Specifica Tecnica

Gruppo SteakHolders — Progetto MaaP

Informazioni sul documento	
Versione	4.0.0
Redazione	Gianluca Donato
	Luca De Franceschi
Verifica	Enrico Rotundo
Approvazione	Federico Poli
Uso	Esterno
Distribuzione	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
	Gruppo SteakHolders
	CoffeeStrap

Descrizione

Questo documento descrive la specifica tecnica e l'architettura del prodotto sviluppato dal gruppo SteakHolders per la realizzazione del progetto MaaP.

Registro delle modifiche

Versione	Data	Persone coinvolte	Descrizione
4.0.0	2014-03-24	Federico Poli (Progettista)	Approvazione.
3.1.0	2014-03-24	Enrico Rotundo (Progettista)	Verifica.
3.0.5	2014-03-22	Gianluca Donato (Progettista)	Miglioramento e correzione diagrammi package.
3.0.4	2014-03-22	Gianluca Donato (Progettista)	Incremento classi Back-end.
3.0.3	2014-03-21	Gianluca Donato (Progettista)	Incremento classi Front-end.
3.0.2	2014-03-20	Luca De Franceschi (Progettista)	Correzione diagrammi.
3.0.1	2014-03-19	Luca De Franceschi (Progettista)	Correzione librerie e framework utilizzati: incrementate descrizioni.
3.0.0	2014-02-25	Federico Poli (Responsabile)	Approvazione.
2.1.0	2014-02-18	Enrico Rotundo (Verificatore)	Verifica.
2.0.2	2014-02-16	Gianluca Donato (Progettista)	Aggiornate descrizioni package e classi.
2.0.1	2014-02-15	Luca De Franceschi (Progettista)	Correzione documento.
2.0.0	2014-02-05	Luca De Franceschi (Responsabile)	Approvazione.
1.2.0	2014-02-03	Enrico Rotundo (Verificatore)	Verifica.
.1.8	2014-02-02	Nicolò Tresoldi (Progettista)	Stesura Stima di fattibilità.
1.1.7	2014-02-01	Nicolò Tresoldi (Progettista)	Stesura Tracciamento.
1.1.6	2014-02-01	Luca De Franceschi (Progettista)	Descrizione Design Pattern utilizzati.
1.1.5	2014-01-31	Federico Poli (Progettista)	Aggiornamento capitolo descrizione architettura.
1.1.4	2014-01-31	Serena Girardi (Progettista)	Aggiunta dei diagrammi di attività.

1.1.3	2014-01-29	Gianluca Donato (Progettista) Federico Poli (Progettista)	Stesura descrizione package e classi Front-end.
1.1.2	2014-01-29	Serena Girardi (Progettista)	Aggiunta degli scenari al Back-end.
1.1.1	2014-01-28	Serena Girardi (Progettista) Federico Poli (Progettista)	Stesura descrizione package e classi Back-end.
1.1.0	2014-01-22	Nicolò Tresoldi (Verificatore)	Verifica.
1.0.6	2014-01-18	Giacomo Fornari (Progettista)	Stesura diagramma classi Front-end.
1.0.5	2014-01-18	Serena Girardi (Progettista)	Stesura appendice Design Pattern.
1.0.4	2014-01-18	Federico Poli (Progettista)	Aggiunto diagramma dei package.
1.0.3	2014-01-16	Federico Poli (Progettista)	Stesura capitolo descrizione architettura.
1.0.2	2014-01-15	Gianluca Donato (Progettista)	Stesura capitolo Tecnologie utilizzate.
1.0.1	2014-01-14	Federico Poli (Progettista)	Stesura capitolo introduzione.
1.0.0	2014-01-14	Gianluca Donato (Progettista)	Creazione scheletro.

Indice

Elenco delle tabelle

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

Questo documento ha come scopo quello di definire la progettazione ad alto livello per il prodotto.

Verranno presentati l'architettura generale secondo la quale saranno organizzate le varie componenti software e i *Design Pattern*$_G$ utilizzati nella creazione del prodotto. Verrà dettagliato il tracciamento tra le componenti software individuate ed i requisiti. Qualora vengano apportate modifiche o aggiunte al presente documento sarà necessario informare tempestivamente ogni membro del gruppo.

1.2 Scopo del prodotto

Lo scopo del progetto è la realizzazione di un *framework*$_G$ per generare interfacce web di amministrazione dei dati di *business*$_G$ basato su *stack*$_G$ *Node.js*$_G$ e *MongoDB*$_G$. L'obiettivo è quello di semplificare il processo di implementazione di tali interfacce che lo sviluppatore, appoggiandosi alla produttività del framework MaaP, potrà generare in maniera semplice e veloce ottenendo quindi un considerevole risparmio di tempo e di sforzo. Il fruitore finale delle pagine generate sarà infine l'*esperto di business*$_G$ che potrà visualizzare, gestire e modificare le varie entità e dati residenti in *MongoDB*$_G$. Il prodotto atteso si chiama *MaaP*$_G$ ossia *MongoDB as an admin Platform*.

1.3 Glossario

Al fine di evitare ogni ambiguità relativa al linguaggio impiegato nei documenti viene fornito il *Glossario v5.0.0*, contenente la definizione dei termini marcati con una G pedice.

1.4 Riferimenti

1.4.1 Normativi

- *Norme di Progetto v5.0.0*;
- Capitolato d'appalto C1: MaaP: MongoDB as an admin Platform: http://www.math.unipd.it/~tullio/IS-1/2013/Progetto/C1.pdf;
- *Analisi dei Requisiti v4.0.0*;
- Verbale del 2013-12-05;
- Verbale del 2013-12-18.

1.4.2 Informativi

- Presentazione capitolato d'appalto: http://www.math.unipd.it/~tullio/IS-1/2013/Progetto/C1p.pdf;
- Ingegneria del software - Ian Sommerville - 8a edizione (2007), Parte terza: *Progettazione*, Capitolo 11: *Progettazione architetturale*, Capitolo 14: *Progettazione orientata agli oggetti*;
- Dall'idea al codice con UML 2 - L. Baresi, L. Lavazza, M. Pianciamore - 1a edizione (2006);
- Design Patterns - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - 1a edizione italiana (2008);
- Learning JavaScript Design Patterns - Addy Osmani - 1a edizione (2012);
- Patterns of Enterprise Application Architecture - Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford - edizione Novembre 2002;
- Node.js - Marc Wandschneider - 1a edizione (2013).

2 Tecnologie utilizzate

L'architettura è stata progettata utilizzando diverse tecnologie, alcune delle quali espressamente richieste nel capitolato d'appalto. Vengono di seguito elencate e descritte le principali tecnologie impiegate e le motivazioni del loro utilizzo.

- **Node.js**: piattaforma per il *back-end*$_G$;
- **Express**: framework per la realizzazione dell'applicazione web in *Node.js*$_G$;
- **MongoDB**: database di tipo *NoSQL*$_G$ per la parte di recupero e salvataggio dei dati;
- **Mongoose**: libreria per interfacciarsi con il driver di **MongoDB**;
- **AngularJS**: framework *JavaScript*$_G$ la realizzazione del *front-end*$_G$.

2.1 Node.js

Node.js è una *piattaforma*$_G$ software sviluppata in C, C++ e JavaScript, costruita sul motore *V8 JavaScript Engine*$_G$ di *Google Chrome*$_G$ che permette di realizzare facilmente applicazioni di rete scalabili e veloci. ***Node.js***$_G$ utilizza ***JavaScript***$_G$ come linguaggio di programmazione, il quale, grazie al suo modello *event-driver*$_G$ con chiamate di *I/O*$_G$ non bloccanti, risulta essere leggero e efficiente.

I principali vantaggi derivanti dall'utilizzo di *Node.js*$_G$ sono:

- **Approccio asincrono**: *Node.js*$_G$ permette di accedere alle risorse del sistema operativo in modalità *event-driven*$_G$ e non sfruttando il classico modello basato su processi o thread concorrenti utilizzato dai classici web server. Ciò garantisce una maggiore efficienza in termini di prestazioni, poiché durante le attese il runtime può gestire qualcos'altro in maniera asincrona;
- **Architettura modulare**: lavorando con *Node.js*$_G$ è molto facile organizzare il lavoro in librerie, importare i *moduli*$_G$ e combinarli fra loro. Questo è reso molto comodo attraverso il node package manager$_G$ (**npm**) attraverso il quale lo sviluppatore può contribuire e accedere ai *package*$_G$ messi a disposizione dalla community.

Le applicazioni Node.js vengono eseguite su un singolo thread, sebbene Node.js utilizzi un modello multi-thread per la gestione degli eventi legati a file e connessioni di rete.

2.2 Express

Express è un *framework*$_G$ minimale, basato sul design pattern architetturale *MVC*$_G$ per creare applicazioni web con *Node.js*$_G$. Express offre funzionalità che semplificano e aumentano le potenzialità di *Node.js*$_G$, fornendo una migliore implementazione del sistema di *routing*$_G$, incrementando le funzioni di richiesta e risposta estendendole per una maggior flessibilità, integrando nuovi *middleware*$_G$, ed agevolando la realizzazione delle *viste*$_G$.

Express non limita l'utente nella scelta del linguaggio di templating, lo aiuta a gestire le route, le request e le view.

2.3 MongoDB

MongoDB$_G$ è un database *NoSQL*$_G$ *open source*$_G$ scalabile e altamente performante di tipo document-oriented, in cui i dati sono archiviati sotto forma di documenti in stile *JSON*$_G$ con schemi dinamici che *MongoDB*$_G$ chiama *BSON*$_G$, secondo una struttura semplice e potente.

I principali vantaggi derivati dal suo utilizzo sono:

- **Alte performance**: non ci sono *join* che possono rallentare le operazioni di lettura o scrittura. L'indicizzazione include gli indici di chiave anche sui documenti innestati e sugli array, permettendo una rapida interrogazione al database;
- **Affidabilità**: alto meccanismo di replicazione su server;
- **Schemaless**: non esiste nessuno *schema*$_G$, è più flessibile e può essere facilmente trasposto in un modello ad oggetti;
- Permette di processare parallelamente i dati (*Map-Reduce*$_G$);
- Tipi di dato più flessibili.

Altre funzionalità comprendono la possibilità di creare delle query ad hoc, l'*Auto-Sharding*, ovvero la capacità di scalare orizzontalmente e di aggiungere nuove macchine al database operativo. Inoltre, MongoDB supporta la definizione di collection con una dimensione fissata. Questo particolare tipo di collection mantiene l'ordine di inserimento e, una volta raggiunta la dimensione massima prefissata, si comporta come una lista circolare.

2.4 Mongoose

Mongoose è una libreria per interfacciarsi a *MongoDB*$_G$ che permette di definire degli schemi per modellare i dati del database, imponendo una certa struttura per la creazione di nuovi *Document*$_G$. Inoltre, fornisce molti strumenti utili per la validazione dei dati, per la definizione di query e per il cast dei tipi predefiniti.

La documentazione di *Mongoose*$_G$ è ben fornita e descrive le API in maniera esaustiva, fornendo degli *snippet*$_G$ del codice sorgente.

Per interfacciare l'application server con *MongoDB*$_G$ sono disponibili diversi progetti *open source*$_G$. Per questo progetto è stato scelto di utilizzare *Mongoose*$_G$, attualmente il più diffuso.

2.5 AngularJS

AngularJS è un *framework*$_G$ architetturale per applicazioni dinamiche,patrocinato da Google. Uno dei vantaggi più grandi che caratterizzano questo *framework*$_G$ è la possibilità di integrare e utilizzare molte funzioni utilizzando quasi esclusivamente l' HTML grazie all'approccio dichiarativo, permettendo di estenderne la sintassi per esprimere le componenti dell'applicazione in maniera chiara e succinta.

Le caratteristiche principali che caratterizzano questo *framework*$_G$ sono:

- **Data Binding** : è un' approccio automatico di aggiornare la vista ogni ogniqualvolta il model cambia e viceversa. Aiuta lo sviluppo eliminando la manipolazione del DOM allo sviluppatore.

- **Dependency injection** : permette di descrivere in maniera dichiarativa quali sono le dipendenze che l'applicazione possiede, isolando i comportamenti e le responsabilità dei componenti garantendo un facile rimpiazzo di quest'ultimi. Questo meccanismo favorisce inoltre la testabilità del codice dell'applicazione.

3 Descrizione architettura

3.1 Metodo e formalismo di specifica

Le scelte progettuali per lo sviluppo di MaaP sono state fortemente influenzate dallo stack tecnologico utilizzato.

In primo luogo il progetto è basato su Node.js ed è scritto quindi in JavaScript: un linguaggio che è (tra le altre caratteristiche) orientato agli oggetti (OOP_G), ma che lascia grande libertà al programmatore nella scelta della tecnica da utilizzare per l'implementazione di pattern come l'*incapsulamento*$_G$ e l'*ereditarietà*$_G$. Al contrario di altri linguaggi (C++, Java e derivati) non c'è un costrutto esplicito con il quale il programmatore può definire classi.

Progettare il sistema con un'architettura ad oggetti classica non permette di rappresentare in modo naturale la gestione dinamica dei tipi e le caratteristiche tipiche degli stili di programmazione funzionali. In certi casi è stato necessario introdurre interfacce e classi "fittizie", che non verranno codificate. Dato che questo introduce numerosi schematismi che appesantiscono i diagrammi e che non sono richiesti dal linguaggio di programmazione, si è cercato di limitarli soltanto ai casi in cui sono particolarmente utili.

Per rappresentare l'utilizzo delle funzioni come parametri tipico della programmazione funzionale è stato necessario valutare se rappresentarlo come classe o se utilizzare una notazione partico: lare, dato che i diagrammi delle classi UML si adattano poco all'utilizzo di codice proveniente dalla programmazione funzionale. La prima opzione avrebbe richiesto di raddoppiare quasi il numero di classi progettate, quindi con l'intenzione di mantenere la specifica tecnica chiara e maneggevole si è scelto di utilizzare una notazione ad hoc. Tale notazione è della forma "`function(<parametri>)`" e rappresenta il tipo di dato di una funzione che richiede i parametri "`<parametri>`".

Il nostro approccio alla progettazione è stato contemporaneamente top-down e bottom-up. Da un lato siamo partiti suddividendo il sistema in front-end e back-end, definendo l'interfaccia di comunicazione, scegliendo di seguire in ciascuno l'organizzazione suggeritaci dai framework (Express e Angular.js). Dall'altro lato siamo partiti dal basso, componendo e cercando di riutilizzare il più possibile le librerie già esistenti. Per far questo abbiamo prima cercato e confrontato con attenzione la struttura e le scelte sia di progetti open source che di progetti proposti come best practice.

L'approccio top-down è stato schematizzato nei diagrammi di deployment e dei package. Per la costruzione dei diagrammi delle classi, invece, questo approccio si è rivelato essere poco produttivo e rigoroso. I diagrammi delle classi proposti sono quindi *uno dei possibili diagrammi* che descrivono l'applicazione, qualsiasi gerarchia o relazione complicata tra le classi verrebbe tradotta pressapoco nello stesso codice.

Per descrivere il sistema si è rivelato molto più comodo utilizzare i diagrammi di sequenza e di attività in un approccio bottom-up, descrivendo l'interazione tra i singoli oggetti senza preoccuparci della loro classificazione. In questo modo siamo anche riusciti a descrivere alcuni dei meccanismi tipici dell'applicazione, in particolar modo l'ordine in cui agiscono i *middleware*$_G$ di Express. Riteniamo che saranno molto utili per la progettazione di dettaglio e per la codifica.

A posteriori, riteniamo che sarebbe stato molto meglio progettare il sistema seguendo uno stile di scomposizione modulare *orientato alle funzioni*[1] o a flusso di dati piuttosto che *a oggetti*. Con questa modifica radicale al metodo di specifica sarebbe stato possibile rappresentare in modo più naturale diverse delle caratteristiche salienti delle tecnologie e librerie utilizzate, che si strutturano meglio come pipeline piuttosto che come classi.

I diagrammi di deployment, dei *package*$_G$, delle classi, di sequenza e di attività presentati di seguito utilizzano la specifica *UML*$_G$ 2.0, incrementata con la convenzione per la rappresentazione delle funzioni. Nei diagrammi dei *package*$_G$ in particolare, i package colorati rappresentano librerie esterne.

3.2 Architettura generale

L'architettura del progetto si suddivide innanzitutto in una componente Client, costituita dal browser degli utenti che interagiranno con il *front-end*$_G$ dell'applicazione, e in una componente WebServer, su cui verrà posto il *back-end*$_G$. Riguardo al server che ospita i database (la cui configurazione non è compito del progetto) non è necessario che risieda sullo stesso nodo su cui è posto il *back-end*$_G$.

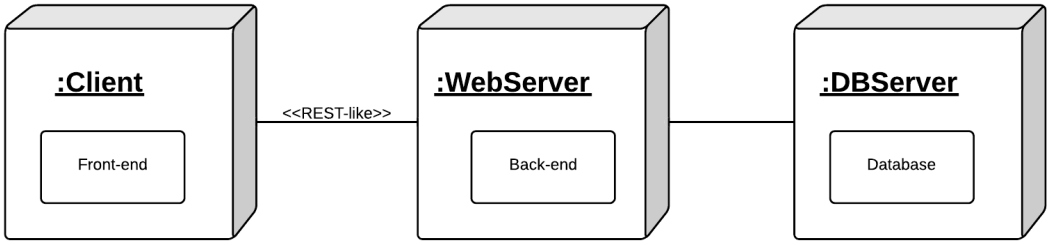

Figura 1: Diagramma di deployment

3.3 Interfaccia REST-like

Per l'interfaccia della componente Back-end di MaaP si è scelto di utilizzare uno stile *REST-like*$_G$, ovvero basato sullo stile *REST*$_G$ ma modificato per permettere l'autenticazione (tramite cookie) e l'attivazione di determinate operazioni. All'interno di un'unica sessione utente, a partire dall'operazione di login fino a quella di logout, l'interfaccia con cui si accede agli elementi delle collection può considerarsi effettivamente *REST*$_G$.

I motivi che hanno spinto alla scelta di *REST*$_G$ sono:

- Semplicità di utilizzo;
- Facile integrazione con i framework esistenti (Angular.js e Express);
- Indipendenza dal linguaggio di programmazione utilizzato;

[1] Descritto alla sezione 11.3.2 del libro "Ingegneria del software - Sommerville - 8a edizione (2007)"

$REST_G$ utilizza il concetto di risorsa, ovvero un aggregato di dati con un nome (URI_G) e una rappresentazione, su cui è possibile invocare le operazioni $CRUD_G$ tramite la seguente corrispondenza:

Risorsa	**URI della collection** es. http://example.com/users	**URI di un elemento** es. http://example.com/users/42
GET	**Fornisce** informazioni sui membri della collection.	**Fornisce** una rappresentazione dell'elemento della collection in: dicato, espresso in un appropriato formato.
PUT	Non usata.	**Sostituisce** l'elemento della col: lection indicato, o se non esiste, lo **crea**.
POST	**Crea** un nuovo elemento nella collection. La URI del nuovo elemento è generata automatica: mente ed è di solito restituita dall'operazione.	Non usato.
DELETE	Non usata.	**Cancella** l'elemento della collection indicato.

Per il formato di rappresentazione dei dati è stato scelto $JSON_G$, in quanto si integra molto facilmente con i framework utilizzati e con il linguaggio $JavaScript_G$, a differenza di XML_G o CSV_G che richiederebbero l'utilizzo di librerie specifiche.

3.3.1 Back-end

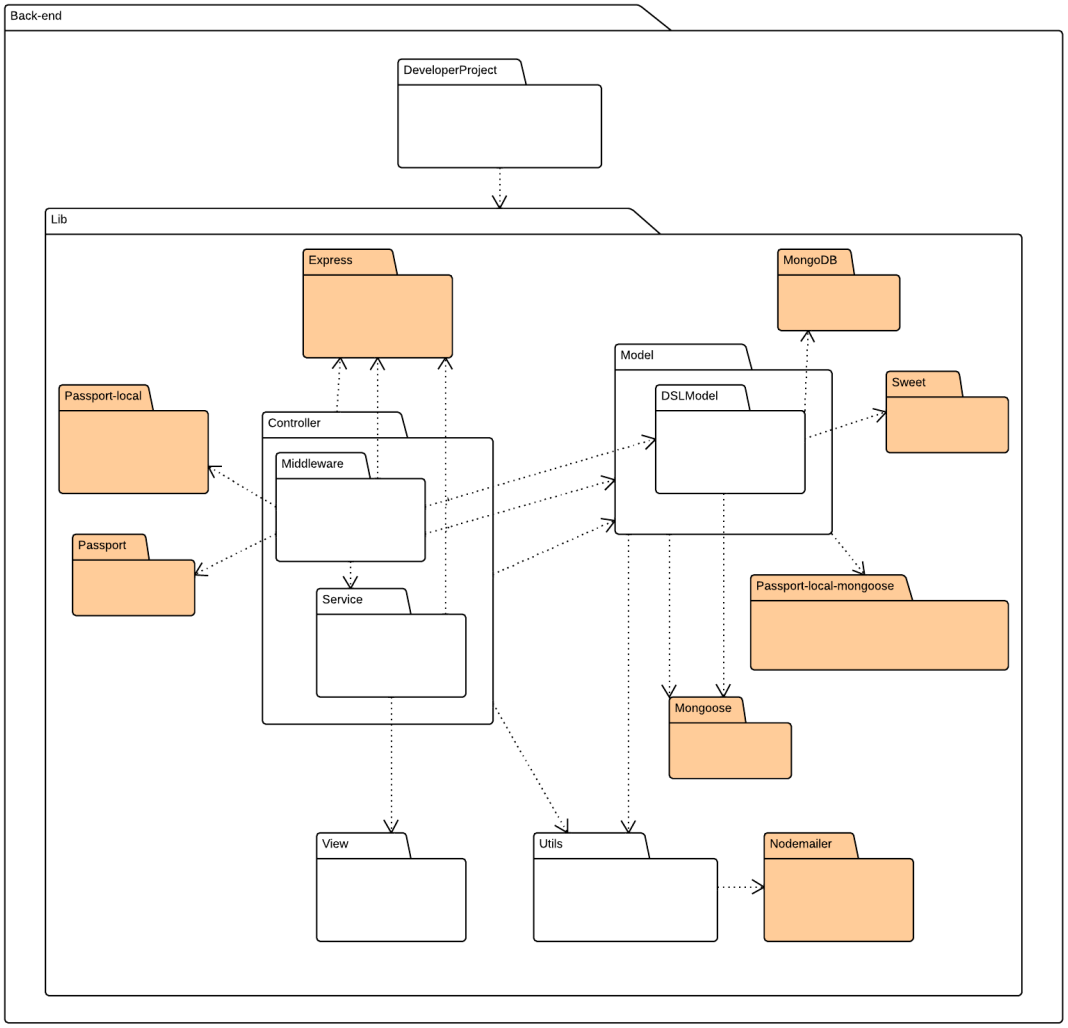

Figura 2: Diagramma dei package del Back-end

L'architettura del *back-end*$_G$ si serve del pattern architetturale *MVC*$_G$ (*Model-View-Controller*), suddividendo i controller tradizionali dai controller *middleware*$_G$, come incoraggiato dal framework Express. Nei diagrammi non è rappresentata la *view* poiché essa consiste nel *front-end*$_G$.

Il *front-end*$_G$, come descritto più avanti, utilizza anch'esso internamente un'architettura della famiglia *MVC*$_G$, ma dal punto di vista del *back-end*$_G$ può considerarsi semplicemente una *view*. La comunicazione tra i due avviene utilizzando il formato *JSON*$_G$ e la conversione dalla rappresentazione interna alla presentazione testuale (*JSON*$_G$) è automatica e diretta. La struttura dati inviata, in particolare, coincide con la componente *model* del front-end.

3.3.2 Front-End

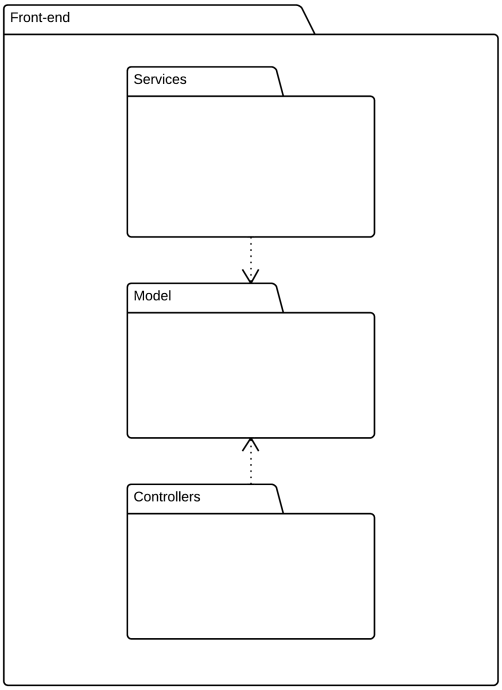

Figura 3: Diagramma dei package del Front-end

Il *front-end*$_G$ è gestito dal *framework*$_G$ *AngularJS*$_G$, la cui architettura è definita *MVW*$_G$ (ossia *Model-View-Whatever*) per la caratteristica di non corrispondere esattamente ad uno dei modelli classici. Nell'architettura si è scelto di descrivere i *package*$_G$ del controller e del model, nonché un package che definisce i servizi con i quali i controller potranno interagire con il *back-end*$_G$ e popolare i modelli di AngularJS. La view non è rappresentata poiché consiste unicamente in dei file statici di template scritti in un linguaggio molto simile all'*HTML*$_G$. Tali file risiedono fisicamente sul web-server, assieme alle librerie *JavaScript*$_G$ e ad altri file statici e vengono caricati dal controller alla richiesta.

4 Back-end

4.1 Interfaccia REST

Ad ogni richiesta il server può rispondere con un messaggio di errore nel formato $JSON_G$ e inviato con un codice $HTTP_G$ della tipologia 4xx o 5xx. Il formato $JSON_G$ del messaggio di errore sarà:

```
{
  "code": [codice numerico dell'errore],
  "message": [descrizione testuale dell'errore],
  "data": [eventuali dati aggiuntivi sull'errore]
}
```

Di seguito sono elencate le risorse REST associate al tipo di metodo che è possibile richiedere su esse e i permessi richiesti per poter effettuare la richiesta.
I tipi di permessi possibili sono:

- **Utente**: questa risorsa può essere richiesta da qualsiasi tipo di utente;
- **Utente Autenticato**: questa risorsa può essere richiesta solo dagli utenti autenticati a $MaaP_G$;
- **Admin**: tale risorsa può essere richiesta solo da utenti con livello Admin.

/profile	**GET**	**Utente Autenticato**
Restituisce i dati relativi all'utente.		
/profile	**POST**	**Utente**
Crea una nuova sessione associata all'utente, corrisponde al login.		
/profile	**PUT**	**Utente Autenticato**
Modifica i dati utente.		
/profile	**DELETE**	**Utente Autenticato**
Elimina la sessione utente, corrisponde al logout.		

/register	**POST**	**Utente**
Crea una richiesta di registrazione.		

/dashboard	**GET**	**Utente**
Restituisce i dati da visualizzare nella dashboard.		

/password/forgot	**POST**	**Utente**

Crea una richiesta di recupero password.

/users	**GET**	**Admin**
Restituisce la lista di tutti gli utenti.		
/users	**POST**	**Admin**
Crea un nuovo utente.		

/users/{user id}	**GET**	**Admin**
Restituisce i dati corrispondenti all'utente con id {user id}.		
/users/{user id}	**PUT**	**Admin**
Modifica i dati dell'utente con id {user id}.		
/users/{user id}	**DELETE**	**Admin**
Elimina l'utente con id {user id}.		

/collection	**GET**	**Utente Autenticato**
Restituisce la lista delle collection.		

/collection/{collection name}	**GET**	**Utente Autenticato**
Restituisce la lista di document della collection {collection name}.		

/collection/{collection name}/{document id}	**GET**	**Utente Autenticato**
Restituisce la lista di attributi del Document {document id} appartenente alla collection {collection name}		
/collection/{collection name}/{document id}	**PUT**	**Admin**
Modifica il document {document id}.		
/collection/{collection name}/{document id}	**DELETE**	**Admin**
Elimina il document con id {document id}.		

/action/{action name}/{collection name}	**PUT**	**Utente Autenticato**
Esegue l'azione {action name} sulla Collection {collection name}.		

/action/{action name}/{collection name}/{document id}	**PUT**	**Utente Autenticato**
Esegue l'azione {action name} sul Document {document name} della Collection {collection name}.		

4.2 Descrizione packages e classi

4.2.1 Back-end

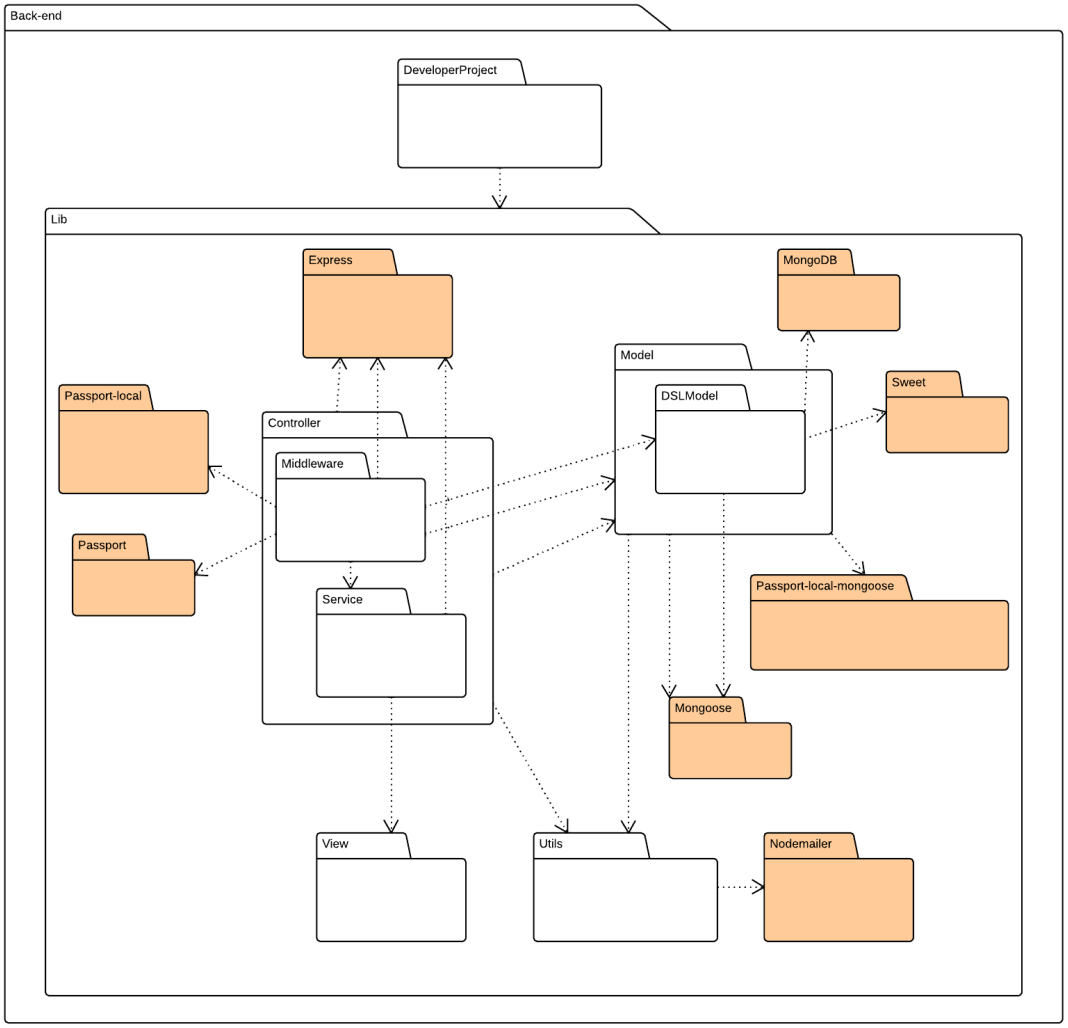

Figura 4: Diagramma dei packages Back-end

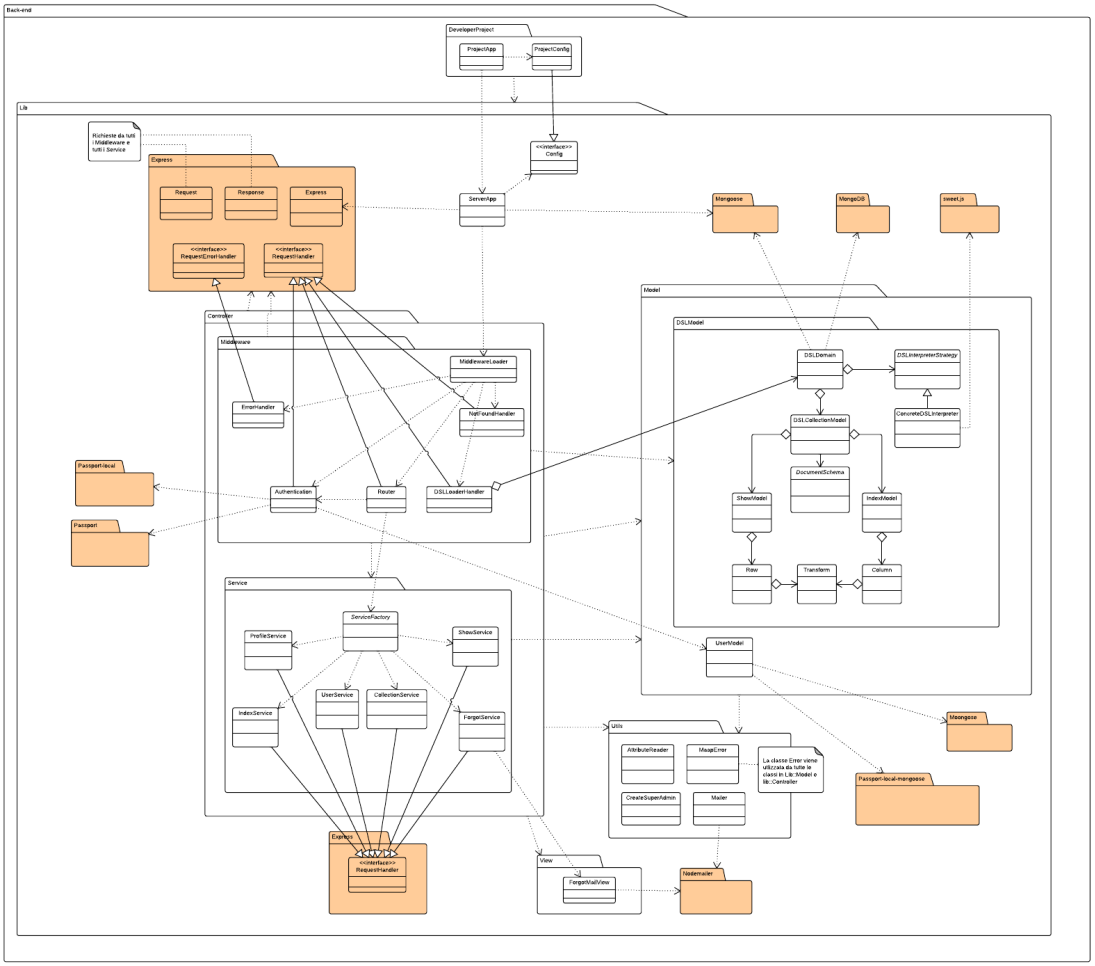

Figura 5: Diagramma delle classi Back-end

4.2.1.1 Informazioni sul package

4.2.1.1.1 Descrizione

Package$_G$ che racchiude tutta la componente di *Back-end*$_G$. Comprende la libreria dell'applicazione *MaaP* e il package del progetto sviluppato dal developer che andrà ad utilizzare tale libreria.

4.2.1.1.2 Package contenuti

- DeveloperProject
- Lib

4.2.2 Back-end::DeveloperProject

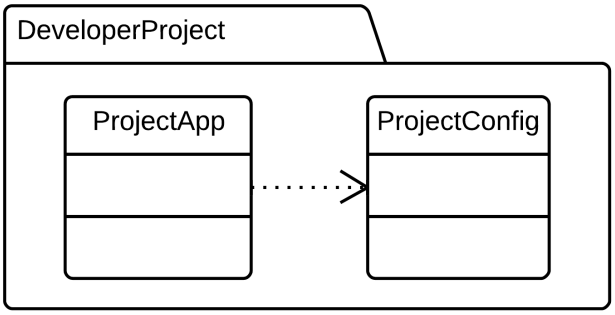

Figura 6: Componente DeveloperProject

4.2.2.1 Informazioni sul package

4.2.2.1.1 Descrizione

Questo $Package_G$ ha il compito di fornire la configurazione e avviare il web server di $MaaP_G$. Consiste negli oggetti che dovranno essere predisposti dal developer che vorrà installare il framework $MaaP_G$. L'installazione dell framework $MaaP_G$ fornisce uno $scaffold_G$ dei file e delle classi necessarie per il funzionamento dell'applicazione. Sarà compito del developer modificare tali file inserendo i dati corretti.

4.2.2.1.2 Interazioni con altri componenti

- Back-end::Lib

4.2.2.2 Classi

4.2.2.2.1 ProjectConfig

Descrizione

Questa classe rappresenta la configurazione di un'applicazione.

Utilizzo

Viene passato come parametro al costruttore della classe ServerApp per configurare l'applicazione.

Classi Ereditate

- Back-end::Lib::Config

4.2.2.2.2 ProjectApp

Descrizione

Classe modificabile dall'utente-developer che si occupa di configurare e avviare il server dell'applicazione.

Utilizzo

Internamente avvia il server utilizzando la classe ServerApp, a cui passa i parametri di configurazione del progetto definiti con un oggetto della classe ProjectConfig.

Relazioni con altre classi

- Back-end::DeveloperProject::Config::ProjectConfig
- Back-end::Lib::ServerApp

4.2.3 Back-end::Lib

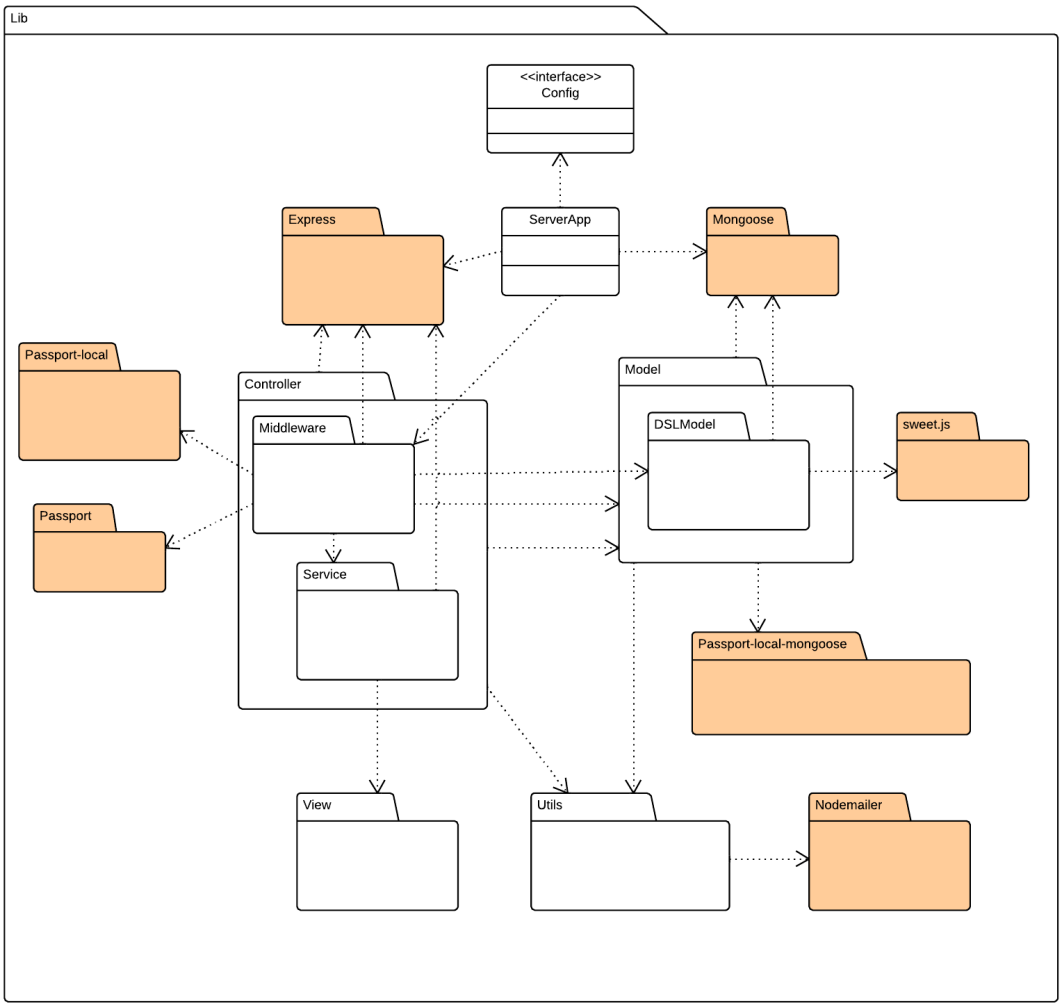

Figura 7: Componente Lib

4.2.3.1 Informazioni sul package

4.2.3.1.1 Descrizione

$Package_G$ che costituisce la libreria principale dell'applicazione $MaaP_G$, che verrà fornita ai developer per installare e utilizzare l'applicazione. Comprende gli script per l'installazione, non rappresentati nei diagrammi in quanto non sono modellati come oggetti.

4.2.3.1.2 Package contenuti

- Controller
- Model
- View
- Utils

4.2.3.2 Classi

4.2.3.2.1 Config

Descrizione

Classe che rappresenta l'interfaccia della classe di configurazione dell'applicazione.

Utilizzo

Viene utilizzata per descrivere tutti i parametri dell'applicazione. Quando viene creata una ServerApp le viene passato un oggetto di questo tipo ed essa avvierà l'applicazione a partire da questa configurazione.

Estensioni

- Back-end::DeveloperProject::Config::ProjectConfig

4.2.3.2.2 ServerApp

Descrizione

Classe che si occupa di avviare il server e di invocare il *middleware*$_G$. È il componente client del *Design Pattern*$_G$ *Chain of responsibility*$_G$. Utilizza i pacchetti Mongoose ed Express.

Utilizzo

Viene utilizzato per avviare l'applicazione. Internamente inizializza la catena gestione delle chiamate utilizzando la classe `Back-end::Lib::Middleware::MiddlewareLoader`.

Relazioni con altre classi

- Back-end::Lib::Controller::Middleware::MiddlewareLoader
- Back-end::Lib::Config

4.2.4 Back-end::Lib::View

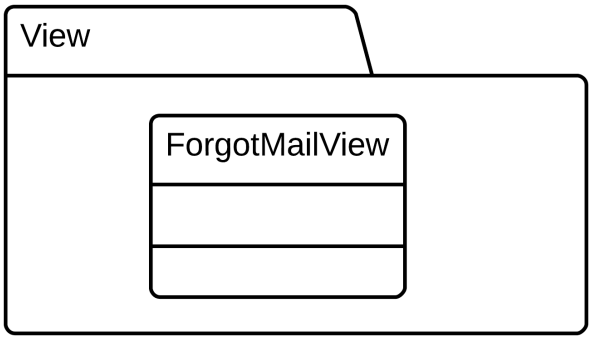

Figura 8: Componente View

4.2.4.1 Informazioni sul package

4.2.4.1.1 Descrizione

$Package_G$ contenente le classi che costituiscono i template, utilizzati ad esempio per le email di recupero- password.

4.2.4.2 Classi

4.2.4.2.1 ForgotMailView

Descrizione

Classe che fornisce una rappresentazione della mail.

Utilizzo

Viene utilizzata come template di email da inviare nel caso in cui l'utente richieda il recupero password.

4.2.5 Back-end::Lib::Controller

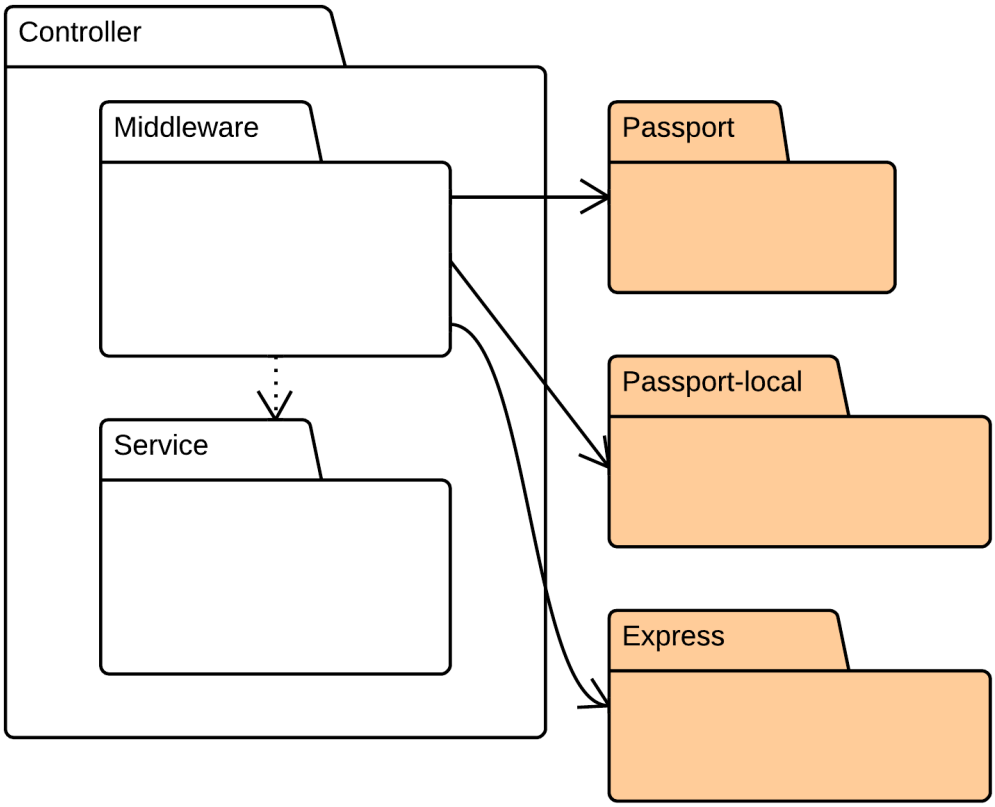

Figura 9: Componente Controller

4.2.5.1 Informazioni sul package

4.2.5.1.1 Descrizione

$Package_G$ contenente le componenti che gestiscono la logica con cui vengono elaborate le richieste inviate all'applicazione.

4.2.5.1.2 Package contenuti

- Service
- Middleware

4.2.6 Back-end::Lib::Controller::Middleware

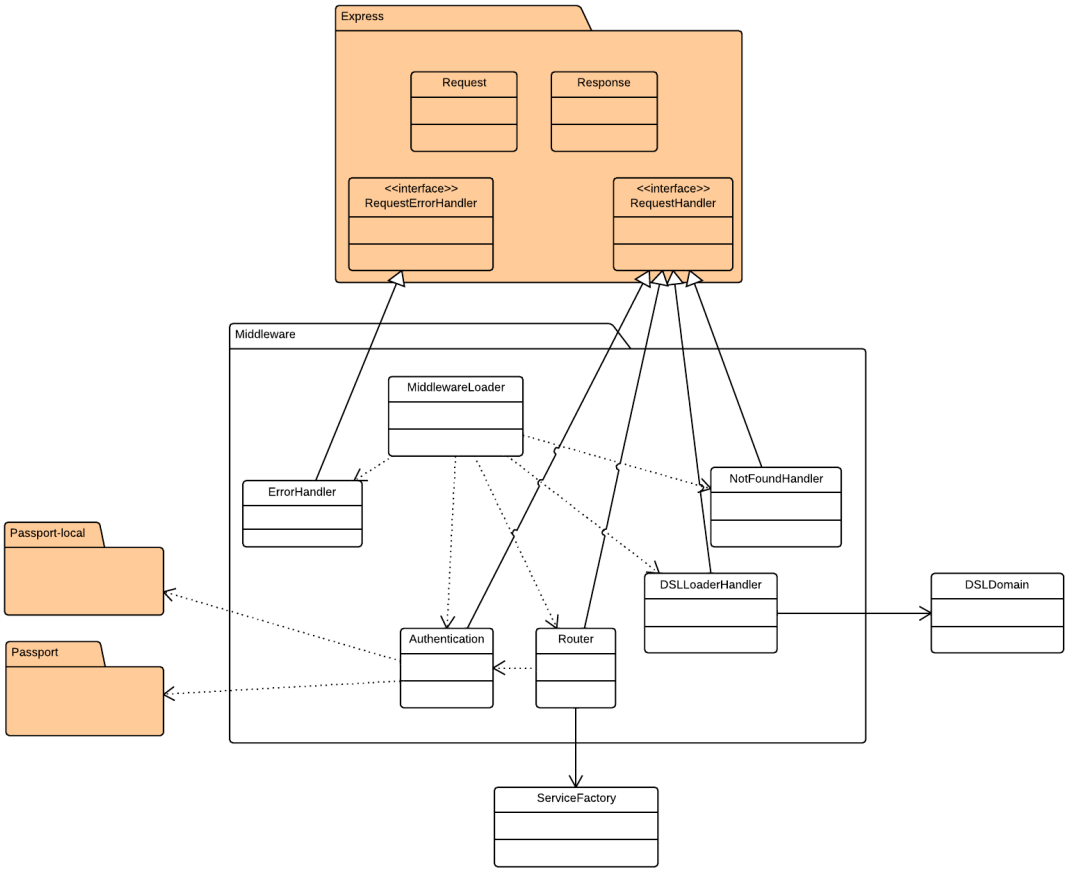

Figura 10: Componente Middleware

4.2.6.1 Informazioni sul package

4.2.6.1.1 Descrizione

Package$_G$ contenente classi che costituiscono gli handler della catena di chiamate a cui viene passata la responsabilità di gestire una richiesta, decorando quest'ultima con parametri e metodi utilizzabili dai controller. Costituisce una parte controller nell'architettura *MVC*$_G$ del *Back-end*$_G$.

4.2.6.1.2 Interazioni con altri componenti

- Back-end::Lib::Controller::Service
- Back-end::Lib::Model

- Back-end::Lib::Model::DSLModel

4.2.6.2 Classi

4.2.6.2.1 MiddlewareLoader

Descrizione

Classe che definisce un'interfaccia comune per tutte le richieste dell'applicazione. È uno dei componenti ConcreteHandler del *Design Pattern*$_G$ *Chain of responsibility*$_G$.

Utilizzo

Viene utilizzato per istanziare in modo nascosto all'applicazione tutti i *middleware*$_G$ presenti nel componente `Back-end::Lib::Middleware`.

Relazioni con altre classi

- Back-end::Lib::Controller::Middleware::DSLLoaderHandler
- Back-end::Lib::Controller::Middleware::NotFoundHandler
- Back-end::Lib::Controller::Middleware::Router
- Back-end::Lib::Controller::Middleware::ErrorHandler

4.2.6.2.2 DSLLoaderHandler

Descrizione

Classe che si occupa di caricare i *DSL*$_G$ presenti nel sistema. È uno dei componenti ConcreteHandler del *Design Pattern*$_G$ *Chain of responsibility*$_G$.

Utilizzo

Viene utilizzata per caricare i *DSL*$_G$ delle *Collection*$_G$ all'interno del *database*$_G$.

Relazioni con altre classi

- Back-end::Lib::Model::DSLModel::DSLDomain

4.2.6.2.3 Authentication

Descrizione

Classe che si occupa dell'autenticazione di un'utente. È uno dei componenti ConcreteHandler del *Design Pattern*$_G$ *Chain of responsibility*$_G$.

Utilizzo

Viene utilizzata per verificare i dati inseriti dall'utente nella pagina di login e controllare l'effettiva corrispondenza delle credenziali nel *database*$_G$.

Relazioni con altre classi

- Back-end::Lib::Model::UserModel

4.2.6.2.4 NotFoundHandler

Descrizione

Classe che si occupa la gestione dell'errore di pagina non trovata. È uno dei componenti ConcreteHandler del *Design Pattern*$_G$ *Chain of responsibility*$_G$.

Utilizzo

Viene utilizzata per generare una pagina 404 di errore nel caso in cui l'*URI*$_G$ passato non corrisponda ad una risorsa presente nell'applicazione.

4.2.6.2.5 Router

Descrizione

Classe che si occupa della richiesta di risorse. È uno dei componenti Handler del *Design Pattern*$_G$ *Chain of responsibility*$_G$. Ha una relazione con la classe Authentication, poiché fa uso di alcuni metodi per controllare l'autenticazione.

Utilizzo

Si occupa di smistare la richiesta in base all'*URI*$_G$ ricevuto e ad invocare l'opportuno metodo di creazione sulla classe `Back-end::Lib::Controller::ControllerFactory`.

Relazioni con altre classi

- Back-end::Lib::Controller::Service::ServiceFactory
- Back-end::Lib::Controller::Middleware::Authentication

4.2.6.2.6 ErrorHandler

Descrizione

Questa classe gestisce gli errori generati nei precedenti middleware o controller. Invia al client una risposta con stato HTTP 500 (server error) con una descrizione dell'errore nel formato JSON. È uno dei componenti ConcreteHandler del *Design Pattern*$_G$ *Chain of responsibility*$_G$.

Utilizzo

Questo middleware viene utilizzato per ultimo nella catena di gestione delle richieste di Express, in modo da gestire tutti gli errori generati precedentemente.

4.2.7 Back-end::Lib::Controller::Service

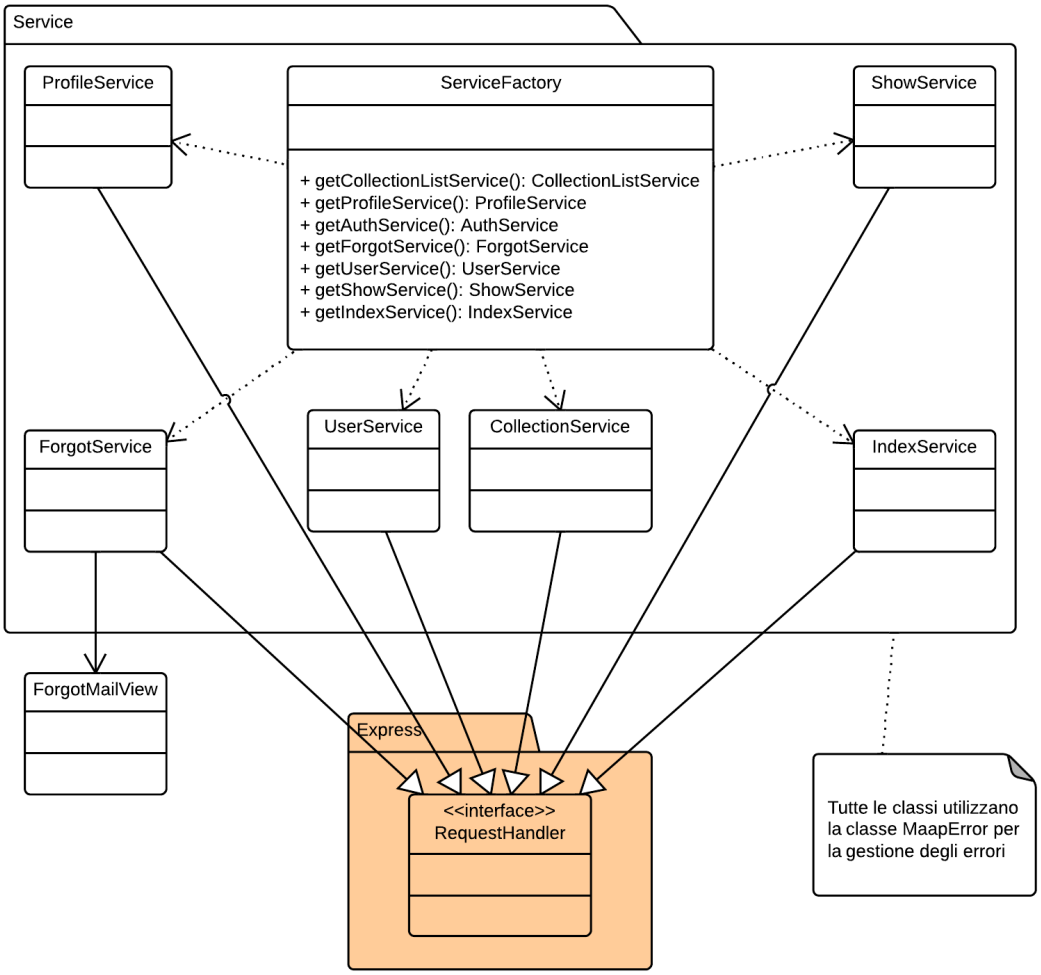

Figura 11: Componente Service

4.2.7.1 Informazioni sul package

4.2.7.1.1 Descrizione

$Package_G$ per il componente che realizza parte controller nell'architettura mvc nel back-end. Contiene classi per le funzionalità di controllo e visualizzazione delle risorse, dove ogni classe gestisce in modo esclusivo una sola di queste, in base all' URI_G .

4.2.7.1.2 Interazioni con altri componenti

- Back-end::Lib::View
- Back-end::Lib::Utils

4.2.7.2 Classi

4.2.7.2.1 UserService

Descrizione

Classe che si occupa della varie operazioni che l'admin può compiere sugli utenti dell'applicazione. È uno dei componenti product del *Design Pattern*$_G$ *Factory method*$_G$.

Utilizzo

Viene utilizzata per visualizzare la *index-page*$_G$ degli utenti, visualizzare le relative *show-page*$_G$, eliminare un utente e modificare il profilo. Mette a disposizione dei metodi per effettuare tutte queste operazioni.

4.2.7.2.2 IndexService

Descrizione

Classe di gestione per la risorsa index È uno dei componenti product del *Design Pattern*$_G$ *Factory method*$_G$.

Utilizzo

Viene utilizzata per gestire la risorsa corrispondente all'index-page di un *Document*$_G$, of: frendo metodi per restituirne gli attributi, effettuarne la modifica o la cancellazione e dele: ga la visualizzazione dell'index-page alla classe `Back-end::Lib::DSLModel::DSLDomain`.

4.2.7.2.3 ServiceFactory

Descrizione

Classe che si occupa di istanziare e restituire una classe *Service*. Rappresenta il componente creator del *Design Pattern*$_G$ *Factory method*$_G$ ed è un *Design Pattern*$_G$ *Singleton*$_G$.

Utilizzo

Viene costruita una sola volta dalla classe *Back-end::Lib::Middleware::Router* e si occupa di creare e restituire l'oggetto *Service* richiesto.

Relazioni con altre classi

- Back-end::Lib::Controller::Service::UserService
- Back-end::Lib::Controller::Service::IndexService
- Back-end::Lib::Controller::Service::ProfileService
- Back-end::Lib::Controller::Service::ShowService

- Back-end::Lib::Controller::Service::ForgotService
- Back-end::Lib::Controller::Service::CollectionService

4.2.7.2.4 ProfileService

Descrizione

Classe che rappresenta la gestione di un profilo utente, il login e il logout. È uno dei componenti product del *Design Pattern*$_G$ *Factory method*$_G$.

Utilizzo

Viene utilizzata per visualizzare il profilo dell'utente, tramite GET, e per editarlo tramite PUT. Viene anche utilizzata per gestire i dati di e le operazioni relativi all'autenticazione utente e al suo logout dall'applicazione, occupandosi della creazione della sessione utente e della sua distruzione tramite *cookies*$_G$.

4.2.7.2.5 ShowService

Descrizione

Classe che si occupa della gestione della risorsa show-page. È uno dei componenti *product* del *Design Pattern*$_G$ *Factory method*$_G$.

Utilizzo

Viene utilizzata per gestire una richiesta della risorsa show-page, delegando alla classe *Back-end::Lib::DSLModel::DSLDomain* il compito di eseguire la query e restituire i dati in formato JSON.

4.2.7.2.6 ForgotService

Descrizione

Classe che rappresenta il sistema di recupero e ripristino password. È uno dei componenti product del *Design Pattern*$_G$ *Factory method*$_G$.

Utilizzo

La classe fornisce dei metodi per effettuare una richiesta di reset password e, in un secondo momento, procedere al suo ripristino. La richiesta di reset avviene mandando un'email all'indirizzo dell'utente tramite la classe `Back-end::Lib::Middleware::Mailer`. All'interno di questo messaggio sarà presente un link che procederà ad effettuare il login dell'utente e a reindirizzarlo nella pagina di modifica profilo, dalla quale potrà modificare la password.

Relazioni con altre classi

- Back-end::Lib::View::ForgotMailView

4.2.7.2.7 CollectionService

Descrizione

Classe di gestione per la risorsa Collection. È uno dei componenti product del *Design Pattern*$_G$ *Factory method*$_G$.

Utilizzo

Viene utilizzata per gestire la risorsa corrispondente alle Collection, offrendo metodi per restituire tutte le collection presenti nell'applicazione.

4.2.8 Back-end::Lib::Model

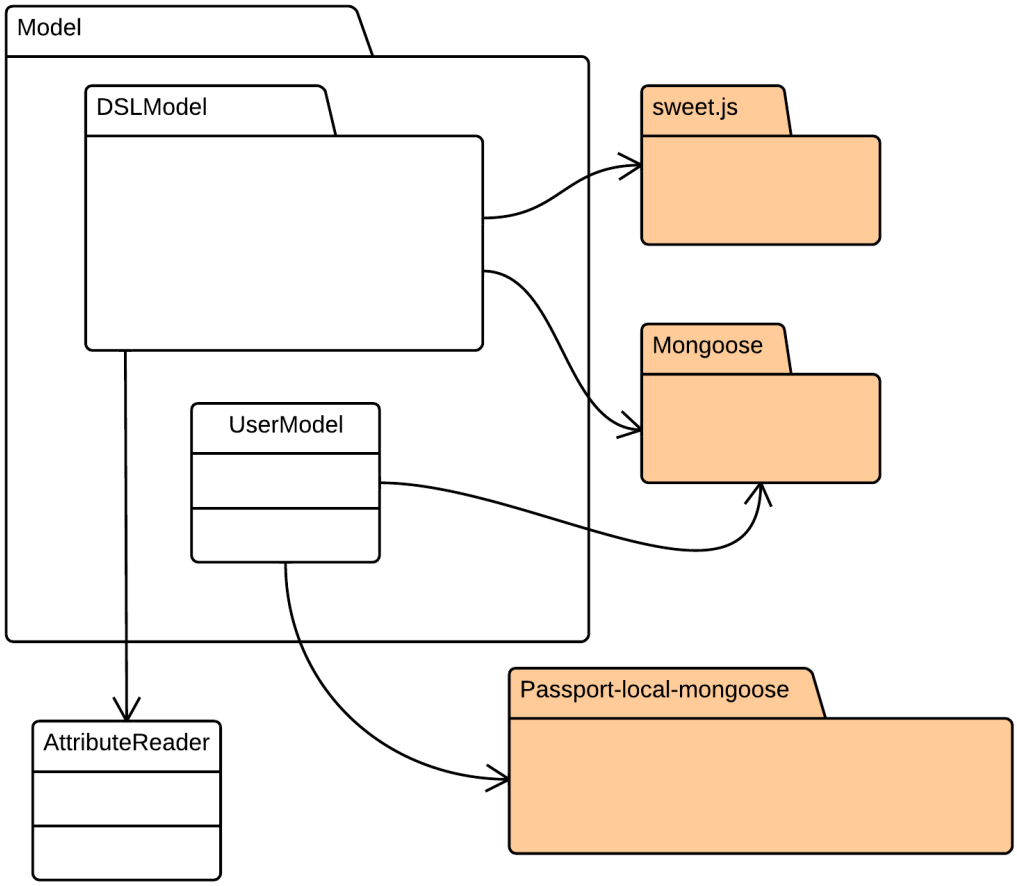

Figura 12: Componente Model

4.2.8.1 Informazioni sul package

4.2.8.1.1 Descrizione

Package$_G$ contenente le componenti che gestiscono i dati utilizzati dall'applicazione e l'interfacciamento con il database

4.2.8.1.2 Package contenuti

- DSLModel

4.2.8.2 Classi

4.2.8.2.1 UserModel

Descrizione

Classe che si occupa dei metodi per la gestione dei dati utente.

Utilizzo

Viene utilizzata per l'interfacciamento con la libreria *Mongoose*$_G$ per la registrazione dello schema dei dati, e con la libreria passport-local-mongoose per il popolamento automatico dello schema con campi dati e metodi predefiniti. Il costruttore del modello dello schema dei dati viene registrato nella *Factory*$_G$ di *Mongoose*$_G$ ed ogni istanza condividerà la stessa connessione al server.
L' istanza della classe UserModel descrive un documento della collection, i metodi di istanza definiscono le operazioni sul documento mentre i metodi statici della classe descrivono le operazioni e le query che è possibile fare sulla Collection in accordo con i metodi di Mongoose, che utilizzano allo stesso modo la distinzione tra metodi statici e di istanza per operazioni sulla Collection e sul Document (rispettivamente).

4.2.9 Back-end::Lib::Model::DSLModel

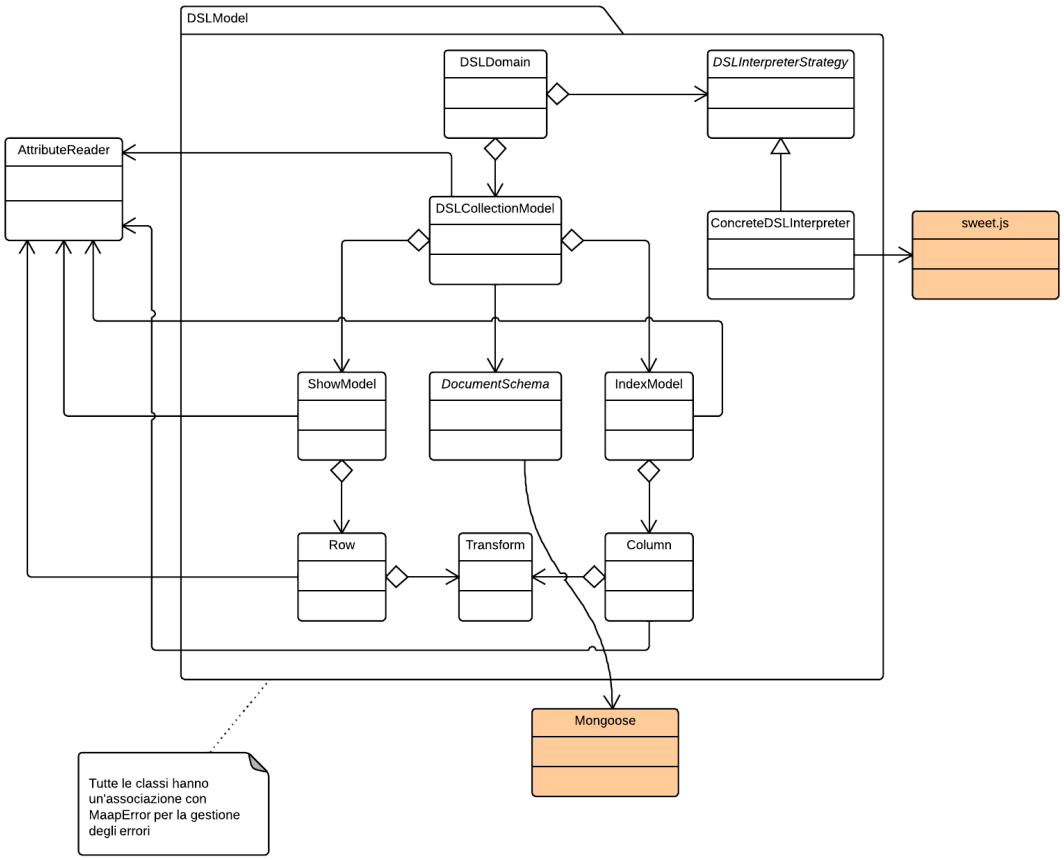

Figura 13: Componente DSLModel

4.2.9.1 Informazioni sul package

4.2.9.1.1 Descrizione

$Package_G$ costituito da classi per la definizione delle regole di business sui dati definite tramite il DSL_G . Il $package_G$ contiene principalmente classi che si occupano del caricamento del DSL_G e della sua rappresentazione in un modello ad oggetti. Costituisce la componente model dell'architettura MVC del back-end.

4.2.9.2 Classi

4.2.9.2.1 DSLDomain

Descrizione

Classe che si occupa di caricare i file DSL_G. Implementa il *Design Pattern*$_G$ *registry*$_G$.

Utilizzo

Viene utilizzata per caricare dinamicamente tutti i DSL_G a partire dal *database*$_G$ che le viene passato.

Relazioni con altre classi

- Back-end::Lib::Model::DSLModel::DSLCollectionModel
- Back-end::Lib::Model::DSLModel::DSLInterpreterStrategy

4.2.9.2.2 DSLCollectionModel

Descrizione

Classe che si occupa di definire il model della *Collection*$_G$ a partire dal DSL_G. Si ispira all'*Abstract Syntax Tree*$_G$.

Utilizzo

È l'oggetto risultante dell'interpretazione del DSL_G. Definisce una rappresentazione interna di una *Collection*$_G$.

Relazioni con altre classi

- Back-end::Lib::Model::DSLModel::IndexModel
- Back-end::Lib::Model::DSLModel::ShowModel

4.2.9.2.3 DSLInterpreterStrategy

Descrizione

Classe astratta che definisce l'interfaccia dell'algoritmo di interpretazione del linguaggio DSL_G utilizzato. È il componente strategy del *Design Pattern*$_G$ *strategy*$_G$.

Utilizzo

Viene utilizzata per incapsulare e rendere intercambiabile l'algoritmo di interpretazione del linguaggio DSL_G. In questo modo, se in futuro vi fosse necessità di cambiare l'algoritmo di interpretazione l'algoritmo può variare indipendentemente dal client che ne farà uso.

Estensioni

- Back-end::Lib::Model::DSLModel::DSLInterpreterStrategy::DSLConcreteStrategy

4.2.9.2.4 DSLConcreteStrategy

Descrizione

Classe che concretizza l'interprete del DSL_G. È uno dei componenti ConcreteStrategy del $Design\ Pattern_G$ $Strategy_G$.

Utilizzo

Viene utilizzata per implementare l'algoritmo utilizzato nell'interfaccia `Back-end::Lib::DSLModel::DSLInter` per l'interpretazione del linguaggio DSL_G. Conterrà al suo interno un metodo che genererà il $parser_G$ a partire da una grammatica regolare.

Classi Ereditate

- Back-end::Lib::Model::DSLModel::DSLInterpreterStrategy

4.2.9.2.5 IndexModel

Descrizione

Classe che racchiude tutte le informazioni relative ad una index-page. Tali informazioni vengono dichiarate dal developer nel DSL. È composta da un numero variabile di colonne, definite dalla classe `Back-end::Lib::DSLModel::Column`.

Utilizzo

Questa classe viene creata dalla componente che si occupa di caricare il DSL (interpretandolo o facendone il parsing) e utilizzata dalla classe `DSLCollectionModel`.

Relazioni con altre classi

- Back-end::Lib::Model::DSLModel::Column

4.2.9.2.6 ShowModel

Descrizione

Classe che racchiude tutte le informazioni relative ad una show-page. Tali informazioni vengono dichiarate dal developer nel DSL. È composta da un numero variabile di attributi, definiti dalla classe `Back-end::Lib::DSLModel::Row`.

Utilizzo

Questa classe viene creata dalla componente che si occupa di caricare il DSL (interpretandolo o facendone il parsing).

Relazioni con altre classi

- Back-end::Lib::Model::DSLModel::Row

4.2.9.2.7 Row

Descrizione

Classe che racchiude tutte le informazioni relative ad una riga di una show-page. Tali informazioni vengono dichiarate dal developer nel DSL.

Utilizzo

Questa classe viene creata dalla componente che si occupa di caricare il DSL (interpretandolo o facendone il parsing).

4.2.9.2.8 Transformation

Descrizione

Classe che racchiude tutte le informazioni relative alla modalità con cui i dati prelevati dal database verranno modificati prima di essere inviati al front-end. Tali trasformazioni vengono dichiarate dal developer nel DSL. Questa classe rappresenta una funzione da chiamare sul valore degli attributi

Utilizzo

Questa classe viene creata dalla componente che si occupa di caricare il DSL (interpretandolo o facendone il parsing).

4.2.9.2.9 DocumentSchema

Descrizione

Questa classe astratta si occupa di definire uno schema mongoose per le Collection e fornisce alcuni metodi statici per effettuare operazioni sulla base di dati, in particolare l'estrazione, rimozione e aggiornamento di un Document.

Utilizzo

Viene utilizzata dalle classi `IndexModel` e `ShowModel` per effettuare operazioni sulla base di dati.

4.2.9.2.10 Column

Descrizione

Classe che racchiude tutte le informazioni relative ad una colonna di una index-page. Tali informazioni vengono dichiarate dal developer nel DSL.

Utilizzo

Questa classe viene creata dalla componente che si occupa di caricare il DSL (interpretandolo o facendone il parsing).

Relazioni con altre classi

- Back-end::Lib::Model::DSLModel::Transformation

4.2.10 Back-end::Lib::Utils

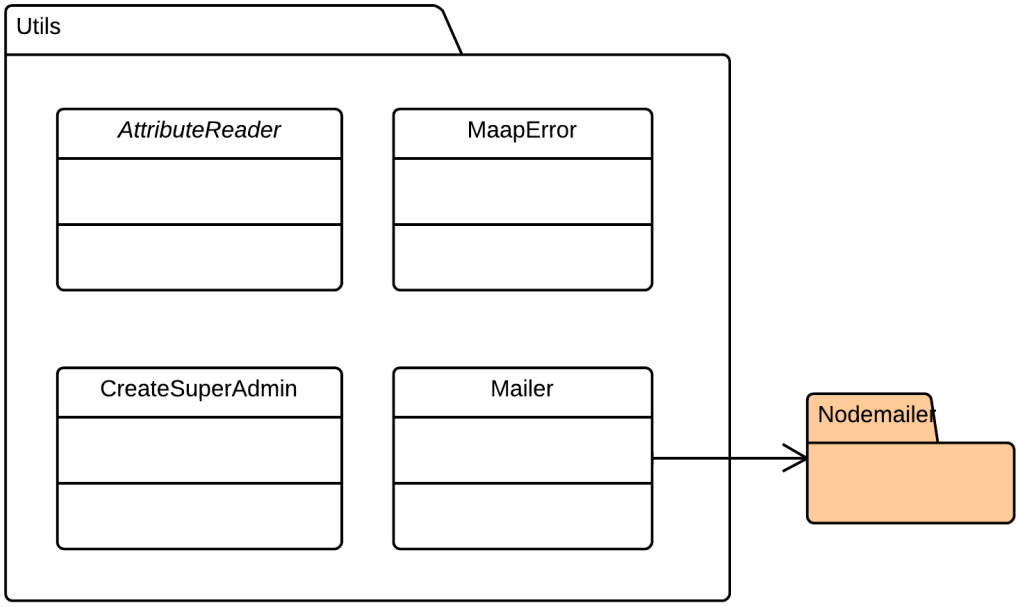

Figura 14: Componente Utils

4.2.10.1 Informazioni sul package

4.2.10.1.1 Descrizione

Package$_G$ comprendente le classi di generica utilità, che non rientrano nella classificazione tra model, view e controller.

4.2.10.2 Classi

4.2.10.2.1 Mailer

Descrizione

Classe che si occupa dell'invio di email. È uno dei componenti subsystem class del *Design Pattern*$_G$ *Facade*$_G$ e handler del *Design Pattern*$_G$ *Chain of responsibility*$_G$.

Utilizzo

Viene utilizzata per inviare un'email ad un utente che ha effettuato la richiesta di recupero password.

4.2.10.2.2 MaapError

Descrizione

Classe che rappresenta un errore all'interno del package `Back-end::Lib`.

Utilizzo

Viene utilizzata da tutte le classi presente all'interno del package `Back-end::Lib` per rappresentare un errore generato, identificandolo tramite nome, descrizione e codice.

4.2.10.2.3 AttributeReader

Descrizione

Questa classe astratta si occupa di effettuare delle verifiche sugli attributi che vengono passati dalla lettura del file DSL.

Utilizzo

Viene utilizzata dalle classi `DslCollectionModel`, `IndexModel`, `ShowModel`, `Row`, `Column` per effettuare controlli sui parametri.

4.3 Scenari

In tutti i diagrammi di sequenza di questa sezione i messaggi inviati tra gli oggetti che compongono il Backend sono asincroni. In questo modo la gestione di qualsiasi richiesta non blocca il processo del server durante il recupero di risorse dal database o da disco. Questa è una delle scelte e caratteristiche fondamentali dell'ambiente Node.js e pertanto se ne è tenuto sin dall'inizio della progettazione.

4.3.1 Gestione generale delle richieste

Nel diagramma di sequenza che rappresenta lo scenario della gestione richieste viene mostrata l'iterazione tra server e middleware, alcuni dei quali sono offerti da Express, altri sono definiti dall'utente o da altre librerie. I *Middleware*$_G$ si distinguono in Middleware di gestione delle richieste e Middleware di gestione degli errori, a seconda del numero di parametri con cui vengono invocati.

Segue un elenco ordinato dei middleware utilizzati. L'ordine in cui elaborano la richiesta è determinante, poiché ciascuno costituisce un handler del pattern Chain of Responsibility e ha la facoltà di interrompere la catena di chiamate.

- express.compress(): *Middleware*$_G$ per comprimere con il formato *gzip*$_G$ le comunicazioni.
- express.logger(): *Middleware*$_G$ utilizzato per registrare un log delle richieste, utile per fare il *debugging*$_G$ dell'applicazione.
- express.json(): *Middleware*$_G$ che estrae dalle richiesta i parametri che sono nel formato JSON.

- express.urlencoded(): *Middleware*$_G$ che estrae dalle richiesta i parametri di tipo www-form-encoded, arrivati ad esempio con una richiesta POST.
- express.methodOverride(): *Middleware*$_G$ utilizzato per permettere anche ai vecchi browser di avere un modo per fare richieste PUT e DELETE.
- express.cookieParser(): *Middleware*$_G$ che analizza i *cookie*$_G$.
- express.cookieSession(): *Middleware*$_G$ per la gestione di sessioni utente basate su cookies.
- Authentication: *Middleware*$_G$ da noi scritto per gestire l'autenticazione. Utilizza nello specifico:
  - passport.initialize(): *Middleware*$_G$ utilizzato per l'inizializzazione di Passport.
  - passport.session(): *Middleware*$_G$ che permette di memorizzare i record della sessione utente per mantenerne lo stato di login.
- express.router(): *Middleware*$_G$ con cui Express gestisce le richieste, smistandole a diversi controller in base alla URI e al metodo HTTP con cui sono state richieste (GET, PUT, POST, DELETE).
- express.static(): *Middleware*$_G$ per servire contenuti statici.
- NotFoundHandler: un *Middleware*$_G$ da noi scritto per gestire le richieste che non vengono gestite da nessun controller (errore client 404).
- ErrorHandler: *Middleware*$_G$ da noi scritto per gestire gli errori sollevati da uno dei precedenti middleware (errore server 500).

Nel seguente diagramma di sequenza viene rappresentata una generica richiesta al server. Le classi e il comportamento sono state progettate in funzione del design pattern della Chain of Responsibility che il framework Express utilizza internamente (si veda la sezione 8.4.1). Specificando meglio i possibili comportamenti del middleware, un middleware può terminare la propria esecuzione in tre diversi modi:

1. Un middleware può terminare la propria esecuzione eseguendo la callback passatagli come parametro. Verranno quindi eseguiti i successivi middleware.
2. In caso di errore, un middleware può eseguire la callback passandogli l'errore come parametro: `next(error)`. In questo modo la callback passerà il controllo al prossimo middleware della catena che è in grado di gestire un errore.
3. Come terza alternativa, un middleware può terminare la propria esecuzione senza eseguire la callback. In questo caso la richiesta non verrà più gestita da nessun altro middleware.

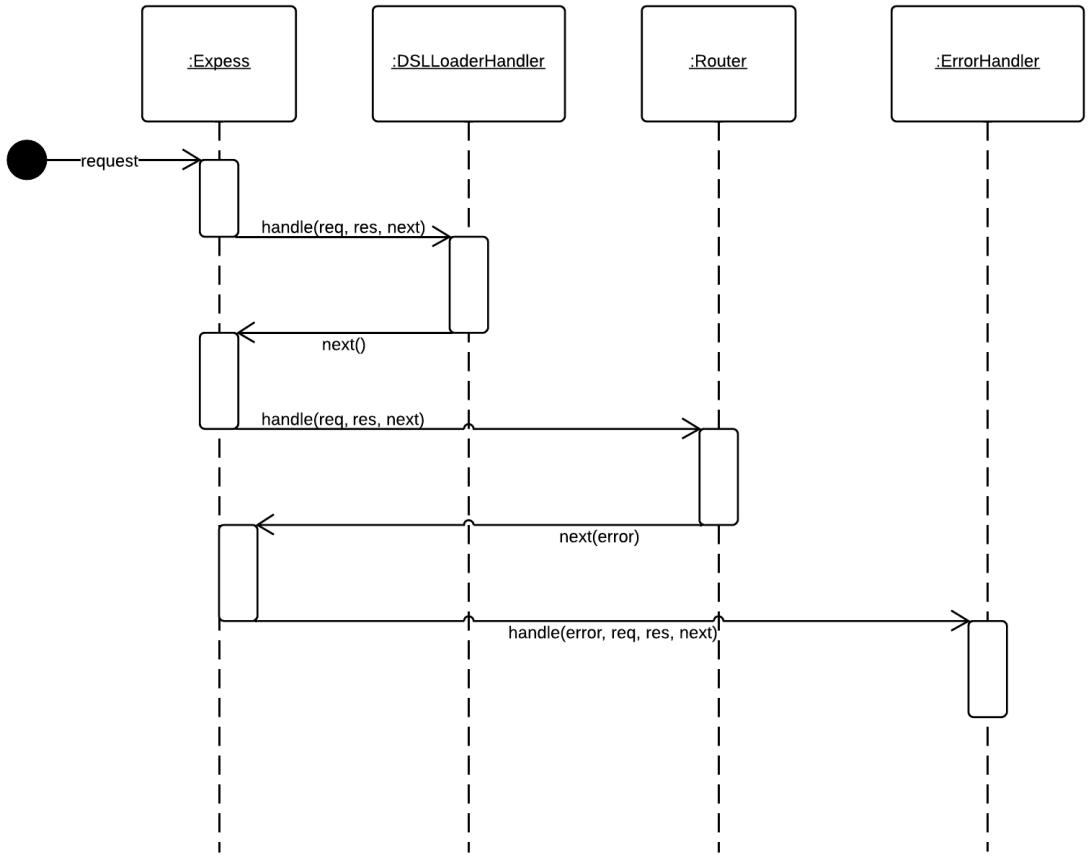

Figura 15: Diagramma di sequenza per la gestione di una richiesta

Nel seguente diagramma di sequenza viene mostrato il comportamento del middleware di routing, il quale internamente confronta l'uri della richiesta con alcune espressioni regolari per decidere a quale service passare il controllo.

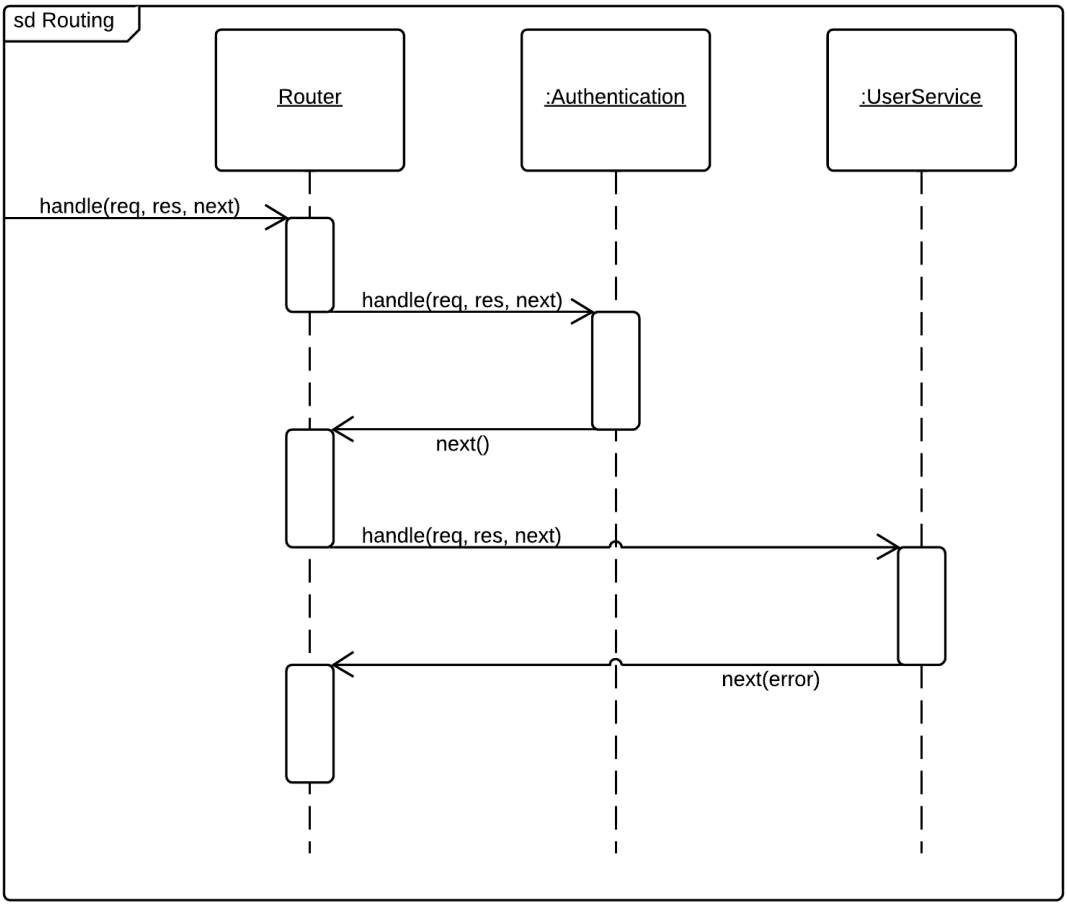

Figura 16: Diagramma di sequenza per il routing di una richiesta

4.3.2 Fallimento vincolo “utente autenticato”

Per ogni richiesta bisogna verificare che il permesso dell'utente che l'ha chiesta, corrisponda al permesso che la risorsa necessita per poter essere effettuata.
Tale scenario rappresenta il fallimento di una richiesta richiedente come vincolo per poter essere effettuata che l'utente abbia un permesso di tipo ”utente autenticato”, dove la verifica dei permessi è gestita da *Authentication* che manda un next(error) per il fallimento di tale vincolo al router il quale avrà compito di gestirlo.

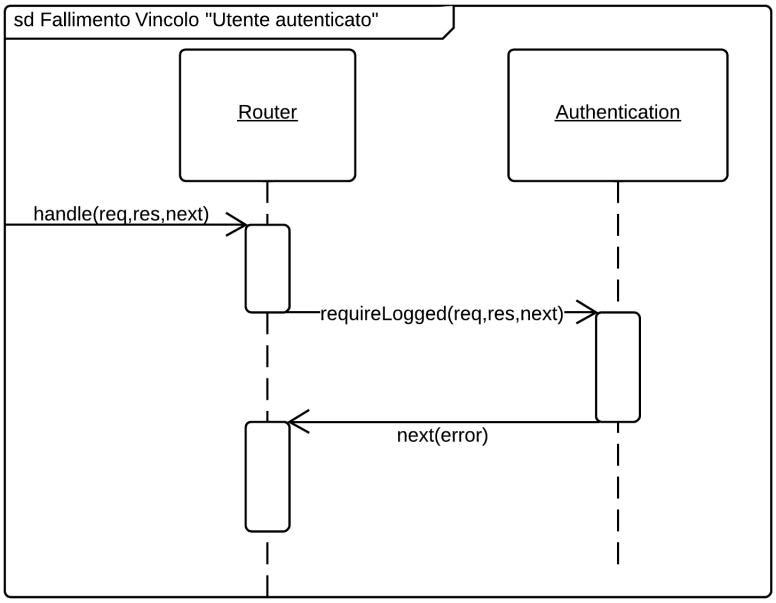

Figura 17: Fallimento vincolo “utente autenticato”

4.3.3 Fallimento vincolo “utente non autenticato”

Il seguente diagramma di sequenza rappresenta lo scenario in cui fallisce la verifica del vincolo di permesso ”utente autenticato”. La richiesta viene gestita da *Authentication* che verifica con esito negativo i permessi e rimanda un next(error) al router.

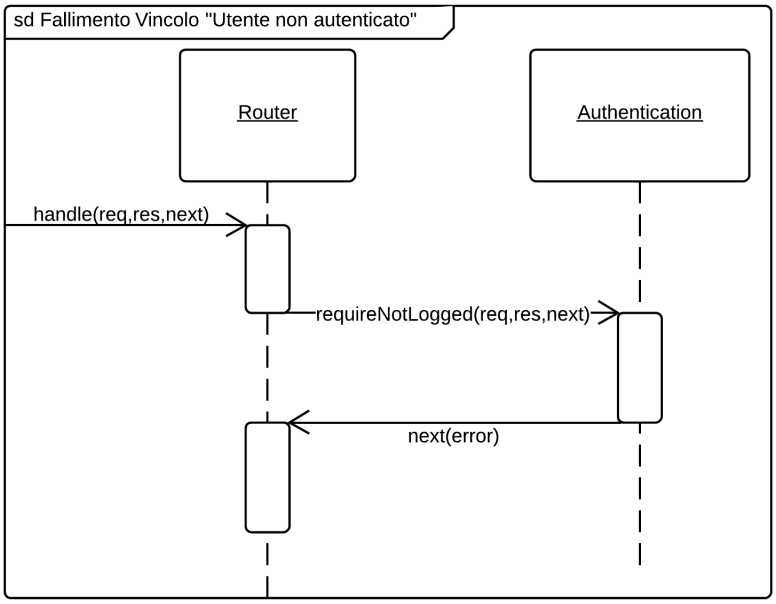

Figura 18: Fallimento vincolo “utente non autenticato”

4.3.4 Fallimento vincolo “utente admin”

Nel diagramma seguente viene rappresentato lo scenario in cui si richiedono permessi ”Admin” per poter gestire la richiesta corrispondente e la verifica effettuata dal middleware *Authentication* fallisce.

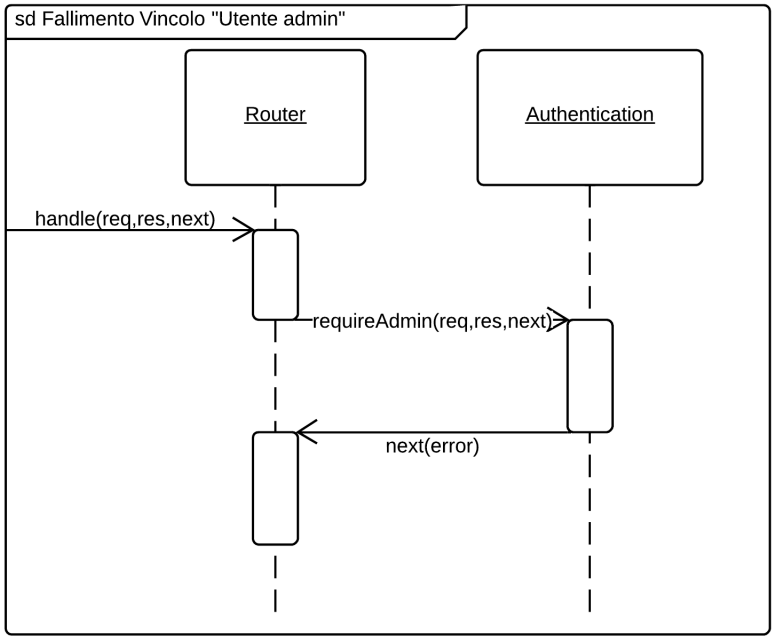

Figura 19: Fallimento vincolo “utente admin”

4.3.5 Fallimento vincolo “utente super admin”

Nel diagramma seguente viene rappresentato lo scenario in cui si richiedono permessi ”Super Admin” per poter gestire la richiesta corrispondente e la verifica effettuata dal middleware *Authentication* fallisce.

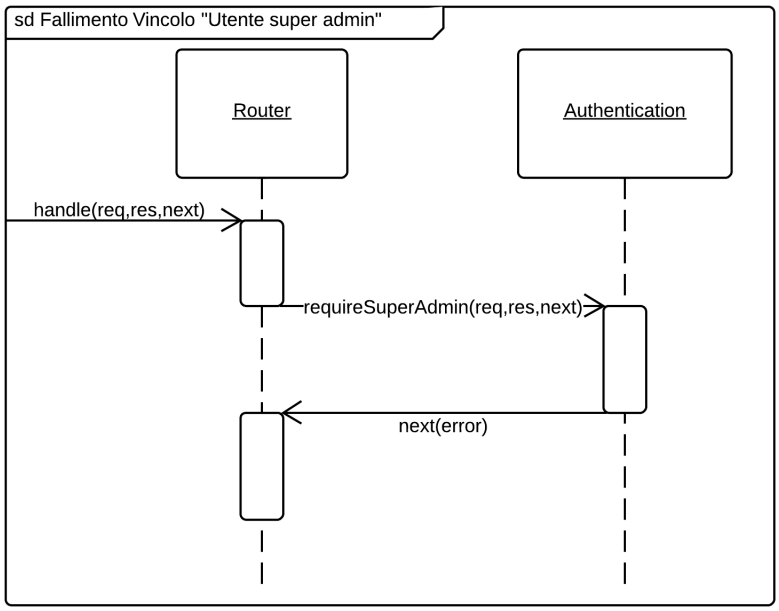

Figura 20: Fallimento vincolo “utente super admin”

4.3.6 Richiesta POST /profile

Il seguente scenario mostra la gestione di una richiesta POST per la risorsa profile, richiamato *requireNotLogged()*, *Authentication* non risponde con errore, chiamando successivamente *ProfileService* che gestisce la verifica dei parametri e nell'opzione che questa fallisca, manda in risposta un next(error) che il router andrà a gestire.

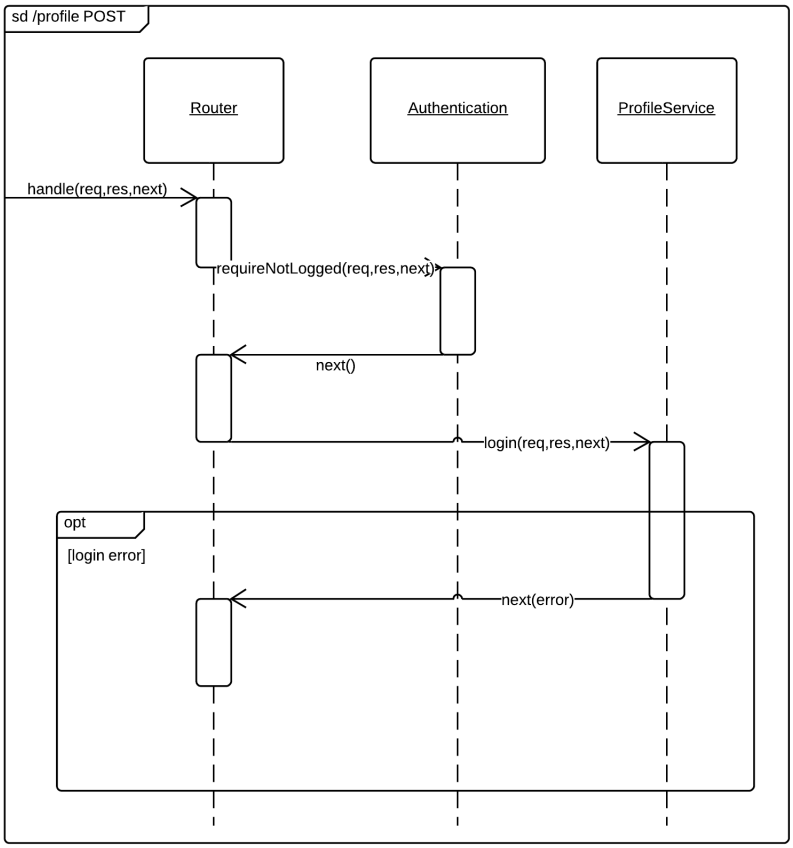

Figura 21: Richiesta POST /profile

4.3.7 Richiesta DELETE /profile

Il diagramma di sequenza mostra lo scenario di una richiesta DELETE per la risorsa profile. La verifica dei permessi in *Authentication* non fallisce, innescando la chiamata al successivo $middleware_G$ *ProfileService* che gestisce l'eliminazione della risorsa.

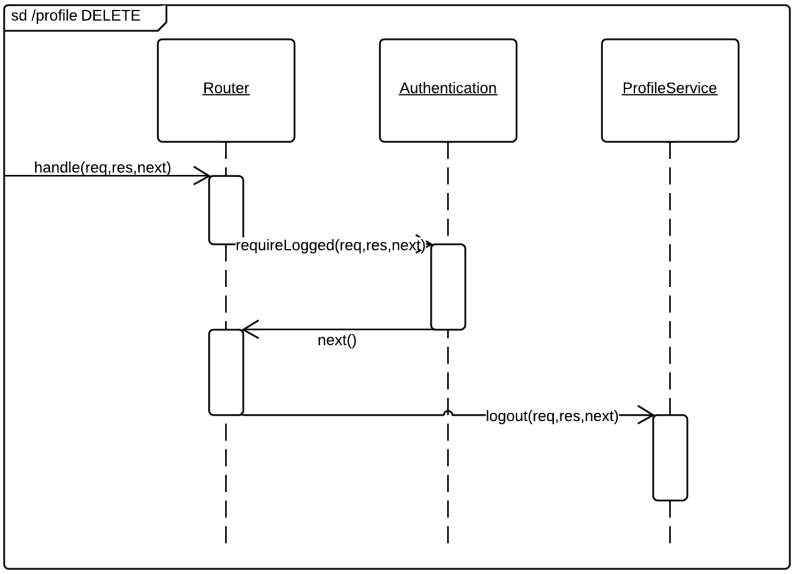

Figura 22: Richiesta DELETE /profile

4.3.8 Richiesta GET /profile

Il seguente diagramma rappresenta lo scenario di una richiesta GET per ottenere la risorsa Profile, la verifica dei permessi non fallisce e la richiesta viene gestita da *ProfileService*.

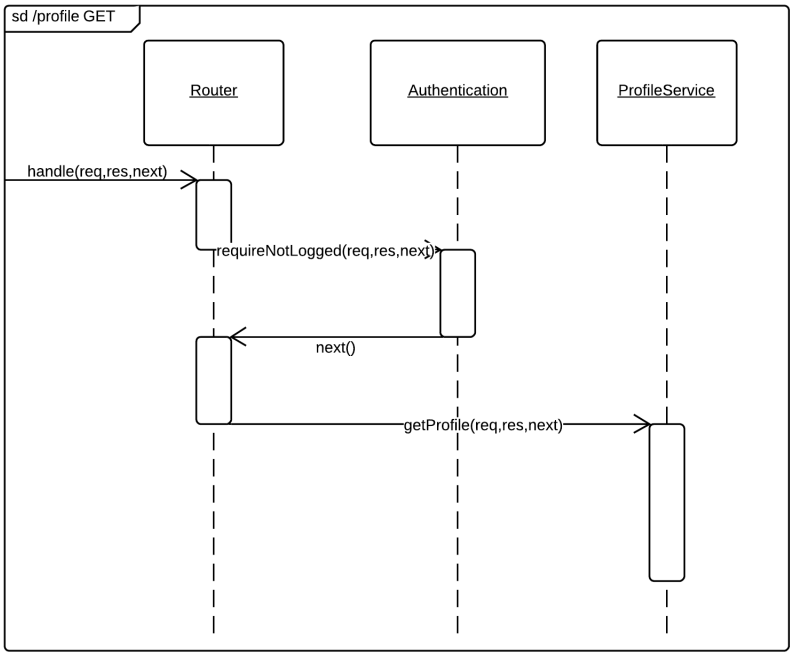

Figura 23: Richiesta GET /profile

4.3.9 Richiesta PUT /profile

Viene rappresentato lo scenario di una richiesta PUT per la risorsa Profile, il *requireLogged()* non fa ritornare un errore, viene chiamato il successivo $middleware_G$ *ProfileService* chiamando il metodo *updatePassword()* che gestisce la richiesta di modifica della password del profilo. Nell'opzione che i parametri passati per la modifica del profilo siano errati, *updatePassword()* chiamerà la callback passandogli la descrizione dell'errore come parametro che il router andrà a gestire.

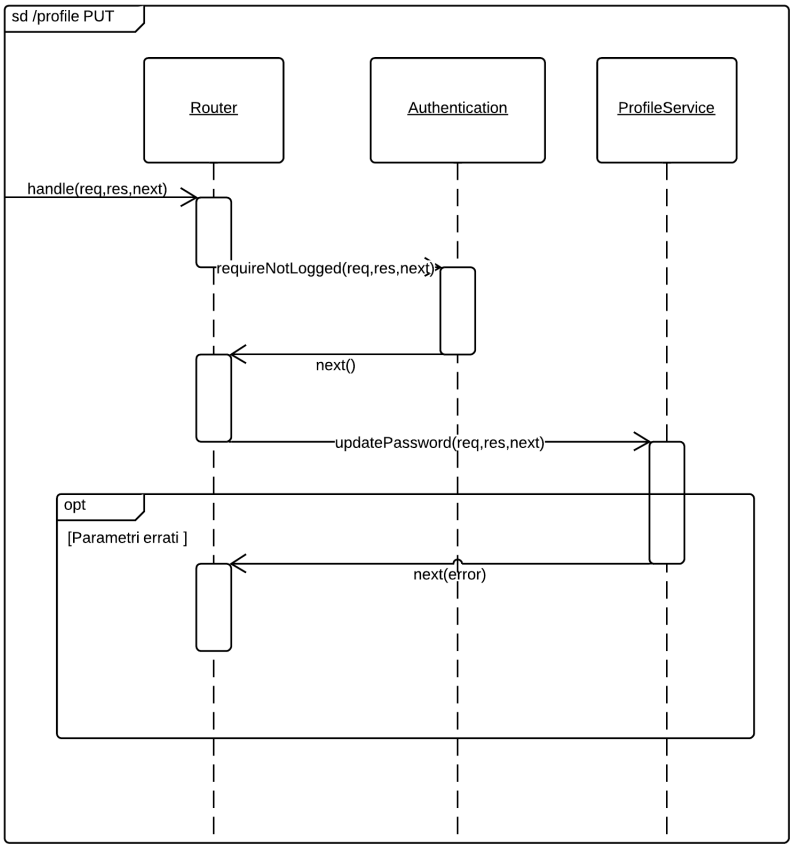

Figura 24: Richiesta PUT /profile

4.3.10 Richiesta POST /password/forgot

Viene rappresentato lo scenario di una richiesta POST per la risorsa password forgot, *Authentication* non risponde errore e il controllo passa a *ForgotService* che gestisce la richiesta. Se i parametri passati per la richiesta sono errati, il service *Authentication* risponderà con un errore.

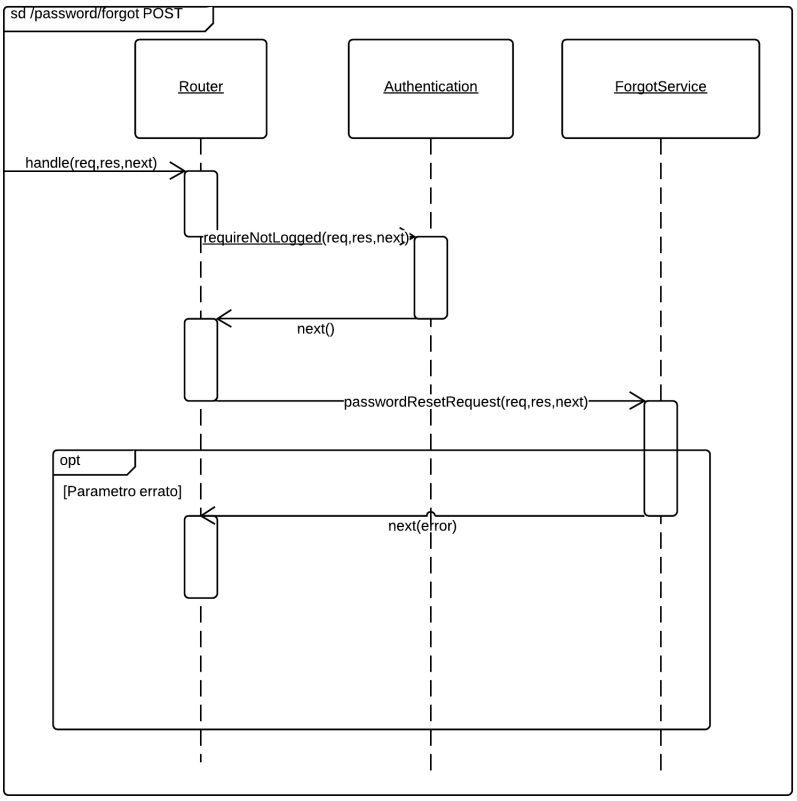

Figura 25: Richiesta POST /password/forgot

4.3.11 Richiesta GET /users

Viene rappresentato nel seguente diagramma di sequenza lo scenario di una richiesta GET per la risorsa user, *requireAdmin()* non fallisce ed il controllo viene passato a *UserService* che gestisce la richiesta.

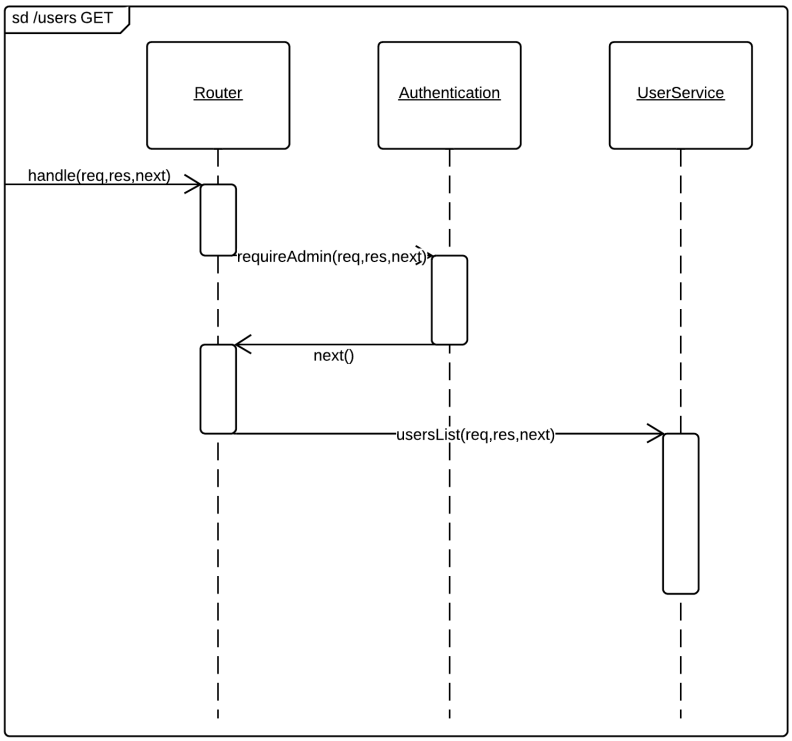

Figura 26: Richiesta GET /users

4.3.12 Richiesta POST /users

Il seguente diagramma di sequenza rappresenta lo scenario di una richiesta POST per la risorsa user, la verifica di *requireLogged()* dei permessi utente non fallisce e viene passato il controllo a *UserService* tramite chiamata del metodo *createUser()* che gestisce la richiesta di creazione di un nuovo user, e nel caso la verifica dei parametri passati per la creazione di quest'ultimo siano errati, il service chiamerà la callback con l'errore.

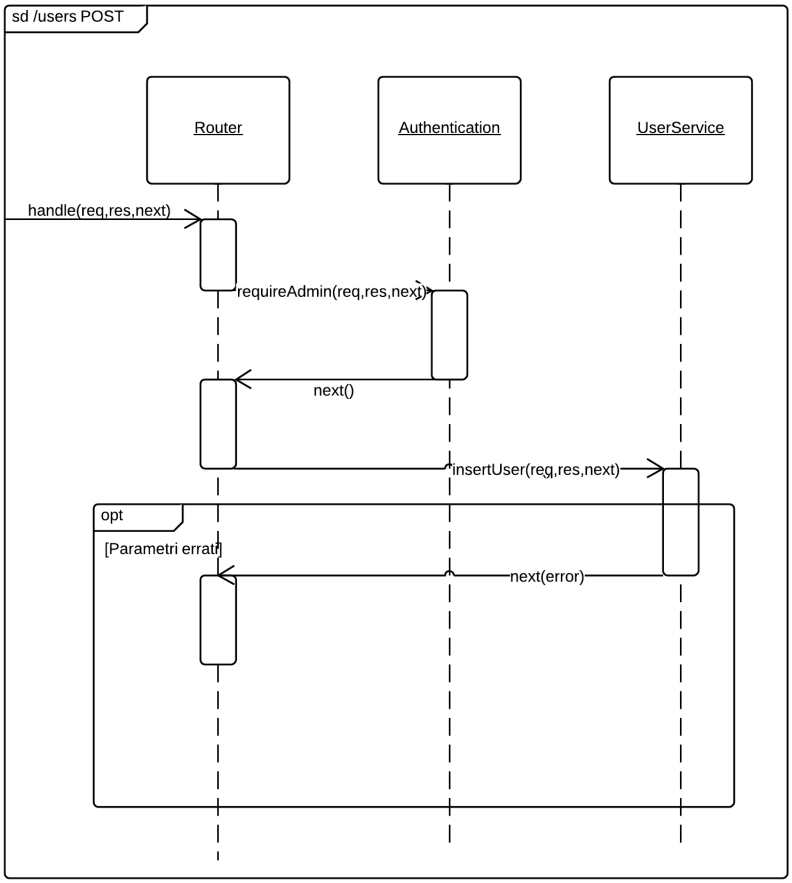

Figura 27: Richiesta POST /users

4.3.13 Richiesta GET /users/{user id}

Lo scenario rappresenta una richiesta GET di una risorsa User id, vengono verificati i permessi attraverso *requireAdmin()* che passa poi il controllo a *userService* invocando il metodo *getUser()* dandogli come attributo l'id dell'user da restituire. Nell'opzione che l'id dell'user sia errato, non corrispondendo a nessun user esistente, verrà richiamata una next(error).

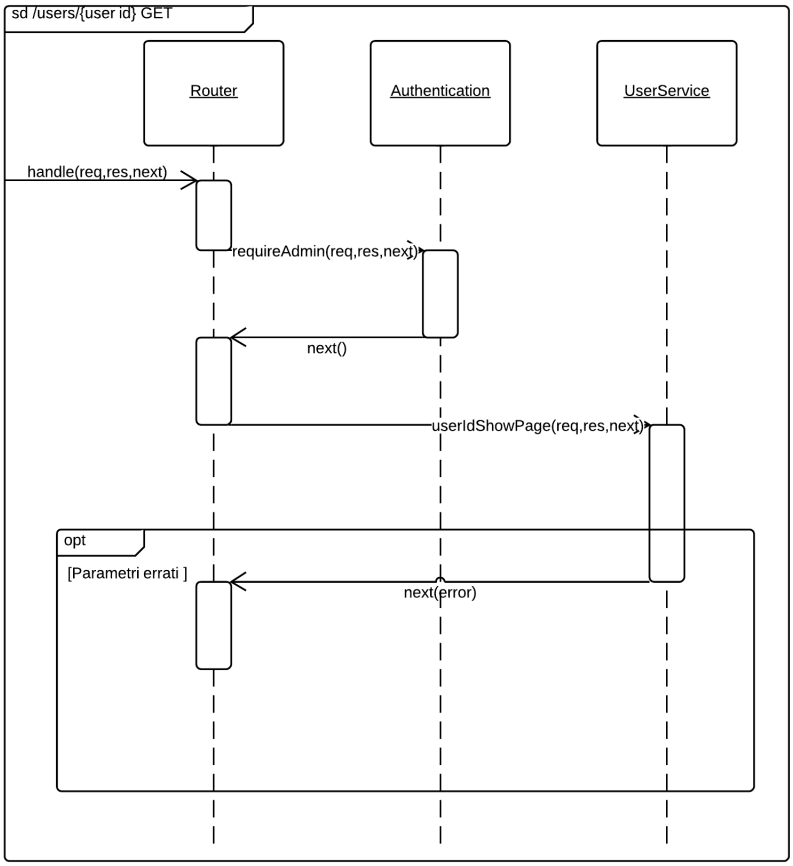

Figura 28: Richiesta GET /users/{user id}

4.3.14 Richiesta PUT /users/{user id}

Nel seguente diagramma di sequenza viene rappresentato lo scenario di una richiesta PUT per la risorsa User id nel quale *requireAdmin()* non restituisce un errore e passa il controllo a *userService* che gestisce la richiesta di modifica del livello corrispondente all'userId che gli è stato passato come attributo. *updateLevel()* verifica inoltre che l'userId passatogli come attributo non corrisponda all'id dello stesso user che ha effettuato la richiesta o corrisponda ad un id di un superAdmin e controlla che i parametri passati non siano errati, altrimenti risponderà con errore.

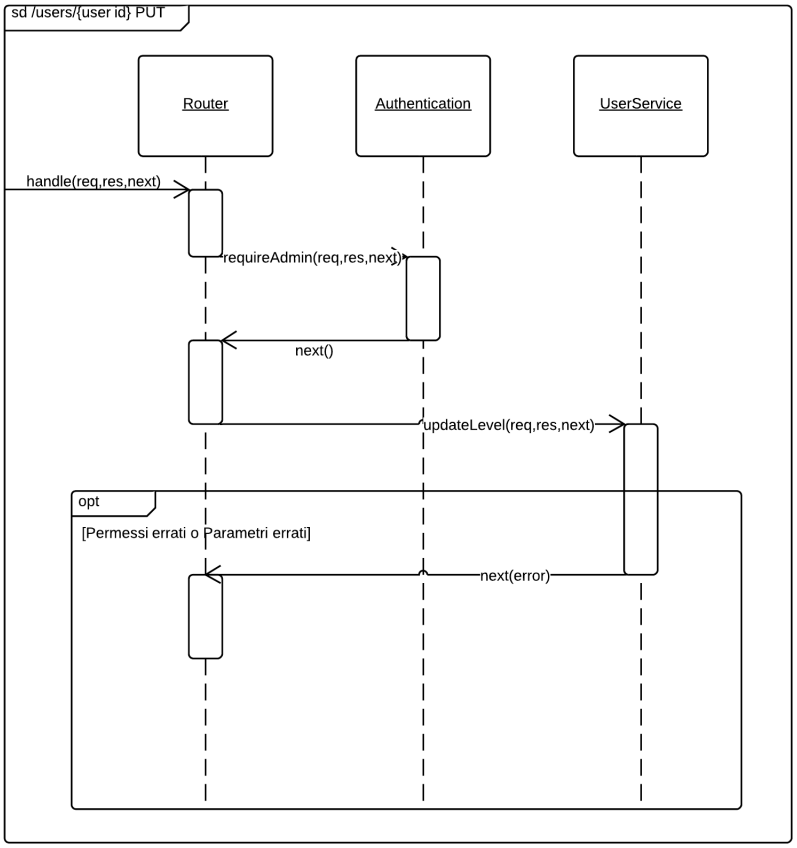

Figura 29: Richiesta PUT /users/{user id}

4.3.15 Richiesta DELETE /users/{user id}

Il diagramma di sequenza rappresenta lo scenario di una richiesta DELETE per la risorsa user id, *requireAdmin()* non restituisce un errore e la richiesta viene gestita da *deleteUser()* per procede con l'eliminazione dell'user cui id gli è stato passato come attributo. Nell'opzione che l'id passatogli corrisponda all'id dell'utente che ha effettuato la richiesta o ad un id di un superAdmin, *UserService* restituisce un next(error). Il service si preoccupa inoltre di verificare che i parametri siano corretti.

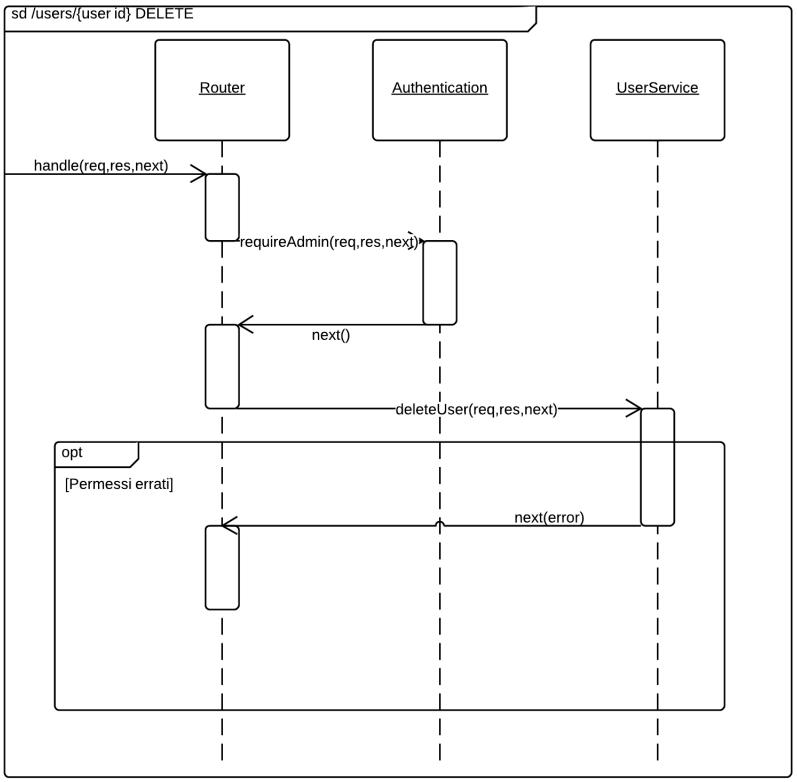

Figura 30: Richiesta DELETE /users/{user id}

4.3.16 Richiesta GET /collection

Il diagramma seguente rappresenta lo scenario di una richiesta GET per la risorsa collection, il service *Authentication* innescherà la chiamata del successivo service *CollectionService* che gestirà la richiesta di restituzione della lista di $Collection_G$.

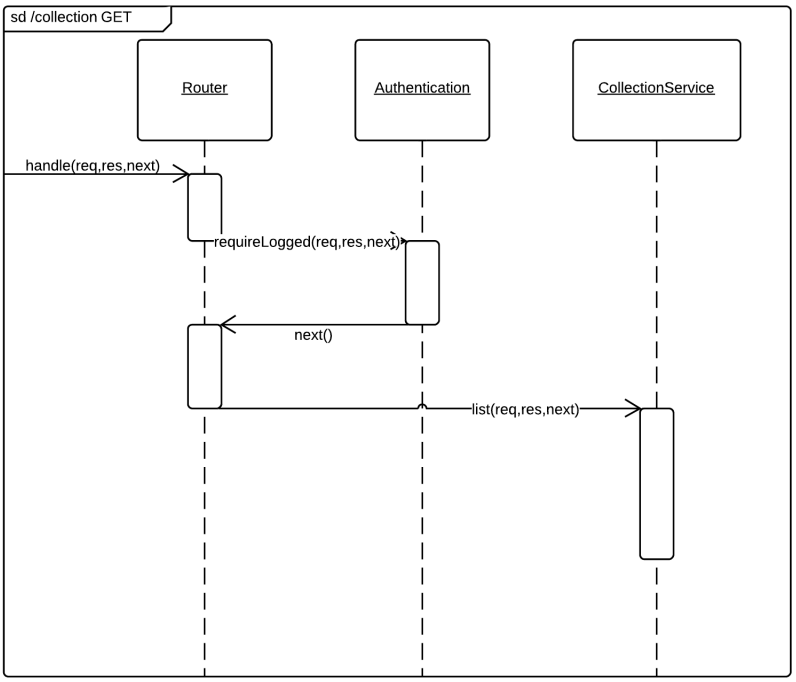

Figura 31: Richiesta GET /collection

4.3.17 Richiesta GET /collection/{collection name}

Il diagramma seguente rappresenta lo scenario di una richiesta GET per la risorsa collection Name, il service *Authentication* innescherà la chiamata del successivo service *indexService* al quale verrà passato come parametro l'id della $Collection_G$ per la restituzione dell'index page corrispondente.
Nell'opzione che l'id sia errato il service chiamerà la callback passandogli la descrizione dell'errore come parametro.

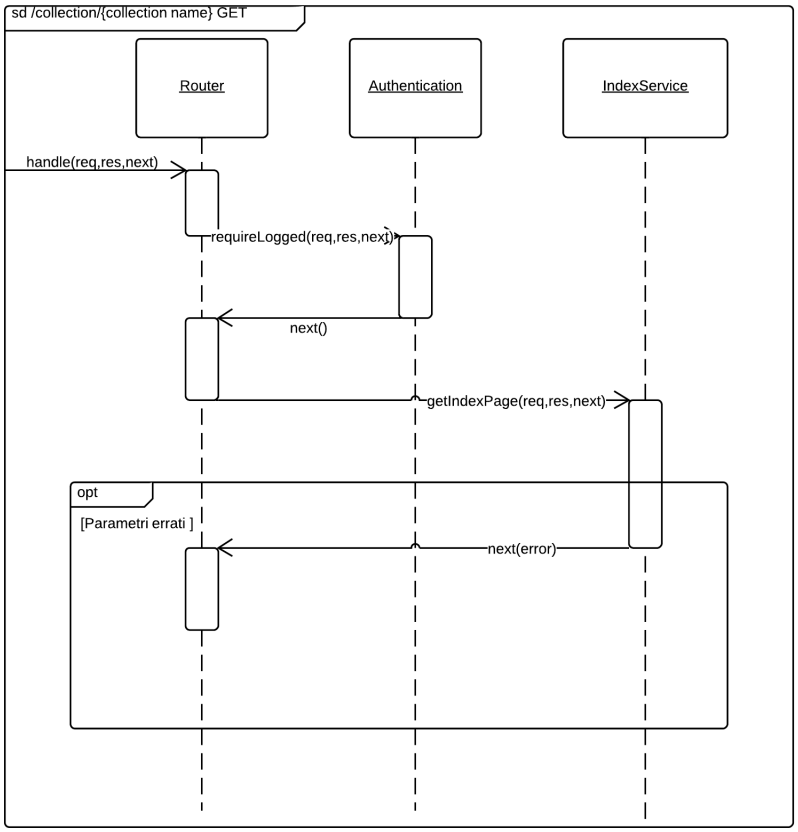

Figura 32: Richiesta GET /collection/{collection name}

4.3.18 Richiesta GET /collection/{collection name}/{document id}

Il diagramma di sequenza rappresenta lo scenario di una richiesta GET per la risorsa collection name document, nel quale al service *showService* viene passato l'id del document di cui mostrare la show page. Nell'opzione l'id sia errato, non corrispondendo ad un $Document_G$ valido, *showService* restituisce una next(error).

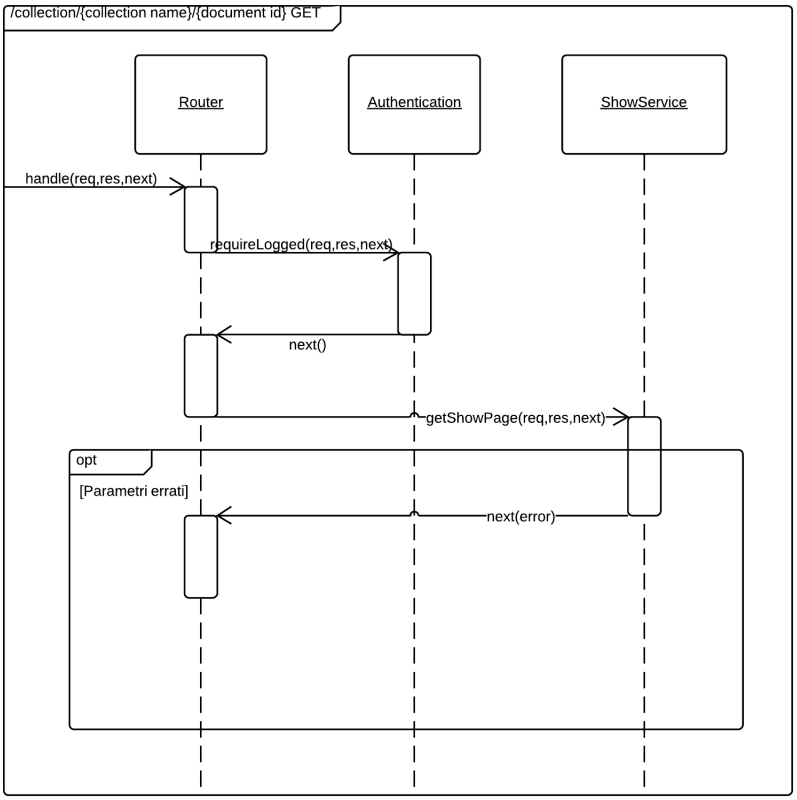

Figura 33: Richiesta GET /collection/{collection name}/{document id}

4.3.19 Richiesta DELETE /collection/{collection name}/{document id}

Il seguente scenario rappresenta la richiesta DELETE per una risorsa Collection name document, dopo che i permessi sono stati verificati il controllo è passato a *ShowService* il quale gestirà la richiesta di eliminazione del document il cui id gli è stato passato come parametro. Se l'id è errato, verrà restituito un errore.

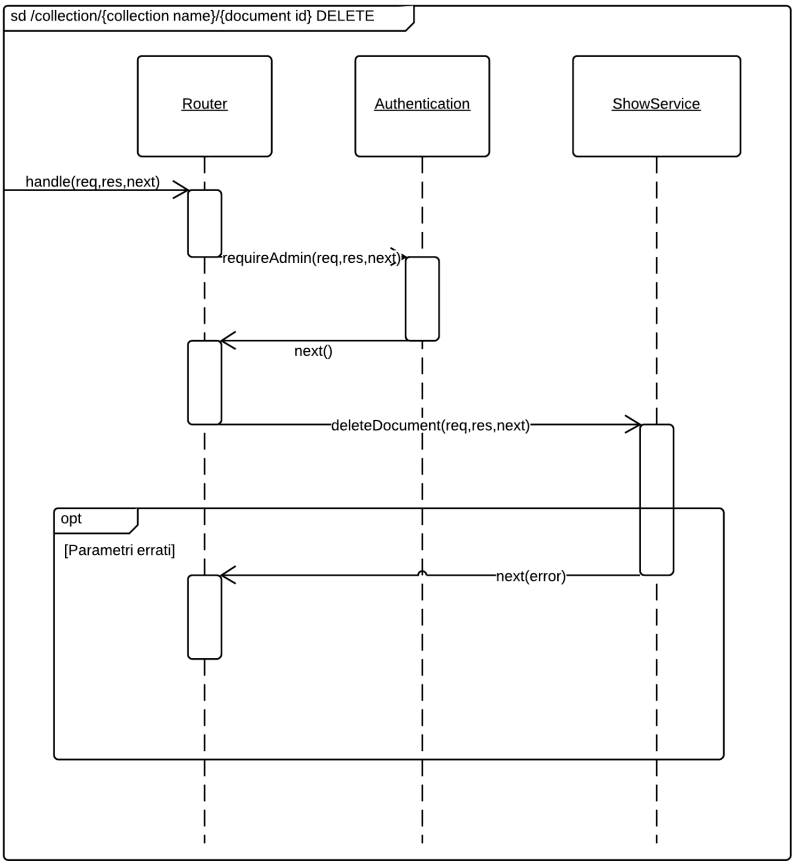

Figura 34: Richiesta DELETE /collection/{collection name}/{document id}

4.4 Descrizione librerie aggiuntive

Vengono di seguito descritte le librerie aggiuntive utilizzate dal Back-end che corrispondono nei diagrammi precedenti ai packages colorati. La scelta è stata effettuata cercando di valutare la diffusione, il livello di stabilità, l'assenza di errori noti.

- **Passport**: è un middleware il cui unico scopo è gestire le richieste di autenticazione per *Node.js*$_G$. Estremamente flessibile e modulare, Passport può essere facilmente inserito in qualsiasi applicazione web basata su Express. Con Passport è possibile cambiare il meccanismo di autenticazione, ovvero la *strategia* da adottare senza creare dipendenze superflue, scegliendo tra quelle proposte dalla libreria o personalizzandone per adattarle alle proprie esigenze. Inoltre, Passport mantiene persistentemente le sessioni di login e permette di configurare agevolmente le azioni in caso di fallimento o successo dell'autenticazione;
- **Passport-local**: è una libreria che permette di autenticare un utente con Passport identificandosi come una *strategia* precedentemente citata, utilizzando uno username e una password. Come Passport, si integra facilmente con qualsiasi applicazione o framework che supporta lo stile di connessione tra le componenti tramite il middleware, incluso Express;
- **Passport-local-mongoose**: è un plugin per Mongoose che semplifica la costruzione di un sistema di autenticazione con Passport. Permette di inserire l'email per l'autenticazione, aggiunge allo schema l'hash della password e il valore salt;
- **Nodemailer**: è un modulo che permette di mandare facilmente e-mail con Node.js tramite *SMTP*$_G$. Tra le varie caratteristiche, supporta i caratteri *Unicode*$_G$, contenuto *HTML*$_G$ e testo non formattato, allegati, immagini integrate nel codice *HTML*$_G$, i protocolli di comunicazione *SSL*$_G$ e *STARTTLS*$_G$;
- **Sweet.js**: è un modulo che permette di definire macro utilizzando la sintassi del linguaggio Javascript. Con Sweet.js è possibile modellare un linguaggio *ad hoc* basato su JavaScript, collezionare le macro in un singolo modulo e condividerlo sul *Node Packaged Modules*$_G$.

5 Front-end

5.1 Descrizione packages e classi

5.1.1 Front-end

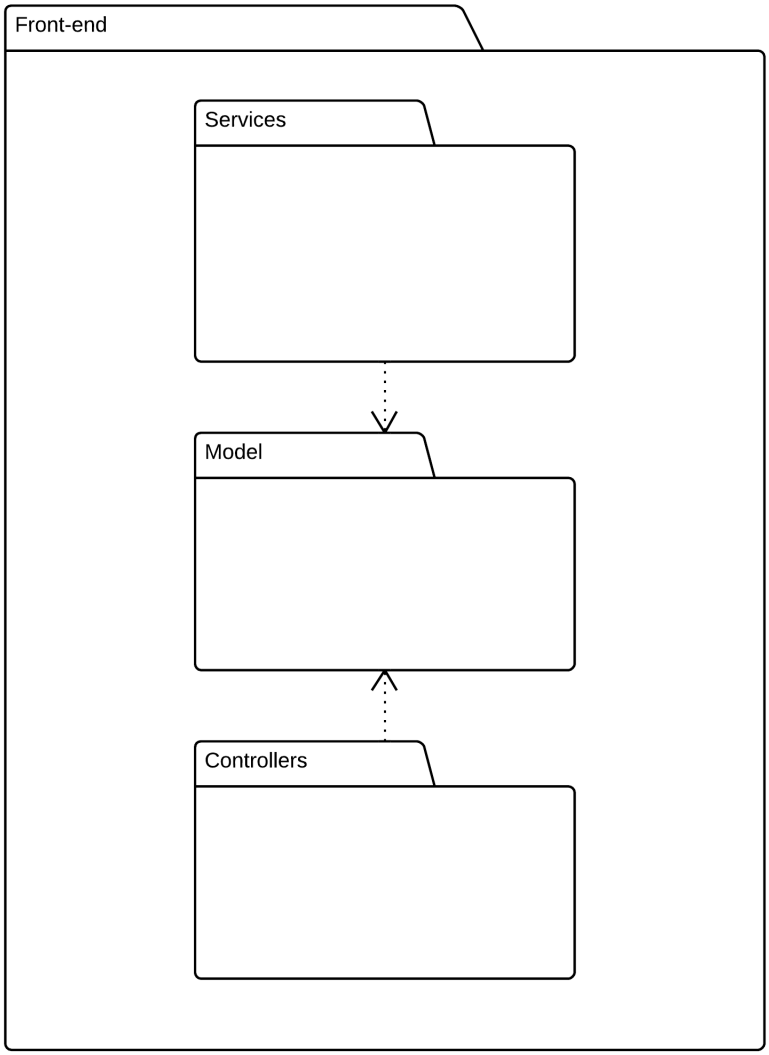

Figura 35: Diagramma dei packages Front-end

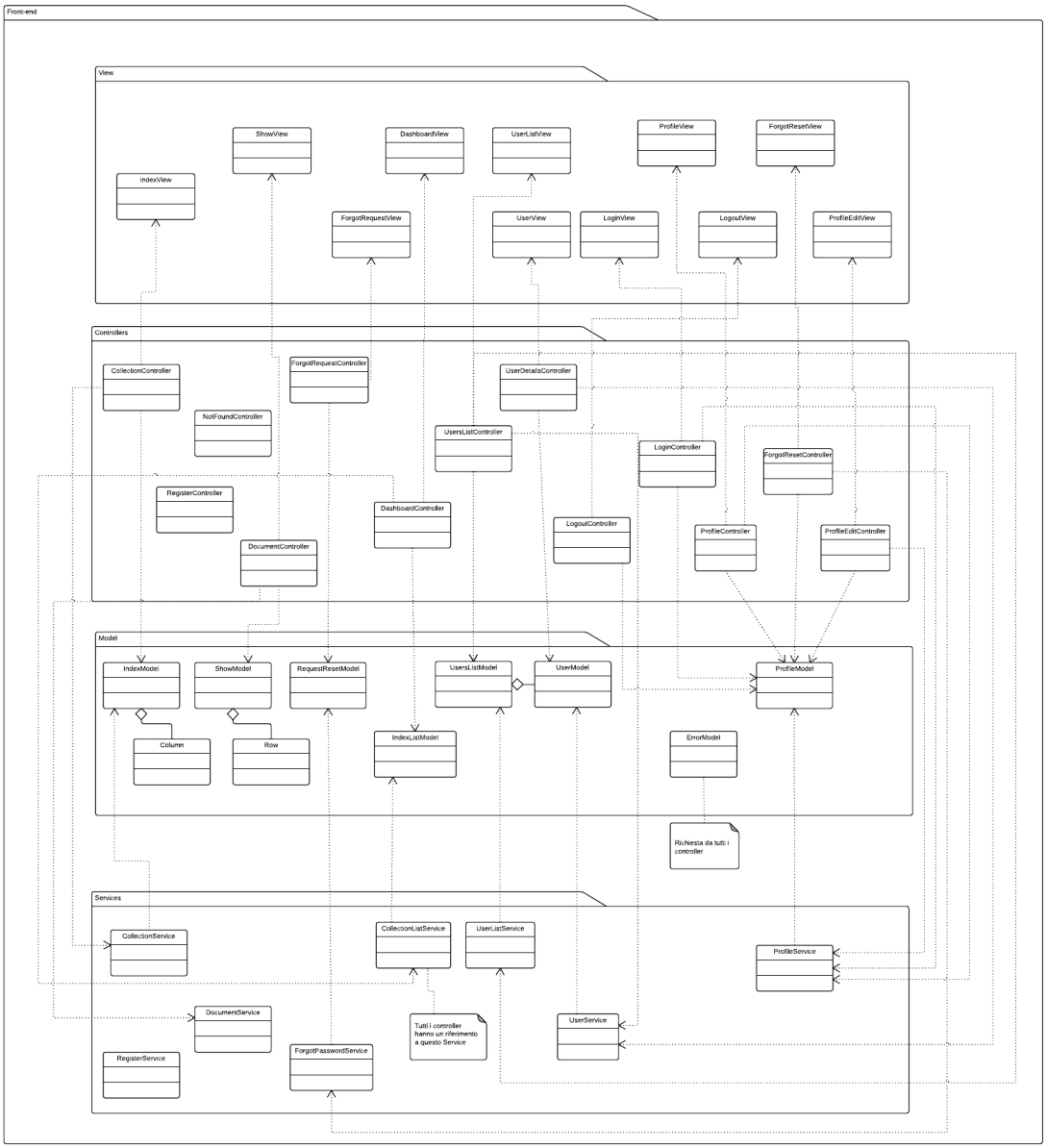

Figura 36: Diagramma delle classi Front-end

5.1.1.1 Informazioni sul package

5.1.1.1.1 Descrizione

Package$_G$ che racchiude tutta la componente di *Front-end*$_G$. Comprende il sottosistema che viene eseguito nei browser degli utenti e che fornisce l'interfaccia grafica all'utente non-tecnico che utilizzerà l'applicazione.

5.1.1.1.2 Package contenuti

- View
- Controllers
- Services
- Model

5.1.2 Front-end::Services

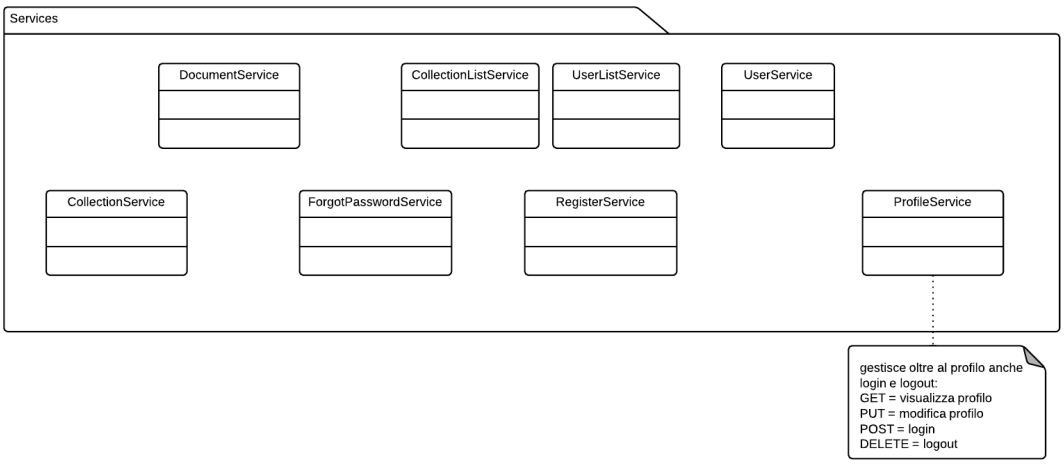

Figura 37: Componente Services

5.1.2.1 Informazioni sul package

5.1.2.1.1 Descrizione

Package$_G$ comprendente le classi che descrivono i meccanismi con cui il *Front-end*$_G$ può interfacciarsi con le *API*$_G$ del *Back-end*$_G$. Permette di recuperare i dati da inserire nel model e permette di azionare determinate procedure sul *Back-end*$_G$ (per esempio la richiesta di recupero password o le "action" definite nel DSL).

5.1.2.2 Classi

5.1.2.2.1 UserListService

Descrizione

Questa classe permette il recupero delle risorse REST rappresentanti gli utenti registrati all'applicazione tramite la chiamata /users

Utilizzo

La funzionalità offerta dalla classe è quella di poter fornire al Controller la lista degli utenti presenti nel database delle credenziali. Tale funzionalità richiede che l'utente sia un admin.

Relazioni con altre classi

- Front-end::Model::UsersListModel

5.1.2.2.2 ForgotPasswordService

Descrizione

Questa classe si occupa di inviare al server una richiesta di recupero password tramite la chiamata /password/lost e la conseguente modifica attraverso la chiamata /password/reset.

Utilizzo

La funzionalità offerta dalla classe è quella di interagire col server delegando quest'ultimo all'invio di una mail all'utente per il recupero della password e successivamente alla sua modifica.

Relazioni con altre classi

- Front-end::Model::RequestResetModel

5.1.2.2.3 CollectionService

Descrizione

Questa classe permette il recupero della risorsa REST rappresentante la Collection tramite la chiamata /collection/{collection_id}/{page}/{sort}/{order}

Utilizzo

La funzionalità offerta dalla classe è quella di poter fornire al Controller la lista di Document presenti nella Collection.

Relazioni con altre classi

- Front-end::Model::CollectionModel

5.1.2.2.4 CollectionListService

Descrizione

Questa classe permette il recupero delle risorse REST rappresentanti le Collections tramite la chiamata / collections.

Utilizzo

La funzionalità offerta dalla classe è quella di poter fornire al Controller la lista delle Collections registrate dallo sviluppatore e presenti nel database delle collections. Tale funzionalità richiede che l'utente sia registrato.

Relazioni con altre classi

- Front-end::Model::CollectionListModel

5.1.2.2.5 ProfileService

Descrizione

Questa classe permette il recupero delle risorsa REST rappresentante il profilo utente tramite la chiamata /profile.

Utilizzo

Le funzionalità offerte dalla classe sono:

- elenco dei dati relativi all'utente (GET);
- modifica dei dati utente (PUT);
- creazione della sessione utente (POST);
- eliminazione della sessione utente (DELETE).

Per la funzionalità di visualizzazione dei dati, di modifica del profilo e di eliminazione della sessione è richiesto che l'utente sia autenticato.

Relazioni con altre classi

- Front-end::Model::ProfileModel

5.1.2.2.6 DocumentService

Descrizione

Questa classe permette il recupero delle risorse REST rappresentanti i Document di una Collection tramite la chiamata /collections/{collectionId}/{documentId}

Utilizzo

Le funzionalità offerte dalla classe sono:

- elenco dei dati relativi al Document
- modifica dei dati relativi al Document
- rimozione del Document

Tali funzionalità richiedono che l'utente sia autenticato al sistema.

5.1.2.2.7 UserService

Descrizione

Questa classe permette il recupero della risorsa REST rappresentante l'utente tramite la chiamata /users/{user_id}

Utilizzo

Le funzionalità offerte dalla classe sono:

- elenco dei dati relativi all'utente.
- modifica della password relativa al utente.
- elevare o declassare un utente ad admin
- rimozione dell'utente.

Tali funzionalità richiedono che l'utente sia un admin.

Relazioni con altre classi

- Front-end::Model::UserModel

5.1.3 Front-end::Controllers

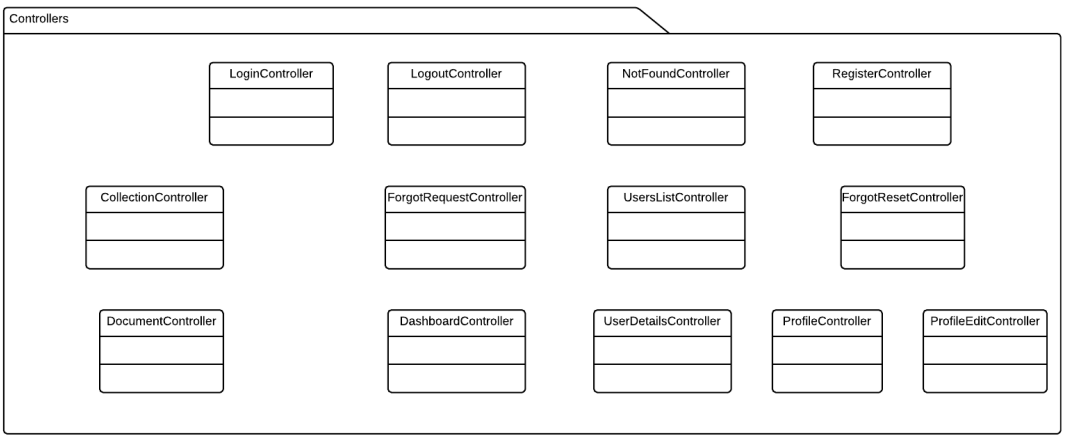

Figura 38: Componente Controllers

5.1.3.1 Informazioni sul package

5.1.3.1.1 Descrizione

Package$_G$ comprendente le classi che costituiscono i controller del componente *Front-end*$_G$. Ogni controller gestisce le operazioni e la logica applicativa riguardante una

determinata pagina, e specifica quale view verrà utilizzata per la presentazione all'utente dei dati.

5.1.3.2 Classi

5.1.3.2.1 LoginController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina di Login.

Utilizzo

Viene utilizzata per generare la pagina di login all'applicazione. Prima della creazione della view viene effettuato un controllo sull'esistenza di una sessione utente. In caso positivo il controller si occuperà di visualizzare una pagina nella quale l'utente verrà avvertito che un'autenticazione è già stata effettuata, altrimenti si procederà alla pagina di Login predefinita. Una volta che richiede un'autenticazione viene utilizzata classe `Front-End::Services::ProfileService`, la quale si occuperà di comunicare con il Back-End, il quale effettuerà il controllo sulle credenziali e in caso positivo effettuerà l'autenticazione dell'utente.

Relazioni con altre classi

- Front-end::View::LoginView

5.1.3.2.2 LogoutController

Descrizione

Classe che gestisce l'operazione di logout di un utente.

Utilizzo

Questa controller si occupa di distruggere la sessione attuale, se esiste, e non genera una view ma reindirizza l'utente automaticamente alla pagina di Login.

5.1.3.2.3 ForgotResetController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina di reset della password.

Utilizzo

Si occupa di generare la pagina di reset, prelevare quindi la nuova password inserita dall'utente nella view e chiamare l'apposito service che si occuperà del reset interagendo con il back-end.

5.1.3.2.4 ProfileEditController

Descrizione

Classe che gestisce le operazioni di modifica di un utente attraverso la pagina di modifica profilo.

Utilizzo

Utilizza la casse `Front-End::Services::ProfileService` per popolare la classe `Front-End::Model::ProfileModel` con i dati dell'utente. Quest'ultima classe fornirà un metodo accessorio attraverso il quale il controller può ottenere i dati e generare la pagina popolando correttamente lo scope della classe `Front-End::View::ProfileView`.

Relazioni con altre classi

- Front-end::View::ProfileEditView

5.1.3.2.5 UserDetailsController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina profilo di un utente visualizzabile dall'admin.

Utilizzo

Utilizza la classe `Front-End::Service::UserService`, che si occupa di popolare la classe `Front-End::Model::UserModel` con i dati dell'utente richiesto. Quest'ultima classe conterrà un metodo accessorio tramite il quale il controller può prelevare i dati e generare la pagina popolando correttamente lo scope della classe `Front-End::View::UserView`.

Relazioni con altre classi

- Front-end::View::UserDetailsView

5.1.3.2.6 ProfileController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina profilo di un utente.

Utilizzo

Utilizza la classe `Front-End::Services::ProfileService` per popolare la classe `Front-End::Model::ProfileModel` con i dati dell'utente che ha effettuato la richiesta. Quest'ultima classe fornirà un metodo accessorio attraverso il quale il controller potrà prelevare i dati e generare la pagina, popolando correttamente lo scope.

Relazioni con altre classi

- Front-end::View::ProfileView
- Front-end::Services::ProfileService

5.1.3.2.7 CollectionController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina di gestione della Collection.

Utilizzo

Utilizza la classe `Front-End::Services::CollectionService` per popolare corretta: mente la classe `Front-End::Model::IndexModel`. Quest'ultima fornirà un metodo accessorio attraverso il quale il controller può ottenere i dati e generare la pagina di visualizzazione di tutti i *Document*$_G$, popolando correttamente lo scope.

Relazioni con altre classi

- Front-end::Services::CollectionService
- Front-end::View::CollectionView

5.1.3.2.8 UsersListController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina di gestione degli utenti.

Utilizzo

Viene utilizzata per generare la pagina di visualizzazione della lista di utenti presenti nel: l'applicazione. In primo luogo utilizzerà la classe `Front-End::Services::UserListService` per popolare la classe `Front-End::Model::UserListModel` dalla quale otterrà in seguito la lista degli utenti attraverso una chiamata a una sua funzione.

Relazioni con altre classi

- Front-end::Services::UserListService
- Front-end::Services::UserService
- Front-end::View::UserListView

5.1.3.2.9 ForgotRequestController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina di richiesta di recupero password.

Utilizzo

Genera una pagina in cui viene visualizzato un campo di testo nel quale l'utente può inserire la propria mail ed effettuare una richiesta di ripristino password. Il controller permette quindi di inviare al Back-end la richiesta attraverso la classe `Front-End::Services::ForgotPasswordService`. Sarà poi compito del *Back-end*$_G$

inviare all'utente una mail contenente il link che bisogna aprire per poter scegliere una nuova password.

Relazioni con altre classi

- Front-end::Services::ForgotPasswordService
- Front-end::View::ForgotResetView

5.1.3.2.10 DashboardController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina dashboard.

Utilizzo

Viene utilizzata per generare la pagina dashboard, che fungerà da *home* dell'applicazione ovvero la prima pagina che un utente visualizza quando effettua l'autenticazione. Utilizza la classe `Front-End::Services::CollectionListService` per popolare correttamente tutte la classe `Front-End::Model::CollectionListModel`, dalla quale otterrà la lista delle $Collection_G$ registrate nell'applicazione mediante una chiamata a una sua funzione. A questo punto, una volta ottenuti i dati, il controller genera la pagina dashboard, popolando correttamente lo scope con i dati ottenuti.

Relazioni con altre classi

- Front-end::Services::CollectionListService
- Front-end::View::DashboardView

5.1.3.2.11 DocumentController

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina di gestione di un Document.

Utilizzo

Utilizza la classe `Front-End::Services::DocumentService` per popolare correttamente la classe `Front-End::Model::ShowModel`, la quale fornirà un metodo accessorio attraverso il quale il controller può ottenere i dati e generare la pagina popolando correttamente lo scope.

Relazioni con altre classi

- Front-end::Model::DocumentModel
- Front-end::View::DocumentView
- Front-end::Services::DocumentService

5.1.4 Front-end::Model

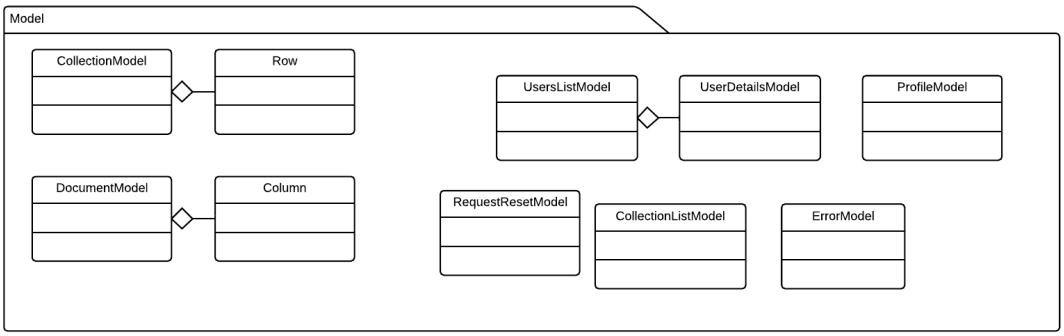

Figura 39: Componente Model

5.1.4.1 Informazioni sul package

5.1.4.1.1 Descrizione

Package$_G$ che comprende le classi dei modelli dei dati utilizzati dal *Front-end*$_G$. Servono a fornire ai controller e ai service le informazioni su quali campi potranno aspettarsi negli oggetti che arrivano tramite le *API*$_G$ del *Back-end*$_G$.

Le classi di questo *package*$_G$ sono state progettate, ma si prevede che non verranno codificate poiché verrà sfruttato lo stile di *duck-typing*$_G$ della gestione dei tipi di *JavaScript*$_G$.

5.1.4.2 Classi

5.1.4.2.1 ErrorModel

Descrizione

È la classe che rappresenta il modello dati dell'errore.

Utilizzo

Utilizzato da tutti i controller per poter accedere alle informazioni riguardanti l'errore.

5.1.4.2.2 ProfileModel

Descrizione

È la classe che rappresenta la struttura dati dell'utente.

Utilizzo

Permette al ProfileService di avere una rappresentazione delle informazioni dell'utente da scambiare con il back-end, al ProfileController e al ProfileEditController per ottenere il dati dell'utente da visualizzare nella view della pagina profilo e al ForgotResetController per la modifica della password.

5.1.4.2.3 RequestResetModel

Descrizione

È il modello che descrive i dati dell'utente che richiede un recupero della password.

Utilizzo

Fornisce una rappresentazione sotto forma di oggetto delle informazioni scambiate con il back-end e permette al ForgotPasswordService e al ForgotRequestController di poter accedere ai dati dell'utente.

5.1.4.2.4 UserModel

Descrizione

È la classe che rappresenta la struttura dati dell'utente.

Utilizzo

Fornisce una rappresentazione sotto forma di oggetto delle informazioni scambiate con il back-end e permette allo UserService e allo UserController di poter accedere agli attributi dell'utente.

5.1.4.2.5 CollectionListModel

Descrizione

È la classe che rappresenta la struttura dati delle Collections.

Utilizzo

Fornisce una rappresentazione sotto forma di oggetto delle informazioni scambiate con il back-end e permette alla CollectionListService e alla DashboardController di poter accedere alla lista delle Collections.

5.1.4.2.6 DocumentModel

Descrizione

È la classe che rappresenta la struttura dati dei Document relativi ad una Collection.

Utilizzo

Fornisce una rappresentazione sotto forma di oggetto delle informazioni scambiate con il back-end e permette al DocumentService e al DocumentController di poter accedere agli attributi del Document.

5.1.4.2.7 CollectionModel

Descrizione

È la classe che rappresenta il modello delle Collection.

Utilizzo

Fornisce una rappresentazione sotto forma di oggetto delle informazioni scambiate con il back-end e permette al CollectionService e al CollectionController di poter accedere alla lista delle Collections.

5.1.4.2.8 UsersListModel

Descrizione

È la classe che rappresenta la struttura dati dell'utente.

Utilizzo

Fornisce una rappresentazione sotto forma di oggetto delle informazioni scambiate con il back-end e permette allo UserListService e allo UserListController di poter accedere alla lista degli utenti.

5.1.5 Front-end::View

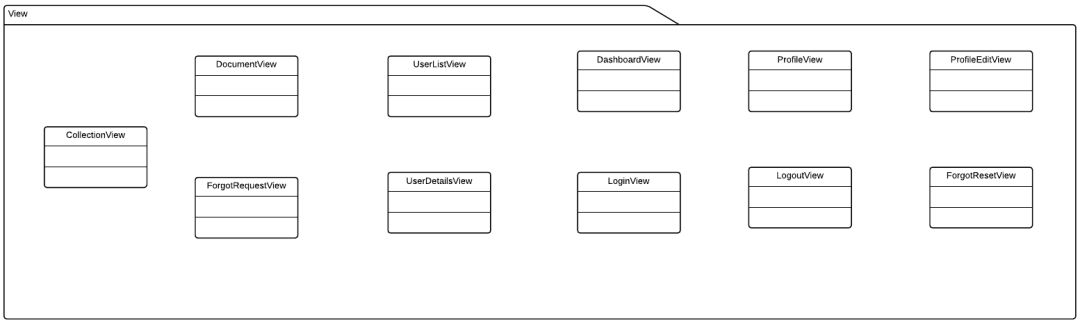

Figura 40: Componente View

5.1.5.1 Informazioni sul package

5.1.5.1.1 Descrizione

Package$_G$ comprendente le classi che costituiscono la view del componente *Front-end*$_G$. Ogni view rappresenta una pagina html con determinati campi `scope`, i quali verranno popolati con i dati richiesti.

5.1.5.2 Classi

5.1.5.2.1 DashboardView

Descrizione

Classe che descrive la pagina che visualizza la dashboard. In questo momento la dashboard contiene la lista delle collection presenti nel sistema.

Utilizzo

Viene utilizzata dalla classe `DashboardController` per generare la pagina dashboard.

5.1.5.2.2 LoginView

Descrizione

Questa classe si occupa di descrivere la pagina di login dell'applicazione mettendo a disposizione dell'utente un form all'interno del quale inserire email e password. Viene inoltre messo a disposizione un link per richiedere il ripristino della password.

Utilizzo

Viene utilizzata dalla classe `LoginController` per generare la pagina di Login dell'applicazione.

5.1.5.2.3 ProfileEditView

Descrizione

Questa classe descrive la pagina che si occupa di modificare i dati dell'utente attualmente autenticato.

Utilizzo

Viene utilizzato dalla classe `Front-end::Controller::ProfileEditController` per generare correttamente la pagina di modifica profilo.

5.1.5.2.4 CollectionView

Descrizione

La classe si occupa di descrivere la pagina che visualizza i documenti della collection selezionata.

Utilizzo

Viene utilizzata dalla classe `CollectionController` per generare la index-page di una Collection.

5.1.5.2.5 DocumentView

Descrizione

Classe descrive la pagina che visualizza le coppie chiave valore del documento selezionato.

Utilizzo

Viene utilizzata dalla classe `DocumentController` per generare la show-page di un Document.

5.1.5.2.6 ProfileView

Descrizione

Classe che rappresenta la pagina che visualizza le informazioni dell'utente attualmente autenticato.

Utilizzo

Viene utilizzata dalla classe `Front-end::Controller::ProfileController` per generare la pagina di visualizzazione del profilo dell'utente generato.

5.1.5.2.7 ForgotRequestView

Descrizione

Classe che rappresenta la pagina che permette all'utente di richiedere il reset della propria password tramite l'inserimento della propria email.

Utilizzo

5.1.5.2.8 UserListView

Descrizione

Questa classe si occupa di rappresentare la pagina contenente l'elenco di tutti gli utenti presenti nel sistema.

Utilizzo

Viene utilizzata dalla classe `Front-End::Controller::UserListController` per generare la pagina di visualizzazione degli utenti.

5.1.5.2.9 ForgotResetView

Descrizione

Classe che rappresenta la pagina che permette all'utente di resettare la propria password. Viene reindirizzato a questa pagine tramite un link presente nell'email ricevuta a seguito della compilazione di ForgotRequestView.

Utilizzo

Viene utilizzato dalla classe `Front-End::Controller::ForgotResetController` per generare correttamente la pagina di reset della password.

5.1.5.2.10 UserDetailsView

Descrizione

Questa classe si occupa di descrivere la pagina di visualizzazione delle informazioni sull'utente selezionato.

Utilizzo

Viene utilizzata dalla classe `Front-End::UserDetailsController` per generare correttamente la pagina di visualizzazione delle informazione di un utente.

6 Diagrammi di attività

Vengono in seguito illustrati i diagrammi di attività prodotti durante la progettazione architetturale, i quali descrivono le iterazioni dell'utente con il sistema $MaaP_G$. È stato scelto di dividere i diagrammi in due categorie principali, in modo analogo a quanto fatto nella descrizione dei casi d'uso dell'*Analisi dei requisiti*:

- **Applicazione $MaaP_G$**, in cui verranno descritte le iterazioni che un utente può fare all'interno di un'applicazione generata dal $framework_G$;
- **Framework $MaaP_G$**, in cui verrà descritto il modo in cui uno sviluppatore può creare un'applicazione.

Inizialmente per ogni categoria verrà fornito uno schema ad alto livello, per poi andare sempre più nel dettaglio tramite sotto-diagrammi più specifici. Per comodità di visualizzazione le attività che verranno *esplose* sono marcate in grassetto.

Al fine di rendere il diagramma leggibile abbiamo considerato implicito il fatto che un utente possa in qualsiasi momento uscire dall'applicazione $MaaP_G$, per esempio chiudendo la finestra del browser.

6.1 Applicazione MaaP

Vengono di seguito descritte tutte le iterazioni che un utente può effettuare con un'applicazione generata dal $framework_G$ $MaaP_G$.

6.1.1 Attività principali

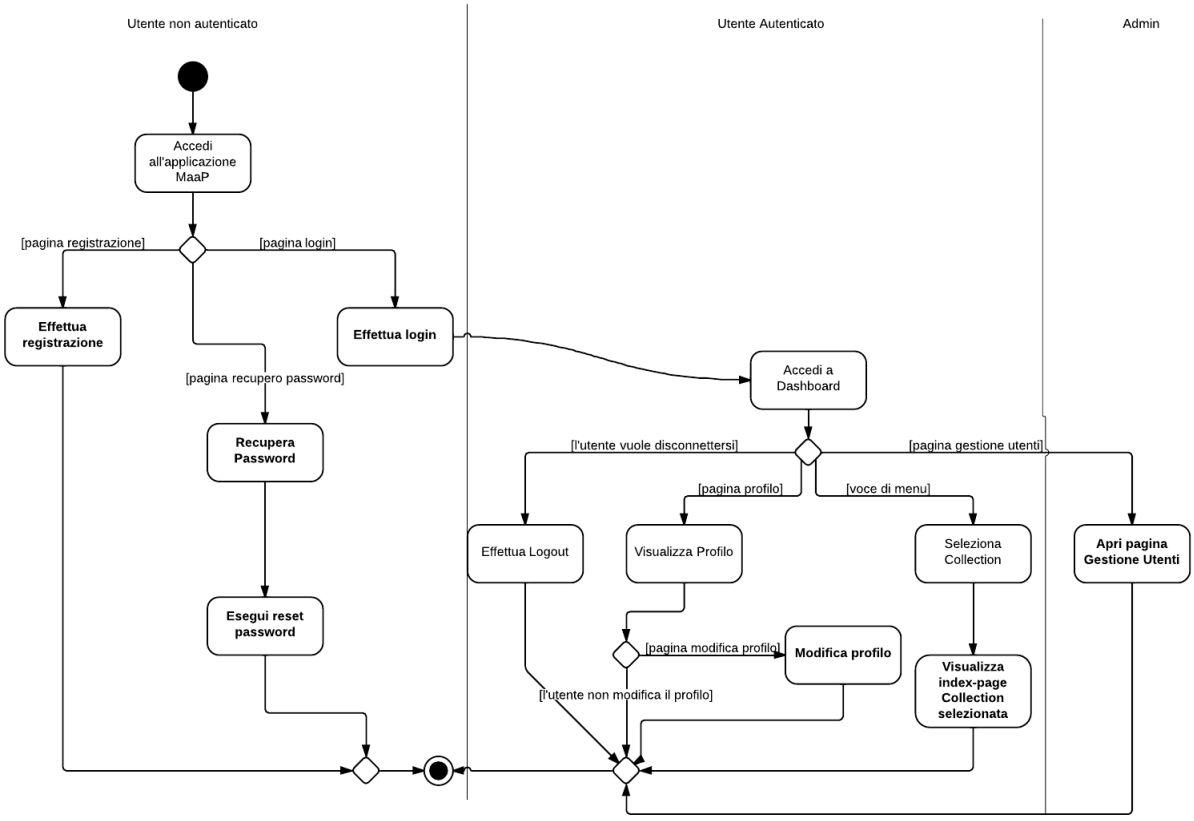

Figura 41: Diagramma di attività - attività principali di un'applicazione MaaP

Sostanzialmente un'applicazione generata da $MaaP_G$ è composta da una serie di pagine web all'interno delle quali un utente può navigare. Un utente accede inizialmente all'applicazione web in una pagina statica in cui può effettuare tre cose:

- Registrarsi al sistema;
- Effettuare il login;
- Recuperare la propria password.

Una volta che l'utente ha effettuato il login viene direttamente indirizzato alla $Dashboard_G$, dalla quale può navigare all'interno dell'applicazione ed effettuare diverse operazioni:

- Effettuare il logout;
- Visualizzare il proprio profilo e di conseguenza modificarlo;
- Selezionare una $Collection_G$ esistente.

Nel caso in cui l'utente avesse i privilegi di admin può inoltre accedere ad una specifica pagina di gestione degli utenti iscritti.

6.1.2 Effettua registrazione

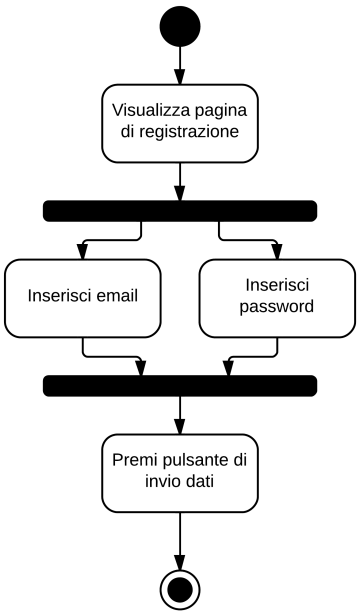

Figura 42: Diagramma di attività - Registrazione di un utente

L'utente si trova all'interno della pagina di registrazione e sostanzialmente deve inserire la propria email e la propria password all'interno di due campi di testo. Una volta inseriti l'utente deve premere il pulsante di invio dati; il sistema $MaaP_G$ procederà dunque alla verifica delle credenziali e, se quest'ultima avrà successo, alla registrazione dell'utente.

6.1.3 Recupera password

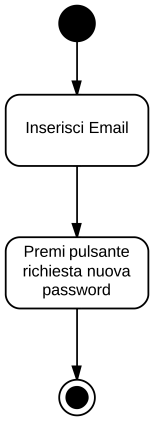

Figura 43: Diagramma di attività - Recupero password

L'utente si trova all'interno della pagina di recupero password, la quale presenta un campo di testo nel quale l'utente dovrà inserire il proprio indirizzo email. Una volta inserito preme il pulsante di richiesta di una nuova password; il sistema $MaaP_G$ procederà dunque alla verifica dell'indirizzo email e, se quest'ultima avrà esito positivo, invierà un'email all'utente con le relative istruzioni per il ripristino della password.

6.1.4 Esegui reset password

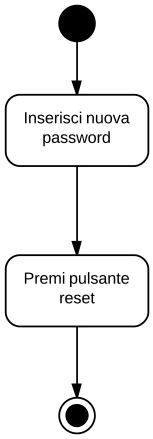

Figura 44: Diagramma di attività - Reset della password dell'utente

L'utente avrà ricevuto un'email con al suo interno un link ad una pagina univoca dell'applicazione $MaaP_G$ e quindi si troverà in una pagina con al suo interno un campo di testo nel quale inserire la nuova password. Una volta inserita la password deve premere il pulsante di reset; il sistema $MaaP_G$ procederà dunque al cambio password per l'utente corrente nel $database_G$ delle credenziali.

6.1.5 Effettua login

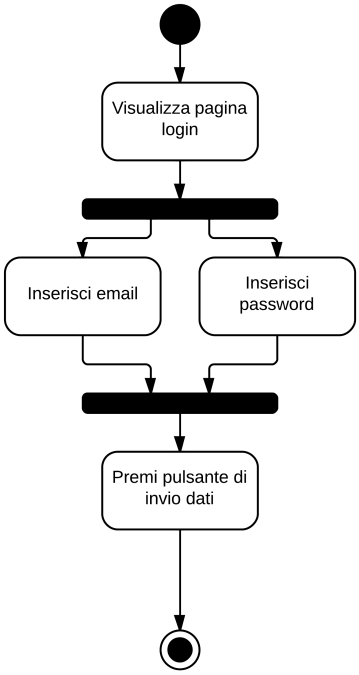

Figura 45: Diagramma di attività - Login dell'utente

L'utente, che precedentemente avrà effettuato la registrazione al sistema, accede all'interno dell'applicazione tramite una pagina di login. Al suo interno saranno presenti due campi di testo in cui l'utente dovrà inserire la propria email e la propria password. Una volta inserite dovrà premere il pulsante di login; il sistema $MaaP_G$ procederà dunque alla verifica delle credenziali e, se l'esito di tale verifica risulterà positivo, effettuerà il login dell'utente all'applicazione, reindirizzandolo alla $dashboard_G$.

6.1.6 Modifica profilo

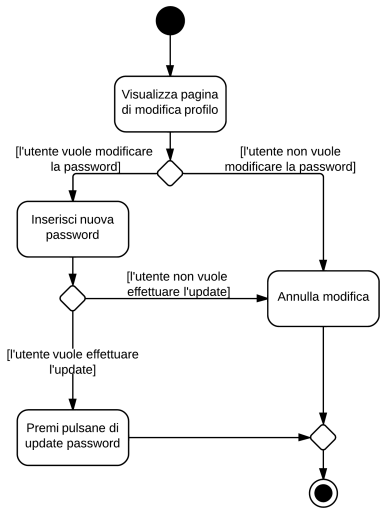

Figura 46: Diagramma di attività - Modifica profilo utente

L'utente autenticato accede all'interno della propria pagina profilo, dalla quale può decidere di modificare la propria password. Sarà dunque presente un campo di testo in cui l'utente inserirà la nuova password e un bottone tramite il quale invierà la richiesta di modifica; il sistema $MaaP_G$ procederà dunque alla modifica della password dell'utente. L'utente in ogni momento può decidere di annullare le modifiche e tornare alla pagina precedente.

6.1.7 Index-page Collection

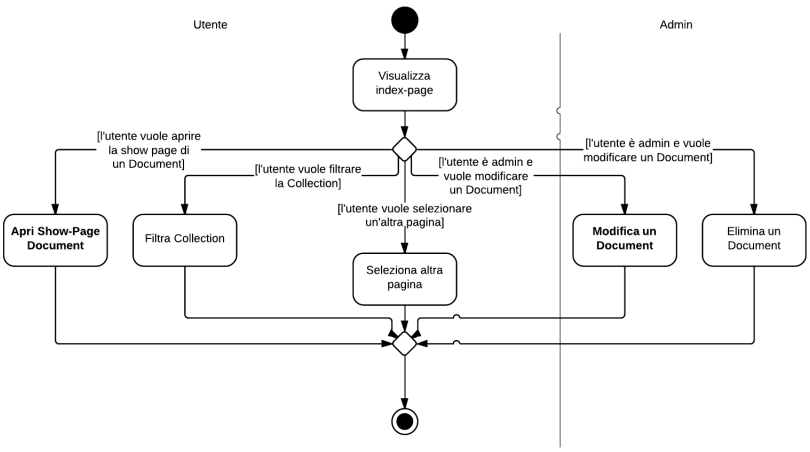

Figura 47: Diagramma di attività - Visualizzazione index-page della Collection selezionata

L'utente ha selezionato una $Collection_G$ dal menu e ora si trova all'interno di una pagina che visualizza una tabella contenente tutti i $Document_G$ della $Collection_G$ con alcuni attributi visualizzabili. A questo punto è in grado di fare diverse operazioni:

- Può aprire la relativa $show\text{-}page_G$ di un $Document_G$ selezionando il link che la apre;
- Può applicare un filtro ai $Document_G$ visualizzati in modo da visualizzare un sottoinsieme della tabella;
- Se la tabella risulta distribuita su più pagine può accedere alle pagine successive;

Se l'utente dispone dei privilegi di admin può inoltre:

- Modificare un $Document_G$ cliccando sul link *edit* visualizzato in ciascuna riga della tabella;
- Eliminare un $Document_G$ cliccando sul link *delete* visualizzato in ciascuna riga della tabella;

6.1.8 Show-page Document

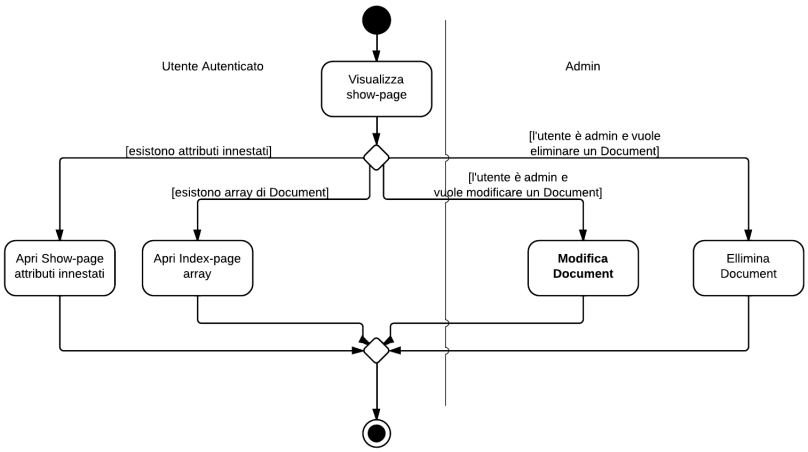

Figura 48: Diagramma di attività - Visualizzazione show-page del Document selezionato

L'utente ha selezionato un $Document_G$ dalla $index\text{-}page_G$ e ora si trova davanti una pagina di visualizzazione dettagliata del $Document_G$ selezionato. Sostanzialmente questa pagina conterrà una tabella contenente gli attributi visualizzabili del $Document_G$. Un utente all'interno di questa pagina può:

- Aprire la $show\text{-}page_G$ di un attributo innestato, se ne esiste uno;
- Aprire la $index\text{-}page_G$ di un array di $Document_G$, se ne esiste uno.

Se l'utente possiede i privilegi di admin può inoltre:

- Modificare gli attributi del $Document_G$;
- Eliminare il $Document_G$ corrente.

6.1.9 Apri pagina gestione utenti

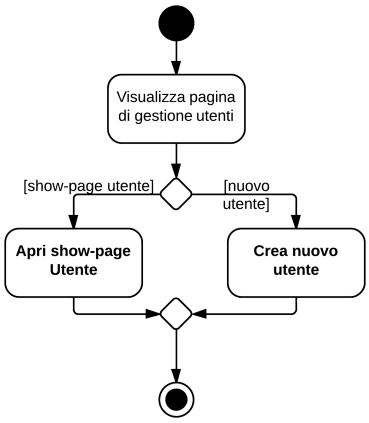

Figura 49: Diagramma di attività - Pagina di gestione degli utenti

Un admin dell'applicazione può accedere a una pagina in cui poter gestire gli utenti. Essa consiste fondamentalmente in una *index-page*$_G$ contenente la lista di tutti gli utenti presenti nel sistema. L'admin può da questa pagina selezionare un utente, e visualizzare quindi la sua relativa *show-page*$_G$, o crearne uno nuovo aprendo la pagina di creazione.

6.1.10 Apri show-page utente

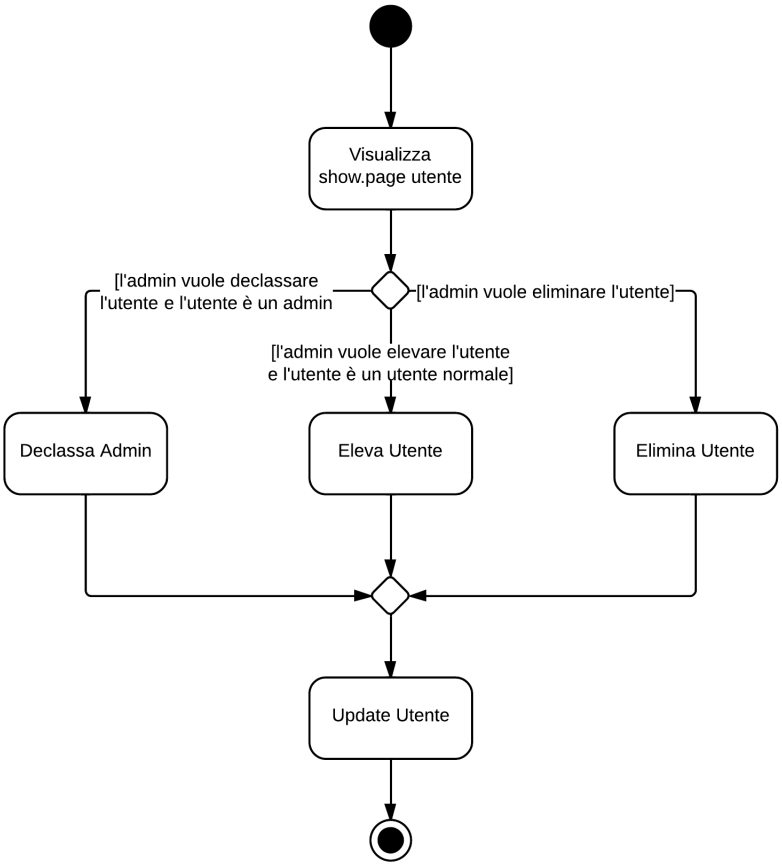

Figura 50: Diagramma di attività - Pagina di visualizzazione di un utente

In questa pagina l'admin visualizza la *show-page*$_G$ dell'utente selezionato e può compiere le seguenti operazioni:

- Se l'utente selezionato è un admin può declassarlo e portarlo a livello di utente normale. Naturalmente non può declassare se stesso e il *super-admin*, in modo da far sì che in qualsiasi momento sia presente almeno un admin nel sistema;
- Se l'utente non è un admin può elevarlo da utente normale a livello di admin;
- Eliminare l'utente selezionato dal sistema.

Il sistema *MaaP*$_G$ si occuperò di apportare tutte le modifiche effettuate dall'admin al *database*$_G$ delle credenziali.

6.1.11 Crea un nuovo utente

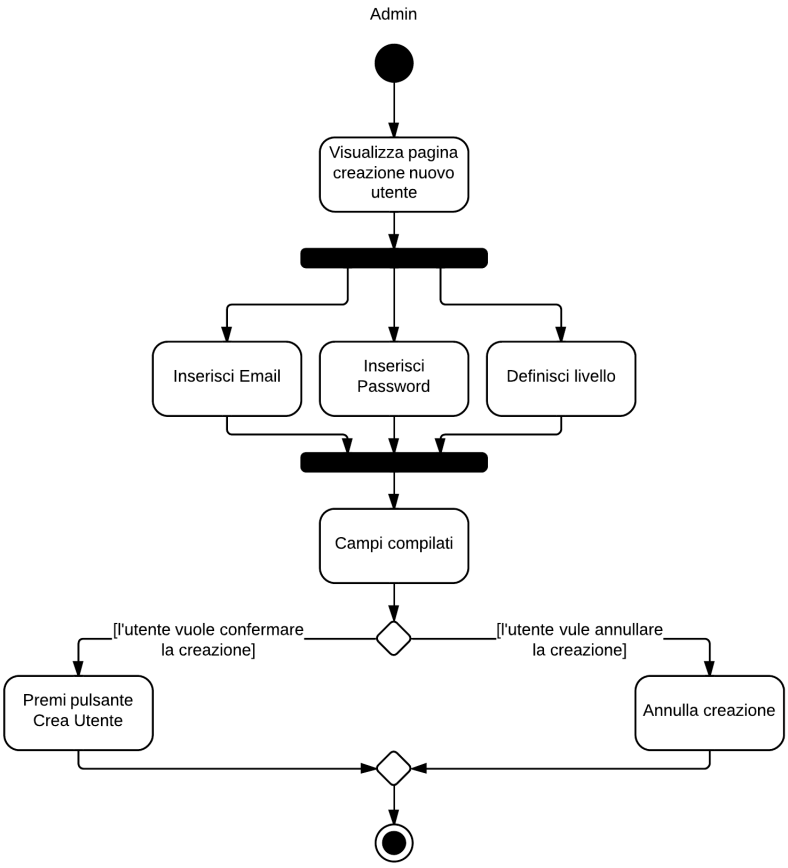

Figura 51: Diagramma di attività - Pagina di creazione di un nuovo utente

L'admin entra in un'apposita pagina di creazione di un nuovo utente e al suo interno può definire:

- L'indirizzo email del nuovo utente;
- La password del nuovo utente;
- Il livello del nuovo utente, che potrà essere o utente normale o admin;

Una volta completate le modifiche l'utente può decidere di confermare o annullare le modifiche, premendo i relativi pulsanti. Il sistema $MaaP_G$, nel caso in cui l'utente abbia deciso di confermare le modifiche, si occuperà di inserire nel $database_G$ delle credenziali il nuovo utente creato.

6.2 Framework MaaP

Viene di seguito descritta la procedura con la quale viene creata una nuova applicazione $MaaP_G$. Fondamentalmente lo sviluppatore si deve prendere carico di installare tutte le librerie necessarie al corretto funzionamento del $framework_G$. Una volta ottenute tutte le *dipendenze* potrà da linea di comando inizializzare un nuovo progetto $MaaP_G$. Vengono descritti in seguito i diagrammi di attività per la creazione di una nuova applicazione da parte del sistema.

6.2.1 Crea nuova applicazione

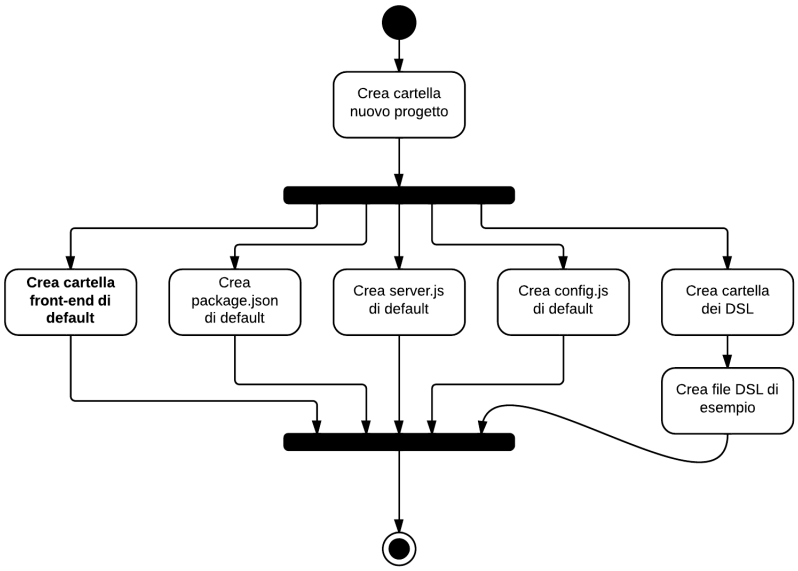

Figura 52: Diagramma di attività - Creazione scheletro nuova applicazione

Il $framework_G$ $MaaP_G$ si occupa di creare tutti i file necessari al corretto funzionamento dell'applicazione nella sua versione di *default*. In particolare si occupa di:

- Creare una cartella dove andranno tutti i file relativi al $front\text{-}end_G$;
- Creare il file `package.json` di default, nel quale verrà descritta l'applicazione specificando, ad esempio, il nome, le dipendenze, la versione;
- Il file `server.js` di default, il quale fornisce uno script da eseguire per avviare il server;
- Il file `config.js` di default nel quale viene configurata l'applicazione impostando ad esempio i database;
- Una cartella contenente tutti i file DSL_G che lo sviluppatore andrà a configurare. Di default questa cartella conterrà inizialmente un file DSL_G di esempio.

6.2.2 Creazione cartella front-end di default

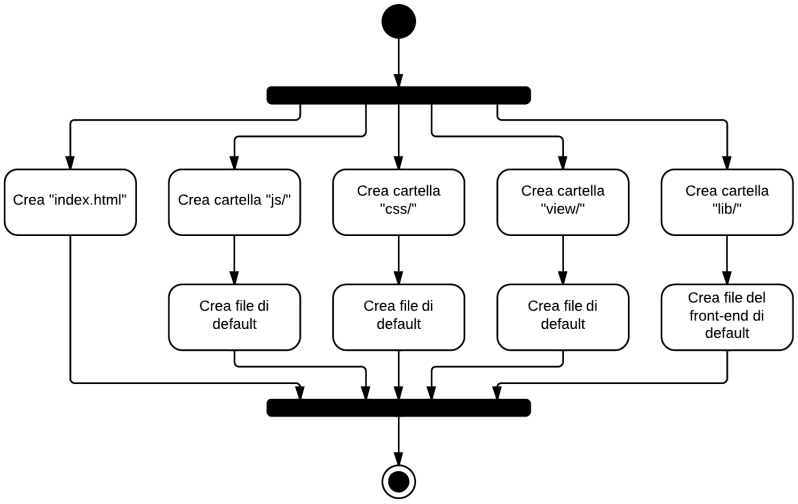

Figura 53: Diagramma di attività - Creazione scheletro nuova applicazione

All'interno di questa cartella sono presenti tutti i file di *default* per il corretto funzionamento del front-end. Oltre alla pagina `index.html` che fungerà da *home* dell'applicazione verrà creato:

- La cartella `js/` all'interno della quale saranno presenti tutti gli script *javascript*$_G$ necessari al corretto funzionamento dell'applicazione;
- La cartella `css/` la quale conterrà tutti i fogli di stile per la *presentazione* dell'applicazione;
- La cartella `view/` nella quale saranno presenti i file *html* che fungeranno da *template*;
- La cartella `lib/` in cui saranno presenti le librerie *javascript*$_G$ già predisposte per il *front-end*$_G$.

All'interno di ciascuna cartella saranno presenti inoltre i relativi file di *default*.

7 Stime di fattibilità e di bisogno di risorse

Durante la progettazione dell'architettura, oltre alle tecnologie e librerie consigliate e richieste dal proponente, ne sono state ricercate altre in modo da poter utilizzare funzionalità già pronte, garantendo una maggiore fattibilità nel ricoprire le esigenze progettuali.

Gli strumenti e le tecnologie integrate a quelle richieste dal capitolato sono :

- Passport
- Passport-local
- Passport-local-mongoose
- Nodemailer

Le tecnologie adottate sono attualmente molto diffuse: si trovano innumerevoli esempi, progetti, librerie, tutorial al riguardo. Da un lato alcune tecnologie non sono del tutto mature, visto che la gran parte dei progetti basati su di esse non raggiungono la versione "stabile". Dall'altro lato, però, il supporto della comunità è una grande risorsa: per ogni tipo di problema tecnico è molto facile trovare qualcuno che spieghi come risolverlo.

Questo viene in aiuto ai membri del gruppo, la cui maggioranza non aveva sufficienti conoscenze degli strumenti utilizzati per la realizzazione del progetto. Conoscenze che sono state approfondite grazie anche alla realizzazione di diversi prototipi interni, relativi all'applicazione front-end, alla realizzazione dell'interfaccia REST e alla realizzazione del parser del linguaggio DSL_G. Tali conoscenze continueranno ad essere sviluppate da ognuno dei componenti di SteakHolders.

Gli strumenti definiti durante la progettazione sono ritenuti adeguati per garantire una soddisfacibilità delle necessità progettuali, inoltre sono *open source*$_G$ e quindi di facile reperimento rendendo il bisogno di risorse non problematico.

8 Design pattern

Un *Design Pattern*$_G$ descrive problemi che si ripetono molteplici volte nel nostro ambiente. Oltre al problema descrive anche soluzioni eleganti ad esso e i risultati che si ottengono nell'applicarlo. È fondamentale per qualsiasi progettista conoscere a fondo i *Design Pattern*$_G$, in quanto facilita l'attività di progettazione, favorisce la riusabilità e dà benefici enormi in termini di manutenibilità. Fondamentalmente possiamo suddividere i *Design Pattern*$_G$ in quattro categorie:

- ***Design Pattern***$_G$ **architetturali**, che esprimono schemi di base per impostare l'organizzazione strutturale di un sistema software;
- ***Design Pattern***$_G$ **creazionali**, che forniscono un'astrazione del processo di istanziazione degli oggetti;
- ***Design Pattern***$_G$ **strutturali**, che si occupano delle modalità di composizione di classi e oggetti per formare strutture complesse;
- ***Design Pattern***$_G$ **comportamentali**, che si occupano di algoritmi e dell'assegnamento di responsabilità tra oggetti collaboranti.

Per un approfondimento e un richiamo teorico dei *Design Pattern*$_G$ utilizzati nel progetto *MaaP*$_G$ si rimanda all'Appendice A. In seguito verranno descritti i *Design Pattern*$_G$ implementati.

Nota: alcune immagini del presente capitolo non sono UML standard ma si sono semplici ritagli di diagrammi già presenti e visualizzati nei capitoli precedenti. Questo è stato fatto per fornire una visualizzazione mirata del contesto in cui viene utilizzato il design pattern.

8.1 Design Pattern Architetturali

8.1.1 MVC

- **Scopo**: Questo pattern è utilizzato per separare le responsabilità dell'applicazione a diversi componenti e permettere di fare una chiara divisione presentazione, struttura dei dati e operazioni su di essi.
- **Utilizzo**: Viene utilizzato dall'applicazione principalmente per delegare il ruolo di presentazione dei dati al *front-end*$_G$, lasciando al *back-end*$_G$ la gestione della logica dell'applicazione (autenticazione, corrispondenza tra *API*$_G$ e operazioni sui dati) e la logica di business. Nel *back-end*$_G$ è presente una chiara distinzione tra questi tre componenti:
  - Il package `Back-end::Lib::Model` è la componente **model** del pattern, e si occupa di gestire la logica interna dei dati degli utenti e delle *Collections*$_G$. Costruisce dei modelli di dati interfacciandosi con il database *MongoDB*$_G$;
  - Il package `Back-end::Lib::View` è la componente **view** del pattern, e si occupa di fornire un template per la composizione delle email da inviare per il recupero della password. Le richieste inoltrate al *back-end*$_G$, inoltre, ricevono come risposta dei dati in formato *JSON*$_G$. La rappresentazione dei dati che viene fornita in output può essere considerata implicitamente la componente **view** dell'intero package;

– Il package `Back-end::Lib::Controller` è la componente **controller** del pattern, e si occupa ricevere le richieste in ingresso, di interagire quindi con il *model* prelevando i dati e di restituirli al richiedente in formato $JSON_G$.

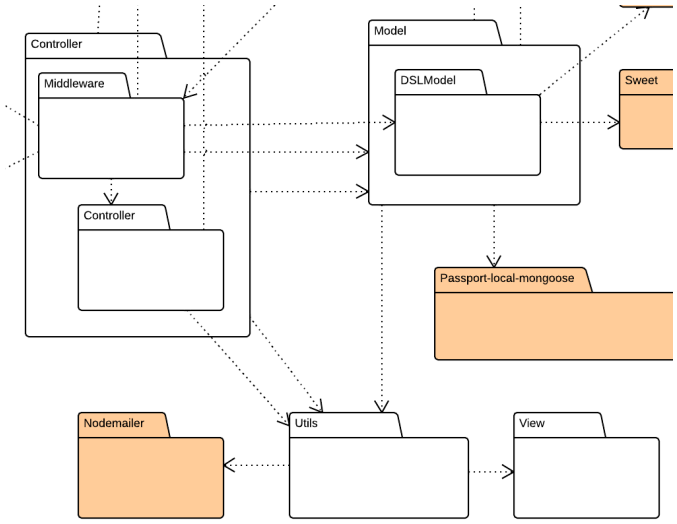

Figura 54: Contestualizzazione di MVC

8.1.2 MVW

- **Scopo**: È un *Design Pattern*$_G$ simile a *MVC*$_G$ che permette di avere una corrispondenza più diretta e automatica tra la *view* e il *model*. L'acronimo *MVW*$_G$ sta per *Model-View-Whatever*, dove *Whatever*, secondo i progettisti di *Angular.js*$_G$, indica *whatever works for you*.
- **Utilizzo**: È il *Design Pattern*$_G$ utilizzato dal *framework*$_G$ *Angular.js*$_G$, con il quale viene sviluppata la parte *front-end*$_G$ dell'applicazione *MaaP*$_G$. Un maggiore approfondimento è fornito in appendice.

8.1.3 Middleware

- **Scopo**: Si è scelto di utilizzare questo *Design Pattern*$_G$ per fornire un *intermediario* tra i vari componenti software dell'applicazione in modo da semplificare notevolmente la loro connessione e collaborazione. Questo pattern in generale è molto utile nello sviluppo e nella gestione di di sistemi distribuiti complessi, e in questo contesto il progetto *MaaP*$_G$ si colloca perfettamente.
- **Utilizzo**: Viene utilizzato dal *framework*$_G$ *Express*$_G$ attraverso il modulo *connect* per fornire una libreria di funzioni comuni. Definisce una serie di *livelli* (o funzioni) per gestire le varie richieste dell'applicazione e richiamare i rispettivi *handler*. Tutti i componenti del middleware sono collegati l'uno con l'altro e ricevono a turno una richiesta in ingresso, finché uno di questi non decide di partire con l'elaborazione per poi chiamare la funzione `next`. Come si può notare è molto legato a *Chain of*

Responsibility$_G$, che verrà descritto in seguito. Tutti i componenti di *Express*$_G$ vengono utilizzati con il metodo `use` di *Express*$_G$. Nella progettazione architetturale è utilizzato nel package `Back-End::Lib::Middleware`.

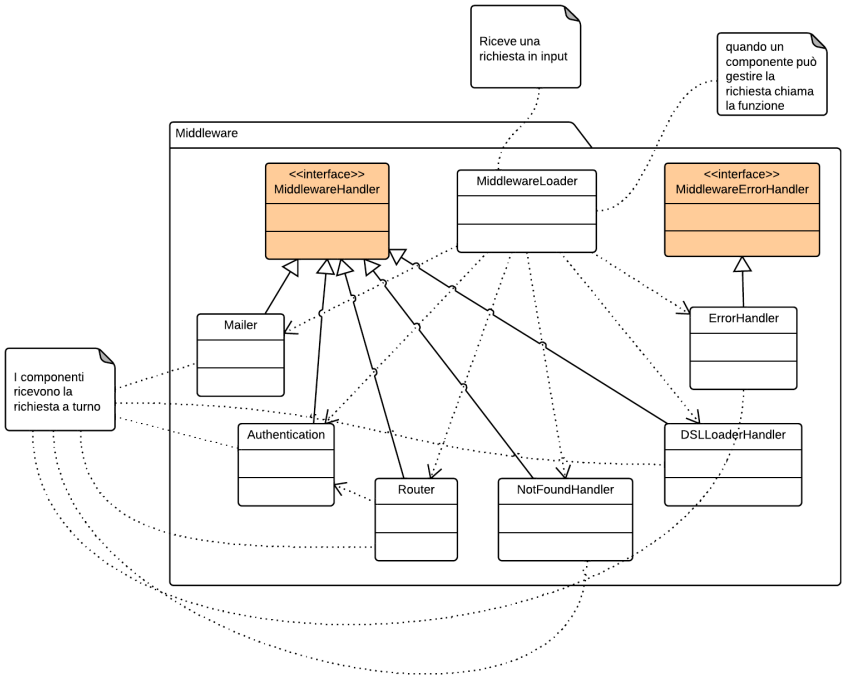

Figura 55: Contestualizzazione di Middleware

8.2 Design Pattern Creazionali

8.2.1 Registry

- **Scopo**: Viene utilizzato per ottenere oggetti a partire da altri oggetti che hanno un'associazione con esso. Questa ricerca viene effettuata tramite una *classe registro*, che conterrà una funzione di ricerca in base a una chiave.
- **Utilizzo**: Le diverse *Collection*$_G$ presenti nell'applicazione si differenziano per il loro nome. Utilizzando questo pattern, quando arriva una richiesta la classe che lo implementa sarà in grado di fornire il file *DSL*$_G$ corretto in quanto possiederà al suo interno un registro sul quale sarà possibile effettuare una ricerca. È implementato nella classe `Back-End::Lib::DSLModel`. Alla sua creazione verrà caricato il registro. Questa classe inoltre conterrà un metodo di ricerca e un metodo di caricamento del file DSL.

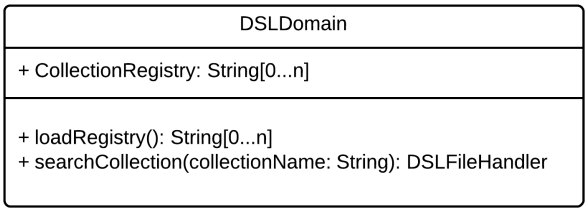

Figura 56: Contestualizzazione di Registry

8.2.2 Factory method

- **Scopo**: Nel contesto di *Node.js*$_G$ questo pattern viene usato creare una classe e restituire una sua istanza attraverso una *funzione factory* che verrà esportata dal modulo. In questo modo si potrà costruire e ottenere qualsiasi classe definita in un modulo.
- **Utilizzo**: Alle basi del *routing*, che utilizza la rappresentazione *REST*$_G$, vi sarà un controller associato per l'esecuzione delle diverse funzioni a seconda dell'URL indicato. In base a quest'ultimo dev'essere istanziata l'apposita classe che si occuperà di effettuare le sue funzioni. Per creare un oggetto di quella classe ci si avvale di una classe *factory*, la quale si occuperà di invocare la costruzione dell'oggetto. Nell'architettura del progetto il pattern è implementato nella classe `Back-End::Lib::Controller::ControllerFactory`.

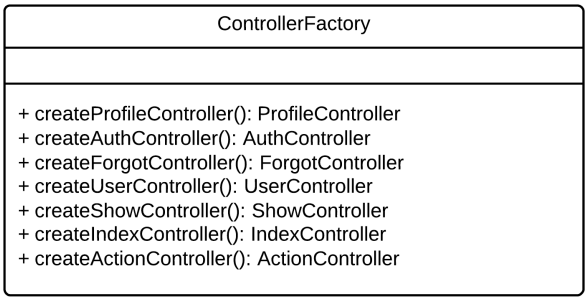

Figura 57: Contestualizzazione di Factory Method

8.2.3 Singleton

- **Scopo**: Viene utilizzato per le classi che devono avere un'unica istanza durante l'esecuzione dell'applicazione;
- **Utilizzo**: Ogni modulo di *Node.js*$_G$ è nativamente un singleton, perché viene caricato al primo `require` e poi tutti i successivi riferiscono allo stesso.

8.3 Design Pattern Strutturali

8.3.1 Facade

- **Scopo**: Viene utilizzato per rendere visibili solamente alcune cose agli altri oggetti ed avere un unico punto di accesso semplificato a un sottosistema fornendo un'interfaccia di alto livello e minimizzando dunque le comunicazioni e le dipendenze.
- **Utilizzo**: Viene utilizzato all'interno della classe `Back-End::Lib::Middleware::MiddlewareLoader`, la quale utilizza *facade* per nascon: dere l'esistenza di tutti i *middleware*$_G$ alla `ServerApp`. In questo modo le richieste vengono delegate agli oggetti appropriati senza che la classe cliente conosca le classi del sottosistema. Sarà il *Facade* che si occuperà di trasferire la comunicazione all'oggetto appropriato.

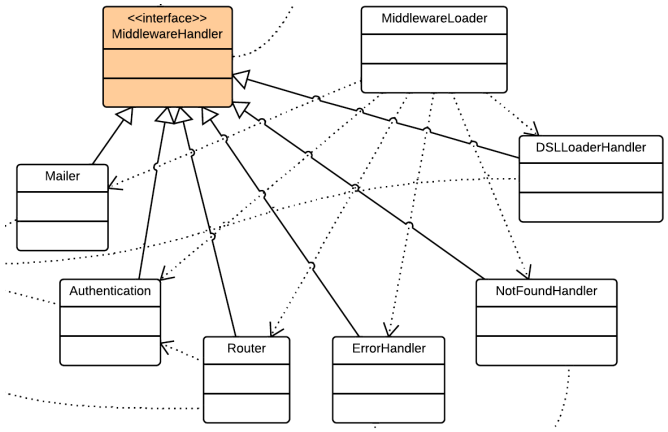

Figura 58: Contestualizzazione di Facade in `Back-End::Lib::Middleware::MiddlewareLoader`

8.4 Design Pattern Comportamentali

8.4.1 Chain of Responsibility

- **Scopo**: Viene utilizzato per far sì che un oggetto a cui viene effettuata una richiesta possa esaudire le richieste di più oggetti. In questo modo si evita l'accoppiamento fra il mittente di una richiesta e il destinatario. Tutti gli oggetti destinatari della richiesta sono *concatenati* tra di loro. Ogni nodo della catena se può esaudire la richiesta la effettua, altrimenti delega l'onere al nodo successivo. La catena viene attraversata finché un nodo non può eseguire l'ordine del mittente.
- **Utilizzo**: *Express*$_G$ usa *chain of Responsibility* per la gestione dei *middleware*$_G$ e del *routing*$_G$. Come già accennato è particolarmente legato al pattern *Middleware*. Viene utilizzato nella nostra architettura all'interno del package `Back-End::Lib::Middleware`. La classe `Back-End::Lib::Middleware::MiddlewareHandler` gestisce la richiesta scorrendo tutta la lista delle sottoclassi e richiamando il metodo `next` finché una di queste non può soddisfarla.

Nel gergo del framework Express i middleware corrispondono agli oggetti ConcreteHandler del design pattern. Sebbene il comportamento e lo scopo sia quasi identico, l'implementazione di Express presenta alcune differenze:

- I middleware di Express possono essere classi con un metodo `handle` o semplici funzioni, in pieno accordo con lo stile funzionale utilizzato dalla grande maggioranza delle librerie e delle applicazioni scritte in Node.js. Nel progetto è stata utilizzata principalmente la seconda versione.
- Il Design Pattern prevede che l'oggetto `Handler` abbia un riferimento `successor` all'`Handler` successivo. Express invece passa al metodo di esecuzione del middleware una callback. Il middleware, eseguendo la callback, passa nuovamente il controllo all'oggetto del server di Express il quale passerà il controllo al successivo middleware.
- Express divide i middleware in due gruppi: i middleware standard e i middleware di gestione degli errori, si distinguono per il numero di parametri che possono gestire (3 e 4, rispettivamente). Ogni middleware può decidere se passare il controllo al prossimo middleware standard o se passare il controllo al prossimo middleware di gestione degli errori (passandogli anche l'errore da gestire). Questa funzionalità serve per permettere la gestione di errori senza utilizzare i costrutti `try catch`, tipici dei linguaggi imperativi ma inefficaci quando si utilizza lo stile di programmazione asincrono.
- Ogni middleware di Express deve essere invocato con i seguenti parametri: l'eventuale errore da gestire (se è un middleware di gestione degli errori), l'oggetto della richiesta, l'oggetto della risposta, la callback da utilizzare per passare il controllo al successivo middleware. L'ordine è importante.

Per ulteriori dettagli si consiglia http://stephensugden.com/middleware_guide/ nella versione del 3 settembre 2014 (https://web.archive.org/web/20130901132345/http://stephensugden.com/middleware_guide/).

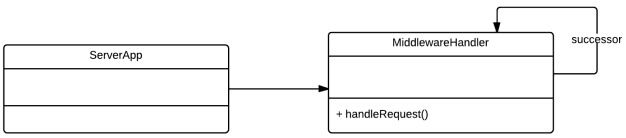

Figura 59: Contestualizzazione di Chain of Responsibility

8.4.2 Strategy

- **Scopo**: Questo pattern serve per definire una famiglia di algoritmi e renderli intercambiabili, in modo che essi possano variare indipendentemente dal client che ne fa utilizzo. In un progetto software che guarda al futuro e che verrà manutenuto è fondamentale poter effettuare modifiche alle procedure in modo non intrusivo.
- **Utilizzo**: Viene utilizzato all'interno della classe `Back-End::Lib::DSLModel::DSLInterpreterStrategy` in modo da permettere in futuro cambiamenti all'algoritmo di interpretazione del DSL_G senza dover intervenire sulla classe che ne fa uso, ovvero `Back-End::Lib::DSLModel::DSLDomain`.

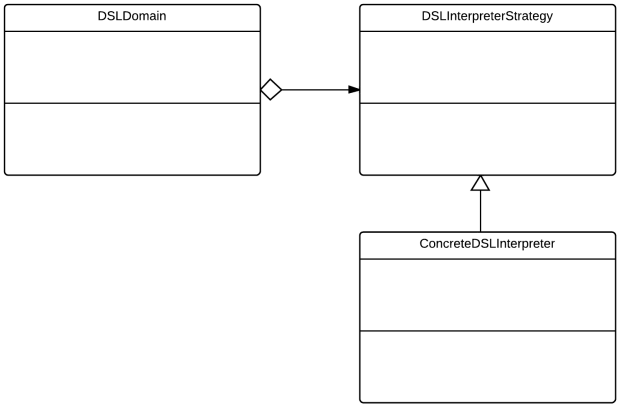

Figura 60: Contestualizzazione di Strategy

8.4.3 Command

- **Scopo**: Viene usato per parametrizzare gli oggetti rispetto a un'azione da compiere.
- **Utilizzo**: Viene utilizzato nel *package*$_G$ `Back-End::DSLModel` per definire le azioni personalizzate da intraprendere sulle *Collection*$_G$ o sui *Document*$_G$. In particolare `Back-End::DSLModel::DocumentAction` e `Back-End::DSLModel::CollectionAction` rappresentano ciascuno il componente *Command* del pattern (applicato in due contesti diversi). La componente *ConcreteCommand* del pattern consiste in una delle due precedenti classi estese dinamicamente, ridefinendo un metodo.

9 Tracciamento

9.1 Tracciamento componenti - requisiti

Componente	Requisiti
Front-end	
Front-end::Controllers	RA10 1 RA10 1.1 RA10 1.2 RA1O 1.3.2 RA10 2.3 RA1D 3 RA1O 4 RA1O 4.1 RA1O 4.1.1 RA1O 4.1.2 RA1O 4.1.3 RA1O 5 RA1O 5.1 RA1O 5.3 RA1O 6 RA1O 6.1 RA1O 6.1.1 RA1O 6.2 RA1O 6.2.1 RA1O 6.2.2 RA1O 6.2.3 RA1O 6.2.4 RA1O 6.2.5 RA1D 11 RA1D 12 RA1D 13 RA1D 13.1
Front-end::Model	RA10 1.3.1 RA1O 6.1.3

Front-end::Services	RA10 2 RA10 2.1 RA1O 4 RA10 2.3 RA1O 5 RA1O 5.1 RA1O 5.3 RA1O 6 RA1O 6.1 RA1O 6.1.1 RA1D 13
Back-end	
Back-end::DeveloperProject	RF1O 8.1.1 RF1O 8.1.2 RF1O 8.1.3 RF1O 8.1.4 RF1F 8.3 RF1O 14 RF1O 14.1 RF1O 14.2
Back-end::Lib	
Back-end::Lib::Controller::Middleware	RF1O 8.2 RA1D 12
Back-end::Lib::Controller::Service	RA10 2.3 RA1O 4.1 RA1O 4.1.1 RA1O 4.1.2 RA1O 4.1.3 RA1O 5 RA1O 5.1 RA1O 5.3 RA1O 6.1.2 RA1D 13.1

Back-end::Lib::Model::DSLModel	RA1O 4.1 RA1O 4.1.1 RA1O 4.1.2 RA1O 4.1.3 RA1O 5 RA1O 5.1 RA1O 5.3 RA1O 6 RA1O 6.1.2 RF1O 7 RF1O 8 RF1O 9 RF1O 9.1 RF1O 9.2 RF1O 9.3 RF1O 9.4 RA1O 18
Back-end::Lib::View	
Back-end::Lib::Controller::Middleware	RA10 1.3 RA10 2.2 RA1O 6.1.3 RF1O 8 RF1O 8.1

9.2 Tracciamento requisiti - componenti

Requisito	Componenti
RA10 1	Front-end::Controllers
RA10 1.1	Front-end::Controllers
RA10 1.2	Front-end::Controllers
RA10 1.3	Back-end::Lib::Controller::Middleware
RA10 2	Front-end::Services
RA10 2.1	Front-end::Services
RA10 2.2	Back-end::Lib::Controller::Middleware
RA10 2.3	Front-end::Controllers Front-end::Services Back-end::Lib::Controller::Service
RA1D 3	Front-end::Controllers
RA1O 4	Front-end::Controllers Front-end::Services
RA1O 4.1	Front-end::Controllers Back-end::Lib::Model::DSLModel
RA1O 4.1.1 RA1O 4.1.2	Front-end::Controllers Back-end::Lib::Model::DSLModel Front-end::Controllers Back-end::Lib::Model::DSLModel
RA1O 4.1.3	Front-end::Controllers Back-end::Lib::Model::DSLModel
RA1O 5	Front-end::Controllers Front-end::Services Back-end::Lib::Controller::Service Back-end::Lib::Model::DSLModel
RA1O 5.1	Front-end::Controllers Front-end::Services Back-end::Lib::Controller::Service Back-end::Lib::Model::DSLModel

RA1O 5.3	Front-end::Controllers Front-end::Services Back-end::Lib::Controller::Service Back-end::Lib::Model::DSLModel
RA1O 6	Front-end::Controllers Front-end::Services Back-end::Lib::Controller::Service Back-end::Lib::Model::DSLModel
RA1O 6.1	Front-end::Controllers Front-end::Services
RA1O 6.1.1	Front-end::Controllers Front-end::Services
RA1O 6.1.2	Back-end::Lib::Controller::Service Back-end::Lib::Model::DSLModel
RA1O 6.1.3	Back-end::Lib::Controller::Middleware
RA1O 6.2	Front-end::Controllers
RF1O 7	Back-end::Lib::Model::DSLModel
RF1O 8	Back-end::Lib::Model::DSLModel Back-end::Lib::Controller::Middleware
RF1O 8 .1	Back-end::Lib::Controller::Middleware
RF1O 8 .1.1	Back-end::DeveloperProject
RF1O 8 .1.2	Back-end::DeveloperProject
RF1O 8 .1.3	Back-end::DeveloperProject
RF1O 8 .1.4	Back-end::DeveloperProject
RF1O 8.2	Back-end::Utils::UserModel
RF1F 8.3	Back-end::DeveloperProject
RF1O 9	Back-end::Lib::Model::DSLModel
RF1O 9.1	Back-end::Lib::Model::DSLModel
RF1O 9.2	Back-end::Lib::Model::DSLModel
RF1O 9.3	Back-end::Lib::Model::DSLModel

RF1O 9.4	Back-end::Lib::Model::DSLModel
RA1D 11	Front-end::Controllers
RA1D 12	Front-end::Controllers Back-end::Model::UserModel
RA1D 13	Front-end::Controllers Front-end::Services
RA1D 14	Back-end::DeveloperProject
RA1D 14.1	Back-end::DeveloperProject
RA1D 14.2	Back-end::DeveloperProject
RA1O 18	Back-end::Lib::Model::DSLModel

A Descrizione Design Pattern

A.1 Design Pattern Architetturali

A.1.1 MVC

Model-View-Controller (MVC) è un pattern per l'implementazione di interfacce utente. Esso divide un'applicazione software in tre parti interconnesse, in modo da separare nettamente la rappresentazione interna dei dati dal modo in cui essa viene presentata all'utente. Il componente centrale, il modello, consiste di dati business, regole, logica e funzioni. Una *view* può essere qualsiasi output dell'informazione, come ad esempio un testo o un diagramma. Si possono avere molteplici *view* della stessa informazione. La terza parte, il *controller*, si occupa di accettare degli input e di convertirli in *comandi* per il *model* o per la *view*.

Oltre a dividere l'applicazione in queste tre componenti, MVC_G si occupa anche di definire le interazioni tra esse:

- Un *controller* può inviare comandi al *model* per aggiornare il suo stato. Può inoltre inviare comandi alla sua *view* associata, in modo da cambiarne la presentazione;
- Un *model* quando cambia il suo stato interno notifica le sue *view* e i suoi *controller* associati. Questo permette alle *view* di cambiare la loro presentazione e ai *controller* di cambiare il loro insieme di comandi disponibili.
- Una *view* viene aggiornata dal *controller* sui dati che necessita per generare un output per l'utente.

Sebbene inizialmente sviluppata per applicazioni desktop, MVC_G è stato usato moltissimo come architettura per le Web application in tutti i principali linguaggi di programmazione. Moltissimi $framework_G$ commerciali e non sono stati progettati utilizzando questo pattern.

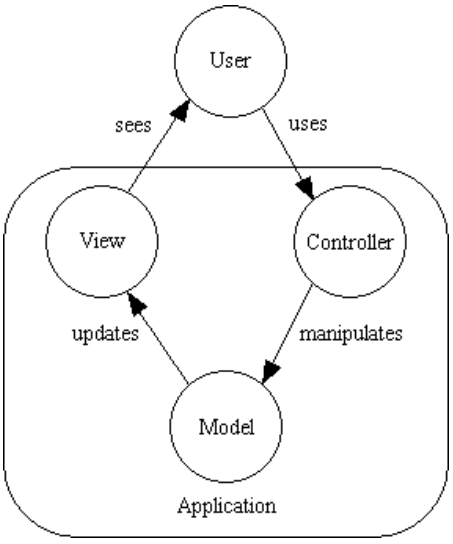

Figura 61: Struttura logica di Model-View-Controller

A.1.2 Middleware

Il *Middleware*$_G$ è uno strato software che si interpone tra l'applicazione software e il sistema operativo per semplificarne le comunicazioni e la gestione di input/output. Viene solitamente utilizzato in applicazioni distribuite e facilita l'interoperabilità, fornendo servizi che permettono la comunicazione tra applicazioni di sistemi operativi diversi. La distinzione tra lo strato software del sistema operativo è, per alcune entità, arbitraria; può infatti accedere che il *Middleware*$_G$ fornisca dei servizi abitualmente attribuibili a un sistema operativo. I primi utilizzi di *Middleware*$_G$ risalgono agli anni '80, come soluzione ai problemi di comunicazione tra applicazioni nuove e meno recenti. I servizi *Middleware*$_G$ forniscono un set di interfacce che permetto a un'applicazione di:

- Localizzare facilmente applicazioni o servizi in una rete;
- Filtrare dati per renderli *user-friendly* oppure anonimizzarli per renderli pubblicabili, proteggendone la privacy;
- Essere indipendente dai servizi di rete;
- Essere affidabile e sempre disponibile;
- Aggiungere attributi complementari.

Si tratta quindi di funzionalità leggermente più specializzate da quelle normalmente offerte da un sistema operativo. L'avvento del web ha avuto una forte ripercussione sulla diffusione dei software di *Middleware*$_G$. Essi hanno infatti permesso l'accesso sicuro da remoto a database locali. I tipi di *Middleware*$_G$ sono:

- **Message-Oriented Middleware** (*MOM*$_G$): sono *Middleware*$_G$ dove le notifiche degli eventi vengono spedite come messaggi tra sistemi o componenti. I messaggi inviati al client vengono memorizzati fintanto che non vengono gestiti, nel frattempo il client può svolgere altro lavoro;
- **Enterprise messaging system**: è un tipo di *Middleware*$_G$ che facilita il passaggio di messaggi tra sistemi diversi o componenti in formato standard, spesso utilizzando servizi web o *XML*$_G$;
- **Message broker**: è parte dell *entreprise messaging system*. Accoda, duplica, traduce e spedisce messaggi a sistemi o componenti diverse;
- **Enterprise Service Bus**: è definito come qualche tipo di *Middleware*$_G$ integrato che supporta sia *MOM*$_G$ che dei servizi web;
- **Intelligent Middleware**: gestisce il processamento in tempo reale di grandi volumi di segnali che trasforma in informazioni di business. Particolarmente adatto per architetture scalabili e distribuite;
- **Content-Centric Middleware**: questo tipo di *Middleware*$_G$ fornisce una semplice astrazione con la quale le applicazioni possono inoltrare richieste per contenuti univocamente identificati, senza occuparsi su come e dove vanno ottenuti.

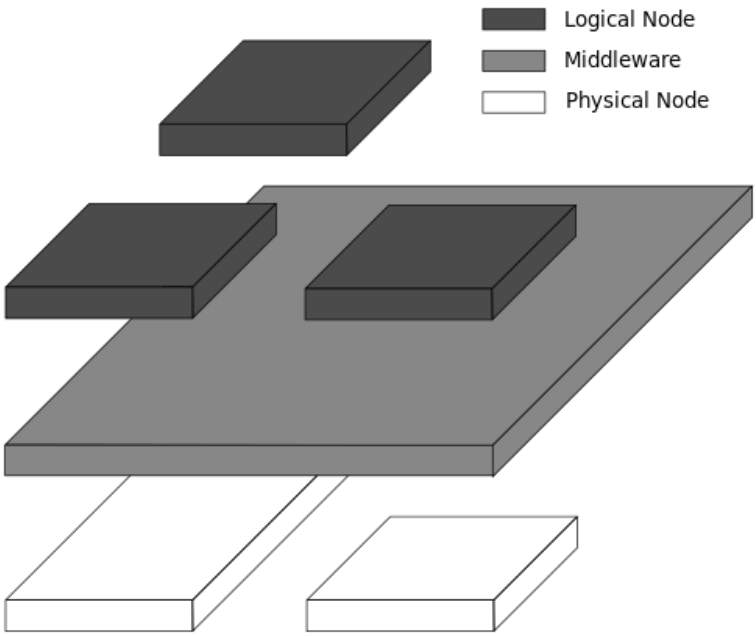

Figura 62: Struttura logica di Middleware

A.2 Design Pattern Creazionali

A.3 Singleton

Il *Singleton*$_G$ è un design pattern creazionale che permette di avere un'unica istanza di una classe con un unico punto di accesso noto. Tale condizione è tipica di alcuni contesti e trova risvolti pratici in svariate applicazioni. Per permettere l'implementazione di questo pattern è sufficiente che la classe stessa si occupi di tracciare la propria istanziazione e bloccarla qualora sia già avvenuta almeno una volta. Il *Singleton*$_G$ dovrebbe essere estensibile usando il *subclassing*. Il client può utilizzarne l'estensione senza quindi modificarne il codice.

L'utilizzo di questo pattern comporta una serie di conseguenze:

- Accesso controllato alla singola istanza: poiché la classe *Singleton*$_G$ incapsula la sua unica istanza, è in grado di controllare quando e come i client vi accedono;
- Namespace pulito: l'utilizzo di questo pattern risulta migliore rispetto all'uso di variabili globali poiché scongiura l'inquinamento del namespace globale;
- Permette raffinamenti di operazioni e rappresentazioni: il *Singleton*$_G$ dovrebbe venire sempre esteso prima dell'utilizzo, che in termini pratici si traduce in un operazione banale. Questo può avvenire anche a runtime;
- Eventualmente permette un numero variabile di istanze: questo pattern permette, se necessario, di avere istanze multiple mantenendo però il controllo sul numero;

- Flessibilità: un modo per avere una funzionalità riconducibile al *Singleton*$_G$ è quello di utilizzare le operazioni sulle classi, come per esempio la keyword `static` del C++, ma in questo modo è più difficile controllarne il design e permetterne più istanze. Inoltre nel linguaggio succitato le funzioni statiche non sono mai virtuali, rendendone impossibile l'utilizzo polimorfo alle sottoclassi che le ridefiniscono.

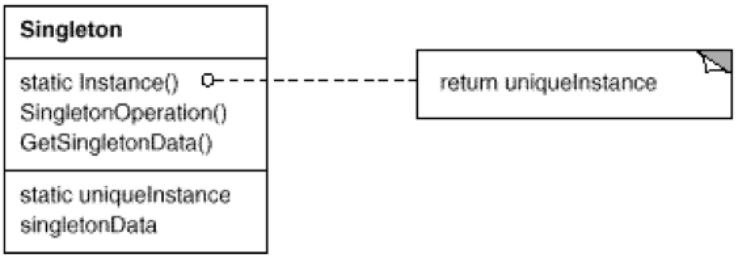

Figura 63: Struttura logica di Singleton

A.3.1 Registry

Il *Registry*$_G$ è simile ad un oggetto globale che gli altri oggetti usano per accedere a servizi e oggetti comuni. Quando si vuole recuperare un oggetto capita spesso di accedervi tramite un altro oggetto legato da un qualche tipo di associazione, ma in alcuni casi non è possibile conoscere a priori l'oggetto da cui partire, così vi è la necessità di avere un metodo di look-up accedibile tramite il *Registry*$_G$. Le interfacce del *Registry*$_G$ possono essere metodi statici.

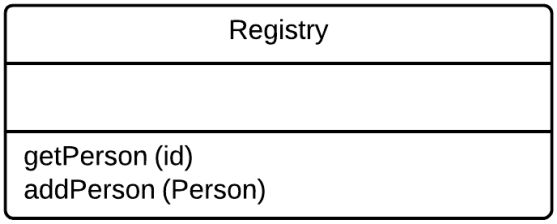

Figura 64: Struttura logica di Registry

A.3.2 Factory method

Questo pattern definisce un'interfaccia per la creazione di un oggetto, lasciando alle sottoclassi la decisione sulla classe che deve essere istanziata. Consente inoltre di deferire l'istanziazione di una classe alle sottoclassi. Tra i suoi utilizzi ci sono i seguenti casi:

- Quando una classe non è in grado di sapere in anticipo le classi degli oggetti che deve creare;

- Quando una classe vuole che le sue sottoclassi scelgano gli oggetti da creare;
- Quando le classi delegano la responsabilità a una o più classi di supporto e si vuole localizzare in un punto ben preciso la conoscenza di quale o quali classi di supporto vengano delegate.

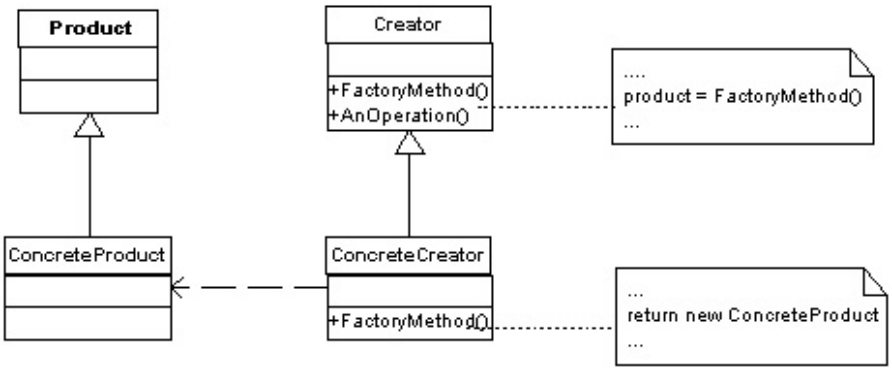

Figura 65: Struttura logica di Factory Method

A.4 Design Pattern Strutturali

A.4.1 Facade

Questo pattern fornisce un'interfaccia unificata per un insieme di interfacce presenti in un sottosistema. $Facade_G$ definisce un'interfaccia di alto livello che rende il sottosistema più semplice da utilizzare. Suddividere un sistema in sottosistemi aiuta a ridurne la complessità. Può essere utilizzato nei seguenti casi:

- Quando si vuole fornire un'interfaccia semplice a un sottosistema complesso. La complessità dei sottosistemi tende ad aumentare con la loro evoluzione. Molti pattern, quando applicati, portano a un aumento nel numero di classi piccole. Ciò rende il sottosistema maggiormente riusabile e semplice da personalizzare, ma di utilizzo più difficile per i client che non richiedono alcuna personalizzazione. Un *facade* può fornire una vista semplice di base su un sottosistema che si rivela essere sufficiente per la maggior parte dei client. Soltanto i client che richiedono una personalizzazione maggiore dovranno guardare dietro la facciata;
- Nei casi in cui sono molte le dipendenze fra i client e le classi che implementano un'astrazione. Introducendo un *facade* si disaccoppia il sottosistema dai client e dagli altri sottosistemi, promuovendo quindi la portabilità e l'indipendenza di sottosistemi;
- Quando si vogliono organizzare i sottosistemi in una struttura a livelli. Un *facade* può essere utilizzato per definire un punto di ingresso ad ogni livello. Nel caso in cui i sottosistemi non siano indipendenti e le dipendenze esistenti possano essere semplificate facendo comunicare tra loro i sottosistemi soltanto attraverso i rispettivi oggetti *facade*.

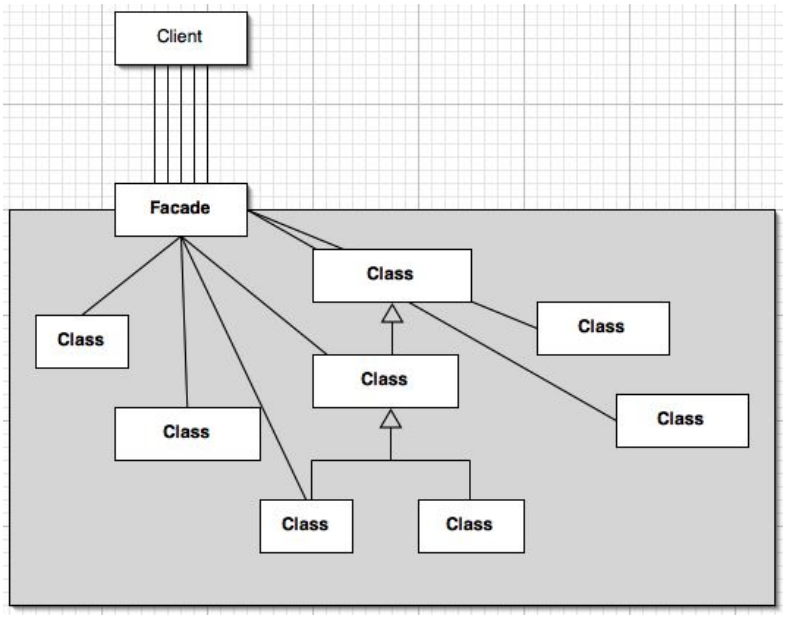

Figura 66: Struttura logica di Facade

A.5 Design Pattern Comportamentali

A.5.1 Chain of Responsibility

Il *Chain of Responsibility*$_G$ è un pattern comportamentale che permette di separare i *sender* dai *receiver* delle richieste. La richiesta attraversa una catena di oggetti per essere intercettata solo quando raggiunge il proprio gestore. Viene utilizzato quando non è possibile determinare staticamente il *receiver* oppure l'insieme di oggetti gestori cambia dinamicamente a runtime. Le richieste vengono dette *implicite* poiché il *sender* non ha alcuna conoscenza sull'identità del ricevente. Per permettere alla richiesta di attraversare la catena e per rimanere *implicita*, ogni *receiver* condivide un interfaccia comune per gestire le richieste ed accedere al proprio successore. La gerarchia che vorrà inviare richieste dovrà avere una superclasse che dichiara un metodo *handler* generico. La specializzazione di tale metodo avviene tramite *overriding* nelle sottoclassi opportune, come illustrato in figura 67.

L'utilizzo di questo pattern comporta una serie di conseguenze:

- Ridotto accoppiamento: gli oggetti non sono a conoscenza di chi gestirà la richiesta ma sanno solo che verrà gestita in modo appropriato. Inoltre non bisognerà manutenere i riferimenti a tutti i possibili riceventi;
- Aggiunge flessibilità nell'assegnamento delle responsabilità degli oggetti: è possible distribuire le responsabilità tra gli oggetti a runtime modificandone la gerarchia. Staticamente è possibile usare il *subclassing* per specializzare i gestori;
- Non c'è garanzia che la *request* venga gestita, questo può avvenire quando la catena non è stata costruita in modo rigoroso.

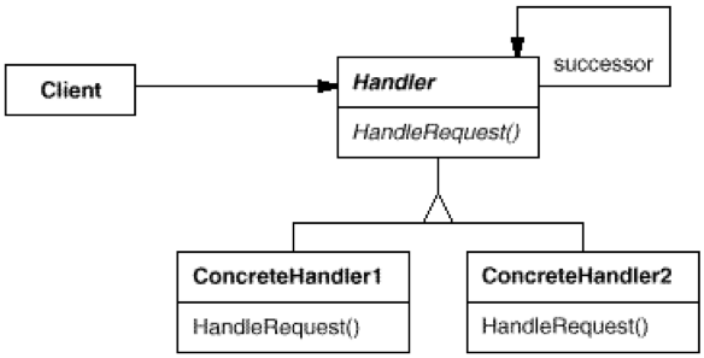

Figura 67: Struttura del Chain of Responsibility

A.5.2 Strategy

Strategy ha come scopo quello di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Permette agli algoritmi di variare indipendentemente dal client che ne fa uso. È opportuno usare il pattern *strategy* nei seguenti casi:

- Molte classi correlate differiscono fra loro solo per il comportamento. *Strategy* fornisce un modo per configurare una classe con un comportamento scelto fra tanti;
- Sono necessarie più varianti di un algoritmo. Per esempio, è possibile definire più algoritmi con bilanciamenti diversi fra occupazione in memoria, velocità di esecuzione, ecc. Possiamo usare il pattern *Strategy* quando queste varianti sono implementate sotto forma di gerarchia di classi di algoritmi;
- Un algoritmo usa una struttura dati che non dovrebbe essere resa nota ai client. Il pattern *strategy* può essere usato per evitare di esporre strutture dati complesse e specifiche dell'algoritmo;
- Una classe definisce molti comportamenti che compaiono all'interno di scelte condizionali multiple. Al posto di molte scelte condizionali si suggerisce di spostare i blocchi di codice correlati in una classe `Strategy` dedicata.

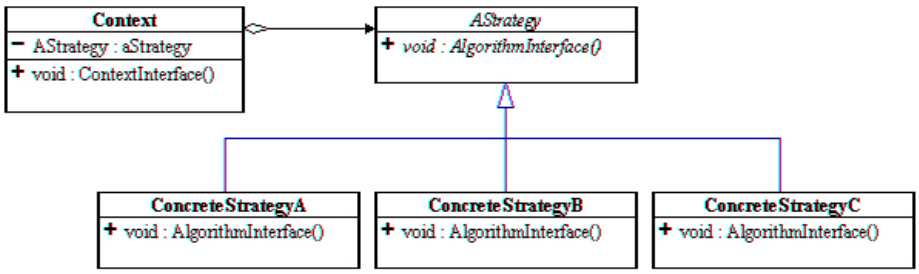

Figura 68: Struttura logica di Strategy

A.5.3 Dependency Injection

Il *Dependency Injection*$_G$ è un *Design Pattern*$_G$ che permette la separazione del comportamento degli oggetti dalla loro dipendenze. Invece di istanziare le classi in modo diretto ogni componente riceve i riferimenti agli altri componenti necessari come parametri nel costruttore. Un utilizzo comune è quello con i *plugin* che vengono caricati dinamicamente. Gli elementi coinvolti sono:

- Un dipendente consumatore;
- Una dichiarazione delle dipendenze tra la componenti, definita come contratto di un interfaccia;
- Un injector che crea istanze di classi che implementano una data dipendenza su richiesta.

Il *dependent object* dichiara da quali componenti dipende. L'*injector* decide quali classi soddisfano suoi requisiti e in caso affermativo gliele fornisce. Questa operazione può avvenire anche a runtime. Questo è un chiaro vantaggio poiché possono essere create dinamicamente diverse implementazioni di un componente software da passare allo stesso test. In questo modo il test può testare componenti diverse senza sapere che le loro implementazioni sono diverse. Lo scopo principale di questo pattern è quello di permettere una selezione a runtime su più implementazioni di una interfaccia dipendente. È particolarmente utile per fornire delle implementazioni di *stub*$_G$ per componenti complesse, ma anche per gestire i plugin e per inizializzare servizi software. I test di unità comportano delle problematiche, poiché spesso richiedono la presenza di una parte di infrastruttura non ancora implementata. Il *Dependency Injection*$_G$ semplifica il processo di testing per un istanza isolata. Poiché le componenti dichiarano le proprie dipendenze, un test può automaticamente istanziare le componenti necessarie.

L'utilizzo di questo pattern comporta una serie di conseguenze:

- Vi è una riduzione di *Boilerplate code*$_G$ poiché il lavoro di set up delle dipendenze viene gestito da un componente dedicato;
- Offre una certa flessibilità di configurazione perché diverse implementazione di un servizio posso essere usate senza essere ricompilate;

- Facilita la scrittura di codice testabile;
- Le dipendenze dichiarate sono *black box*$_G$, questo rende più difficile trovare gli errori al loro interno;
- Le dipendenze non completamente implementate o errate generano errori a runtime e non a tempo di compilazione;
- Rende il codice più difficile da manutenere;
- L'*injection* a runtime di dipendenze va ad inficiare le prestazioni;
- I benefici sono difficilmente commisurabili rispetto ai costi di implementazione.

Di seguito vengono elencati tre modi con cui un oggetto può ricevere un riferimento da un modulo esterno:

- **Interface injection**: l'oggetto fornisce un interfaccia che gli utenti possono implementare in modo da ottenere a runtime le dipendenze;
- **Setter injection**: il *dependent module* espone un metodo *setter* che il *framework*$_G$ usa per iniettarvi le dipendenze;
- **Constructor injection**: le dipendenze vengono fornite tramite il costruttore della classe.

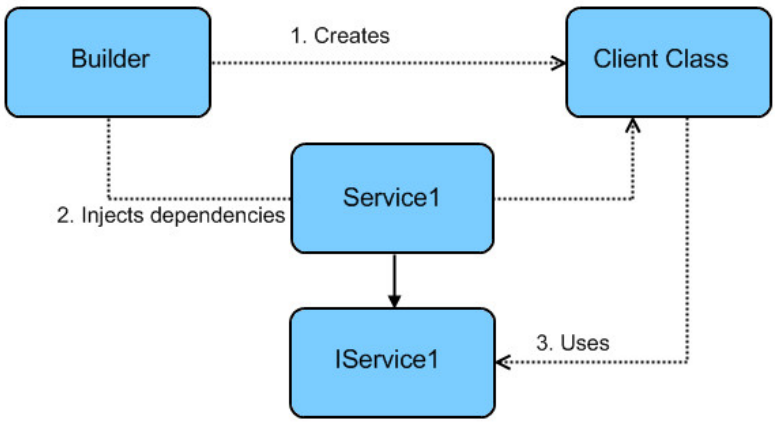

Figura 69: Struttura logica di Dependency Injection

A.5.4 Command

Il command pattern è uno dei *Design Pattern*$_G$ che 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*. L'obiettivo è rendere variabile l'azione del *client* senza però conoscere i dettagli dell'operazione stessa. Altro aspetto importante è che il destinatario della richiesta può non essere deciso staticamente all'atto dell'istanziazione del *Command* ma dev'essere ricavato a tempo di esecuzione. È possibile incapsulare un'azione in modo che questa sia atomica. È così possibile implementare un paradigma basato su transazioni in cui un insieme di operazioni è svolto in toto o per nulla.

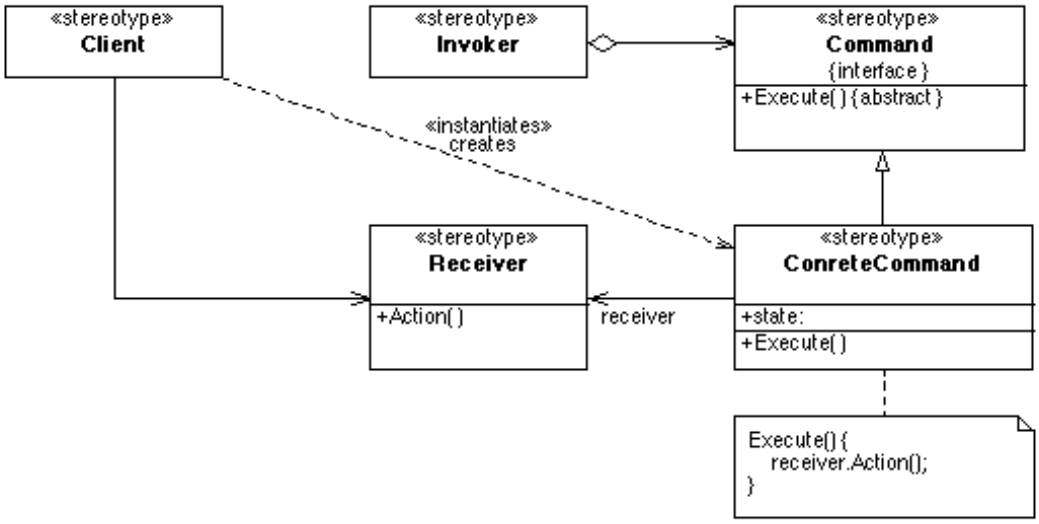

Figura 70: Struttura logica di Command