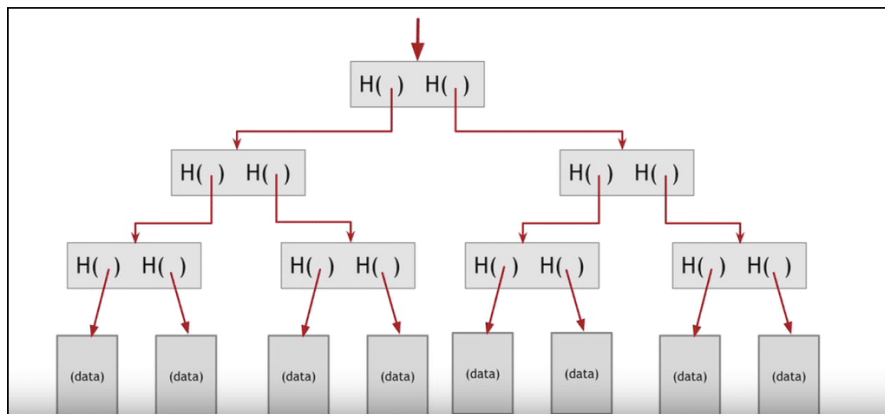


Figura 3.3: Albero di Merkle

In caso di alterazioni in un qualsiasi nodo dell'albero queste sarebbero rilevabili in quanto causerebbero delle incoerenze nei puntatori presenti nei nodi al livello superiore. L'uso di questa struttura nel contesto di una *blockchain* semplifica la verifica dell'attribuzione dei blocchi e permette di dover conservare solo la radice nel blocco di testa.

Algoritmo di consenso

Abbiamo visto come l'uso di funzioni di [hash](#) e di strutture dati particolari rendono la *blockchain* un'affidabile e verificabile base di dati centralizzata. Uno dei principali

fattori di successo della tecnologia è stata però la natura decentralizzata di essa. Questa è possibile grazie all'uso di molteplici nodi in sostituzione ad un *server* centralizzato. La presenza di un algoritmo di consenso rende possibile ai vari nodi partecipanti accordarsi su cosa debba essere scritto o meno nella catena. Assunto che i vari nodi (anche malevoli) ricevano un input valido l'algoritmo di consenso assicura le seguenti proprietà:

- * l'algoritmo termina quando tutti i nodi onesti sono in accordo sul valore;
- * l'algoritmo assicura che il valore è stato prodotto da un nodo onesto.

In questo caso, un nodo onesto viene scelto per inserire il proprio blocco in coda alla catena. In questo modo, l'algoritmo permette al sistema di generare fiducia in un contesto in cui nessun nodo si fida dell'altro.

Tipicamente in una *blockchain* avvengono le seguenti operazioni. A intervalli regolari, ogni nodo propone le proprie transazioni in sospeso per essere inserite come prossimo blocco. Successivamente si attua l'algoritmo di consenso in cui ogni nodo propone come input il blocco che intendono inserire. Alcuni nodi potrebbero essere malevoli ed inserire transazioni non valide nel proprio blocco di input, ma possiamo assumere che gli altri nodi siano onesti. Se l'algoritmo di consenso ha successo viene selezionato il blocco da inserire in coda. Questa attività viene detta *mining*. Nessuno decide quale sarà il prossimo nodo ad essere inserito nella *blockchain*.

C'è comunque da ricordare che la presente è una trattazione ad alto livello, e quindi non verranno presentati particolari algoritmi di consenso. Mi limito quindi a presentare le principali caratteristiche che questo deve avere:

- * equità;
- * velocità;
- * dimostrabilità;
- * resistenza al problema dei generali bizantini;
- * efficiente;
- * resistente ad attacchi *Dos* e *DDos*.

3.2.4 Permissionless e Permissioned blockchain

Al fine di poter valutare la fattibilità dell'utilizzo di *Ethereum* quale [blockchain](#) sottostante all'ITF è necessario avere in mente le due principali categorie di *blockchain*: **permissionless** e **permissioned**.

Permissioned [blockchain](#)

Una permissioned [blockchain](#) pone dei vincoli sulla partecipazione alla rete. Solo i nodi autorizzati possono partecipare all'algoritmo di consenso dei blocchi. Le autorizzazioni possono essere date singolarmente quindi i vari nodi possono avere o meno le seguenti possibilità:

- * lettura dei blocchi;
- * scrittura dei blocchi;
- * esecuzione di codice (se prevista dalla *blockchain*);
- * verifica dei nodi.

Permissionless blockchaing

Una permissionless [blockchain](#) è una rete in cui qualsiasi nodo può partecipare al processo di verifica dei blocchi. Ogni nodo ha tutte le precedenti quattro proprietà.

Ethereum *Ethereum* è una piattaforma decentralizzata pubblica ed *open-source* basata sulla creazione di [SmartContract](#). Permette la creazione di applicazioni che operano su [blockchain](#) in modo che non ci sia alcuna possibilità di *downtime*, censura, frodi o interferenze da terze parti. Rappresenta una dei principali esempi di rete *permissionless*. La piattaforma è stata rilasciata nel corso del 2014 ed è mantenuta dalla *Ethereum Foundation*, e questo fa di *Ethereum* una delle più longeve [blockchain](#) disponibili. Ciò comporta la presenza di una documentazione abbastanza nutrita rispetto ai competitor e di un buon numero di strumenti già disponibili. L'elevata popolarità della tecnologia e alcune sue caratteristiche non presenti nei competitor, ha fatto sì che l'attuazione di tale algoritmo è la principale causa dell'intrinseca lentezza di questa tecnologia. Esso infatti deve prevedere un *puzzle* di complessità adeguata a rendere vani eventuali tentativi di manomissioni. In caso un utente della *blockchain* non fosse intenzionato a che una notevole quantità di sviluppatori abbiano deciso di utilizzarla. Il sito ² mantiene una vetrina di oltre milleseicento esempi. Sono presenti tutte le più significative applicazioni ora in produzione; si fa notare che molte di esse sono state le fonti dei più diffusi pattern *Ethereum*.

Ethereum al fine del raggiungimento del consenso necessita di una dimostrazione di interesse verso la [blockchain](#), in gergo *Proof of Interest*. Questa nel nostro specifico caso si concretizza in una *Proof of Work*, cioè nella risoluzione di un *puzzle di ricerca*. La risoluzione di tale problema (detta *mining*) è la principale causa dell'intrinseca lentezza di questa tecnologia. Esso infatti deve prevedere un *puzzle di ricerca* di complessità adeguata a rendere vani eventuali tentativi di manomissioni³. In caso un utente della *blockchain* non fosse intenzionato a eseguire questo lavoro ha la possibilità di pagare un nodo della rete per farlo al posto suo⁴. In base alla quantità pagata si può richiedere una potenza di calcolo minore o maggiore e di conseguenza velocità di esecuzione differenti.

Il costo per *minare* una transazione varia in base alla mole di dati da immettere nella rete e alla complessità del codice che si vuole eseguire. Durante le attività di codifica si è notato come operazioni di complessità lineari potessero portare a costi estremamente elevati. La velocità del *mining* viene inpostata tramite un valore detto *gas*. All'aumentare di questo valore corrisponde una potenza di calcolo maggiore, quindi una risoluzione del *puzzle di ricerca* più o meno veloce. Per calcolare il prezzo finale di una transazione è necessario moltiplicare il costo base con il valore del *gas*.

Programmare SmartContract *Solidity* è il principale linguaggio di programmazione usato per scrivere [SmartContract](#). Nonostante sia presente un'implementazione basata su *Go*, questa è ancora acerba e non largamente utilizzata e per questo motivo tale implementazione non verrà trattata nel documento. *Solidity* è un linguaggio di programmazione ad oggetti ad alto livello. Il suo sviluppo è stato fortemente influenzato da linguaggi quali *C++*, *Python* e *Javascript*. Gli *SmartContract* così

²site:state-dapps

³All'aumentare degli interessi presenti nella rete aumenta anche la complessità del problema.

⁴I pagamenti vengono effettuati tramite un sistema di valuta digitale. La moneta utilizzata è chiamata *Ether*.

scritti vengono poi trasformati in bytecode e quest'ultimo viene eseguito dall'*Ethereum Virtual Machina* (EVM). Il linguaggio seppur non completamente maturo offre la maggior parte delle caratteristiche tipiche di un linguaggio ad oggetti. Infatti *Solidity* è fortemente tipato, supporta l'ereditarietà, librerie esterne e tipi definiti dall'utente. A sottolineare la bontà del linguaggio si evidenzia come in *Solidity* sia presente il concetto di interfaccia, caratteristica non presente in linguaggi ben più longevi. Queste caratteristiche rappresentano un notevole vantaggio per *Ethereum* rispetto ai diretti competitor, i quali spesso utilizzano linguaggi acerbi e/o a basso livello. Si è ritenuto che un linguaggio con le caratteristiche precedentemente descritte sia fondamentale per la buona riuscita del progetto, soprattutto in un'ottica di manutenibilità e estendibilità.

Breve nota sull'applicabilità dei pattern Nonostante il linguaggio permetta l'applicabilità dei più diffusi pattern si vuole far notare come nel contesto di una [blockchain](#) Permissionless questi risultino spesso controproducenti. Durante la progettazione e l'applicazione dei pattern vanno sempre ricordati i seguenti punti:

- * l'esecuzione di un metodo che modifica la *blockchain* si paga in base al lavoro che viene effettivamente svolto, in quanto è necessaria l'attuazione dell'algoritmo di consenso;
- * complessità lineari portano a costi di transazione difficilmente accettabili;
- * plugin come *Metamask* calcolano il massimo costo possibile di una transazione, in caso il credito non sia sufficiente la transazione fallisce. Ne consegue che un ciclo *for* su una lista di un elemento viene stimato presupponendo che la lista sia completamente piena;
- * la velocità di esecuzione varia in base alla somma pagata per questa⁵, anche con somme estremamente alte o su reti locali i tempi potrebbero essere considerati non giustificabili per la maggior parte degli utenti;
- * ogni oggetto e campo dato si paga in base al loro spazio occupato;
- * il costo della moneta e quindi delle transazioni è fortemente variabile. Approcci che, oggi risultano economici, possono diventare economicamente insostenibili a distanza di pochi giorni.

A seguito dei precedenti punti dovrebbe risultare più evidente come pattern che prevedono alta complessità temporale e spaziale siano inaccettabili su una rete *Ethereum*. Ad esempio i pattern *Command* e *Decorator* risultano difficilmente giustificabili. Sono invece presenti pattern pensati appositamente per *Ethereum*, questi sono presenti nella documentazione ufficiale *Solidity*⁶. Particolarmente utili al contesto del progetto in esame ritengo possano essere utili i seguenti pattern:

- * Owner Pattern;
- * Vote Pattern
- * WhiteList Pattern.

⁵Per i motivi descritti in [3.2.4](#) riguardo al commissionamento ad altri nodi della rete delle attività di *mining*.

⁶[site:solidity-documentation](#).

Complessità e pratiche non convenzionali I punti precedentemente stilati nel paragrafo sull'applicabilità dei pattern portano anche delle notevoli differenze in termini di pratiche di stile di programmazione. Tra queste riporto:

- * l'uso di liste e *array* è fortemente sconsigliato, vanno preferite strutture dati con accesso costante. *Solidity* fornisce il tipo *mapping*;
- * la creazione di oggetti (in termini *Solidity* contratti) ha un costo notevole. Una buona pratica è quella di utilizzare ADT (*Abstract Data Type*) differenti, come le strutture;
- * cicli *for* che portano complessità lineare dovrebbero essere evitati, elaborazioni di questo tipo dovrebbero essere affidate a *server* esterni o a livello *client-side*;
- * l'utilizzo dei puntatori (in *Solidity* *address*) nasconde completamente il tipo dell'oggetto puntato rendendo vano il controllo dei tipi. Andrebbe evitato il più possibile.

Si fa notare come in particolare l'ultimo punto degeneri completamente il concetto di programmazione ad alto livello.

Strumenti Vista la relativa maturità della tecnologia, esistono diversi strumenti utili:

- * **Truffle**;
- * **Ganache**;
- * **Mist**;
- * **Parity**;
- * **Metamask**;
- * **Status**;
- * **Microsoft Azure**.

Per una descrizione più specifica consultare [A.1](#).

Valutazione applicabilità soluzione Ethereum Al fine di poter valutare correttamente da ogni punto di vista l'applicabilità di una soluzione basata su *Ethereum* quale base della componente ITF, si procede ad analizzare in maniera analitica le sei caratteristiche presentate nel capitolo 'ITF – Identity Trust Fabric'.

* **Fiducia**

Questa caratteristica è ottenuta da *Ethereum* da una combinazione di diversi fattori quali:

- utilizzo di incentivi economici, il pagamento per effettuare operazioni;
- utilizzo di prove di interesse (*Proof of Interest*).

Le prove di interesse possono essere di due tipi:

- *Proof of Stake*, l'esibizione di un interesse;

- *Proof of Work*, l'uso di potenza di calcolo per risolvere un problema matematico. Queste metodologie fanno in modo che solo chi realmente interessato possa influenzare l'algoritmo di consenso dei blocchi. Questo rende minore la possibilità di un "51 percent attack"⁷. C'è comunque da ricordare che un attacco di questo tipo è praticamente impossibile.

Per queste ragioni si ritiene una rete *Ethereum* sia completamente soddisfacente per quanto riguarda l'aspetto fiducia al pari di una rete di tipo *permissioned*.

* Garanzia

Lo studio⁸ evidenzia come questo rappresenti un punto critico. Infatti riporta che il raggiungimento di questo obiettivo è fortemente condizionato dall'efficacia dell'algoritmo di consenso e dai nodi presenti nella rete. Lo studio prosegue facendo notare che la presenza di nodi malevoli, oltre che mettere a rischio l'algoritmo di consenso, può compromettere anche il corretto funzionamento dell'ITF. Trattandosi infatti di una *blockchain* pubblica ogni nodo è in grado di visionare il contenuto di ogni singolo contratto, inclusi i dati e i metodi presenti. Per quanto riguarda i dati questo potrebbe non essere un problema in quanto si può conservare una versione codificata del dato. Per quanto riguarda i metodi invece questo non è possibile, ed anzi, potrebbe rendere in grado ad un attaccante di trovare eventuali bachi e criticità dell'ITF. Il servizio *Azure* potrebbe permettere di creare reti private.

* Tracciabilità

Lo studio Gartner⁹ evidenzia come in una rete *permissionless* la tracciabilità temporale non sia possibile, questo perché in una rete distribuita ogni nodo può avere un concetto di tempo proprio. Questo però non risulta possibile in nessun approccio risolutivo all'ITF basato su *blockchain*, infatti le reti *permissioned* applicano *timestamp* a livello di blocco e non di transazione. Anche ammettendo che ci sia un concetto di tempo comune tra i nodi, le transizioni rimarrebbero temporalmente non tracciabili. La cosa potrebbe permettere ad un blocco di alterare l'ordine delle transazioni. Tale complicazione in una rete *permissioned* può essere superata creando blocchi immutabili e ogni qual volta si voglia fare una modifica, si dovrà creare un nuovo blocco. In questo modo ci sarà solo una transazione di creazione blocco il cui *timestamp* coinciderà con il *timestamp* del blocco. L'approccio in *Ethereum* rimane in ogni caso impraticabile. Attualmente non sono note ulteriori tecniche per la tracciabilità temporale in *Ethereum*, motivo per cui l'attribuzione di un riferimento temporale dovrà essere effettuato lato client, con i conseguenti limiti di sicurezza.

* Sicurezza

La confidenzialità dei dati, anche se non presente nativamente in *Ethereum*, è facilmente ottenibile immagazzinando nei contratti solo un *hash* dei dati. L'integrità dei dati invece è garantita dalla prova di lavoro che utilizza la *blockchain* come già ribadito nella sezione Fiducia. La disponibilità invece è garantita dalle caratteristiche di distribuzione di ogni *blockchain*. Un ulteriore punto di considerazione da fare è che chiunque ha la possibilità di vedere il contenuto di ogni [SmartContract](#) incluso il codice dei metodi. Questo come già detto può

⁷site:51-attack.

⁸farah:The-Dawn-of-Decentralized-Identity.

⁹farah:The-Dawn-of-Decentralized-Identity.

comportare la possibilità da parte di un attaccante di individuare eventuali errori logici. Ogni contratto dovrà comunicare con gli altri attraverso chiamate a metodi pubblici, in quanto non c'è in *Ethereum* nessun concetto di visibilità dei metodi di tipo *protected* o *package*. Questo rende possibile da parte di qualsiasi utente della rete di utilizzare questi metodi in maniera malevole. Questo tipo di problematica è facilmente superabile applicando i dovuti pattern *Solidity* quali *WhiteList Pattern* e *Owner Pattern*. L'applicazione dei pattern però comporterebbe un notevole aumento in termini di complessità e costo soprattutto in presenza di logiche di accesso variegate e dinamiche. Inoltre, in caso di liste di utenti autorizzati, queste potrebbero risultare onerose in termini di costo.

* **Scalabilità**

Ethereum per poter applicare l'algoritmo del consenso, fa utilizzo di una prova di lavoro che deve essere fatta in occasione di ogni transazione. La prova consiste nella risoluzione di un problema crittografico la cui difficoltà è dinamica in base a diversi fattori della *blockchain*, quali valore dell'*Ether*, numero di utenti, numero di transazioni, etc. Oltretutto si nota come anche in lettura ci sia una lentezza che difficilmente potrebbe essere ritenuta accettabile da un utente medio. Per avere prova di questo fatto si può prendere in esame una qualsiasi *Dapp* presente al seguente link¹⁰. La questione pone anche limiti, come già citato, in termini di costo.

3.2.5 Conclusioni

Da quanto è emerso l'utilizzo della tecnologia *Ethereum* quale base dell'ITF, pone una serie di vantaggi e svantaggi. Di seguito si propone una sintetica trattazione dei punti fondamentali, per maggiori dettagli si consiglia la lettura dell'intero documento. I vantaggi sono:

- * *Ethereum* offre un linguaggio ad alto livello e ad oggetti a differenza di altri competitor;
- * *Ethereum* offre una notevole maturità e anche un'ampia platea di strumenti, molti dei quali estremamente maturi e largamente utilizzati.

Gli svantaggi sono:

- * *Ethereum* è una rete pubblica, non è possibile fare nessuna restrizione di privilegi sui nodi partecipanti alla rete. Ciò potrebbe rappresentare un problema di sicurezza;
- * la comunicazione verso dispositivi mobili non è verificabile da quest'ultimi, in quanto dovrebbe avvenire tramite comunicazione *REST*;
- * sono presenti forti limitazioni in termini di costo e velocità, il sistema risulterebbe lento ed estremamente costoso. Ciò comporta notevoli difficoltà sulla scalabilità del servizio.

Si ritiene che un approccio basato su *Ethereum* sull'ITF sia possibile, le eventuali criticità di sicurezza e fiducia verso i dispositivi mobili sono superabili con una buona progettazione. L'unico fattore veramente critico rimane la scalabilità del sistema, fatto che, a mio parere, è sufficiente per ritenere *Ethereum* non adatto all'utilizzo, soprattutto

¹⁰ [site:state-dapps](https://github.com/state-dapps).

in un'ottica commerciale. Quindi se pure possibile, non si consiglia l'utilizzo di *Ethereum*. Per ragioni di facilità di sviluppo e di *time to market* si è ritenuto comunque di adottare la scelta di *Ethereum* come base del componente ITF.

3.3 Studio di fattibilità Identity Wallet

3.3.1 Sintesi dello studio di fattibilità

Lo studio inizia descrivendo come l'IW si cali in questo contesto. Si prosegue prendendo in considerazione le alternative di sviluppo mobile e desktop. A seguito di un'analisi dei possibili utenti emerge una netta preferenza per lo sviluppo mobile. Vengono poi trattati i principali strumenti e librerie disponibili per lo sviluppo ed è consigliabile uno sviluppo multi piattaforma, con target *Android* e *iOS*. In conclusione emerge una preferenza per il *framework Xamarin*.

3.3.2 Descrizione componente Identity Wallet

Il progetto ha come scopo la creazione di un *Identity Wallet* (IW). L'applicativo si colloca nel contesto di un'estensione del servizio *Monokee* basato su [blockchain](#). L'estensione offre un sistema di [Identity Access Management](#) composto da quattro principali fattori:

- * Identity Wallet (IW)
- * Service Provider (SP)
- * Identity Trust Fabric (ITF)
- * Trusted Third Party (TTP)

In sintesi, l'estensione dovrà operare al fine di fornire la possibilità ad un utente di registrare e gestire la propria identità automaticamente tramite l'IW, mandare i propri dati (IPP) all'ITF la quale custodirà la sua identità e farà da garante per le asserzioni provenienti dai TTP. Inoltre il SP dovrà essere in grado, con le informazioni provenienti da IW e ITF, di garantire o meno l'accesso ai propri servizi. Il software IW, più dettagliatamente, dovrà assolvere i seguenti compiti: nell'ambito della registrazione di un utente il *Wallet* deve:

- * generare e immagazzinare in locale una chiave pubblica;
- * generare e immagazzinare in locale una chiave privata;
- * creare l'*hash* della chiave pubblica e inviarla all'ITF;
- * incrementare le informazioni (PII) relative alle identità che il *Wallet* gestisce.

Nell'ambito della presentazione dei dati ad un *Service Provider* deve:

- * invio della chiave pubblica al *service provider*;
- * invio di un puntatore all'*hash* della chiave pubblica interna al ITF;
- * invio di altre informazioni utili presenti nel ITF;
- * gestire ulteriori *layer di sicurezza*, quali impronta digitale, *QR code*, autenticazione multi fattore)

nell'ambito della richiesta di accesso ad un servizio deve:

- * inviare una richiesta di accesso ad un servizio con i dati relativi all'identità al *Service Provider*;
- * attendere la risposta del *Service Provider*.

3.3.3 Studio del dominio

Dominio Applicativo

L'applicativo IW dovrà essere usato in un contesto prevalentemente lavorativo. Non si escludono però ulteriori applicazioni future in ambito *Consumer*. In ogni caso è indirizzato a utenti senza specifiche conoscenze informatiche e con minimo *training* tecnologico. Il software quindi dovrà essere il più accessibile e semplice possibile e per tali ragioni si pensa ad un suo utilizzo prevalentemente in ambito mobile, anche se non si esclude a priori la possibilità di una versione Desktop. L'applicativo mobile deve essere disponibile per la più ampia gamma di utenti possibili.

Dominio Tecnologico

Un eventuale applicativo Desktop ha un'elevata probabilità di non rientrare nei tempi dello stage, motivo per la quale si è deciso di non tenerlo in considerazione. L'applicativo quindi dovrà essere fruibile tramite un'applicazione mobile multiplatforma sviluppabile entro i limiti temporali della durata dello stage. Per queste ragioni lo studio del dominio tecnologico si incentrerà principalmente su tecnologie multi platforma mobili. Verrà comunque tenuto in considerazione anche lo sviluppo nativo.

Studio diffusione sistemi operativi mobili Procedo di seguito ad un'analisi sulla diffusione dei vari sistemi operativi mobili. Le informazioni di seguito riportate provengono da Kantar¹¹ società di analisi inglese e fanno riferimento al trimestre che va da novembre 2016 a gennaio 2017. I dati Kantar fanno emergere come *Android*, *iOS* e *Windows Phone* rappresentano, in questa sequenza ed in ogni mercato, i sistemi più diffusi. I restanti sistemi non raggiungono cifre significative. Ponendo maggiore attenzione ai primi tre sistemi si nota come *Android* nell'area EU5 rappresenti i tre quarti del mercato. In Giappone, Stati Uniti, Australia e Gran Bretagna invece la situazione risulta più bilanciata con una sostanziale parità. *Windows Phone* in ogni mercato si pone in terza posizione con percentuali che non superano mai l'otto per cento. Individuando nell'area EU5 il principale mercato per *MonoKee* si ritiene che il prodotto IW debba essere sviluppato per i sistemi *Android* e *iOS*, dando la precedenza al primo. Non si ritiene necessario lo sviluppo di un'applicazione *Windows Phone* in quanto difficilmente attuabile nei tempi dello stage.

Tecnologie per lo sviluppo Segue un approfondimento relativo alle potenziali tecnologie con cui sviluppare l'IW. Data la necessità di sviluppare sia per *Android*, che per *iOS* l'analisi si concentrerà principalmente su *framework* multi platforma senza comunque ignorare la possibilità di sviluppi nativi.

¹¹site:kantar-study.

Sviluppo multiplatforma Le applicazioni multiplatforma si dividono sostanzialmente in due categorie: le applicazioni **ibride** e le applicazioni **multiplatforma native**. Le prime consistono in applicazioni scritte in un linguaggio *cross-platform* (in genere *Javascript* e *HTML5*) come le *web app*, che però possono essere incapsulate tramite **Web View** nel linguaggio nativo di una certa piattaforma. Hanno un discreto accesso al sistema operativo anche se non sono state scritte interamente nel linguaggio specifico della piattaforma. Le seconde consistono in applicazioni scritte utilizzando particolari *framework* che permettono, da del codice comune, di generare differenti versioni in base al sistema operativo di destinazione. Le applicazioni così generate sono totalmente paragonabili ad applicazioni native sviluppate nel linguaggio specifico della piattaforma. Tra le principali alternative multi platforma si ritengono particolarmente interessanti le seguenti:

- * React Native;
- * Cordova;
- * Xamarin;

Segue un'analisi descrittiva dei vari *framework*.

React Native: è un *framework* di sviluppo mobile derivato da *React*. Il progetto è sviluppato e mantenuto da *Facebook*. *React Native* si focalizza nello sviluppo di UI tramite componenti scritti in *JavaScript*, con un approccio funzionale e flusso di dati unidirezionale. A differenza di *React*, *React Native* non manipola il DOM del browser, ma una struttura diversa, ma non solo; i componenti non vengono scritti a partire da elementi *HTML* o simili (i.e. *Bootstrap* o *Grommet*), bensì a partire da un set di componenti base presenti nella libreria. La libreria permette di sviluppare applicazioni per *iOS* e *Android*.

Cordova: è un framework open-source per lo sviluppo di applicazioni mobili che propone un approccio ibrido e non nativo. Permette di usare tecnologie web ampiamente utilizzate, quali *HTML5*, *CSS3*, *Javascript*, per la codifica. Il software così prodotto verrà eseguito in appositi wrapper diversi per ogni piattaforma, quindi in maniera non nativa. Il framework è sviluppato da *Apache* ed ormai ha raggiunto un elevato grado di maturità. Rappresenta uno dei primi framework per lo sviluppo multi piattaforma.

Xamarin: è un *framework* per lo sviluppo di applicazioni native e multi piattaforma con *C Sharp*. Il *framework* si basa sul progetto *open source Mono* e offre pieno supporto non solo alle piattaforme *Android* e *iOS* ma anche a *Windows Phone*. La possibilità di sviluppare anche per *Windows Phone* potrebbe risultare un punto a favore rispetto agli altri *framework*. *Xamarin* si compone di tre componenti principali: *Xamarin.Android*, *Xamarin.iOS*, *Xamarin.Forms*. L'ultimo componente si pone come strumento completamente neutro rispetto alla piattaforma. Grazie a queste componenti è possibile gestire in *C Sharp* tutte le caratteristiche di *Android*, *iOS* e *Windows Phone*.

La tabella 3.1 riassume quanto appena detto. Data l'impossibilità degli approcci ibridi, quali *Cordova*, di sfruttare a pieno le caratteristiche tipiche delle diverse piattaforme mobili, si ritiene di scartare questo tipo di soluzioni. Inoltre si evidenziano difetti come una mancata o incompleta integrazione dell'aspetto grafico con la specifica piattaforma e una maggiore lentezza nell'esecuzione e accesso alle risorse locali. Di conseguenza sarebbe più opportuno l'utilizzo di un *framework* che permetta di scrivere

Tabella 3.1: Tabella comparivi framework sviluppo applicazioni mobili

Framework	Approccio	Piattaforme supportate	Linguaggio
React Native	Nativo	iOS, Android	Javascript
Cordova	Ibrido	iOS, Android	HTML5, CSS3, Javascript
Xamarin	Nativo	iOS, Android, WP	C Sharp

applicazioni in maniera nativa. Richiudendo la visione ai soli approcci nativi, *Xamarin*, rispetto a *React Native*, lascia aperte le porte ad una eventuale applicativo *Windows Phone*. Oltretutto *C Sharp* utilizza un linguaggio che rispetto a *Javascript* fornisce una tipizzazione forte e caratteristiche più orientate agli oggetti. Si consiglia quindi l'utilizzo di *Xamarin* o *React Native* con la preferenza per il primo.

Sviluppo Nativo Un'applicazione nativa è un'applicazione mobile sviluppata interamente nel linguaggio del dispositivo sul quale vengono eseguite, ovvero *Java* per *Android* e *Swift* o *Object-C* per *iOS*. Il loro utilizzo presenta diversi vantaggi rispetto allo sviluppo multi piattaforma:

- * interazione con tutte le caratteristiche del dispositivo consentendo l'utilizzo al 100%;
- * maggiore velocità offrendo quindi una *User Experience* di più alto livello;
- * facilità di integrazione di terze parti tramite utilizzo di SDK ufficiali.

Il primo punto non dovrebbe rappresentare un plus in quanto l'IW debba usufruire di feature particolari dei dispositivi. È da notare che uno sviluppo nativo richiede il doppio delle risorse necessarie poichè prevede lo sviluppo di due applicazioni completamente diverse (*Android* e *iOS*), con *framework* e quindi con architetture potenzialmente diverse. Riassumendo, dato che:

- * l'applicativo che si dovrà sviluppare non prevede particolari requisiti prestazionali;
- * l'alto costo in termini orari di sviluppare soluzioni differenti ha una forte probabilità di non rientrare nei tempi previsti dall'attività di stage;

si ritiene non conveniente lo sviluppo parallelo di più applicazioni native.

Conclusioni sulla scelta del framework A seguito di quanto detto nelle sezioni "Sviluppo multiplatforma" e "Sviluppo nativo" si ritiene quindi più conveniente lo sviluppo di un'applicazione multi piattaforma. Nello specifico si consiglia l'utilizzo di *framework* quali *React Native* e *Xamarin* con la preferenza per quest'ultimo.

Breve considerazione sullo sviluppo mobile Dall'analisi del dominio applicativo emerge come un'applicazione di tipo mobile sia la scelta più adatta. La scelta è basata sulla tipologia di utenti e sul tipico uso ipotizzato per l'applicazione. Tuttavia come precedentemente detto l'obbligatorietà di comunicare con l'ITF tramite API REST snaturerebbe il concetto stesso di [blockchain](#), poichè l'IW vedrebbe il componente REST come fonte centralizzata e non verificabile delle informazioni. Ne un'applicazione Desktop risulterebbe notevolmente più appropriata dal punto di vista tecnologico, ma fuori contesto dal punto di vista dell'uso previsto. Al fine di effettuare una scelta finale bisogna tenere sempre in mente questi due fattori e considerare cosa si vuole ottenere.

Ritengo che l'utente dell'IW non sia in grado di apprezzare questo concetto di fiducia e che potrebbe apprezzare maggiormente il fatto che l'applicativo sia mobile.

3.3.4 Motivazioni

Aspetti Positivi

A seguito dell'analisi sopra proposta sono stati individuati i seguenti aspetti positivi:

- * lo sviluppo di un'applicazione mobile *Android* e *iOS* porterebbe *MonoKee* alla portata della quasi totalità dei possibili utenti;
- * uno sviluppo con un *framework* multi piattaforma abbatterebbe i costi di produzione dell'applicazione, pur garantendo risultati accettabili;
- * i *framework* multi piattaforma portati in esame (*Xamarin* e *React Native*) sono ampiamente utilizzati e supportati da grandi aziende IT. Questo garantisce un elevato grado di affidabilità e una ampia documentazione;
- * un eventuale uso di *Xamarin* potrebbe facilitare una successiva implementazione di un'applicazione *Windows Phone*.
- * seppur *MonoKee* utilizza una soluzione basata su *blockchain*, l'IW non risulta colpito da questa ulteriore complessità.

Fattori di rischio

Durante la fase di analisi iniziale sono stati individuati alcuni possibili rischi a cui si potrà andare incontro. Si è quindi proceduto a elaborare delle possibili soluzioni per far fronte a tali rischi.

1. Comunicazione IW-ITF

Descrizione: La comunicazione tra IW e ITF dovrebbe avvenire attraverso chiamate alla *blockchain*. Questo comporta l'uso di librerie per dispositivi mobili poco collaudate e piene di incognite..

Soluzione: Provvedere ad una comunicazione basata su protocollo RESTful..

2. Visione centralizzata

Descrizione: Un'applicazione mobile di questo tipo per l'IW potrebbe non essere considerata come strumento di IAM distribuito, ma potrebbe essere vista come centralizzata..

Soluzione: Inserire note all'interno dell'applicazione o all'interno del sito di *Monokee* per rendere edotti gli utenti del reale funzionamento del servizio..

3. Inesperienza nello sviluppo Xamarin

Descrizione: Il team non ha esperienza nello sviluppo di applicazioni mobili..

Soluzione: Rendere edotto il responsabili del progetto, il quale metterà a disposizione del personale per impartire lezioni sul *framework Xamarin*..

3.3.5 Conclusione Studio di fattibilità IW

Da questo primo studio di fattibilità emerge come, da un punto di vista dell'utente, lo sviluppo di un'applicazione mobile sia maggiormente adatto. Invece, da un punto di

vista tecnologico, risulta come ci siano delle problematiche inerenti alla comunicazione tra i componenti IW e ITF. Riguardo questo si ritiene che lo sviluppo di un applicativo Desktop risulterebbe più adatto, ma molto probabilmente mal visto dalla maggioranza degli utenti finali. Per quanto detto si conclude ribadendo la fattibilità del progetto come applicazione mobile sviluppata con un *framework* multi piattaforma. Per la scelta del *framework* si consiglia *Xamarin*.

3.4 Studio di fattibilità Service Provider

3.4.1 Sintesi dello studio di fattibilità

Lo studio inizia descrivendo come il SP si cali in questo contesto. Si prosegue analizzando il dominio applicativo. Da questo emerge un utilizzo da personale specializzato in orario lavorativo. Si è effettuata, poi, un'analisi sui due principali tipi di sviluppo: distribuito o centralizzato. Alla fine di una breve analisi emerge una preferenza per l'ultimo. All'interno del documento sono presenti anche una trattazione di una serie di tecnologie (sia a livello di librerie per la comunicazione con la rete, che di librerie *front end*) che lo sviluppo di un applicativo di questo tipo potrebbe avere bisogno.

3.4.2 Descrizione Service Provider

Il progetto ha come scopo la creazione di un componente chiamato *Service Provider* (SP). L'applicativo si colloca nel contesto di un'estensione del servizio *Monokee* basata su [blockchain](#). L'estensione offre un sistema di [Identity Access Management](#) composto da quattro principali fattori:

- * Identity Wallet (IW);
- * Service Provider (SP);
- * Identity Trust Fabric (ITF);
- * Trusted Third Party (TTP).

In sintesi, l'estensione dovrà operare al fine di fornire la possibilità ad un utente di registrare e gestire la propria identità automaticamente tramite l'IW, mandare i propri dati (PII) all'ITF la quale custodirà la sua identità e farà da garante per le asserzioni proveniente dalle TTP. Inoltre, il SP dovrà essere in grado con le informazioni provenienti dall'IW e dall'ITF di garantire o meno l'accesso ai propri servizi. Si fa notare come il componente SP non rappresenta il reale fornitore del servizio, ma solo un elemento dell'architettura che lo rappresenta. Il reale servizio viene erogato da organizzazioni esterne le quali comunicano con il componente SP per garantire o meno l'accesso. Il software SP, più dettagliatamente, dovrà assolvere ai seguenti compito: nell'ambito della ricezione dei dati da un *Identity Wallet* (IW) deve:

- * ricevere da parte dell'IW la chiave pubblica (o l'*hash* di questa);
- * ricevere un riferimento alla locazione dell'*hash* della chiave pubblica all'interno dell'ITF;
- * ricevere altre informazioni necessarie da parte dell'IW con relativo riferimento all'interno dell'ITF;

- * gestire il trasferimento dei dati tramite codice QR.

Nell'ambito della verifica dei dati provenienti dall'IW deve:

- * usare la chiave pubblica dell'IW e il riferimento per verificare l'identità e le varie altre informazione passate dal *wallet*;
- * generare e comparare gli *hash* dei valori ottenuti con quelli presenti nell'ITF;
- * verificare che l'identità e le altre informazioni ottenute siano sufficienti a garantire l'accesso al servizio.

Nell'ambito dell'accesso il SP deve:

- * a seguito della verifica comunica il risultato all'organizzazione che fornisce il servizio, in modo tale da garantire l'accesso all'utente dell'IW.

3.4.3 Studio del dominio

Dominio applicativo

L'applicativo SP dovrà essere usato come strumento abilitatore da parte dei vari fornitori di servizi a partecipare al progetto *MonoKee*. Da un primo studio si pensa che il target di questi servizi sarà lavorativo, successivamente potrà essere considerato l'introduzione di servizi Consumer. Si tratta sostanzialmente di un'applicazione di tipo *server* con scopi essenzialmente di comunicazione. Data la vasta varietà di servizi e di necessità che potrebbe avere il fornitore non risulta definibile un comportamento standard che il SP dovrà tenere, ma si dovrà adattare caso per caso. Possiamo comunque ipotizzare che il suo funzionamento sia necessario solo durante l'orario di ufficio, quindi dalle 7.00 alle 18.00, fuori questi orari sarà possibile fare manutenzione. Nell'ottica dell'introduzione di servizi consumer si dovrebbe comunque tener conto di una disponibilità maggiore. L'applicativo dovrebbe offrire un'interfaccia di manutenzione, accessibile tramite interfaccia grafica da parte del personale del fornitore del servizio. Il *software* deve essere utilizzato dal personale IT delle varie organizzazioni che utilizzano il servizio, per questa ragione si può dare per scontato che l'utente generico possieda delle competenze informatiche avanzate.

3.4.4 Dominio tecnologico

Il *service provider* deve operare come intermediario tra l'IW, l'ITF e il reale fornitore del servizio. Le comunicazioni dovrebbero seguire lo schema proposto in figura 3.4. A seguito dello studio di fattibilità relativo all'IW è emerso come la connessione tra IW e SP debba avvenire tramite protocollo REST. Mentre la comunicazione con l'ITF deve avvenire tramite [blockchain](#). Relativamente alla comunicazione verso il fornitore vero e proprio non si possono fare considerazioni in quanto queste possono variare significativamente.

Si evidenziano essenzialmente due principali opzioni per la costruzione di questo applicativo. Il primo è l'utilizzo, anche per esso come per l'ITF, di un approccio totalmente distribuito basato su *blockchain*. Il secondo approccio consiste nello sviluppo di un'applicazione tradizionale.

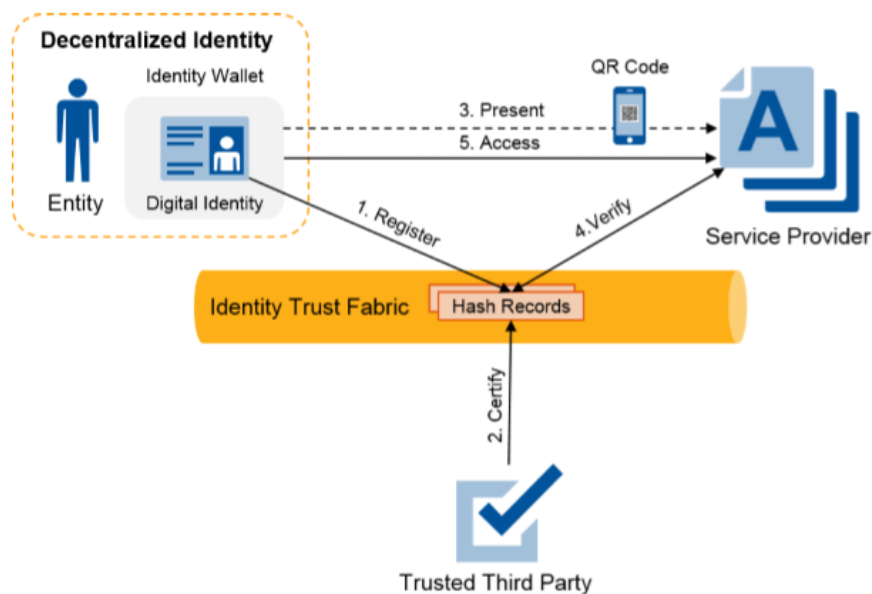


Figura 3.4: Diagramma flussi tra i vari componenti

Sviluppo distribuito

Questo approccio prevede che la logica applicativa sia totalmente affidata a codice eseguito su [blockchain](#). Questo comporterebbe una disponibilità continuativa sempre garantita e altre caratteristiche di affidabilità e sicurezza. Come punto negativo si evidenzia che una soluzione del genere implicherebbe l'uso di molte tecnologie non completamente mature e di linguaggi in molti casi incompleti. Inoltre questa scelta implicherebbe l'utilizzo della stessa *blockchain* presente nell'ITF. I vantaggi attribuibili a questo approccio non sono considerati fondamentali al fine di una buona implementazione del SP, di contro gli svantaggi risultano particolarmente pesanti. L'applicazione di un approccio di questo genere anche se possibile risulta sconsigliato. Uno sviluppo di questo tipo, almeno in reti di tipo *Permissionless*, comporta un cambio di stile di programmazione dovuto all'alto costo delle operazioni. Come descritto nello studio tecnologico *Ethereum* ciò comporta le seguenti diversità:

- * alcuni pattern non risultano applicabili;
- * complessità lineari sono difficilmente giustificabili;
- * uso di pattern ad hoc.

Sviluppo tradizionale

La seconda opzione risulta essere una più tradizionale applicazione *server* che comunica tramite librerie alla *blockchain*. Si consiglia l'uso di linguaggi fortemente tipati quali:

- * C++;
- * C Sharp;
- * Java.

Data l'alta diffusione di *Javascript* e *NodeJS* nella comunità *Ethereum* si consigliano pure questi. In base al linguaggio scelto e alla tecnologia *blockchain* scelta, si dovranno utilizzare differenti librerie per effettuare la comunicazione tra la rete e l'applicativo. Nel caso di *Ethereum* si propongono le seguenti librerie:

Tabella 3.2: Tabella comparivi linguaggio per sviluppo SP

Linguaggio	Libreria	Note
C++	cpp-ethereum	-
C Sharp	Nethereum	Questa soluzione si collega particolarmente bene alla scelta di Xamarin per
JS e NodeJS	Web3	-
Java	Web3j	-

Di seguito si procederà alla trattazione di alcuni degli strumenti che si ritengono più utili.

Nethereum : è uno strumento che permette una facile integrazione con il client in applicazioni .NET. Fornisce una suite di librerie *open source* che aiutano a prototipare applicazione .NET velocemente. E' disponibile anche nel sotto insieme *Xamarin*. La documentazione è presente e sembra ben strutturata e di ottima qualità. Inoltre potrebbe rappresentare una buona soluzione in caso di scelta di *Microsoft Azure Blockchain*. Il sito del progetto è il seguente www.nethereum.com.

Web3 : è una collezione di librerie che permettono di interagire con un nodo remoto o locale usando una connessione *HTTP* o *IPC*. *Web3* è presente in *npm*, *meteor*, *pure js*. Per il suo funzionamento è necessario avere un client attivo nel proprio computer. *Web3* supporta *Mist* e *Metamask*. Il sito del progetto è il seguente: web3js.readthedocs.io.

Web3J : si tratta di una libreria analoga a *Web3* per *Java*.

Mist : è un browser sviluppato direttamente dal team *Ethereum* in grado di operare transazioni direttamente nella *blockchain* senza la necessità di possedere un intero nodo. È estremamente immaturo e non utilizzabile in produzione. Si riporta di seguito il sito del progetto: github.com/ethereum/mist.

Metamask : è uno plugin disponibile per i browser *Chrome*, *Firefox* e *Opera* che permette di interfacciarsi alla rete *Ethereum* senza la necessità di eseguire in intero nodo della rete. Il *plugin* include un *wallet* con cui l'utente può inserire il proprio account tramite la chiave privata. Una volta inserito l'account il *plugin* farà da tramite tra l'applicazione e la rete. In caso, invece, si opti per la scelta di *Hyperledger* la scelta risulterebbe molto più semplice in quanto la *blockchain* in questione fornisce un'API per la comunicazione REST.

Scelta del client Ethereum

Il *client* è un componente che implementa il protocollo di comunicazione di *Ethereum*. *Ethereum* diversamente da *Hyperledger* presenta una realtà molto varia e frastagliata. Solo dal punto di vista del *client* ci sono multiple implementazioni per differenti sistemi operativi ed in differenti linguaggi. Questa diversità viene vista dalla *community* come un indicatore di salute per l'intero ecosistema. Il protocollo in ogni caso è sempre

lo stesso ed è definito nel così detto *Yellow Paper*¹², in sostanza si basa sull'utilizzo di file *Json*. Fino al settembre 2016 erano presenti le alternative esposte in tabella 3.3. I client appena citati sono accessibili tramite apposite librerie, un buon esempio

Tabella 3.3: Tabella comparivi client Ethereum

Client	Linguaggio	Sviluppatore
Go-ethereum	Go	Ethereum Foundation
Parity	Rust	Ethcore
C++-ethereum	C++	Ethereum Foundation
Pyethapp	Python	Ethereum Foundation
Ethereumjs-lib	Javascript	Ethereum Foundation
Ethereum(J)	Java	<ether.camp>
Ruby-ethereum	Ruby	Jan Xie
EthereumH	Haskell	BlockApps

potrebbe essere *web3* per *Javascript*.

Scelta del client Hyperledger

In caso si dovesse optare per una scelta basata su *Hyperledger* quale base dell'ITF bisognerà ricadere sull'unica soluzione proposta dal team di sviluppo, cioè quella di utilizzare direttamente una comunicazione REST. Il team offre uno strumento detto *Composer* con il quale si potrà definire un'interfaccia per operare la comunicazione REST. Al fine di poter gestire efficientemente queste chiamate lato applicazione SP si consigliano le librerie esposte in tabella 3.4. Data l'amplissima diffusione delle API

Tabella 3.4: Tabella comparivi client Ethereum

Linguaggio	Librerie	Sito
.NET	WCF REST Starter Kit	www.asp.net/downloads/starter-kits/wcf-rest
.NET	OpenRasta	www.openrasta.org
.NET	Service Stack	www.servicestack.net
Java	Jersey	www.jersey.java.net
Java	RESREasy	www.jboss.org/resteasy
Java	Restlet	www.restlet.org
C++	linavajo	www.libnavajo.org
C++	C++ RESTful framework	www.github.com/corvusoft/restbed
C++	C++ REST SDK	www.github.com/Microsoft/cpprestsdk

REST e di queste librerie non si procede ad una trattazione analitica.

¹²site:ethereum-yellow-paper.

3.4.5 Conclusioni scelta sviluppo

Considerando quanto precedentemente detto, lo sviluppo tradizionale sembrerebbe avere meno incognite e un ampio repertorio di librerie utilizzabili. Si propone, quindi un'architettura del secondo tipo. Inoltre, si vuole fare notare come l'utilizzo delle soluzioni .NET potrebbero rivelarsi molto vantaggiose in quanto facilmente integrabili con il componente IW e *Azure Blockchain*.

3.4.6 Motivazioni

Aspetti positivi

A seguito dell'analisi sopra proposta sono stati individuati i seguenti aspetti positivi:

- * lo sviluppo di un'applicazione server tradizionale comporta uno sviluppo molto semplice e immediato, grazie anche alla disponibilità di un ampio repertorio di librerie;
- * la comunicazione con la blockchain risulta in ogni caso facilmente implementabile grazie all'utilizzo di apposite librerie.
- * esiste un'ampia scelta di librerie front end per ogni possibile linguaggio di sviluppo.

Fattori di rischio

4. Inesperienza nello sviluppo *C Sharp*

Descrizione: Non si ha esperienza nello sviluppo di applicazioni *C Sharp*..

Soluzione: Rendere edotto il responsabili del progetto, il quale metterà a disposizione del personale per impartire supporto su *C Sharp*..

5. difficoltà di integrazione con *Monokee*

Descrizione: Il componente SP è il componente che deve essere integrato in *Monokee*..

Soluzione: Rendere edotto il responsabili del progetto, il quale metterà a disposizione del personale..

6. Prestazioni delle chiamate alla rete *blockchain*

Descrizione: Il componente SP deve interrogare la rete *blockchain*, questo potrebbe rappresentare un problema di performance.

Soluzione: Rendere edotto il responsabili del progetto, valutare soluzioni alternative..

3.4.7 Conclusioni

Dal presente studio emerge come la creazione di un SP sviluppato come applicativo server, e non come applicazione distribuita, possa essere un'ottima opzione implementativa. Lo studio non presenta particolari rischi. In definitiva, si ritiene che un approccio di questo tipo sia fattibile nei tempi dello stage.

3.5 Obiettivi

Scopo delle attività di stage è il raggiungimento dei seguenti obiettivi.

Obiettivi minimi

- * codifica dei moduli SP e IW;
- * documenti di analisi dei requisiti;
- * documenti di architettura.

Obiettivi opzionali

- * documenti di progettazione;
- * documenti di testing;
- * documenti di validazione (anomalie e bug).

3.6 Pianificazione

Il lavoro durante lo stage è stato svolto seguendo la seguente pianificazione iniziale:

- * **Studio Fattibilità** (40 ore): questa fase è focalizzata allo studio della tecnologia *blockchain* da adottare e il suo impiego nello specifico caso d'uso. La valutazione verrà effettuata valutando le capacità di utilizzo delle tecnologie e interpretazione delle informazioni.

Prodotti attesi:

1. Documento: Monokee – Identity Wallet, studio di Fattibilità
2. Documento: Monokee – Service Provider, studio di fattibilità

- * **Analisi requisiti** (40 ore): al termine di questo periodo i casi d'uso saranno definiti e si avrà il tracciamento requisiti-casi d'uso. I requisiti saranno una rappresentazione delle 5 funzionalità core che i moduli dovranno erogare:

1. Registrazione: il *Wallet* crea l'identità digitale dell'utente e ne associa una chiave privata e pubblica; interagisce quindi con il componente ITF per registrare l'associazione *Identità-Service Provider*.
2. Certificazione (da capire il coinvolgimento del *Wallet* e del *Service Provider*): un ente terzo può validare l'identità dell'utente tramite un processo di "*identity proofing*"; in caso di validazione positiva l'ente terzo può certificare l'identità (o una parte degli attributi del profilo) firmandoli con la propria chiave privata
3. Presentazione: la chiave pubblica e il riferimento a dove trovarne l'*hash* viene inviato al Service Provider; il fornitore del servizio a questo punto può chiedere l'invio di ulteriori attributi dell'utente che possono essere presenti nel suo *Wallet*; gli attributi verranno inviati firmati tramite *QR-Code*
4. Verifica: il *Service Provider* utilizza le informazioni ricevute per verificare l'identità e gli attributi tramite un confronto dei valori *hash* nell'ITF.
5. Access: a seguito di una verifica positiva dell'identità il *provider* dei servizi concede l'accesso all'applicazione/servizio.

Prodotti attesi:

1. Documento: Monokee – Identity Wallet, analisi e specifica dei requisiti
 2. Documento: Monokee – Service Provider, analisi e specifica dei requisiti
- * **Progettazione architetturale** (40 ore): avrà come risultato l'architettura generale che implementa le funzionalità rilevate dai casi d'uso. La valutazione verrà effettuata valutando le capacità di progettazione di un'architettura a partire dalle funzionalità individuate; Prodotti attesi:
1. Documento: Monokee – Identity Wallet, architettura
 2. Documento: Monokee – Service provider, architettura
- * **Progettazione dettaglio** (60 ore): come risultato si avrà la definizione dei metodi in pseudo-codice. La valutazione verrà effettuata valutando le capacità di traduzione in pseudo-codice dell'architettura progettata. Prodotti attesi:
1. Documento: Monokee – Identity Wallet, progettazione
 2. Documento: Monokee – Service Provider, progettazione
- * **Codifica e Verifica** (120 ore): sarà realizzata la codifica dei metodi e saranno effettuati i test di unità e integrazione. La valutazione verrà effettuata valutando l'apprendimento e la capacità di implementazione della tecnologia scelta; Prodotti attesi:
1. Sorgenti del modulo Identity Wallet basati su tecnologia mobile (da valutare l'implementazione tramite Xamarin o strumenti nativi)
 2. Sorgenti del modulo Service Provider
 3. Documento: Monokee – Identity Wallet, testing
 4. Documento: Monokee – Service Provider, testing
- * **Validazione** (20 ore): al termine si avrà il prodotto software richiesto. Verrà valutato il software risultante tramite fase di testing. Prodotti attesi:
1. Documento: Monokee – Identity Wallet, anomalie e bug
 2. Documento: Monokee – Service Provider, anomalie e bug

Capitolo 4

Analisi dei requisiti

Breve introduzione al capitolo

Questo capitolo ha lo scopo di fornire una definizione dei requisiti individuati per la creazione del prodotto Identity Wallet (IW). Le metodologie usate sono tratte dal capitolo quattro di **som:swe** Più in particolare il presente capitolo si prefigge di:

- * individuare le fonti per la deduzione dei requisiti;
- * dedurre i requisiti dalle fonti;
- * descrivere i requisiti individuati;
- * catalogare i requisiti individuati;
- * fissare un ordine di priorità tra i requisiti individuati;

4.1 Specifiche in Linguaggio Naturale

Il linguaggio naturale ha un'enorme potenza espressiva ma essendo inerentemente ambiguo può portare ad incomprensioni; è quindi necessario limitarne l'utilizzo e standardizzarlo, in modo da ridurre al minimo le possibili ambiguità. È comunque fondamentale evitare di utilizzare espressioni e acronimi che possano essere fraintendibili dagli stakeholders, a tal proposito in fondo al documento è presente una lista degli acronimi utilizzati.

4.2 Specifiche in Linguaggio Strutturato

Il linguaggio strutturato mantiene gran parte dell'espressività del linguaggio naturale, fornendo però uno standard schematico che permette l'uniformità della descrizione dei vari requisiti. Sebbene l'utilizzo di un linguaggio strutturato permetta di organizzare i requisiti in modo più ordinato e comprensibile, talvolta la ridotta espressività rende difficile la definizione di requisiti complessi. A tal proposito è possibile integrare la specifica in linguaggio strutturato con una descrizione in linguaggio naturale.

4.3 Specifiche in Linguaggio UML Use Case

Per la definizione dei diagrammi UML dei casi d'uso, viene utilizzato lo standard UML 2.0¹. Nei diagrammi dei casi d'uso vengono mostrati gli attori coinvolti in un'interazione con il sistema in modo schematico, indicando i nomi delle parti coinvolte. Eventuali informazioni aggiuntive possono essere espresse testualmente.

4.4 Analisi dei requisiti IW

4.4.1 Casi d'uso

Per lo studio dei casi di utilizzo del prodotto sono stati creati dei diagrammi. I diagrammi dei casi d'uso (in inglese *Use Case Diagram*) sono diagrammi di tipo [Unified Modeling Language \(UML\)](#) dedicati alla descrizione delle funzioni o dei servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso.

Descrizione Attori

I tipi di attori principali che andranno ad interagire direttamente con il sistema sono essenzialmente tre:

- * utente;
- * utente non registrato;
- * utente autenticato.

Tra di essi è presente una relazione di generalizzazione che vede l'attore utente come generalizzazione degli attori utente non registrato e utente registrato. Questo tipo di generalizzazione viene rappresentata graficamente in figura 4.1. Sono stati individuati



Figura 4.1: Gerarchia utenti user case

i seguenti attori secondari: ITF, *MonoKee*.

¹[site:uml](#).

Attori principali

- * **Utente:** l'attore utente è un fruitore generico del sistema. Potrebbe avere o non avere effettuato l'accesso all'applicazione. Da lui derivano gli attori utente non registrato e utente autenticato.
- * **Utente non registrato:** l'attore utente non registrato è una particolare specializzazione dell'attore utente. Unica sua caratteristica è quella di non essere riconosciuto come utente di *MonoKee*.
- * **Utente autenticato:** l'attore utente autenticato è una particolare specializzazione dell'attore utente. Rappresenta un utente che ha effettuato l'accesso al sistema e che è stato riconosciuto all'interno del sistema *MonoKee*.

Attori secondari

- * **ITF:** è il componente dell'estensione che ha il compito di conservare e convalidare tutte le informazioni provenienti dall'IW.
- * **MonoKee:** è il componente centrale dell'attuale servizio *MonoKee*. Ha il compito di fornire le informazioni di accesso del servizio *MonoKee*.

UC1: Azioni utente generico

Figura 4.2: Use Case - UC1: Azioni utente generico

Descrizione L'utente può visualizzare le informazioni sull'applicazione

Attore primario Utente

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione

Postcondizioni L'utente ha eseguito le azioni che desiderava compiere in relazione alle sue possibilità

Scenario principale

1. UC1.1 Visualizza info applicazione

Scenari alternativi Nessuno

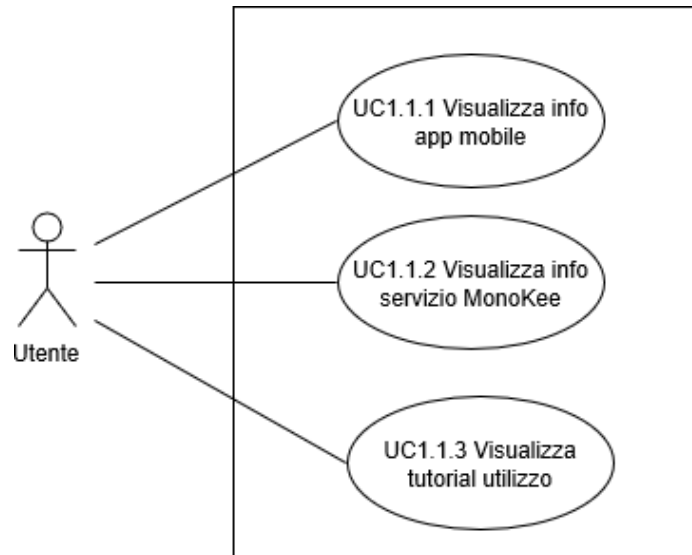
UC1.1 – Visualizza info applicazione

Figura 4.3: Use Case - UC1.1 – Visualizza info applicazione

Descrizione Il sistema deve visualizzare le informazioni relative all'applicazione mobile e al servizio MonoKee

Attore primario Utente

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione

Postcondizioni L'utente ha visualizzato le informazioni che desiderava riguardo l'applicazione

Scenario principale

1. UC1.1.1 Visualizza info applicazione
2. UC1.1.2 Visualizza info servizio MonoKee
3. UC1.1.3 Visualizza tutorial utilizzo

Scenari alternativi Nessuno

UC1.1.1 – Visualizza info app mobile

Descrizione Il sistema deve visualizzare le informazioni tecniche relative all'applicazione mobile

Attore primario Utente

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione ed ha richiesto la visualizzazione delle informazioni tecniche relative all'applicazione mobile

Postcondizioni L'utente ha visualizzato le informazioni con le informazioni tecniche relative all'applicazione mobile

Scenario principale L'utente visualizza un messaggio con le informazioni tecniche relative all'applicazione mobile

Scenari alternativi Nessuno

UC1.1.2 – Visualizza info servizio MonoKee

Descrizione Il sistema deve visualizzare le informazioni relative al servizio *MonoKee*

Attore primario Utente

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione ed ha richiesto la visualizzazione delle informazioni relative al servizio *MonoKee*

Postcondizioni L'utente ha visualizzato le informazioni con le informazioni relative al servizio *MonoKee*

Scenario principale L'utente visualizza un messaggio con le informazioni relative al servizio MonoKee

Scenari alternativi Nessuno

UC1.1.3 – Visualizza tutorial utilizzo

Descrizione Il sistema deve visualizzare un tutorial su come utilizzare l'applicazione IW

Attore primario Utente

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione ed ha richiesto la visualizzazione di un tutorial su come utilizzare l'applicazione IW

Postcondizioni L'utente ha visualizzato il tutorial su come utilizzare l'applicazione IW

Scenario principale L'utente visualizza un tutorial su come utilizzare l'applicazione IW

Scenari alternativi Nessuno

UC2 – Azioni utente non registrato

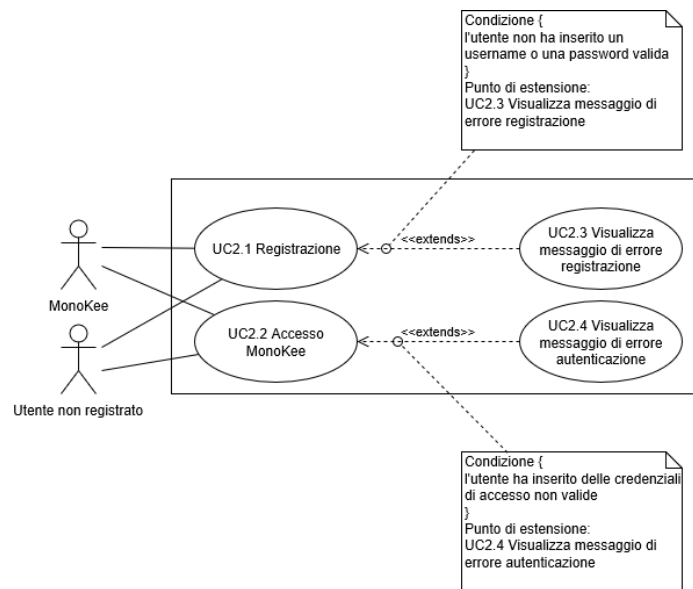


Figura 4.4: Use Case - UC2: Azioni utente non registrato

Descrizione L'utente non registrato può eseguire le operazioni di registrazione e accesso al servizio *MonoKee*

Attore primario Utente non registrato

Attore secondario *MonoKee*

Precondizioni L'utente ha avviato l'applicazione ed non è ancora riconosciuto nel sistema

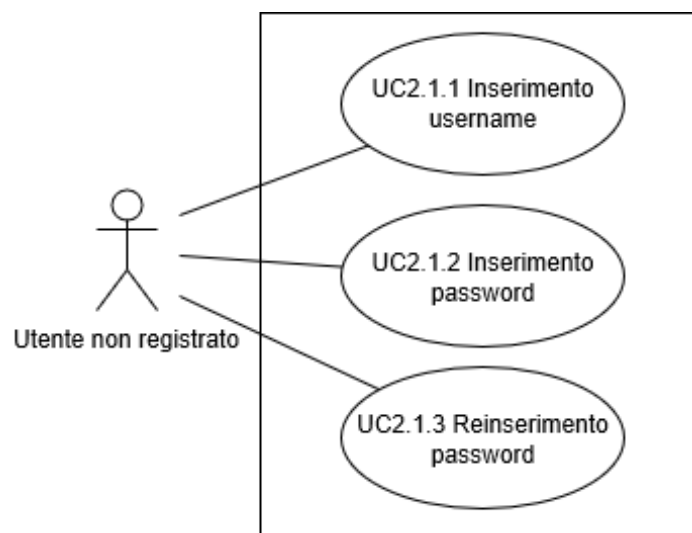
Postcondizioni L'utente ha eseguito le azioni che desiderava compiere in relazione alla condizione di non essere registrato

Scenario principale

1. UC2.1 Registrazione
2. UC2.2 Accesso *MonoKee*

Scenari alternativi

1. l'utente ha fornito dati di registrazione non validi o il doppio inserimento della password non coincide: UC2.3 Visualizzazione messaggio di errore registrazione.
2. l'utente ha fornito username e password non corrispondenti ha nessun utente registrato al servizio: UC2.4 Visualizzazione messaggio di errore autenticazione.

UC2.1 – Registrazione**Figura 4.5:** Use Case - UC2.1: Registrazione

Descrizione L'utente non registrato può eseguire l'operazione di registrazione

Attore primario Utente non registrato

Attore secondario *MonoKee*

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed ha espresso la volontà di effettuare la registrazione al servizio *MonoKee*

Postcondizioni L'utente ha eseguito l'operazione di registrazione al sistema

Scenario principale

1. UC2.1.1 Inserimento username
2. UC2.1.2 Inserimento password
3. UC2.1.3 Reinserimento password

Scenari alternativi Nessuno

UC2.1.1 – Inserimento username

Descrizione L'utente non registrato deve inserire un username per l'operazione di registrazione

Attore primario Utente non registrato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed il sistema richiede l'inserimento di un username per l'operazione di registrazione

Postcondizioni L'utente ha inserito l'username per la registrazione

Scenario principale L'utente non registrato inserisce una stringa tramite l'utilizzo di una *text box*

Scenari alternativi Nessuno

UC2.1.2 – Inserimento password

Descrizione L'utente non registrato deve inserire una password per l'operazione di registrazione

Attore primario Utente non registrato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed il sistema richiede l'inserimento di una password per l'operazione di registrazione

Postcondizioni L'utente ha inserito la password per la registrazione

Scenario principale L'utente non registrato inserisce una stringa tramite l'utilizzo di una *text box*

Scenari alternativi Nessuno

UC2.1.3 – Reinserimento password

Descrizione L'utente non registrato deve reinserire la password per l'operazione di registrazione

Attore primario Utente non registrato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed il sistema richiede il reinserimento di una password per l'operazione di registrazione

Postcondizioni L'utente ha reinserito la password per la registrazione

Scenario principale L'utente non registrato inserisce una stringa tramite l'utilizzo di una *text box*

Scenari alternativi Nessuno

UC2.2 – Accesso MonoKee

Figura 4.6: Use Case - UC2.1: Accesso MonoKee

Descrizione L'utente non registrato può eseguire l'operazione di autenticazione

Attore primario Utente non registrato

Attore secondario *MonoKee*

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed ha espresso la volontà di effettuare l'autenticazione al servizio *MonoKee*

Postcondizioni L'utente ha eseguito l'operazione di accesso al sistema ed è quindi ora riconosciuto come utente autenticato

Scenario principale

1. UC2.1.1 Inserimento username
2. UC2.1.2 Inserimento password

Scenari alternativi Nessuno

UC2.2.1 – Inserimento username

Descrizione L'utente non registrato deve inserire un *username* per l'operazione di autenticazione

Attore primario Utente non registrato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed il sistema richiede l'inserimento di un *username* per l'operazione di autenticazione

Postcondizioni L'utente ha inserito *username* per l'autenticazione

Scenario principale L'utente non registrato inserisce una stringa tramite l'utilizzo di una *text box*

Scenari alternativi Nessuno

UC2.2.2 – Inserimento password

Descrizione L'utente non registrato deve inserire una password per l'operazione di autenticazione

Attore primario Utente non registrato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed il sistema richiede l'inserimento di una password per l'operazione di autenticazione

Postcondizioni L'utente ha inserito la password per l'autenticazione

Scenario principale L'utente non registrato inserisce una stringa tramite l'utilizzo di una *text box*

Scenari alternativi Nessuno

UC2.3 – Visualizza messaggio di errore registrazione

Descrizione L'utente non registrato fornisce username già esistente o il doppio inserimento della password non coincide

Attore primario Utente non registrato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed il sistema ha inserito un *username* già esistente o delle password non coincidenti durante la registrazione richiede l'inserimento di una password per l'operazione di autenticazione

Postcondizioni L'utente ha visualizzato un messaggio di errore relativo all'impossibilità di effettuare la registrazione con i dati forniti

Scenario principale L'utente visualizza un messaggio di errore relativo all'impossibilità di effettuare la registrazione con i dati forniti

Scenari alternativi Nessuno

UC2.4 – Visualizza messaggio di errore autenticazione

Descrizione L'utente non registrato fornisce username e password che non corrispondono a nessun utente registrato al servizio *MonoKee*

Attore primario Utente non registrato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, non è ancora riconosciuto nel sistema ed il sistema ha inserito un username e una password che non corrispondono a nessun utente registrato al servizio *MonoKee*

Postcondizioni L'utente ha visualizzato un messaggio di errore relativo all'impossibilità di effettuare l'autenticazione

Scenario principale L'utente visualizza un messaggio di errore relativo all'impossibilità di effettuare l'autenticazione

Scenari alternativi Nessuno

UC3 – Azioni utente autenticato

Descrizione L'utente autenticato può eseguire le operazioni legate alla gestione della sua identità e alla presentazione dei propri dati ad un SP

Attore primario Utente Autenticato



Figura 4.7: Use Case - UC3: Azioni utente autenticato

Attore secondario ITF

Precondizioni L'utente ha avviato l'applicazione ed è riconosciuto nel sistema come utente di *MonoKee*

Postcondizioni L'utente ha eseguito le azioni che desiderava compiere in relazione alla condizione essere riconosciuto come utente di *MonoKee*

Scenario principale

1. UC3.1 Visualizza QR dell'informazione certificata
2. UC3.2 Visualizza chiave pubblica
3. UC3.3 Visualizza chiave privata
4. UC3.4 Inserimento informazione personale
5. UC3.5 Visualizza lista certificazioni
6. UC3.6 Elimina certificazione

Scenari alternativi Nessuno

UC3.1 – Visualizza QR dell'informazione certificata

Descrizione L'utente autenticato può visualizzare nel proprio schermo un codice QR che rappresenta un'informazione certificata

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee* e ha richiesto di visualizzare il codice QR di una certificazione precedentemente inserita.

Postcondizioni L'utente ha visualizzato il codice QR che rappresenta la certificazione selezionata

Scenario principale L'utente seleziona e poi visualizza il codice QR che rappresenta la certificazione selezionata

Scenari alternativi Nessuno

UC3.2 – Visualizza chiave pubblica

Descrizione L'utente autenticato può visualizzare la chiave pubblica generata al momento della registrazione

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee* e ha richiesto la visualizzazione della chiave pubblica.

Postcondizioni L'utente ha visualizzato la propria chiave pubblica precedentemente generata

Scenario principale L'utente visualizza la propria chiave pubblica precedentemente generata

Scenari alternativi Nessuno

UC3.2 – Visualizza chiave privata

Descrizione L'utente autenticato può visualizzare la chiave privata generata al momento della registrazione

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee* e ha richiesto la visualizzazione della chiave privata.

Postcondizioni L'utente ha visualizzato la propria chiave privata precedentemente generata

Scenario principale L'utente visualizza la propria chiave privata precedentemente generata

Scenari alternativi Nessuno

UC3.4 – Inserimento informazione personale



Figura 4.8: Use Case - UC3.4: Inserimento informazione personale

Descrizione L'utente autenticato può inserire una certificazione e sottometerla all'ITF

Attore primario Utente Autenticato

Attore secondario ITF

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee*, e ha intende inserire una nuova certificazione alla propria identità

Postcondizioni L'utente ha inserito la certificazione e questa è stata presentata all'ITF

Scenario principale

1. UC3.4.1 Inserimento nome certificazione
2. UC3.4.2 Inserimento descrizione certificazione
3. UC3.4.3 Visualizza resoconto

Scenari alternativi Nessuno

UC3.4.1 – Inserimento nome certificazione

Descrizione L'utente autenticato deve inserire un nome per l'operazione di inserimento certificazione

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema ed il sistema richiede l'inserimento di un nome per l'operazione di inserimento certificazione

Postcondizioni L'utente ha inserito il nome per l'inserimento della certificazione

Scenario principale L'utente autenticato inserisce una stringa tramite l'utilizzo di una *text box*

Scenari alternativi Nessuno

UC3.4.2 – Inserimento descrizione certificazione

Descrizione L'utente autenticato deve inserire una descrizione per l'operazione di inserimento certificazione

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema ed il sistema richiede l'inserimento di una descrizione per l'operazione di inserimento certificazione

Postcondizioni L'utente ha inserito la descrizione per l'inserimento della certificazione

Scenario principale L'utente autenticato inserisce un insieme di stringhe tramite l'utilizzo di una *text box*

Scenari alternativi Nessuno

UC3.4.3 – Visualizza resoconto

Descrizione L'utente autenticato può visualizzare un resoconto dei dati inseriti durante la procedura di inserimento certificazione

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee*, ha iniziato una procedura di inserimento certificazione e ha richiesto la visualizzazione del resoconto dei dati inseriti

Postcondizioni L'utente ha visualizzato un resoconto dei dati inseriti durante la procedura di inserimento certificazione

Scenario principale L'utente visualizza un resoconto dei dati inseriti durante la procedura di inserimento certificazione

Scenari alternativi Nessuno

UC3.5 – Visualizza lista certificazioni

Figura 4.9: Use Case - UC3.5: Visualizza lista certificazioni

Descrizione L'utente autenticato può visualizzare una lista con il nome e l'identificativo della certificazione associate alla propria identità

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee* e ha richiesto la visualizzazione della lista delle certificazioni

Postcondizioni L'utente ha visualizzato la lista delle certificazioni

Scenario principale

1. UC3.5.1 Visualizza singola certificazione

Scenari alternativi Nessuno

UC3.5.1 – Visualizza singola certificazione

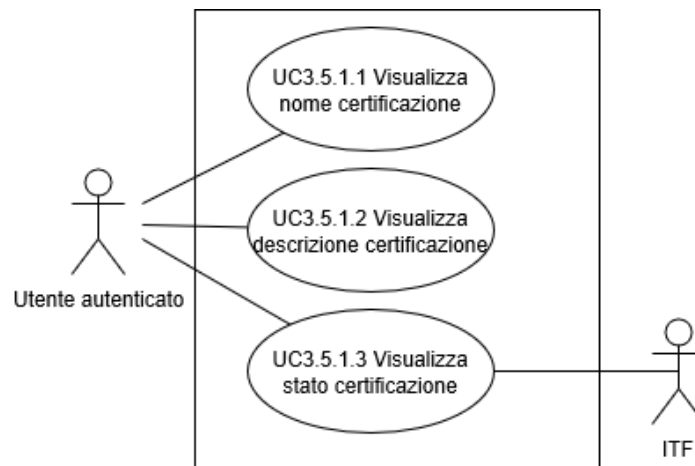


Figura 4.10: Use Case - UC3.5.1: Visualizza singola certificazione

Descrizione L'utente autenticato può visualizzare i dettagli di una certificazione selezionata della lista delle certificazioni

Attore primario Utente Autenticato

Attore secondario ITF

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee* e ha richiesto la visualizzazione di una specifica *entry* della lista delle certificazioni

Postcondizioni L'utente ha visualizzato i dettagli di una specifica certificazione della lista

Scenario principale

1. UC3.5.1.1 Visualizza nome certificazione
2. UC3.5.1.2 Visualizza descrizione certificazione
3. UC3.5.1.3 Visualizza stato certificazione

Scenari alternativi Nessuno

UC3.5.1.1 – Visualizza nome certificazione

Descrizione L'utente autenticato può visualizzare il nome di una certificazione selezionata della lista delle certificazioni

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee* e ha richiesto la visualizzazione del nome di una specifica *entry* della lista delle certificazioni

Postcondizioni L'utente ha visualizzato il nome di una specifica certificazione della lista

Scenario principale L'utente visualizza il nome di una specifica certificazione della lista

Scenari alternativi Nessuno

UC3.5.1.2 – Visualizza descrizione certificazione

Descrizione L'utente autenticato può visualizzare la descrizione di una certificazione selezionata della lista delle certificazioni

Attore primario Utente Autenticato

Attore secondario Nessuno

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee* e ha richiesto la visualizzazione della descrizione di una specifica *entry* della lista delle certificazioni

Postcondizioni L'utente ha visualizzato la descrizione di una specifica certificazione della lista

Scenario principale L'utente visualizza la descrizione di una specifica certificazione della lista

Scenari alternativi Nessuno

UC3.5.1.3 – Visualizza stato certificazione

Descrizione L'utente autenticato può visualizzare lo stato di una certificazione selezionata della lista delle certificazioni. L'informazione proviene dall'ITF.

Attore primario Utente Autenticato

Attore secondario ITF

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di *MonoKee* e ha richiesto la visualizzazione dello stato di una specifica *entry* della lista delle certificazioni

Postcondizioni L'utente ha visualizzato lo stato di una specifica certificazione della lista

Scenario principale L'utente visualizza una stringa che può essere confermata da un TTP o non confermata.

Scenari alternativi Nessuno

UC3.6 – Elimina certificazione

Descrizione L'utente autenticato può eliminare una certificazione selezionata

Attore primario Utente Autenticato

Attore secondario ITF

Precondizioni L'utente ha avviato l'applicazione, è riconosciuto nel sistema come utente di MonoKee e ha richiesto l'eliminazione di una specifica certificazione

Postcondizioni La certificazione non è più presente dal sistema e pure dall'ITF

Scenario principale L'utente seleziona e poi esprime la volontà di eliminare la certificazione certificata

Scenari alternativi Nessuno

4.4.2 Tracciamento dei requisiti

Fonti

Per la deduzione dei requisiti utente e di sistema, che verranno presentati nelle sezioni a seguire, sono stati usati come fonti lo studio *Gartner*², il capitolo *Studio di fattibilità IW* e gli *Use Case* presentati nella sezione *Casi d'uso*. La struttura e le convenzioni usate sono ispirate dal capitolo di **som:swe**. In seguito vengono riportate le categorie che vengono usate per la catalogazione:

- * F: requisito funzionale;
- * V: requisito di vincolo;
- * Q: requisito di qualità.

Per l'attribuzione della priorità viene usata la tecnica *MoSCoW*, quindi gli indici usati sono i seguenti:

- * M: must;
- * S: should;
- * C: could;
- * W: will.

Nelle tabelle 4.1, 4.3 e 4.2 sono riassunti i requisiti e il loro tracciamento con gli *use case* delineati in fase di analisi.

²[farah:The-Dawn-of-Decentralized-Identity](#).

Tabella 4.1: Tabella del tracciamento dei requisiti funzionali

Codice	Descrizione	Fonte
R[F][C]0001	Il sistema potrebbe permettere ad un utente di visualizzare le informazioni dell'applicazione	UC1, UC1.1
R[F][C]0002	Il sistema potrebbe permettere di visualizzare le info tecniche dell'applicazione	UC1.1.2
R[F][C]0003	Il sistema potrebbe permettere di visualizzare una descrizione del servizio MonoKee	UC1.1.2
R[F][C]0004	Il sistema potrebbe permettere di visualizzare un tutorial esplicativo sul suo utilizzo	UC1.1.3
R[F][M]0005	Il sistema deve permettere di potersi registrare al servizio	UC2, UC2.1
R[F][M]0006	Il sistema deve permettere di essere riconosciuto dal sistema <i>MonoKee</i>	UC2, UC2.2
R[F][M]0007	Il sistema deve visualizzare un messaggio di errore in caso i dati forniti durante la registrazione non dovessero essere validi	UC2, UC2.3
R[F][M]0008	Il sistema deve visualizzare un messaggio di errore in caso i dati forniti durante la procedura di autenticazione non dovessero essere corretti	UC2, UC2.4
R[F][M]0009	Il sistema deve permettere di inserire uno username nell'ottica della procedura di registrazione	UC2.1.1
R[F][M]0010	Il sistema deve permettere di inserire una password nell'ottica della procedura di registrazione	UC2.1.2
R[F][M]0011	Il sistema deve permettere di reinserire la password nell'ottica della procedura di registrazione	UC2.1.3
R[F][M]0012	Il sistema deve permettere di inserire uno <i>username</i> nell'ottica della procedura di autenticazione	UC2.2.1
R[F][M] 0013	Il sistema deve permettere di inserire una password nell'ottica della procedura di autenticazione	UC2.2.2
R[F][M] 0014	Il sistema deve permettere ad un utente autenticato di poter generare un codice QR di un certificato inserito nel sistema	UC3, UC3.1
R[F][M] 0015	Il sistema deve permettere ad un utente autenticato di visualizzare la chiave pubblica	UC3, UC3.2
R[F][M] 0016	Il sistema deve permettere ad un utente autenticato di visualizzare la chiave privata	UC3, UC3.3
R[F][M] 0017	Il sistema deve permettere ad un utente autenticato di inserire un'informazione personale	UC3, UC3.4
R[F][M] 0018	Il sistema deve permettere ad un utente autenticato di visualizzare una lista di certificazioni associate alla propria identità	UC3, UC3.5
R[F][M] 0019	Il sistema deve permettere ad un utente autenticato di eliminare una certificazione associata alla propria identità	UC3, UC3.6
R[F][M] 0020	Il sistema deve permettere ad un utente autenticato di inserire il nome della certificazione nel contesto dell'inserimento di certificazione	UC3.4.1
R[F][M] 0021	Il sistema deve permettere ad un utente autenticato di una descrizione della certificazione nel contesto dell'inserimento di una certificazione	UC3.4.2

R[F][M] 0022	Il sistema deve permettere ad un utente autenticato di visualizzare un resoconto dei dati inseriti durante la procedura di inserimento certificato	UC3.4.3
R[F][M] 0023	Il sistema deve permettere ad un utente autenticato di visualizzare i dettagli di una singola certificazione	UC3.5.1
R[F][M] 0024	Il sistema deve permettere ad un utente autenticato di visualizzare il nome di una certificazione esistente	UC3.5.1.1
R[F][M] 0025	Il sistema deve permettere ad un utente autenticato di visualizzare la certificazione di una certificazione esistente	UC3.5.1.2
R[F][S] 0026	Il sistema dovrebbe permettere ad un utente autenticato di visualizzare lo stato di una certificazione esistente	UC3.5.1.3

Tabella 4.2: Tabella del tracciamento dei requisiti di vincolo

Codice	Descrizione	Fonte
R[V][M] 0027	Il sistema deve offrire le proprie funzionalità come applicazione mobile	IW Studio di fattibilità
R[V][M] 0028	Il sistema è implementato tramite l'uso di <i>Xamarin</i>	IW Studio di fattibilità
R[V][M] 0029	Il progetto prevede almeno i seguenti quattro ambienti di sviluppo: <i>Local</i> , <i>Test</i> , <i>Staging</i> , <i>Production</i>	IW Studio di fattibilità
R[V][M] 0030	Il prodotto è sviluppato utilizzando uno strumento di <i>linting</i>	IW Studio di fattibilità
R[V][M] 0031	Il sistema deve mantenere la chiave privata sempre in locale	IW Studio di fattibilità

Tabella 4.3: Tabella del tracciamento dei requisiti qualitativi

Codice	Descrizione	Fonte
R[Q][S] 0032	Il progetto prevede un ragionevole set di test di unità e di test di integrazione	-
R[Q][S] 0033	I test possono essere eseguiti localmente o come parte di integrazione continua	-
R[Q][S] 0034	Il sistema solo alla fine sarà testato nel <i>network</i> pubblico di prova	-
R[Q][S] 0035	Il codice sorgente del prodotto e la documentazione necessaria per l'utilizzo sono versionati in <i>repository</i> pubblici usando <i>GitHub</i> , <i>BitBuket</i> o <i>GitLab</i>	-
R[Q][C] 0036	Lo sviluppo si eseguirà utilizzando un approccio incrementale	IW Studio di fattibilità

Tabella 4.4: Tabella del tracciamento dei requisiti con le fonti

Codice	Fonte
--------	-------

R[F][C]0001	UC1, UC1.1
R[F][C]0002	UC1.1.2
R[F][C]0003	UC1.1.2
R[F][C]0004	UC1.1.3
R[F][M]0005	UC2, UC2.1
R[F][M]0006	UC2, UC2.2
R[F][M]0007	UC2, UC2.3
R[F][M]0008	UC2, UC2.4
R[F][M]0009	UC2.1.1
R[F][M]0010	UC2.1.2
R[F][M]0011	UC2.1.3
R[F][M]0012	UC2.2.1
R[F][M] 0013	UC2.2.2
R[F][M] 0014	UC3, UC3.1
R[F][M] 0015	UC3, UC3.2
R[F][M] 0016	UC3, UC3.3
R[F][M] 0017	UC3, UC3.4
R[F][M] 0018	UC3, UC3.5
R[F][M] 0019	UC3, UC3.6
R[F][M] 0020	UC3.4.1
R[F][M] 0021	UC3.4.2
R[F][M] 0022	UC3.4.3
R[F][M] 0023	UC3.5.1
R[F][M] 0024	UC3.5.1.1
R[F][M] 0025	UC3.5.1.2
R[F][S] 0026	UC3.5.1.3
R[V][M] 0027	IW Studio di fattibilità
R[V][M] 0028	IW Studio di fattibilità
R[V][M] 0029	IW Studio di fattibilità
R[V][M] 0030	IW Studio di fattibilità
R[V][M] 0031	IW Studio di fattibilità
R[Q][S] 0032	-
R[Q][S] 0033	-
R[Q][S] 0034	-
R[Q][S] 0035	-
R[Q][C] 0036	IW Studio di fattibilità

Tabella 4.5: Tabella del tracciamento delle fonti con i requisiti

Fonte	Codice
-------	--------

UC1	R[F][C]0001
UC1.1	R[F][C]0001
UC1.1.2	R[F][C]0002
UC1.1.2	R[F][C]0003
UC2	R[F][M]0005, R[F][M]0006, R[F][M]0007, R[F][M]0008
UC2.1	R[F][M]0005
UC2.2	R[F][M]0006
UC2.3	R[F][M]0007
UC2.4	R[F][M]0008
UC2.1.1	R[F][M]0009
UC2.1.2	R[F][M]0010
UC2.1.3	R[F][M]0011
UC2.2.1	R[F][M]0012
UC2.2.2	R[F][M] 0013
UC3	R[F][M] 0014, R[F][M] 0015, R[F][M] 0016, R[F][M] 0017, R[F][M] 0018, R[F][M] 0019
UC3.1	R[F][M] 0014
UC3.2	R[F][M] 0015
UC3.3	R[F][M] 0016
UC3.4	R[F][M] 0017
UC3.5	R[F][M] 0018
UC3.6	R[F][M] 0019
UC3.4.1	R[F][M] 0020
UC3.4.2	R[F][M] 0021
UC3.4.3	R[F][M] 0022
UC3.5.1	R[F][M] 0023
UC3.5.1.1	R[F][M] 0024
UC3.5.1.2	R[F][M] 0025
UC3.5.1.3	R[F][S] 0026
IW Studio di fattibilità	R[V][M] 0027, R[V][M] 0028, R[V][M] 0029, R[V][M] 0030, R[V][M] 0031, R[Q][C] 0036
-	R[Q][S] 0032, R[Q][S] 0033, R[Q][S] 0034, R[Q][S] 0035

4.5 Analisi dei requisiti SP

4.5.1 Casi d'uso

Per lo studio dei casi di utilizzo del prodotto sono stati creati dei diagrammi. I diagrammi dei casi d'uso (in inglese *Use Case Diagram*) sono diagrammi di tipo [UML](#) dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso.

Descrizione Attori

I tipi di utente che andranno ad interagire direttamente con il sistema si dividono in due categorie:

- * Servizio convenzionato;
- * Utente IW.

Tra gli attori precedentemente citati non è però prevista alcuna funzionalità in comune e non emerge quindi la necessità di avere una gerarchia. In immagine 4.11 è proposta una visualizzazione grafica di quanto detto. Non sono stati invece individuati attori secondari che partecipano al sistema.



Figura 4.11: Gerarchia utenti user case

Attori principali

- * Servizio convenzionato: l'attore servizio convenzionato è quello che nell'analisi del dominio è stato definito come *Real Service Provider* (RSP). Si tratta del fornitore reale del servizio.
- * Utente IW: l'attore utente IW è una persona fisica che utilizza la nostra applicazione mobile al fine di operare l'accesso ad un servizio convenzionato in MonoKee.

Attori secondari Non sono presenti attori secondari.

UC1: Azioni servizio convenzionato

Descrizione Il servizio convenzionato può reindirizzare verso al sistema una richiesta di accesso e ricevere i dati di accesso PII verificati.

Attore primario Servizio convenzionato

Attore secondario Nessuno



Figura 4.12: Use Case - UC1: Azioni servizio convenzionato

Precondizioni Il servizio convenzionato ha richiesto una richiesta di accesso e l'utente che l'ha effettuata a richiesto l'accesso tramite il nostro servizio.

Postcondizioni Il servizio ha eseguito le azioni che desiderava compiere in relazione alle sue possibilità

Scenario principale

1. UC1.1 Reindirizzamento accesso
2. UC1.2 Ricezione PII verificati

Scenari alternativi Nessuno

UC1.1: Reindirizzamento accesso

Descrizione Un servizio convenzionato può inoltrare al sistema richieste di accesso

Attore primario Servizio convenzionato

Attore secondario Nessuno

Precondizioni Il servizio convenzionato ha ricevuto una richiesta di accesso

Postcondizioni Il sistema ha ricevuto la richiesta di accesso e procederà ad eseguirla

Scenario principale Il servizio convenzionato inoltra la richiesta di accesso ed il sistema la immagazzina per prendersene carico

Scenari alternativi Nessuno

UC1.2: Ricezione PII verificate

Descrizione Il sistema deve, in risposta ad un inoltro di richiesta di accesso, inviare al servizio convenzionato l'esito della verifica e, in caso di successo, le PII in chiaro necessarie per effettuare l'oggetto

Attore primario Servizio convenzionato

Attore secondario Nessuno

Precondizioni Il servizio convenzionato ha precedentemente inoltrato una richiesta di accesso al sistema

Postcondizioni Il sistema ha ricevuto l'esito della verificata ed in caso le PII necessarie per l'accesso in chiaro

Scenario principale Il sistema riceve l'esito della verificata ed in caso le PII necessarie per l'accesso in chiaro

Scenari alternativi Nessuno

UC2: Azioni utente IW

Figura 4.13: Use Case - UC2: Azioni utente IW

Descrizione L'utente IW può eseguire le operazioni per l'accesso

Attore primario Utente IW

Attore secondario *MonoKee*

Precondizioni Nessuna

Postcondizioni L'utente ha eseguito le azioni che desiderava compiere in relazione alla condizione.

Scenario principale

1. UC2.1 Sottomissione codice QR

Scenari alternativi Nessuno

UC2.1: Sottomissione codice QR

Descrizione L'utente IW può eseguire l'operazione di sottomissione di codice QR

Attore primario Utente IW

Attore secondario Nessuno

Precondizioni Il servizio convenzionato ha inoltrato l'utente al nostro sistema di accesso e l'utente ha generato il codice QR dall'IW

Postcondizioni Il sistema ha catturato il codice QR

Scenario principale Il sistema accende la webcam del computer e cattura il codice QR che presenta l'utente.

Scenari alternativi Nessuno

4.5.2 Diagramma delle attività

Al fine di descrivere il corretto flusso che il componente deve utilizzare viene utilizzato un diagramma di attività. L'unica operazione che il componente dovrà gestire al fine di garantire gli scopi che si prefigge è la gestione di un inoltro d'accesso da parte di un RSP.

Ora si procederà ad una breve descrizione del diagramma in figura 4.14. Il flusso parte con l'arrivo di una richiesta di accesso da parte di un RSP, questo le seguenti operazioni in maniera sequenziale:

- * inoltro della richiesta verso il nostro sistema SP;
- * il sistema visualizza una schermata dove richiede la sottomissione del codice QR;
- * cattura del codice QR;
- * decodifica le PII in chiaro dal codice QR.

Poi il flusso si divide in tre operazioni differenti:

- * la prima con il compito di inviare una richiesta di verifica all'ITF per ogni PII decodificato dal codice QR;
- * la seconda con il compito di interfacciarsi al sistema MonoKee per ottenere l'associazione tra account e servizio e la lista dei PII necessari;
- * il terzo con il compito di aspettare gli esiti delle verifiche dall'ITF.

In caso l'associazione sia presente e corretta e tutte le PII necessarie sono verificate allora si procede con la comunicazione dei dati verso il reale fornitore del servizio. In caso, o si riceva un esito negativo di una PII necessaria, o non tutte quelle necessarie siano state presentate tramite il codice QR, si procede alla comunicazione dell'errore di autenticazione ed alla conclusione del flusso. In ogni caso se dopo 40 secondi dalla decodifica del codice QR il sistema non ha effettuato l'accesso viene visualizzato un messaggio di errore ed il flusso termina.

4.5.3 Tracciamento dei requisiti

Fonti

Per la deduzione dei requisiti utente e di sistema, che verranno presentati nelle sezioni a seguire, sono stati usati come fonti lo studio Gartner³, il capitolo *Studio di fattibilità SP* e gli Use Case presentati nella sezione *Casi d'uso*. La struttura e le convenzioni usate sono ispirate dal capitolo di **som:swe**. In seguito vengono riportate le categorie che vengono usate per la catalogazione:

- * F: requisito funzionale;
- * V: requisito di vincolo;
- * Q: requisito di qualità.

Per l'attribuzione della priorità viene usata la tecnica *MoSCoW*, quindi gli indici usati sono i seguenti:

- * M: must;
- * S: should;
- * C: could;
- * W: will.

Nelle tabelle 4.6, 4.8 e 4.7 sono riassunti i requisiti e il loro tracciamento con gli *use case* delineati in fase di analisi.

³farah:The-Dawn-of-Decentralized-Identity.

Tabella 4.6: Tabella del tracciamento dei requisiti funzionali

Codice	Descrizione	Fonte
R[F][M]0001	Il sistema deve permettere ad un servizio convenzionato di inoltrare le richieste di accesso ricevute al nostro sistema	UC1, UC1.1, DA1
R[F][M]0002	Il sistema deve inviare l'esito della verifica al reale fornitore del servizio	UC1, UC1.2, DA1
R[F][M]0003	Il sistema deve inviare i PII in chiaro in caso di verifica positiva al reale fornitore del servizio	UC1, UC1.2, DA1
R[F][M]0004	Il sistema deve permettere ad un utente dell'IW di sottomettere un codice QR generato dall'applicazione IW.	UC2, UC2.1, DA1
R[F][M]0005	Il sistema deve visualizzare una schermata di accesso	DA1
R[F][M]0006	Il sistema deve catturare nella schermata di accesso il codice QR attraverso l'uso della webcam	DA1
R[F][M]0007	Il sistema deve essere in grado di decodificare le informazioni contenute in un codice QR	DA1
R[F][M]0008	Il sistema deve essere in grado di fare l' <i>hash</i> di una PII	DA1
R[F][M]0009	Il sistema deve essere in grado di inviare una richiesta di verifica per un particolare PII	DA1
R[F][M]0010	Il sistema deve essere in grado di eseguire l'operazione di hash e invio richiesta verifica per ogni PII presenta in un codice QR	DA1
R[F][M]0011	Il sistema deve inviare una richiesta dell'associazione utente-servizio a <i>MonoKee</i> classico	DA1
R[F][M]0012	Il sistema deve essere in grado di ricevere le informazioni richiesta dell'associazione utente servizio da <i>MonoKee</i> classico	DA1
R[F][M]0013	Il sistema deve visualizzare un messaggio di errore in caso cui l'associazione utente-servizio non sia presente per il servizio richiesto	DA1
R[F][M]0014	Il sistema deve essere in grado di ricevere l'esito della verifica di un singolo PII proveniente dall'ITF	DA1
R[F][M]0015	Il sistema deve visualizzare un messaggio di errore in caso cui la verifica di una PII richiesta sia negativa	DA1
R[F][M]0016	Il sistema deve visualizzare un messaggio di errore in caso non tutte le verifiche delle PII necessarie tornino in 40 secondi.	DA1
R[F][M]0017	Il sistema deve in caso di presenza dell'associazione e del ritorno positivo di tutte le verifiche necessarie inviare i dati PII al SP reale	DA1

Tabella 4.7: Tabella del tracciamento dei requisiti di vincolo

Codice	Descrizione	Fonte
R[V][M] 0018	Il sistema deve offrire le proprie funzionalità come applicazione server centralizzata	SP Studio di fattibilità
B	Il sistema è implementato tramite in linguaggi .NET	SP Studio di fattibilità

B	Il progetto prevede almeno i seguenti quattro ambienti di sviluppo: <i>Local</i> , <i>Test</i> , <i>Staging</i> , <i>Production</i>	SP Studio di fattibilità
B	Il prodotto è sviluppato utilizzando uno strumento di <i>linting</i>	SP Studio di fattibilità
B	Il sistema deve comunicare con la rete <i>blockchain</i> tramite un <i>client Ethereum</i> .	SP Studio di fattibilità

Tabella 4.8: Tabella del tracciamento dei requisiti qualitativi

Codice	Descrizione	Fonte
R[Q][S] 0023	Il progetto prevede un ragionevole set di test di unità e di test di integrazione	-
R[Q][S] 0024	I test possono essere eseguiti localmente o come parte di integrazione continua	-
R[Q][S] 0025	Il sistema solo alla fine sarà testato nel server di prova	-
R[Q][S] 0026	Il codice sorgente del prodotto e la documentazione necessaria per l'utilizzo sono versionati in repository pubblici usando GitHub, BitBucket o GitLab	-
R[Q][C] 0027	Lo sviluppo si eseguirà utilizzando un approccio incrementale	SP Studio di fattibilità
R[Q][C] 0028	Il sistema potrebbe essere testato con l'ITF migrato nella rete di prova Ropsten	ITF Studio tecnologico

Tabella 4.9: Tabella del tracciamento dei requisiti con le fonti

Codice	Fonte
R[F][M]0001	UC1, UC1.1, DA1
R[F][M]0002	UC1, UC1.1, DA1
R[F][M]0003	UC1, UC1.1, DA1
R[F][M]0004	UC2, UC2.1, DA1
R[F][M]0005	DA1
R[F][M]0006	DA1
R[F][M]0007	DA1
R[F][M]0008	DA1
R[F][M]0009	DA1
R[F][M]0010	DA1
R[F][M]0011	DA1
R[F][M]0012	DA1
R[F][M]0013	DA1
R[F][M]0014	DA1
R[F][M]0015	DA1
R[F][M]0016	DA1
R[F][M]0017	DA1
R[V][M] 0018	SP Studio di fattibilità

R[V][M] 0019	SP Studio di fattibilità
R[V][M] 0020	SP Studio di fattibilità
R[V][M] 0021	SP Studio di fattibilità
R[V][C] 0022	SP Studio di fattibilità
R[Q][S] 0023	-
R[Q][S] 0024	-
R[Q][S] 0025	-
R[Q][S] 0026	-
R[Q][C] 0027	SP Studio di fattibilità
R[Q][C] 0028	ITF Studio tecnologico

Tabella 4.10: Tabella del tracciamento dei fonte con requisiti

Fonte	Requisiti
UC1	R[F][M]0001 R[F][M]0002 R[F][M]0003
UC2	R[F][M]0004
UC1.1	R[F][M]0001
UC1.2	R[F][M]0002 R[F][M]0003
UC2.1	R[F][M]0004
DA1	R[F][M]0001 R[F][M]0002 R[F][M]0003 R[F][M]0004 R[F][M]0005 R[F][M]0006 R[F][M]0007 R[F][M]0008 R[F][M]0009 R[F][M]0010 R[F][M]0011 R[F][M]0012 R[F][M]0013 R[F][M]0014 R[F][M]0015 R[F][M]0016 R[F][M]0017

SP Studio di fattibilità	R[V M]0018 R[V M]0019 R[V M]0020 R[V M]0021 R[V M]0022 R[Q C]0027
-	R[Q S]0023 R[Q S]0024 R[Q S]0025 R[Q S]0026
ITF Studio tecnologico	R[Q C]0028

4.6 Riepilogo requisiti

4.6.1 Riepilogo requisiti IW

In tabella 4.11 vengono riportati la quantità dei requisiti individuati per l'IW suddivisi per tipo e per priorità.

Tabella 4.11: Riepilogo requisiti IW

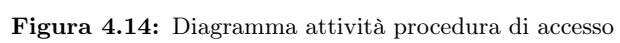
Categoria	Must	Should	Could	Will
Funzionale	21	1	4	0
Di vincolo	5	0	0	0
Di qualità	0	4	1	0

4.6.2 Riepilogo requisiti SP

In tabella 4.12 vengono riportati la quantità dei requisiti individuati per l'SP suddivisi per tipo e per priorità.

Tabella 4.12: Riepilogo requisiti SP

Categoria	Must	Should	Could	Will
Funzionale	17	0	0	0
Di vincolo	4	0	1	0
Di qualità	0	4	2	0



Capitolo 5

Progettazione e codifica

Breve introduzione al capitolo

Il presente capitolo ha lo scopo di presentare e dimostrare l'architettura per i componenti IW e SP che dovranno funzionare nel contesto dell'estensione del prodotto Monokee.

5.1 Componente Identity Wallet

Questa sezione inizia con una generica introduzione all'architettura *Xamarin* ed si conclude presentando una prima ipotesi di architettura in formato UML 2.0.

5.1.1 Tecnologie e strumenti

Il componente *Identity Wallet* è sviluppato come applicazione mobile. Questo contesto implica differenti tecnologie che comunicano e interagiscono fra loro. Le funzionalità di persistenza vengono offerte tramite tre diverse tecnologie: *file system*, *blockchain*, e server *Monokee*. La logica di *business* è implementata usando il *framework* .NET. L'interfaccia, invece, usa il pattern MVVM (*Model View View Model*). L'IW utilizza una classica architettura a strati (*N-tier architecture*). Trattandosi di un sistema mobile multi piattaforma questa architettura è stata calata nel contesto e quindi si è deciso di basarla sul concetto di *Portable Class Libraries* (PCL) presentato da *Xamarin*.

Portable Class Libraries PCL

[Portable Class Libraries](#)^[g] è un approccio alla condivisione del codice tra le diverse edizioni dell'app destinate a diversi sistemi operativi mobili sviluppati da *Xamarin*. Segue un diagramma esplicativo di come si sviluppa una tipica architettura PCL. Il diagramma in figura 5.1 è tratto da www.xamarin.com.

Ogni “*Platform-Specific Application Project*” (*iOS app*, *Android app*, *Windows Phone app*) referencia la [Portable Class Libraries](#). Esistono essenzialmente due parti: quelle specifiche per la piattaforma e quelle condivise. Obiettivo del progetto è quello di rendere meno corpose possibile le parti specifiche. Sarà poi possibile impiegare caratteristiche di una determinata piattaforma attraverso l'utilizzo del design pattern *Dependency Injection* (DI). Applicare i principi della DI significa definire nel codice condiviso interfacce (classi astratte) che vengono implementate (estese) in ogni piattaforma tramite sottoclassi (*Strategy Pattern*). A questo punto sarà possibile integrare le

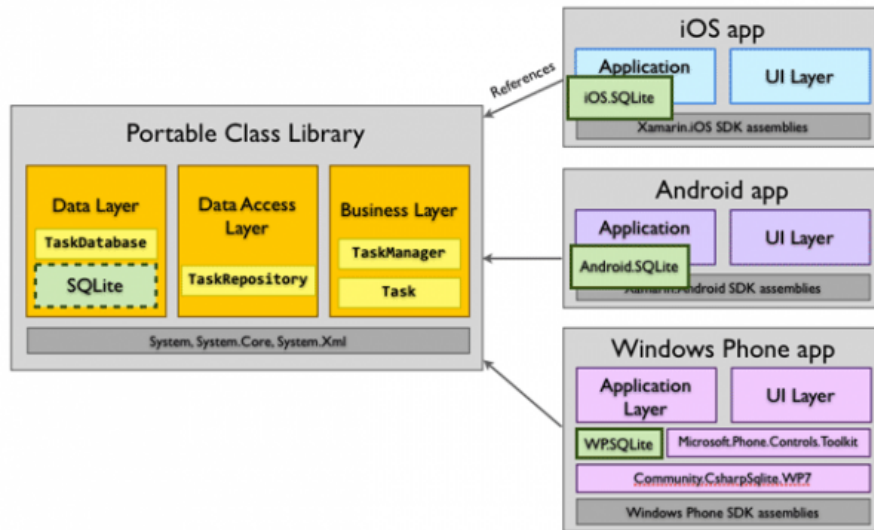


Figura 5.1: Architettura PCL

specifiche implementazioni all'interno della PCL. *Xamarin* per questo scopo offre la classe *DependencyService*.

5.1.2 Overview

Come già detto, l'applicativo è strutturato come una *N-tier application* consistente dei seguenti *layer*:

- * layer presentazione;
- * logica di business;
- * layer di accesso ai dati.

Quando si sviluppa un'applicazione è importante scegliere se sviluppare un *thin Web-based client* o un *rich client*. Ovviamente, considerando il nostro contesto, ricadiamo nel primo caso, infatti quasi tutta la logica e la persistenza rientrano sul componente ITF. In figura 5.2 un'immagine esplicativa dell'architettura ideata. Come si può notare il principale pattern utilizzato per gestire l'interazione con l'utente è il *Model View ViewModel* (MVVM). Tutte le elaborazioni vengono effettuate dallo strato di *business*, mentre per la persistenza ci si affida principalmente alla risorsa *Monokee* tramite comunicazione REST, oppure all'ITF tramite l'utilizzo di un client *Ethereum*. Tutto verrà sviluppato utilizzando il *framework* .NET.

5.1.3 Progettazione

In figura 5.3 viene presentato il diagramma di massima dell'architettura dell'IW. Il diagramma è stato redatto seguendo lo standard *UML 2.0*. Subito a seguire viene descritta ogni classe.

**Figura 5.2:** Architettura PCWL

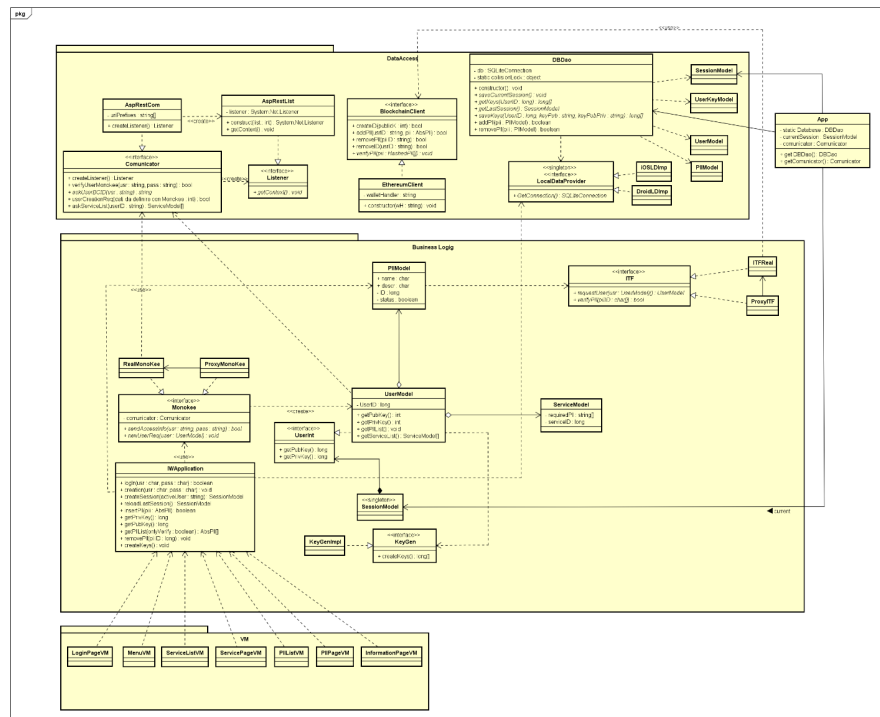


Figura 5.3: Architettura IW

BusinessLogic

Nel contesto dell'architettura *N-tier* adottata, il *BusinessLogic layer* è un gruppo di classi che si occupa di effettuare e di mantenere tutte le regole definite dai documenti IW - Analisi di massima, IW - Studio di fattibilità.

IWApplication: classe che ha il compito di fornire una *facade* per i vari *ViewModel*, di conseguenza tutte le azioni possibili tramite l'interfaccia sono implementate da essa.

Monokee: interfaccia che ha il compito di fornire un'astrazione del servizio *Monokee*. L'interfaccia con *RealMonokee* e *ProxyMonokee* rappresenta un'applicazione del pattern *Proxy*.

RealMonokee: classe che rappresenta il reale oggetto *Monokee* e che dialoga poi con *RESTComp* per ottenere i dati. Questa classe con *RealMonokee* e *ProxyMonokee* rappresenta un'applicazione del pattern *Proxy*.

ProxyMonokee: classe che rappresenta un *proxy* dell'oggetto *Monokee*, questa classe applica una politica di acquisizione pigra. Questa classe con *RealMonokee* e *ProxyMonokee* rappresenta un'applicazione del pattern *Proxy*.

KeyGen: interfaccia che ha lo scopo di definire una strategia di generazione chiavi. Fa parte di un'applicazione dello *Strategy Pattern* ed è stata pensata in un'ottica in cui vi possano essere vari modi per generare una chiave a seconda del sistema operativo usato.

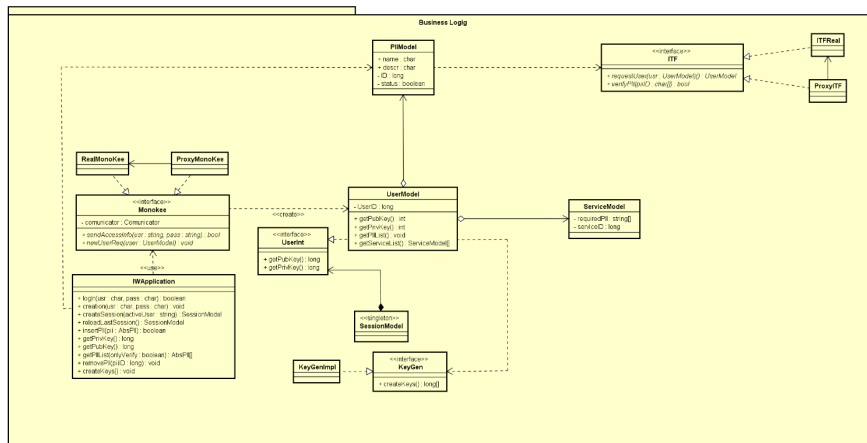


Figura 5.4: Diagramma BusinessLogic Layer IW

KeyGenImpl: classe che rappresenta una possibile implementazione dell'interfaccia *KeyGen*. Fa parte di un'applicazione dello *Strategy pattern*.

Session: classe con lo scopo di immagazzinare tutti i dati di una sessione attiva; questa può essere generata dal *file system* o creata da zero. Deve essere presente in istanza singola e inoltre contiene le informazioni utente.

UserInt: interfaccia che rappresenta un qualsiasi utente dell'applicazione. È implementata solamente da *UserMonokee*. Questo oggetto viene creato dall'interfaccia *Monokee*.

UserMonokee: classe che rappresenta un utente proveniente dal server *Monokee*. Implementa l'interfaccia *UserInt*. Un utente di questo tipo possiede un aggregato di servizi e una lista PII; potenzialmente può contenere le chiavi.

Service: classe che rappresenta un servizio di cui l'utente ha diritto. Possiede un ID e fornisce una lista di PII che dovranno essere presentati al fine di eseguire l'accesso.

KeyProv: classe che ha il compito di occuparsi della generazione delle chiavi private e pubbliche. Viene usata da *UserInt* e a sua volta usa *LocalDataProvider*.

LocalDataProvider: interfaccia che ha il compito di fornire in singolo punto dove ottenere informazione dal *file system* locale. Successivamente deve essere implementata in base al sistema operativo su cui girerà.

iOSLDImp: rappresenta l'implementazione per *iOS* di *LocalDataProvider*.

DroidLDImp: rappresenta l'implementazione *Android* di *LocalDataProvider*.

AbsPII: interfaccia che rappresenta una generica PII. Ha una sola possibile implementazione, ma può rendere più semplice l'implementazione di future PII.

PIIImpl: classe che rappresenta l'attuale ed unica PII. Consiste di un nome, un identificativo e una descrizione. Una PII può essere verificata o meno tramite l'uso di *PIIChecker*.

ITF: interfaccia che ha il compito di fornire un’astrazione del componente *Identity Trust Fabric*. Con *ITFReal* e *ProxyITF* rappresenta un’applicazione del pattern *Proxy*.

RealITF: classe che rappresenta il reale oggetto ITF. Essa dialoga poi con il *BlockchainClient* per ottenere i dati. Con *RealITF* e *ProxyITF* rappresenta un'applicazione del pattern *Proxy*.

ProxyITF: classe che rappresenta un *proxy* dell'oggetto *Monokee*. Applica una politica di acquisizione pigra. Con *RealITF* e *ProxyITF* rappresenta un'applicazione del pattern *Proxy*.

PIIChecker: classe che ha il compito di verificare tramite ITF la veridicità di una PII.

DataAccess Layer

Nel contesto dell'architettura *N-tier* adottata il *DataAccess layer* è un gruppo di classi che si occupano di interfacciarsi con gli strumenti di persistenza utilizzati dall'applicazione. Questi sono: *Monokee* (tramite RESTful), ITF e una base di dati locale al dispositivo.

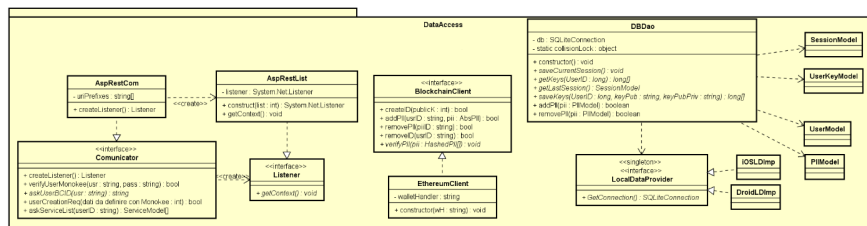


Figura 5.5: Diagramma DataAccess Layer IW

RestComp: interfaccia che ha il compito di rappresentare una generica strategia di comunicazione REST. Essa viene utilizzata da *RealMonokee* per ottenere i dati relativi all'utente.

RestImpl: possibile implementazione della strategia di comunicazione REST. Implementa l'interfaccia *RestComp*.

BlockchainClient: interfaccia che ha il compito di rappresentare una generica strategia di comunicazione con la rete *blockchain*. Questa astrazione permette di slegare dall'architettura dipendenze con le varie implementazioni di *blockchain* e anche di *client*.

EthereumClient: possibile implementazione di *BlockchainClient* che utilizza la rete *Ethereum*. Questa classe poi userà la libreria *Nethereum*.

PresentationLayer

Questo *layer* contiene i vari controllori che gestiscono le viste. Si è previsto di creare un controller per ogni pagina dell'applicazione. L'applicazione utilizza il pattern MVVM, quindi, i controllori sono delle *ViewModel* (VM) che contengono i dati e le operazioni. Tra i dati e la vista sono presenti dei *binding* da realizzare utilizzando gli strumenti forniti da *Xamarin*. Le VM operano le loro azioni tramite l'utilizzo della classe *IWFacade*.

Tutte queste classi devono estendere da *INotifyPropertyChanged*. In figura 5.6 il diagramma esplicativo del *layer*.

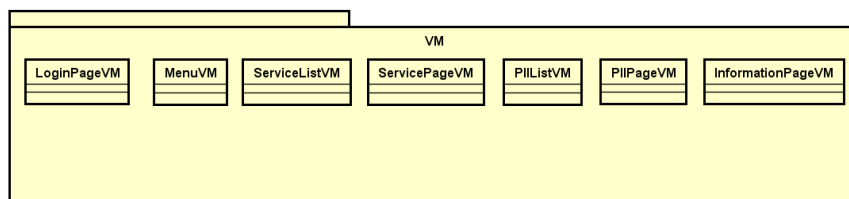


Figura 5.6: Diagramma UML VM Layer

LoginPageVM: classe che ha lo scopo di gestire la pagina di *log in* e quindi avere lo stato e le operazioni necessarie.

MenuVM: classe che ha lo scopo di gestire il menù dell'applicazione e quindi avere lo stato e le operazioni necessarie.

ServiceListVM: classe che ha lo scopo di gestire la pagina che presenta la lista dei service a cui può accedere l'utente e quindi avere lo stato e le operazioni necessarie.

ServicePage: classe che ha lo scopo di gestire la pagina con le informazioni relative ad un singolo servizio e quindi avere lo stato e le operazioni necessarie.

PIIListVM: classe che ha lo scopo di gestire la pagina che presenta la lista delle PII che possiede l'utente e quindi avere lo stato e le operazioni necessarie.

PIIPageVM: classe con lo scopo di gestire la pagina che visualizza le informazioni relative ad una specifica PII e quindi avere lo stato e le operazioni necessarie.

InformationPageVM: classe che ha lo scopo di gestire la pagina che fornisce le informazioni sull'applicazione, sul servizio *Monokee* e le istruzioni per l'uso.

5.1.4 Design Pattern utilizzati

Al fine di garantire elevate doti di qualità e manutenibilità dell'architettura sono stati usati una serie di design pattern. Di seguito segue una breve descrizione di questi.

Communicator : incapsula i dettagli interni della comunicazione in un componente separato che poi può essere implementato usando tecnologie diverse; risultato utile per rendere gli altri componenti quanto più indipendenti da come comunicano con l'esterno.

Data Transfer Object (DTO) : oggetto che ha il compito di racchiudere le informazioni utili a diverse componenti. Va a ridurre i metodi necessari per la comunicazione e, in generale, la semplifica.

Entity Translator : oggetto che trasforma un dato in forma utile per essere usato nella logica di *business*. Esso è stato usato per interfacciarsi con il *client Ethereum* e il *server Monokee*.

Lazy Acquisition Proxy : Ritarda l'acquisizione delle risorse il più a lungo possibile. Esso è stato ampiamente utilizzato, specialmente per rendere più leggera possibile la creazione dei dati dell'utente e della verifica dei dati nell'ITF.

Strategy Pattern : oggetto che permette di separare l'esecuzione di un metodo dalla classe che lo contiene. Usando un'interfaccia per astrarre il metodo è poi possibile creare molteplici implementazioni. Ciò è risultato molto utile nel contesto di un'applicazione multi piattaforma in cui alcune procedure dovevano essere implementate in nativo. Oltretutto ha reso possibile separare il metodo dall'implementazione.

Dependency Injection : pattern che permette di delegare il controllo della creazione oggetti ad un oggetto esterno. Esso permette di semplificare la gestione delle dipendenze e, nel contesto dello *strategy pattern*, permette di inserire l'implementazione corretta.

Model-View-Controller : separa il codice per l'interfaccia grafica in tre componenti separati: Modello (il dato), Vista (l'interfaccia), e Controllore (il responsabile della logica), con particolare attenzione alla vista. Nel progetto viene usata una sua particolare declinazione chiamata MVVM.

5.2 Componente Service Provider

La sezione inizia con una generica introduzione alle architetture *Event Driven*. Si è deciso di utilizzare un approccio *Broken topology*; la scelta è motivata dalla maggior indipendenza tra i vari componenti rispetto ad un approccio *Mediator topology*. Infine si conclude con la presentazione una prima ipotesi di architettura in formato UML 2.0.

5.2.1 Tecnologie e strumenti

Il componente *Service Provider* è sviluppato come applicazione server, questo implica possibili accessi multipli al servizio da parte di vari *Real Service Provider* (RSP) che inoltrano le loro richieste di accesso. L'applicativo fa uso di diverse fonti per espletare le proprie funzioni. Più dettagliatamente queste sono: *Monokee*, RSP e ITF. Da questo primo studio architetturale non sembrerebbe necessario l'uso di una base di dati locale. Considerato quanto appena detto si è ritenuta particolarmente adatta un'architettura *Event Driven* basata sull'utilizzo di code. Per la comunicazione con il RSP e con *Monokee* si è deciso di utilizzare un approccio basato sulle API RESTful. Invece per la comunicazione verso l'ITF si è deciso di utilizzare un client *Ethereum*.

Architettura Event Driven

Questa tipologia di architettura rappresenta uno dei principali esempi di pattern architettura asincrono. Produce applicati altamente scalabili e facilmente adattabili ad ogni carico di utilizzo. Se applicata bene fornisce la possibilità di avere eventi con un singolo scopo ([single responsibility principle](#)^[8]) e con un basso livello di accoppiamento. Questo è reso possibile dalla gestione asincrona di questi eventi. Ci sono due possibili approcci a questa architettura:

- * Mediator topology;
- * Broker topology.

Mediator topology

Un evento generalmente possiede una serie di passi ordinati per essere eseguito. In questa approccio ci sono quattro componenti che interagiscono fra loro:

- * una o più code di eventi;
- * un mediatore di eventi;
- * uno o più esecutori di eventi;
- * dei canali di eventi.

Gli eventi possono essere di due tipi:

- * eventi iniziali;
- * eventi di processamento.

In figura 5.7 si riporta una generica architettura *Event Driven Mediator Topology*.

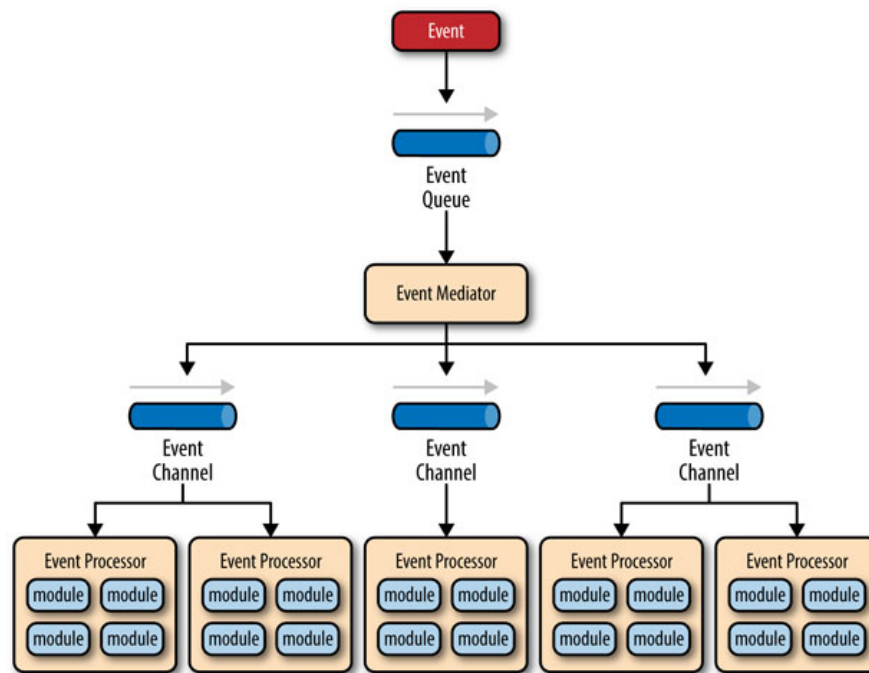


Figura 5.7: Schema Mediator Topology

Mediatore di eventi Il mediatore (l'*Event Mediator*) ha il compito di orchestrare i passi necessari per rispondere ad un evento iniziale; per ogni passo invia uno specifico evento di processamento ad un canale (*Event Channel*). Il mediatore non applica nessun tipo di logica, conosce solo i passi necessari per gestire l'evento iniziale e quindi li genera.

Canale di eventi Si tratta generalmente di un canale di comunicazione asincrono. Questo può essere di due tipi:

- * coda di messaggi;
- * topic di messaggi.

Esecutore di eventi Contiene la vera logica di business per processare ogni evento. Sono auto contenuti, indipendenti e scarsamente accoppiati.

Broker topology

In questo approccio non è presente un mediatore centrale. Il flusso dei messaggi viene distribuito dai vari esecutori, creando una catena di eventi che generano a loro volta altri eventi. Risulta molto utile nel caso in cui il flusso sia molto semplice.

In questo approccio ci sono due principali componenti:

- * un *broker* che contiene tutti i canali;
- * vari esecutori di eventi.

In figura 5.8 si riporta una generica architettura *Event Driven Broker Topology*.

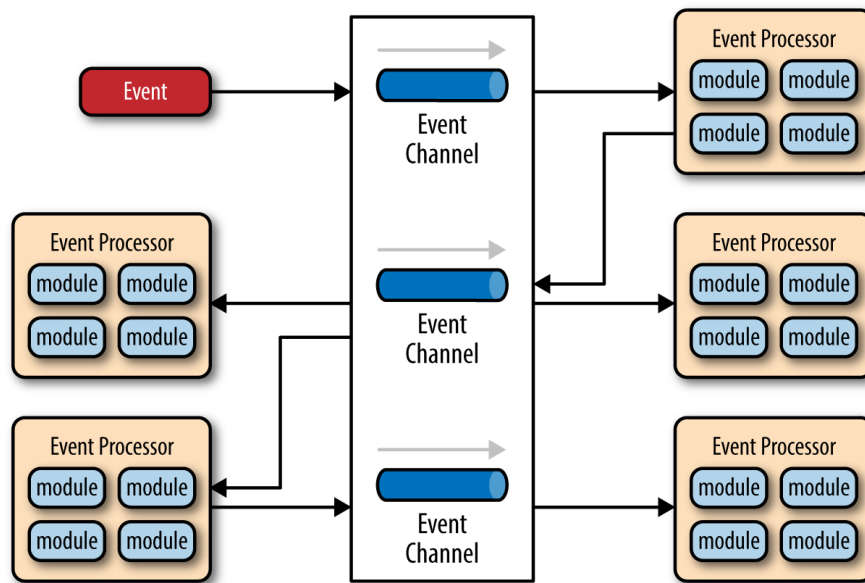


Figura 5.8: Schema Broker Topology

Considerazioni

Di seguito si evidenziano alcune vantaggi e svantaggi in maniera analitica ¹:

Agilità generale I cambiamenti sono generalmente isolati e possono essere fatti velocemente con piccoli impatti.

Facilità di deploy È dovuta all'alto disaccoppiamento degli esecutori. Questa nota vale particolarmente per la tipologia *Broker* in quanto non presenta il mediatore.

Testabilità Richiede strumenti specializzati per generare eventi, questo potrebbe rendere i test di sistema difficili. I test di unità invece sono facilmente implementabili.

Scalabilità La natura indipendente dei componenti rende facile scalare questi in base alle necessità permettendo così un *tuning* delle risorse molto fine.

Facilità di sviluppo È il principale svantaggio di queste architetture.

Uno dei principali svantaggi di questo tipo di architettura è la complessità di implementazione, dovuta al fatto che operazioni sono completamente asincrone e concorrenti. Si è comunque ritenuta questa architettura nella sua variante *Broker Topology* adatta allo scopo soprattutto per questioni di *performance*, scalabilità e facilità di *deploy*.

¹site:event-driven

5.2.2 Overview

Come già detto l'applicazione sarà strutturata con una architettura *Event Driven* di tipo *Broker topology*, questo implica che la logica di funzionamento sia incapsulata nei vari passaggi tra le varie code. Gli esecutori sono i seguenti cinque:

- * **Starter**: con il compito di ascoltare gli eventi iniziali dei vari RSP e di ricevere i vari dati ottenuti tramite codici QR;
- * **RetriveInfo**: con il compito di ottenere le informazioni necessarie da *Monokee*;
- * **PageResponce**: con il compito di generare e visualizzare le pagine nel browser dell'utente, sia di fallimento che di comunicazione;
- * **PiiDataHandler**: con il compito di verificare i dati nell'ITF e verificare che questi siano sufficienti per effettuare l'accesso;
- * **RSPSendingWork**: con il compito di inviare al RSP le informazioni di accesso.

Gli eventi sono i seguenti:

- * **AccessRequest**: generato dallo *Starter* e eseguito dal *RetriveInfo*;
- * **PageResponce**: generato dal *RetriveInfo* in caso di errore o per mostrare il lettore QR, dal *PiiDataHandler* in caso di *login* o in caso di insuccesso della verifica;
- * **VerificationWork**: generato dallo *Starter* per verificare i dati forniti tramite il QR e quelli forniti da *RequireInfo* siano conformi e verificati;
- * **RSPSendingWork**: generato da *PiiDataHandler* in caso di verifica positiva.

Il diagramma in figura 5.9 rappresenta come i vari eventi di lavoro si distribuiscono tra i vari esecutori.

Lo *Starter* quando riceve una richiesta d'accesso da parte del RSP procede a generare il lavoro di *AccessRequest*, una volta ricavati tutti i dati necessari per l'accesso da *Monokee*, viene affidato al *PageResponce* l'incarico di visualizzare la pagina che richiede l'inserimento del QR. I dati verranno poi inseriti dall'utente e attraverso lo *Starter* verrà creato un lavoro di verifica dei dati inseriti e se questi sono sufficienti ad accedere al servizio, tramite un'ulteriore accesso a *Monokee*. In caso di esito positivo viene creato un lavoro di invio dati verso il RSP altrimenti verrà visualizzata una pagina di errore.

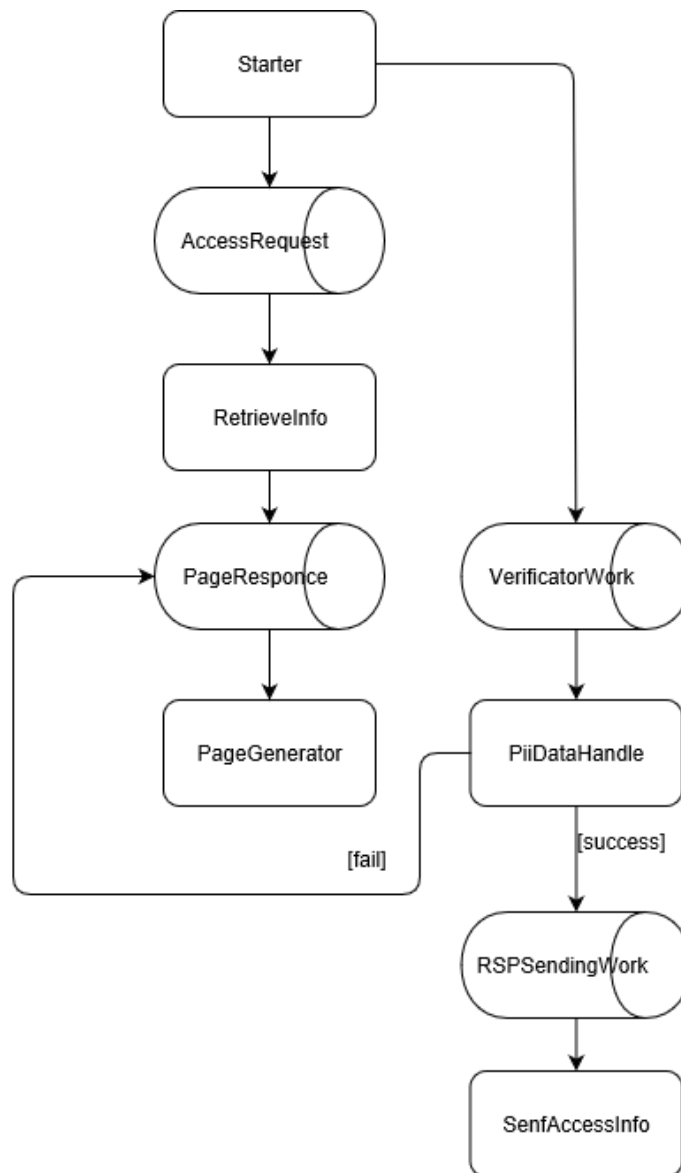


Figura 5.9: Flusso eventi SP

5.2.3 Progettazione

In figura 5.10 si presenta un diagramma delle classi che attua la gestione delle code sopra espletata. Il diagramma è stato redatto in formato *UML 2.0*, con leggere modifiche relativo alla rappresentazione delle varie istanze del template *CommandQueue*. Questo è stato fatto al fine di rendere più leggibile e comprensibile il diagramma. Come si può notare sono presenti componenti non presenti nella precedente trattazione. Questi servono per effettuare le comunicazioni con l'ambiente esterno. Si è deciso per questioni di semplicità di non creare code separate.

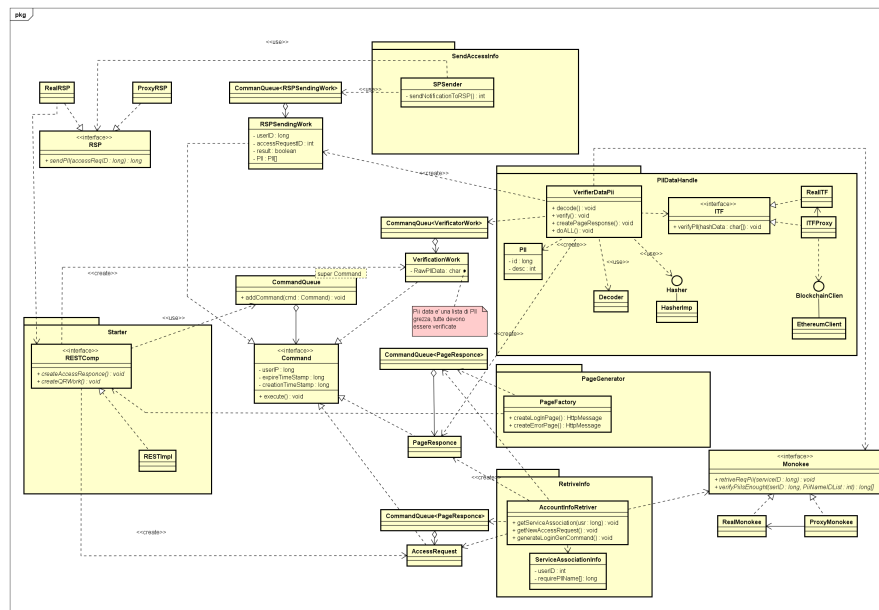


Figura 5.10: Diagramma delle classi del modulo SP

Starter

Lo *Starter* rappresenta l'esecutore iniziale. Questo rimane in ascolto di eventuali richieste di accesso inoltrate dagli *RSP* e da inizio all'esecuzione di questa. Riceve inoltre i dati provenienti dai codici QR. In figura 5.11 viene riportato il diagramma UML dell'esecutore. Le classi più significative sono di seguito descritte.

Reicever: è un'interfaccia con il compito di ricevere le comunicazioni provenienti dal sito di login e, quindi, inoltrare i messaggi alla coda successiva (*RetrieveInfo*). Rappresenta il *driver* del componente *Starter*.

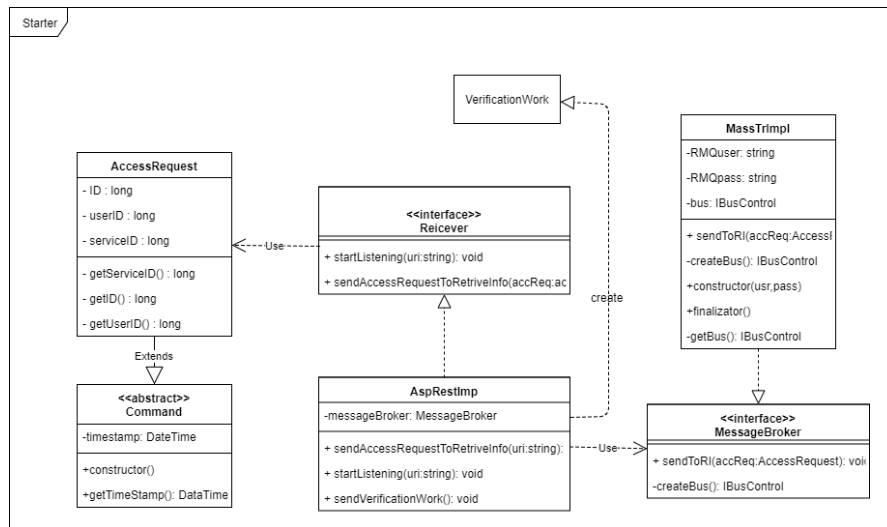
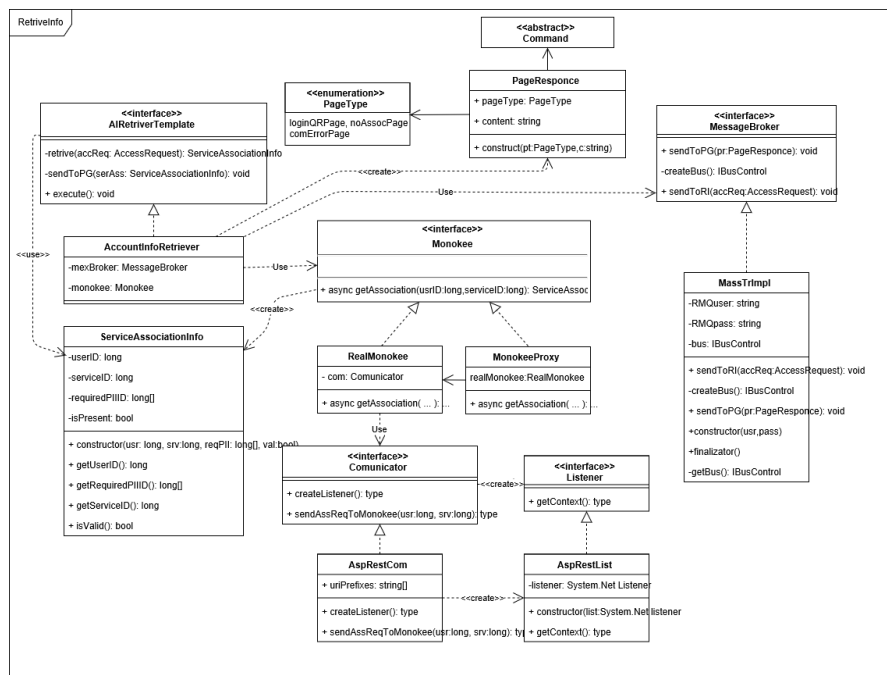
AspRestImpl: è una possibile implementazione della strategia di comunicazione tramite *REST*. Implementa l'interfaccia *Reicever*.

MessageBroker: è un'interfaccia con il compito di definire i metodi necessari alla gestione dalla rete di comunicazione *RabbitMQ*.

MassTransitImpl: questa classe implementa il *MessageBroker* utilizzando la libreria *MassTransit*.

RetriveInfo

È l'esecutore con il compito di ottenere le informazioni necessarie da *Monokee*. In caso di associazione presente manda il lavoro di creazione pagina di login, in caso di associazione non presente manda il lavoro di creazione pagina di errore. In figura 5.12 viene riportato il diagramma UML dell'esecutore. Le classi più significative sono di seguito descritte.

Figura 5.11: Rappresentazione UML di *Starter*Figura 5.12: Rappresentazione UML di *RetriveInfo*

AIRetrriverTemplate: questa interfaccia ha lo scopo di ricevere gestire e inoltrare i messaggi alla coda successiva (*RetrieveInfo*). Rappresenta un template (*Template pattern*) del *driver* dell'esecutore *RetriveInfo*.

AccountInfoRetriever: implementazione di *AIRetrriverTemplate*.

ServiceAssociationInfo: questa classe rappresenta le informazioni ottenute da *Monokee*, riporta i dati della *AccessRequest* che l'ha generata e ci aggiunge l'informazione relativa alla lista delle *PII* richieste e la presenta o meno della associazione in *Monokee*. È un oggetto immutabile.

PageType: Questo *enumeration* definisce le varie possibili pagine creabili dall'esecutore *PageGenerator*. I possibili tipi sono:

- * *loginQRPage*: è una pagina mostra cattura il codice QR con le informazioni necessarie;
- * *noAssocPage*: è una pagina che comunica che l'utente che ha effettuato una richiesta per un servizio a cui non è abilitato (non possiede l'associazione in *Monokee*);
- * *comErrorPage*: è una pagina che comunica un generico errore di comunicazione.

Monokee: si tratta di un'interfaccia con il compito di fornire un'astrazione del servizio *Monokee*. Questa interfaccia con *RealMonokee* e *ProxyMonokee* rappresenta un'applicazione del pattern *Proxy*.

RealMonokee: è una classe che rappresenta il reale oggetto *Monokee*, questa classe poi dialoga con *RESTComp* per ottenere i dati. Questa classe con *RealMonokee* e *ProxyMonokee* rappresenta un'applicazione del pattern *Proxy*.

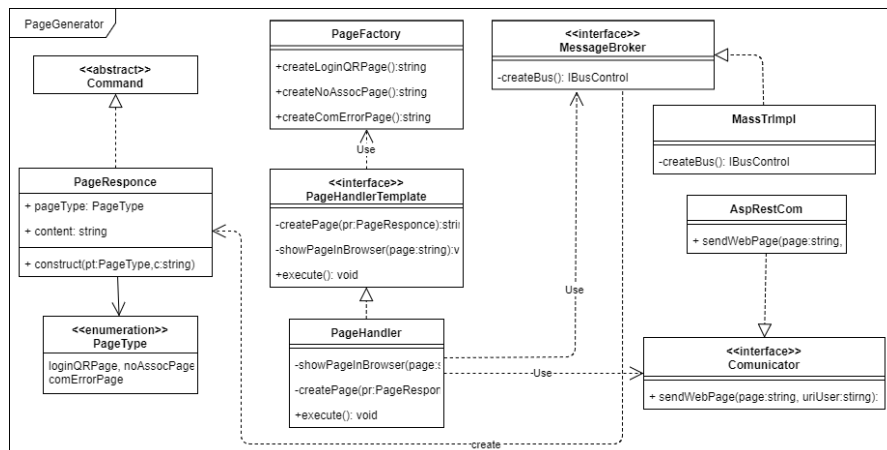
ProxyMonokee: è una classe che rappresenta un *proxy* dell'oggetto *Monokee*, questa classe applica una politica di acquisizione pigra. Questa classe con *RealMonokee* e *ProxyMonokee* rappresenta un'applicazione del pattern *Proxy*.

Communicator: questa classe fornisce un'interfaccia per gestire tutte le informazioni attraverso fonti esterne. Deve essere usata per comunicare con *Monokee* e il *Real Service Provider*.

AspRestCom: questa classe implementa *Communicator* tramite l'utilizzo della libreria *Asp.NET*.

Listener: È un'interfaccia con il compito di rimanere in ascolto su determinati *uri*. È un oggetto non mutabile.

AspRestList: È un oggetto con il compito di rimanere in ascolto su determinati *uri*. È un oggetto non mutabile. Questa classe è un *wrapper* del *listener* di *System.Net*. Implementa *Listener*.

Figura 5.13: Rappresentazione UML di *PageGenerator*

PageGenerator

È l'esecutore con il compito di ascoltare le richieste di creazione pagina provenienti dagli altri esecutori e quindi di generarle come richiesto e di inviarle al *browser* del richiedente. In figura 5.13 viene riportato il diagramma UML dell'esecutore. Le classi più significative sono di seguito descritte.

PageHandlerTemplate: questa interfaccia consiste in un *template* per la creazione di un *driver* per la gestione e l'esecuzione dei *Command* di *PageGenerator*.

PageHandler: Questa classe implementa la struttura fornita da *PageHandlerTemplate*.

PIIDataHandler

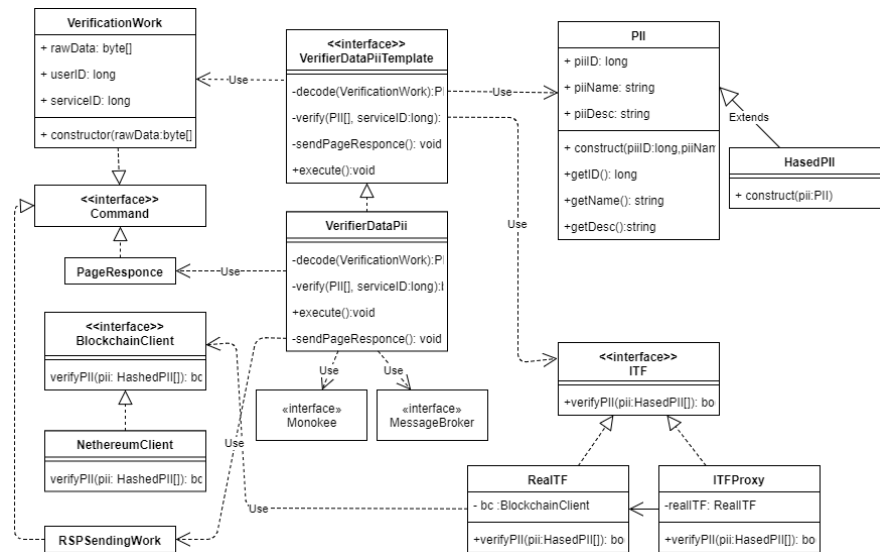
Questo esecutore ha il compito di ricevere nella propria coda dallo *Starter* una serie di lavori di verifica e quindi di verificare questi sia verso *Monokee* sia verso l'*ITF*. In questo esecutore sono presenti le interfacce *Command*, *Monokee* e *MessageBroker* che non verranno presentate. In figura ?? viene riportato il diagramma UML dell'esecutore. Le classi più significative sono di seguito descritte.

VerificationDataPiiTemplate: questa interfaccia rappresenta il *template* da seguire per eseguire l'algoritmo di gestione di un *VerificationPiiWork*. Dalla ricezione fino al suo esaurimento.

VerificationDataPii: Questa classe rappresenta un'applicazione del *template* da seguire per eseguire l'algoritmo di gestione di un *VerificationPiiWork*. Dalla ricezione fino al suo esaurimento.

Pii: è una classe che rappresenta una PII. Contiene l'id, la descrizione di una PII.

HashedPii: questa classe rappresenta una PII con le informazioni *piiName* e *piiDesc* sotto forma di *Hash*, questa classe dev'essere usata per comunicare con l'*ITF*. L'oggetto è immutabile.

Figura 5.14: Rappresentazione UML di *PIIDataHandler*

ITF: si tratta di un'interfaccia con il compito di fornire un'astrazione del componente ITF. Questa interfaccia con *RealITF* e *ProxyITF* rappresenta un'applicazione del pattern *Proxy*.

RealITF: è una classe che rappresenta il reale oggetto ITF, questa classe poi dialoga con il *BlockchainClient* per ottenere i dati. Questa classe con *RealITF* e *ProxyITF* rappresenta un'applicazione del pattern *Proxy*.

ITFProxy: è una classe che rappresenta un *proxy* dell'oggetto ITF, questa classe applica una politica di acquisizione remota. Questa classe con *RealITF* e *ITFProxy* rappresenta un'applicazione del pattern *Proxy*.

BlockchainClient: questa interfaccia ha il compito di rappresentare un canale di comunicazione verso gli *SmartContract*. Deve essere atea rispetto alla tipologia di *blockchain* usata.

NethereumClient: Questa classe rappresentare un canale di comunicazione verso gli *SmartContract* di una rete *Ethereum*. Fa uso della libreria *.NET Nethereum* per instaurare la comunicazione.

SendAccessInfo

Questo esecutore ha il compito di inviare le informazioni necessarie per effettuare il *login* al *RSP* ed al *back-end* di *Monokee*. Queste informazione deve essere fornite in forma non di *hash*. I *Command* provengono da *PIIDataHandle*. In figura ?? viene riportato il diagramma UML dell'esecutore. Le classi più significative sono di seguito descritte.

RSPSenderTemplate: questa interfaccia rappresenta il *template* da seguire per eseguire il lavoro dell'esecutore. In altre parole rappresenta l'algoritmo da eseguire per espletare il lavoro di invio attributi al *RSP*.

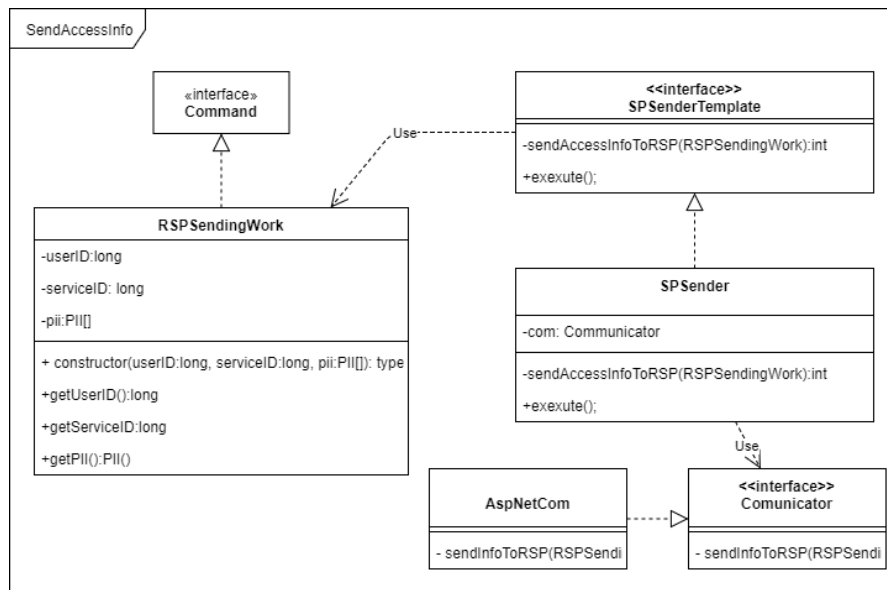


Figura 5.15: Rappresentazione UML di *SendAccessInfo*

RSPSender: questa classe implementa il *template* da seguire per eseguire il lavoro dell'esecutore. In altre parole rappresenta l'algoritmo da eseguire per espletare il lavoro di invio attributi al *RSP*.

CommonElement Questo componente contiene una serie di elementi usati da più esecutori, primi fra tutti gli oggetti *Command*. Pertanto si è deciso di creare una libreria condivisa tra i vari esecutori. Le classi più significative sono di seguito descritte.

Command: È una classe che rappresenta un generico evento nel contesto dell'architettura *event driven*. Questa interfaccia viene poi implementata da:

- * **AccessRequest:** generato dallo *Starter* e eseguito dal *RetriveInfo*, rappresenta il lavoro per gestire la richiesta di accesso;
- * **PageResponse:** generato dal *RetriveInfo* in caso di errore o per mostrare il lettore QR, dal *PiiDataHandler* in caso di *login* o in caso di insuccesso della verifica. Rappresenta il lavoro di generazione e sottomissione delle pagine all'utente;
- * **VerificationWork:** generato dallo *Starter* per verificare i dati forniti tramite il QR e quelli forniti da *Monokee* siano conformi e verificati;
- * **RSPSendingWork:** generato da *PiiDataHandler* in caso di verifica positiva. Rappresenta il lavoro di sottomissione dati in caso di verifica positiva.

Hasher: È un'interfaccia che ha il compito di eseguire l'*hash* di un dato. Rappresenta un'applicazione dello *strategy pattern*.

HasherImpl: È una classe che implementa un'implementazione della classe *Hasher*. Esegue l'*hash* di un dato. Rappresenta con *Hasher* un'applicazione dello *strategy pattern*.

5.2.4 Design Pattern utilizzati

Al fine di garantire elevate doti di qualità e manutenibilità dell'architettura sono stati usati una serie di design pattern. Di seguito segue una breve descrizione di questi.

Command Pattern permette di isolare la porzione di codice che effettua un'azione (eventualmente molto complessa) dal codice che ne richiede l'esecuzione; l'azione è incapsulata nell'oggetto *Command*.

Remote Proxy fornisce una rappresentazione locale di un oggetto remoto remote.

Strategy Pattern è un oggetto che permette di separare l'esecuzione di un metodo dalla classe che lo contiene. Usando un'interfaccia per astrarre il metodo è poi possibile crearne molteplici implementazioni. Questo è risultato molto utile nel contesto di un'applicazione multi piattaforma in cui alcune procedure andavano implementate in nativo. Oltre all'appena citato vantaggio questo ha reso possibile separare il metodo dall'implementazione.

Dependency Injection è un pattern che permette di delegare il controllo della creazione oggetti ad un oggetto esterno. Questo permette di semplificare la gestione delle dipendenze e nel contesto dello *strategy pattern* permette di inoculare l'implementazione corretta.

Factory Method è un pattern che permette di convogliare tutte le funzioni di creazione di vari elementi ad un oggetto unico.

5.3 Implementazione

Le attività di implementazione e codifica sono state la parte di maggior impegno e sforzo dell'intero progetto. Hanno occupato in termini orari 120 ore, le quali rappresentano quasi il 40% della durata del progetto. Esse hanno fatto emergere diversi errori di progettazione e di codifica, inoltre ci sono stati diverse ripensamenti da parte dell'azienda che hanno portato a riprogettare interi moduli. Nei paragrafi seguenti si cercherà di presentare le implementazioni più significative all'interno del progetto.

5.3.1 Procedura di login

L'applicativo mobile non prevedeva una gestione degli account propria, ma utilizzava gli account già presenti nel sistema *Monokee*. Questo ha necessitato quindi la codifica di un'apposita procedura di *login* in modo di autenticare un utente in base ad una coppia di valori *email* e *password*.

La suddetta procedura consiste principalmente in due chiamate HTTPS, la prima con lo scopo di ottenere l'id dell'utente tramite la mail fornita, la seconda allo scopo di verificare se la *password* sia conforme all'id ottenuto dalla prima chiamata.

La prima fase era attuata da un metodo chiamato **Task<string> RetriveUserID(string usr)**. Questo metodo inviava tramite una POST il seguente *json*, con il quale forniva le informazioni necessarie ad ottenere l'id :

Listing 5.1: Json della prima request di login

```
{
    email = "user@dominio.com",
    mobile = "false",
}
```

La risposta doveva invece seguire il seguente schema:

Listing 5.2: Json risposta user_id

```
{
    success = "true",
    message = "ok",
    user_id = "df3c92kzz1",
}
```

La risposta in alcuni casi poteva presentare altri campi dati, ma questi venivano ignorati dall'applicativo. Risposte che non presentavano le precedentemente citate tre informazioni o che non riportavano i valori "true" ed "ok" rispettivamente su "success" e "message" causavano il lancio di un'eccezione che faceva comparire un messaggio di fallimento a schermo.

Una volta ottenuto il valore "user_id" si procedeva alla verifica della password col metodo **getTokenID(string userId, string password)**. Questo avveniva sempre tramite l'uso di una comunicazione POST. Nella request veniva mandato il seguente *json*:

Listing 5.3: Json richiesta token

```
{
    user_id = "df3c92kzz1",
    password = "Passw0rd",
    mobile = "false",
    persistence = "false",
    cc_key = "key",
    salt = "salt",
    domain_id = "5b475bd150e783334b5bb861",
}
```

Il campo "persistence" è utile alla gestione interna del dato e non rientra in nessun modo nel progetto. I parametri "cc_keys" e "salt" invece sono due valori tipici delle procedure di *login*. Questi servono a rendere più complessi gli attacchi a dizionario verso il sistema, infatti i due valori per essere generati richiedono un elevato onere computazionale sia in termini di spazio che di tempo, in quanto devono rispettare delle determinate condizioni. La codifica del codice per generare questi due valori ha richiesto innumerevoli sforzi e si è rivelata essere non banale. L'applicativo *Monokee* ha la caratteristica di offrire ai propri utenti domini separati, un tipico utilizzo di questi è per esempio il dominio aziendale e quello personale. Il "domain_id" fornito da rappresenta il dominio personale. L'applicativo per ragioni di semplicità faceva l'accesso sempre allo stesso dominio.

La risposta doveva seguire il seguente schema:

Listing 5.4: Json risposta token

```
{
    success = "true",
    message = "ok",
    token = "qrdj99d7f9da0b2nf93Kd9LL"
}
```

Il significato di "success" e "message" rimane analogo a quello precedentemente descritto, mentre il valore "token" rappresenta un codice da inserire nell'*header* sotto il nome di "Authorization" nelle successive chiamate in modo tale da essere risposto. Questo codice ha validità di 24 ore, al seguito delle quali scade ed è necessario rieffettuare la procedura di *login*. Risulta estremamente importante al momento del *logout* rimuovere dall'*header* il *token*, in quando una successiva operazione di *login* ne avrebbe aggiunto un altro causando un doppio *token*. Questo avrebbe fatto fallire le successive richieste, seppur consentendo l'accesso.

Per ragioni progettuali interne a *Monokee* le chiamate se ricevute vengono sempre risposte con uno *status code* 200, a prescindere del fatto che queste siano accettate o rifiutate. Per questa ragione al fine di sostituire i vari codici di errore vengono usati i campi "success" e "message".

La codifica di questa procedura non ha presentato particolari difficoltà se non quella della generazione del *cc_key* e del *salt*.

5.3.2 Uso del database SQLite

Nel contesto dell'applicazione mobile per effettuare la persistenza dei dati si è deciso di utilizzare una base di dati relazionale. La scelta è ricaduta su *SQLite*. Si è deciso di operare con questa base di dati utilizzando il design pattern *data access object* e apposite librerie di sistema che lo implementavano, questa pratica si è rivelata essere molto pratica ed efficace ai fini del progetto. Ovviamente risulta limitata comparata rispetto all'uso del codice *sql*, ma le necessità applicative non richiedevano simili finezze.

A titolo di esempio si discute ora della memorizzazione di un PII.

Listing 5.5: codice creazione DAO

```
public DBDao()
{
    database = DependencyService
        .Get<ILocalDataProvider>()
        .GetConnection();
    database.CreateTable<PIIModel>();
    database.CreateTable<UserModel>();
}
```

Il codice appena presentato rappresenta la creazione dell'oggetto che fornisce l'accesso alla base di dati, come si può notare il corpo del costruttore crea una connessione tramite l'uso di *DependencyService*, successivamente crea le tabelle. Ovviamente in caso le tabelle siano già presenti, le istruzioni non alterano la base di dati.

A differenza di quanto uno si potrebbe aspettare, la funzione *CreateTable* non richiede informazione sui tipi di dato e/o sulle colonne da creare. Queste informazioni vengono dedotte dal tipo che gli viene fornito, il quale andrà a rappresentare il modello di una tupla della tabella.

Si propone adesso il codice di *PIIModel*:

Listing 5.6: codice PIIModel

```
[Table("PIIModel")]
public class PIIModel
{

    [PrimaryKey, AutoIncrement]
    public int Id;

    [Indexed(Name = "nameId", Order = 1, Unique = true)]
    public string UserID;

    [Indexed(Name = "nameId", Order = 2, Unique = true)]
    public string Name;
    [NotNull]

    //continua ...
}
```

I campi dati pubblici vengono considerati come i valori delle colonne, mentre altre informazioni necessarie vengono fornite tra parentesi quadre. Queste sono per esempio le chiavi, i valori non nulli, i vincoli di integrità, il nome della tabella etc...

Sequendo questa tecnica la gestione del database risulta particolarmente semplice. Vengono ora riportati degli esempi di codice per l'inserimento e la rimozione di una PII dalla base di dati:

Listing 5.7: codice aggiunta e rimozione PIIModel

```
public int AddPII(PIIModel pii)
{
    lock (collisionLock)
    {
        if (pii.Id != 0)
        {
            database.Update(pii);
            return pii.Id;
        }
        else return database.Insert(pii);
    }
}

public int DeletePII(int piiID)
{
    lock(collisionLock)
    {
        return database.Delete<PIIModel>(piiID);
    }
}
```

Ovviamente interrogazioni più complicate risultano non essere possibile tramite funzioni di libreria, per queste particolari occasioni si è dovuto utilizzare del codice *sql*.

Listing 5.8: Esempio di query sql

```
public List<PIIModel> GetPIIs(string name, string user\_id)
{
    //query sql
}
```

```

        lock (collisionLock)
        {
            return (from i in database.Table<PIIModel>()
                    where (i.Name == name && i.UserID==user\_id)
                    select i).ToList();
        }
    }
}

```

L'uso di queste tecniche e dei modelli ha permesso di limitare al massimo l'utilizzo di codice *sql*, inoltre ha permesso una più facile gestione del dato in quanto la risposta ad una *query* veniva già presentata in forma di oggetto.

5.3.3 Implementazione del databinding

L'applicazione mobile richiedeva la creazione di molteplici schermate che dovevano sempre rimanenere aggiornare rispetto ad un particolare oggetto e viceversa. Questo risultava particolarmente importante in quanto era fondamentale che l'interfaccia fosse ben separata dalla logica di *business*, affinché una modifica nella logica o nel modello di dominio non si rifletta sull'interfaccia.

A questi scopi *Xamarin* offre la cosiddetta tecnica del *databinding* tra interfaccia e oggetto. Ora si ripropone l'oggetto *PIIModel*, questa volta però non omettendo il codice necessario per il *databinding*.

Listing 5.9: esempio di oggetto in databinding

```

[Table("PIIModel")]
public class PIIModel : INotifyPropertyChanged
{
    private int piiID;
    [PrimaryKey, AutoIncrement]
    public int Id
    {
        get { return piiID; }
        set {
            piiID = value;
            OnPropertyChanged(nameof(Id));
        }
    }

    \\altro codice ommesso ...

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this,
            new PropertyChangedEventArgs(propertyName));
    }
}

```

Come si può notare per poter effettuare il *databinding* è necessario che la classe *PIIModel* implementi l'interfaccia *INotifyPropertyChanged*, che definisce l'evento *PropertyChanged* di tipo *PropertyChangedEventHandler*. Questo evento prende un'istanza della classe *PropertyChangedEventArgs* che definisce la proprietà *PropertyName* di tipo *string*, attraverso la quale è possibile sapere quale proprietà nell'oggetto *PIIModel* è cambiata (permettendo all'evento di accedere a quella proprietà).

Attraverso la proprietà pubblica *Id*, offriamo la possibilità lato codice di ottenere le informazioni sul valore corrente attraverso il *get*, e di impostare un nuovo valore per tale proprietà attraverso il *set* ogni qualvolta il valore assunto dalla variabile *name* differisce da quella corrente. Proprio in quest'ultima porzione viene richiamato il metodo *OnPropertyChanged*, che prenderà in ingresso il nome della proprietà che deve essere aggiornata innescando il meccanismo sopra descritto.

Per effettuare l'effettiva collegamento basta inserire nel codice che gestisce la schermata la seguente istruzione:

Listing 5.10: codice di connessione

```
BindingContext = PIIModel;
```

Questo renderà possibile nel codice grafico (XAML) di poter utilizzare i valori dell'oggetto. Di seguito viene mostrato un esempio:

Listing 5.11: esempio di vista che usa il databinding

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/
                  xaml"
              x:Class="IdentityWallet.PIIVisualizerPage"
              Title="{Binding Name}">
  <ContentPage.Content>
    <ScrollView>
      <StackLayout VerticalOptions="StartAndExpand" Padding
                  ="20">
        <Label Text="Id"/>
        <Label x:Name="piiID" Text="{Binding Id}"/>

        <Label Text="Name" />
        <Entry x:Name="piiName" Text="{Binding Name}"/>

        <- altro codice ->
      </StackLayout>
    </ScrollView>
  </ContentPage.Content>
</ContentPage>
```

5.3.4 Instaurazione della rete di messaggi

Il modulo SP è composto da diversi componenti, i quali sono residenti in programmi differenti e separati fra loro. L'unico modo di comunicazione che utilizzano è un [broker di messagistica](#)^[8] di nome *RabbitMQ*. Questo deve essere eseguito su un server e verrà utilizzato dai vari componenti tramite il suo *uri*. L'applicativo richiede che nel server sia installato *Erlang*. Una volta averlo installato si può procedere all'installazione di *RabbitMQ* seguendo la guida presente al seguente url <https://www.rabbitmq.com/download.html>. Per eseguire correttamente le funzionalità del modulo SP è inoltre necessario creare una rete col nome di "test"; per fare questo è necessario abilitare l'interfaccia di configurazione di *RabbitMQ* seguendo la seguente guida <https://www.rabbitmq.com/management.html>. Una volta abilitata l'interfaccia questa sarà disponibile al seguente indirizzo <http://localhost:15672/#/connections>.

La pagina che si presenterà avrà una scheda chiamata "network" dalla quale sarà possibile aggiungere la rete di nome "test". Fatto questa la configurazione di *RabbitMQ* è terminata; ora basta avviare il *broker* tramite terminale usando il comando *rabbitmq-server start*.

5.3.5 Gestione delle code e dei messaggi

Come già discusso il modulo SP ha un'architettura del tipo *even driven*. L'applicazione di tale libreria ha necessitato dell'uso di una libreria per la gestione e l'invio dei vari messaggi.

Il modulo SP è diviso in molteplici esecutori ognuno del quale presenta del codice per la gestione dei messaggi equivalente, a titolo di esempio si procede ad esporre l'esecutore *ITFVerifier*.

All'avvio l'esecutore ha il compito di creare un oggetto in grado di individuare la rete che gestisce i messaggi (questa era disponibile all'uri definito in *GeneralSetting.RabbitMQHost*) e quindi effettuare il collegamento ad essa tramite la sottomissione di una coppia *username, password*. Successivamente era necessario definire cosa doveva essere fatto dei messaggi in arrivo e soprattutto quale tipo di messaggi dovessero essere ascoltati. La parte finale del codice proposto mostra le istruzioni necessarie a fare quanto appena detto. Tramite *ReceiveEndpoint* viene stabilito il nome della coda dove verranno inseriti i messaggi catturati e il comportamento da intraprendere per ognuno di questi messaggi. Come comportarsi viene definito tramite una *arrow function* che istanzia un *Consumer* con un tipo da noi definito (*VerificationWorkConsumer*). I messaggi da inserire nella coda di ricezione non vengono decisi in base ad un'indicazione del mandante, ma tramite il tipo dell'oggetto che viene implementato da *VerificationWorkConsumer*. L'ultima riga ha lo scopo di eseguire il lavoro in modalità asincrona.

Listing 5.12: creazione di un collegamento a *RabbitMQ*

```
_busControl = Bus.Factory.CreateUsingRabbitMq(x =>
{
    IRabbitMqHost host = x.Host(new Uri(GeneralSetting.
        RabbitMQHost), h =>
    {
        h.Username("guest");
        h.Password("guest");
    });

    x.ReceiveEndpoint(host, "VerQueue",
        e => { e.Consumer<VerificationWorkConsumer>(); })
        ;

});

TaskUtil.Await(() => _busControl.StartAsync());
```

Adesso risulta utile analizzare il codice del *Consumer*. Come si può notare la classe implementa *IVerificationWork*, pertanto nella coda che porta il nome di "VerQueue" verranno inseriti solo i messaggi che hanno come tipo *IVerificationWork*. Unico metodo implementabile è *Consume* il quale rappresenta il codice da effettuare.

Listing 5.13: esempio di *Consumer*

```
public class VerificationWorkConsumer : IConsumer<
    IVerificationWork>
{
    public async Task Consume(ConsumeContext<IVerificationWork>
        context)
    {
        //operazioni da effettuare
    }
}
```

Andiamo ora a vedere come si invia un messaggio un *VerificationWork*:

Listing 5.14: codice invio di un messaggio

```
public async Task SendToVerificationAsync(IVerificationWork
    verWork)
{
    await _busControl.Publish(verWork);
}
```

Come potete vedere questo avviene tramite la funzione *Publish* dell'oggetto *_busControl*.

5.3.6 Interazioni con la *blockchain*

Sia l'applicativo mobile IW, che l'applicativo server SP hanno previsto l'uso della stessa libreria *Nethereum* al fine di effettuare le necessarie chiamate alla *blockchain Ethereum*. Questa libreria implementa il protocollo Ethereum² in ambiente *.NET*.

Il protocollo discerne tra due tipi di chiamate, quelle in **sola lettura** e quella in **scrittura**; le prime sono identificate dai modificatori *pure* o *view*.

Le chiamate in sola lettura sono immediate e non richiedono l'esecuzione di una transazione, per queste ragioni hanno prestazione paragonabile a quelle di un linguaggio tradizionale e non richiedono la vengano fornite dall'*ether*.

Le seconde invece, dovendo modificare la *blockchain*, hanno bisogno di effettuare una transazione con la conseguente esecuzione dell'algoritmo di consenso descritto nel capitolo 3.2.3. Questo porta ai problemi di prestazioni e costo descritti in 3.2.4

Prima di poter procedere a qualsiasi operazione era necessario creare un collegamento alla rete *Ethereum* usata (nel nostro caso una rete locale all'azienda). Il seguente codice mostra le istruzioni necessarie.

Listing 5.15: Connessione alla rete di test

```
account = new Account(GeneralSetting.accountPrivKey);
web3 = new Web3(account, blockchain_address);
```

La variabile *web3* è l'oggetto che rappresenta la connessione. Il primo parametro del costruttore non è strettamente necessario, ma stabilisce un *account* di default con il quale verranno pagate le transazioni. Si è deciso in fase di codifica di non minare direttamente le aggiunte di blocchi, ma di pagare la transazione. Per essere riconosciuti dalla rete è sufficiente fornire la chiave privata.

²site:ethereum-yellow-paper.

Ora è necessario richiamare un contratto presente nella rete. Per fare questo necessitiamo di due informazioni: l'**indirizzo** e l'**application binary interface**^[8]. L'*ABI* di un contratto consiste in un *json* contenente la definizione di tutti i metodi presenti nel contratto. Nel frammento 5.16 si mostra l'*ABI* della chiamata che andremo ad effettuare, invece nel frammento 5.17 si mostra la creazione dell'oggetto contratto. Dal contratto poi si ottiene la funzione.

Listing 5.16: Esempio di ABI

```
[
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": true,
        "name": "_PII_ID",
        "type": "string"
      },
      {
        "indexed": false,
        "name": "result",
        "type": "bool"
      }
    ],
    "name": "eventResponse",
    "type": "event"
  },
  {
    "constant": false,
    "inputs": [
      {
        "name": "_publicKey",
        "type": "string"
      },
      {
        "name": "_PII_ID",
        "type": "string"
      },
      {
        "name": "_name",
        "type": "string"
      },
      {
        "name": "_description",
        "type": "string"
      }
    ],
    "name": "verify",
    "outputs": [
      {
        "name": "",
        "type": "bool"
      }
    ],
    "payable": false,
  }
]
```



```

        "stateMutability": "nonpayable",
        "type": "function"
    }
]

```

Listing 5.17: Creazione del contratto

```

SPFContract = web3.Eth.GetContract(abi,
    "contract_address");
var verifyFunction = SPFContract.GetFunction("verify");

```

Una volta ottenuto l'oggetto *verifyFunction* non ci resta che effettuare la chiamata. Il metodo *verify* effettua modifiche, quindi è necessario effettuare una transazione. La funzione *SendTransactionAndWaitForReceiptAsync* effettua la chiamata del metodo *Solidity verify* e aspetta che questa fallisca o abbia successo. Una transazione non restituisce mai il risultato della chiamata, ma una ricevuta che ne attesta la riuscita o il fallimento. Per ottenere il risultato bisogna usare i cosiddetti *eventi* che vanno lanciati dal codice *Solidity* con il contenuto del valore di ritorno. Gli eventi sono comunicati in *broadcast* a tutti gli utenti connessi alla rete, di conseguenza ottenere il risultato richiede una serie di procedure particolari. La seconda parte del codice nel frammento [5.18](#) mostra il codice necessario per filtrare gli eventi in base alla ricevuta e quindi ottenere il valore.

Listing 5.18: Creazione del contratto

```

receipt = await verifyFunction.
    SendTransactionAndWaitForReceiptAsync(
        "account_address",
        null,
        ITFID,
        description,
        publicKeyBytes)
    .ConfigureAwait(false);

//CODICE PER GESTIRE L'EVENTO DI RISPOSTA
var verifyEvent = SPFContract.GetEvent("eventResponse");
var filterOnITFID = verifyEvent.CreateFilterInput(
    new BlockParameter(receipt.BlockNumber),
    BlockParameter.CreateLatest());
var log = await verifyEvent.GetAllChanges<VerificationEvent>(
    filterOnITFID);
result = log[0].Event.Result;

```

Le chiamate a metodi *Solidity* detti *puri*, invece, sono più semplici e restituiscono direttamente il risultato. Di seguito un esempio di una chiamata di questo tipo.

Listing 5.19: Esempio di chiamata a metodo puro

```

getFunction = SPFContract.GetFunction("getFromID");
string result = await getFunction.CallAsync(par1, par2);

```

5.3.7 Procedura di accesso a servizio

Di seguito viene esposte le varie azione che vengono scatenate durante una tipica operazione di accesso ad un servizio. Il progetto è utilizzabile con tutti i servizi compatibili con lo standard [Security Assertion Markup Language](#).

Si propone come esempio l'applicativo *Salesforce*.

Tramite l'interfaccia di *Salesforce* abbiamo creato un apposito dominio di accesso. In questo modo l'utente potrà in fase di *login* specificare il dominio con il quale effettuare l'accesso. Le immagini 5.16 e 5.17 mostrano come indicare il dominio.

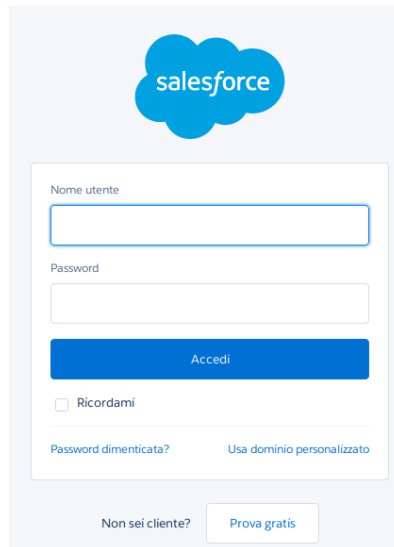
The image shows the Salesforce login interface. At the top is the Salesforce logo. Below it is a form with two input fields: 'Nome utente' (Username) and 'Password'. A blue 'Accedi' (Login) button is positioned below the password field. Underneath the button is a checkbox labeled 'Ricordami' (Remember me). At the bottom of the form are two links: 'Password dimenticata?' (Forgot password?) and 'Usa dominio personalizzato' (Use custom domain). Below the form, there are two more links: 'Non sei cliente?' (Not a customer?) and 'Prova gratis' (Try for free).

Figura 5.16: Schermata di login di Salesforce

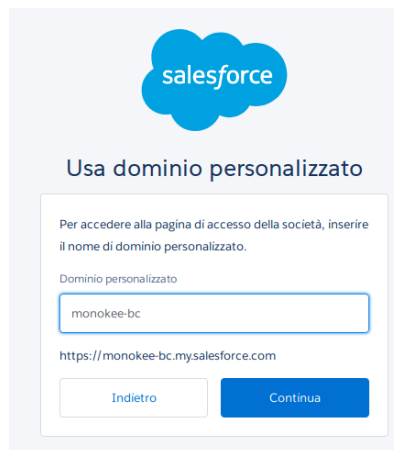
The image shows the 'Usa dominio personalizzato' (Use custom domain) page in Salesforce. It features the Salesforce logo at the top. The main heading is 'Usa dominio personalizzato'. Below this, there is a text instruction: 'Per accedere alla pagina di accesso della società, inserire il nome di dominio personalizzato.' (To access the company's login page, enter the custom domain name). There is a text input field labeled 'Dominio personalizzato' (Custom domain) containing the text 'monokee-bc'. Below the input field, the URL 'https://monokee-bc.mysalesforce.com' is displayed. At the bottom, there are two buttons: 'Indietro' (Back) and 'Continua' (Continue).

Figura 5.17: Schermata di scelta dominio

Indicando il dominio corretto il browser procederà a reindirizzare l'utente presso un apposito sito interno al *back-end* di *Monokee*.

L'inoltro verso il sito scatenerà le seguenti operazioni:

1. collegamento verso un canale *WebSocket* gestito dal componente *ResultSender* del modulo SP;
2. attivazione della *web-cam* del computer e cattura di immagini fino all'individuazione di un codice QR compatibile con quello generato dall'applicativo mobile IW;
3. invio tramite comunicazione POST del contenuto del codice QR al modulo SP, la risposta alla POST conterrà un *nonce* da usare in seguito;
4. verifica da parte del componente SP delle informazioni di accesso fornite tramite la POST e quindi invio del risultato e del *nonce* tramite il canale *WebSocket*. Il componente SP si occuperà inoltre di comunicare l'esito anche al *back-end* di *Monokee*, il quale si occuperà della generazione della *SAMLResponse*³ corrispondente;
5. inserimento della *SAMLResponse* da parte del *back-end* di *Monokee* all'interno di una *form* presente nel sito;
6. ricezione da parte del sito dell'esito della verifica tramite il canale *WebSocket*;
7. verifica dell'uguaglianza del *nonce* ricevuto dalla POST con quello ricevuto dall'esito della verifica e verifica della positività dell'esito ;
8. in caso entrambe le verifiche abbiano riscontro positivo invio tramite POST della *SAMLResponse* al *back-end* di *Salesforce*;
9. reindirizzamento verso l'applicativo *Salesforce*. Ora si è riconosciuti come utenti.

Qualora una qualsiasi delle precedenti operazioni fallisse o desse esito negativo il sito mostrerà un messaggio di errore a schermo e richiederà la risottimissione del codice QR.

³La *SAMLResponse* consiste in *XML* contenente tutte le informazioni necessarie a *Salesforce* per permettere l'accesso. Volendo fare un esempio nel caso di un servizio che richieda *username* e *password*, l'*XML* conterrà questi due dati con l'aggiunta di una serie di firme e informazione utili a garantire la sicurezza del protocollo. Per maggiori dettagli consultare https://www.samltool.com/generic_sso_res.php.

Capitolo 6

Verifica e validazione

6.1 Verifica

Secondo lo standard ISO/IEC 12207:2008¹ la verifica è un processo di supporto che si occupa di accertarsi che l'esecuzione di un'attività non abbia introdotto errori durante il periodo in esame. Ci sono due tipi di verifica: **statica** e **dinamica**. La verifica statica è estremamente utile in quanto non richiede che il prodotto sia eseguibile. Può essere effettuata tramite due tecniche:

- * ispezione;
- * analisi a pettine.

La verifica dinamica richiede l'esecuzione del codice e può essere automatica e ripetibile tramite l'uso di apposite [suite di test](#)^[g]. I test rappresentano uno dei principali esempi di verifica dinamica. Durante l'attività di stage è stata adottata una strategia che prevedeva la creazione di test subito dopo la fase di progettazione. Ogni qual volta si prevedeva un componente, venivano redatti i test che lo riguardavano e successivamente veniva fatta la progettazione in dettaglio. Ciò permetteva di progettare, in modo immediato, ad alto livello pensando al requisito astratto, ed anche in dettaglio grazie ai test che dovevano essere completamente automatizzati ed eseguiti ad ogni commit del codice. Per permettere uno sviluppo agevole si è utilizzata una tecnica di gestione del repository detta *branch-pull*.

Essa prevedeva che ogni attività di codifica dovesse essere eseguita in un *branch* creato appositamente per lo scopo. Alla fine dell'attività si procedeva con una *pull request* verso il *branch* principale, la quale veniva accettata se, e solo se, i test passavano totalmente.

Le attività di verifica erano mirate al raggiungimento dei seguenti obiettivi:

- * rilevazione di errori di codifica;
- * rilevazione di modifiche nei requisiti;
- * rilevazione di modifiche nella progettazione;
- * individuazione dell'uso di componenti di cui non si conosce chiaramente il comportamento;

¹ISO:Systems-and-software-engineering.

- * rilevazione di integrazioni tra componenti non adatte.

Un punto critico è stato trovare un giusto quantitativo di test da produrre in modo da non superare le scadenze inerenti alle attività di codifica, per questo abbiamo deciso di produrre almeno un test per metodo, ed uno per classe. Data l'elevata difficoltà nel prevedere test di integrazione e di sistema abbiamo optato di farli solo se ci fosse stato tempo utile. L'uso di queste tecniche ha permesso di avere anche una certa libertà nelle modifiche al codice (già integrato nel sistema) in quanto ha fornito, almeno in parte, anche dei test di regressione.

Lo svolgimento del processo ha permesso di ottenere varie metriche quali:

- * test coverage;
- * percentuale di test passati;
- * copertura dei requisiti.

6.1.1 Attività di verifica statica

I componenti, basandosi sullo stesso linguaggio, hanno condiviso le procedure per la verifica statica. Il principale strumento di utilizzato è stato il *linter SonarLint*. Questo strumento ha permesso di mantenere lo stesso stile di scrittura in ogni parte del progetto e di individuare potenziali errori logici e cattive pratiche. Sono state anche utilizzate numerose funzionalità presenti in *Visul Studio* che hanno permesso *refactoring* automatici del codice e strumenti di analisi statica.

6.1.2 Realizzazione dei test

Entrambi i componenti codificati durante le ore di stage condividevano lo stesso linguaggio di programmazione, ma utilizzavano [framework](#) diversi. Nonostante molte librerie fossero in comune questo non lo era per i *framework* di test. I test per il componente SP sono stati codificati usando *MSTest*, mentre quelli per il componente IW hanno usato *Xamarin.UITest*.

Altra particolare difficoltà è stata riscontrata nei test di integrazione del componente SP perché lavora in stretto contatto con l'applicativo *Monokee* e con i componenti IW e ITF. Data l'elevata mutabilità dell'applicativo *Monokee* si è deciso di realizzare doppie versioni di ogni test; la prima prevedendo l'uso di *mock*, la seconda effettuando una reale comunicazione con i componenti precedentemente citati. Ciò ha permesso una più semplice localizzazione dei problemi.

Il componente IW è un'applicazione mobile le cui interazioni con elementi esterni sono di natura occasionale e con il fine unico di ottenere informazioni verificabili, motivo per cui si è scelto di non usare dei *mock* ma di prevedere direttamente test che comunicassero con gli elementi esterni.

6.2 Validazione

La validazione si occupa di accertarsi che il prodotto sviluppato sia quello realmente desiderato. In genere è fatta a prodotto finito ed è utile al fine di capire se il prodotto soddisfa il cliente e gli utenti finali. La principale attività di validazione è stata svolta durante l'ultima settimana di lavoro attraverso prove e dimostrazioni del prodotto e

si è verificato, con la presenza del tutor aziendale, la corretta implementazione dei requisiti dedotti durante le prime fasi di analisi.

L'esito, nonostante non avesse la totalità dei requisiti soddisfatti, è stato soddisfacente. Oltretutto è da sottolineare che molti requisiti sono stati ritenuti di importanza accessoria dall'azienda e quindi sostituiti con altre funzionalità non previste inizialmente.

6.2.1 Validazione requisiti componente IW

In tabella 6.1 si mostra lo stato di validazione di ogni requisiti del componente IW, i quali possono essere:

- * implementati: se il requisito è stato implementato correttamente e riconosciuto come tale dal tutor aziendale;
- * non implementato: se il requisito non è stato inserito nel progetto o non funziona come da aspettative;
- * cancellato: se il requisito risultava non più di interesse o non più compatibile con il progetto da parte del tutor aziendali.

Tabella 6.1: Tabella validazione IW

Codice	Stato
R[F][C]0001	non implementato
R[F][C]0002	non implementato
R[F][C]0003	non implementato
R[F][C]0004	non implementato
R[F][M]0005	annullato
R[F][M]0006	implementato
R[F][M]0007	annullato
R[F][M]0008	implementato
R[F][M]0009	annullato
R[F][M]0010	annullato
R[F][M]0011	annullato
R[F][M]0012	implementato
R[F][M] 0013	implementato
R[F][M] 0014	implementato
R[F][M] 0015	implementato
R[F][M] 0016	implementato
R[F][M] 0017	implementato
R[F][M] 0018	implementato
R[F][M] 0019	implementato
R[F][M] 0020	implementato
R[F][M] 0021	implementato
R[F][M] 0022	implementato
R[F][M] 0023	implementato
R[F][M] 0024	implementato
R[F][M] 0025	implementato
R[F][S] 0026	non implementato

R[V][M] 0027	implementato
R[V][M] 0028	implementato
R[V][M] 0029	implementato
R[V][M] 0030	implementato
R[V][M] 0031	implementato
R[Q][S] 0032	implementato
R[Q][S] 0033	implementato
R[Q][S] 0034	implementato
R[Q][S] 0035	implementato
R[Q][C] 0036	implementato

I requisiti:

- * R[F][C]001;
- * R[F][C]002;
- * R[F][C]003;
- * R[F][C]004

non sono stati implementati perchè pensati in un'ottica di distribuzione al grande pubblico dell'applicazione e perciò posticipabili a successivi rilasci.

I requisiti:

- * R[F][M]0005;
- * R[F][M]0007;
- * R[F][M]0009;
- * R[F][M]0010;
- * R[F][M]0011

sono stati cancellati in quanto le funzionalità che proponevano non dovevano essere offerte dall'applicazione ma dal portale attualmente esistente di *Monokee*.

Il requisito R[F][S] 0026 non è stato implementato in quanto prevedeva una continua comunicazione con l'ITF e l'uso di notifiche *push*. Lo sforzo sarebbe stato eccessivo rispetto al valore che avrebbe apportato la funzionalità, motivo per cui si è deciso di rimandare lo sviluppo a successive versioni dell'applicativo.

6.2.2 Validazione requisiti componente SP

In tabella 6.2 mostra lo stato di validazione di ogni requisito del componente SP.I requisiti possono essere:

- * implementati: se il requisito è stato implementato correttamente e riconosciuto come tale dal tutor aziendale;
- * non implementato: se il requisito non è stato inserito nel progetto o non funziona come aspettato;
- * cancellato: se il requisito è stato ritenuto non più di interesse o non più compatibile con il progetto da parte del tutor aziendali.

Tabella 6.2: Tabella di validazione SP

Codice	Stato
R[F][M]0001	implementato
R[F][M]0002	implementato
R[F][M]0003	implementato
R[F][M]0004	implementato
R[F][M]0005	implementato
R[F][M]0006	implementato
R[F][M]0007	implementato
R[F][M]0008	implementato
R[F][M]0009	implementato
R[F][M]0010	implementato
R[F][M]0011	implementato
R[F][M]0012	implementato
R[F][M]0013	implementato
R[F][M]0014	implementato
R[F][M]0015	implementato
R[F][M]0016	implementato
R[F][M]0017	implementato
R[V][M] 0018	implementato
R[V][M] 0019	implementato
R[V][M] 0020	implementato
R[V][M] 0021	implementato
R[V][C] 0022	implementato
R[Q][S] 0023	implementato
R[Q][S] 0024	implementato
R[Q][S] 0025	implementato
R[Q][S] 0026	implementato
R[Q][C] 0027	implementato
R[Q][C] 0028	implementato

I requisiti relativi al componente SP sono stati complementamente implementati e validati dal tutor aziendale.

Capitolo 7

Conclusioni

7.1 Conoscenze acquisite

Il progetto si colloca in un servizio più grande qual è *Monokee*, motivo per cui si è reso necessario uno sviluppo che tenesse conto di differenti progetti esistenti, di cui alcuni già in produzione ed altri in via di sviluppo (i.e. il componente ITF). Questo mi ha permesso di lavorare in un contesto che richiedeva di interagire costantemente con altri team e quindi operare in maniera controllata, disciplinata e coordinata. È stato quindi fondamentale acquisire pratiche di ingegneria del software e attuare correttamente le pratiche aziendali (queste si basavano su [Scrum](#)).

Il piano di lavoro prevedeva lo sviluppo di due componenti: l'**Identity Wallet** (IW) e il **Service Provider** (SP). Questi sono da considerarsi due prodotti completamente separati tra di loro che concorrono, in aggiunta al componente ITF (sviluppato da un altro stagista), a fornire un unico servizio. Sono applicativi di natura diversa; il primo un'applicazione mobile, il secondo un'applicazione server, che, di conseguenza hanno richiesto conoscenze diverse dandomi la possibilità di acquisire più competenze.

Di seguito viene proposta una lista delle principali competenze acquisite durante le attività di stage e non precedentemente conosciute:

- * apprendimento del linguaggio C#;
- * sviluppo di interfacce touch in Xamarin;
- * creazione di servizi RESTful;
- * apprendimento dell'uso di WebSocket;
- * uso del framework .NET Core;
- * uso del framework Asp.NET;
- * progettazione di un'architettura Event Driven;
- * uso di RabbitMQ e MassTransit;
- * integrazione con SAML.

Inoltre ho avuto l'opportunità di approfondire e mettere in pratica alcune delle conoscenze acquisite durante il corso di laurea triennale; tra queste riporto:

- * metodologie e pratiche di ingegneria del software;
- * uso di HTML5 e javascript;
- * uso di lambda funzioni (*arrow function*) in un linguaggio imperativo;
- * ideazione e codifica di test.

7.2 Valutazione personale

Il periodo di stage svolto presso *IvoxIT*, a conclusione del mio percorso di studi, è stato fondamentale per approfondire ed ampliare le conoscenze in mio possesso; tra queste, alcune di carattere tecnologico, altre di carattere metodologico e, a tal proposito, indispensabili sono stati i vari corsi di programmazione e il corso di ingegneria del software. Oltre a ciò, ho avuto modo di sviluppare nuove competenze, apprendere tecnologie e sistemi che mi torneranno utili nel futuro.

In questi mesi ho avuto l'occasione di lavorare ed inserirmi in un gruppo estremamente affiatato e coeso con cui ho avuto la possibilità di confrontarmi al fine di rendere il mio progetto veramente utile agli scopi aziendali. Ho potuto fare affidamento su persone con grande esperienza e capacità, e questo mi ha permesso di rendere più veloci e di maggior qualità le attività di analisi, progettazione e codifica. Ho inoltre apprezzato come i miei colleghi, anche se non a conoscenza della particolare tecnologia, disponessero di attitudine e metodo, cosa che permetteva loro un rapido apprendimento e la capacità di fornirmi un aiuto concreto. I rapporti instaurati non sono stati solo di natura prettamente didattica e lavorativa, ma anche di amicizia e stima reciproca. Penso di aver avuto modo di lavorare e consultarmi con persone di elevata caratura sia lavorativa che personale.

Il progetto è stato sviluppato nei tempi previsti e nonostante le grandi fluttuazioni in termini di requisiti e di aspetti progettuali che questo ha avuto, il risultato è stato soddisfacente. Tuttavia il prodotto finale ha fatto emergere le innumerevoli problematiche legate all'uso della tecnologia [blockchain](#). Una blockchain permissionless ha infatti tempi di elaborazione difficilmente accettabili da un utente medio. Nel corso della settimana finale si sono svolti test di prestazione sulla rete [Ropsten](#) che hanno mostrato come anche per le chiamate più semplici, che richiedevano almeno un'operazione di scrittura, ci fosse un tempo minimo di attesa pari a 30s. Ciò nonostante, a seguito di operazioni di ottimizzazione, siamo riusciti ad eseguire le chiamate alla blockchain in parallelo arrivando ad un tempo totale necessario al login pari a 32s. Tale lentezza è dovuta alla necessaria *proof of interest*, che in una rete di questo tipo consiste nel risolvere un problema matematico (*proof of work*) di difficoltà variabile in base alle esigenze. In conclusione, ritengo che i vantaggi legati all'uso di una blockchain siano:

- * affidabilità;
- * disponibilità;
- * eliminazione di intermediari

non sufficienti a giustificare nella maggioranza dei casi una durata così elevata per una semplice operazione di login. Inoltre, uno degli obiettivi principali dell'azienda era quello di inglobare tale sistema in quello attuale. Ciò ha portato ad una serie di scelte che hanno annullato alcune fondamentali caratteristiche della blockchain. Il

componente SP rappresenta un sistema centralizzato e quindi un grosso *point of failure* che mina l'elevata disponibilità tipica della blockchain (in quanto sistema distribuito). Un altro punto critico riguarda invece l'uso di account gestiti direttamente dall'azienda per tutte le operazioni. Lo scopo era quello di rendere gli utenti liberi dal dover gestire i propri account, ma in una previsione di alto utilizzo del sistema renderebbe complicato per una persona verificare le transazioni e quindi farebbe venir meno la caratteristica di fiducia tipica delle blockchain, in quanto la mole di transazioni renderebbe difficile l'identificazione delle proprie.

In conclusione da questa esperienza ho appreso che una buona preparazione sia in termini di conoscenze che di competenze, è sì necessaria, ma non preclusiva ai fini del lavoro. È invece indispensabile l'attitudine e il pensiero analitico, il quale permette di trovare la soluzione più adatta di fronte ai problemi e affrontare al meglio le varie difficoltà.

Infine, avendo avuto uno spaccato reale di quello che il mio percorso di studi si può riversare a livello professionale, ho avuto maggiore chiarezza dei possibili ambiti occupazionali e di conseguenza ho potuto fare delle considerazioni, rafforzando così la mia convinzione nel proseguire con gli studi magistrali e maturando prospettive più precise.

La conclusione è il punto dove ti
sei stufato di pensare.

Arthur Bloch

Appendice A

Appendice A

A.1 Strumenti utili per lavorare su Ethereum

Di seguito una breve descrizione di alcuni strumenti utili per lavorare su *Ethereum*.

- * **Truffle**: è una suite di development e testing. Permette di compilare, buildare ed effettuare la migrazione degli SmartContract. Inoltre ha funzioni di debugging e di scripting. La suite offre la possibilità di effettuare test degli SmartContract sia in Javascript (con l'utilizzo di Chai), sia in Solidity. Si riporta di seguito il sito del progetto: **site:truffle**
- * **Ganache**: è uno strumento rapido che permette di creare e mantenere in locale una rete blockchain Ethereum personale. Può essere usata per eseguire test, eseguire comandi e per operazioni di controllo dello stato mentre il codice esegue. Si riporta di seguito il sito del progetto: **site:ganache**
- * **Mist**: è un browser sviluppato direttamente dal team Ethereum in grado di operare transazioni direttamente nella blockchain senza la necessità di possedere un intero nodo. È estremamente immaturo e non utilizzabile in produzione. Si riporta di seguito il sito del progetto: **site:mist**
- * **Parity**: è un client Ethereum che permette di operare sulla rete senza necessità di possedere un intero nodo. Questa soluzione a differenza di Mist dovrebbe risultare più facilmente integrabile nel prodotto senza che l'utente ne abbia consapevolezza. Si riporta di seguito il sito del progetto: **site:parity** .
- * **Metamask**: è uno plugin disponibile per i browser Chrome, Firefox, e Opera. Permette di interfacciarsi alla rete Ethereum senza la necessità di eseguire in intero nodo della rete. Il plugin include un wallet con cui l'utente può inserire il proprio account tramite la chiave privata. Una volta inserito l'account il plugin farà da tramite tra l'applicazione e la rete. Metamask è utilizzato dalla maggioranza delle applicazioni Ethereum presenti on line, questo però rappresenterebbe un componente esterno compatibile con pochi browser desktop. Si riporta di seguito il sito del progetto: **site:metamask** .
- * **Status**: è un progetto che propone una serie di [Application Program Interface](#) che permettono di sviluppare un'applicazione mobile nativa operante direttamente su blockchain senza la necessità di possedere un intero nodo. Il sito del progetto

propone una serie di applicazioni che utilizzano Status. Tuttavia nessuna di queste applicazioni risulta attualmente rilasciate in nessuno store. Status risulta in early access ed è disponibile per Android e iOS. Il sito del progetto è il seguente:
site:status

- * **Microsoft Azure:** “Ethereum Blockchain as a Service” è un servizio fornito da Microsoft e ConsenSys che permette di sviluppare a basso costo in un ambiente di dev/test/produzione. Permette di creare reti private, pubbliche e di consorzio. Queste reti saranno poi accessibili attraverso la rete privata Azure. Questa tecnologia rende facile l’integrazione con Cortana Analytics, Power BI, Azure Active Directory, O365 e CRMOL.

Bibliografia