SP – 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.	Simone Ballarin

Scopo del documento

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

Sintesi del documento

Questo documento inizia con la descrizione del prodotto che si andrà a implementare, prosegue poi con una generica introduzione alle architetture Event Driven. Viene poi scelto di utilizzare un approccio Broken topology, la scelta è motivata dalla maggiore indipendenza tra i vari componenti rispetto ad un approccio Mediator topology. Infine si conclude presentando una prima ipotesi di architettura in formato UML 2.0.

Riferimenti

- SP – Analisi requisiti;
- SP – Studio di fattibilità;
- www.math.unipd.it/~rcardin/sweb/2018/Software%20Architecture%20Patterns_4x4.pdf

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:

- 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, mandare i propri dati 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 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 dall'Identity Wallet;
- generare e comparare gli hash dei valori ottenuti con quelli presenti nell'ITF;
- verificare che l'identità e le altre informazioni ottenute siamo 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.

Si intende, quindi, sviluppare il componente Service come un'applicazione server che opera in collaborazione con l'ITF al fine di la veridicità dei dati provenienti dall'IW e quindi garantire o meno l'accesso al servizio. Questa connessione deve avvenire tramite il protocollo JSON-RFC descritto nello Yellow Paper. Questi dati vengono presentati sotto forma di codice QR. L'applicazione deve inoltre avere un'interfaccia web accessibile tramite Internet dalla quale il personale del fornitore può configurare il servizio. Si fa notare come con SP non si intenda il servizio o il fornitore dei servizi, ma semplicemente il componente MonoKee che ha lo scopo di interfacciarsi con questi.

Architettura di massima

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

Questo tipologia di architettura rappresenta uno dei principali esempi di pattern architettura asincrono. Produce applicati altamente scalabili e facilmente adattabili ad ogni carico di utilizza. Se applicata bene fornisce la possibilità di avere eventi con un singolo scopo (Single Responsibility Principle) 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

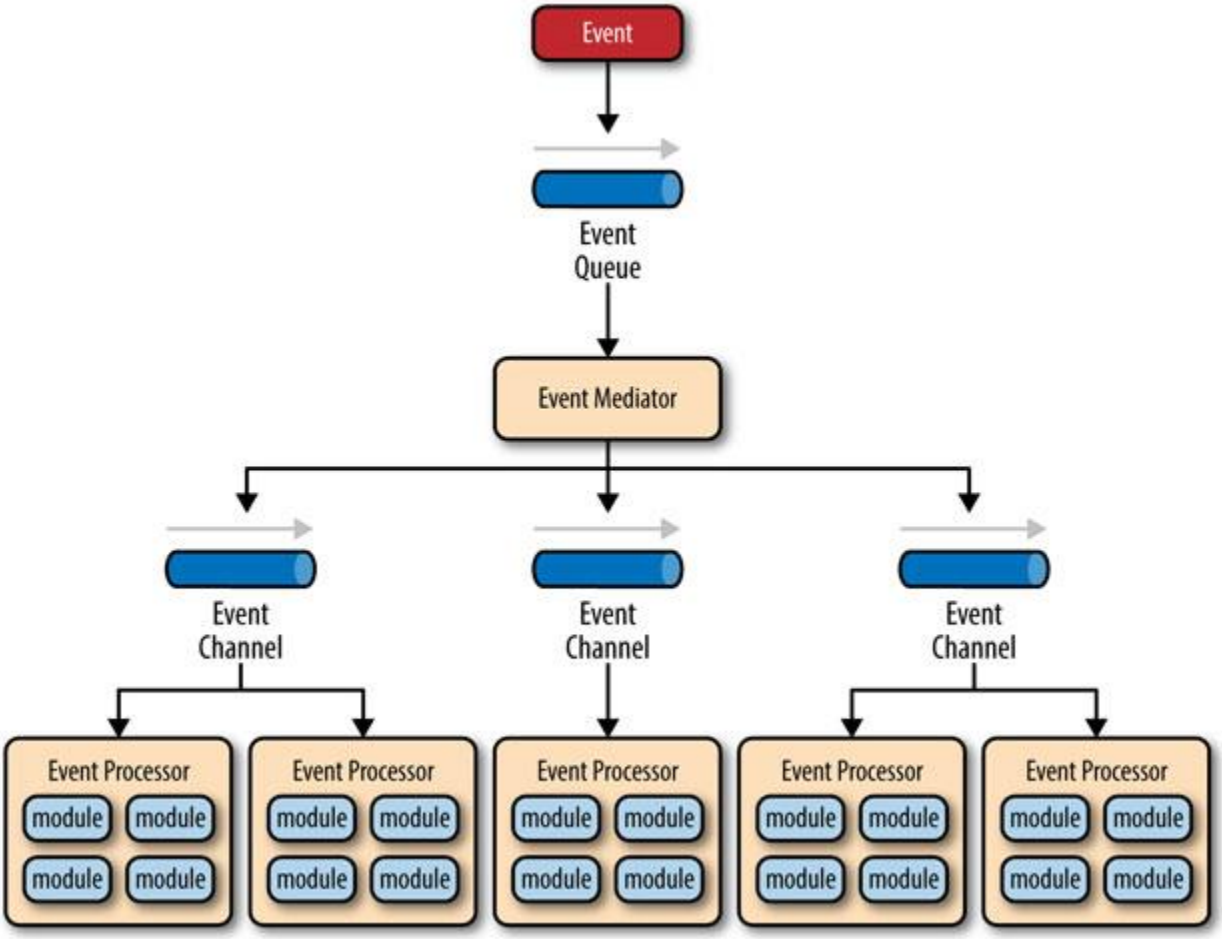
Un evento generalmente possiede una serie di passi ordinati per essere eseguito. In questa approccio ci sono quattro componenti che interagiscono fra loro:

- 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.

Di seguito si riporta una generica architettura Event Driven 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 processo ogni evento. Sono auto contenuti, indipendenti ed scarsamente accoppiati.

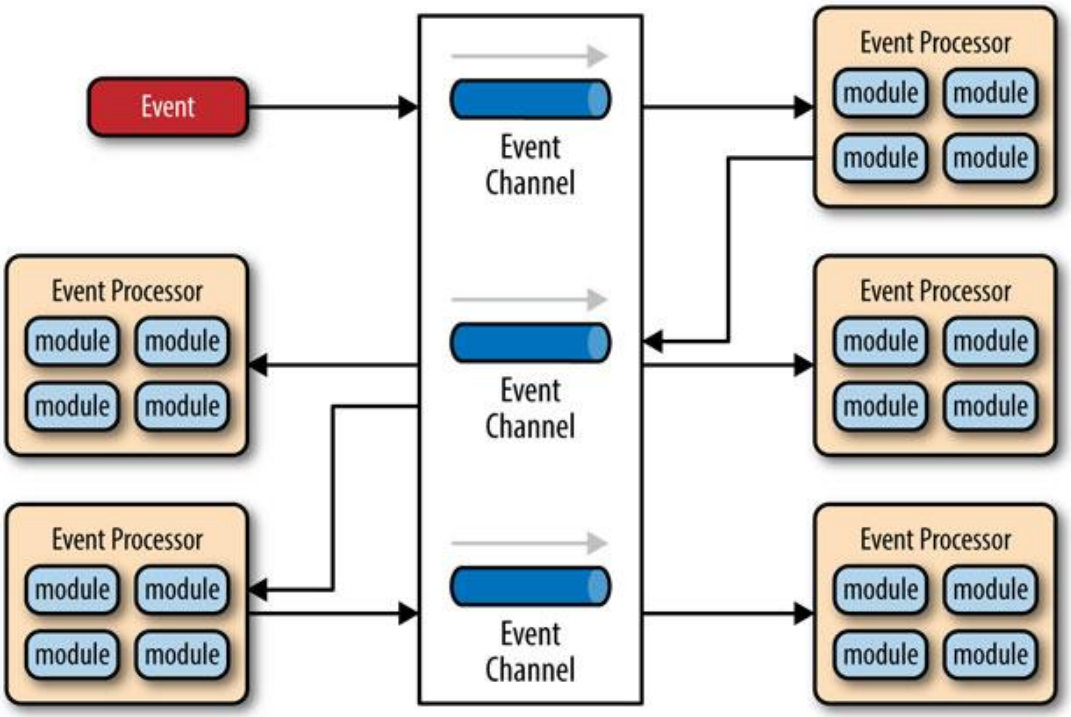
Broker topology

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.

Di seguito si riporta una generica architettura Event Driven Broker Topology.



Considerazioni

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

Caratteristica	**Considerazioni**
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 difficoltosi. 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 architettura.

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.

Overview

Come già detto l'applicazione sarà strutturata con una architettura Event Driven di tipo Broker topoligy, questo implica che la logica di funzionamento sia incapsulata nei vari passaggi tra le varie code.

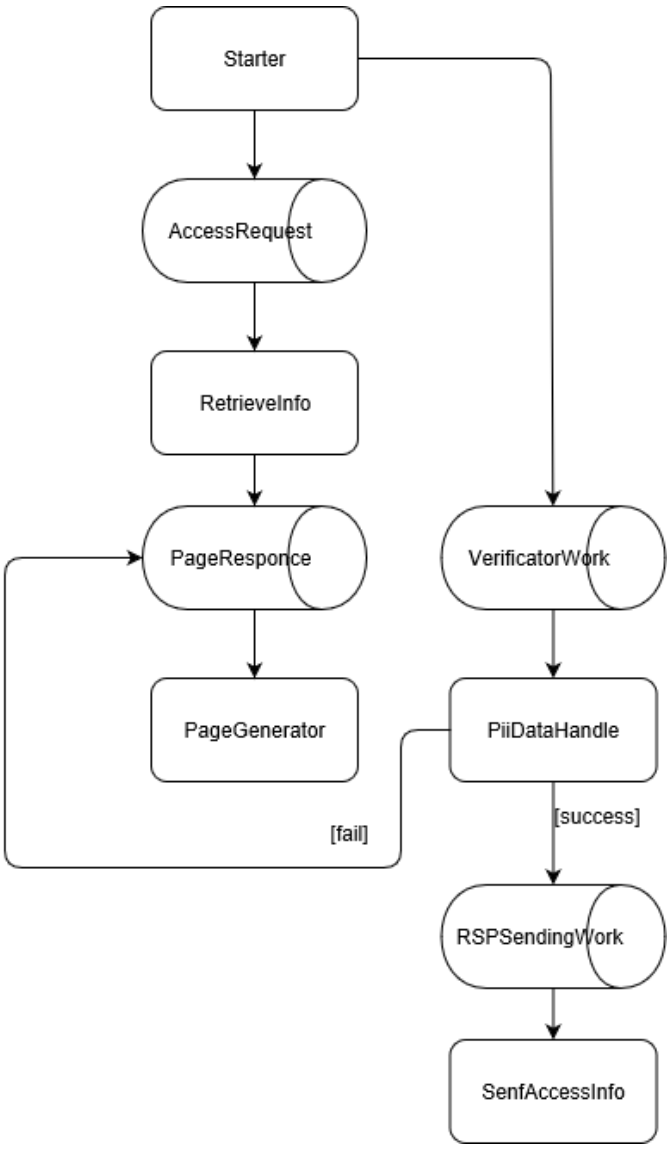
Gli esecutori sono i seguenti cinque:

- 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 broswer 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 seguente diagramma 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.

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.

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.

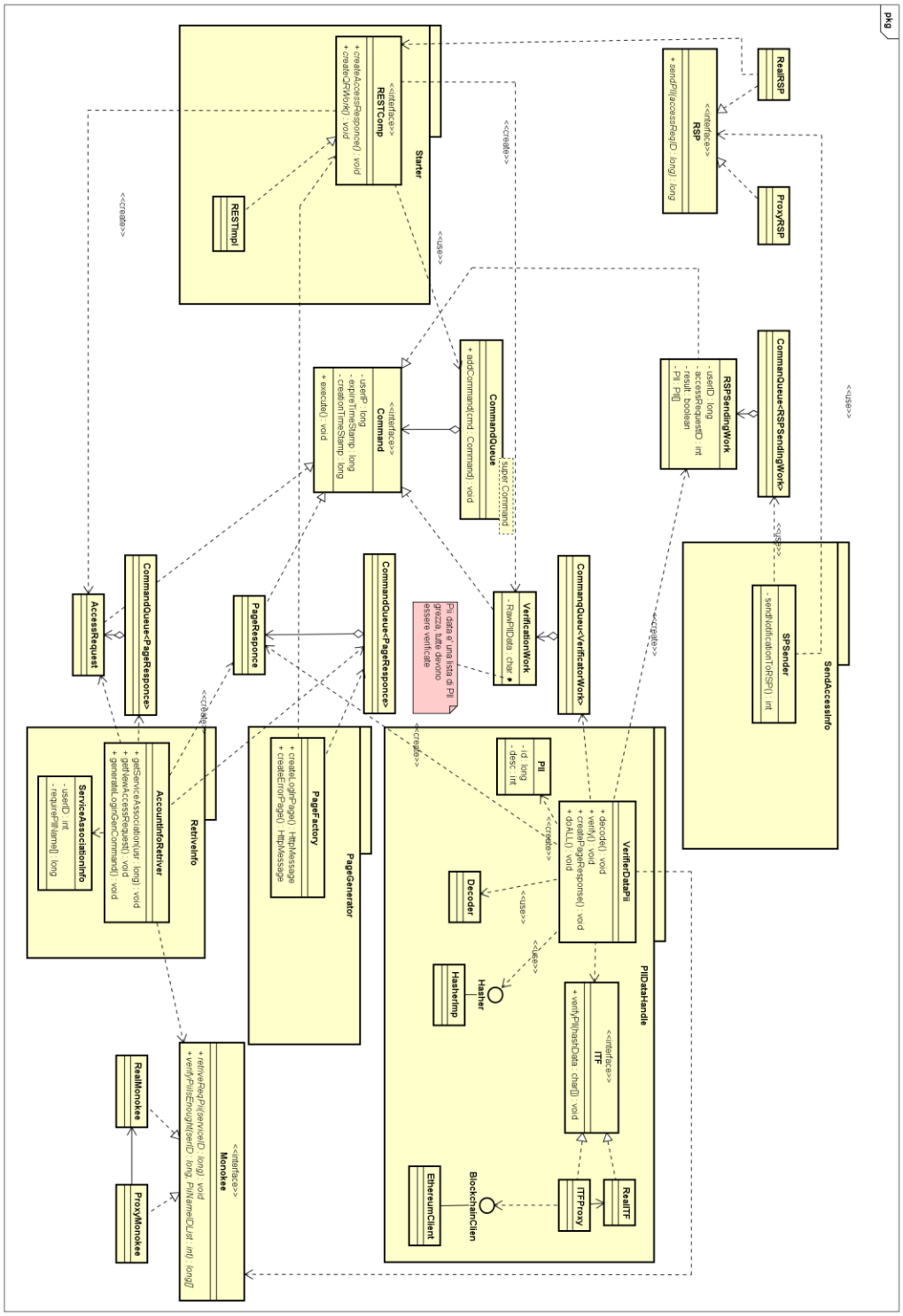
Strategy Pattern: è un oggetto che permette di separare l'esecuzione di un metodo dalla classe che lo contiene. Usando un'interfaccia per astrarre il metodo è poi possibile crearne molteplici implementazioni. Questo è risultato molto utile nel contesto di un'applicazione multi piattaforma in cui alcune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

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

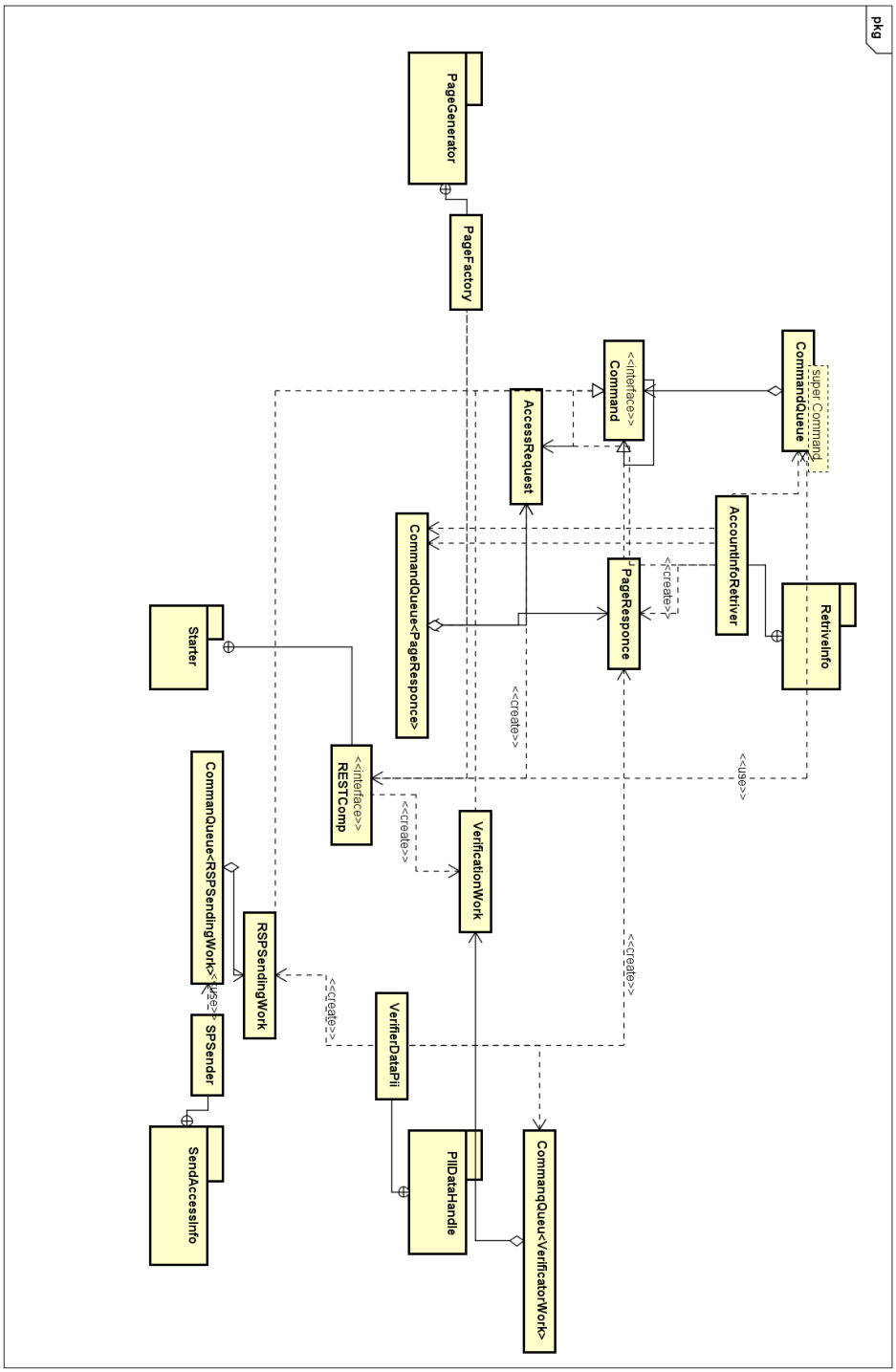
Diagrammi UML

Ora 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.

Si riporta anche una versione del diagramma che nasconde l'interno dei vari componenti.



Descrizione ADT

Nome	Descrizione
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.
RSP	**si** tratta di un'interfaccia con il compito di fornire un'astrazione del componente real service provider. Questa interfaccia con RSPReal e ProxyRSP rappresenta un'applicazione del pattern Proxy.
RealRSP	è una classe che rappresenta il reale oggetto RSP, questa classe poi dialoga con il RESTComp per ottenere i dati. Questa classe con RealRSP e ProxyRSP rappresenta un'applicazione del pattern Proxy.
ProxyRSP	è una classe che rappresenta un proxy dell'oggetto RSP, questa classe applica una politica di acquisizione remota. Questa classe con RealRSP e ProxyRSP rappresenta un'applicazione del pattern Proxy.
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; • 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. 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.
CommandQueue	Questo template definisce una coda di command. Dispone delle funzionalità per gestire la coda in maniera concorrente.
Account Retriver	È la classe che ha il compito di guidare l'esecuzione di un AccessRequest.
ServiceAssociationInfo	È una classe generata da Monokee che rappresenta un l'associazione tra utente e servizio e il nome dei PII richiesti.
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
PageFactory	È la classe che si occupa di eseguire l'evento PageResponce e quindi di generare le pagine e inviarle.

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.
VerifierDataPii	È la classe che ha il compito di gestire i VerificationWork, si tratta di un'applicazione di Templete Patter. Ha il compito di verificare le informazioni nell'ITF e di inviare una RSPSendingWork in caso di successo o una PageResponce in caso di fallimento.
PII	È una classe che rappresenta una PII. Contiene l'id, la descrizione di una PII.
Decoder	è una classe che ha il compito di decodificare le informazioni presentate tramite il codice QR e quindi generare una serie di PII.
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.
SPSender	È la classe che ha il compito di eseguire i RSPSendingWork. Invia tramite il componente RestCompOut le informazioni relative l'accesso al RSP.