

**ALMA MATER STUDIORUM - UNIVERSITA DI BOLOGNA**

---

**SCUOLA DI INGEGNERIA E ARCHITETTURA**

Dipartimento di Informatica - Scienza e Ingegneria - DISI  
Corso di Laurea Triennale in Ingegneria Informatica

**TESI DI LAUREA**

in

**LABORATORIO DI AMMINISTRAZIONE DI SISTEMI**

**Analisi e sviluppo di un gestore di dati  
personali secondo il modello MyData**

**CANDIDATO**

Giada Martina Stivala

**RELATORE**

Chiar.mo Prof. Marco Prandini

**CORRELATORE**

Dott. Andrea Melis

**Anno Accademico 2016/17  
Sessione II**



*Curiosity killed the cat,  
but satisfaction brought it back*



# Indice

<b>1</b>	<b>Introduzione</b>	<b>9</b>
1.1	MyData, Big Data . . . . .	10
1.2	Struttura della tesi . . . . .	11
<b>2</b>	<b>Contesto di sviluppo</b>	<b>13</b>
2.1	Mobility As A Service . . . . .	13
2.1.1	Mobility Profile e Journey Planner . . . . .	14
2.2	Smart Mobility For All . . . . .	14
2.3	GDPR? . . . . .	15
2.4	Personal Clouds? . . . . .	15
<b>3</b>	<b>Architettura MyData</b>	<b>17</b>
3.1	Entità Fondamentali . . . . .	17
3.2	Service Registry, Service Linking . . . . .	17
3.2.1	OAuth 2.0 . . . . .	18
3.3	Autorizzazioni e Consent . . . . .	18
3.3.1	Kantara Consent & Information Sharing Work Group . . . . .	19
3.3.2	User Managed Access . . . . .	19
3.4	Personal Data Storage . . . . .	19
<b>4</b>	<b>Analisi e Design</b>	<b>21</b>
4.1	Accounting e servizi . . . . .	21
4.2	Operatore MyData . . . . .	22
4.3	Service Registry, Service Linking . . . . .	23
4.4	Consent . . . . .	23
4.5	Personal Data Storage . . . . .	24
4.6	Rappresentazione di dati non noti a priori . . . . .	24
<b>5</b>	<b>Progettazione</b>	<b>27</b>
5.1	Flusso del programma . . . . .	27
5.2	Accounting . . . . .	27

5.2.1	IUser, MyDataUser . . . . .	28
5.2.2	IAccount, Account . . . . .	29
5.3	SecurityManager . . . . .	30
5.4	IMyData, MyData . . . . .	31
5.5	Autorizzazioni e Consent . . . . .	32
5.5.1	ConsentManager, ConsentStatus . . . . .	32
5.5.2	ServiceConsent, DataConsent . . . . .	33
5.6	PersonalDataVault . . . . .	35
5.7	Metadata, DataSet . . . . .	37
5.8	IService, AbstractService, MostLikelyNextTrip . . . . .	38
5.9	ServiceRegistry . . . . .	38
5.10	Uso delle eccezioni . . . . .	38
<b>6</b>	<b>Conclusioni e sviluppi futuri</b>	<b>41</b>
	<b>Bibliografia</b>	<b>43</b>







# Capitolo 1

## Introduzione

Negli ultimi anni, con la diffusione degli smartphone e l'aumento dell'accesso a Internet, la tecnologia è diventata sempre più legata all'utente dietro lo schermo, con esempi che possono spaziare dai social network ai dispositivi di monitoraggio della salute e dell'attività fisica. La quantità di informazioni raccolte tramite questi servizi non si limita a ciò che l'utente sceglie di condividere (foto, email, transazioni online), ma comprende anche dati osservati (abitudini di navigazione, dati di geolocalizzazione) e deduzioni (*targeted advertising*, previsioni sul flusso del traffico). (Source: Information Accountability Foundation, World Economic Forum, Marc E. Davis) [25]

L'importanza dei dati personali è diventata tale da richiedere l'esistenza di leggi che ne regolamentassero l'utilizzo, come il regolamento generale sulla protezione dei dati (GDPR, *General Data Protection Regulation- Regolamento UE 2016/679*). [http://static.ow.ly/docs/Regulation\\_consolidated\\_text\\_EN\\_-47uW.pdf](http://static.ow.ly/docs/Regulation_consolidated_text_EN_-47uW.pdf)

In questo contesto si sviluppa il modello *MyData*, fondato sull'idea secondo cui ogni utente ha il controllo sui propri dati personali, ed è a conoscenza dell'utilizzo che ne viene fatto, in modo trasparente. D'altra parte, l'architettura *MyData* vuole offrire al mercato e alle imprese un contesto di sviluppo di applicazioni software in cui vengono rispettate le leggi in materia di protezione dei dati sensibili, favorendo l'interoperabilità fra di esse.

L'approccio *MyData*, seppur innovativo, non è senza precedenti. Molti dei suoi concetti fondamentali sono anche alla base dei *personal cloud*, come ad esempio la possibilità di contenere in modo sicuro i dati personali dell'utente e offrire interoperabilità fra diversi servizi, utilizzando concretamente i dati memorizzati.

Questa tesi si propone di studiare i concetti fondamentali alla base del modello *MyData*, e implementare un prototipo di gestore di dati personali che ne rispetti le specifiche. A titolo di esempio, si è scelto di utilizzare come servizio

“consumatore” di dati personali un sistema di previsione del viaggio più probabile.

L’obiettivo è quello di realizzare un sistema che permetta all’utente di monitorare l’utilizzo dei suoi dati personali in tempo reale ed in modo trasparente, attraverso l’implementazione di una politica di controllo degli accessi. Inoltre, il gestore opera in un contesto in cui il rispetto delle politiche di sicurezza è fondamentale: anche se la realizzazione di un sistema sicuro non rientra negli obiettivi del lavoro di tesi, sono presenti alcuni accorgimenti che puntano in questa direzione.

## 1.1 MyData, Big Data

Il progetto *MyData* nasce in Finlandia, all’università di Aalto, partendo dall’idea di rafforzare i diritti digitali dell’individuo. All’interno del contesto vi è anche la stringente legislazione europea sulla protezione dei dati personali, e la volontà di offrire alle aziende un nuovo modo di approcciarsi al cliente, basato non su incomprensibili “Termini e Condizioni” ma sulla reciproca fiducia.

Attualmente, grandi quantità di dati vengono continuamente raccolti senza utilizzarne pienamente la ricchezza dell’informazione. Il loro percorso e le operazioni di processamento a cui vengono sottoposti sono sempre più numerose e cresce la difficoltà nel conservarne le tracce. Inoltre, l’utilizzo di software proprietari limita la possibilità di analisi dei dati a causa della scarsa interoperabilità fra le soluzioni adottate. Infine, va anche considerato l’impatto che tale modus operandi ha sugli utilizzatori finali: secondo un sondaggio riportato dal World Economic Forum, “Fully 78% of consumers think it is hard to trust companies when it comes to use of their personal data” (Orange, The Future of Digital Trust, 2014) [25].

*MyData* cerca di trovare una risposta a questi temi attraverso un cambio di paradigma che ponga l’utente al centro, attribuendogli la capacità di gestire i propri dati personali e di comprendere l’uso che ne viene fatto. Allo stesso tempo, la qualità dei suggerimenti ricevuti in base ai dati raccolti aumenterebbe grazie alla maggiore disponibilità degli stessi. L’elaborazione dei dati potrebbe favorire anche conoscenze più approfondite del proprio comportamento da utenti (self tracking), anche attraverso la ricezione di un compenso per il processamento dei propri dati (data monetization)[24].

L’approccio a livello infrastrutturale permetterebbe l’indipendenza da specifici settori (sanità e salute, finanza, ecc.), favorendo una completa portabilità dei dati. Il rispetto e l’aderenza alle leggi verrebbe realizzato dall’infrastruttura stessa, consentendo alle aziende di sviluppare i software in maniera meno vincolata, acquisendo al contempo la fiducia del cliente grazie alla trasparenza e all’affidabilità garantite da *MyData*. Nel complesso, il paradigma *MyData* si contrappone

all'attuale standard, Big Data, proponendo nuove soluzioni che non trascurino la realtà del mercato in cui il software viene usato.

## 1.2 Struttura della tesi

In questa sezione si illustra brevemente la struttura della tesi.

Nel capitolo 2 si presenta il contesto in cui si sviluppa l'idea e la necessità di un gestore di dati personali. Viene descritto un esempio di implementazione dei principi di MyData nell'ambito Smart Mobility.

Nel capitolo 3 si presenta l'architettura di *MyData* ed i principi che ne sono alla base. Questi sono il punto di partenza considerati per lo studio di fattibilità e l'implementazione del progetto. All'interno delle diverse sezioni si considerano ad alto livello le soluzioni implementative proposte dal team di sviluppo MyData.

Il capitolo 4 è dedicato all'analisi del problema. Si individuano le problematiche poste dalla realizzazione del gestore di dati personali e le soluzioni più adatte, mettendo in evidenza le entità in gioco e le relazioni fra di esse.

La fase di realizzazione viene trattata nel capitolo 5. Qui verranno mostrate concretamente le scelte implementative derivate dalle considerazioni fatte nei capitoli precedenti.

Infine, nel capitolo 6 si mostreranno le conclusioni e gli sviluppi futuri del lavoro di tesi.



## Capitolo 2

# Contesto di sviluppo

[12]

### 2.1 Mobility As A Service

*MyData*, grazie al coinvolgimento del Ministero dei Trasporti, ha avuto un primo banco di prova all'interno del concetto di Mobility As A Service. Il progetto finlandese ha ricevuto un notevole slancio grazie al lavoro di tesi svolto da Sonja Heikkilä[23], nella quale l'autrice propone un nuovo concetto di mobilità per la città di Helsinki per rispondere alle sfide sempre più frequenti poste al settore dei trasporti. Le principali difficoltà risiedono nell'inaffidabilità dei mezzi pubblici, unico modo per raggiungere determinate mete, comparata con la difficoltà ad usare l'auto di proprietà, causa mancanza parcheggi o rischio di ingorghi durante il viaggio.

Il paradigma MaaS descrive un nuovo utilizzo delle tecnologie applicato ai mezzi di trasporto, che propone il passaggio dall'auto di proprietà a mezzi di trasporto condivisi. Non si tratta solo di una migliore gestione dei mezzi pubblici, che proponga ad esempio soluzioni di pagamento online o potenziamento delle corse in base alla richiesta: Mobility As A Service si applica anche ai taxi, biciclette o sistemi di car sharing.

Questo cambiamento consentirebbe di aumentare l'efficienza nell'uso dei mezzi di trasporto, eliminando gli sprechi che inevitabilmente derivano dal possesso di un autoveicolo e dal suo inutilizzo. L'accesso del privato ai mezzi di trasporto avverrebbe, in questa nuova ottica, attraverso un software in grado di calcolare una ottimizzazione per i mezzi condivisi, ad esempio raggruppando gli utenti per fasce orarie, tratte comuni e generiche preferenze. Grazie inoltre ad una gestione comune dei mezzi di trasporto diversificati fra loro, la pianificazione del viaggio

può comprendere tratte percorse in modalità diverse (ad esempio automobile e bicicletta).

Da questa proposta è nata MaaS Global[10], una startup (?) finlandese che, durante l'autunno 2016, rilascerà l'app per smartphone "Whim", con la quale sarà possibile muoversi all'interno della città di Helsinki secondo il paradigma Mobility As A Service. Inizialmente, si potrà fare uso di mezzi di trasporto quali il trasporto pubblico urbano, i taxi e le auto a noleggio, ma saranno presto integrati anche i servizi di bike e car sharing.

### 2.1.1 Mobility Profile e Journey Planner

Come proof of concept, sono state realizzate dall'università di Aalto due applicazioni, rilasciate su piattaforma Android e iOS, Mobility Profile [11] e Journey Planner [9].

Mobility Profile raccoglie i dati personali dell'utente, li mantiene in un database relazionale e svolge le operazioni di calcolo del prossimo viaggio più probabile. Questa applicazione funziona come base di appoggio per Journey Planner, che riceve i suggerimenti calcolati e restituisce un feedback al processo sottostante, in modo da migliorarne l'accuratezza. Le API di comunicazione fra le due sono: `requestSuggestions()`, `requestTransportModePreferences()`, e `sendSearchedRoute(Place startLocation, Place destination)`.

In questo esempio, Mobility Profile non rispetta pienamente le specifiche dettate per *MyData*, ma è comunque possibile riconoscerne alcune caratteristiche all'interno del progetto. Ne sono esempio la richiesta esplicita di un permesso (revocabile in ogni momento) per l'utilizzo di dati personali, come gli impegni del calendario e lo storico delle posizioni GPS, e anche lo sviluppo di una applicazione separata per l'utilizzo dei risultati (Journey Planner) rispetto a quella che raccoglie i dati e li processa.

## 2.2 Smart Mobility For All

Rimanendo sul filone della mobilità intelligente, il progetto Smart Mobility for All [20] si sviluppa in modo indipendente da MaaS.

L'idea centrale su cui si fonda la piattaforma SMALL è la costruzione di un insieme di servizi di mobilità, potenzialmente anche per mezzi di trasporto molto diversi fra loro, che gestisca l'intero ciclo di vita di un viaggio, dalla prenotazione all'effettivo spostamento e all'arrivo a destinazione. Si tratta di Smart Mobility poiché attraverso la raccolta e l'analisi dei dati di un utente è possibile fare inferenze sugli spostamenti futuri, proponendo anche l'acquisto dei titoli di viaggio più adatti alle abitudini e alle necessità dell'individuo.

Ancora in via di sviluppo, si basa sul concetto di modularità che punta a fare di ogni servizio un modulo separato e indipendente. In questo modo si applica efficacemente il principio di suddivisione delle responsabilità e migliora la manutenibilità sia del sistema nel complesso che dei singoli servizi.

All'interno dei vari componenti dell'infrastruttura si potrebbe prevedere un gestore di dati personali del quale si porta, in questa tesi, un esempio semplificato. In questo contesto risulta evidente l'importanza della gestione dei dati personali, non solo in termini di protezione della privacy ma anche con uno sguardo all'interoperabilità e al corretto funzionamento di un insieme di servizi eterogenei.

## **2.3 GDPR?**

//è rilevante una sezione su questo tema?

## **2.4 Personal Clouds?**

//è rilevante una sezione su questo tema?





## Capitolo 3

# Architettura MyData

Al fine di comprendere appieno le scelte effettuate all'interno del progetto, si evidenziano brevemente le componenti e la struttura del modello *MyData* [16].

<https://docs.google.com/document/d/liUVRssJ6TmxzNZbw8XuMJ1A5oXetgthcHFGHYfCpPhw/edit>  
(broken?)

### 3.1 Entità Fondamentali

L'architettura di *MyData* si costruisce su quattro componenti base: l'utente finale, detto anche *Account Owner*, l'operatore *MyData*, o *Operator*, e due generiche entità che, da una parte, “producono” dati, e, dall'altra, li “consumano”. Esse sono definite rispettivamente *Source* e *Sink*. Mentre i ruoli di *Account Owner* e di *Operator* sono generalmente statici, quelli di *Source* e *Sink* sono fortemente variabili nel tempo e si possono applicare anche ad entità molto diverse fra loro, poiché definiti con un alto livello di astrazione. Convenzionalmente, si può identificare un servizio come “consumatore” di dati, mentre l'account dell'utente può essere un “produttore” di dati personali. In *MyData*, è possibile altresì che un servizio occupi entrambi i ruoli, o anche che l'*Operator* stesso rientri in questa classificazione quando si trova a compiere operazioni sui dati.

Il ruolo di *Operator* comprende operazioni di vario genere fra i quali vi sono la gestione degli utenti, dei servizi e delle interazioni che avvengono fra le due parti. Esso si occupa anche di gestire l'Audit Log di tutte le operazioni che coinvolgono tali interazioni.

### 3.2 Service Registry, Service Linking

Con *Service Registry* viene indicata quella parte dell'*Operator* che contiene un database di tutti i servizi registrati presso quell'operatore. Esso contiene anche

la funzionalità di *Service Discovery*, utilizzata dagli utenti per trovare nuovi servizi da utilizzare. In particolare, gli utenti possono venire a conoscenza di un nuovo servizio tramite un suggerimento, calcolato in base a corrispondenze fra caratteristiche dell'utente e del servizio, oppure tramite ricerca diretta.

Ogni nuovo servizio che vuole essere utilizzabile all'interno dell'architettura *MyData* deve quindi sottoporsi ad una procedura di registrazione al termine della quale, in caso di successo, viene inserito all'interno del *Registry*.

Durante questa procedura, il servizio deve fornire almeno una descrizione del suo comportamento in formato machine-readable e human-readable: la prima permette a procedure automatiche una elaborazione corretta di suggerimenti, la seconda è rivolta direttamente all'utente finale.

L'iscrizione di un utente presso un servizio avviene tramite un processo chiamato *Service Linking*, in cui l'Operatore *MyData* si occupa di realizzare una identificazione mutua delle parti. Tutti i token e le firme digitali scambiate durante il procedimento sono espresse in notazione JSON.

Al termine del *Service Linking* viene prodotto un *Service Link Record*, necessario per ogni futura interazione fra l'utente ed il servizio.

### 3.2.1 OAuth 2.0

//Da completare [17] <http://tutorials.jenkov.com/oauth2/authorization-code-request-response.html>

## 3.3 Autorizzazioni e Consent

Come specificato dal GDPR, ogni operazione svolta sui dati personali di un utente deve essere stata autorizzata dallo stesso tramite un permesso che acquista in questo contesto una valenza legale.

La funzione di un permesso, o Consent, è particolarmente rilevante: definisce quali dati possono essere utilizzati e in che modalità, e identifica le entità *Source* e *Sink* fra le quali avviene lo scambio. Il processo di *Service Linking* deve essere stato completato con successo affinché sia possibile fare richiesta di autorizzazione, e ciò viene verificato tramite ispezione del *Service Link Record*.

Il ruolo dell'*Operator* in questa situazione è quello di recuperare le informazioni corrispondenti al servizio presso il *Service Registry*: queste vengono presentate all'utente che decide se acconsentire o meno al processamento di un determinato insieme di dati da parte di un servizio specificato. È possibile per l'utente chiedere una ridefinizione delle richieste del servizio, ma non al di sotto del limite previsto per un corretto svolgimento del servizio stesso.

Nel caso la procedura di autorizzazione si concluda con successo, viene prodotto un *Consent Record* che contiene tutte le specifiche negoziate fra le parti insieme agli identificatori dell'utente e del servizio. Esso viene memorizzato all'interno dell'account, ma è possibile che il servizio o l'Operatore ne facciano richiesta successivamente.

Per dare la possibilità all'utente di ritirare il permesso accordato, un Consent ha tre stati possibili: *Active*, *Disabled* e *Withdrawn*. Il primo è lo stato standard di funzionamento, in cui l'accesso ai dati è consentito; si hanno poi gli stati "disabilitato" e "ritirato", in cui l'accesso è impedito. Nel caso in cui il permesso sia stato ritirato è necessario provvedere all'emissione di una nuova autorizzazione, mentre in caso di stato disabilitato, è possibile attuare un cambiamento di stato, riportandolo al valore attivo.

Questo protocollo di autorizzazione all'utilizzo dei dati rispetta quindi quanto affermato dal GDPR, poiché il permesso viene dato volontariamente e in modo chiaro. Non è ambiguo, è informato, grazie alla specifica delle risorse necessarie, ed è possibile ritirarlo in ogni momento.

### **3.3.1 Kantara Consent & Information Sharing Work Group**

//Da completare  
[1]

### **3.3.2 User Managed Access**

[21]

## **3.4 Personal Data Storage**

Nonostante non venga dato un peso rilevante a questa componente all'interno dei documenti di *MyData*, non è possibile prescindere dall'esistenza di un database che mantenga tutti i dati relativi ad un *Account Owner*. Non si tratta infatti solo di dati personali, ma di ogni dato utilizzato da un generico servizio o ad esempio inserito volontariamente dall'utente.

Non è specificato se il Personal Data Storage (spesso chiamato anche Personal Data Vault) faccia parte dell'ecosistema dell'operatore: ciò è possibile, ma non sono da escludere implementazioni alternative che prevedono il salvataggio delle informazioni presso il dispositivo dell'utente.

L'accesso ai dati contenuti all'interno del Personal Data Storage è possibile solo in presenza di un adeguato *Consent Record*, ma le modalità di accesso non vengono regolamentate in maniera dettagliata. Si parla infatti genericamente

di “Data API”, con le quali un *Sink* ottiene (in caso di richiesta legittima) un determinato *Resource Set* da un *Source*.

## Capitolo 4

# Analisi e Design

Il progetto del gestore di dati personali è cominciato con un'analisi delle specifiche *MyData* all'interno di uno studio di fattibilità. Le soluzioni adottate sono generalmente di portata “enterprise” e come tali inadatte ad un progetto di tesi. Per questo motivo ho cercato di seguire, durante il progetto, le stesse linee guida e i principi posti alla base di *MyData* preferendo, quando possibile, una implementazione più semplice e adatta al contesto.

Da quanto emerso nei paragrafi precedenti, si rende necessaria la presenza dei componenti di alto livello per la realizzazione del gestore di dati personali. Questi sono:

- Un utente finale che disponga di un account presso l'Operatore *MyData* e di un account presso il servizio di calcolo del prossimo viaggio più probabile;
- Il servizio che realizzi la logica di business, ad esempio un servizio che calcola il prossimo viaggio più probabile (*Most Likely Next Trip*);
- L'Operatore *MyData*;
- Un *Service Registry*, presso cui il servizio è registrato;
- Un gestore dei permessi per l'utilizzo del servizio e l'accesso ai dati dell'utente;
- Un Personal Data Vault per la memorizzazione dei dati.

### 4.1 Accounting e servizi

[13]

Al fine di gestire gli utenti all'interno del progetto, sono stati previsti due diversi tipi di account. Il primo si ottiene per mezzo dell'iscrizione dell'utente

finale al servizio *MyData*. Ciò richiede l’inserimento obbligatorio di alcuni dati anagrafici come nome, cognome e data di nascita, che è possibile tuttavia modificare e arricchire in seguito. Questo tipo di account permette di associare ad ogni utente un Personal Data Vault, e costituisce punto di riferimento per tutti i singoli account creati presso i servizi. Non è possibile creare account duplicati.

Il secondo tipo di account fa riferimento alla registrazione che avviene al primo utilizzo di un servizio. Nel caso considerato, ad esempio, l’utente creerà un account specifico per il servizio *Most Likely Next Trip*, associato al suo account presso *MyData*; pertanto, ogni utente potrà avere un numero di account specifici pari ai servizi presso cui si è registrato. Inoltre, viene mantenuto l’elenco di tutti i permessi che l’utente ha generato per il servizio corrispondente.

Per quanto riguarda il servizio *Most Likely Next Trip*, si è scelto di utilizzare un componente già pronto, sviluppato da Nicola Ferroni nel suo lavoro di tesi “Un sistema di previsione degli itinerari per applicazioni di smart mobility”[22]. All’interno di questo progetto, si svilupperà un esempio di come sia possibile far interagire le due entità, regolando lo scambio di dati in modo quanto più possibile aderente al modello *MyData*.

## 4.2 Operatore MyData

[14] La realizzazione di un completo *MyData Operator* comporta la costruzione di diversi componenti. Le funzioni dell’Operatore comprendono la gestione del *Service Registry*, degli account utente e del processo di *Service Linking*. L’Operatore deve inoltre occuparsi della reciproca identificazione fra le parti nel processo di autorizzazione del servizio, durante la firma dei *Consent*, e autorizzare il flusso di dati quando ne viene fatta richiesta.

In questo lavoro si è preferito suddividere le responsabilità corrispondenti a queste operazioni in una molteplicità di classi, invece che un’unica entità, in conformità a principi di semplicità come il rasoio di Occam o l’acronimo KISS (Keep It Simple, Stupid). Pertanto, si realizzeranno separatamente:

- Un gestore di *Consent*, che si occupi dell’emissione dei permessi e del cambio di stato degli stessi in caso di richiesta da parte dell’utente.
- Un *Service Registry* semplificato, dove ogni servizio si registra per poter essere accessibile all’interno di *MyData*.
- Un’entità che metta in comunicazione il Personal Data Storage con il servizio, recuperando i dati necessari (punto di *policy enforcement*).
- Un gestore per le operazioni di autenticazione fra utente e servizio.

Per questo motivo, si sceglie nel seguito del progetto di non citare più il concetto di *Operator*, facendo invece riferimento alle entità particolari.

### 4.3 Service Registry, Service Linking

Una implementazione proposta dal team di sviluppo finlandese può essere trovata all'indirizzo [15].

Inizialmente non era stata prevista alcuna implementazione di *Service Registry*, principalmente a causa della complessità non solo della struttura del componente, ma anche delle operazioni di registrazione di un nuovo servizio e di *Service Discovery*.

Come verrà descritto meglio nel capitolo 5 dedicata alla Progettazione, questo componente si è rivelato indispensabile per il funzionamento del sistema, ma la sua complessità è stata notevolmente ridotta. Infatti, si è scelto di non includere nessuna delle due descrizioni del servizio menzionate nelle specifiche, ma solo una indicazione sui tipi di dato necessari al suo funzionamento.

Questa scelta ha avuto una ripercussione anche sul protocollo di *Service Linking*, che ha perso di significato in mancanza dell'entità *Service Registry*. Tuttavia, vista la sua importanza concettuale, ho scelto di proporla comunque una variante all'interno del meccanismo delle autorizzazioni e del *Consent*.

### 4.4 Consent

A questo punto, si presenta la necessità di realizzare due tipi diversi di permessi.

Il primo contiene la semantica associata al procedimento di *Service Linking*: descrive quali sono le entità in gioco, fornendone una mutua autenticazione e contiene uno stato per indicare la validità del permesso considerato. I valori possibili di questo stato ricalcano quelli indicati nel modello *MyData* (sezione 3.3): *Active*, *Disabled*, *Withdrawn*. Questo *Consent* viene emesso ogni volta che l'utente fa richiesta di utilizzare un servizio, e in base al valore del suo stato permette (o meno) l'effettivo processamento dei dati.

Il secondo tipo di permesso viene utilizzato come token di accesso al Personal Data Storage ogni volta che si instaura un flusso di dati, sia esso in ingresso o in uscita dal PDS. Questo tipo di *Consent* contiene un riferimento ai tipi di dato scambiati durante la transazione, in modo da poter verificare se essi coincidano con quelli dichiarati dal servizio in fase di registrazione. Inoltre, può essere utilizzato una volta sola ed è possibile ottenerlo solo se il permesso che lega l'utente ed il servizio corrente ha stato attivo.

Entrambi i *Consent* vengono memorizzati all'interno dell'account utente corrispondente al servizio che ne ha fatto richiesta.

## 4.5 Personal Data Storage

L'ipotesi iniziale per il Personal Data Storage era quella di realizzare un componente indipendente dalle scelte dell'utente o dalle necessità del servizio, come anche dalle scelte implementative.

Un esempio di indipendenza dalle scelte dell'utente è il seguente. Il servizio *Most Likely Next Trip* utilizza al suo interno i dati di un calendario, secondo l'assunto per cui l'utente deve compiere un viaggio per portarsi nel luogo dell'evento inserito. La soluzione più immediata è quella di salvare all'interno del PDS gli impegni necessari, o la struttura dati del calendario in un file specifico per l'applicazione di Calendario utilizzata dall'utente. Questa scelta però non solo è poco efficiente, ma vincola all'utilizzo di un particolare tipo di calendario, impedendo successive modifiche o estensioni. Una soluzione alternativa potrebbe essere quella di utilizzare uno standard di rappresentazione di calendari, attualmente il formato *.ics*, anche se è ancora diffusa la versione precedente, *.vcs*. Questa scelta, seppure legata ancora ad un tipo di implementazione a file, offre maggiore interoperabilità grazie alla scelta di un formato supportato anche da servizi esterni.

Alternativamente, è possibile astrarre ad un livello ancora maggiore l'implementazione del calendario, ad esempio utilizzando le API di servizi online (come quelle di Google Calendar[3]) per integrare quelli inseriti nel Personal Data Storage. Attraverso l'utilizzo di molteplici livelli di astrazione sarebbe possibile soddisfare la richiesta dell'utente di utilizzare uno specifico calendario, senza però cablarne questa scelta nell'implementazione.

In ultima analisi, anche a causa di una implementazione già presente all'interno del progetto *Most Likely Next Trip* si è scelto di adottare una persistenza che faccia uso di file di testo. Tuttavia, attraverso l'uso di interfacce ho cercato di non propagare le dipendenze dall'implementazione scelta all'esterno della classe, in modo da permettere un più semplice refactoring in caso di sviluppi futuri.

## 4.6 Rappresentazione di dati non noti a priori

Una delle maggiori difficoltà incontrate durante lo studio di *Service Registry* e Personal Data Storage riguardava la necessità di descrivere tipi di dato non noti a priori, e di possedere strumenti per il loro processamento. Questa problematica interessa il *Service Registry* al momento dell'iscrizione del servizio presso *MyData*



e il Personal Data Storage quando questo prende parte ad uno scambio di dati con il servizio.

Le specifiche del modello *MyData* consentono di ottenere l'indipendenza da tipi di dato specifici tramite l'utilizzo linguaggi che descrivono la tassonomia e le strutture dati necessarie al funzionamento del servizio. In particolare, viene fatto uso di RDF (Resource Description Framework), che permette la rappresentazione di informazioni all'interno del web [18].

Alcuni esempi utilizzati in *MyData* sono W3C's Data Catalog Vocabulary[2] e The RDF Data Cube Vocabulary[19], utilizzati all'interno del *Service Registry* nella descrizione del servizio.

Per ottenere l'indipendenza e l'interoperabilità al centro del modello di *MyData*, utilizzando allo stesso tempo una soluzione più adatta ad un contesto ridotto, ho preso considerato la possibilità di trasmettere i dati necessari in linguaggio JSON.



## Capitolo 5

# Progettazione

Nei paragrafi successivi sono illustrate le fasi di implementazione del gestore di dati personali, motivando le scelte implementative e le eventuali differenze che si sono verificate rispetto a quanto detto in Analisi.

Per la realizzazione del gestore è stato utilizzato il linguaggio Java[7] [8].

Fra i principi generali seguiti in Progettazione troviamo l'inversione delle dipendenze, la separazione delle responsabilità, il principio di sostituibilità di Liskov e il già citato rasoio di Occam. Secondo il principio di inversione delle dipendenze, è necessario che le dipendenze presenti all'interno del codice non siano fra classi ma fra interfacce, in modo da evitare che la struttura possa risentire di cambiamenti che avvengono a basso livello. Il principio di separazione delle responsabilità stabilisce che a ogni classe è attribuito un solo compito, da svolgere e completare interamente, ma mai più di uno: lo sviluppo di classi aventi più responsabilità genera dipendenze non volute fra le classi, rendendo il codice fragile. Il principio di sostituibilità di Liskov, infine, si applica ai casi di ereditarietà fra classi e ne regola il rapporto: ogni sottoclasse deve poter essere utilizzata al posto della classe base senza che sia evidenziata la differenza.

### 5.1 Flusso del programma

inserire grafico

### 5.2 Accounting

Al centro è collocata la classe corrispondente all'utente *MyData* che, confermando quanto detto in Analisi, è collegata agli account dei servizi e al **PersonalData-Vault** dell'utente. Una novità è invece rappresentata dalla coppia **ISecurity-Manager**, **SecurityManager** creata per soddisfare i requisiti di sicurezza relativi

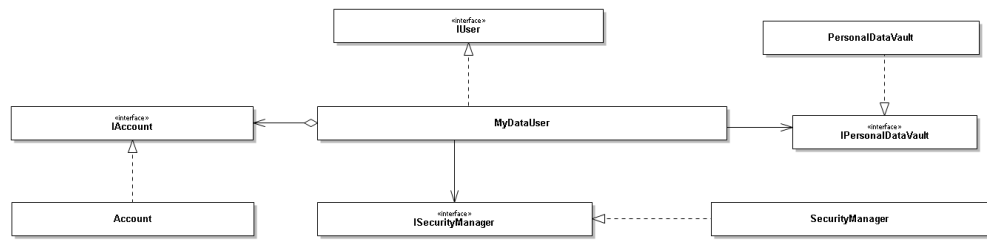


Figura 5.1: caption

alla mutua autenticazione fra utente e servizio. Si è deciso di sviluppare separatamente questa classe per un principio di separazione delle responsabilità e per consentire una estendibilità più semplice in caso di sviluppi futuri.

### 5.2.1 IUser, MyDataUser

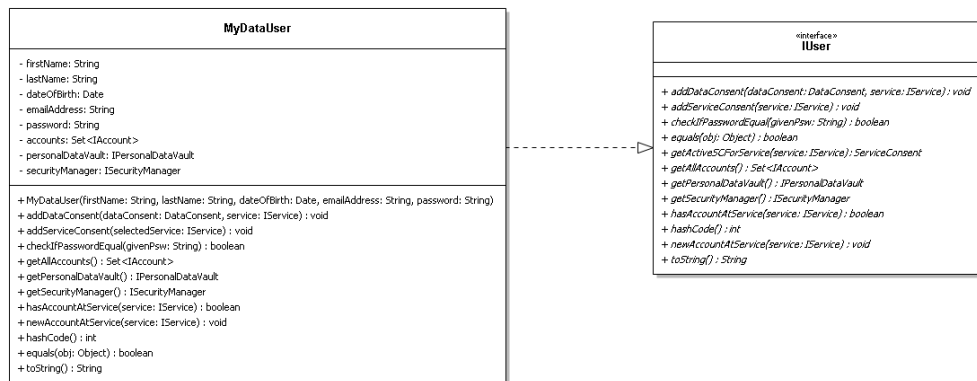


Figura 5.2: caption

Questa classe modella un generico utente dell'architettura *MyData*. I field al suo interno sono un esempio delle caratteristiche che si è scelto di modellare e fra essi i più rilevanti sono indirizzo email e password in quanto permettono il login per utenti già registrati. L'indirizzo mail è stato adottato, inoltre, come identificatore unico di un utente all'interno di *MyData* e questa caratteristica è stata implementata mediante l'override della funzione `equals(Object obj)`.

Si evidenzia inoltre la presenza di un `Set<IAccount> accounts` che realizza l'associazione fra un utente e gli account presso i servizi a cui si è registrato. La scelta di un `Set` permette di implementare il vincolo secondo cui ogni utente può avere un solo account presso un certo servizio ed è efficace anche perché è superfluo mantenere un insieme ordinato di account.

Questa classe, inoltre, ha la funzione di interfacciare gli altri componenti del gestore, compresa la GUI, con gli account utente. A tal fine presenta i metodi `addServiceConsent (IService service)`, `addDataConsent (DataConsent dataConsent, IService service)`, `hasAccountAtService (IService service)`. La classe `Account`, infatti, è stata modellata con visibilità package protected per impedire l'accesso a classi esterne al package `users`: di conseguenza, anche la creazione di nuovi account avviene attraverso questa classe, in particolare nella funzione `newAccountAtService (IService service)`. All'interno del metodo troviamo l'istanziatura di un nuovo account insieme ad una chiamata alla classe `ConsentManager` che realizza quanto anticipato al paragrafo 4.4 a pagina 23. In questo modo si ottiene un esempio di *Service Linking* e l'esito di questa operazione viene concretizzato in un oggetto `ServiceConsent`. Si rimandano però ulteriori dettagli a quanto evidenziato nella sezione 5.5 a pagina 32.

### 5.2.2 IAccount, Account

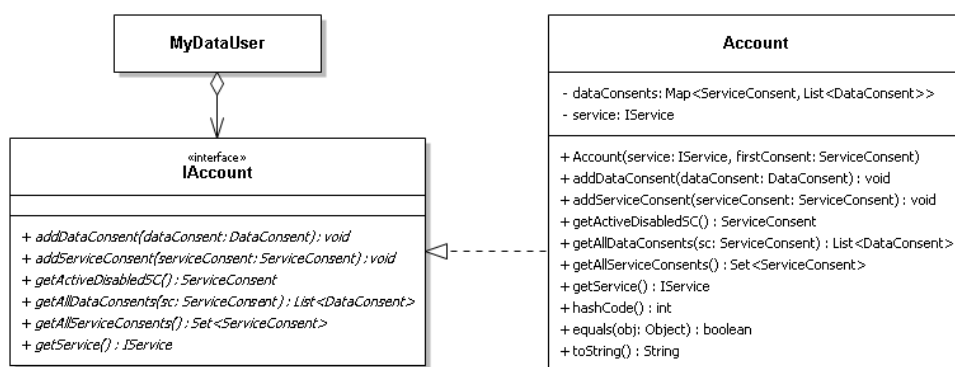


Figura 5.3: caption

La classe `Account` è abbastanza semplice, poiché si occupa semplicemente di implementare la logica di basso livello nelle operazioni di gestione degli account.

Fra queste vi sono i controlli sullo stato dei Consent memorizzati, la gestione dello storico di tutti i Consent emessi per quel servizio `service`, o ancora il matching fra i due tipi di Consent (`ServiceConsent`, `DataConsent`, dettagliati al paragrafo 5.5.2 a pagina 33).

La memorizzazione dei Consent all'interno della classe è stata ottenuta mediante l'utilizzo combinato delle strutture dati `Map<ServiceConsent, List<DataConsent>`. Questa modalità permette di esprimere diversi concetti a livello semantico. Come prima cosa, per i Consent sul flusso di dati si è scelto di utilizzare la classe base `DataConsent` invece che le sue due implementazioni, in modo da

poterli memorizzare indiscriminatamente. Ciò verifica l'utilizzo del principio di sostituibilità di Liskov. Inoltre, la scelta di una **List** come value all'interno di una mappa permette di descrivere un flusso di dati (all'interno dello stesso **ServiceConsent**) per il quale si sono rivelate necessarie una molteplicità di interazioni fra *Source* e *Sink*, ognuna delle quali modellata da un **DataConsent**.

Ad ogni istanza di **ServiceConsent** corrisponde quindi una **Collection** dei **DataConsent** emessi durante il periodo di validità dello stesso, ed è possibile avere un unico **ServiceConsent** attivo in un determinato istante di tempo.

### 5.3 SecurityManager

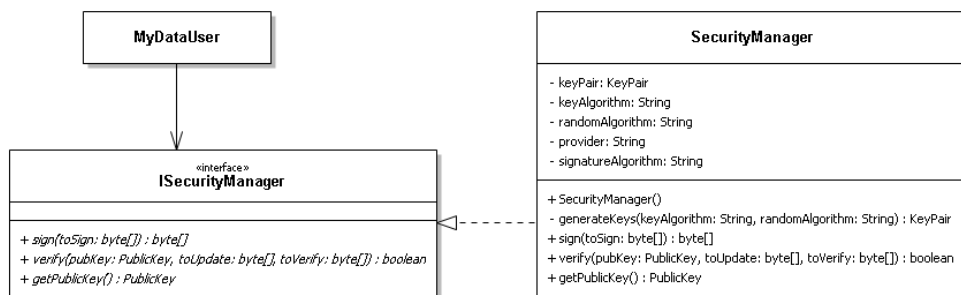


Figura 5.4: caption

Come anticipato nella sezione 5.2, si è resa necessaria l'implementazione di una entità che garantisca il rispetto di alcune politiche di sicurezza.

In particolare, in questa implementazione si è scelto di dare maggiore rilevanza all'aspetto di reciproca autenticazione fra utente e servizio, piuttosto che ad altre problematiche come ad esempio memorizzazione e trasmissione sicura dei dati. A tal fine, si è scelto di utilizzare alcuni componenti già pronti all'interno dell'infrastruttura Java, in particolare all'interno della Java Cryptography Architecture [6].

Per realizzare un protocollo di sfida e risposta, la classe **SecurityManager** contiene una coppia di chiavi **KeyPair**, insieme ad alcune costanti che rappresentano gli algoritmi ed il provider scelto per l'implementazione degli stessi.

L'interfaccia **ISecurityManager** ha la funzione di astrarre dalla particolare implementazione (ogni servizio potrebbe ad esempio preferire una implementazione specifica), e pertanto espone solamente i metodi di firma e verifica necessari al completamento dell'operazione di autenticazione.

## 5.4 IMyData, MyData

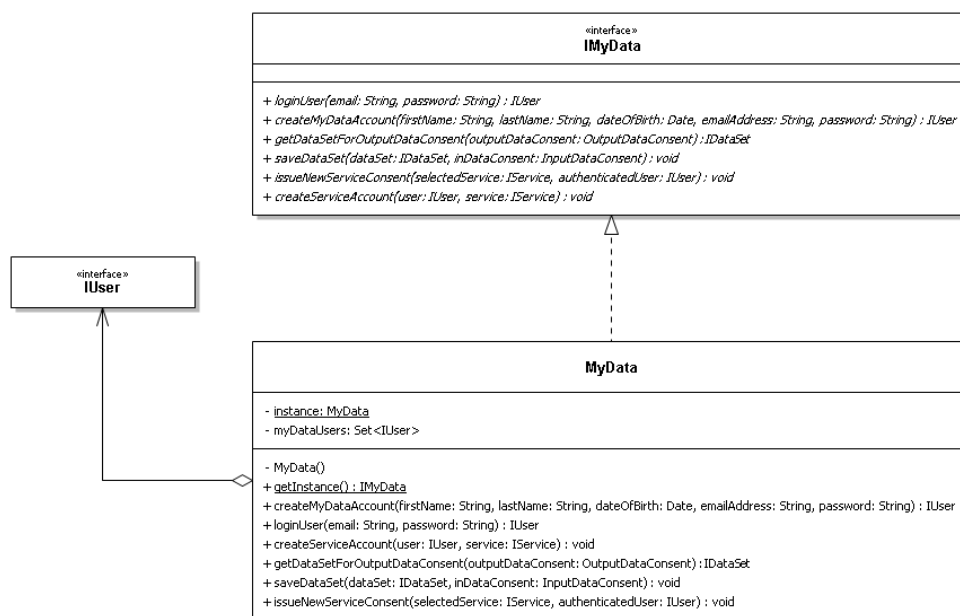


Figura 5.5: caption

La classe **MyData** svolge all'interno del gestore di dati personali un importante ruolo di coordinazione fra le parti, poiché realizza al suo interno una parte dell'Operatore *MyData*, secondo quanto specificato in 4.2. Essa è un punto di riferimento per l'interfaccia utente, alla quale fornisce i dati da elaborare e mostrare a video, e dalla quale riceve le richieste effettuate dall'utente.

Si occupa quindi come prima cosa di registrare e autenticare gli utenti (metodi **createMyDataAccount** e **loginUser**) in modo da impedire la creazione di duplicati. I controlli in questo senso vengono effettuati su un **HashSet<IUser>** contenuto all'interno della classe (si sfrutta la proprietà della struttura dati **Set** di non ammettere duplicati). La creazione di un nuovo account presso un determinato servizio viene gestita da questa classe, tramite invocazione dell'opportuno metodo esposto dall'interfaccia **IUser**, insieme alla richiesta di nuovi **ServiceConsent** in caso di utente già registrato.

In secondo luogo, ogni servizio che ha richiesto e ottenuto il permesso di accedere a uno specifico insieme di dati personali di un utente fa riferimento alla classe **MyData**, che si occupa di intercedere presso il Personal Data Vault per ottenere quanto richiesto. L'operazione si svolge sia per i dati in ingresso che per i dati in uscita dal Vault mediante i metodi **getDataSetForOutputDataConsent** (Out-

`putDataConsent outputDataConsent`) e `saveDataSet (IDataSet dataSet, InputDataConsent inputDataConsent)`. In entrambi i casi, viene controllata la validità del Consent emesso prima di effettuare la richiesta di dati personali.

Infine, si evidenzia la realizzazione della classe `MyData` come Singleton mediante l'utilizzo di un costruttore privato e di un campo `instance` di tipo `MyData`. Ciò assicura la presenza di un unico Operatore di questo tipo all'interno del programma.

## 5.5 Autorizzazioni e Consent

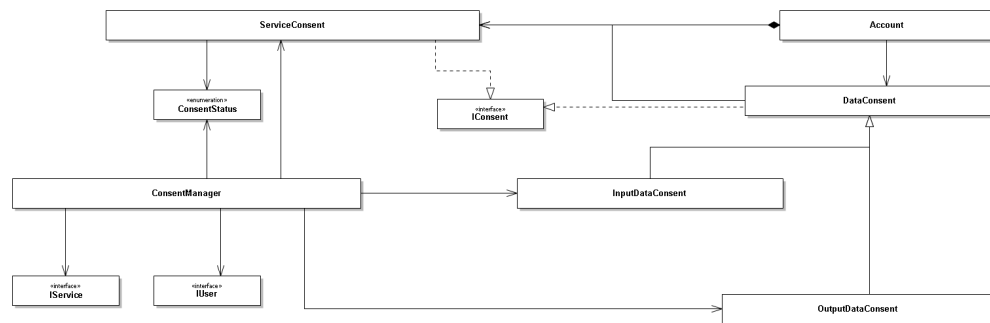


Figura 5.6: caption

All'interno dell'architettura realizzata per la gestione delle autorizzazioni e dei permessi è possibile individuare alcune entità fondamentali: la classe `ConsentManager`, che realizza il gestore di permessi previsto nella sezione 4.2, e le due tipologie di permessi, anch'esse previste in fase di Analisi (sezione 4.4).

È presente infine anche l'enumerativo `ConsentStatus`, attraverso il quale si rappresentano gli stati del rapporto fra un utente ed un servizio generici.

### 5.5.1 ConsentManager, ConsentStatus

La classe `ConsentManager` si occupa dell'erogazione, in caso di richiesta legittima, di vari tipi di permessi ai servizi che ne fanno richiesta. Poiché il suo scopo è quello di garantire il rispetto di un determinato protocollo di assegnazione dei permessi, essa è prima di tutto una classe implementativa e per questo motivo non è previsto alcun tipo di astrazione (ad esempio tramite interfaccia).

Si potrebbe considerare di rendere la classe `final`, per impedire estensioni o ridefinizioni del comportamento. Le motivazioni a supporto di questa scelta risiedono nella garanzia di una maggiore sicurezza. Tuttavia, nel complesso ciò porterebbe ad una eccessiva rigidità del codice, impedendo aggiornamenti del protocollo anche in casi di legittima necessità.



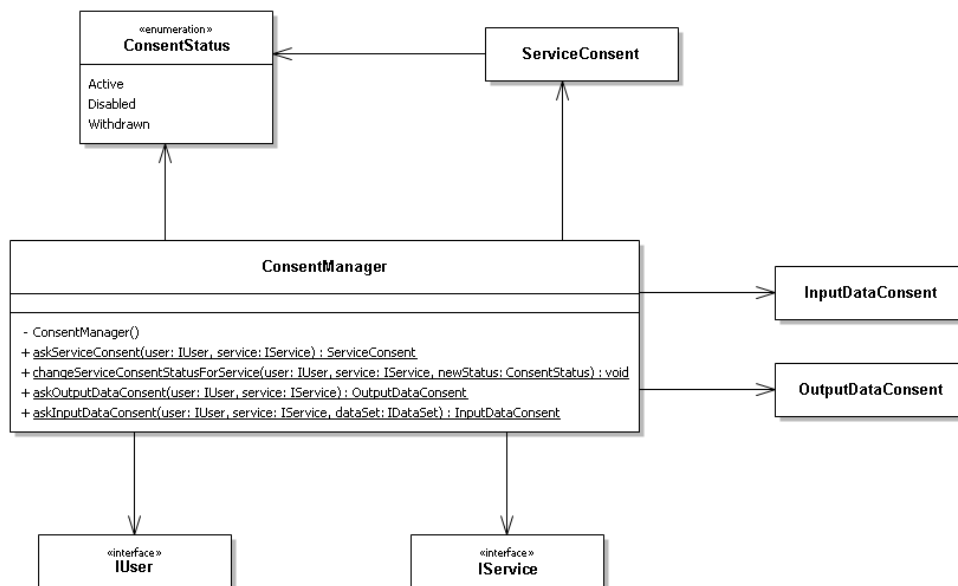


Figura 5.7: caption

Per la sua implementazione, ho cercato di fare in modo che essa realizzasse un servizio il più possibile indipendente dal resto dell'architettura circostante. Pertanto, la classe **ConsentManager** non mantiene alcuno stato interno, presenta costruttore privato e non ha dipendenze rilevanti. Inoltre, poiché la procedura di verifica dei requisiti è costante e indipendente dai parametri di ingresso, ogni metodo è stato realizzato come `static`.

I tipi di Consent erogati dalla classe **ConsentManager** sono **ServiceConsent**, **InputDataConsent** e **OutputDataConsent**, ognuno con un metodo dedicato. È presente anche la procedura `changeServiceConsentStatusForService` (`IUser user, IService service, ConsentStatus newStatus`), che permette all'utente di cambiare lo stato del **ServiceConsent** correntemente attivo o disabilitato, secondo quanto previsto dalle specifiche di *MyData* e successivamente in fase di Analisi (sezione 4.4).

### 5.5.2 ServiceConsent, DataConsent

I permessi utilizzati all'interno del gestore di dati personali ed erogati dalla classe **ConsentManager** sono istanze delle classi **ServiceConsent**, **InputDataConsent** e **OutputDataConsent**. Come è possibile osservare dal diagramma UML in figura 5.8, **InputDataConsent** e **OutputDataConsent** estendono la classe **DataConsent**: la loro funzione è principalmente semantica, in quanto non aggiungono logica

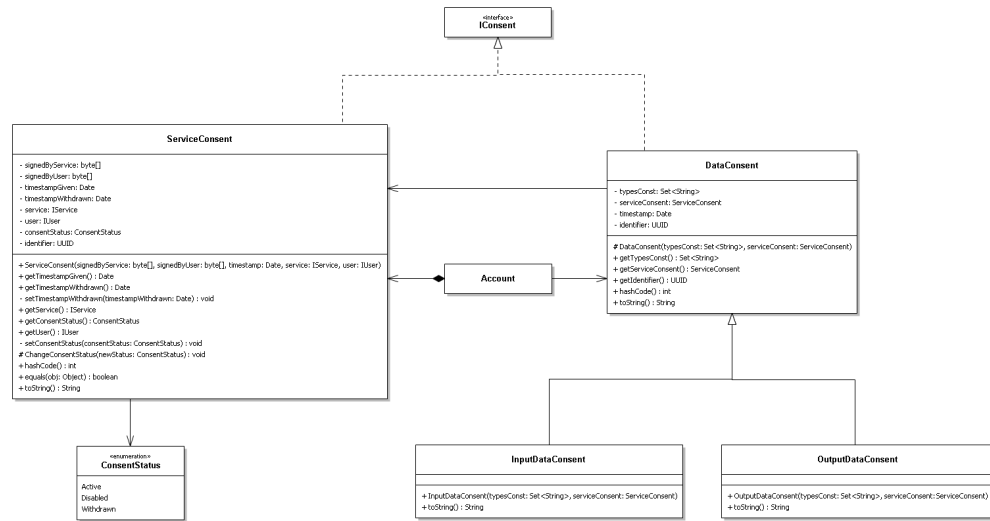


Figura 5.8: caption

al programma ma descrivono il verso del flusso di dati che si crea con il Personal Data Vault. Pertanto, descriverò principalmente le caratteristiche delle classi **ServiceConsent** e **DataConsent**, che costituiscono il punto focale della realizzazione dei permessi descritti in *MyData*.

Nonostante la differenza di realizzazione e di utilizzo, entrambe le classi **ServiceConsent** e **DataConsent** implementano una interfaccia comune **IConsent**. Questa è una interfaccia *marker* necessaria per esprimere una somiglianza a livello semantico, in quanto entrambe le classi descrivono un tipo di autorizzazione.

Dal diagramma UML è possibile dedurre il ruolo della classe **Account** rispetto ai due tipi di Consent. Come accennato infatti in 5.2.2, essa mantiene al suo interno una mappa di corrispondenze fra **ServiceConsent** e liste di **DataConsent**.

Nel primo caso, la relazione è rappresentata mediante il simbolo “rombo nero”, che qualifica la classe **Account** come “contenitore” di istanze della classe **ServiceConsent**. In particolare, il rombo nero descrive un tipo di relazione molto stretta fra le due parti, e la scelta è dovuta alle specifiche di *MyData*, secondo cui non è possibile registrarsi presso un servizio senza ottenere un Consent. Ciò è stato implementato mediante l’emissione di un **ServiceConsent** prima della creazione effettiva dell’account, e il legame fra i due avviene tramite il passaggio di questo permesso al costruttore della classe **Account**.

La classe **ServiceConsent** realizza il primo – e il più rilevante – dei due tipi di permessi previsti per il gestore di dati personali. Al suo interno troviamo i *token* firmati da utente e servizio per la mutua autenticazione, insieme ai rispettivi

riferimenti; vi sono inoltre anche alcuni campi per l'identificazione del Consent stesso e la sua collocazione temporale.

Per quanto riguarda invece i **DataConsent**, essi si comportano come *access token* per il Personal Data Vault validi una sola volta e conservati come storico dell'accesso ai dati personali. A tal fine, un **DataConsent** contiene al suo interno il **Set<String>** con l'elenco dei tipi di dato a cui il servizio beneficiario può accedere. Per impedire accessi illegittimi, viene sempre controllata la corrispondenza fra i tipi di dato dichiarati in fase di registrazione e quelli richiesti alla creazione del **DataConsent**. Poiché il servizio non può interrogare direttamente il Personal Data Vault, gli scambi avvengono in base a quanto dichiarato all'interno del **Set<String>**.

Infine, si evidenzia dal diagramma UML che ogni **DataConsent** mantiene un riferimento al **ServiceConsent** attivo al momento della sua emissione, al fine di avere una migliore tracciabilità delle transazioni di dati, e che il costruttore della stessa classe ha visibilità package - protected in modo da obbligare l'uso delle sottoclassi alle classi esterne.

## 5.6 PersonalDataVault

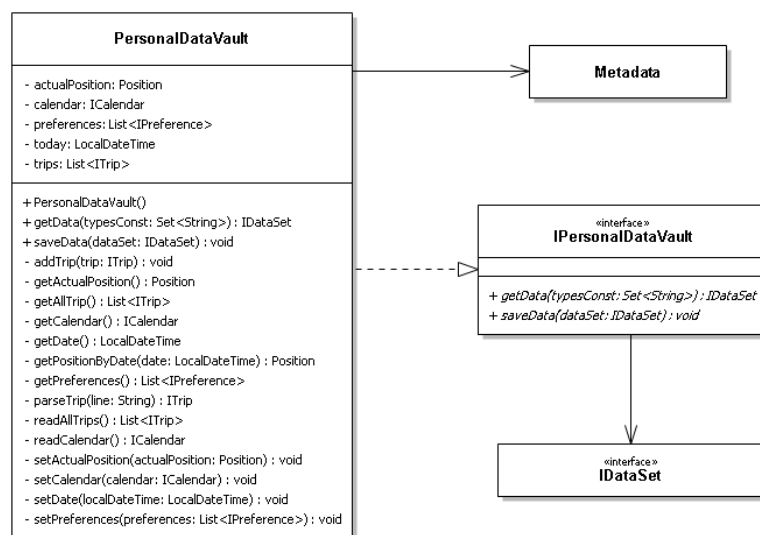


Figura 5.9: caption

In questa sezione si presenta la realizzazione del Personal Data Vault, in cui vengono mantenuti i dati personali dell'utente. L'interfaccia **IPersonalData-**

Vault espone i metodi `getData (Set<String> typesConst)` (che restituisce un `IDataSet`) e `saveData (IDataSet dataSet)` che permettono l'accesso ai dati.

La signature dei metodi garantisce l'indipendenza dai tipi di dato in ingresso o in uscita dal Vault grazie all'uso di oggetti di tipo `DataSet`, incapsulati in opportune interfacce `IDataSet`, il cui funzionamento è spiegato in dettaglio nella sezione 5.8. Inizialmente, in fase di Analisi, avevo preso in considerazione la possibilità di realizzare gli scambi di dati da e per il Personal Data Vault utilizzando il linguaggio JSON (come anticipato nella sezione 4.6). Questa scelta, dovuta alla volontà di realizzare un sistema interoperabile e aderente agli standard contemporanei, si è rivelata difficilmente realizzabile in fase di Progettazione.

In questo senso, come prima cosa ho studiato i componenti Java disponibili nell'architettura per realizzare la conversione in stringhe JSON: `JSONArray`, `JsonObject` e altre sotto-interfacce di `JsonValue` [5]. Il motivo per cui ho scelto di non utilizzarli è che questi non offrono alcun metodo di utilità per la conversione da oggetto a stringa, e la trasposizione di ogni field va realizzata manualmente sia in serializzazione che in deserializzazione. Una scelta di questo tipo implicherebbe un precedente accordo fra le parti per stabilire come interpretare le stringhe inviate, e una forte dipendenza dalla particolare implementazione delle classi.

Come seconda opzione ho considerato di utilizzare la libreria Google Gson [4], in quanto essa risolve i problemi incontrati nel corso del primo tentativo grazie ai metodi `gson.toJson(obj)` per la serializzazione e `gson.fromJson(json, obj.class)` per la deserializzazione. Questo approccio funziona perfettamente in caso di oggetti che contengono tipi primitivi, ma mostra qualche limitazione quando si introducono oggetti di tipo generico e Collection di oggetti, siano essi di oggetti di un unico tipo o di tipi diversi. Il motivo risiede nell'implementazione della Java Virtual Machine, e in particolare nella sua caratteristica di *Type Erasure* per la quale ogni oggetto a basso livello “perde” il suo tipo particolare per diventare un `Object`. Questo non crea problemi in serializzazione ma in deserializzazione, quando risulta impossibile recuperare il tipo originario dell'oggetto da deserializzare. Poiché l'utilizzo della libreria Gson nel contesto del `PersonalDataVault` si sarebbe collocato all'interno dei casi non completamente supportati, ho scelto di scartare anche questa seconda possibilità.

La soluzione da me proposta a questo problema consiste nell'utilizzo di una struttura dati `DataSet` che permetta lo scambio di dati di qualunque tipo all'interno del sistema, purché il loro tipo sia stato preventivamente dichiarato in fase di registrazione (sezione 5.9). La classe `DataSet` è descritta in dettaglio nella sezione 5.7.

All'interno della classe `PersonalDataVault` troviamo alcuni metodi privati necessari per la gestione dei dati memorizzati (ad esempio la lettura e la scrittura su file, con opportune funzioni di utility necessarie per il parsing). Le scelte

effettuate in questo senso derivano da una precedente implementazione e sono state adattate solo in minima parte per motivi di compatibilità.

Per quanto riguarda invece i metodi `getData (Set<String> typesConst)` e `saveData (IDataSet dataSet)`, essi si occupano di prelevare dati, o salvarli, in base a quanto specificato dal `Set<String>` contenente l'insieme dei tipi di dato.

## 5.7 Metadata, DataSet

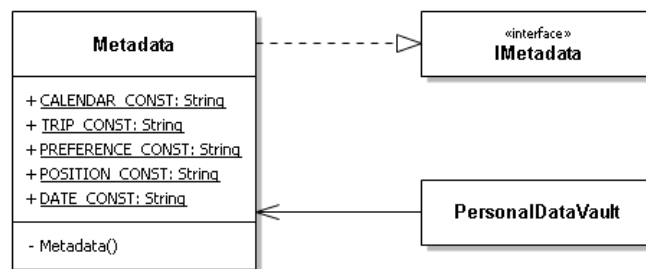


Figura 5.10: caption

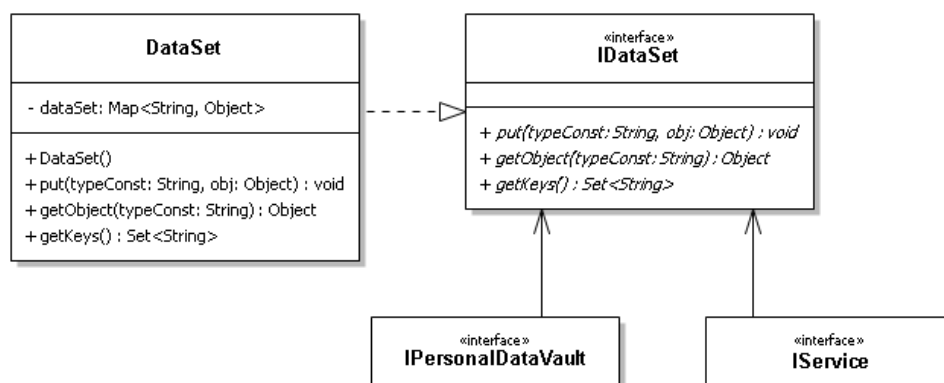


Figura 5.11: caption

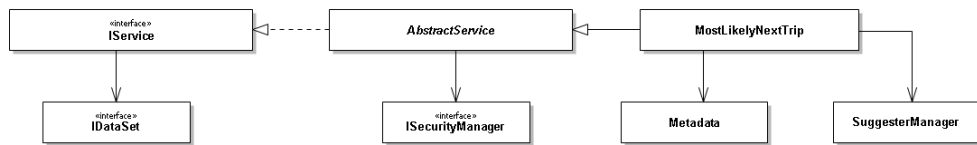


Figura 5.12: caption

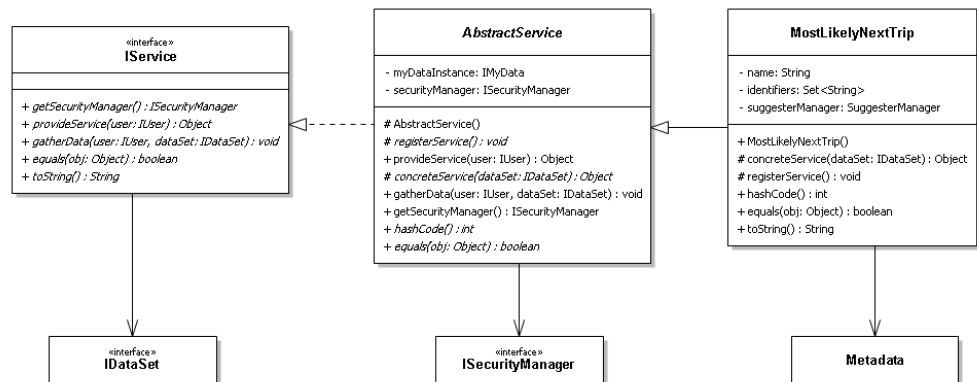


Figura 5.13: caption

## 5.8 IService, AbstractService, MostLikelyNextTrip

## 5.9 ServiceRegistry

## 5.10 Uso delle eccezioni

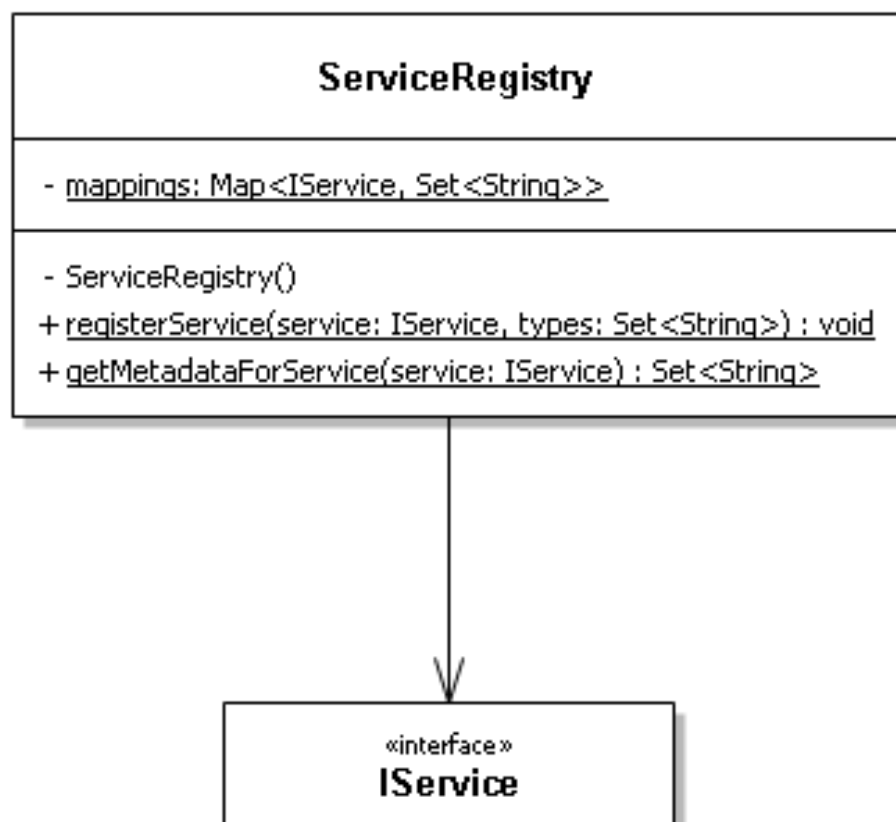


Figura 5.14: caption





## Capitolo 6

# Conclusioni e sviluppi futuri

placeholder.



# Bibliografia

- [1] Consent receipt specification by kantara consent & information sharing work group. Ultima visita Nov. 2016.
- [2] Data catalog vocabulary. Technical report. Ultima visita Nov. 2016.
- [3] Google calendar api at google developers. Technical report. Ultima visita Nov. 2016.
- [4] Google gson library on github. Technical report. Ultima visita Nov. 2016.
- [5] Java api specification - jsonobject. Technical report. Ultima visita Nov. 2016.
- [6] Java cryptography architecture (jca) reference guide. Technical report. Ultima visita Nov. 2016.
- [7] The java language specification. Technical report. Ultima visita Nov. 2016.
- [8] Java platform, standard edition 8 api specification. Technical report. Ultima visita Nov. 2016.
- [9] Journey planner repository on github. Technical report. Ultima visita Nov. 2016.
- [10] Maas global website. Ultima visita Nov. 2016.
- [11] Mobility profile repository on github. Technical report. Ultima visita Nov. 2016.
- [12] Mydata 2016. advancing human centric personal data. Ultima visita Nov. 2016.
- [13] Mydata account implementation on github. Technical report. Ultima visita Nov. 2016.

- [14] Mydata operator implementation on github. Technical report. Ultima visita Nov. 2016.
- [15] Mydata service registry implementation on github. Technical report. Ultima visita Nov. 2016.
- [16] Mydata specification on github. Technical report. Ultima visita Nov. 2016.
- [17] Oauth 2.0 website. Ultima visita Nov. 2016.
- [18] Rdf 1.1 concepts and abstract syntax. Technical report. Ultima visita Nov. 2016.
- [19] The rdf data cube vocabulary. Technical report. Ultima visita Nov. 2016.
- [20] Small documentation on github. Technical report. Ultima visita Nov. 2016.
- [21] User managed access. Ultima visita Nov. 2016.
- [22] Nicola Ferroni. Un sistema di previsione degli itinerari per applicazioni di smart mobility. 2016.
- [23] Sonja Heikkilä. Mobility as a service – a proposal for action for the public administration - case helsinki, 2014.
- [24] Antti Poikola, Kai Kuikkaniemi, and Harri Honko. Mydata – a nordic model for human-centered personal data management and processing. Technical report. Ultima visita Nov. 2016.
- [25] World Economic Forum with A. T. Kearney. Rethinking personal data: A new lens for strengthening trust. Technical report. Ultima visita Nov. 2016.