



Specifica Tecnica

Gruppo SteakHolders – Progetto MaaP

Informazioni sul documento

Versione	3.0.0
Redazione	Serena Girardi Gianluca Donato Federico Poli Nicolò Tresoldi Luca De Franceschi
Verifica	Enrico Rotundo
Approvazione	Luca De Franceschi
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
3.0.0	2014-02-05	Luca De Franceschi (Responsabile)	Approvazione.
2.2.0	2014-02-03	Enrico Rotundo (Verificatore)	Verifica.
2.1.8	2014-02-02	Nicolò Tresoldi (Progettista)	Stesura Stima di fattibilità.
2.1.7	2014-02-01	Nicolò Tresoldi (Progettista)	Stesura Tracciamento.
2.1.6	2014-02-01	Luca De Franceschi (Progettista)	Descrizione Design Pattern utilizzati.
2.1.5	2014-01-31	Federico Poli (Progettista)	Aggiornamento capitolo descrizione architettura.
2.1.4	2014-01-31	Serena Girardi (Progettista)	Aggiunta dei diagrammi di attività.
2.1.3	2014-01-29	Gianluca Donato (Progettista) Federico Poli (Progettista)	Stesura descrizione package e classi Front-end.
2.1.2	2014-01-29	Serena Girardi (Progettista)	Aggiunta degli scenari al Back-end.
2.1.1	2014-01-28	Serena Girardi (Progettista) Federico Poli (Progettista)	Stesura descrizione package e classi Back-end.
2.1.0	2014-01-22	Nicolò Tresoldi (Verificatore)	Verifica.
2.0.6	2014-01-18	Giacomo Fornari (Progettista)	Stesura diagramma classi Front-end.
2.0.5	2014-01-18	Serena Girardi (Progettista)	Stesura appendice Design Pattern.
2.0.4	2014-01-18	Federico Poli (Progettista)	Aggiunto diagramma dei package.
2.0.3	2014-01-16	Federico Poli (Progettista)	Stesura capitolo descrizione architettura.
2.0.2	2014-01-15	Gianluca Donato (Progettista)	Stesura capitolo Tecnologie utilizzate.
2.0.1	2014-01-14	Federico Poli (Progettista)	Stesura capitolo introduzione.
2.0.0	2014-01-14	Gianluca Donato (Progettista)	Creazione scheletro.



Indice

1	Introduzione	7
1.1	Scopo del documento	7
1.2	Scopo del prodotto	7
1.3	Glossario	7
1.4	Riferimenti	7
1.4.1	Normativi	7
1.4.2	Informativi	7
2	Tecnologie utilizzate	9
2.1	Node.js	9
2.2	Express.js	9
2.3	MongoDB	9
2.4	Mongoose	10
3	Descrizione architettura	11
3.1	Metodo e formalismo di specifica	11
3.2	Architettura generale	12
3.3	Interfaccia REST-like	12
3.3.1	Back-end	14
3.3.2	Front-End	15
4	Back-end	16
4.1	Interfaccia REST	16
4.2	Descrizione packages e classi	18
4.2.1	Back-end	18
4.2.1.1	Informazioni sul package	18
4.2.2	Back-end::Lib	20
4.2.2.1	Informazioni sul package	20
4.2.3	Back-end::Lib::AuthModel	21
4.2.3.1	Informazioni sul package	21
4.2.3.2	Classi	21
4.2.4	Back-end::DeveloperProject	22
4.2.4.1	Informazioni sul package	22
4.2.4.2	Classi	22
4.2.5	Back-end::Lib::DSLModel	23
4.2.5.1	Informazioni sul package	23
4.2.5.2	Classi	24
4.2.6	Back-end::Lib::MailView	28
4.2.6.1	Informazioni sul package	28
4.2.6.2	Classi	28
4.2.7	Back-end::Lib::Middleware	29
4.2.7.1	Informazioni sul package	29
4.2.7.2	Classi	29
4.2.8	Back-end::Lib::Controller	32
4.2.8.1	Informazioni sul package	32
4.2.8.2	Classi	32



4.3 Scenari	35
4.3.1 Gestione generale delle richieste	35
4.3.2 Fallimento vincolo “utente autenticato”	38
4.3.3 Fallimento vincolo “utente non autenticato”	39
4.3.4 Fallimento vincolo “utente admin”	40
4.3.5 Richiesta POST /login	41
4.3.6 Richiesta DELETE /logout	42
4.3.7 Richiesta GET /profile	43
4.3.8 Richiesta PUT /profile	44
4.3.9 Richiesta POST /password/forgot	45
4.3.10 Richiesta GET /users	46
4.3.11 Richiesta POST /users	47
4.3.12 Richiesta GET /users/{user id}	48
4.3.13 Richiesta PUT /users/{user id}	49
4.3.14 Richiesta DELETE /users/{user id}	50
4.3.15 Richiesta GET /collection	51
4.3.16 Richiesta GET /collection/{collection name}	52
4.3.17 Richiesta GET /collection/{collection name}/{document id}	53
4.3.18 Richiesta PUT /collection/{collection name}/{document id}	54
4.3.19 Richiesta DELETE /collection/{collection name}/{document id}	55
4.3.20 Richiesta PUT /action/{action name}/{collection name}	56
4.3.21 Richiesta PUT /action/{action name}/{collection name}/{document id}	57
4.4 Descrizione librerie aggiuntive	58
5 Front-end	59
5.1 Descrizione packages e classi	59
5.1.1 Front-end	59
5.1.1.1 Informazioni sul package	59
5.1.1.2 Classi	62
5.1.2 Front-end::Controllers	62
5.1.2.1 Informazioni sul package	62
5.1.2.2 Classi	62
5.1.3 Front-end::Services	65
5.1.3.1 Informazioni sul package	65
5.1.3.2 Classi	66
5.1.4 Front-end::Model	69
5.1.4.1 Informazioni sul package	69
5.1.4.2 Classi	69
6 Diagrammi di attività	72
6.1 Applicazione MaaP	72
6.1.1 Attività principali	73
6.1.2 Effettua registrazione	74
6.1.3 Recupera password	75
6.1.4 Esegui reset password	75
6.1.5 Effettua login	76
6.1.6 Modifica profilo	77
6.1.7 Index-page Collection	78
6.1.8 Show-page Document	79
6.1.9 Modifica Document	80



6.1.10	Apri pagina gestione utenti	81
6.1.11	Apri show-page utente	82
6.1.12	Crea un nuovo utente	83
6.2	Framework MaaP	83
6.2.1	Crea nuova applicazione	84
6.2.2	Creazione cartella front-end di default	85
7	Stime di fattibilità e di bisogno di risorse	86
8	Design pattern	87
8.1	Design Pattern Architetturali	87
8.1.1	MVC	87
8.1.2	MVV	87
8.1.3	Middleware	88
8.2	Design Pattern Creazionali	88
8.2.1	Registry	88
8.2.2	Factory method	88
8.2.3	Singleton	88
8.3	Design Pattern Strutturali	88
8.3.1	Facade	88
8.4	Design Pattern Comportamentali	89
8.4.1	Chain of responsibility	89
8.4.2	Strategy	89
8.4.3	Command	89
9	Tracciamento	90
9.1	Tracciamento componenti - requisiti	90
9.2	Tracciamento requisiti - componenti	93
Appendici		96
A	Descrizione Design Pattern	96
A.1	Design Pattern Architetturali	96
A.1.1	MVC	96
A.1.2	Middleware	97
A.2	Design Pattern Creazionali	98
A.3	Singleton	98
A.3.1	Registry	99
A.3.2	Factory method	100
A.4	Design Pattern Strutturali	100
A.4.1	Facade	100
A.5	Design Pattern Comportamentali	101
A.5.1	Chain of Responsibility	101
A.5.2	Strategy	102
A.5.3	Dependency Injection	103
A.5.4	Command	104



Elenco delle tabelle

Elenco delle figure

1	Diagramma di deployment	12
2	Diagramma dei package del Back-end	14
3	Diagramma dei package del Front-end	15
4	Diagramma dei packages Back-end	18
5	Diagramma delle classi Back-end	19
6	Componente Back-end::Lib	20
7	Componente Back-end::Lib::AuthModel	21
8	Componente Back-end::DeveloperProject	22
9	Componente Back-end::Lib::DSLModel	23
10	Componente Back-end::Lib::MailView	28
11	Componente Back-end::Lib::Middleware	29
12	Componente Back-end::Lib::Controller	32
13	Diagramma Gestione Richiesta	36
14	Diagramma Routing Richiesta	37
15	Fallimento vincolo “utente autenticato”	38
16	Fallimento vincolo “utente non autenticato”	39
17	Fallimento vincolo “utente admin”	40
18	Richiesta POST /login	41
19	Richiesta DELETE /logout	42
20	Richiesta GET /profile	43
21	Richiesta PUT /profile	44
22	Richiesta POST /password/forgot	45
23	Richiesta GET /users	46
24	Richiesta POST /users	47
25	Richiesta GET /users/{user id}	48
26	Richiesta PUT /users/{user id}	49
27	Richiesta DELETE /users/{user id}	50
28	Richiesta GET /collection	51
29	Richiesta GET /collection/{collection name}	52
30	Richiesta GET /collection/{collection name}/{document id}	53
31	Richiesta PUT /collection/{collection name}/{document id}	54
32	Richiesta DELETE /collection/{collection name}/{document id}	55
33	Richiesta PUT /action/{action name}/{collection name}	56
34	Richiesta PUT /action/{action name}/{collection name}/{document id}	57
35	Diagramma dei packages Front-end	59
36	Diagramma delle classi Front-end	61
37	Componente Front-end::Controllers	62
38	Componente Front-end::Services	65
39	Componente Front-end::Model	69
40	Diagramma di attività - Attività principali di un’applicazione MaaP	73
41	Diagramma di attività - Registrazione di un utente	74
42	Diagramma di attività - Recupero password	75
43	Diagramma di attività - Reset della password dell’utente	75



44	Diagramma di attività - Login dell’utente	76
45	Diagramma di attività - Modifica profilo utente	77
46	Diagramma di attività - Visualizzazione index-page della Collection selezionata .	78
47	Diagramma di attività - Visualizzazione show-page del Document selezionato .	79
48	Diagramma di attività - Modifica del Document selezionato	80
49	Diagramma di attività - Pagina di gestione degli utenti	81
50	Diagramma di attività - Pagina di visualizzazione di un utente	82
51	Diagramma di attività - Pagina di creazione di un nuovo utente	83
52	Diagramma di attività - Creazione scheletro nuova applicazione	84
53	Diagramma di attività - Creazione scheletro nuova applicazione	85
54	Struttura logica di Model-View-Controller	96
55	Struttura logica di Middleware	98
56	Struttura logica di singleton	99
57	Struttura logica di registry	99
58	Struttura logica di factory method	100
59	Struttura logica di facade	101
60	Struttura del chain of responsibility.	102
61	Struttura del chain of responsibility.	103
62	Struttura logica di Dependency Injection	104
63	Struttura logica di Command	105



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 fruttore finale delle pagine generate sarà infine l'esperto di business 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 v3.0.0*, contenente la definizione dei termini marcati con una G pedice.

1.4 Riferimenti

1.4.1 Normativi

- *Norme di Progetto v3.0.0*
- 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 v3.0.0*

1.4.2 Informativi

- Presentazione capitolato d'appalto: <http://www.math.unipd.it/~tullio/IS-1/2013/Progetto/C1.pdf>;
- Ingegneria del software - Ian Sommerville - 8a edizione (2007);



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

2.1 Node.js

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

I principali vantaggi dell'utilizzo di *Node.js* sono:

- **Approccio asincrono**: *Node.js* permette di accedere alle risorse del sistema operativo in modalità *event-driven* 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* è molto facile organizzare il lavoro in librerie, importare i *moduli* e combinarli fra loro. Questo è reso molto comodo attraverso il node package manager (**npm**) attraverso il quale lo sviluppatore può contribuire e accedere ai *package* messi a disposizione dalla community.

2.2 Express.js

Express.js è un *framework* minimale per creare applicazioni web con *Node.js*. Richiede *moduli* Node di terze parti per applicazioni che prevedono l'interazione con le *basi di dati*. È stato utilizzato il *framework* *Express.js* per supportare lo sviluppo dell'application server grazie alle utili e robuste features da esso offerte, le quali sono pensate per non oscurare le funzionalità fornite da *Node.js* aprendo così le porte all'utilizzo di moduli per *Node.js* atti a supportare specifiche funzionalità.

2.3 MongoDB

MongoDB è un database *NoSQL* *open source* scalabile e altamente performante di tipo document-oriented, in cui i dati sono archiviati sotto forma di documenti in stile *JSON* con schemi dinamici, 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 definire query complesse utilizzando un linguaggio che non è SQL_G ;
- Permette di processare parallelamente i dati ($Map\text{-}Reduce_G$);
- Tipi di dato più flessibili.

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.

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



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 immediato la gestione dinamica dei tipi e le caratteristiche tipiche degli stili di programmazione funzionali. In certi casi è stato necessario introdurre interfacce “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.

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-practise.

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 molto 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.

¹ Descritto alla sezione 11.3.2 del libro “Ingegneria del software - Sommerville - 8a edizione (2007)”



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 $back-end_g$.

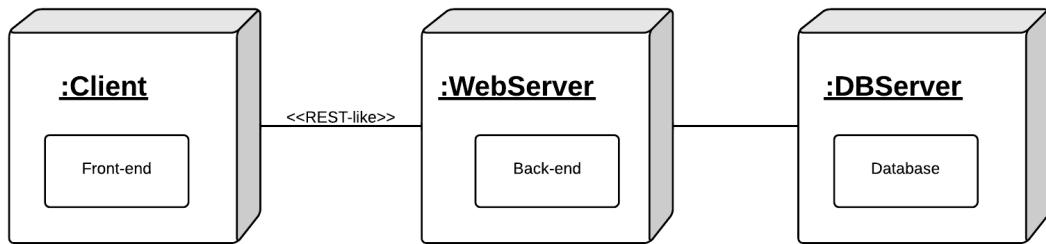


Figura 1: Diagramma di deployment

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;

$REST_g$ 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 indicato, espresso in un appropriato formato.
PUT	Non usata.	Sostituisce l'elemento della collection indicato, o se non esiste, lo crea .
POST	Crea un nuovo elemento nella collection. La URI del nuovo elemento è generata automaticamente 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

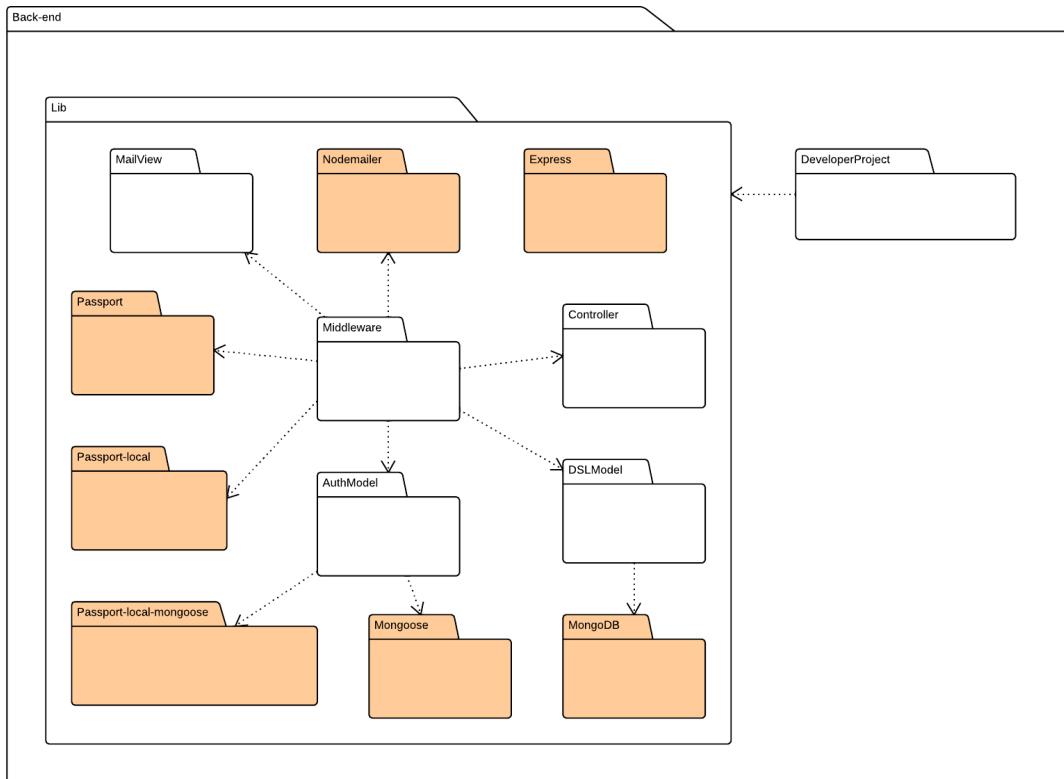


Figura 2: Diagramma dei package del Back-end

L'architettura del $back-end_G$ si serve del pattern architettonicale 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

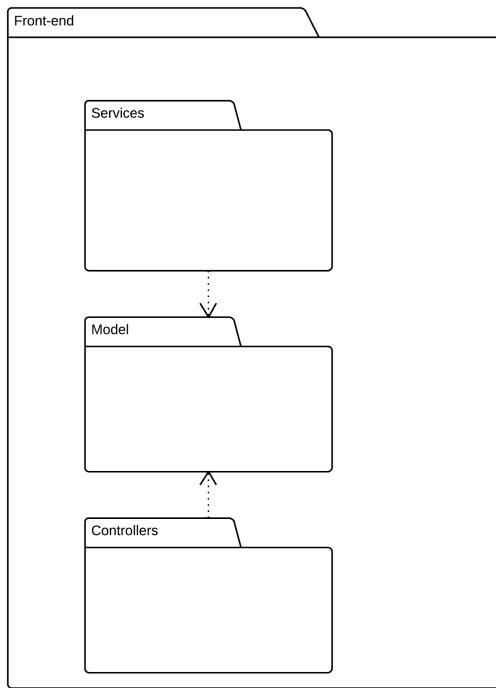


Figura 3: Diagramma dei package del Front-end

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; vengono caricati da *AngularJS_G* nel momento in cui il controller ne fa 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.

/login	POST	Utente
Crea una nuova sessione associata all'utente, corrisponde al login.		

/logout	DELETE	Utente Autenticato
Elimina la sessione utente, corrisponde al logout.		

/profile	GET	Utente Autenticato
Restituisce i dati relativi all'utente.		
/profile	PUT	Utente Autenticato
Modifica i dati utente.		

/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

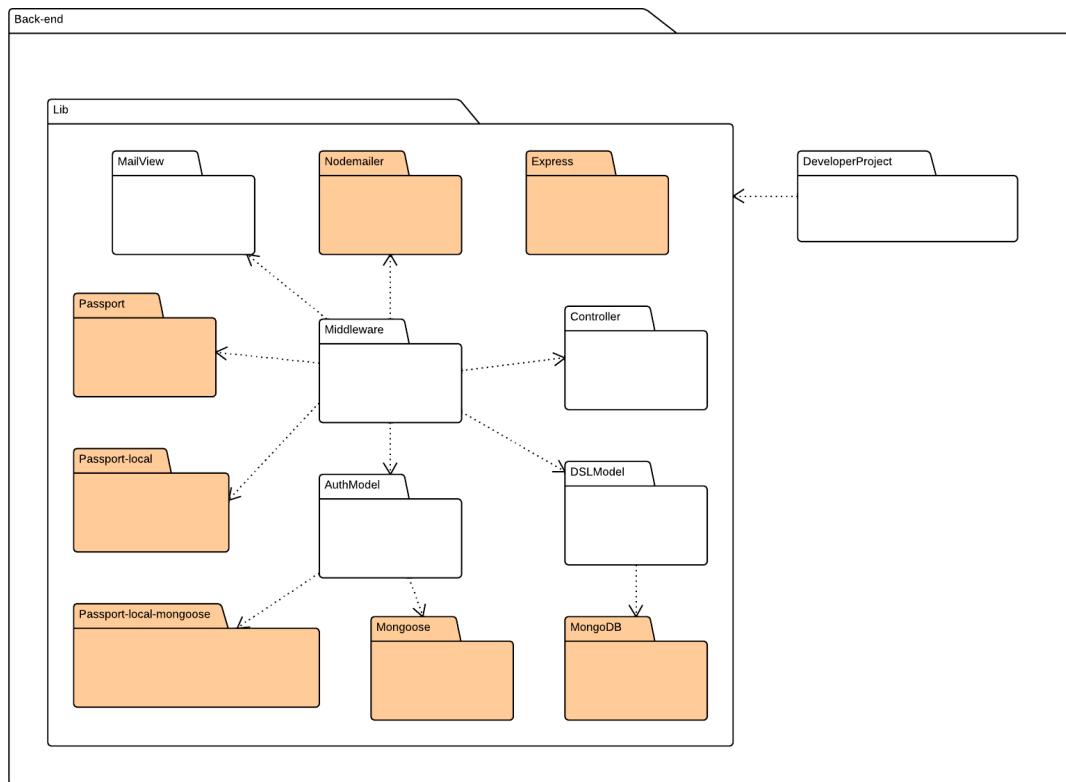


Figura 4: Diagramma dei packages Back-end

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.

I packages colorati nel diagramma, rappresentano librerie esterne.

4.2.1.1.2 Package contenuti

- Back-end::DeveloperProject
 - Back-end::Lib

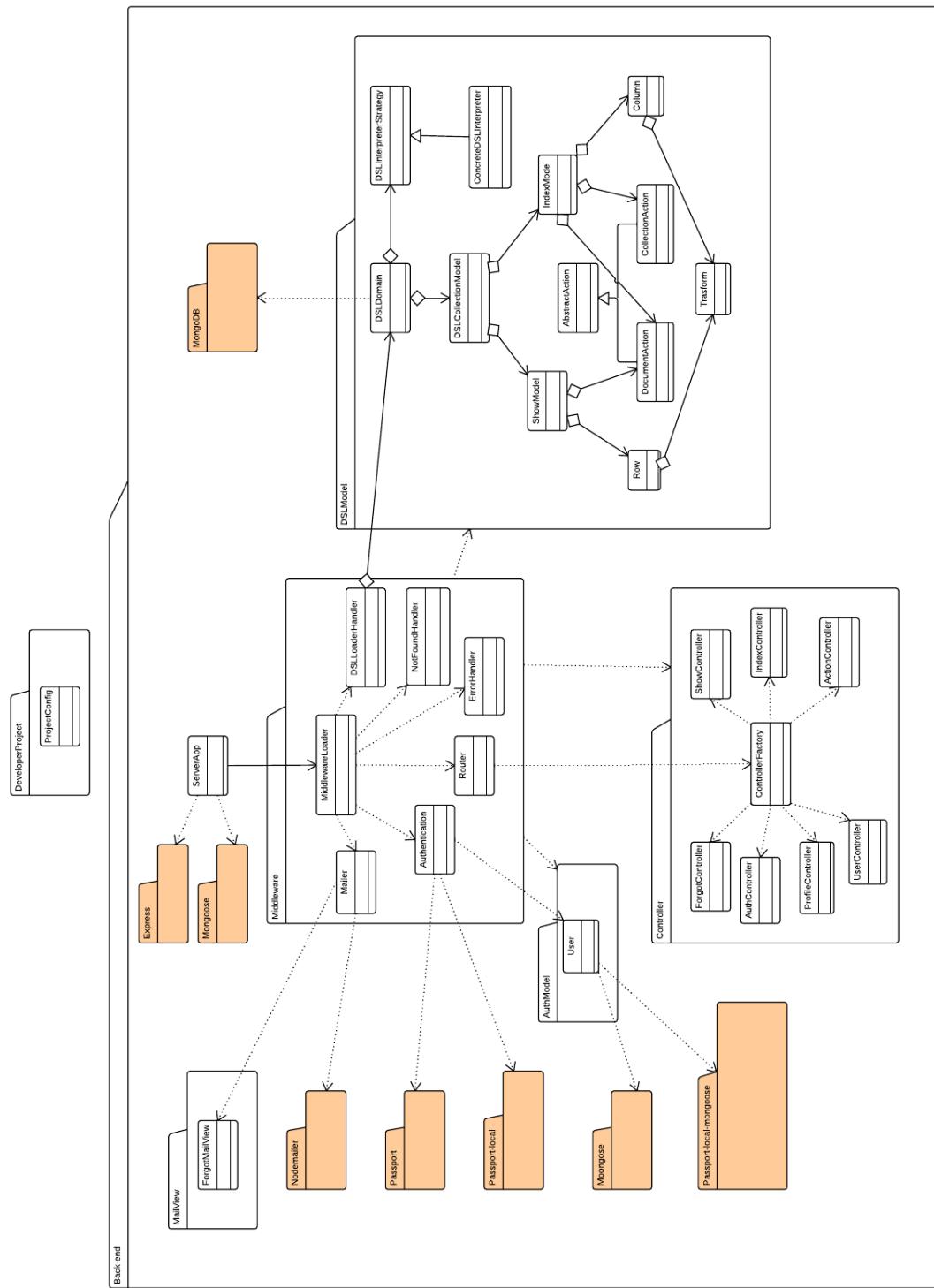


Figura 5: Diagramma delle classi Back-end



4.2.1.1.3 Diagramma delle classi

4.2.2 Back-end::Lib

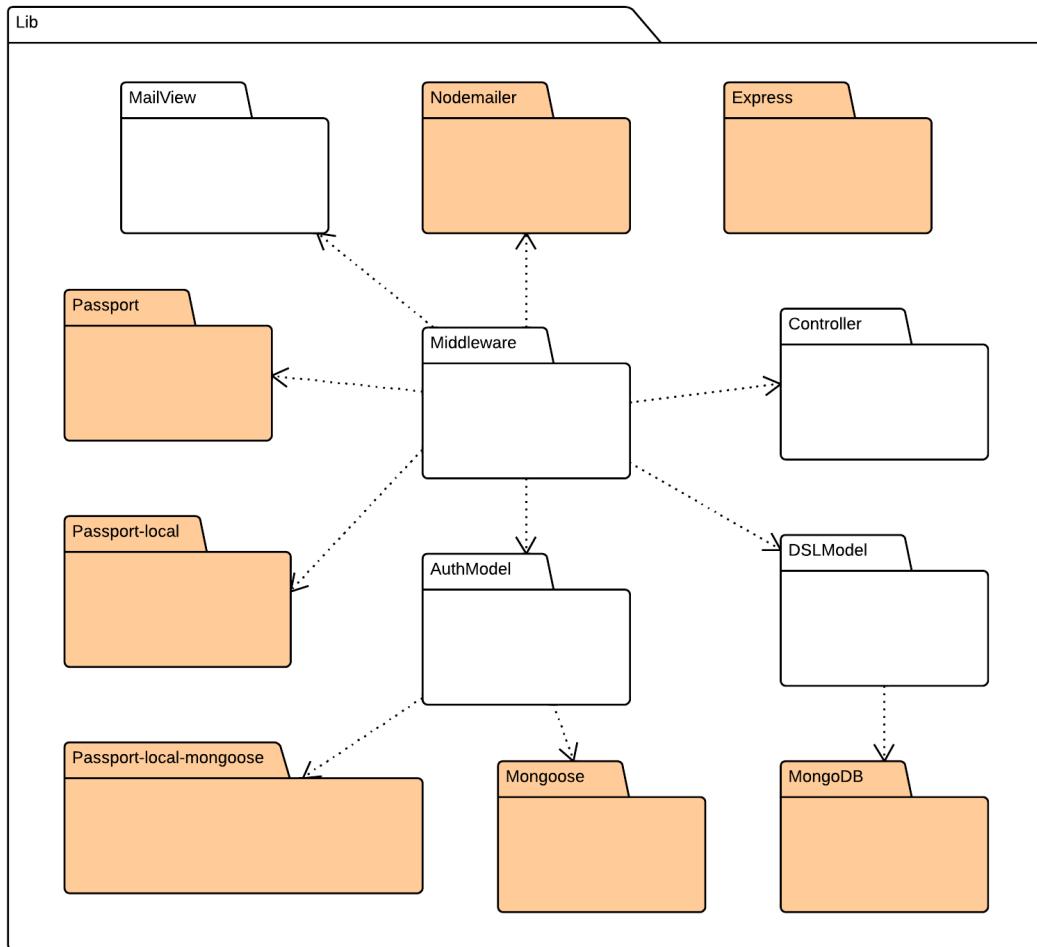


Figura 6: Componente Back-end::Lib

4.2.2.1 Informazioni sul package

4.2.2.1.1 Descrizione

Package_G che costituisce la libreria principale dell'applicazione MaaP, 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.



4.2.2.1.2 Package contenuti

- Back-end::Lib::AuthModel
- Back-end::Lib::DSLModel
- Back-end::Lib::MailView
- Back-end::Lib::Middleware
- Back-end::Lib::Controller

4.2.3 Back-end::Lib::AuthModel

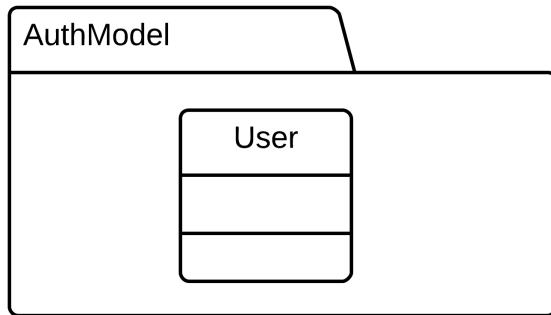


Figura 7: Componente Back-end::Lib::AuthModel

4.2.3.1 Informazioni sul package

4.2.3.1.1 Descrizione

Package_G che gestisce i dati e le operazioni relativi all'autenticazione utente, andando ad aggiungersi alle componenti che compongono la parte model nell'architettura MVC nel back-end.

4.2.3.2 Classi

4.2.3.2.1 Back-end::Lib::AuthModel::User

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.

4.2.4 Back-end::DeveloperProject

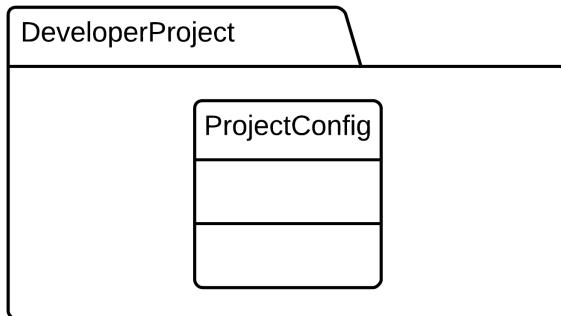


Figura 8: Componente Back-end::DeveloperProject

4.2.4.1 Informazioni sul package

4.2.4.1.1 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.

4.2.4.2 Classi

4.2.4.2.1 Back-end::DeveloperProject::ProjectConfig

Descrizione

Classe che si occupa di configurare il progetto creato dallo sviluppatore.

Utilizzo

Viene utilizzata per descrivere tutti i parametri dell’applicazione. Quando viene creata una *Back-end::Lib::ServerApp* le viene passato un oggetto di questo tipo ed essa avvierà l’applicazione a partire da questa configurazione.



4.2.5 Back-end::Lib::DSLModel

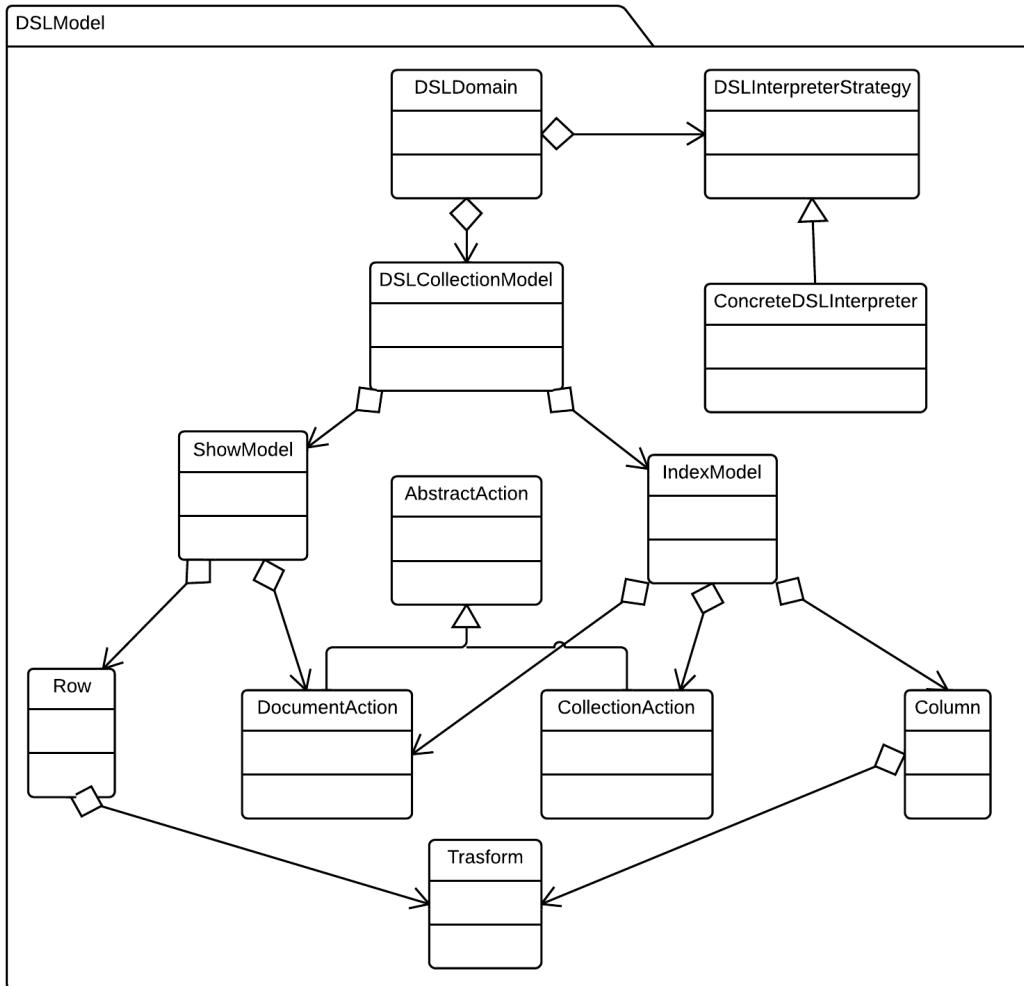


Figura 9: Componente Back-end::Lib::DSLModel

4.2.5.1 Informazioni sul package

4.2.5.1.1 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.



4.2.5.2 Classi

4.2.5.2.1 Back-end::Lib::DSLModel::Row

Descrizione

Classe che racchiude tutte le informazioni relative ad un elemento (una riga) della 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).

4.2.5.2.2 Back-end::Lib::DSLModel::AbstractAction

Descrizione

Questa classe è una classe astratta che rappresenta le azioni definibili dal developer nel DSL, e che potranno essere azionata tramite le API_G del $Back-end_G$ per essere eseguite lato-server.

Utilizzo

Questa classe viene utilizzata per raggruppare elementi comuni alle classi DocumentAction e CollectionAction.

Classi Figlie

- Back-end::Lib::DSLModel::AbstractAction::CollectionAction
- Back-end::Lib::DSLModel::AbstractAction::DocumentAction

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

4.2.5.2.4 Back-end::Lib::DSLModel::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 encapsulare 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.

Classi Figlie

- Back-end::Lib::DSLModel::DSLInterpreterStrategy::ConcreteDSLInterpreter

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

4.2.5.2.6 Back-end::Lib::DSLModel::DSLInterpreterStrategy::ConcreteDSLInterpreter

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::DSLInterpreter 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::DSLModel::DSLInterpreterStrategy

4.2.5.2.7 Back-end::Lib::DSLModel::IndexModel

Descrizione

Classe che racchiude tutte le informazioni relative ad una index-page. Tali informazioni vengono dichiarate dal developer nel DSL. Comprende a sua volta altre classi di tipo Column, DocumentAction e CollectionAction.



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::DSLModel::AbstractAction::CollectionAction
- Back-end::Lib::DSLModel::AbstractAction::DocumentAction

4.2.5.2.8 Back-end::Lib::DSLModel::ShowModel

Descrizione

Classe che racchiude tutte le informazioni relative ad una show-page. Tali informazioni vengono dichiarate dal developer nel DSL. Comprende a sua volta altre classi di tipo Row e DocumentAction.

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::DSLModel::Row

4.2.5.2.9 Back-end::Lib::DSLModel::Trasform

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.

Utilizzo

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

4.2.5.2.10 Back-end::Lib::DSLModel::Column

Descrizione

Classe che racchiude tutte le informazioni relative ad una riga della 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).

Relazioni con altre classi

- Back-end::Lib::DSLModel::Trasform



4.2.5.2.11 Back-end::Lib::DSLModel::AbstractAction::CollectionAction

Descrizione

Questa classe descrive un'azione definita dal developer nel DSL, che potrà essere azionata tramite le API_G del $Back-end_G$ ed eseguita lato-server. Tale azione potrà eseguire operazioni sulla collection corrispondente al `DSLCollectionModel` che contiene indirettamente questa classe. La classe assume il ruolo di Command descritto dal $design\ pattern_G$ Command. Gli oggetti che avranno il ruolo di `ConcreteCommand` verranno creati dinamicamente, definendo a run-time dei metodi aggiuntivi per questa classe.

Utilizzo

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

Classi Ereditate

- `Back-end::Lib::DSLModel::AbstractAction`

4.2.5.2.12 Back-end::Lib::DSLModel::AbstractAction::DocumentAction

Descrizione

Questa classe descrive un'azione definita dal developer nel DSL, che potrà essere azionata tramite le API_G del $Back-end_G$ ed eseguita lato-server. Tale azione potrà eseguire operazioni su un document della collection corrispondente al `DSLCollectionModel` che contiene indirettamente questa classe. La classe assume il ruolo di Command descritto dal $design\ pattern_G$ Command. Gli oggetti che avranno il ruolo di `ConcreteCommand` verranno creati dinamicamente, definendo a run-time dei metodi aggiuntivi per questa classe.

Utilizzo

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

Classi Ereditate

- `Back-end::Lib::DSLModel::AbstractAction`



4.2.6 Back-end::Lib::MailView

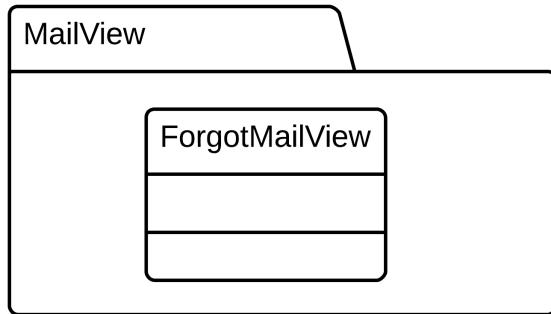


Figura 10: Componente Back-end::Lib::MailView

4.2.6.1 Informazioni sul package

4.2.6.1.1 Descrizione

$Package_G$ contenente le classi che costituiscono i template utilizzati per le email di recupero-password. Verranno utilizzate dal $middleware_G$ Mailer.

4.2.6.2 Classi

4.2.6.2.1 Back-end::Lib::MailView::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.



4.2.7 Back-end::Lib::Middleware

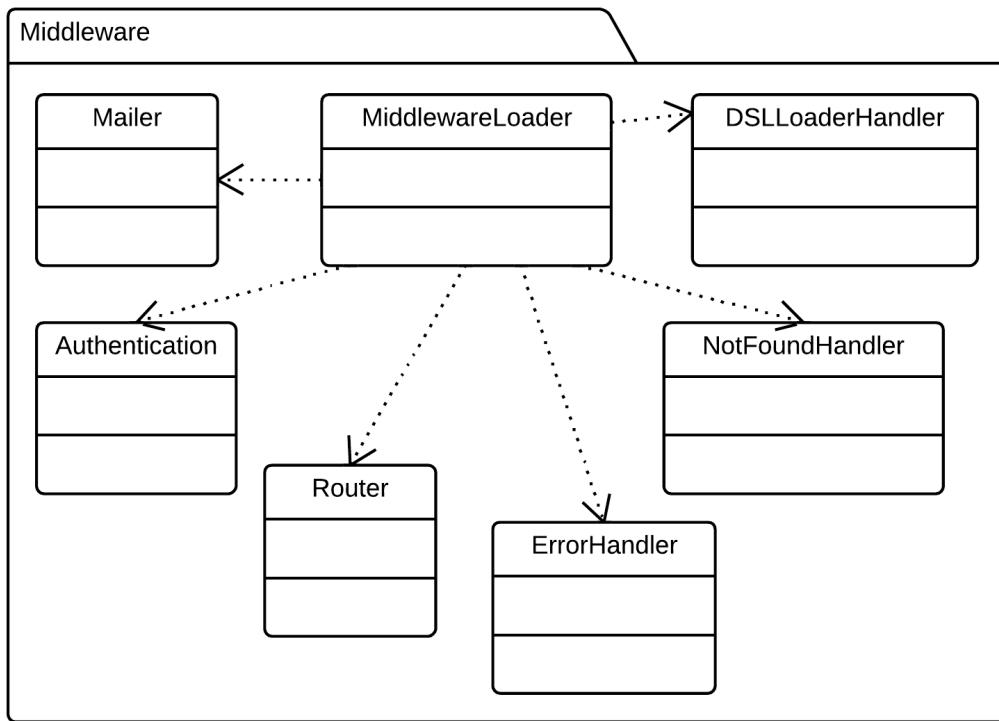


Figura 11: Componente Back-end::Lib::Middleware

4.2.7.1 Informazioni sul package

4.2.7.1.1 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 dell' *application logic_G* nell'architettura *MVC_G* del *Back-end_G*.

4.2.7.2 Classi

4.2.7.2.1 Back-end::Lib::Middleware::Router

Descrizione



Classe che si occupa della richiesta di risorse. È uno dei componenti subsystem class del *Design Pattern_G Facade_G* e handler del *Design Pattern_G Chain of responsibility_G*.

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`.

4.2.7.2.2 Back-end::Lib::Middleware::Authentication

Descrizione

Classe che si occupa dell'autenticazione di un'utente. 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 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::AuthModel::User`

4.2.7.2.3 Back-end::Lib::Middleware::DSLLoaderHandler

Descrizione

Classe che si occupa di caricare i DSL_G presenti nel sistema. 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 caricare i DSL_G delle *Collection_G* all'interno del *database_G*.

Relazioni con altre classi

- `Back-end::Lib::DSLModel::DSLDomain`

4.2.7.2.4 Back-end::Lib::Middleware::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.

Relazioni con altre classi

- `Back-end::Lib::MailView::ForgotMailView`



4.2.7.2.5 Back-end::Lib::Middleware::MiddlewareLoader

Descrizione

Classe che definisce un'interfaccia comune per tutte le richieste dell'applicazione. È la componente facade del *Design Pattern_G Facade_G* e handler 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::Middleware::Router
- Back-end::Lib::Middleware::Authentication
- Back-end::Lib::Middleware::DSLLoaderHandler
- Back-end::Lib::Middleware::Mailer
- Back-end::Lib::Middleware::NotFoundHandler

4.2.7.2.6 Back-end::Lib::Middleware::NotFoundHandler

Descrizione

Classe che si occupa la gestione dell'errore di pagina non trovata. 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 generare una pagina 404 di errore nel caso in cui l'*URI_G* passato non corrisponda ad una risorsa presente nell'applicazione.

4.2.7.2.7 Back-end::Lib::Middleware::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 subsystem class del *Design Pattern_G Facade_G* e handler 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.



4.2.8 Back-end::Lib::Controller

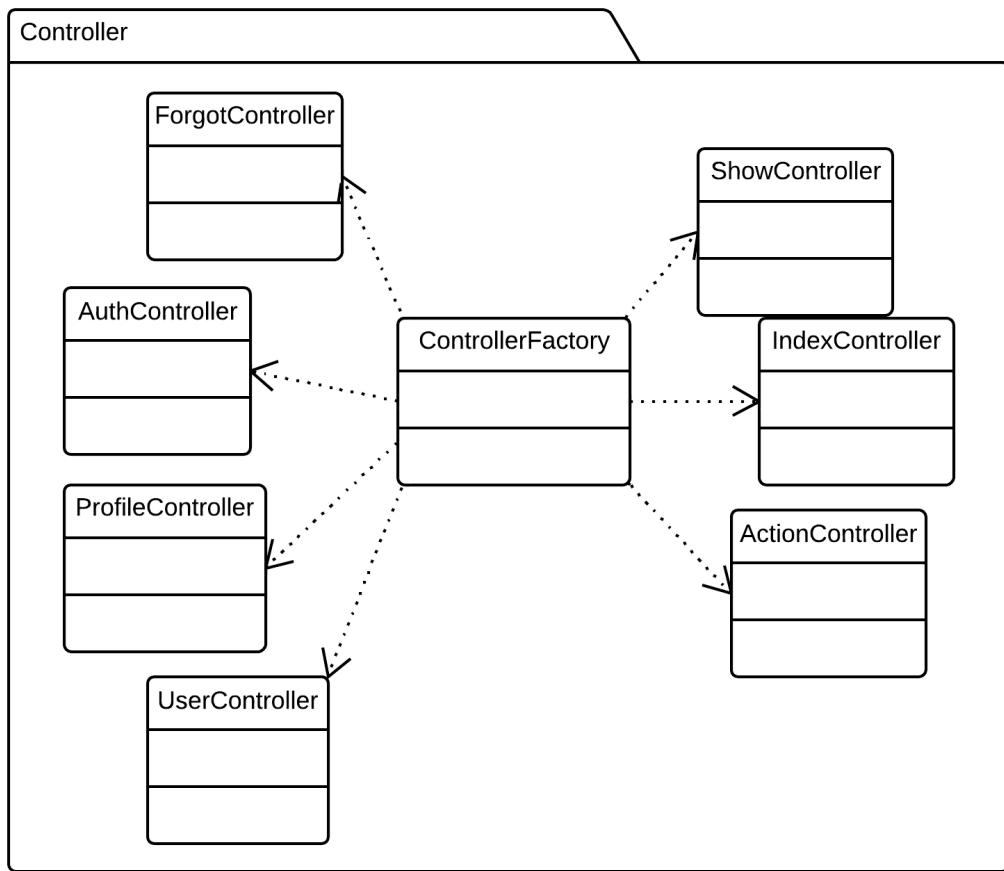


Figura 12: Componente Back-end::Lib::Controller

4.2.8.1 Informazioni sul package

4.2.8.1.1 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*.

4.2.8.2 Classi



4.2.8.2.1 Back-end::Lib::Controller::UserController

Descrizione

Classe che si occupa delle 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.

4.2.8.2.2 Back-end::Lib::Controller::ActionController

Descrizione

Classe che rappresenta i metodi per la gestione della risorsa corrispondente ad un'azione. È uno dei componenti product del *Design Pattern_G Factory method_G*.

Utilizzo

Viene utilizzata per gestire le azioni personalizzate predisposte nelle pagine index-page e show-page, delegando alla classe `Back-end::Lib::DSLModel::DSLDomain` la loro esecuzione avendo quest'ultima l'implementazione delle stesse.

4.2.8.2.3 Back-end::Lib::Controller::ControllerFactory

Descrizione

Classe che si occupa di istanziare e restituire una classe *Controller*. Rappresenta il componente creator del *Design Pattern_G Factory method_G*.

Utilizzo

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

4.2.8.2.4 Back-end::Lib::Controller::AuthController

Descrizione

Classe che rappresenta i metodi per la gestione delle risorse di login e logout. È uno dei componenti product del *Design Pattern_G Factory method_G*.

Utilizzo

Viene 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*.



4.2.8.2.5 Back-end::Lib::Controller::ProfileController

Descrizione

Classe che rappresenta la gestione di un profilo utente. 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.

4.2.8.2.6 Back-end::Lib::Controller::ShowController

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` la sua visualizzazione.

4.2.8.2.7 Back-end::Lib::Controller::IndexController

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*, offrendo metodi per restituirlne gli attributi, effettuarne la modifica o la cancellazione e delega la visualizzazione dell'index-page alla classe `Back-end::Lib::DSLModel::DSLDomain`.

4.2.8.2.8 Back-end::Lib::Controller::ForgotController

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.



4.3 Scenari

4.3.1 Gestione generale delle richieste

Nel diagramma 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 viene rappresentata una generica richiesta al server: utilizzando il pattern della Chain of responsibility, il server invoca in sequenza i middleware, passando come parametri l'oggetto della richiesta, della risposta e una callback per passare il controllo al successivo middleware. In caso di errore, un middleware può chiamare la callback passandogli la descrizione dell'errore come parametro `next(error)`. In questo modo il server passerà il controllo al primo middleware in grado di gestire un errore. Come terza alternativa, un middleware



può terminare la propria esecuzione con un `return`. In tal caso la richiesta non verrà più gestita da nessun altro middleware.

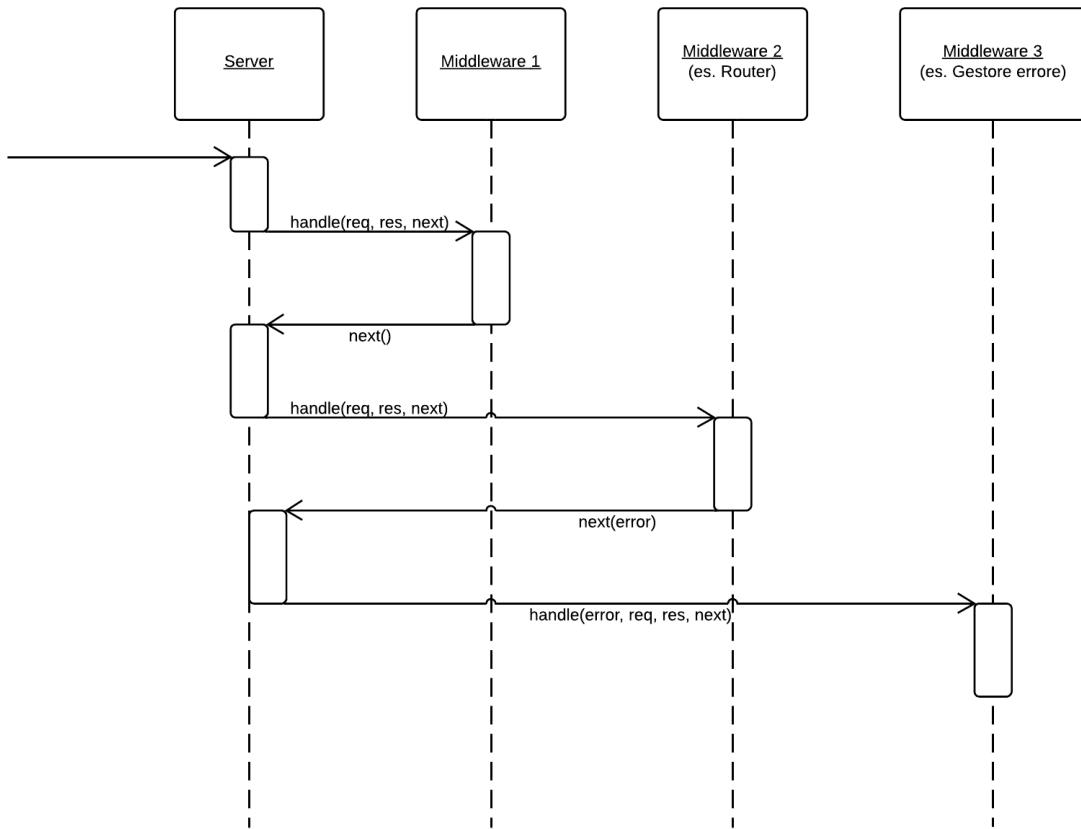


Figura 13: Diagramma Gestione Richiesta

Nel seguente diagramma viene mostrato il comportamento di routing, dove si intende che ogni controllore ha associato un'espressione regolare che specifica su quali richieste agisce.

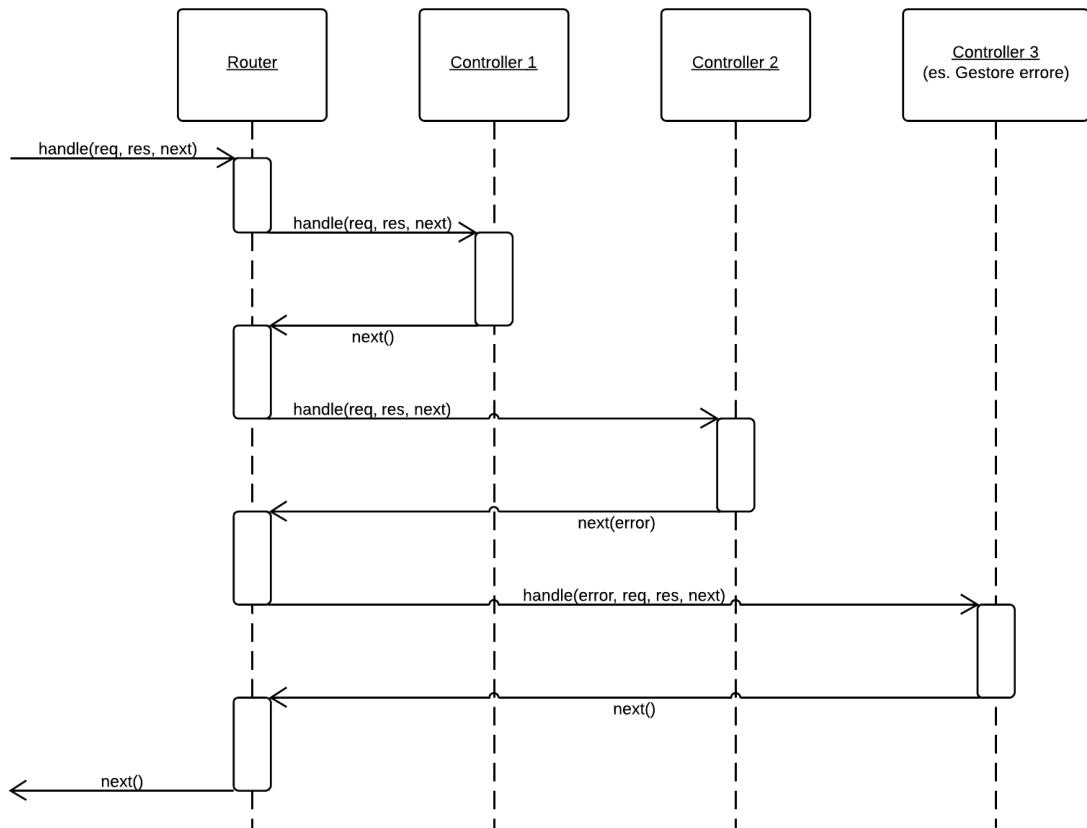


Figura 14: Diagramma Routing Richiesta



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 dal controller *requireLogged* che manda un *next(error)* per il fallimento di tale vincolo al router il quale avrà compito di gestirlo.

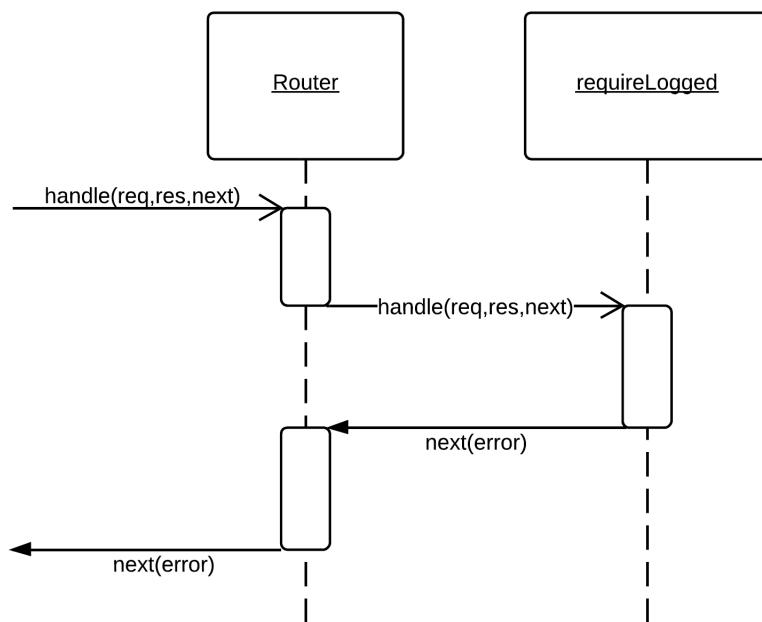


Figura 15: Fallimento vincolo “utente autenticato”



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 *requireNotLogged* che verifica con esito negativo i permessi e rimanda un next(error) al router.

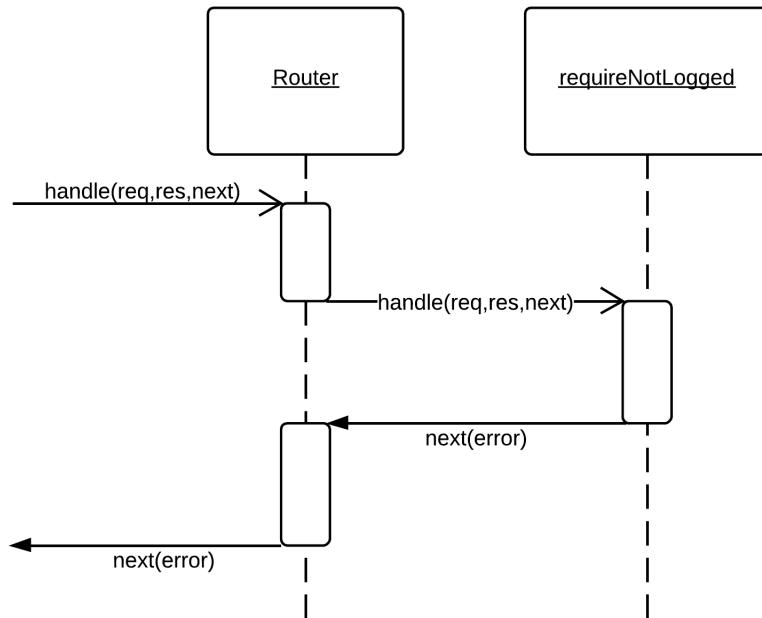


Figura 16: Fallimento vincolo “utente non autenticato”



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 controller *requireAdmin* fallisce.

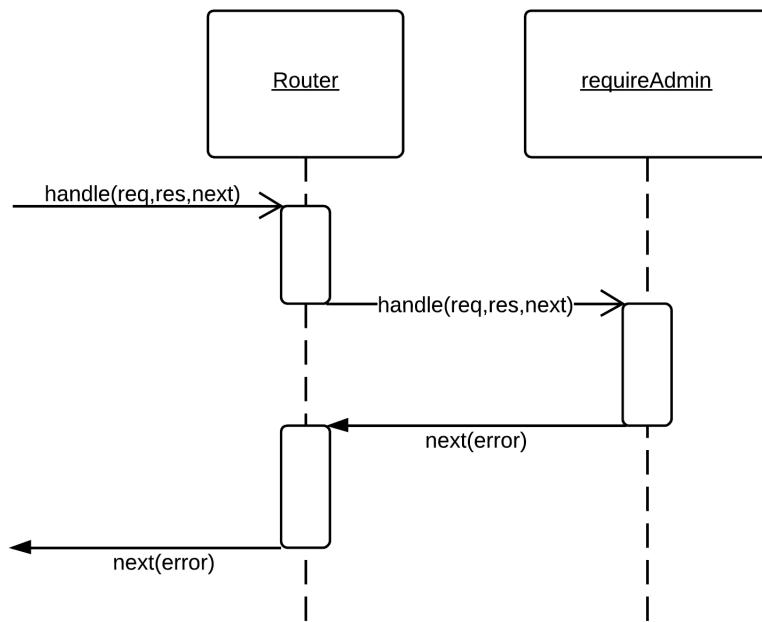


Figura 17: Fallimento vincolo “utente admin”



4.3.5 Richiesta POST /login

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

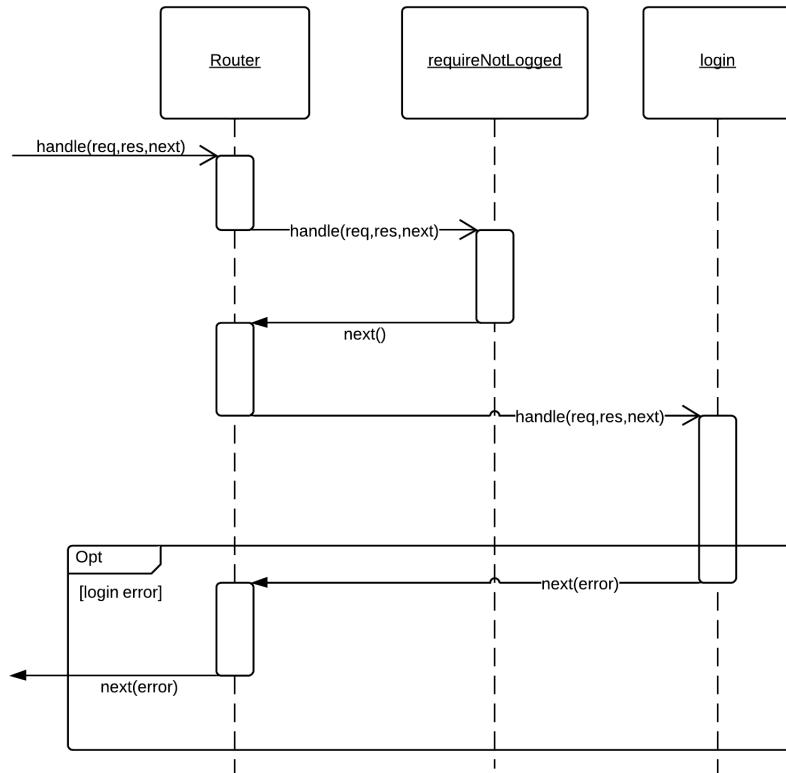


Figura 18: Richiesta POST /login



4.3.6 Richiesta DELETE /logout

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

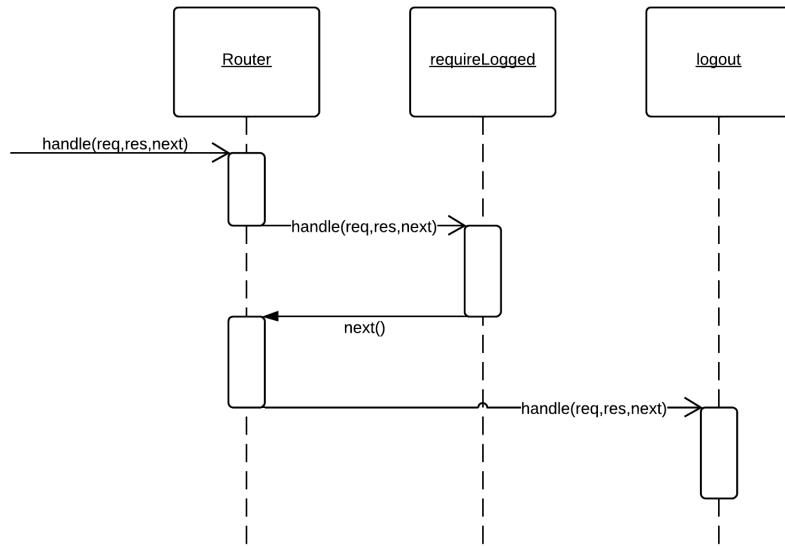


Figura 19: Richiesta DELETE /logout



4.3.7 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 *getProfile*.

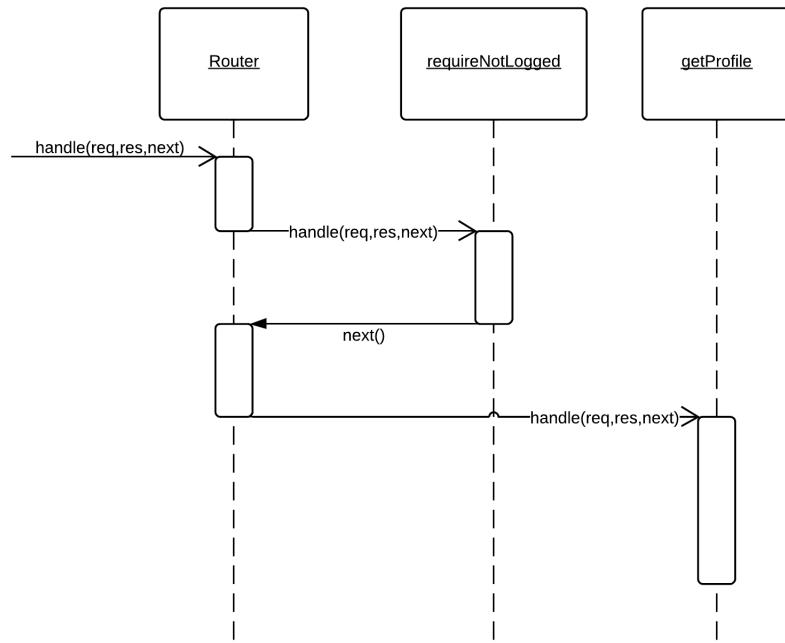


Figura 20: Richiesta GET /profile



4.3.8 Richiesta PUT /profile

Viene rappresentato lo scenario di una richiesta PUT per la risorsa Profile, il *requireLogged* non ritorna un errore, chiamando il successivo *middleware_G* *editProfile* che gestisce la richiesta di modifica dei dati del profilo. Nell'opzione che i parametri passati per la modifica del profilo siano errati, *editProfile* chiamerà la callback passandogli la descrizione dell'errore come parametro che il router andrà a gestire.

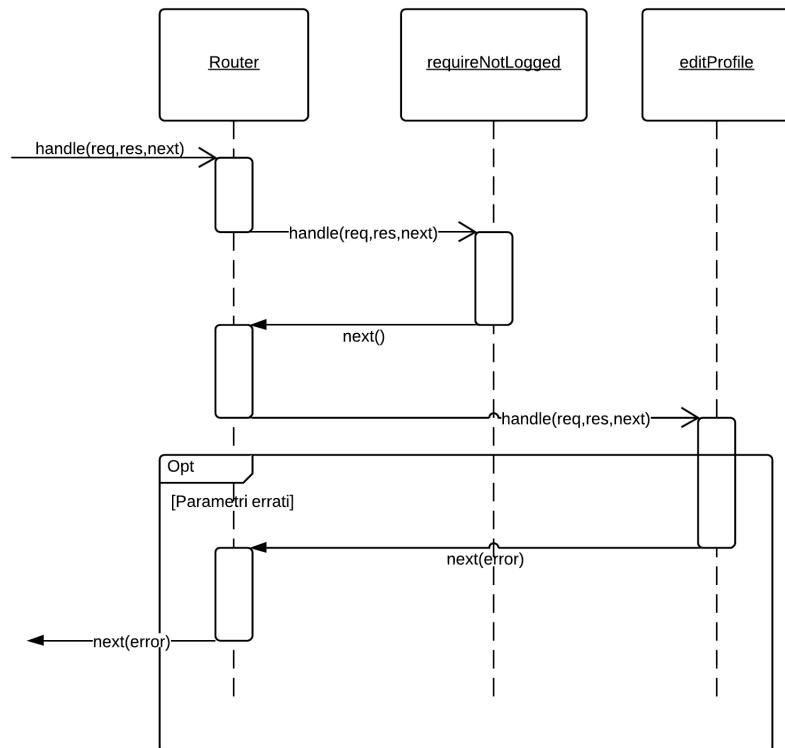


Figura 21: Richiesta PUT /profile



4.3.9 Richiesta POST /password/forgot

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

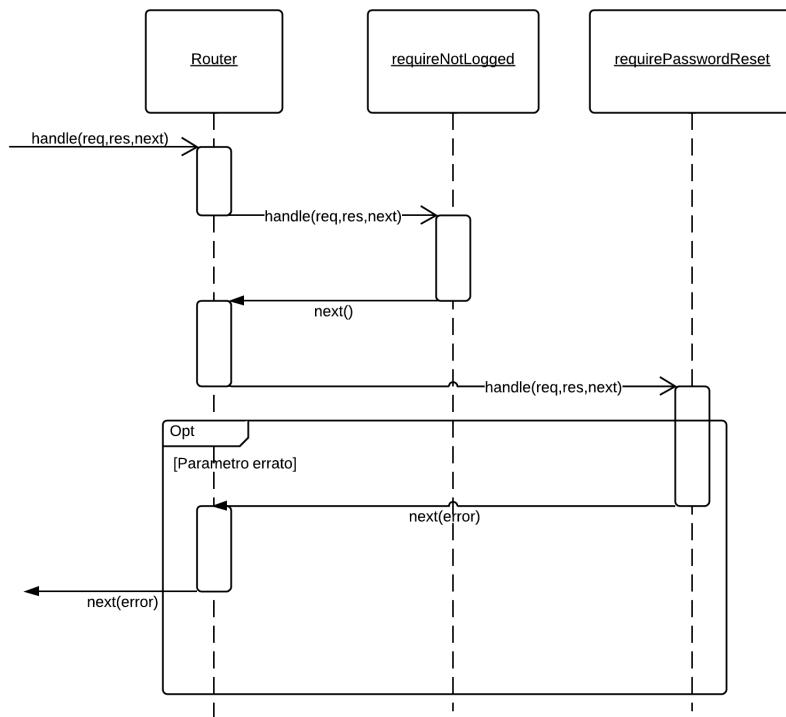


Figura 22: Richiesta POST /password/forgot



4.3.10 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 *getUsers* che gestisce la richiesta.

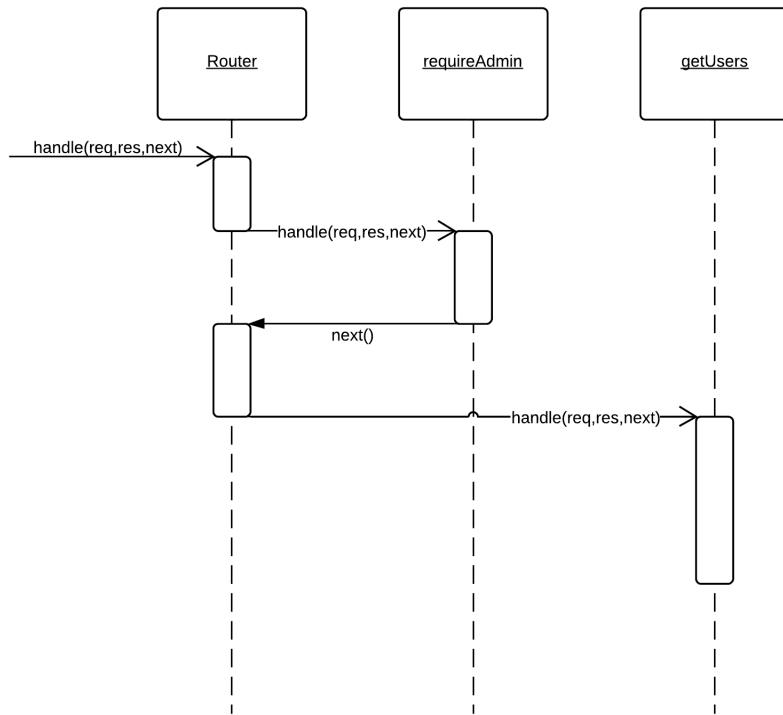


Figura 23: Richiesta GET /users



4.3.11 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 *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 controllore chiamerà la callback con l'errore.

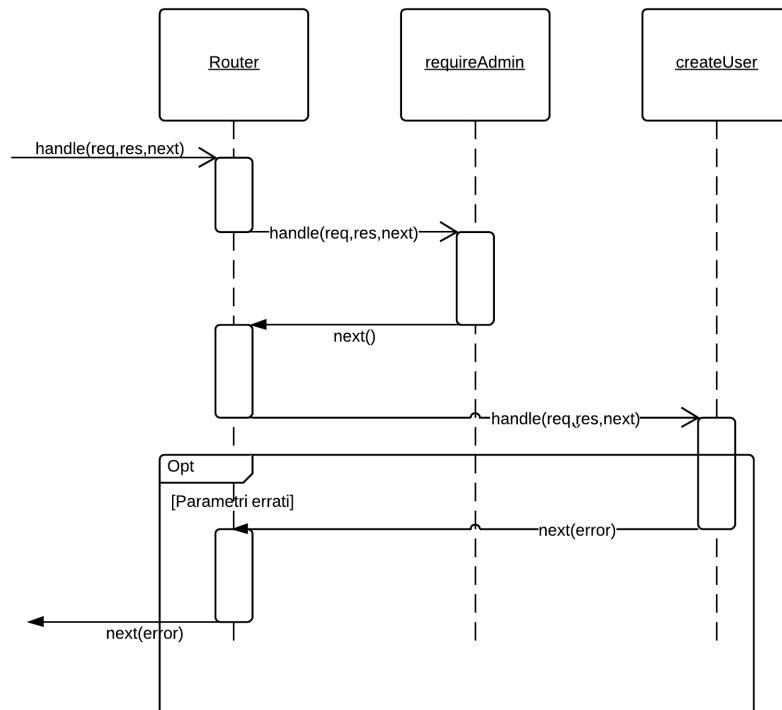


Figura 24: Richiesta POST /users



4.3.12 Richiesta GET /users/{user id}

Lo scenario rappresenta una richiesta GET di una risorsa User id, vengono verificati i permessi attraverso *requireAdmin* che passa il controllo a *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).

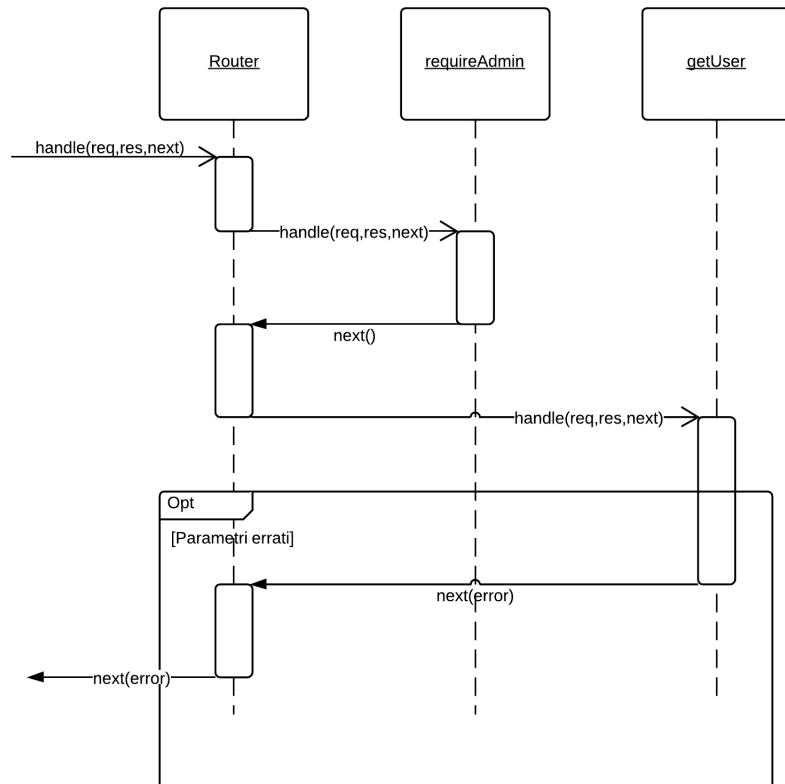


Figura 25: Richiesta GET /users/{user id}



4.3.13 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 *editUser* che gestisce la richiesta di modifica dati dell'user corrispondente all'userId che gli è stato passato come attributo. *EditUser* 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.

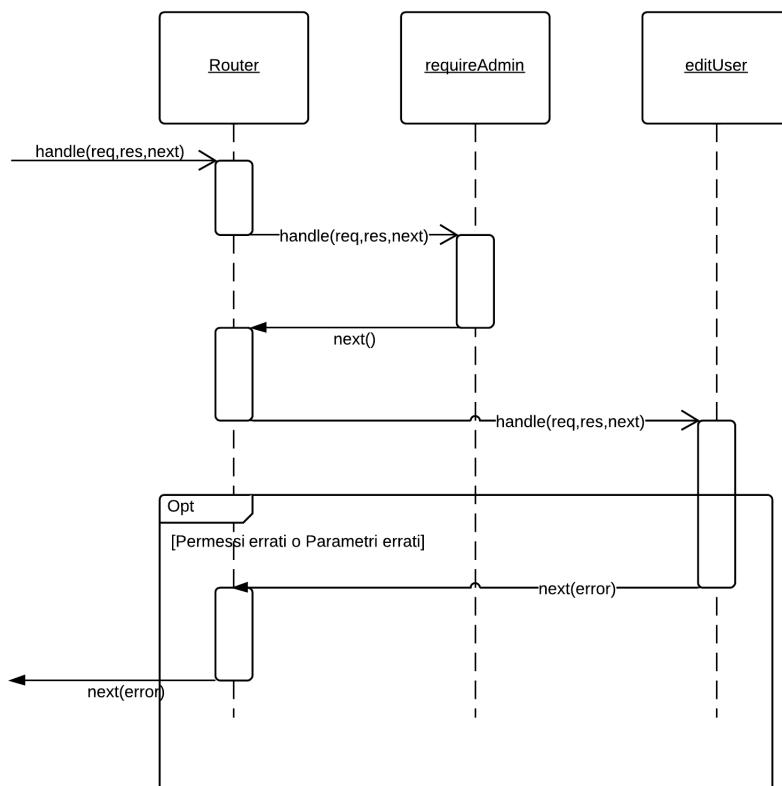


Figura 26: Richiesta PUT /users/{user id}



4.3.14 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 procedere 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, *deleteUser* restituisce un next(error). Il controller si preoccupa inoltre di verificare che i parametri siano corretti.

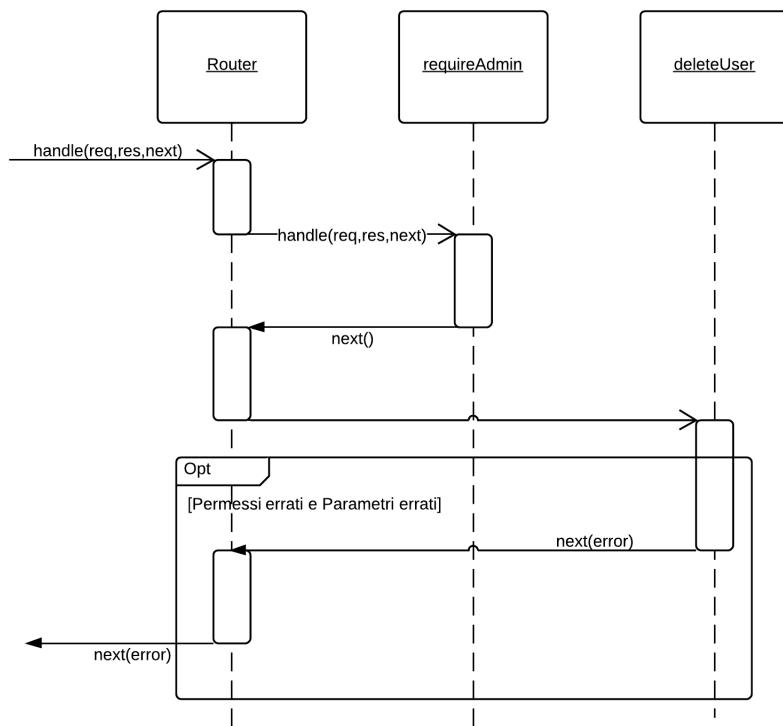


Figura 27: Richiesta DELETE /users/{user id}



4.3.15 Richiesta GET /collection

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

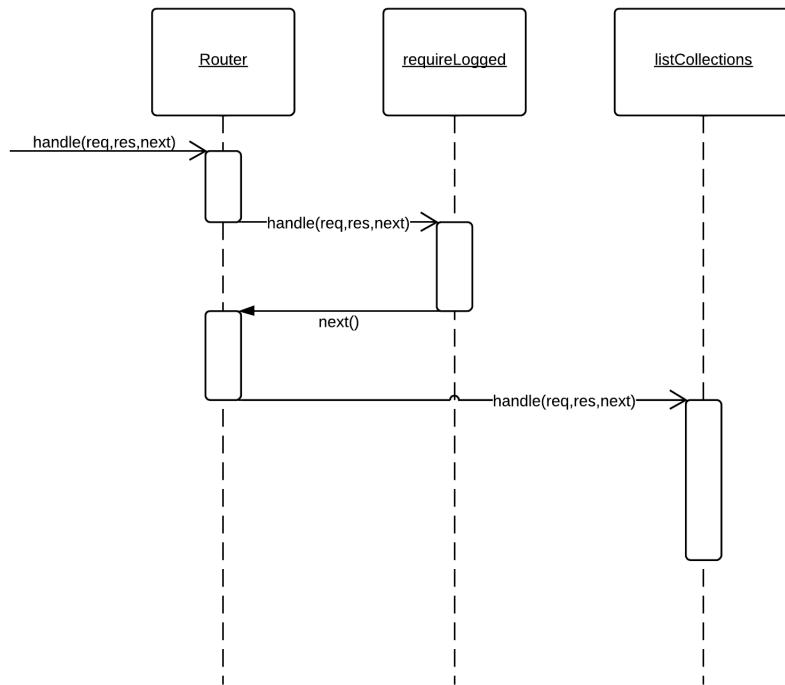


Figura 28: Richiesta GET /collection



4.3.16 Richiesta GET /collection/{collection name}

Il diagramma seguente rappresenta lo scenario di una richiesta GET per la risorsa collection Name, il controller *requireLogged* innescherà la chiamata del successivo controller *indexPage* 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 controller chiamerà la callback passandogli la descrizione dell'errore come parametro.

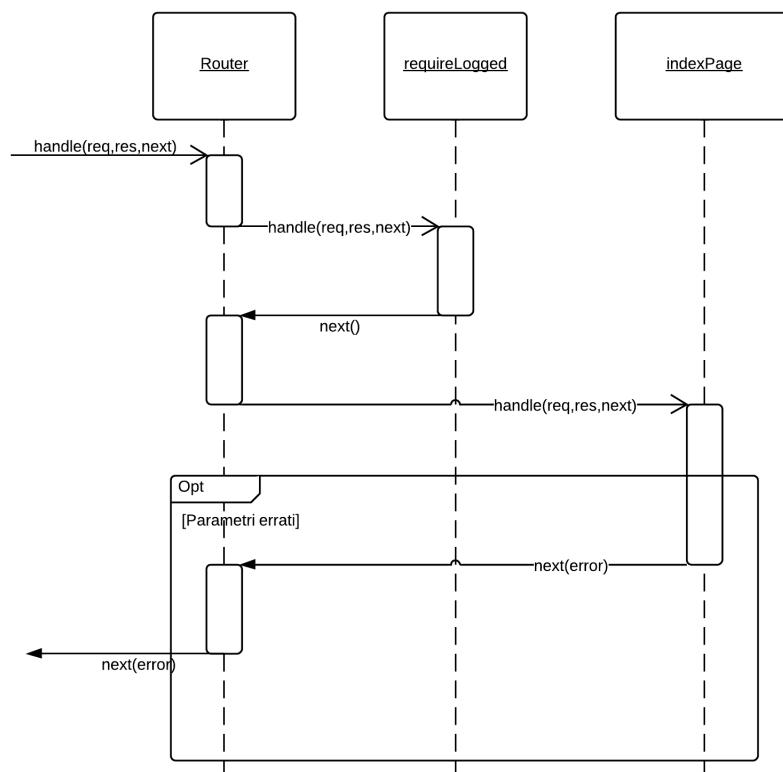


Figura 29: Richiesta GET /collection/{collection name}



4.3.17 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 controller *showpage* 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, *showpage* restituisce una next(error).

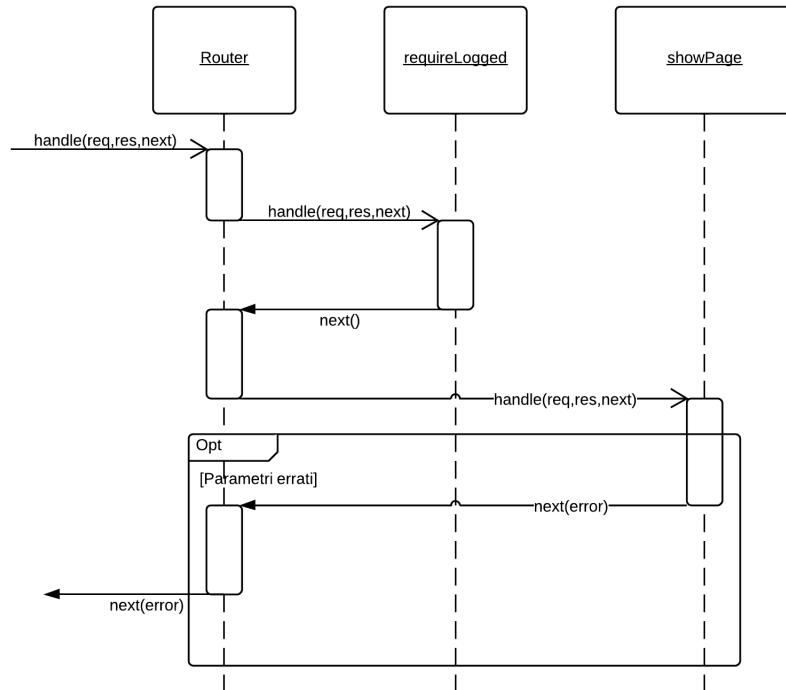


Figura 30: Richiesta GET /collection/{collection name}/{document id}



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

Il seguente diagramma rappresenta lo scenario di una richiesta POST su una risorsa collection name document, il controller *requireAdmin* non risponde con errore e viene passato il controllo a *editDocument* che gestirà la richiesta. Nell'opzione che i parametri passati per la modifica del *Document_G* siano errati, il controller chiamerà la callback passandole l'errore che il router dovrà gestire.

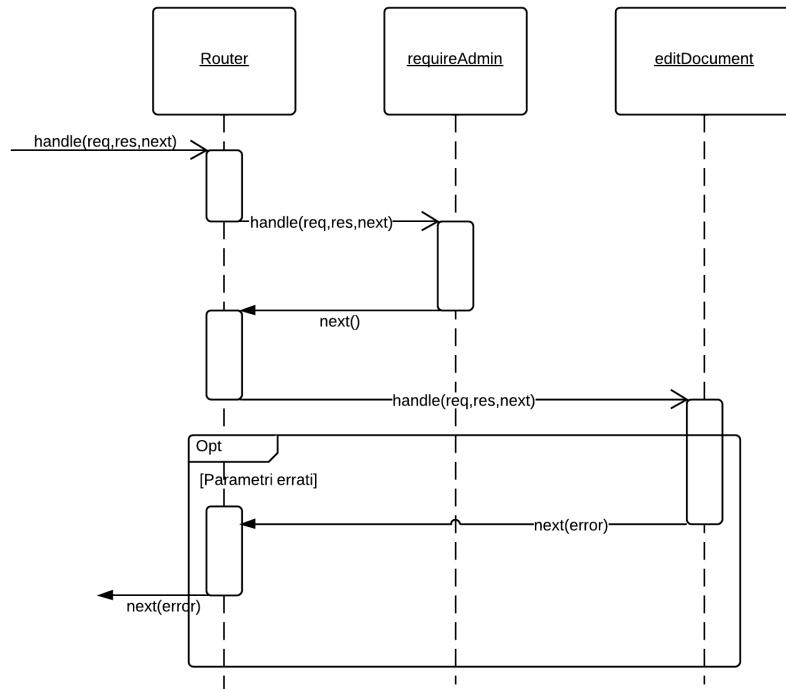


Figura 31: Richiesta PUT /collection/{collection name}/{document id}



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 *deleteDocument* 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.

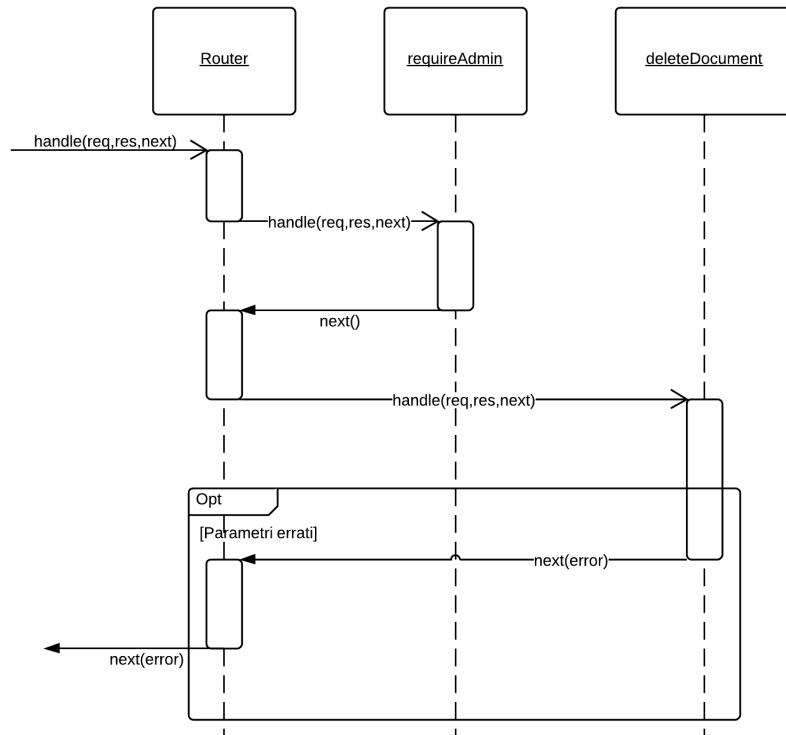


Figura 32: Richiesta DELETE /collection/{collection name}/{document id}



4.3.20 Richiesta PUT /action/{action name}/{collection name}

Il seguente diagramma di sequenza descrive lo scenario di una richiesta PUT per una risorsa Action name Collection. Il controller *collectionAction* si occupa di eseguire i controlli sugli attributi passatogli corrispondenti all'azione da eseguire e alla *Collection_G* sulla quale tale azione deve essere gestita, in caso siano errati chiamerà una next(error).

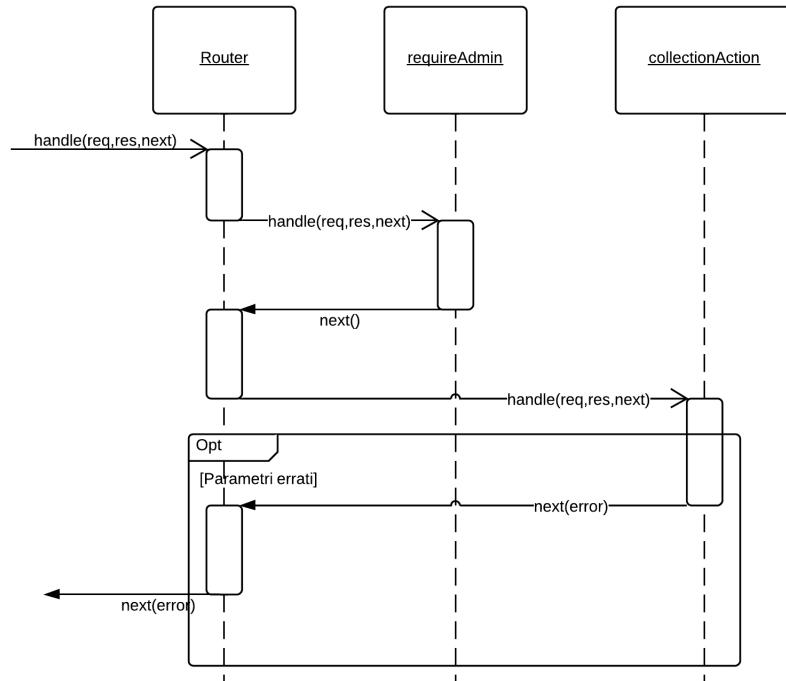


Figura 33: Richiesta PUT /action/{action name}/{collection name}



4.3.21 Richiesta PUT /action/{action name}/{collection name}/{document id}

Il diagramma a seguito mostrato descrive lo scenario di una richiesta PUT per la risorsa Action name Document. Il controller *documentAction* gestirà la richiesta di azione sul document corrispondente all'id passatogli come parametro. Nell'opzione i parametri siano errati, *documentAction* chiamerà la callback passandogli la descrizione dell'errore come parametro.

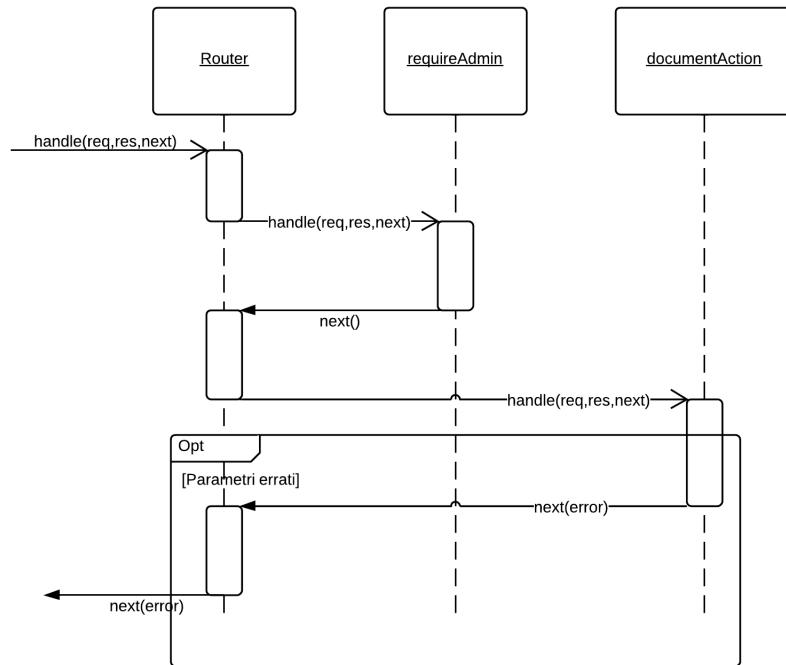


Figura 34: Richiesta PUT /action/{action name}/{collection name}/{document id}



4.4 Descrizione librerie aggiuntive

Vengono di seguito descritte le librerie aggiuntive utilizzate dal Back-end. La scelta è stata effettuata cercando di valutare la diffusione, il livello di stabilità, l'assenza di errori noti.

- **Passport:** è un middleware di autenticazione per $Node.js_G$. Estremamente flessibile e modulare, Passport può essere facilmente inserito in qualsiasi applicazione web basata su Express.
- **Passport-local:** è una libreria che permette di autenticare un utente con Passport usando un username e una password.
- **Passport-local-mongoose:** è un plugin per Mongoose che semplifica la costruzione di un sistema di autenticazione con Passport.
- **Nodemailer:** è un modulo che permette di mandare facilmente e-mail con Node.js, tramite $SMTP_G$. Supporta anche i caratteri $unicode_G$.



5 Front-end

5.1 Descrizione packages e classi

5.1.1 Front-end

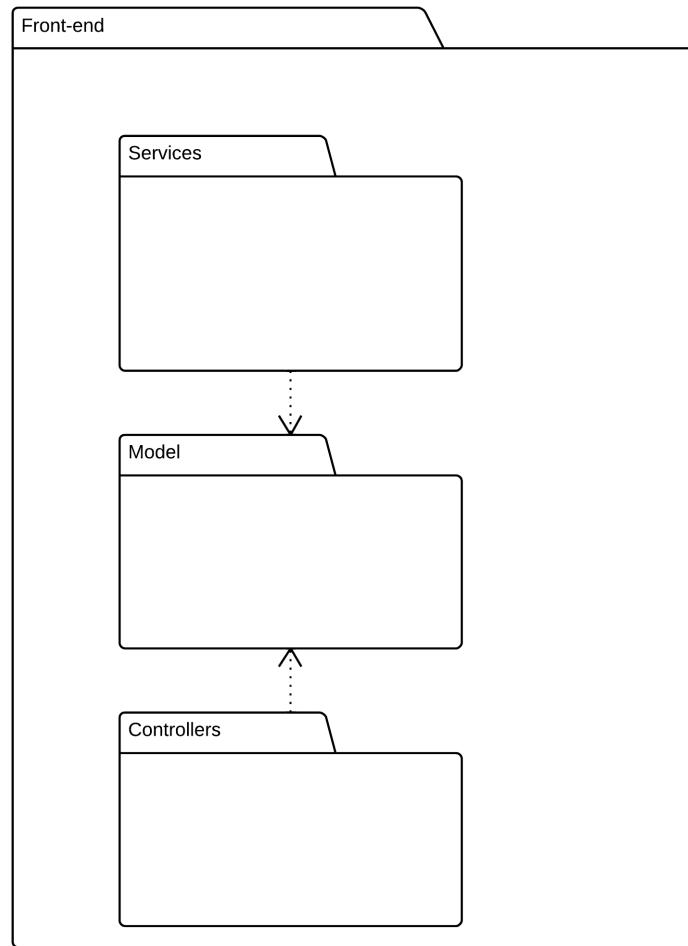


Figura 35: Diagramma dei packages Front-end

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

- Front-end::Controllers
- Front-end::Services
- Front-end::Model

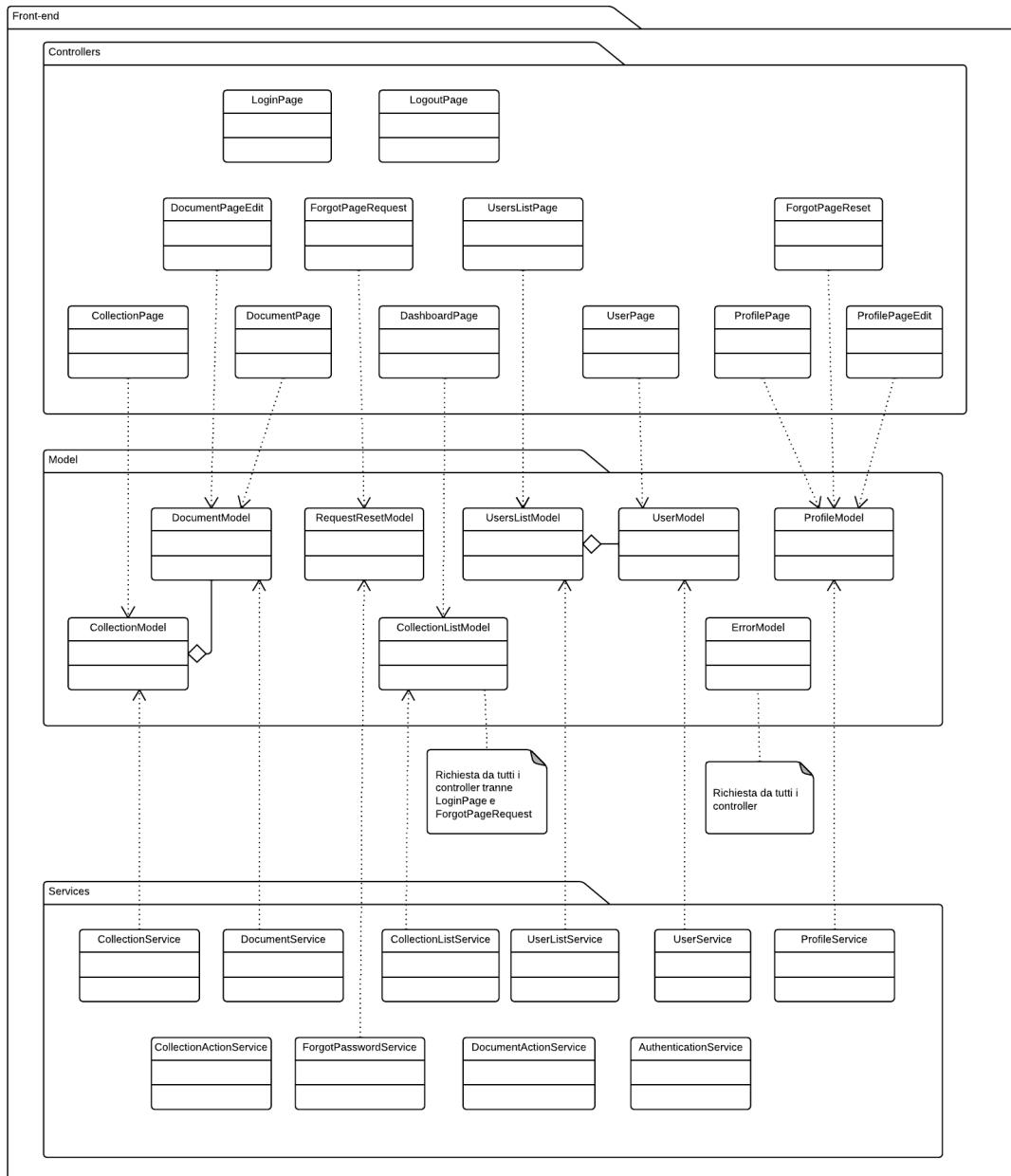


Figura 36: Diagramma delle classi Front-end

5.1.1.1.3 Diagramma delle classi Nel diagramma non sono state rappresentate le dipendenze tra *controller* e *service* in quanto sono gestite automaticamente da *Angular.js_G* utilizzando il *Design Pattern_G Dependency Injection*.



5.1.2 Front-end::Controllers

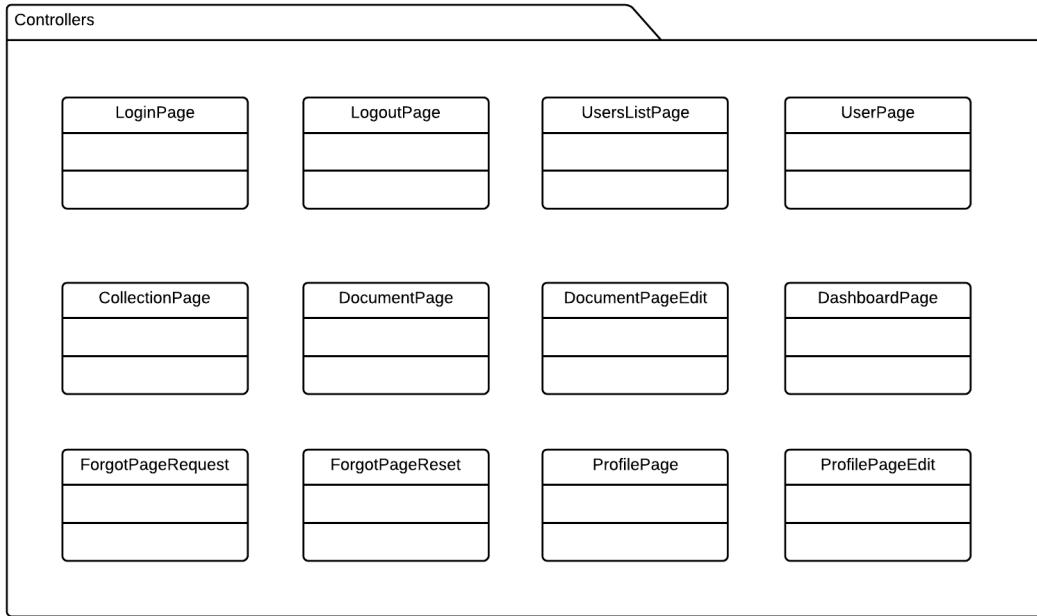


Figura 37: Componente Front-end::Controllers

5.1.2.1 Informazioni sul package

5.1.2.1.1 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.

5.1.2.2 Classi

5.1.2.2.1 Front-end::Controllers::ForgotPageReset

Descrizione

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

Utilizzo

Si occupa di prelevare la nuova password inserita dall'utente nella view e gestirne la modifica.



5.1.2.2.2 Front-end::Controllers::LoginPage

Descrizione

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

Utilizzo

Viene utilizzata per gestire il controllo sull'autenticazione di un utente.

5.1.2.2.3 Front-end::Controllers::CollectionPage

Descrizione

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

Utilizzo

Utilizza la CollectionService per popolare correttamente lo scope con la lista di tutti i Document presenti nella Collection, e la CollectionActionService per gestire le azioni personalizzate sulla Collection.

5.1.2.2.4 Front-end::Controllers::UsersListPage

Descrizione

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

Utilizzo

Utilizza la UserListService per popolare correttamente lo scope fornendo quindi alla view la lista degli utenti.

5.1.2.2.5 Front-end::Controllers::DashboardPage

Descrizione

Classe che gestisce le operazioni e la logica applicativa riguardante la pagina di gestione delle Collections.

Utilizzo

Utilizza la CollectionListService per popolare correttamente lo scope fornendo quindi alla view la lista di tutte le Collections registrate.

5.1.2.2.6 Front-end::Controllers::LogoutPage

Descrizione

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



Utilizzo

5.1.2.2.7 Front-end::Controllers::UserPage

Descrizione

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

Utilizzo

Utilizza lo UserService per popolare correttamente lo scope fornendo quindi alla view i dati dell'utente.

5.1.2.2.8 Front-end::Controllers::DocumentPageEdit

Descrizione

Classe che gestisce le operazioni di modifica di un Document.

Utilizzo

Si occupa di prelevare i dati inseriti dall'admin nella view e utilizzando il DocumentService provvede a gestire la modifica delle informazioni riguardanti il Document.

5.1.2.2.9 Front-end::Controllers::ProfilePageEdit

Descrizione

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

Utilizzo

Si occupa di prelevare i dati inseriti dall'utente nella view e utilizzando il ProfileService provvede a gestire la loro modifica.

5.1.2.2.10 Front-end::Controllers::DocumentPage

Descrizione

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

Utilizzo

Utilizza il DocumentService per popolare correttamente lo scope fornendo quindi alla view la lista degli attributi del Document.

Utilizza il DocumentActionService per gestire le azioni personalizzate sul Document.



5.1.2.2.11 Front-end::Controllers::ProfilePage

Descrizione

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

Utilizzo

Utilizza il ProfileService per popolare correttamente lo scope fornendo quindi alla view i dati dell'utente.

5.1.2.2.12 Front-end::Controllers::ForgotPageRequest

Descrizione

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

Utilizzo

Permette di inviare al Back-end una richiesta di recupero password attraverso il 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.

5.1.3 Front-end::Services

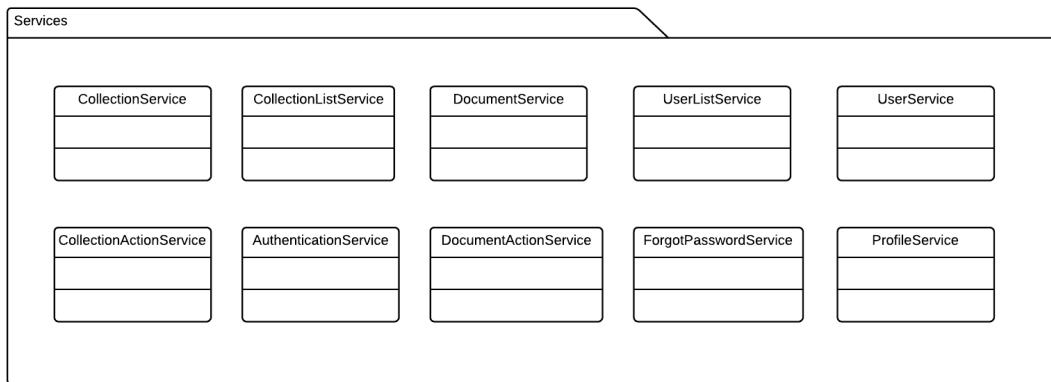


Figura 38: Componente Front-end::Services

5.1.3.1 Informazioni sul package

5.1.3.1.1 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).

5.1.3.2 Classi

5.1.3.2.1 Front-end::Services::AuthenticationService

Descrizione

Questa classe permette la creazione e l'eliminazione della sessione utente attraverso le corrispettive chiamate /login e /logout

Utilizzo

La funzionalità offerta dalla classe è quella di poter fornire al Controller lo stato della sessione utente.

5.1.3.2.2 Front-end::Services::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

5.1.3.2.3 Front-end::Services::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



- modifica dei dati utente

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

Relazioni con altre classi

- Front-end::Model::ProfileModel

5.1.3.2.4 Front-end::Services::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

5.1.3.2.5 Front-end::Services::DocumentActionService

Descrizione

Questa classe permette l'esecuzione (nel backend) delle azioni personalizzate di un Document tramite la chiamata /action/{action name}/{collection name}/{document id}

Utilizzo

La funzionalità offerta dalla classe è quella di poter fornire al Controller il risultato dell'azione personalizzata definita dallo sviluppatore su un Document.

5.1.3.2.6 Front-end::Services::CollectionActionService

Descrizione

Questa classe permette l'esecuzione (nel backend) delle azioni personalizzate di una Collection tramite la chiamata /action/{action name}/{collection name}

Utilizzo

La funzionalità offerta dalla classe è quella di poter fornire al Controller il risultato dell'azione personalizzata definita dallo sviluppatore su una Collection.

5.1.3.2.7 Front-end::Services::CollectionService

Descrizione



Questa classe permette il recupero della risorsa REST rappresentante la Collection tramite la chiamata /collection/{collection_name}

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

5.1.3.2.8 Front-end::Services::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

5.1.3.2.9 Front-end::Services::DocumentService

Descrizione

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

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.

Relazioni con altre classi

- Front-end::Model::DocumentModel

5.1.3.2.10 Front-end::Services::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

5.1.4 Front-end::Model

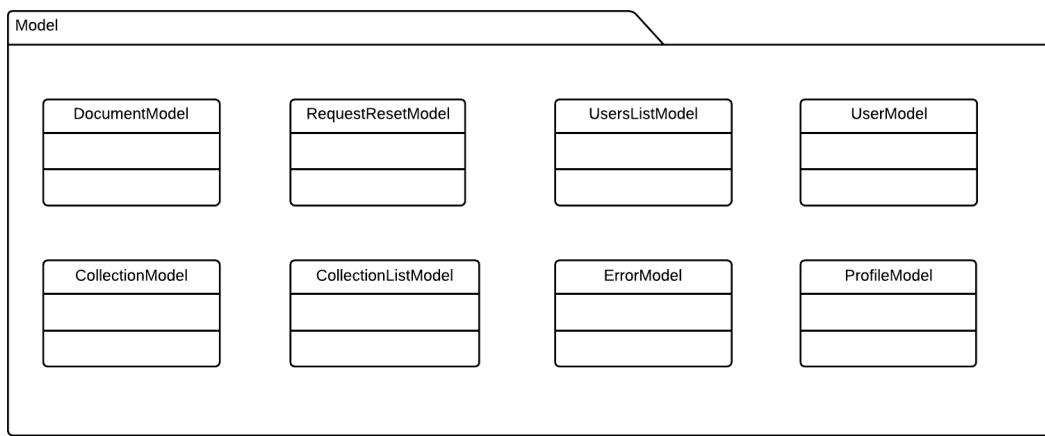


Figura 39: Componente Front-end::Model

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 Front-end::Model::ErrorModel

Descrizione

E 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 Front-end::Model::CollectionListModel

Descrizione

E 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 DashboardPage Controller di poter accedere alla lista delle Collections.

5.1.4.2.3 Front-end::Model::RequestResetModel

Descrizione

E il modello che contiene 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 ForgotPageRequest Controller di poter accedere ai dati dell'utente.

5.1.4.2.4 Front-end::Model::UsersListModel

Descrizione

E 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 UserListPage Controller di poter accedere alla lista degli utenti.

5.1.4.2.5 Front-end::Model::UserModel

Descrizione

E 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 UserPage Controller di poter accedere agli attributi dell'utente.



5.1.4.2.6 Front-end::Model::DocumentModel

Descrizione

E 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 DocumentPage Controller di poter accedere agli attributi del Document.

5.1.4.2.7 Front-end::Model::CollectionModel

Descrizione

E 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 alla CollectionService e alla CollectionPage Controller di poter accedere alla lista delle Collections.

5.1.4.2.8 Front-end::Model::ProfileModel

Descrizione

E 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 ProfilePage Controller e al ProfileEditController per ottenere il dati dell'utente da visualizzare nella view della pagina profilo e al ForgotPageReset Controller per la modifica della password.



6 Diagrammi di attività

Vengono in seguito illustrati i diagrammi di attività prodotti durante la progettazione architettonica, 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

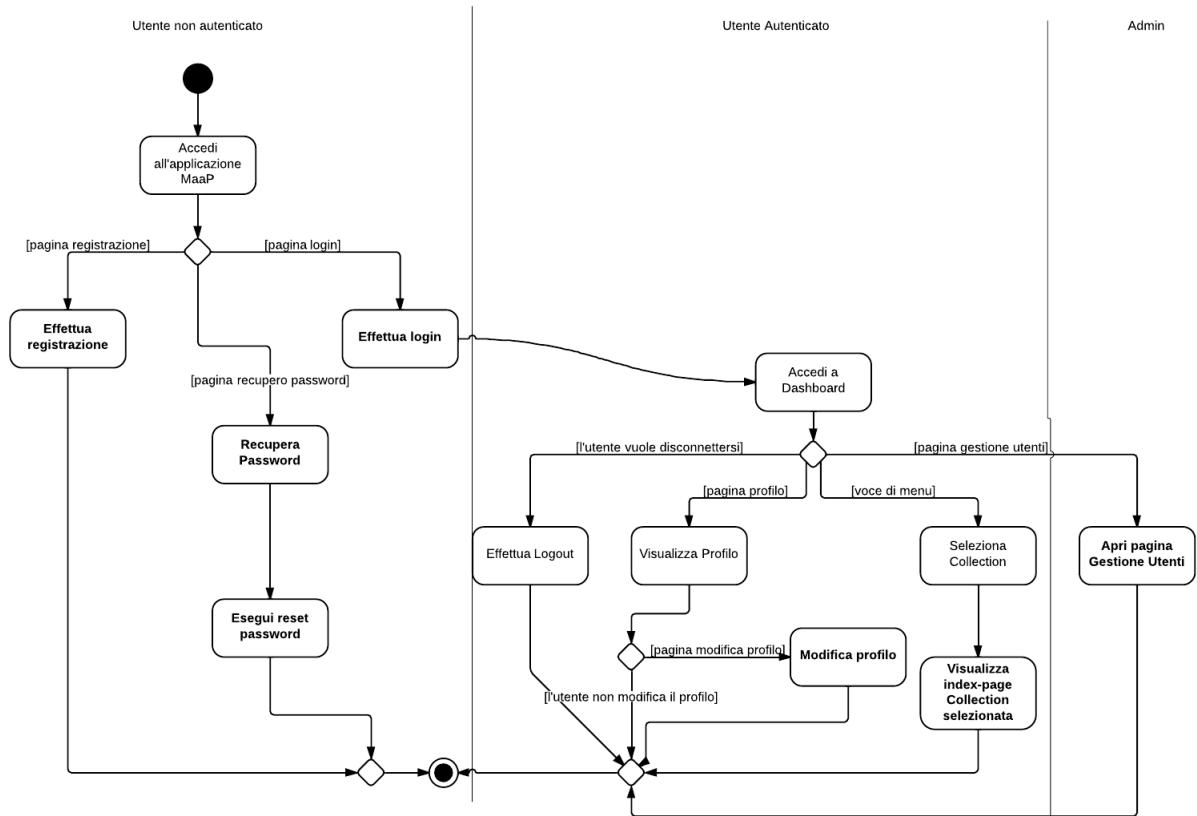


Figura 40: Diagramma di attività - Attività principali di un'applicazione MaaP

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

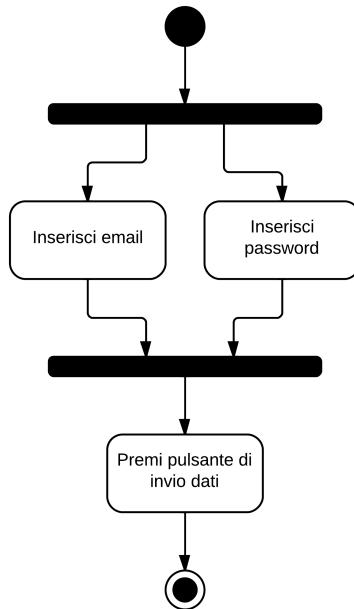


Figura 41: Diagramma di attività - Registrazione di un utente

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

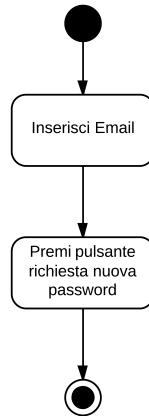


Figura 42: Diagramma di attività - Recupero password

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

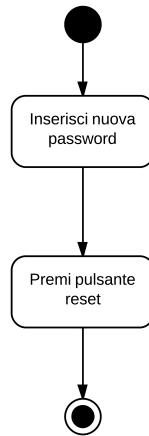


Figura 43: Diagramma di attività - Reset della password dell'utente



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

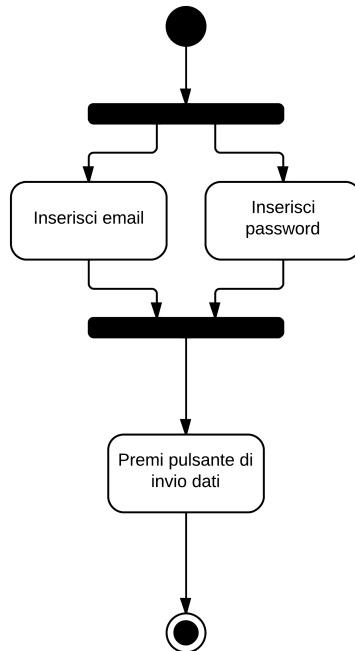


Figura 44: Diagramma di attività - Login dell'utente

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

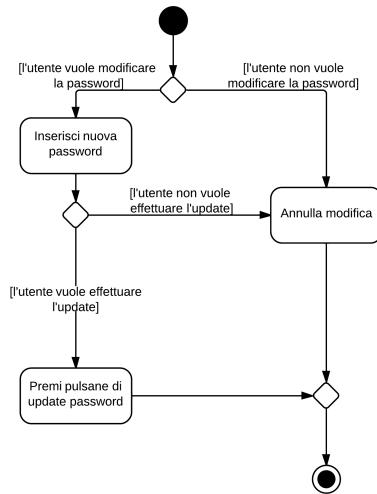


Figura 45: Diagramma di attività - Modifica profilo utente

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

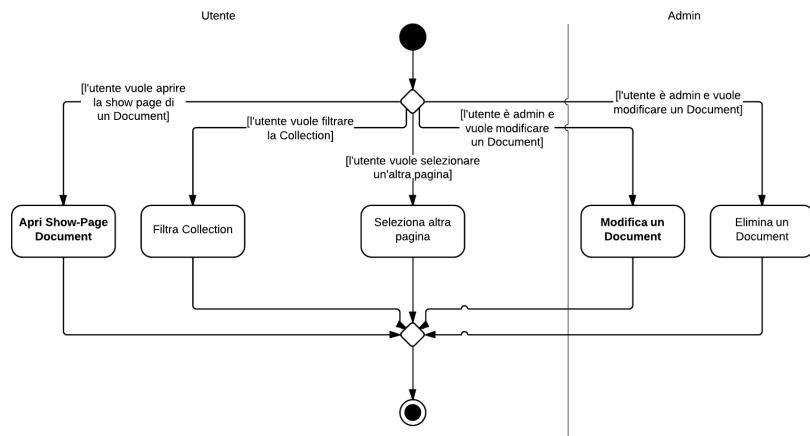


Figura 46: Diagramma di attività - Visualizzazione index-page della Collection selezionata

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 $show-page_G$ 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

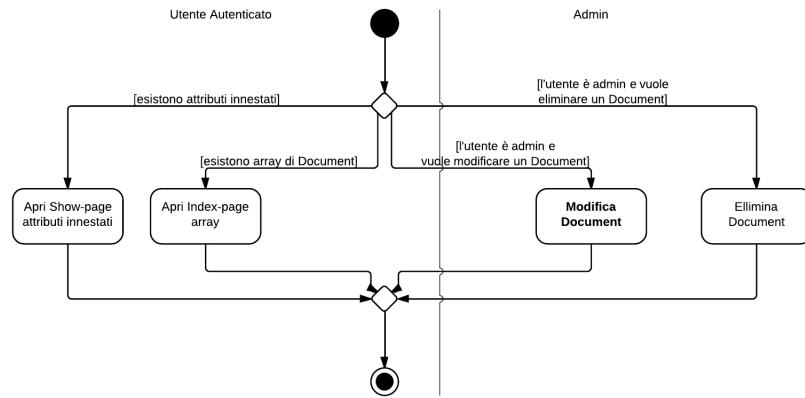


Figura 47: Diagramma di attività - Visualizzazione show-page del Document selezionato

L'utente ha selezionato un $Document_G$ dalla $index-page_G$ 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 $show-page_G$ di un attributo innestato, se ne esiste uno;
- Aprire la $index-page_G$ 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 Modifica Document

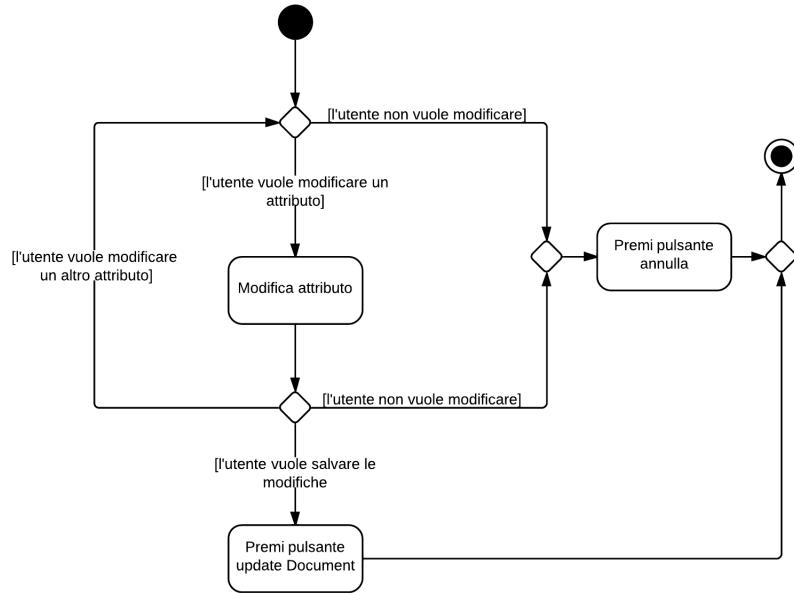


Figura 48: Diagramma di attività - Modifica del Document selezionato

L'utente con privilegi di scrittura può modificare ogni singolo attributo del $Document_G$ selezionato, editando i rispettivi campi di testo. Una volta che ha terminato le modifiche preme il pulsante di *update*; il sistema $MaaP_G$ verificherà che le modifiche apportate siano corrette e rispettino i vincoli del $database_G$. In ogni momento l'utente può comunque decidere di annullare le modifiche e tornare alla pagina precedente.



6.1.10 Apri pagina gestione utenti

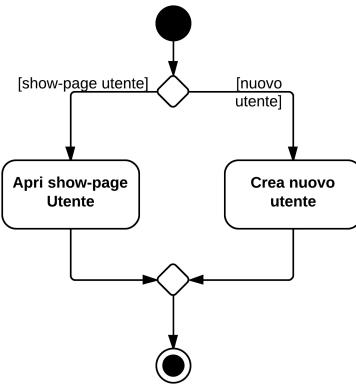


Figura 49: Diagramma di attività - Pagina di gestione degli utenti

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.



6.1.11 Apri show-page utente

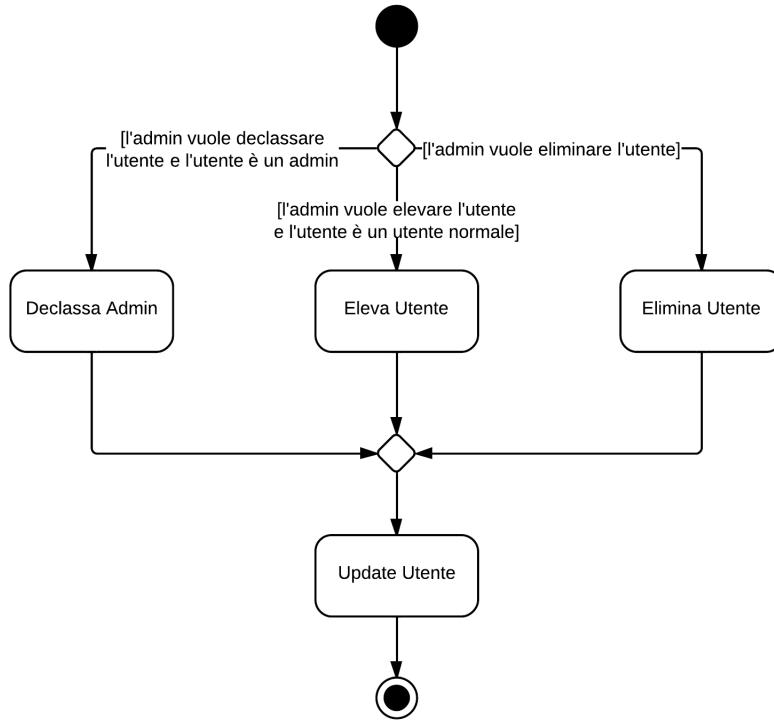


Figura 50: Diagramma di attività - Pagina di visualizzazione di un utente

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 occuperà di apportare tutte le modifiche effettuate dall'admin al $database_G$ delle credenziali.



6.1.12 Crea un nuovo utente

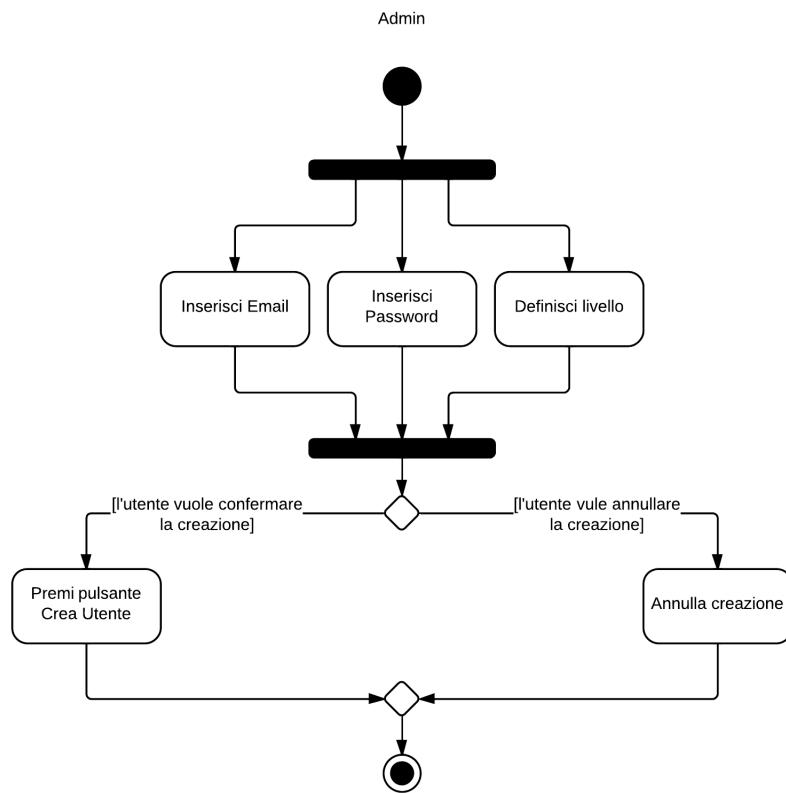


Figura 51: Diagramma di attività - Pagina di creazione di un nuovo utente

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

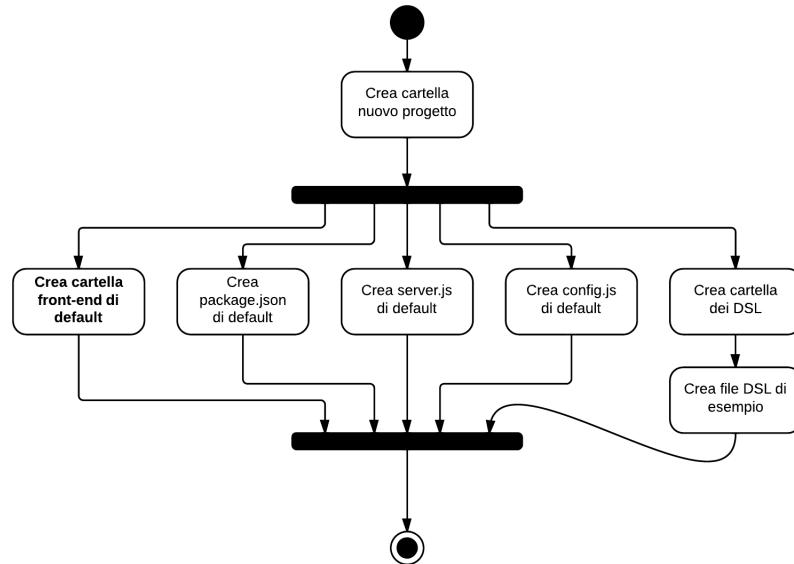


Figura 52: Diagramma di attività - Creazione scheletro nuova applicazione

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-end*;
- 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

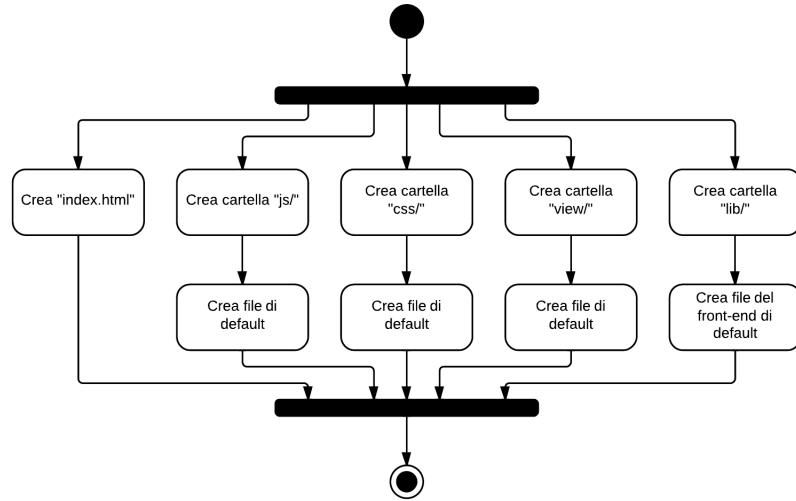


Figura 53: Diagramma di attività - Creazione scheletro nuova applicazione

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 approfondate 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 la soluzione 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.

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 tra componenti di 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.

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*.



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.
- **Utilizzo:** Viene utilizzato da *express_G*, è fortemente legato a 8.4.1. In particolare è utilizzato all'interno del *package_G Back-End::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*.
- **Utilizzo:** Le diverse *Collection_G* presenti nei file *DSL_G* verranno memorizzate per nome in un registro, in modo tale che quando arriva una richiesta venga caricato il file *DSL_G* giusto. In particolare è utilizzato all'interno del *package_G Back-End::DSLModel*.

8.2.2 Factory method

- **Scopo:** Nel contesto di *Node.js_G* questo pattern viene usato creare una classe attraverso una *funzione factory* esportata dal modulo. In questo modo si potrà costruire e ottenere qualsiasi classe definita in un modulo.
- **Utilizzo:** È utilizzato all'interno del *package_G Back-End::Controller*.

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 si riferiscono allo stesso. In architettura è utilizzato per la classe *Back-End::Controller::ControllerFactory*.

8.3 Design Pattern Strutturali

8.3.1 Facade

- **Scopo:** Viene utilizzato per rendere visibili solamente alcune cose agli altri oggetti.
- **Utilizzo:** Viene utilizzato nei seguenti casi:
 - La classe *Back-End::Middleware::MiddlewareLoader* utilizza *facade* per nascondere l'esistenza di tutti i *middleware_G* alla *ServerApp*;
 - La classe *Back-End::Controller::ControllerFactory* utilizza *facade* per nascondere le classi interne.



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.
- **Utilizzo:** *Express_G* usa chain-of-responsability per la gestione dei *middleware_G* e del *routing_G*. In particolare viene utilizzato nell'architettura nel *package_G Back-End::Middleware*.

8.4.2 Strategy

- **Scopo:** In futuro potrebbe essere necessario cambiare l'algoritmo di interpretazione del *DSL_G*. Utilizzando questo pattern l'algoritmo potrà essere cambiato in modo indipendente da chi ne fa uso.
- **Utilizzo:** Viene utilizzato all'interno del *package_G Back-End::DSLModel*.

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 RA10 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::AuthModel	RF1O 8.2 RA1D 12
Back-end::Lib::Controller	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::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::MailView	
Back-end::Lib::Middleware	RA1O 1.3 RA1O 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::Middleware
RA10 2	Front-end::Services
RA10 2.1	Front-end::Services
RA10 2.2	Back-end::Lib::Middleware
RA10 2.3	Front-end::Controllers Front-end::Services Back-end::Lib::Controller
RA1D 3	Front-end::Controllers
RA1O 4	Front-end::Controllers Front-end::Services
RA1O 4.1	Front-end::Controllers Back-end::Lib::DSLModel
RA1O 4.1.1	Front-end::Controllers Back-end::Lib::DSLModel
RA1O 4.1.2	Front-end::Controllers Back-end::Lib::DSLModel
RA1O 4.1.3	Front-end::Controllers Back-end::Lib::DSLModel
RA1O 5	Front-end::Controllers Front-end::Services Back-end::Lib::Controller Back-end::Lib::DSLModel
RA1O 5.1	Front-end::Controllers Front-end::Services Back-end::Lib::Controller Back-end::Lib::DSLModel



RA1O 5.3	Front-end::Controllers Front-end::Services Back-end::Lib::Controller Back-end::Lib::DSLModel
RA1O 6	Front-end::Controllers Front-end::Services Back-end::Lib::Controller Back-end::Lib::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 Back-end::Lib::DSLModel
RA1O 6.1.3	Back-end::Lib::Middleware
RA1O 6.2	Front-end::Controllers
RF1O 7	Back-end::Lib::DSLModel
RF1O 8	Back-end::Lib::DSLModel Back-end::Lib::Middleware
RF1O 8 .1	Back-end::Lib::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::Lib::AuthModel
RF1F 8.3	Back-end::DeveloperProject
RF1O 9	Back-end::Lib::DSLModel
RF1O 9.1	Back-end::Lib::DSLModel
RF1O 9.2	Back-end::Lib::DSLModel
RF1O 9.3	Back-end::Lib::DSLModel



RF1O 9.4	Back-end::Lib::DSLModel
RA1D 11	Front-end::Controllers
RA1D 12	Front-end::Controllers Back-end::Lib::AuthModel
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::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.

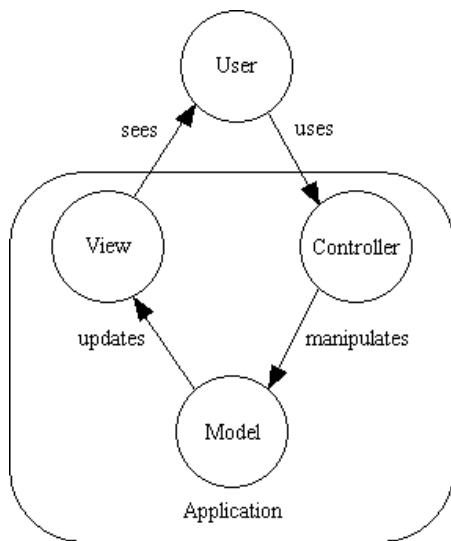


Figura 54: Struttura logica di Model-View-Controller



A.1.2 Middleware

Il $Middleware_G$ è uno strato software che si interpone tra l'applicazione software e il sistema operativo per semplificare 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 permettono 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' *enterprise 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 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.

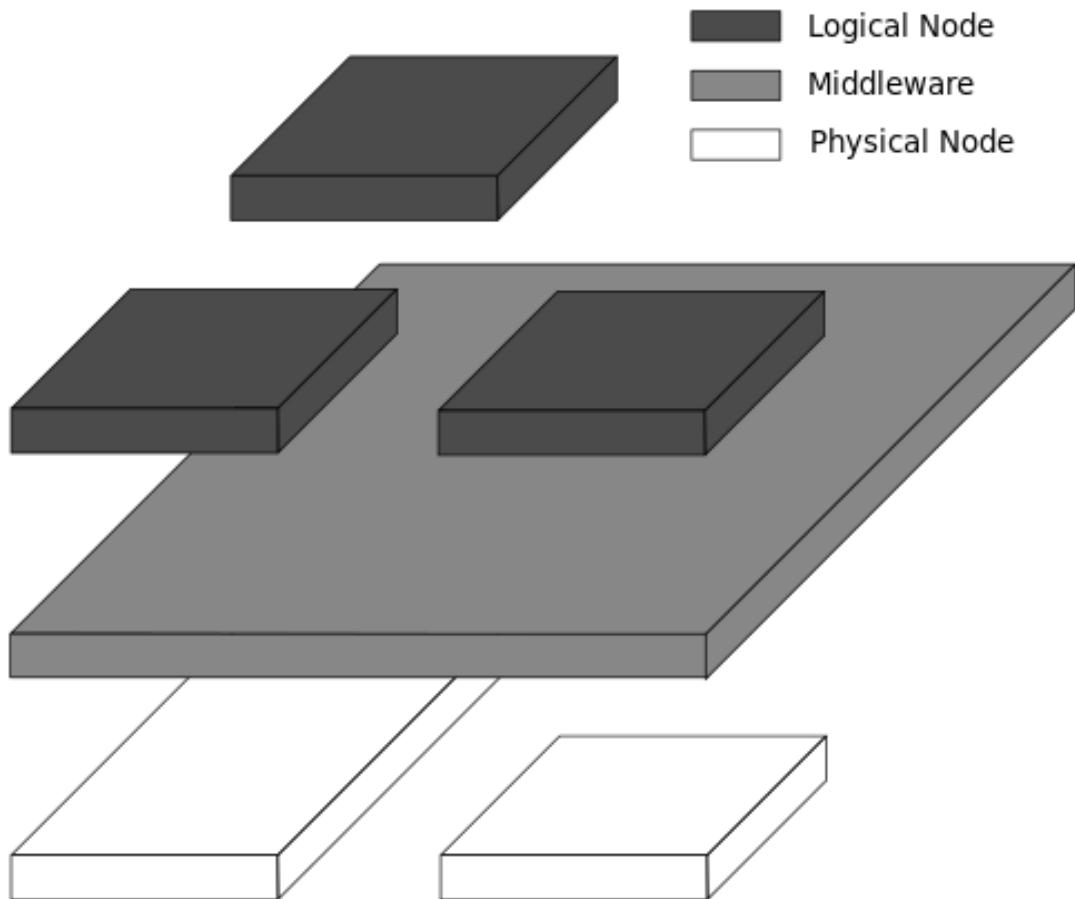


Figura 55: Struttura logica di Middleware

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.

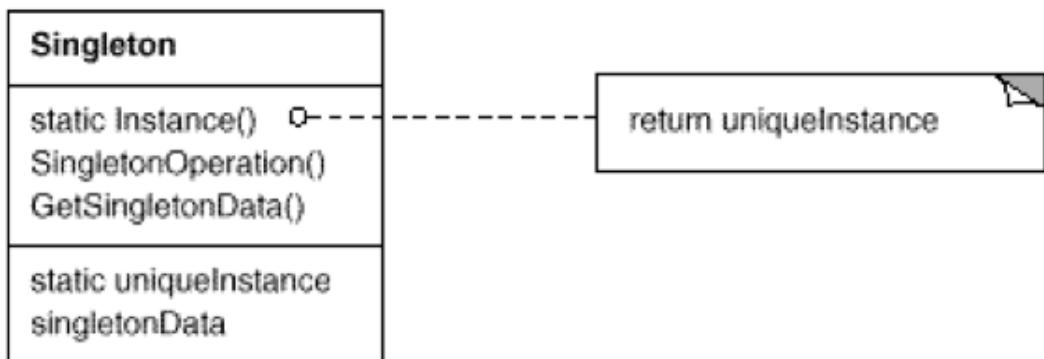


Figura 56: Struttura logica di singleton

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.

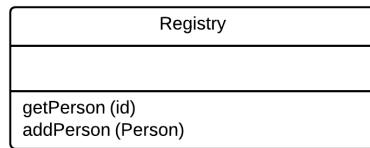


Figura 57: Struttura logica di registry



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.

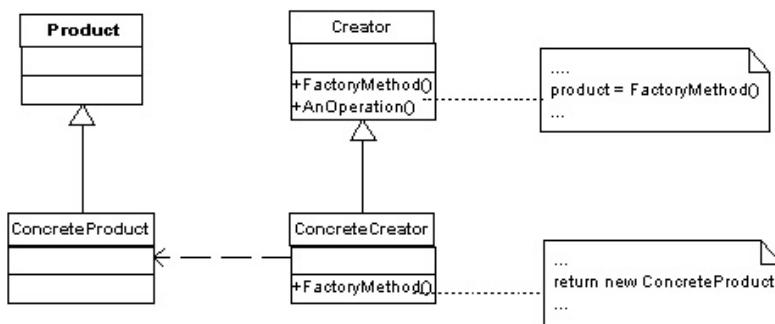


Figura 58: Struttura logica di factory method

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*.

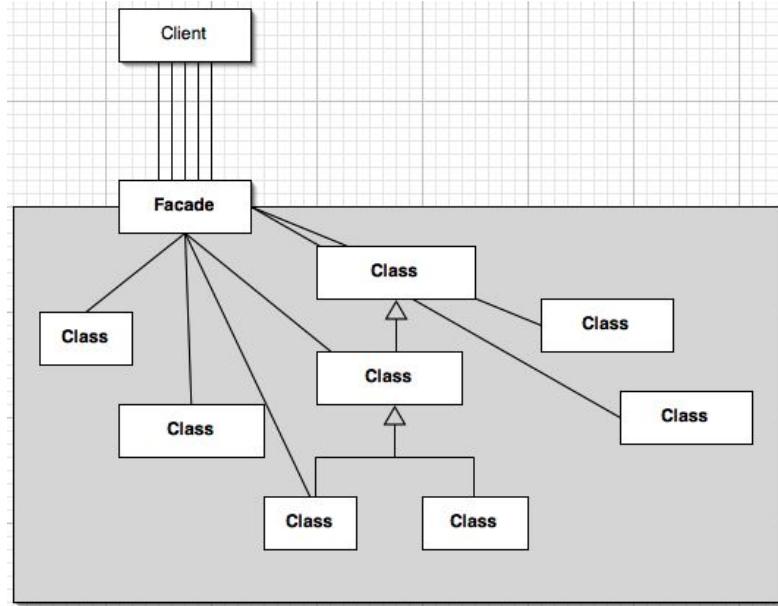


Figura 59: Struttura logica di facade

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 60.

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 possibile 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.

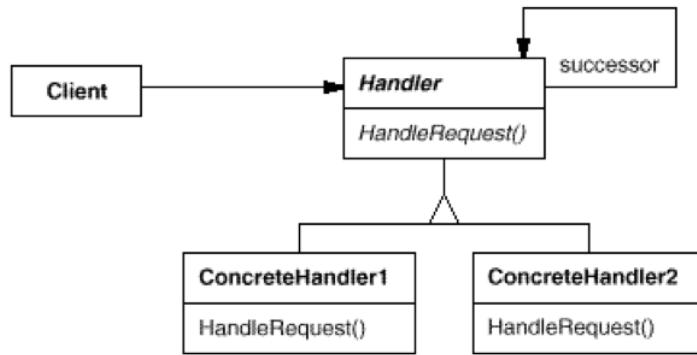


Figura 60: Struttura del chain of responsibility.

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.

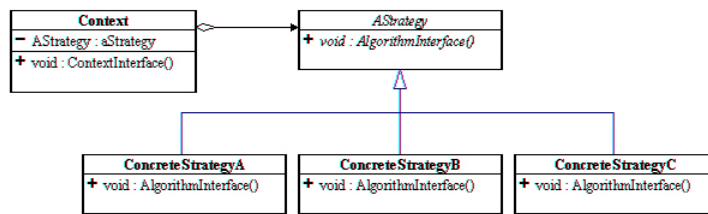


Figura 61: Struttura del chain of responsibility.

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 implementazioni 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.

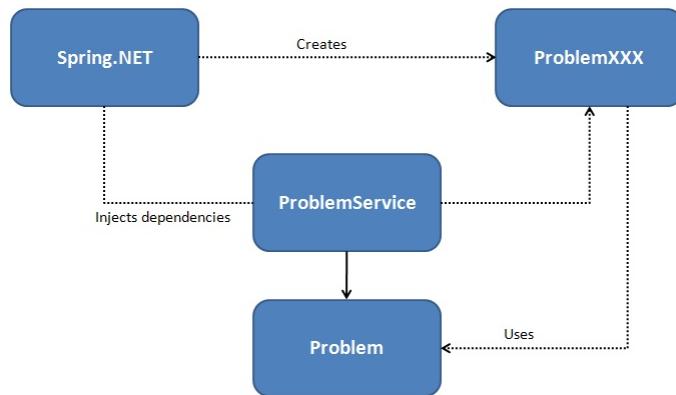


Figura 62: Struttura logica di Dependency Injection

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.

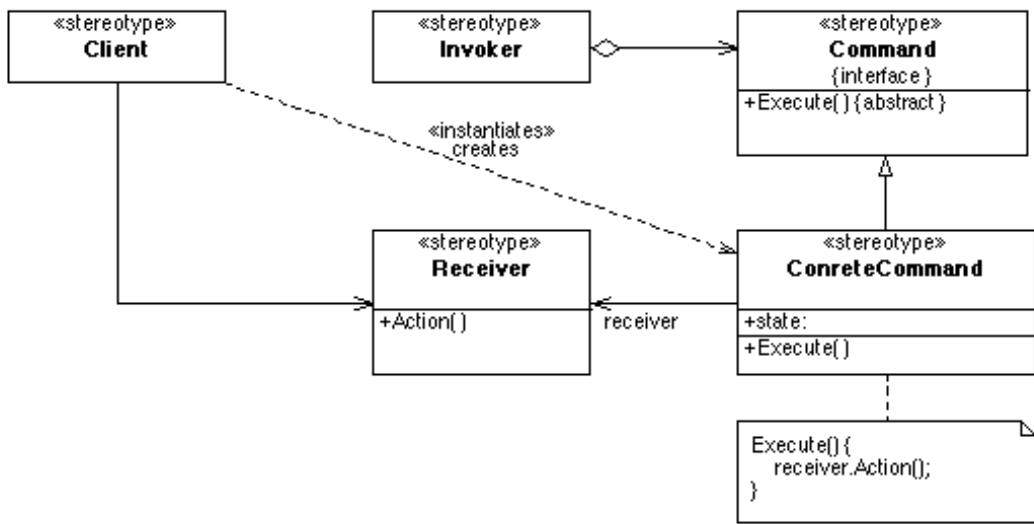


Figura 63: Struttura logica di Command