IW – Architettura di massima

Autore: Simone Ballarin

Data: 25/06/18

Destinatari: Athesys

Diario delle modifiche

Data	Descrizione	Autore
25/06/2018	Creazione documento e stesura dei capitoli: Scopo del documento, Riferimenti, Descrizione, Architettura, Architettura.	Simone Ballarin

Scopo del documento

Il seguente documento *IW - Architettura di massima* ha lo scopo di presentare e dimostrare una prima architettura di massima per il componente Identity Wallet (IW) che dovrà funzionare nel contesto di un'estensione del prodotto MonoKee basata su blockchain.

Sintesi del documento

Questo documento inizia con la descrizione del prodotto che si andrà a implementare, prosegue poi con una generica introduzione all'architettura Xamarin ed infine conclude presentando una prima ipotesi di architettura in formato UML 2.0.

Riferimenti

- Mobile Application Pocket Guide v1.1, Microsoft patterns & practices;
- IW – Analisi requisiti;
- IW – Studio di fattibilità.
- www.xamarin.com

Descrizione

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 (IAM) composto da quattro principali fattori:

1. Identity Wallet (IW)
2. Service Provider (SP)
3. Identity Trust Fabric (ITF)
4. 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, mandare i propri dati (IPP) all'ITF la quale

custodirà la sua identità e farà da garante per le asserzioni proveniente 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 ai seguenti compiti:

nell'ambito della registrazione di un utente il Wallet deve:

- generare e immaganizzare in locale una chiave pubblica;
- generare e immaganizzare in locale una chiave private;
- creare l'hash della chiave la chiave pubblica e inviare l'hash 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, autentificazione 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.

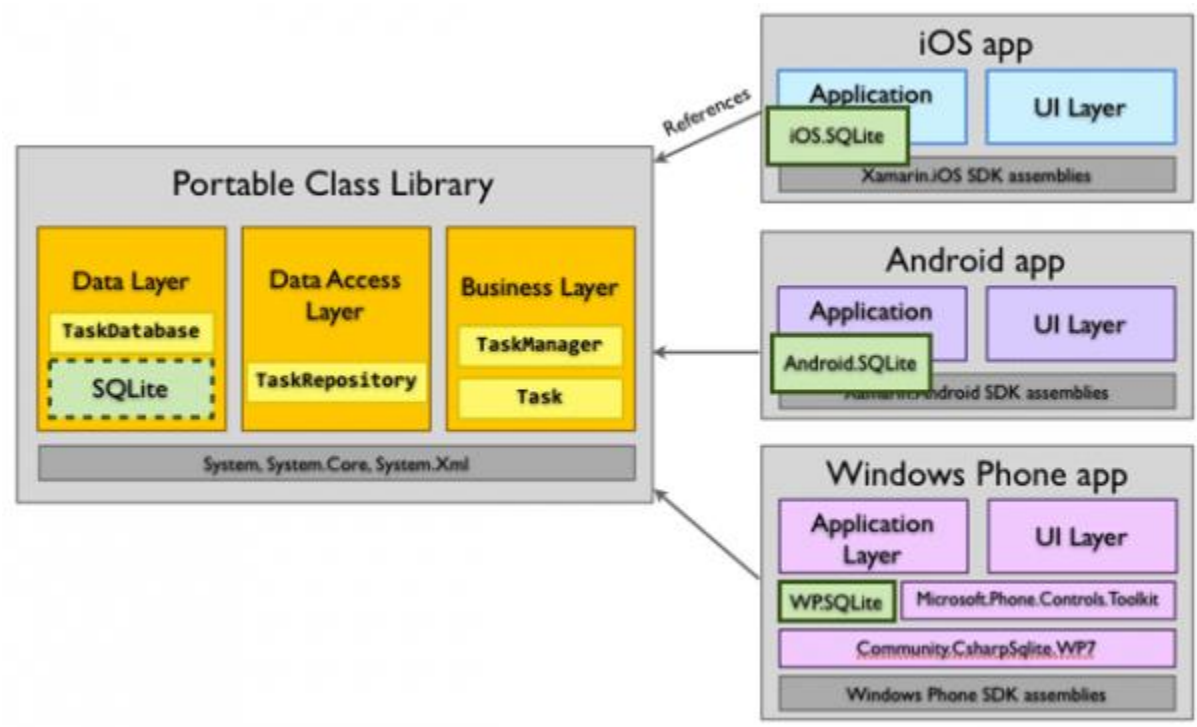
Architettura di massima

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

PCL è un approccio alla condivisione del codice tra le diverse edizioni dell'app destinate a diversi sistemi operativi mobili sviluppato da Xamarin. Segue un diagramma esplicativo di come si sviluppa una tipica architettura PCL. Il diagramma è tratto da www.xamarin.com.



Ogni "Platform-Specific Application Project" (iOS app, Android app, Windows Phone app) referenzia la PCL. Quindi esistono essenzialmente due parti: quelle specifiche per la piattaforma e quelle condivise. Obiettivo del progetto è quello di rendere meno corposa possibile le parti specifiche. Sarà poi possibile impiegare caratteristiche di una determinata piattaforma attraverso l'utilizzo del design pattern Dependency Injection (DI).

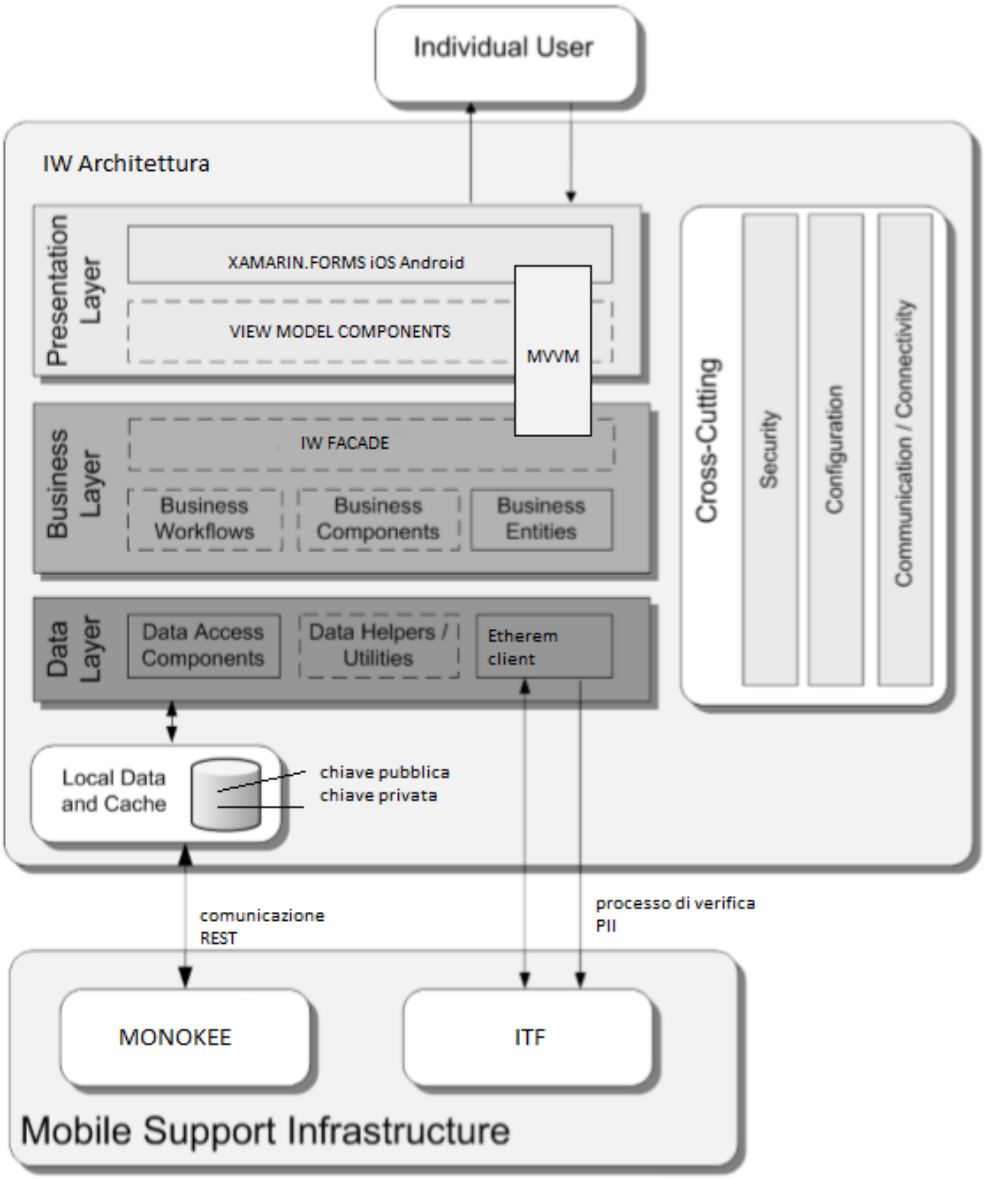
Applicare i principi della DI significa definire nel codice condiviso interfacce (classi astratte) che vengono implementate (estese) in ogni piattaforma tramite sottoclassi (Strategy Pattern). A questo punto, sarà possibile integrare queste specifiche implementazioni all'interno della PCL. Xamarin per questo scopo offre la classe DependencyService.

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 ricadano sul componente ITF.



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 o alla risorsa Monokee tramite comunicazione REST, o all'ITF tramite l'utilizzo di un client Ethereum. Tutto verrà sviluppato utilizzando il framework .NET.

Design Pattern

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 da classi diverse e quindi canali diversi. Questo è risultato utile per rendere gli altri componenti quanto più indipendenti da come comunicano con l'esterno.

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

Entity Translator: Un oggetto che trasforma un dato in una forma utile per essere usato nella logica di business. Questo pattern 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. Questo pattern è stato ampiamente utilizzato, specie per rendere il più leggero possibile la creazione dei dati dell'utente e della verifica dei dati nell'ITF.

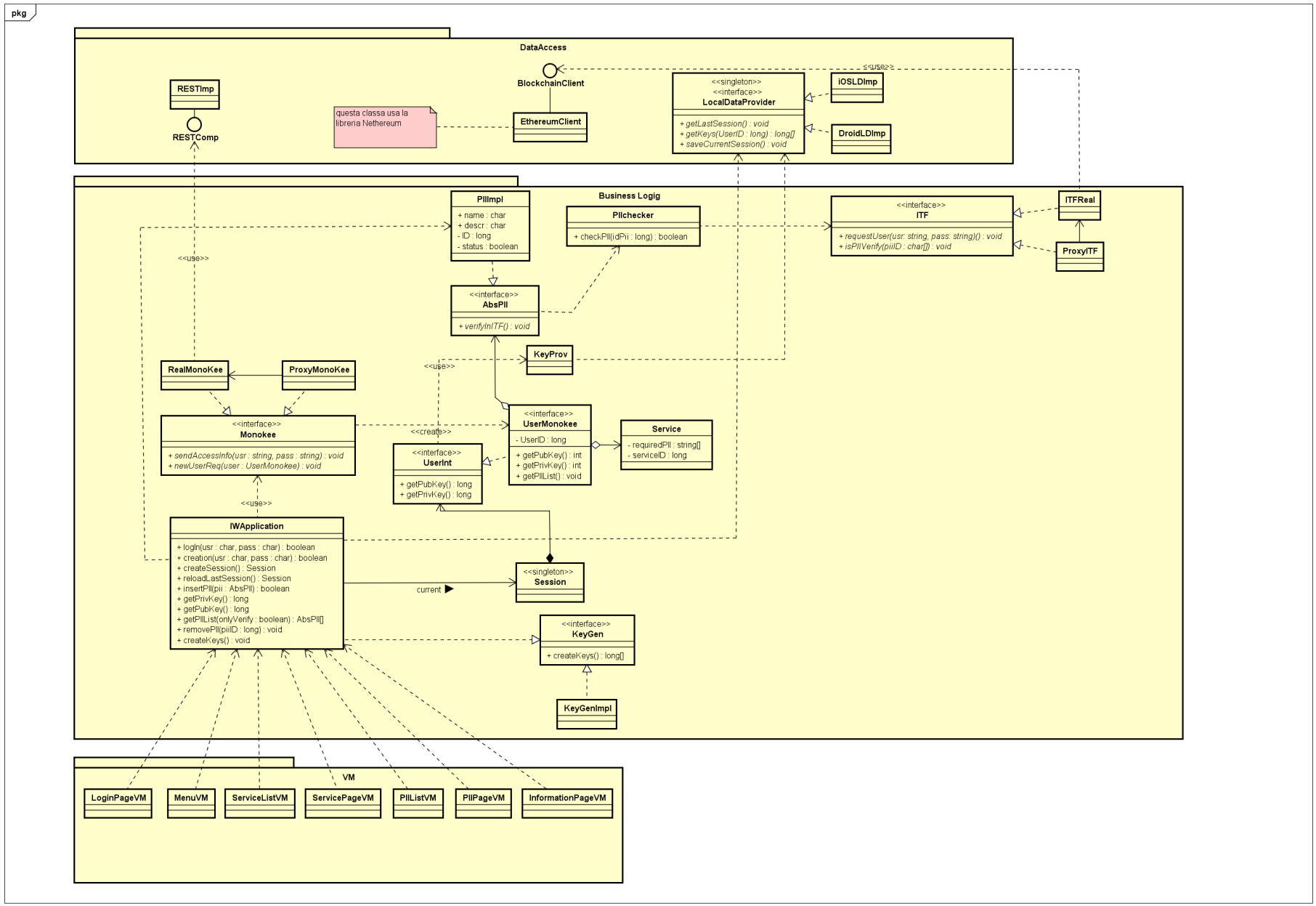
Strategy Pattern: è un oggetto che permette di separare l'esecuzione di un metodo dalla classe che lo contiene. Usando un'interfaccia per astrarre il metodo è poi possibile crearne molteplici implementazioni. Questo è risultato molto utile nel contesto di un'applicazione multi piattaforma in cui alcune 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.

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

Diagrammi UML

Di seguito viene presentato il diagramma di massima dell'architettura dell'IW. Il diagramma è stata redatto seguendo lo standard UML 2.0. Subito a seguire viene descritta ogni classe.


pkg
DataAccess
RESTImp
RESTComp
BlockchainClient
questa classa usa la libreria Nethereum
EthereumClient
<<use>>
<<singleton>>
<<interface>>
LocalDataProvider
+ getLastSession() : void
+ getKeys(UserID : long) : long[]
+ saveCurrentSession() : void
iOSLDImp
DroidLDImp
Business Logig
PIIImpl
+ name : char
+ descr : char
- ID : long
- status : boolean
PIIchecker
+ checkPII(idPii : long) : boolean
<<interface>>
ITF
+ requestUser(usr: string, pass: string)() : void
+ isPIIVerify(piiID : char[]) : void
ITFReal
ProxyITF
<<interface>>
AbsPII
+ verifyInITF() : void
KeyProv
RealMonoKee
ProxyMonoKee
<<interface>>
Monokee
+ sendAccessInfo(usr : string, pass : string) : void
+ newUserReq(user : UserMonokee) : void
<<create>>
<<interface>>
UserMonokee
- UserID : long
+ getPubKey() : int
+ getPrivKey() : int
+ getPIIList() : void
Service
- requiredPII : string[]
- serviceID : long
<<interface>>
UserInt
+ getPubKey() : long
+ getPrivKey() : long
IWApplication
+ logIn(usr : char, pass : char) : boolean
+ creation(usr : char, pass : char) : boolean
+ createSession() : Session
+ reloadLastSession() : Session
+ insertPII(pii : AbsPII) : boolean
+ getPrivKey() : long
+ getPubKey() : long
+ getPIIList(onlyVerify : boolean) : AbsPII[]
+ removePII(piiID : long) : void
+ createKeys() : void
current
<<singleton>>
Session
<<interface>>
KeyGen
+ createKeys() : long[]
KeyGenImpl
VM
LoginPageVM
MenuVM
ServiceListVM
ServicePageVM
PIIListVM
PIIPageVM
InformationPageVM

Logica di business

IWApplication: questa classe ha il compito di fornire una facade per i vari ViewModel. Tutte le azioni possibile tramite l'interfaccia sono quindi implementate da questa classe.

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.

KeyGen: è un'interfaccia che ha lo scopo di definire una strategia di generazione chiavi, fa parte di un'applicazione dello Strategy Pattern. È stata pensata in un'ottica in cui ci possono essere vari modi per generare una chiave a seconda del sistema operativo usato.

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

Session: è una 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 contiene le informazioni utente.

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

UserMonokee: è una classe che rappresenta un utente proveniente dal server Monokee. Implementa l'interfaccia Monokee. Un utente di questo tipo possiede un aggregato di servizi, potenzialmente contiene le chiavi e possiede una lista di PII.

Service: è una 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: è una classe che ha il compito di occuparsi della generazione delle chiavi private e pubbliche. Questa classe viene usata da UserInt e a sua volta usa LocalDataProvider.

LocalDataProvider: è un'interfaccia che ha il compito di fornire in singolo punto dove ottenere informazione dal file system locale. Questa classe poi deve venire 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: è un'interfaccia che rappresenta una generica PII, questa per ora ha una sola possibile implementazione, ma un'interfaccia di questo tipo renderà più semplice l'implementazione di future PII.

PIIImpl: è una 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: si tratta di un'interfaccia con il compito di fornire un'astrazione del componente Identity Trust Fabric. Questa interfaccia con ITFReal 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.

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

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

Layer di accesso ai dati

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

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

BlockchainClient: è un'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: è una possibile implementazione di BlockchainClient che utilizza la rete Ethereum. Questa classe poi userà la libreria Nethereum.

Layer di presentazione

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

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

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