



# Specifiche Tecniche

## Informazioni sul documento

<b>Titolo documento</b>	Specifiche Tecniche
<b>Versione attuale</b>	v2.0.0
<b>Data versione attuale</b>	2012/03/18
<b>Data creazione</b>	2012/01/17
<b>Redazione</b>	Andrea Zironda Luca Guerra Antonio Pretto Luca Lorenzini Umberto Dall'Est
<b>Revisione</b>	Antonio Pretto [v2.0.0] Umberto Dall'Est [v2.0.0] Giacomo Lorigiola [v1.0.0]
<b>Approvazione</b>	Luca Guerra
<b>Stato documento</b>	Formale
<b>Uso</b>	Esterno
<b>Distribuito da</b>	SevenFold
<b>Destinato a</b>	Prof. Tullio Vardanega Dott. Amir Baldissera referente Mentis s.r.l. Dott.ssa Elisa Sartore referente Mentis s.r.l.

## Sommario

Il presente documento ha lo scopo di specificare quella che dovrà essere l'architettura generale del sistema PMAC.

## Diario delle modifiche

Versione	Data	Autore	Modifiche
v2.0.0	2012/03/18	Luca guerra	Approvazione e rilascio seconda versione
v1.7.0	2012/03/16	Antonio Pretto	Ultime modifiche e correzioni errori.
v1.6.0	2012/03/12	Andrea Zironda	Nuova divisione contenuti sottosistema Desktop.
v1.5.1	2012/03/01	Luca Guerra	Cambiamento pattern sottosistema mobile, da MVC a MVP.
v1.5.0	2012/02/28	Luca Guerra	Nuova descrizione sottosistema Mobile
v1.4.0	2012/02/26	Umberto Dall'Est	Correzioni immagini sottosistema Mobile
v1.3.0	2012/02/21	Antonio Pretto	Inserimento immagini sottosistema Server
v1.2.0	2012/02/16	Andrea Zironda	Correzioni sottosistema Server
v1.1.0	2012/02/11	Luca Guerra	Rivisitazione architettura Mobile
v1.0.0	2012/01/31	Stefano Faoro	Approvazione e rilascio prima versione
v0.13.0	2012/01/30	Andrea Zironda	Correzione ortografica generale
v0.12.1	2012/01/30	Antonio Pretto	Correzione diagrammi Sottosistemi Server
v0.12.0	2012/01/29	Luca Lorenzini	Aggiornato diagramma attività mobile
v0.11.0	2012/01/28	Andrea Zironda	Avanzamento Sottosistema Server
v0.10.0	2012/01/27	Luca Lorenzini	Correzione diagrammi Sottosistema Desktop
v0.9.1	2012/01/26	Luca Guerra	Inizio stesura diagrammi di sequenza Sottosistema Mobile
v0.9.0	2012/01/25	Luca Guerra	Creazione capitolo Sottosistema Mobile e prima stesura
v0.8.0	2012/01/24	Andrea Zironda	Creazione capitolo Sottosistema Desktop ed inserimento diagrammi
v0.7.0	2012/01/23	Antonio Pretto	Prima trattazione Sottosistema Server
v0.6.0	2012/01/22	Luca Lorenzini	Correzioni ortografiche varie
v0.5.0	2012/01/22	Luca Guerra	Inizio stesura tecnologie utilizzate
v0.4.0	2012/01/20	Antonio Pretto	Inizio stesura diagrammi server
v0.3.0	2012/01/18	Andrea Zironda	Inizio stesura design patterns
v0.2.0	2012/01/17	Luca Guerra	Creazione capitolo Definizione del Prodotto e inizio trattazione
v0.1.0	2012/01/15	Luca Lorenzini	Creazione documento

## Indice

<b>1 Introduzione</b>	<b>7</b>
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
<b>2 Definizione del prodotto</b>	<b>9</b>
2.1 Metodo e formalismo di specifica . . . . .	9
2.2 Stili architetturali - Client-Server . . . . .	9
2.2.1 Client . . . . .	9
2.2.2 Server . . . . .	9
2.2.3 Interfaccia . . . . .	9
2.2.4 Resource Oriented Architecture . . . . .	10
2.2.5 Presentation . . . . .	11
2.2.6 Logic . . . . .	11
2.2.7 Data . . . . .	12
2.3 Design patterns . . . . .	12
2.3.1 Model-View-Controller . . . . .	12
2.3.2 Active Record . . . . .	13
2.3.3 Decorator . . . . .	13
2.3.4 Observer . . . . .	13
2.3.5 Singleton . . . . .	13
2.4 Tecnologie . . . . .	14
2.4.1 Lato server . . . . .	14
2.4.2 Lato Desktop . . . . .	14
2.4.3 Lato Mobile . . . . .	15
2.5 Librerie Aggiuntive . . . . .	15
2.6 Implementazione . . . . .	15
2.6.1 Sottosistema Server: Webservice . . . . .	15
2.6.2 Sottosistema Desktop: Tool di notifica . . . . .	16
2.6.3 Sottosistema Mobile: Mobile client . . . . .	16
2.6.4 Web client . . . . .	17
<b>3 Sottosistema Server</b>	<b>18</b>
3.1 Modalità di descrizione . . . . .	18
3.2 Diagramma delle classi . . . . .	18
3.2.1 HttpResponse . . . . .	18
3.2.2 Resource . . . . .	19
3.2.3 Diagramma ad oggetti . . . . .	20
3.3 Diagrammi di sequenza . . . . .	21
3.3.1 Resource.create . . . . .	21
3.3.2 Resource.read . . . . .	22
3.3.3 Resource.delete . . . . .	23
3.3.4 Resource.update . . . . .	24
3.3.5 Resource.index . . . . .	25
3.3.6 Resource.new . . . . .	26
3.3.7 Resource.edit . . . . .	27
3.3.8 Richieste non-RESTful . . . . .	27
3.4 Lato Server - Descrizione tecnica . . . . .	27
3.5 Modello . . . . .	27
3.5.1 ActiveModel . . . . .	30
3.5.2 ActiveRecord . . . . .	30

3.6	Controllo . . . . .	30
3.6.1	ActionDispatch . . . . .	30
3.6.2	ActionController . . . . .	30
3.7	Vista . . . . .	30
3.7.1	Viste non HTML . . . . .	31
3.7.2	Viste HTML . . . . .	31
3.7.3	Layouts . . . . .	32
3.8	Implementazione REST . . . . .	32
3.8.1	Struttura di ogni risorsa . . . . .	32
<b>4</b>	<b>Sottosistema Desktop</b>	<b>34</b>
4.1	Diagramma dei componenti . . . . .	34
4.2	Diagramma delle classi . . . . .	36
4.3	Diagramma delle attività . . . . .	39
4.4	Diagrammi di sequenza . . . . .	40
<b>5</b>	<b>Sottosistema Mobile</b>	<b>42</b>
5.1	Componenti del sottosistema . . . . .	42
5.1.1	Diagramma dei componenti . . . . .	42
5.1.2	Descrizioni componenti . . . . .	42
5.2	Diagramma delle classi . . . . .	46
5.3	Diagramma delle attività . . . . .	48
5.4	Diagrammi di sequenza . . . . .	50
<b>6</b>	<b>Tracciamento relazioni componenti - requisiti</b>	<b>54</b>
6.1	Sottosistema Server . . . . .	54
6.2	Sottosistema Desktop . . . . .	56
6.3	Sottosistema Mobile . . . . .	57
<b>7</b>	<b>Stime di fattibilità e di bisogno di risorse</b>	<b>59</b>
7.1	Modalità decisionali . . . . .	59
7.2	Conoscenze tecniche . . . . .	59

## Elenco delle tabelle

1	HTTP/URI/CRUD . . . . .	10
2	Server, tracciamento requisiti-classi . . . . .	54
3	Desktop, tracciamento componenti-requisiti . . . . .	56
4	Desktop, tracciamento requisiti-componenti . . . . .	56
5	Mobile, tracciamento componenti-classi-requisiti . . . . .	57
6	Mobile, tracciamento requisiti-componenti-classi . . . . .	57

## Elenco delle figure

1	Rappresentazione astratta architettura Client-Server . . . . .	9
2	Server, Model-View-Controller . . . . .	13
3	Struttura generale del sistema . . . . .	15
4	Sottosistema Server . . . . .	16
5	Sottosistema Desktop . . . . .	16
6	Sottosistema Mobile . . . . .	17
7	Server - HttpResponse . . . . .	18
8	Server - Resource generica . . . . .	19
9	Server - Resource . . . . .	19
10	Server - Diagramma ad oggetti . . . . .	20
11	Server - Resource.create . . . . .	21

12	Server - Resource.read . . . . .	22
13	Server - Resource.delete . . . . .	23
14	Server - Resource.update . . . . .	24
15	Server - Resource.index . . . . .	25
16	Server - Resource.new . . . . .	26
17	Server - Resource.edit . . . . .	27
18	Desktop, diagramma dei componenti . . . . .	34
19	Desktop, componente Manager . . . . .	35
20	Desktop, componente Account . . . . .	35
21	Desktop, componente Client . . . . .	36
22	Desktop, diagramma delle classi . . . . .	37
23	Desktop, diagramma delle attività . . . . .	39
24	Desktop, diagramma di sequenza: login . . . . .	40
25	Desktop, diagramma di sequenza: logout . . . . .	41
26	Mobile, diagramma dei componenti . . . . .	42
27	Mobile, componente Model . . . . .	43
28	Mobile, componente Presenter . . . . .	43
29	Mobile, componente View . . . . .	44
30	Mobile, componente Util . . . . .	45
31	Mobile, componente Exception . . . . .	45
32	Mobile, diagramma delle classi . . . . .	46
33	Mobile, diagramma delle attività . . . . .	49
34	Login effettuato con successo . . . . .	50
35	Recupera quest assegnate all'utente. . . . .	51
36	Recupera e svolgi una quest. . . . .	52
37	Richiesta di una classifica . . . . .	53

## 1 Introduzione

### 1.1 Scopo del documento

Il documento presente delinea lo studio sulla progettazione ad alto livello eseguita in merito al sistema PMAC. Pertanto in esso viene specificata la struttura generale del sistema indicando i design pattern adottati, i sottosistemi coinvolti con i rispettivi componenti delineati e le attività principali del sistema.

### 1.2 Scopo del prodotto

Il sistema software PMAC si pone come obiettivo la realizzazione di una piattaforma innovativa per l'apprendimento comportamentale nell'ambito della sicurezza del lavoro, che utilizzi le tecniche della gamification per incentivare il coinvolgimento e la partecipazione degli utenti e per scardinare l'instaurarsi di abitudini errate.

### 1.3 Glossario

Per evitare ridondanze tutti i termini e gli acronimi presenti nel seguente documento che necessitano di definizione saranno seguiti da una "g" ad apice ( E.g. User<sup>g</sup> ) alla loro prima occorrenza e saranno riportati in un documento esterno denominato *Glossario.pdf*. Tale documento accompagna e completa il presente e consiste in un listato ordinato di termini e acronimi con le rispettive spiegazioni.

### 1.4 Riferimenti

#### 1.4.1 Normativi

- Norme generali del progetto:  
vedi documento fornito in allegato *NormeDiProgetto\_v3.pdf*
- Capitolato d'appalto:  
<http://www.math.unipd.it/~tullio/IS-1/2011/Progetto/C3.pdf>
- Analisi dei Requisiti:  
vedi documento fornito in allegato *AnalisiDeiRequisiti.pdf*

#### 1.4.2 Informativi

- Ruby:  
[http://en.wikipedia.org/wiki/Ruby\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Ruby_(programming_language))
- Ruby on Rails:  
[http://en.wikipedia.org/wiki/Ruby\\_on\\_Rails](http://en.wikipedia.org/wiki/Ruby_on_Rails)
- Representational State Transfer:  
[http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer)
- Disadattamento dell'impedenza tra paradigma relazionale e ad-oggetti:  
<http://en.wikipedia.org/wiki/Object-relational impedance mismatch>
- Resource oriented architecture:  
[http://en.wikipedia.org/wiki/Resource-oriented\\_architecture](http://en.wikipedia.org/wiki/Resource-oriented_architecture)
- ActiveRecord su framework RoR<sup>g</sup>:  
<https://github.com/rails/rails/blob/master/activerecord/README.rdoc>
- ActiveModel su framework RoR:  
<https://github.com/rails/rails/blob/master/activemodel/README.rdoc>

- Bootstrap:  
<http://twitter.github.com/bootstrap/>
- Libreria libMaya:  
<http://websvn.frubar.net/wsvn/libmaia/>
- Libreria SimpleCrypt:  
[http://developer.qt.nokia.com/wiki/Simple\\_encryption](http://developer.qt.nokia.com/wiki/Simple_encryption)
- Crow's Foot Notation:  
<http://www2.cs.uregina.ca/~bernatja/crowsfoot.html>
- Android SDK:  
<http://developer.android.com/sdk/index.html>
- Guida Ruby on Rails <http://ruby.railstutorial.org/chapters/a-demo-app>

## 2 Definizione del prodotto

### 2.1 Metodo e formalismo di specifica

Per i diagrammi delle classi, dei package, di sequenza e di attività è stato usato il linguaggio UML alla versione 2.0. Per il diagramma ad oggetti a figura 10, è stata utilizzata la notazione Crow's Foot per descrivere le relazioni tra le entità.

### 2.2 Stili architetturali - Client-Server

Tutti gli accessi alle risorse vengono effettuati tramite l'istanziazione di un interfaccia da parte di software cliente residente su un terminale, che tramite il protocollo HTTP interagisce con un software servente residente su un computer remoto.

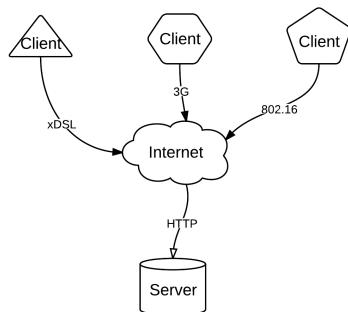


Figura 1: Rappresentazione astratta architettura Client-Server

#### 2.2.1 Client

I Client sono dei software residenti su terminali che permettono l'interazione con il sistema remoto. Per nuove implementazioni su dispositivi, non restringiamo le possibili soluzioni se non per l'obbligo di interazione tramite il protocollo HTTP.

#### 2.2.2 Server

La scelta della configurazione del/dei client esula dagli obiettivi di questo documento. L'ambiente ospitante dovrà quindi soddisfare l'albero delle dipendenze di tutti i componenti. Diamo comunque una visione ad alto livello delle configurazioni adeguate.

#### 2.2.3 Interfaccia

Secondo l'architettura 3-Tier<sup>g</sup>, è logico pensare alle classi di entità clienti nello strato presentation, che interagiscono con una o più entità servente nello strato logic.

La progettazione e lo sviluppo di soluzioni di protocollo personalizzate sarebbe un pesante limite del progetto, che, a nostro parere, è plausibile pensare possa essere esteso come minimo supportando varie sottoclassi di clienti (tipologie, marche, modelli ..).

**Web service** Per una buona estendibilità a questo livello, si rende necessaria un'interfaccia uniformata di comunicazione che, oltre a garantire la semplicità di estensione, elimini le dipendenze dalle implementazioni di entrambi gli strati. Le tecnologie utilizzate a questo livello sono ormai standard di fatto adottati da qualsiasi dispositivo in commercio.

L'implementazione di questi è descritta i maniera esaustiva nella sezione "Representational state trasfer"

#### 2.2.4 Resource Oriented Architecture

Representational State Transfer<sup>g</sup>, di seguito REST<sup>g</sup>, è uno stile architettonico che permette la modellazione dei *business models* tramite *resources* (risorse). Le risorse sono rese accessibili tramite protocollo HTTP sotto forma di web services, e disponibili in varie *representations*. Ogni risorsa espone metodi CRUD che vengono mappati su get, put, post, delete del protocollo HTTP. Il software che adotta questo tipo di struttura è definito come "orientato alla risorsa".

La tipologia di rappresentazione della risorsa richiesta è identificata sull'estensione del file identificato all'interno dell'uri. Ad esempio, una delle rappresentazioni disponibili è quella che incapsula le informazioni in sorgenti interpretabili da un web browser.

Un'altra caratteristica di questo approccio è la mancanza di necessità di tenere traccia dello stato. Ogni metodo è sempre disponibile o nascosto, senza dipendenze dallo stato del processo in esecuzione.

Tramite questa architettura si ottengono:

- Possibilità di comporre semplici richieste relazionali direttamente da uri.
- API pubblica e autenticata, senza necessità di duplicazione di codice
- Interfaccia di comunicazione tra i gli strati presentation e logic
- Stretto legame con le risorse effettivamente presenti su database
- State-less design (anche le sessioni sono modellate su risorse)
- Sicurezza concentrata su singolo componente (HTTPS)
- Disaccoppiamento tra i due strati comunicanti

Questo tipo di approccio aumenta l'estensibilità e la scalabilità del software. Rende inoltre il codice uniformato per risorse, facilmente comprensibile quindi meno dipendente dalle competenze del programmatore.

L'effettiva implementazione RoR<sup>g</sup>, mappa tramite uri non solamente i metodi CRUD ma anche le interfacce web necessarie a lanciarli (azioni index, show, edit).

I seguenti metodi sono esplicitati maggiormente nel paragrafo 3.2.2.

Tabella 1: HTTP/URI/CRUD

HTTP Verb	URI	Action
POST	/resource	CREATE (C)
GET	/resource/id	SHOW (R)
PUT	/resource/id	UPDATE (U)
DELETE	/resource/id	DELETE (D)
GET	/resource	INDEX
GET	/resource/new	NEW
GET	/resource/id/edit	EDIT

## 2.2.5 Presentation

Lo strato presentation si occupa della gestione della presentazione dei dati all'utente finale. All'interno di questo strato rientrano tutti le tipologie di client.

**Webapp** Il mercato dei computer da scrivania è relativamente eterogeneo. La migliore soluzione per ottenere una buona portabilità è lo sviluppo di interfacce web che, eseguendo a un livello di astrazione maggiore tramite il browser, riescono a essere portabili nel maggior numero di dispositivi.

Le applicazioni web sono qualitativamente peggiori rispetto a certi parametri di valutazione (reazione agli stimoli utente, velocità, funzionalità ..), ma per i requisiti del software in progettazione sono sicuramente sufficienti.

**API pubblica** Il paragrafo "Web Service"<sup>g</sup> della sezione precedente illustra il protocollo tramite il quale le entità dello strato presentation comunicano con quelle dello strato logic. Tutte le entità che entrano a contatto con il sistema hanno a disposizione una serie di classi e metodi usufruibili dall'esterno: a tutti gli effetti una libreria remota. L'implementazione effettiva è descritta nel capitolo "Representational state transfer". Per lo scambio di messaggi è stato necessario l'utilizzo di linguaggi di markup per regolare il trasferimento di oggetti su testi semplici. Abbiamo scelto JSON e XML per la loro popolarità e facilità di utilizzo.

## 2.2.6 Logic

Lo strato logico è il responsabile dell'implementazione dell'interfaccia REST. Si occupa di:

- Gestione delle richieste provenienti dall'esterno
- Preparazione e inoltro richieste di modifica dati persistenti
- Preparazione rappresentazioni delle risorse e il loro ritorno

In particolare, per le rappresentazioni a interfacce web:

- Creazione HTML + CSS + JS dinamicamente rispetto alla risorsa richiesta
- Gestione delle problematiche per la corretta fruizione dei contenuti

A questo livello è decisivo il design pattern MVC con decorator per le seguenti motivazioni:

- La modellazione di diverse view è perfettamente mappabile con i diversi tipi di rappresentazioni di risorse
- Come descritto in seguito, il model rappresenta la virtualizzazione dell'interfaccia allo strato sottostante
- Il controller ha la gestione dell'interfaccia REST
- La classe decoratrice libera il controller da responsabilità diverse dalla gestione dell'interfaccia

Questo approccio rende il codice pressoché uniforme per ogni risorsa. Questo determina la facilità di estensione, mantenimento, sviluppo minimizzando la dipendenza dalle competenze dei singoli programmati.

**Application server** Per i compiti descritti nella sezione precedente si utilizzano linguaggi object oriented interpretati. Utilizziamo Ruby e in particolare RoR per il suo alto livello di astrazione, predisposizione alla creazione di interfacce REST, predisposizione a interfacce ORM<sup>g</sup>, creazione di contenuti web. Ruby inoltre ci permette di provare sul campo alcune caratteristiche dei linguaggi funzionali.

**Interfaccia a database** La persistenza dei dati è un compito complesso di grandi difficoltà di implementazione, tanto da meritare software dedicati detti DBMS. Nonostante esistano DBMS<sup>g</sup> orientati agli oggetti, i più efficaci e utilizzati sono relazionali, all'interno dei quali la modellazione delle entità è sostanzialmente differente. Per un buon disaccoppiamento delle componenti quindi è necessaria un'interfaccia che permetta al livello applicazione di comunicare con il sottostante.

**ORM** Object relational mapping, di seguito ORM, è una tecnica per la mappatura di sistemi orientati agli oggetti a sistemi di tipo relazionale. Questa mappatura permette il disaccoppiamento delle parti purtroppo non in maniera indolore. Il livello di superamento delle incompatibilità tra i due mondi è descritto nella sezione successiva. L'implementazione è realizzata tramite il pattern Active Record descritto nella sezione dedicata. Si rimanda al documento informativo *"Disadattamento dell'impedenza tra paradigma relazionale e ad-oggetti"* per ulteriori approfondimenti.

**Adattamento di DBMS esistenti** RoR astrae il pattern Active Record dall'effettivo linguaggio sql sottostante tramite librerie esterne. Per nuove tipologie di DBMS è quindi sufficiente lo sviluppo di una libreria adeguata che implementi le effettive query SQL<sup>g</sup>.

### 2.2.7 Data

Lo strato più basso del software ha le responsabilità di:

- Rendere i dati persistenti nel tempo
- Fornire un'interfaccia avanzata per l'accesso e la modifica dei dati
- Gestire le problematiche di concorrenza nell'accesso ai dati

Queste funzionalità sono soddisfatte dall'uso di DBMS. Come intuibile dalle sezioni precedenti, abbiamo scelto database relazionali per la rappresentazione e manipolazione dei dati, in particolare MySql.

## 2.3 Design patterns

L'architettura del software si avvale della corretta implementazione di design pattern per la soluzione sicura di problematiche progettuali note. In particolare è stata data precedenza al disaccoppiamento delle componenti a favore dell'estensibilità e manutenibilità del progetto.

### 2.3.1 Model-View-Controller

Il pattern architettonico permette la divisione logica tramite la realizzazione di interfacce per i ruoli di modello, vista e controllo. Questo pattern ci permette per esempio la modifica dell'implementazione di uno dei tre senza dover necessariamente modificare gli altri. Molto utile anche per la creazione delle viste specifiche per la chiamata remota a procedure. Nello specifico, il framework RoR presuppone l'utilizzo del pattern.

In figura 3 viene rappresentata l'implementazione del pattern Model-View-Controller dal framework RoR e in particolare come viene gestita un'azione generica (in questo caso richiede User) dai 3 componenti.

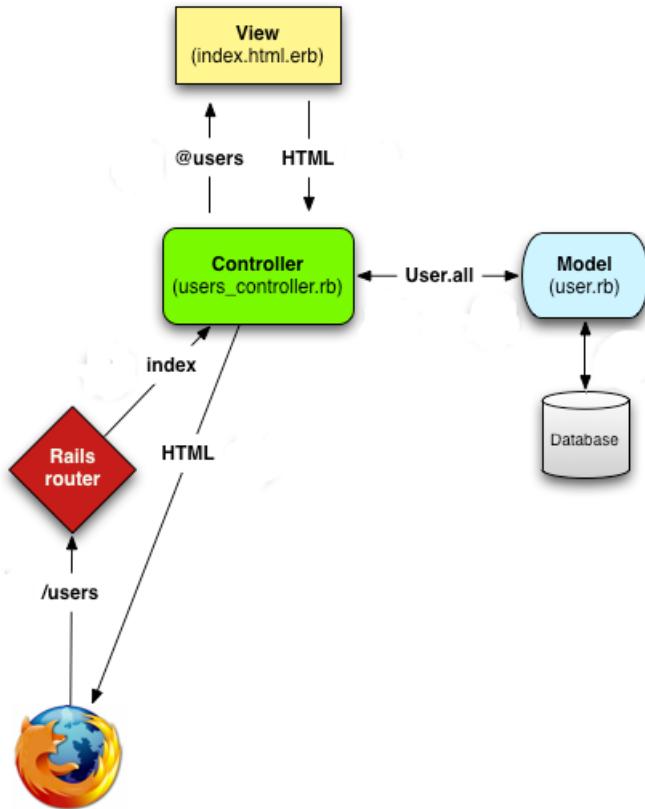


Figura 2: Server, Model-View-Controller

### 2.3.2 Active Record

Il pattern è già incluso nelle librerie offerte dal framework RoR, tramite subclassing della classe ActiveRecord. Si tratta di encapsulare ogni tabella del database relazionale con una classe, mappando gli attributi ai campi dati e gli oggetti istanziati ad ogni record. I metodi della classe garantiscono l'accesso in lettura e scrittura tramite l'effettivo codice SQL. Il normale caso è che la classe sia la parte di modello del pattern MVC, ma possono coesistere casi in cui il modello non sia legato al database (quindi non adottando il pattern). Questo permette l'astrazione del codice SQL dalla tipologia di database sottostante, con enormi vantaggi in termini di mantenibilità e portabilità del codice.

### 2.3.3 Decorator

Usiamo il pattern per la preparazione dei modelli per il loro inserimento nelle viste. Questo permette l'ulteriore divisione logica di procedure riguardanti la preparazione dei dati grezzi per essere esposti nell'interfaccia.

### 2.3.4 Observer

I modelli presenti a livello applicazione nel server possono avere necessità di accendere procedure particolari in certi momenti specifici della loro vita. Per implementare ciò, usiamo il pattern observer.

### 2.3.5 Singleton

Pattern per la creazione di una singola istanza di una classe la quale risulterà accessibile in maniera statica. Tale design pattern risulta utile in questo progetto in quanto sono previste

numerose entità che non devono avere più di una istanza.

Ne è un esempio l'implementazione della classe Client, presente nel sottosistema Desktop che verrà illustrato in seguito.

## 2.4 Tecnologie

Di seguito illustrate le tecnologie (linguaggi, librerie) e le motivazioni di scelta. Per una completa argomentazione si assume che il lettore abbia preso visione del documento informativo riguardo all'argomento corrente.

### 2.4.1 Lato server

La scelta di tecnologie è stata influenzata principalmente dalla ricerca di soluzioni sicure, già testate e scalabili.

**Ruby** Ruby è un linguaggio interpretato che privilegia la semplificazione della codifica per il programmatore. Era originariamente scritto con il linguaggio C, ma ai giorni odierni esistono varie altre alternative (tra le quali Java). Tra le altre cose include un package manager RubyGems, che rende semplice l'installazione di librerie esterne (tra le quali Ruby on Rails).

**Ruby on Rails** Ruby on Rails, di seguito RoR, è un framework opensource relativamente giovane sviluppato sul linguaggio interpretato Ruby. La caratteristica principale che lo differenzia rispetto alla concorrenza è la community di contributors che direttamente o indirettamente rendono disponibili librerie facilmente installabili tramite RubyGems<sup>9</sup>. Il framework rende estremamente astratta la modellazione della realtà sollevando il programmatore dalla progettazione di buona parte della struttura architetturale di un software basato sul web. Per il suo utilizzo cosciente quindi è necessaria la conoscenza del dietro le quinte, che a sua volta ha bisogno di competenze acquisite in precedenza riguardo paradigmi, design pattern e best practice di programmazione.

### 2.4.2 Lato Desktop

**C++** Per la realizzazione del gestore delle notifiche lato Desktop si è deciso di utilizzare il linguaggio di programmazione Object Oriented<sup>9</sup> C++. La scelta di tale linguaggio è dovuta ad un buona efficienza del linguaggio.

- vantaggi:  
efficienza del linguaggio.
- svantaggi:  
gestione della memoria a cura del programmatore, difficoltà per la portabilità.

**Framework Qt** Qt è un framework multipiattaforma per lo sviluppo di programmi con interfaccia grafica tramite l'uso di widget, è basata su linguaggio C++.

- vantaggi:  
il framework mette a disposizione del programmatore molte librerie sia grafiche che logiche fornendo al funzionalità di sviluppo molto completo.
- svantaggi:  
non sono stati trovati svantaggi in merito.

### 2.4.3 Lato Mobile

**Android SDK** Nel nostro ambito Java viene utilizzato per la creazione di un'applicazione interfacciabile su dispositivi mobile con sistema operativo Android. La versione utilizzata è Android SDK 2.2<sup>9</sup>, al fine di permettere alla quasi totalità dei dispositivi di ricevere notifiche push. Questo tipo di notifiche è reso possibile attraverso C2DM<sup>9</sup> (Android Cloud to Device Messaging Framework). Le librerie grafiche utilizzate sono quelle fornite da Android, mentre per l'interfacciamento al Cloud è stato fatto riuso di codice messo a disposizione direttamente dal team di Google.

## 2.5 Librerie Aggiuntive

Usiamo librerie aggiuntive per minimizzare la codifica di elementi ripetitivi. La ripetizione di codice impone limiti all'estensibilità, manutenibilità, modificabilità e alla chiarezza del codice sorgente. L'utilizzo di librerie esterne comporta dispendio di tempo per la ricerca del componente, usabilità, e affidabilità. L'implementazione di codice mantenuto esternamente nel software comporta dipendenze da eventuali nuove versioni, flessibilità delle funzionalità, bugs esterni, prestazioni rallentate. Riteniamo che, valutate tutte le precedenti, l'utilizzo di librerie aggiuntive ed open-source aumenti il valore complessivo al sistema, formandoci allo stesso tempo riguardo alle best-practice utilizzate per scriverle da gruppi con più esperienza del nostro.

## 2.6 Implementazione

In figura 3 vengono riportate le componenti di implementazione della struttura descritta nella sezione Stili architetturali. Le componenti attualmente previste per lo sviluppo sono denominate: lato server, sistema di notifica, client web e client mobile.

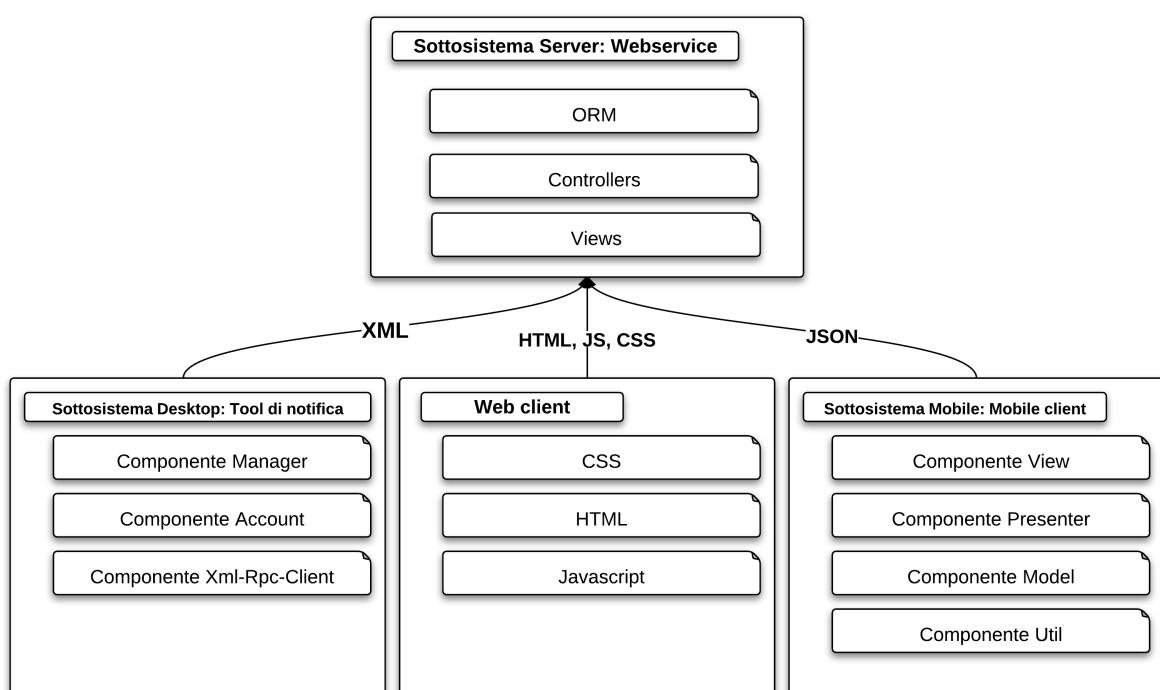


Figura 3: Struttura generale del sistema

### 2.6.1 Sottosistema Server: Webservice

Il sottosistema server è responsabile della gestione di tutte le risorse di PMAC. Si occupa di interagire con i client, della conservazione e elaborazione dei dati. L'interazione con i web client è permessa attraverso la visualizzazione di pagine HTML, coadiuvate da fogli di stile CSS e di

linguaggio lato client Javascript. L'interazione con i client mobile e i tool di notifica desktop è realizzata attraverso comunicazione di messaggi XML e JSON.

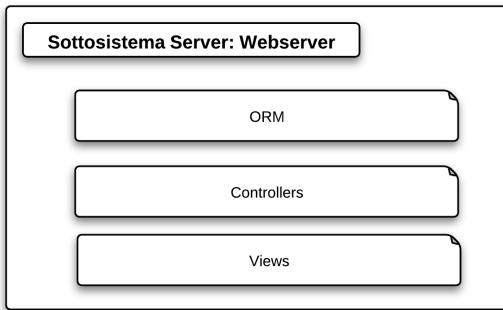


Figura 4: Sottosistema Server

### 2.6.2 Sottosistema Desktop: Tool di notifica

Il sottosistema Desktop si occupa di gestire la ricezione delle notifiche lato desktop per un utente del tipo Desktop-user, e consiste in una applicazione di tipo Tray Icon<sup>9</sup> semplice ed intuitiva.

Una volta installata l'applicazione nell'apposito computer, il Desktop-user potrà effettuare il login al server PMAC attraverso le proprie credenziali. In questo modo l'applicazione potrà controllare periodicamente la presenza di nuove quest da svolgere assegnate allo specifico utente, o in caso contrario, l'utente potrà richiederne di nuove.

Il sottosistema sarà implementato con l'utilizzo del linguaggio di programmazione object oriented C++, e il framework Qt correlato.

Tale sottosistema è stato suddiviso in tre componenti logici come illustrato nella figura 5.

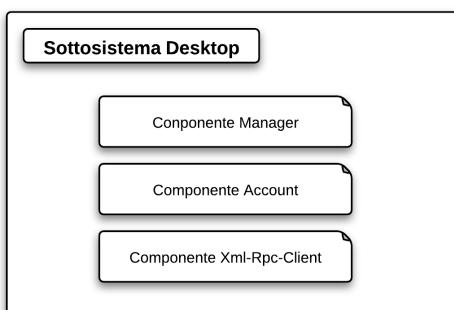


Figura 5: Sottosistema Desktop

### 2.6.3 Sottosistema Mobile: Mobile client

Il sottosistema mobile prevede lo sviluppo di un applicazione dedicata per il sistema operativo Android che offre all'utente un esperienza di navigazione alla pari, o più simile possibile, a quella offerta da una postazione fissa.

Questo sottosistema prevede l'interazione con il server per il recupero di tutte le informazioni necessarie e offre un servizio di notifica push per la segnalazione della presenza di nuove Quest disponibili all'utente autenticato dal dispositivo mobile.

Per lo sviluppo architetturale abbiamo scelto di seguire il design pattern MVP, per una massima

separazione ed estendibilità del codice.

Sono quindi stati previsti i seguenti componenti principali:

- Componente Model, che rappresenta i dati gestiti dall'applicazione.
- Componente View, che consiste nelle classi che gestiscono l'interfaccia grafica proposta all'utente.
- Componente Presenter che è incaricato di gestire le interazioni tra View e Model, in particolare riceve le notifiche generate dell'interazione dell'utente con l'interfaccia grafica e porta a termine le dovute funzioni tramite l'interazione con il Model.

Prevediamo poi l'utilizzo di package di supporto come util ed exceptions che forniscono funzionalità come quelle di parsing e gestione delle eccezioni. Abbiamo ritenuto di separare questi componenti dal resto della struttura perchè non appartengono al design pattern MVP.

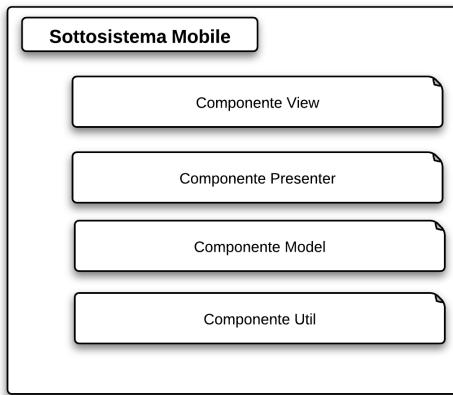


Figura 6: Sottosistema Mobile

#### 2.6.4 Web client

Il web client è un'interfaccia modulare per l'accesso e la modifica dei contenuti utente. Abbiamo identificato due tipologie di pagine web:

- Pagine di accesso alle risorse, derivate dalle necessità CRUD
- Pagine statiche, derivate da altri requisiti (es. informazioni del software)

Le prime saranno uniformate rispetto ad ogni tipo di risorsa. Le seconde invece avranno layout e contenuti diversi. Riteniamo molto importante la fluidità e l'esperienza d'uso utente. Perciò ci avvaliamo di librerie esterne sicure per fogli di stile e javascript per il buon funzionamento cross-browser e il gusto visivo. Per le stesse ragioni in particolare Ajax ed elementi dinamici saranno utilizzati per simulare la responsività di una vera applicazione locale.

### 3 Sottosistema Server

#### 3.1 Modalità di descrizione

Nel presente documento non verrà descritta ogni classe nel dettaglio, ma verrà utilizzato per il sottosistema server un approccio più ad alto livello, orientato alle funzionalità che il webservice offre verso l'esterno. Le richieste esterne verranno effettuate (con qualche eccezione, specificata nella sezione 3.3.8) attraverso l'interfaccia REST precedentemente specificata nel capitolo 2. Le richieste, quindi, saranno tutte verso risorse disponibili nel webservice.

Per questo motivo i diagrammi che seguono saranno orientati alla *risorsa*, le varie classi rappresenteranno una classe logica per le risorse e per le risposte che il webservice può dare ad un generico client. La definizione implementativa è rimandata al documento "Definizione del Prodotto".

#### 3.2 Diagramma delle classi

##### 3.2.1 HttpResponse

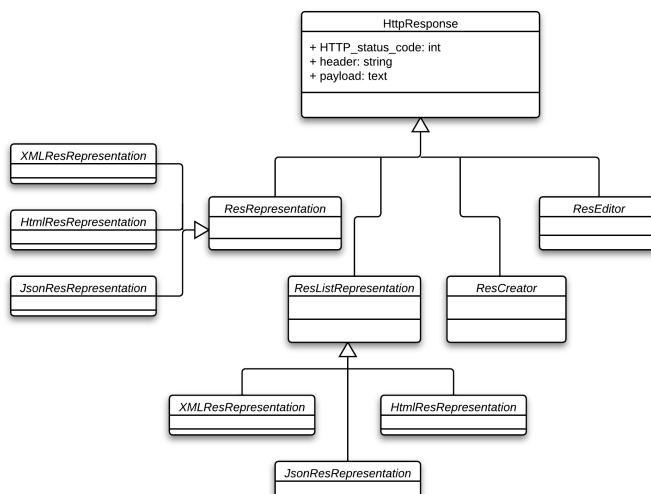


Figura 7: Server - HttpResponse

Lo scopo del diagramma è il presentare le risposte che il webservice può dare ad un generico client. Di seguito la descrizione delle classi:

- Classe `HttpResponse`: rappresenta la risposta HTTP che il webservice restituisce in seguito ad una richiesta REST. E' idealmente caratterizzata da 3 attributi:
  - `HTTP_status_code`: è un intero che rappresenta l'esito della richiesta (e.g. 201 = Created, 401 = Unauthorized)
  - `header`: contiene meta-information sul contenuto del payload
  - `payload`: è l'effettivo messaggio trasmesso tramite HTTP
- Classe `ResEditor`: è una `HttpResponse` contenente una vista HTML con la quale è possibile modificare i campi di una risorsa. Genericamente, una form.
- Classe `ResCreator`: è una `HttpResponse` contenente una vista HTML con la quale è possibile creare una risorsa. Anche in questo caso, genericamente è una form.
- Classe `ResListRepresentation`: è una `HttpResponse` contenente lista di istanze di una generica risorsa. Da essa ereditano le classi `XMLResListRepresentation`, `HtmlResListRepresentation`, `JsonResListRepresentation` le quali incapsulano la lista in, rispettivamente, XML, HTML, JSON.

- Classe ResRepresentation: è una `HttpResponse` contenente una vista HTML che visualizza una risorsa. Da essa ereditano le classi `XMLResRepresentation`, `HtmlResRepresentation`, `JsonResRepresentation` le quali incapsulano la risorsa in, rispettivamente, XML, HTML, JSON.

### 3.2.2 Resource

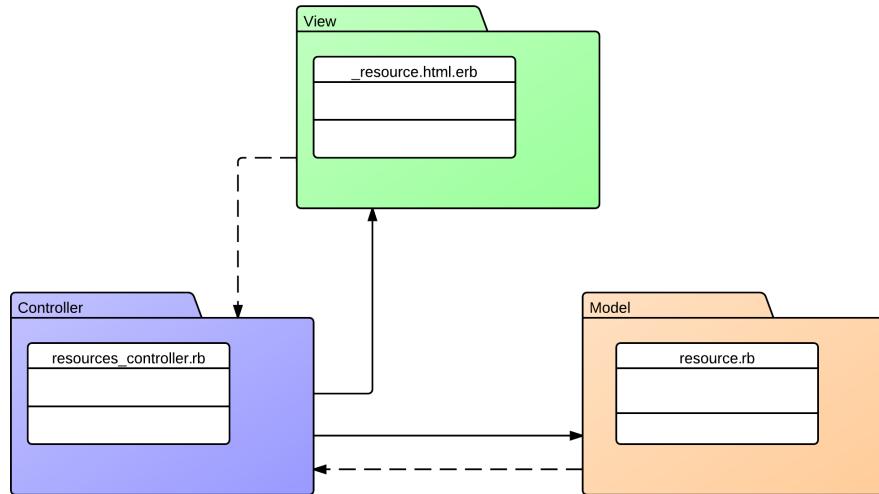


Figura 8: Server - Resource generica

Nel diagramma è rappresentato come è mappata una resource generica secondo il modello model-view-controller del framework RoR.

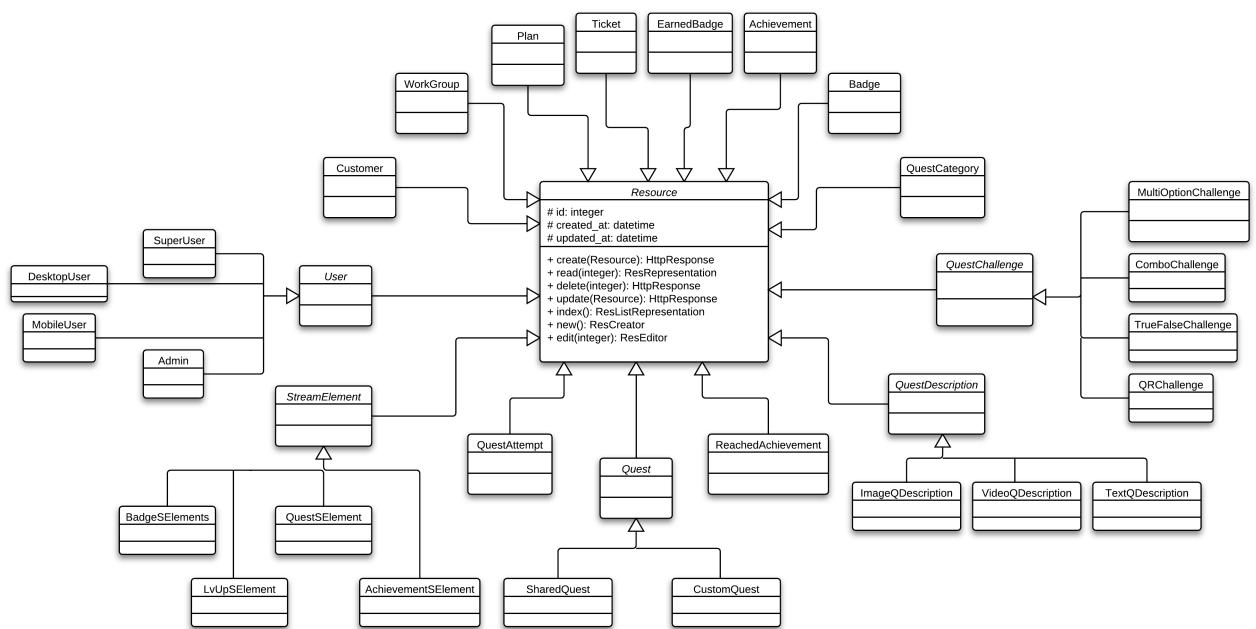


Figura 9: Server - Resource

Questo diagramma vuole mostrare come ogni classe logica disponibile nel webservice sia in realtà una risorsa. Le relazioni e le dipendenze più dettagliate tra le classi saranno visibili in

seguito nel diagramma degli oggetti in figura 10.

La descrizione delle classi è riportata nella sezione 3.5 Modello.

### 3.2.3 Diagramma ad oggetti

Il seguente diagramma ad oggetti vuole mettere in evidenza come una risorsa sia effettivamente una tabella nel DB, e le relazioni tra le varie risorse.

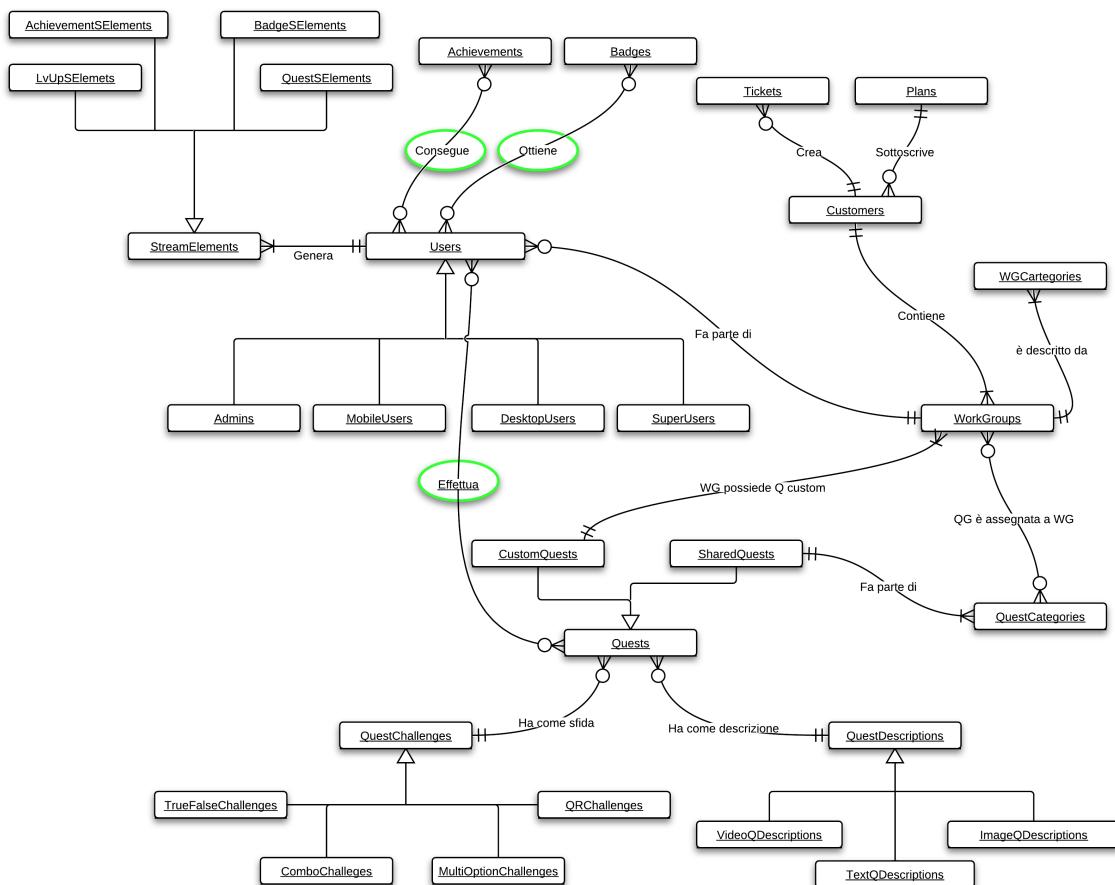


Figura 10: Server - Diagramma ad oggetti

Da notare il fatto che dalle relazioni N:M nascono delle risorse:

- Dalla relazione Users <-> Quests nasce la risorsa QuestAttempt.
- Dalla relazione Users <-> Achievements nasce la risorsa ReachedAchievement.
- Dalla relazione Users <-> Badge nasce la risorsa EarnedBadge.

Altri punti che meritano una spiegazione:

- Dalla relazione N:M WorkGroups <-> QuestCategories è stato deciso di non far emergere una risorsa, seppur una tabella debba necessariamente essere creata.
- L'entità WGCategories è una tabella nata per evitare ridondanze nella denominazione dei WorkGroup, non è stato quindi ritenuto necessario mapparla come risorsa.

### 3.3 Diagrammi di sequenza

Nelle realizzazioni dei diagrammi di sequenza, l'obiettivo è stato il far notare come il webservice produce una risposta a seguito di ognuno dei 7 metodi della classe Resource che un generico client può invocare.

La classe indicata come <Resource>Controller rappresenta il controller della specifica risorsa (C del design pattern MVC), mentre la classe indicata come <Resource> ne rappresenta il modello (M del design pattern MVC).

La richiesta del client è definita dall'effettiva richiesta HTTP, il mapping richiesta HTTP -> (controller, metodo) è dato dalle configurazioni espresse nel file routes.rb. Questo meccanismo sarà discusso nel dettaglio nel documento Definizione di Prodotto.

#### 3.3.1 Resource.create

Il seguente diagramma di sequenza illustra come il webservice crea una nuova istanza di una risorsa. La risorsa viene richiesta dal Client attraverso una GET Http, e tale richiesta perviene al Controller che crea la nuova istanza della risorsa richiesta. La risorsa viene quindi salvata nel modello e restituita al Client tramite protocollo http.

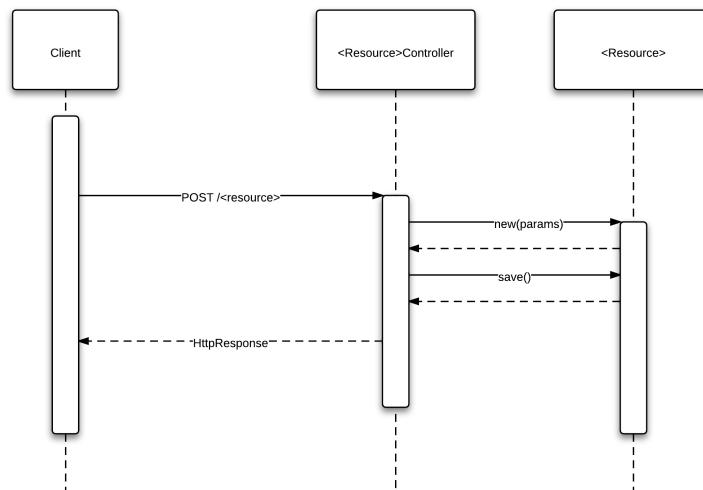


Figura 11: Server - Resource.create

### 3.3.2 Resource.read

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta per la lettura di una specifica risorsa.

Il Client invia una richiesta GET Http indicando l'id della risorsa richiesta e la richiesta arriva al Controller che cerca la risorsa nel modello. Se la risorsa è presente nel modello, questa viene restituita al client mediante una ResRepresentation, altrimenti se la risorsa specificata non esiste, il webservice ritorna al Client una HttpResponse con un codice di stato rappresentante un errore.

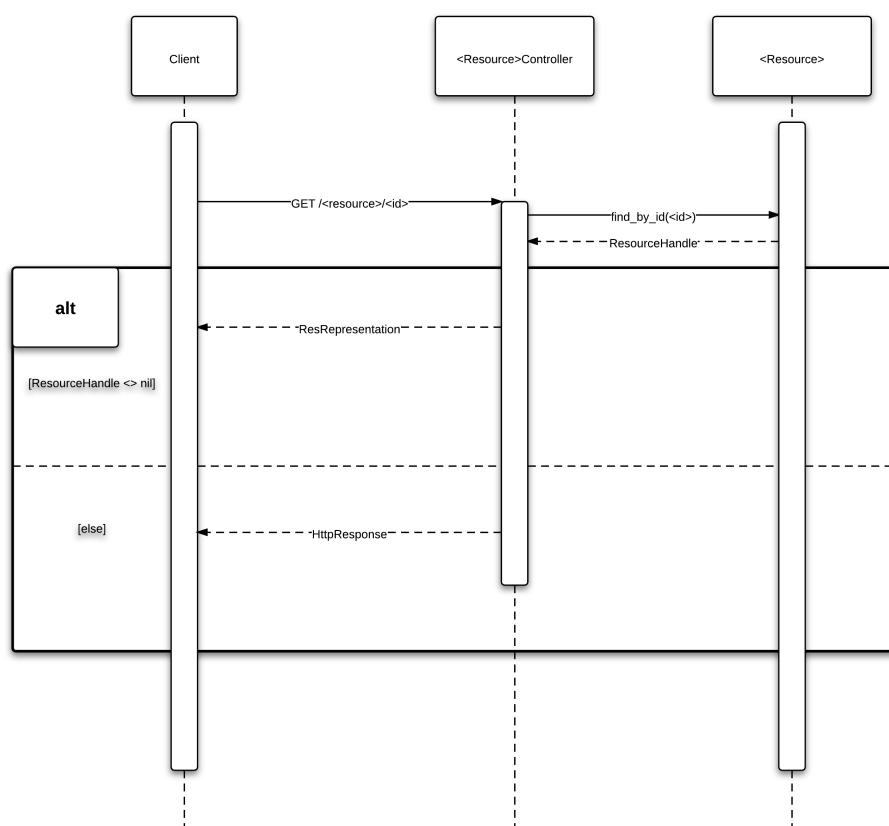


Figura 12: Server - Resource.read

### 3.3.3 Resource.delete

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta per la cancellazione di una specifica risorsa.

Il Client invia una richiesta di cancellare una risorsa indicando l'id della risorsa da cancellare. La richiesta viene captata dal Controller che ricerca nel modello la presenza della risorsa con lo specifico id. Se la risorsa è presente nel modello, questa viene eliminata. Successivamente il Controller invia al Client una HttpResponseMessage indicante l'esito positivo se la risorsa era presente ed è stata cancellata, altrimenti l'HttpResponse avrà un codice di stato rappresentante un errore indicando l'esito negativo della cancellazione.

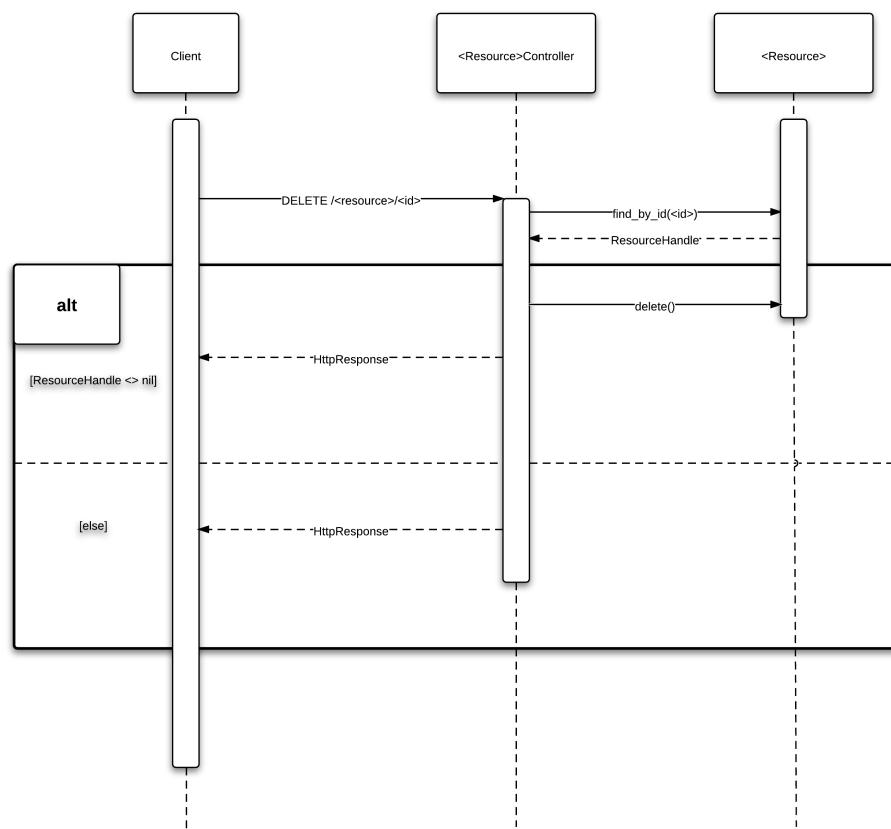


Figura 13: Server - Resource.delete

### 3.3.4 Resource.update

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta per l'update di una specifica risorsa.

Il Client invia una richiesta di effettuare un update di una risorsa indicandone l'id specifico. La richiesta viene captata dal Controller che cerca la risorsa nel modello, e se questa viene trovata, il Controller esegue l'update. Successivamente il Controller ritorna al Client una HttpResponseMessage indicante l'esito dell'operazione.

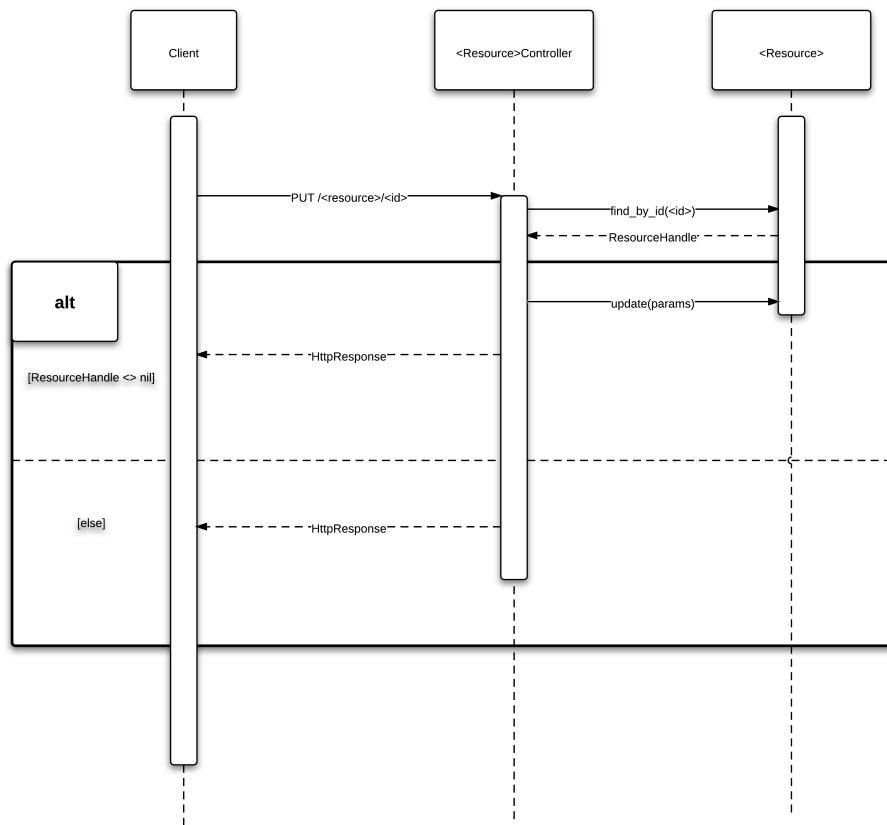


Figura 14: Server - Resource.update

### 3.3.5 Resource.index

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta per la visualizzazione di una lista di istanze di una specifica risorsa.

Il Client invia la richiesta che viene captata dal Controller. Quest'ultimo esamina il modello recuperando le risorse necessarie, e successivamente le restituisce al Client con una ResListRepresentation.

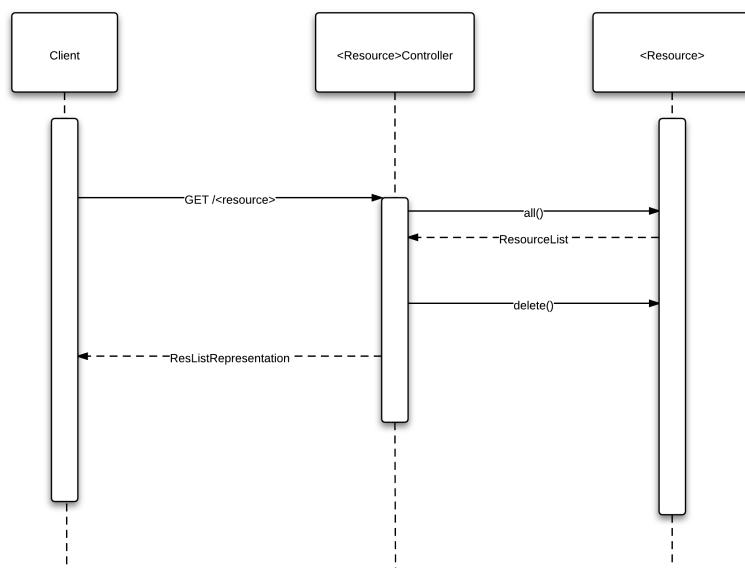


Figura 15: Server - Resource.index

### 3.3.6 Resource.new

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta riguardante la visualizzazione di una form per la creazione di una specifica risorsa. Il Client invia la richiesta per la visualizzazione di una form e la richiesta viene captata dal Controller che ritorna al Client la relativa ResCreator contenente la risposta.

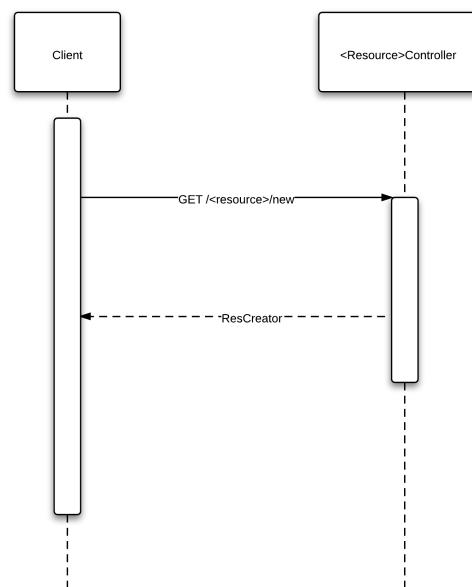


Figura 16: Server - Resource.new

### 3.3.7 Resource.edit

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta riguardante la visualizzazione di una form per la modifica di una specifica risorsa.

Il Client invia la richiesta per la visualizzazione di una form e la richiesta viene captata dal Controller che ritorna al Client la relativa ResEditor contenente la risposta.

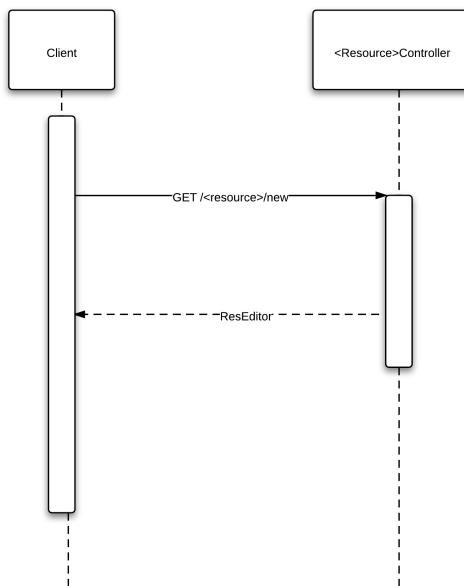


Figura 17: Server - Resource.edit

### 3.3.8 Richieste non-RESTful

Per particolari richieste al webservice non mappabili attraverso risorse REST (e.g. il check di notifiche attraverso il client desktop), è stato scelto di utilizzare il protocollo XMLRPC. Questo verrà implementato lato server tramite un controller specifico.

## 3.4 Lato Server - Descrizione tecnica

La parte server del sistema software si occupa di fornire un’interfaccia ai client per l’interazione con la base di dati. Come conseguenza della architettura RoR, il software è sviluppato utilizzando il design pattern MVC.

- Modello, persistenza
- Controllo, elaborazione
- Vista, consegna

Nelle seguenti sezioni verrà descritta l’implementazione di ogni elemento.

## 3.5 Modello

I modelli sono di solito associati alle risorse e di conseguenza alle entità a livello database; tuttavia le implementazioni possono differire anche notevolmente. E’ plausibile pensare che possano esistere di modelli non persistenti (per esempio le classifiche). Le classi di modello, a livello applicazione, utilizzano le componenti ActiveRecord e ActiveModel (vedere documenti informativi) che forniscono le funzionalità base per ogni modello dati.

Di seguito la descrizione delle classi:

- Classe Resource: E' la classe che rappresenta la risorsa generica. Ha come attributi l'identificativo univoco (id) e la data di creazione e di ultima modifica. Ha come metodi:
  - create: è il metodo attraverso il quale avviene l'istanziazione di una nuova risorsa nel webservice. Riceve come parametro in ingresso la risorsa e restituisce una HttpResponse con Http\_status\_code caratterizzato dall'esito dell'operazione.
  - read: è il metodo con il quale viene restituito una ResRepresentation della risorsa richiesta. Riceve in input l'id univoco della risorsa. Il tipo di risposta (XML, HTML, JSON) è in base al formato richiesto.
  - delete: è il metodo con il quale viene eliminata una risorsa. Riceve in input l'id univoco della risorsa e restituisce una HttpResponse caratterizzata dall'esito dell'operazione.
  - update: è il metodo con il quale viene modificata una risorsa. Riceve in input la risorsa e restituisce una HttpResponse caratterizzata dall'esito dell'operazione.
  - index: è il metodo con il quale viene restituito una ResListRepresentation contenente una lista delle istanze della risorsa. Il tipo di risposta (XML, HTML, JSON) è in base al formato richiesto.
  - new: restituisce un ResCreator con il quale sia possibile caratterizzare una nuova istanza di una risorsa.
  - edit: restituisce un ResEditor con il quale sia possibile modificare una risorsa. Riceve in input l'id univoco della risorsa.

E' marcata come astratta, in quanto istanze di Resource generiche non avrebbero senso.

- Classe User: rappresenta un singolo utente del sistema. Conterrà i dati di autenticazione, i dati personali, e qualsiasi elemento caratterizzante dell'utente. E' una classe astratta: i vari tipi di user sono espressi dalla seguente gerarchia:
  - SuperUser: sottotipo di User, è un utente interno ad ogni azienda cliente, abilitato alla creazione, all'eliminazione e alla modifica dei DesktopUser e dei MobileUser, nonché il sollevamento di ticket, all'interno dell'azienda di appartenenza.
  - DesktopUser: sottotipo di User, è un utente di PMAC abilitato al sistema tramite client desktop.
  - MobileUser: sottotipo di User, è un utente di PMAC abilitato al sistema tramite client mobile.
  - Admin: sottotipo di User, è l'amministratore del sistema.
- WorkGroup: rappresenta un gruppo di lavoro all'interno di un Customer.
- Customer: rappresenta un'azienda cliente. Ogni DesktopUser, MobileUser e SuperUser è associato ad una azienda, tramite il WorkGroup di appartenenza.
- Plan: rappresenta il tipo di contratto sottoscritto dal Customer. Ogni Plan sarà caratterizzato dal numero di licenze, dal prezzo, e da parametri distintivi quali il numero massimo di quest customizzate, la possibilità di quest video, etc.
- Ticket: rappresenta un ticket sollevato da un Customer. E' caratterizzato dalla richiesta testuale associata.
- Achievement: è un obiettivo che uno User può raggiungere. E' caratterizzato dalle condizioni che uno User deve soddisfare per conseguirlo.
- ReachedAchievement: è una doppia User-Achievement, rappresenta il conseguimento di un Achievement da parte di uno User.
- Badge: è un distintivo che un utente può ottenere per meriti particolari (e.g.: miglior utente del mese). Si differenzia dal badge nelle modalità di visualizzazione e dal fatto che i badge possono essere ottenuti da un numero predefinito di utenti, mentre gli achievement sono potenzialmente ottenibili da tutta l'utenza.

- EarnedBadge: è una doppia User-Badge, rappresenta l'ottenimento di un Badge da parte di uno User.
- Quest: rappresenta una quest che uno User può svolgere. E' caratterizzata dalla descrizione della quest e dalla sfida che l'utente deve svolgere. E' una classe astratta, le cui implementazioni sono descritte dalla seguente gerarchia:
  - SharedQuest: sono Quest che possono essere potenzialmente condivise da più Customers.
  - CustomQuest: sono Quest create ad-hoc per un singolo Customer/WorkGroup.
- QuestDescription: rappresenta una descrizione per una quest (e.g un testo, un'immagine, un video). E' una classe astratta, le cui implementazioni sono descritte dalla seguente gerarchia:
  - ImageQDescription: rappresenta una descrizione di una quest composta da un'immagine. E' caratterizzata dall'immagine in questione.
  - VideoQDescription: rappresenta una descrizione di una quest composta da un video. E' caratterizzata dal video in questione.
  - TextQDescription: rappresenta una descrizione di una quest composta da del testo. E' caratterizzata dal testo in questione.
- QuestChallenge: rappresenta la sfida che l'User dovrà svolgere, coerentemente con la QuestDescription visualizzata. E' una classe astratta, le cui implementazioni sono descritte dalla seguente gerarchia:
  - MultiOptionChallenge: è QuestChallenge composta da una domanda alla quale è possibile rispondere con più risposte tra quelle disponibili, non ci sono vincoli sul numero di risposte esatte.
  - ComboChallenge: è una QuestChallenge composta da una domanda alla quale è possibile rispondere con una e una sola risposta tra quelle disponibili.
  - TrueFalseChallenge: è una QuestChallenge composta da più opzioni, le quali possono essere vere o false.
  - QRChallenge: è una QuestChallenge specifica per l'ambiente Mobile, consiste nella scansione di uno specifico QR-Code.
- QuestCategory: rappresenta una categoria di rischio alla quale ogni SharedQuest è associata. Ad ogni WorkGroup verrà assegnato dal PMAC Admin uno specifico gruppo di QuestCategories, in base ai rischi rilevati.
- QuestAttempt: è una doppia User-Quest dove viene registrato il tentativo (riuscito o meno) di svolgimento di una Quest da parte di uno User.
- StreamElement: rappresenta un elemento visualizzabile nello Stream da parte degli User. E' una classe astratta, la seguente gerarchia evidenzia le varie sottoclassi, differenziate dal tipo di evento che porta a generare lo StreamElement:
  - BadgeSElement: rappresenta uno StreamElement generato dall'ottenimento di un Badge da parte di uno User.
  - LvUpSElement: rappresenta uno StreamElement generato da uno User che avanza di livello.
  - QuestSElement: rappresenta uno StreamElement generato da uno User che completa una Quest
  - AcheievemtSElement: rappresenta uno StreamElement generato da uno User che consegne un determinato Achievement.

### 3.5.1 ActiveModel

ActiveModel è un'interfaccia di gestione di funzionalità generiche per modelli. In particolare gestisce:

- Creazione dinamica di metodi per attributi
- *Hooks* per la micro-gestione dei task durante momenti particolari della vita di un'istanza
- Validazione e internazionalizzazione dei dati
- Implementazione del pattern Observer

### 3.5.2 ActiveRecord

ActiveRecord rappresenta l'implementazione del pattern omonimo. Le principali funzionalità sono:

- Gestione delle associazioni.
- Gestione delle aggregazioni.
- Gestione delle validazioni.
- Gestione delle gerarchie.
- Gestione delle transazioni.
- Gestione dell'astrazione dal DBMS tramite adapters (librerie esterne).
- Gestione delle migrazioni.

## 3.6 Controllo

Lo strato controllo è responsabile della ricezione delle richieste, interazione con lo strato modello, elaborazione, preparazione e consegna dei dati allo strato vista. Le componenti del framework coinvolte sono:

- ActionDispatch, gestione del routing.
- ActionController, gestione dell'elaborazione dei dati.

### 3.6.1 ActionDispatch

Principali funzionalità:

- Gestione protocollo HTTP.
- Url mapping per risorse REST e non.

### 3.6.2 ActionController

Classe base per i controller, gestisce le implementazioni dei metodi per la costruzione di viste e filtri per eventuali pre/post-elaborazioni.

## 3.7 Vista

Le componenti di questo strato servono alla costruzione delle viste. Le viste per metodi di risorse possono essere quasi totalmente uniformate. La componente del framework di supporto è ActionView che offre funzionalità di:

- Gestione del codice Ruby embedded.
- Funzioni basilari per composizioni di pagine web.
- Serializzazione per formati XML E JSON.

### 3.7.1 Viste non HTML

Le viste non HTML sono risposte composte da uno o più oggetti, incapsulati su risposte HTTPS autenticate e tramite uno dei formati supportati (attualmente XML e JSON). Nella convenzione, questo tipo di viste sono identificate come *Web Service* e servono alle entità esterne per l'accesso autenticato al sistema. Questa tipologia di viste garantisce un'interfaccia di metodi basilari utilizzabili dall'esterno. Non ci si preoccupa quindi della tipologia del dispositivo che le richiede. I Web Services sono inoltre noti per la loro affidabilità rispetto alle problematiche degli strati di rete sottostanti, in quanto costruiti sopra la porta 80 e protocollo HTTP, che nella maggioranza delle configurazioni non dovrebbe provocare problemi sistemistici. Questa astrazione generalmente pesa sulle prestazioni di rete: gli oggetti trasmessi introducono un nuovo overhead per i dati di protocollo. Nel nostro caso, non dovrebbero verificarsi problemi con connessioni cellulari di ultima generazione (UMTS).

### 3.7.2 Viste HTML

L'approccio per risorse consegue nell'omogeneizzazione delle pagine web necessarie rispetto ai metodi offerti: buona parte delle interfacce è dedicata all'interazione con il sistema secondo l'approccio REST. Questo approccio garantisce un ottimo livello di uniformità delle pagine e quindi la possibilità di raggruppare buona parte del codice necessario in elementi parziali riutilizzabili per composizione. Esistono comunque pagine con necessità particolari per quali è necessario codice dedicato.

**Composizione** Le viste web sono composte da documenti HTML, eseguibili Javascript e fogli di stile. Possiamo considerare le viste web come sorgenti da consegnare al browser dinamicamente per un corretto utilizzo dell'interfaccia. Nel nostro progetto le viste web sono generate da RoR e consegnate al browser richiedente. Il codice HTML è generato dinamicamente con un sistema di layout. I fogli di stile e il codice javascript è aggregato, minificato, identificato e consegnato. Tutte le precedenti operazioni sono svolte dal framework e/o librerie aggiuntive.

**Decorazione** Il codice delle viste web è costruito tramite *embedding* di codice Ruby all'interno di codice HTML tramite tag specifici. Il codice prodotto dovrebbe rispecchiare al massimo livello possibile l'effettivo contenuto della pagina, ma spesso non è così a causa della necessità di definizione di metodi per la preparazione dei dati. Questi metodi rappresentano una dipendenza stretta tra i controller e le view. Le classi decorative raggruppano i metodi "di interfaccia" dei controller, liberandoli dalle responsabilità di "decorare" i dati e inserendo un'interfaccia logica tra i due componenti.

**Assets** Gli assets sono i contenuti esterni necessari al corretto funzionamento della webapp. Includono fogli di stile, codice eseguibile lato client, immagini e in generale contenuti non direttamente reperibili sul documento. Il processo di consegna di questo tipo di files in una situazione ordinaria è responsabilità del webserver. Nella versione 3 di RoR è stato definito un nuovo processo di consegna, descritto ad alto livello nei paragrafi successivi.

- **Aggregazione**

All'interno dei sorgenti dell'applicazione, sia i fogli di stile che il codice javascript (non le librerie) sono contenuti in file organizzati per categorie logiche e divisi per provenienza (codice applicazione, di libreria, o esterno). Nel momento in cui vengono richiesti dall'esterno, vengono unificati all'interno di un unico file (solo quelli di applicazione). Questo sistema permette di associare i sorgenti a categorie logicamente correlate, diminuendo l'entità logica delle componenti a favore della leggibilità del codice.

- **Minificazione**

I sorgenti javascript vengono minificati prima di essere consegnati: si risparmia banda.

- **Caching**

I browsers mantengono cache dei contenuti tramite il nome del file. Viene aggiunto un

suffisso ai nomi di file degli assets composto da un hash che cambia nel momento in cui cambiano i contenuti. In questa maniera i browser sono sempre al corrente di nuovi aggiornamenti e possono comportarsi di conseguenza.

**Fogli di stile e Javascript** L'obiettivo del gruppo didattico non è di tipo grafico/artistico. Lo scopo del progetto però rende di primo piano la necessità di sviluppare un'interfaccia semplice e gratificante. Il gruppo quindi ha deciso di appoggiarsi ad un framework esterno per ottenere una buona baseline grafica, sulla quale lavorare durante le ultime iterazioni. Questo ci permette di:

- Lavorare in modo incrementale sulle funzionalità, senza doversi preoccupare delle necessità grafiche.
- Dedicare maggior tempo allo sviluppo di funzionalità del software, che riteniamo più importanti sia per la nostra formazione che per il proponente.
- Delegare al framework la risoluzione di errori esterni dovuti a implementazioni particolari di browser (attività che non riteniamo formativa).

La personalizzazione del framework è eseguita tramite l'override del suo codice. Questo è un approccio standard, reso semplice da sistema di organizzazione degli assets descritto precedentemente. L'aggiornamento della libreria quindi non comporterà la riscrittura delle modifiche, ma solo un adattamento alle nuove funzionalità.

**Bootstrap** Bootstrap è un framework giovanissimo per lo specifico sviluppo di applicazioni web, che riteniamo una buona scelta per il progetto corrente. Contiene elementi di stile e codice javascript per l'animazione degli elementi. Rimandiamo ai riferimenti informativi per la visione delle sue funzionalità.

**JQuery** JQuery è una libreria per javascript che useremo principalmente per le richieste AJAX. Esiste inoltre una buona comunità che rende disponibile codice per le più disparate funzioni.

### 3.7.3 Layouts

A fine di evitare la duplicazione di codice ripetitivo è stato utilizzato il sistema di inclusione di elementi parziali reso disponibile dal framework. Abbiamo valutato come elementi ripetitivi dell'interfaccia:

- Menù di navigazione
- Footer
- Interfacce CRUD
- Vari elementi di ordine inferiore

Nonostante la possibilità di creare varie tipologie di layout assegnandole a diverse pagine e creare layout innestati, abbiamo scelto, almeno per la prima baseline interna, di mantenerne uno solo per uniformità.

## 3.8 Implementazione REST

E' possibile considerare l'applicativo come il "corpo" per l'esecuzione dei metodi delle risorse. Ogni risorsa, per essere correttamente funzionante, deve essere dichiarata come esistente, definire le specializzazioni dei propri metodi, definire i modi con i quali persiste nel database e avere tutto il necessario per costruire le rappresentazioni.

### 3.8.1 Struttura di ogni risorsa

Per ogni *risorsa REST* verrà generata una *tripla*, come descritto:

**classe Model:**

- Situata in /app/models
- Derivata da ActiveRecord o ActiveModel
- Nome *NomeEntità*
- Nome contenitore *NomeEntità.rb*
- Contiene le validazioni per i dati da registrare su database
- Definisce le relazioni con altri modelli
- Contiene eventuali metodi di pre elaborazione dati grezzi (possibilmente non li contiene)

**classe Controller:**

- Situata in /app/controllers
- Derivata da ApplicationController
- Nome *NomeEntitàController*
- Nome contenitore *NomeEntità\_controller.rb*
- Contiene la definizione dei soli metodi CRUD relativi alla risorsa REST, corrispondenti alle view
- Ogni metodo prevede le tipologie possibili di view da consegnare
- Ogni metodo discrimina le richieste entranti generando le view coerentemente con i parametri ricevuti
- Ogni metodo definisce i requisiti di accesso dell'utente richiedente, ed eventualmente notifica il fallimento

**classe View:**

- Ogni metodo che lo prevede presente nella classe controller possiede una view situata nel contenitore in /app/views/*NomeEntità*/*NomeMetodo*.*TipoView*.erb
- Il codice delle view è semanticamente corretto rispetto alla view che verrà generata (Valutare l'utilizzo della classe decoratrice)
- I sorgenti usano se necessario i metodi forniti dal framework per bypassare la duplicazione di codice (partials).

**eventuale classe Decorator:**

- Situata in /app/decorators
- Nome *NomeEntitàDecorator*
- Nome contenitore *NomeEntità\_decorator.rb*
- Definisce metodi per la preparazione alla presentazione nelle view
- I metodi della classe devono garantire una singola chiamata nel codice della view per un certo compito

## 4 Sottosistema Desktop

Con sottosistema Desktop si intende il tool di notifiche desktop che dovrà essere a disposizione di ogni Desktop-user. Il tool ha l'obiettivo di controllare periodicamente, a login eseguito, la presenza di notifiche riguardanti uno specifico user.

Le notifiche riguarderanno la presenza di nuove quest da svolgere da parte dell'utente.

### 4.1 Diagramma dei componenti

Viene ora presentato il diagramma dei componenti riguardante il sottosistema Desktop. Saranno indicate soltanto le relazioni tra i componenti, mentre le relazioni tra le classi saranno trattate nel diagramma delle classi in figura 22.

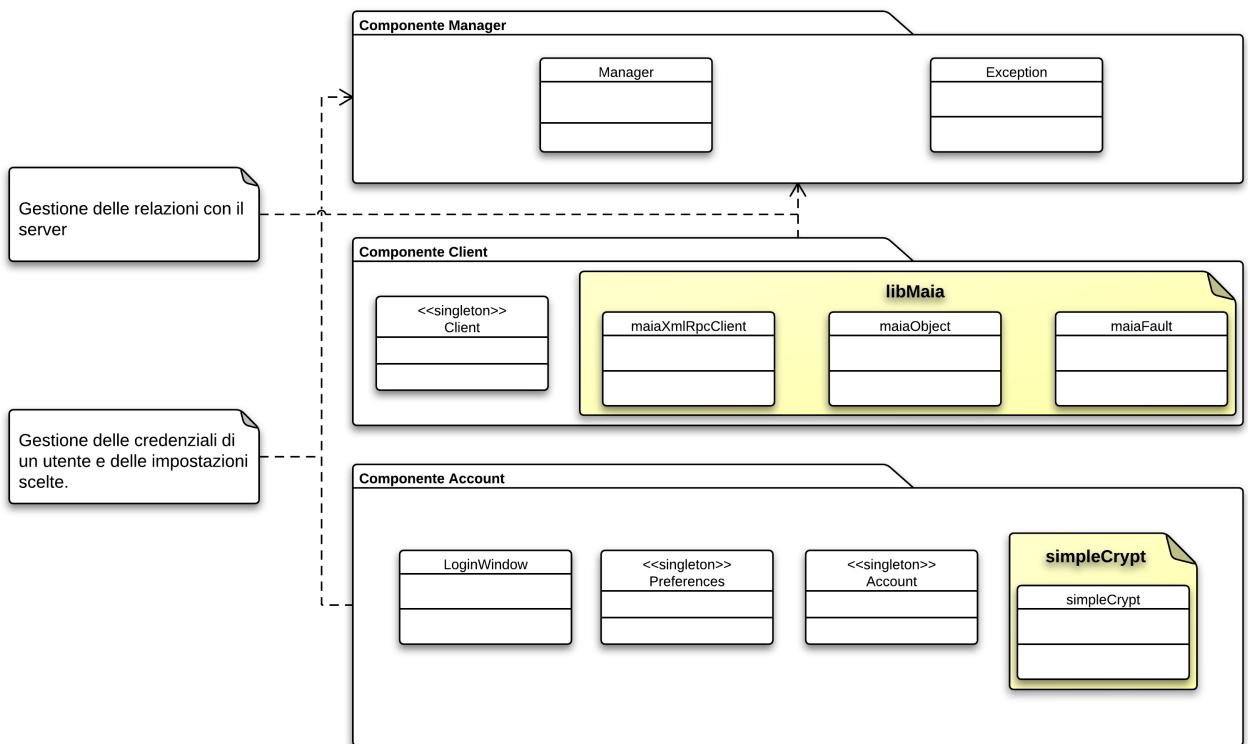


Figura 18: Desktop, diagramma dei componenti

Di seguito sono descritti singolarmente i vari componenti riguardanti il sottosistema desktop.

- **Componente Manager:**

ha il compito di gestire il sottosistema desktop per quanto riguarda la ricezione di nuove notifiche, interagendo con gli altri componenti del sottosistema e coordinandoli tra di loro. Si occupa inoltre della gestione di eventuali eccezioni ed errori previsti che possono essere sollevati.

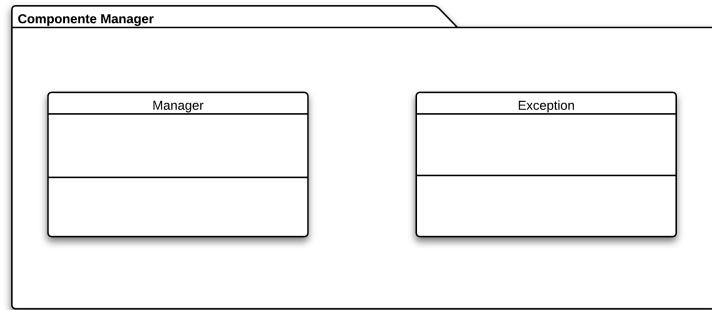


Figura 19: Desktop, componente Manager

- **Componente Account:**

ha il compito di gestire l'interazione dell'utente con l'applicazione.

Il componente infatti si occupa di amministrare:

- l'inserimento da parte di un Desktop-user delle proprie credenziali di accesso quali e-mail e password attraverso una semplice form per il login,
- il salvataggio di tali credenziali e della criptazione della password, qualora l'utente decida di rimanere loggato al sistema,
- le preferenze scelte dall'utente sulle impostazioni dell'applicazione.

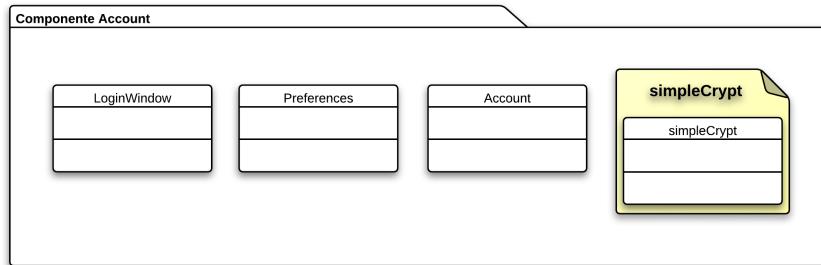


Figura 20: Desktop, componente Account

- **Componente Client:**

ha il compito di gestire l'autenticazione di un Desktop-user al server PMAC attraverso l'invio al server delle credenziali di accesso e la ricezione delle risposte del server. Inoltre ad autenticazione avvenuta invia periodicamente richieste al server per il controllo della presenza di nuove notifiche.

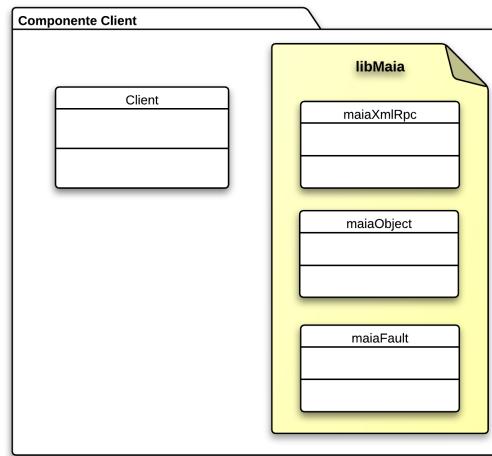


Figura 21: Desktop, componente Client

#### 4.2 Diagramma delle classi

Di seguito viene riportato il diagramma UML delle classi riguardante il sottosistema Desktop. In tale diagramma per ogni classe sono specificati soltanto una parte dei campi dati e i metodi principali, lasciando la trattazione dell'implementazione completa al documento *"Definizione di Prodotto"*.

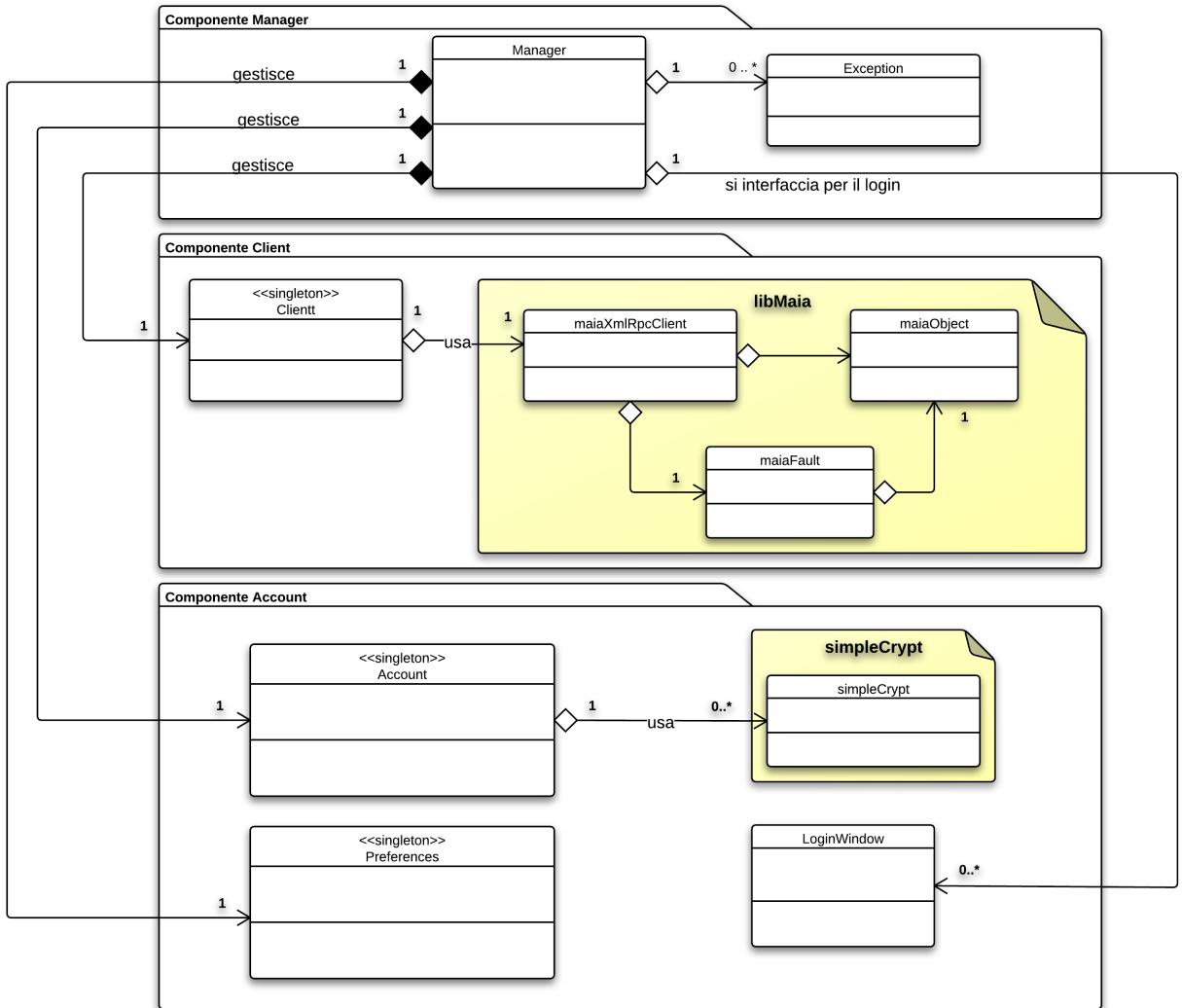


Figura 22: Desktop, diagramma delle classi

### Componente Manager

- **Classe Manager:**

Classe per la gestione del sottosistema desktop, coordina le relazioni tra le classi dei vari componenti.

Tra le varie funzionalità, la classe implementa la creazione della form per il login e l'eventuale logout con la cancellazione del file contenente le credenziali di accesso se queste erano state precedentemente salvate, e la chiusura dell'applicazione.

- **Classe Exception:**

Classe per la gestione di eccezioni o errori previsti sollevati durante l'esecuzione dell'applicazione.

### Componente Account

- **Classe LoginWindow:**

Classe raffigurante la finestra contenente la form per il login di un utente.

Permetterà l'inserimento delle credenziali di accesso quali e-mail e password e darà all'utente la possibilità di selezionare la memorizzazione di tali dati per il mantenimento dell'ac-

cesso in futuro. Tale classe farà inoltre un primo controllo sulla correttezza dei valori inseriti.

- **Classe Preferences:**

Classe che gestisce le impostazioni dell'applicazione: attraverso una interfaccia grafica implementata dalla classe, l'utente può configurare tali impostazioni.

- **Classe Account:**

Classe singleton che gestisce le credenziali di accesso di un utente caricandole virtualmente in memoria.

Tali credenziali sono lette da file se queste sono state memorizzate su scelta dell'utente ad un precedente accesso, altrimenti vengono caricate dalla form del login al momento dell'accesso.

Inoltre tale classe gestisce la scrittura di tali credenziali nell'apposito file per la memorizzazione.

- **Libreria SimpleCrypt:**

Semplice classe per la gestione della sicurezza sul salvataggio dei dati di autenticazione di un utente. Effettua la criptazione e decrittazione di stringhe di testo attraverso l'uso di una key.

- per il cripting riceve in input una stringa da criptare "in chiaro" e ne restituisce la stringa criptata.
- per il decrittazione riceve la stringa criptata e ne restituisce la rispettiva stringa decriptata.

## Componente Client

- **Classe Client:**

Classe singleton che implementa un client xml-rpc che gestisce in modo diretto le comunicazioni con il server, inviando a quest'ultimo i dati per il login, o richiedendo al server l'eventuale presenza di nuove notifiche.

Riceve quindi le risposte del server passandole alla classe Manager che provvederà alla loro gestione.

Le comunicazioni tra questa classe client e il server avvengono attraverso comunicazioni del tipo xml.

- **Libreria libMaia:**

All'interno del componente Client viene utilizzata una libreria per il supporto all'implementazione della classe Client per comunicazioni xml con il sottosistema Server. Tale libreria implementa le funzioni di basso livello necessarie al client.

### 4.3 Diagramma delle attività

In figura 23 viene fornito il diagramma delle attività riguardanti le operazioni disponibili ad un utente per l'applicazione desktop. Data la semplicità e i compiti dell'applicazione che deve unicamente visualizzare la presenza di nuove notifiche, le uniche attività significative possibili corrispondono al login e il logout al/dal sistema Server.

All'avvio dell'applicazione l'utente può effettuare il login al server PMAC attraverso l'inserimento delle credenziali di accesso mail e password. Se il server darà esito positivo l'utente sarà loggato al sistema e riceverà comunicazioni dell'eventuale presenza di notifiche, altrimenti se il server darà esito negativo, l'applicazione indicherà all'utente l'inesattezza delle credenziali inserite.

A login avvenuto l'utente potrà effettuare il logout dal server.

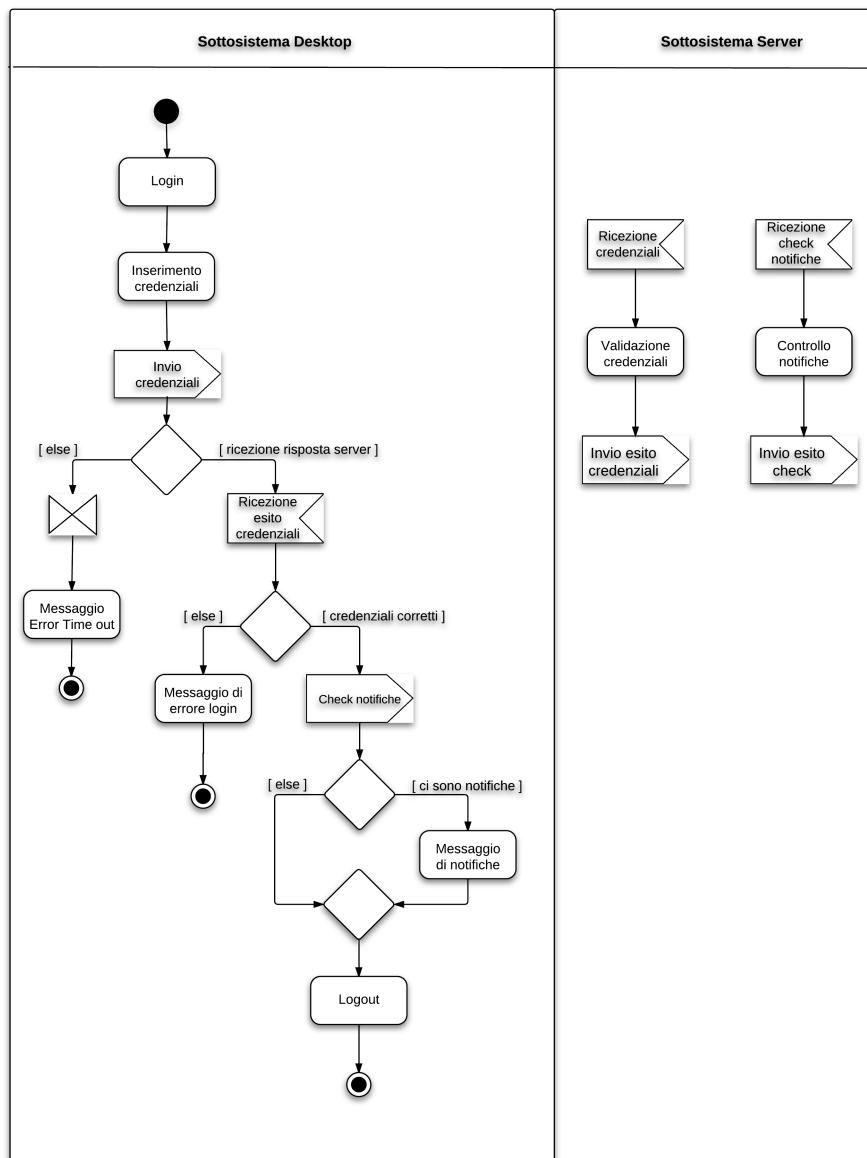


Figura 23: Desktop, diagramma delle attività

#### 4.4 Diagrammi di sequenza

Di seguito sono riportati i diagrammi di sequenza riguardanti le operazioni disponibili per il tool di notifiche desktop.

##### Login:

In figura 24 sono rappresentate le operazioni per il login di un Desktop-user al sistema PMAC. Alla richiesta di login da parte dell'utente viene visualizzata una form contenente degli appositi campi per l'inserimento delle credenziali. Alla conferma di login dell'utente, viene eseguito un primo controllo sui campi inseriti per testarne il corretto formato e se l'esito è positivo, il Manager recupera i valori inseriti nella form, e tramite il Client richiede il login al server. Successivamente il Client riceve la risposta del server: se la risposta è negativa il Manager elimina le credenziali di accesso virtualmente memorizzate e visualizza un messaggio di errore, altrimenti se la risposta è positiva le credenziali di accesso vengono memorizzate e viene visualizzato un messaggio di avvenuto login.

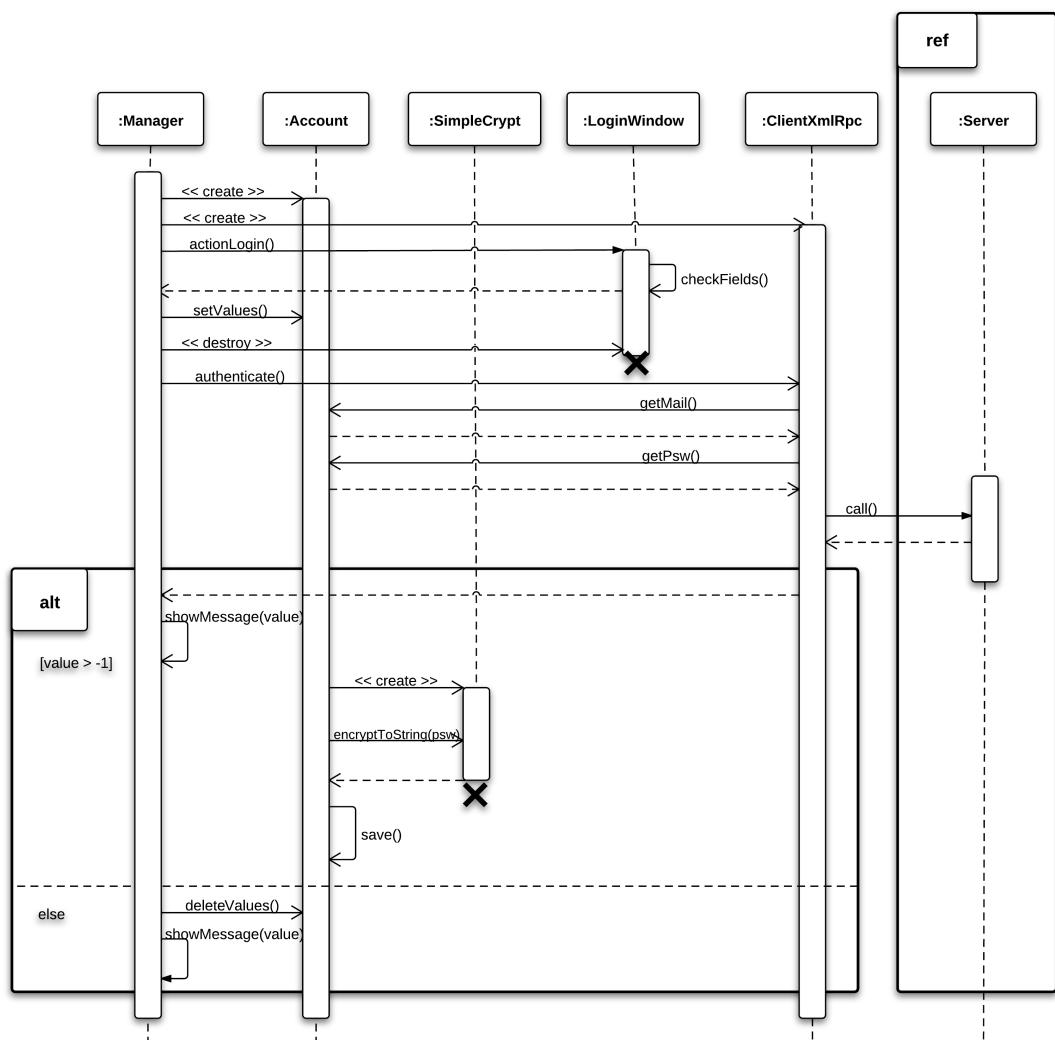


Figura 24: Desktop, diagramma di sequenza: login

**Logout:**

In figura 25 sono rappresentate le operazioni per il logout di un Desktop-user al sistema PMAC. Alla richiesta di logout di un utente, il Manager elimina le credenziali di accesso memorizzate durante la fase di login e visualizza un messaggio di avvenuto logout.

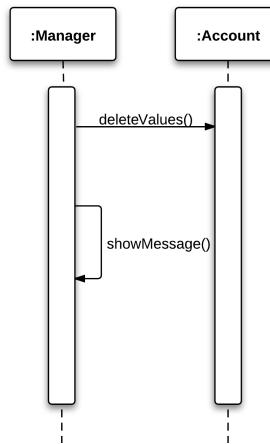


Figura 25: Desktop, diagramma di sequenza: logout

## 5 Sottosistema Mobile

Con sottosistema Mobile si intende la parte del sistema utilizzabile unicamente da dispositivo mobile ma che comunque offre all'utente un'esperienza completa con funzioni disponibili al pari di un utilizzatore desktop.

In particolare un utente mobile deve avere a disposizione un dispositivo con sistema operativo Android con versione almeno 2.2, dal quale potrà, dopo l'installazione dell'apposita applicazione, utilizzare il sistema, quindi principalmente affrontare quest, visualizzare e gestire profili e visualizzare le classifiche della gamification.

Il sistema prevede la segnalazione tramite notifiche push di nuove quest da svolgere, che quindi arriveranno all'utente utilizzatore senza bisogno che quest'ultimo ne controlli la presenza.

La limitazione del sistema mobile si riscontra nell'impossibilità di essere utilizzato dal Super-user, che potrà comunque utilizzarlo, ma come User, quindi senza le possibilità che sono a lui permesse dall'applicazione desktop.

### 5.1 Componenti del sottosistema

#### 5.1.1 Diagramma dei componenti

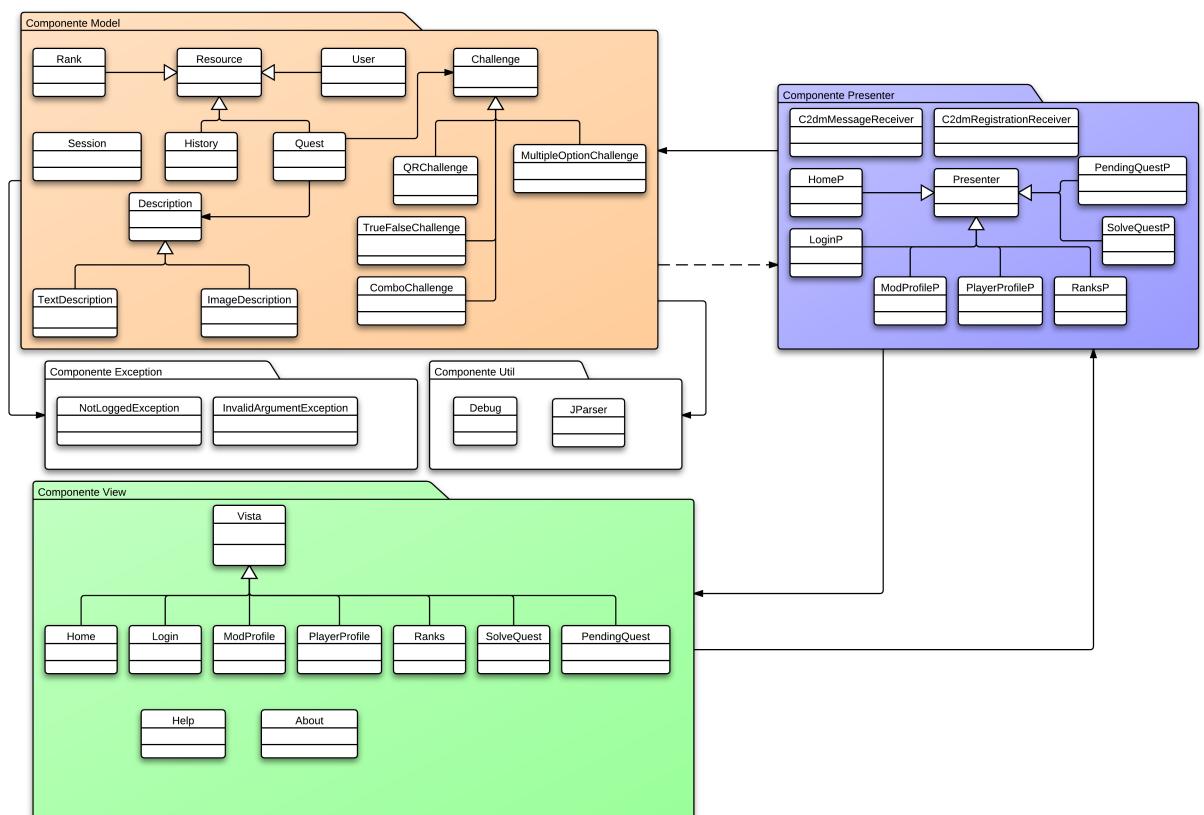


Figura 26: Mobile, diagramma dei componenti

#### 5.1.2 Descrizioni componenti

- **Componente Model:**

Rappresenta il Modello del pattern MVP; contiene tutte le classi relative al modello (User, Rank, Resource, Session) che verranno illustrate in seguito.

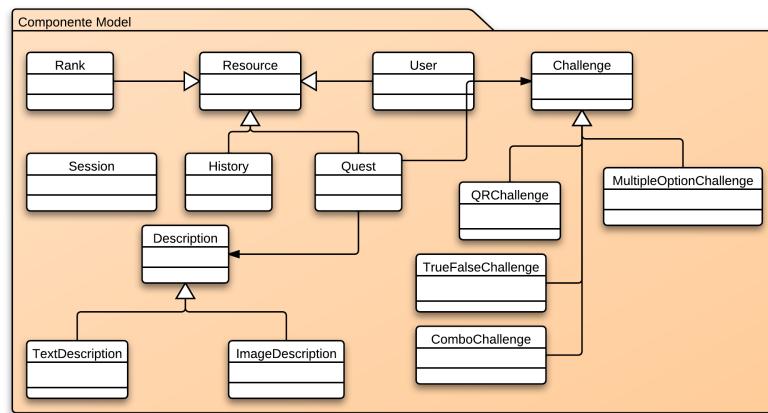


Figura 27: Mobile, componente Model

- **Componente Presenter:**

Rappresenta il Presenter del pattern MVP; risiede tra il componente Model e il componente View, il suo scopo è quello di inoltrare richieste provenienti da una view generica al model e notificare eventuali cambiamenti di stato di quest'ultimo. Può essere visto come *man in the middle*.

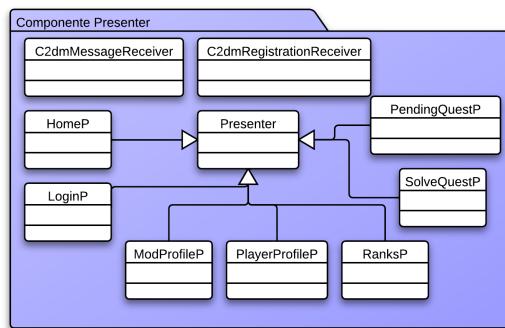


Figura 28: Mobile, componente Presenter

- **Componente View:**

Contiene le classi necessarie al dialogo con l'utente; quasi tutte queste classi ereditano da una classe comune **Vista**.

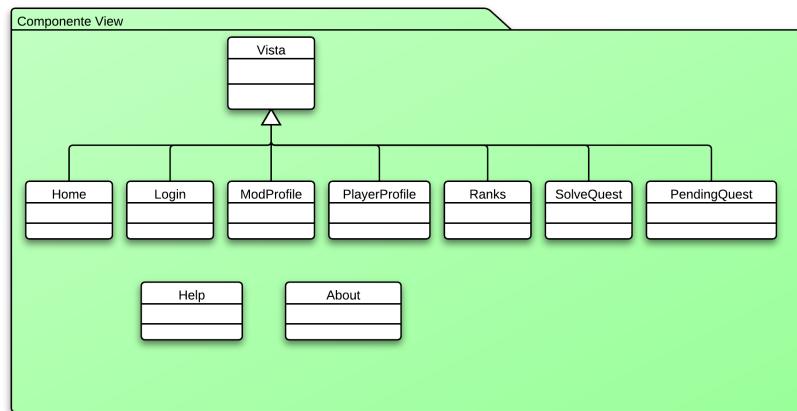


Figura 29: Mobile, componente View

- **Componente Util:**

Elenco di classi per fornire funzionalità generiche al sistema.

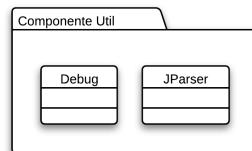


Figura 30: Mobile, componente Util

- **Componente Exception:**

Contiene le classi necessarie alla gestione delle eccezioni del sistema.

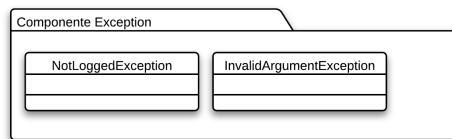


Figura 31: Mobile, componente Exception

## 5.2 Diagramma delle classi

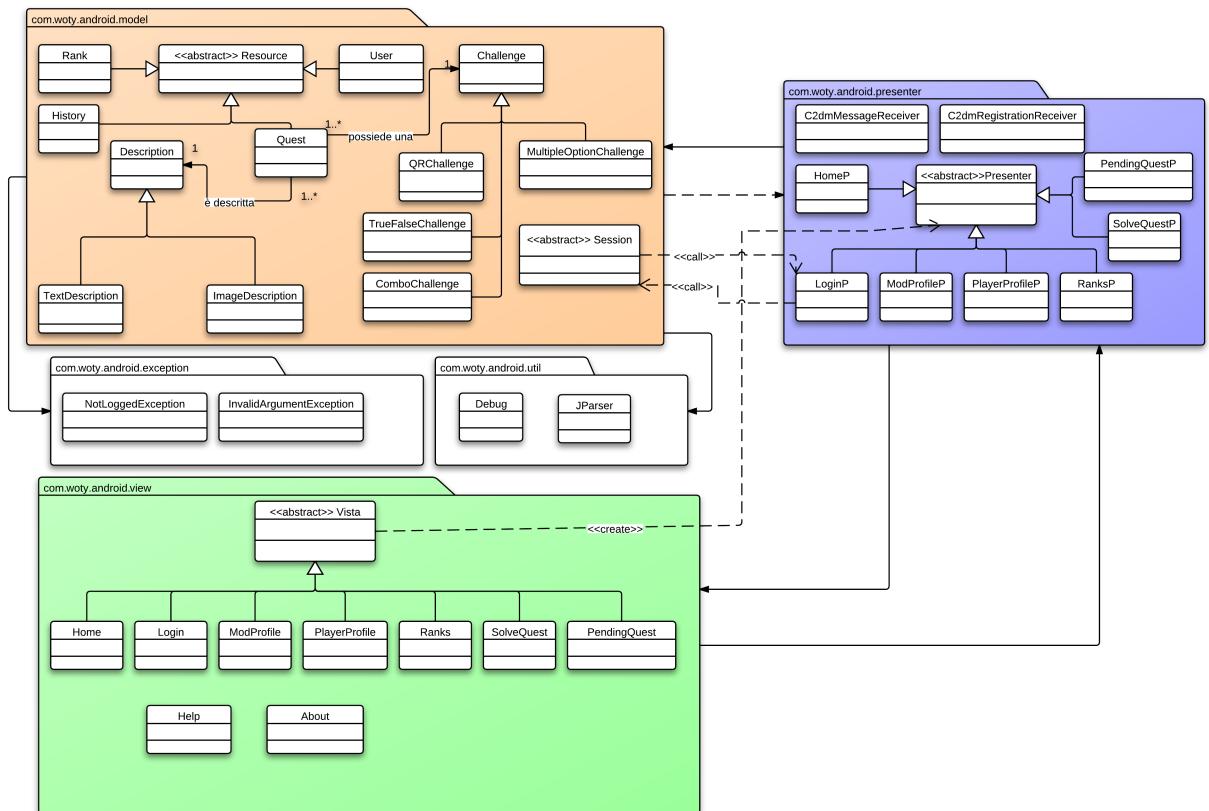


Figura 32: Mobile, diagramma delle classi

### Componente Model

- **Classe Resource:**

Classe astratta che rappresenta una risorsa generica; contiene i metodi principali `setResource` e `getResource` comuni a tutte le risorse.

- **Classe Rank:**

Eredita da `Resource`, corrisponde a una classifica implementata tramite un contenitore di `User`.

- **Classe User:**

Eredita da `Resource`. Identifica la classe responsabile della gestione del profilo di uno `User`. Contiene i campi dati necessari a identificare uno `User` generico e dialoga con la Classe `Session` per la gestione dell'autenticazione.

- **Classe Session:**

Si occupa della fase di autenticazione di un utente, creando una sessione che verrà salvata sul dispositivo. Permette la gestione delle sessioni, implementando operazioni quali `loadSession`, `eraseSession`, `saveSession`.

### Componente View

- **Classe Vista:**

Classe astratta che estende la classe `Activity`, al suo interno è presente un riferimento al

Presenter.

- **Classe Home:**

Sezione principale dell'applicazione, dalla quale saranno disponibili collegamenti alle altre sezioni. Nel momento della sua creazione, andrà a istanziare la corrispondente classe Presenter HomeP.

- **Classe Login:**

Permette l'operazione di login all'utente. Al momento della sua creazione istanzia il corrispettivo Presenter LoginP.

- **Classe ModProfile:**

Sezione nella quale sono disponibili operazioni per la modifica dei dati del proprio profilo. Istanzia il Presenter ModProfileP.

- **Classe PlayerProfile:**

Visualizza il profilo di uno User precedentemente scelto. Andrà a creare il corrispettivo PlayerProfileP.

- **Classe Ranks:**

Visualizza una classifica generica composta da User; selezionando uno User sarà possibile accedere al suo profilo. Istanzia il Presenter RanksP.

- **Classe SolveQuest:**

Svolgimento effettivo della quest, collegata al relativo Presenter SolveQuestP.

- **Classe Help:**

Si occupa di gestire messaggi di aiuto o quant'altro di utile per agevolare l'utilizzo dell'applicazione da parte dell'utente.

- **Classe About:**

Visualizza dati e informazioni tecniche dell'applicazione.

## Componente Presenter

- **Classe C2dmMessageReceiver:**

Classe che si occupa di gestire le notifiche inviate al dispositivo dal cloud (c2dm). Estende la classe BroadcastReceiver.

- **Classe C2dmRegistrationReceiver:**

Classe che registra un dispositivo verso il cloud (c2dm) quand'esso effettua il login. Estende la classe BroadcastReceiver.

- **Classe Presenter:**

Classe astratta che contiene un riferimento a Vista. Ogni Presenter riferito a una vista andrà a estendere questa classe.

- **Classe HomeP:**

Rappresenta il presenter della classe Home.

- **Classe LoginP:**  
Rappresenta il presenter della classe Login.
- **Classe ModProfileP:**  
Rappresenta il presenter della classe ModProfile.
- **Classe PlayerProfileP:**  
Rappresenta il presenter della classe PlayerProfile.
- **Classe RanksP:**  
Rappresenta il presenter della classe Ranks.

#### Componente Util

- **Classe Debug:**  
Classe utilizzata in fase di codifica per testare funzionalità generiche.
- **Classe Jparser:**  
Estrae informazioni da un messaggio Json.

#### Componente Exception

- **Classe NotLoggedException:**  
Utilizzata per indicare che un utente non è loggato nel sistema.
- **Classe InvalidArgumentException:**  
Indica erroneo passaggio di valori generici.

### 5.3 Diagramma delle attività

In figura 33 viene fornito il diagramma delle attività per l'applicativo mobile.  
L'applicazione mette a disposizione dell'utente un menù di navigazione principale dal quale è possibile accedere ad ogni sezione dell'applicativo.  
L'utente all'avvio dell'app effettua il login al server, e una volta loggato visualizza il proprio profilo.

L'utente successivamente attraverso il menù può:

- modificare il profilo, cambiando alcuni campi dati che verranno quindi aggiornati sul server,
- accedere alla sezione quest, dalla quale può svolgere quest nel caso in cui abbia quest in sospeso, oppure richiedere nuove quest da poter svolgere,
- visualizzare la classifica dei migliori utenti e la propria posizione. Di ognuno di questi utenti visualizzati, è possibile vederne il profilo.

In fine l'utente può eseguire il logout dal server.

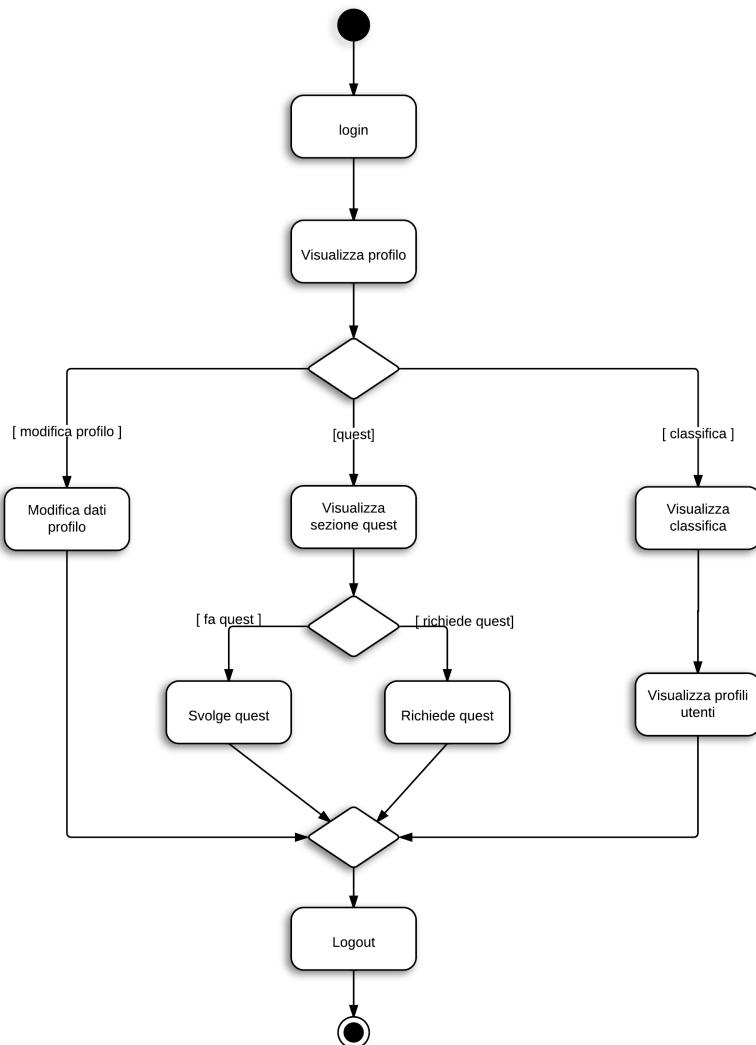


Figura 33: Mobile, diagramma delle attività

## 5.4 Diagrammi di sequenza

Di seguito sono riportati alcuni tra i diagrammi di sequenza che abbiamo ritenuto essere particolarmente importanti riguardo le operazioni disponibili per l'applicazione mobile.

### Login effettuato con successo:

Viene rappresentato il diagramma di sequenza di un'operazione di login andata a buon fine in ambito mobile. La classe Login invia una richiesta al suo Presenter LoginP, che la inoltrerà alla classe Session presente sul model. Quest'ultima dialogando con il server remoto restituirà l'esito positivo o meno dell'operazione.

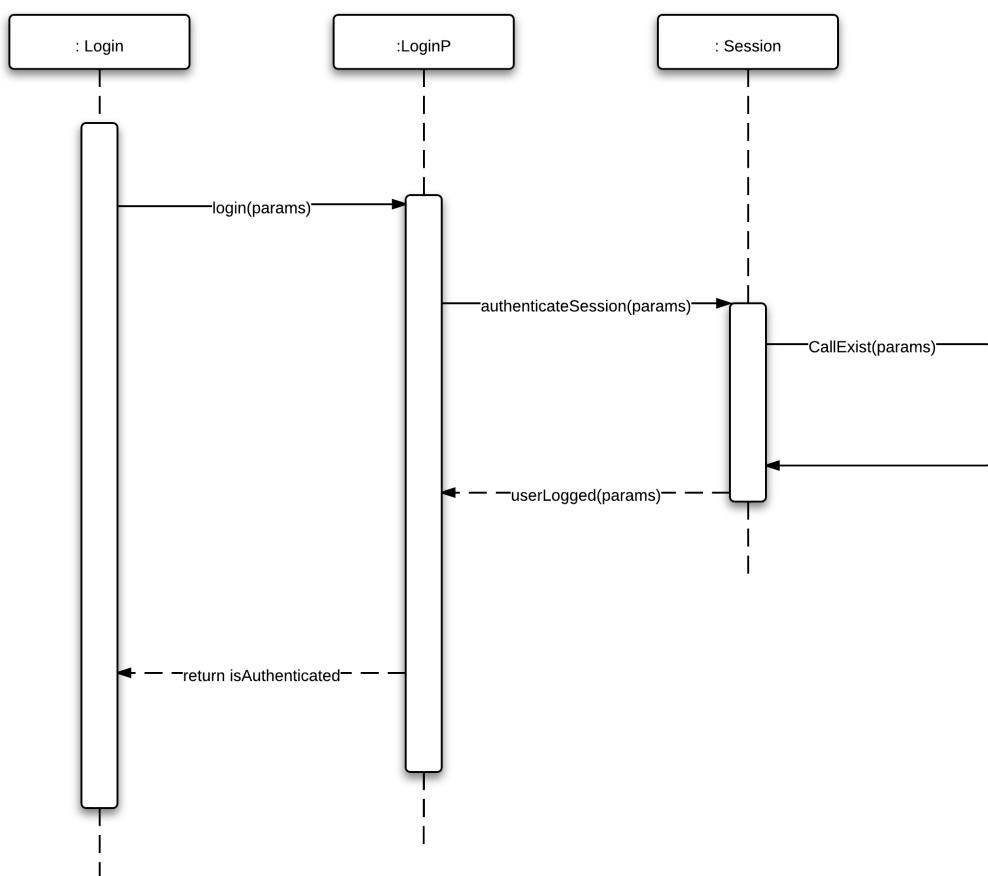


Figura 34: Login effettuato con successo

**Visualizza le quest in attesa di svolgimento:**

Quando una nuova quest viene assegnata a un utente mobile, questa verrà notificata all'applicazione. Una volta che l'utente entrerà nel pannello delle quest in attesa di svolgimento, l'applicazione richiederà al server la lista di quest'ultime.

PendingQuest richiede al suo Presenter PendingQuestP la lista delle quest destinate a quel dato utente. A sua volta PendingQuestP ne farà richiesta a HistoryQuest che andrà a prenderle dal server remoto.

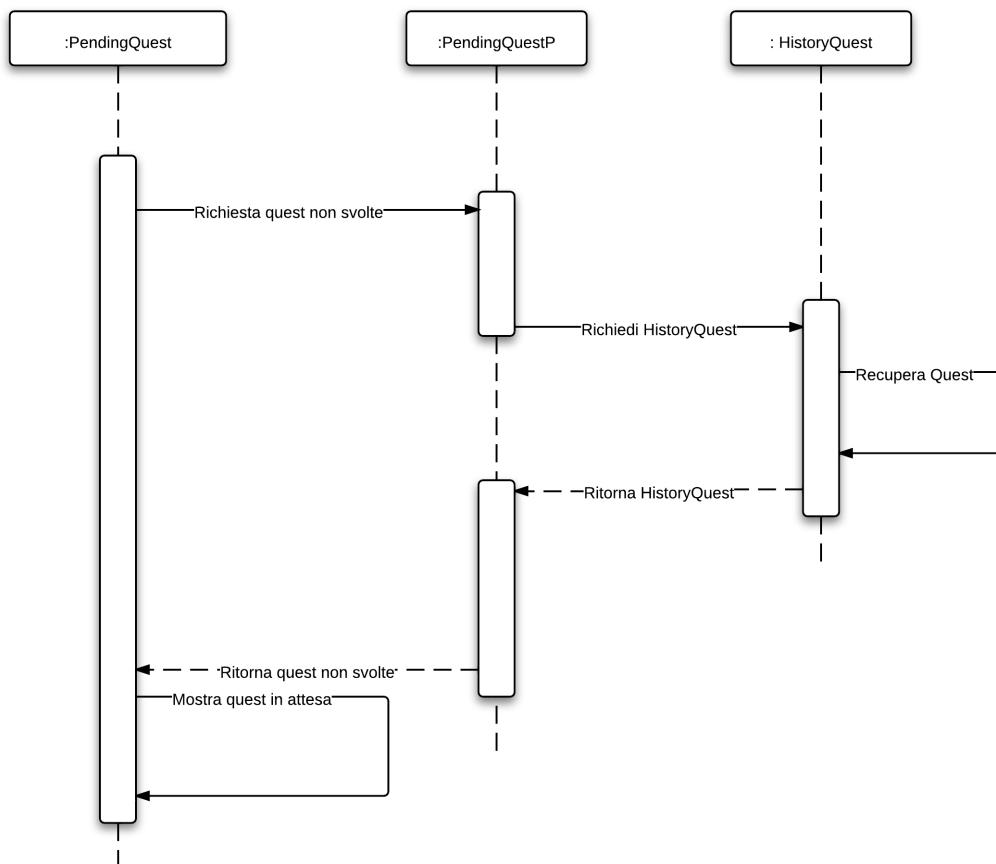


Figura 35: Recupera quest assegnate all'utente.

**Svolgi una quest:**

La classe Quest per essere istanziata richiede l'inizializzazione di due campi: uno di classe Description e uno di classe Challenge. Di conseguenza, durante la creazione di un oggetto Quest, saranno fatte due richieste al server remoto, una per ottenere l'oggetto Description e l'altra per ottenere l'oggetto Challenge.

Una volta terminata questa fase, l'oggetto Quest potrà essere correttamente istanziato e quindi visualizzato sulla relativa vista SolveQuest.

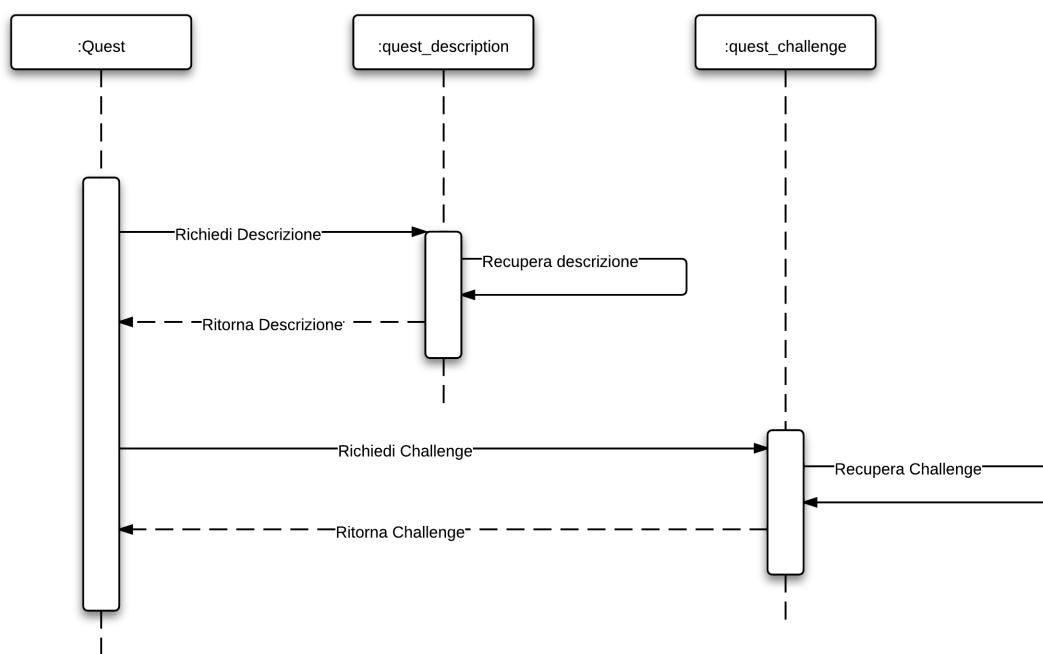


Figura 36: Recupera e svolgi una quest.

**Richiedi una classifica:**

Nel momento in cui un utente volesse visualizzare una particolare classifica dal pannello Ranks, verrà contattato il Presenter RanksP con i parametri richiesti. RanksP tenterà di istanziare la classe di modello Rank che recupererà le informazioni dal server remoto.

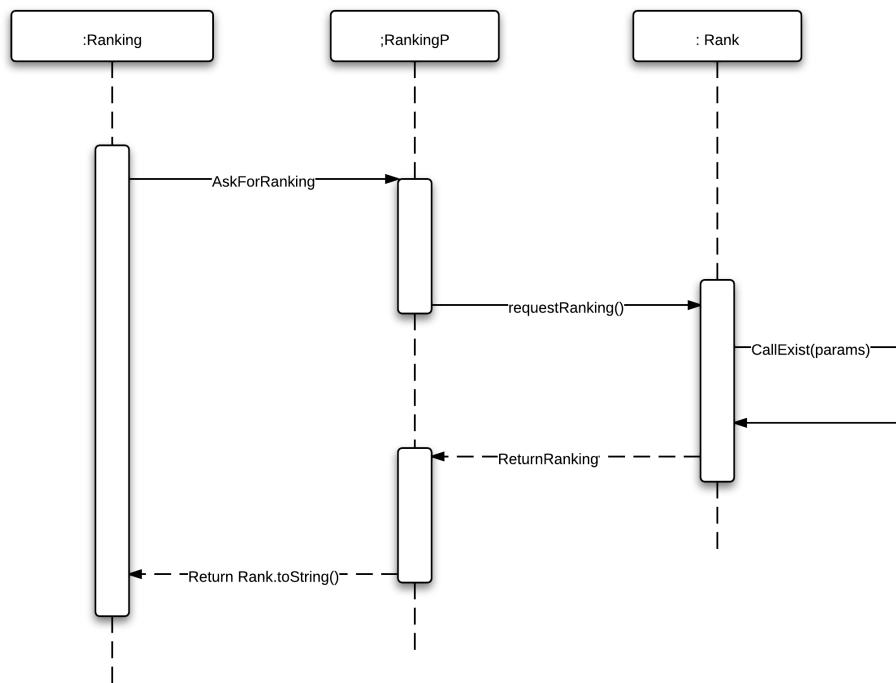


Figura 37: Richiesta di una classifica

## 6 Tracciamento relazioni componenti - requisiti

Di seguito sono illustrati attraverso delle tabelle le relazioni componenti-requisiti: per ogni componente o classe di un sottosistema sono specificati i rispettivi requisiti che questi soddisfano. Per quanto riguarda la notazione gerarchica utilizzata per identificare i requisiti si faccia riferimento al documento *NormeDiProgetto\_v2.pdf*, mentre per le specifiche indicazioni di ogni requisito si veda il documento *AnalisiDeiRequisiti.pdf*.

### 6.1 Sottosistema Server

Tabella 2: Server, tracciamento requisiti-classi

Codice Requisito	Classe
Fo-1	Admin Workgroup
Fo-2	Admin Customer
Fo-2.1	Admin Customer Achievement Badge EarnedBadge
Fo-2.1.1	Achievement
Fo-2.1.2	Badge
Fo-2.2	Admin Customer
Fo-2.2.1	Customer
Fo-2.2.2	Customer
Fo-2.2.3	Customer
Fo-2.2.4	Customer
Fo-2.2.5	Customer
Fo-2.2.6	Customer Workgroup
Fo-2.2.7	Customer SuperUser
Fo-2.3	Admin Customer
Fo-2.3.1	Customer
Fo-2.3.2	Customer
Fo-2.3.3	Customer
Fo-2.3.4	Customer
Fo-2.3.5	Customer
Fo-2.3.6	Customer Workgroup
Fo-2.3.7	Customer SuperUser
Fo-2.4	Admin Customer
Fo-2.5	Admin Ticket

(Continua alla pagina successiva)

(Continua dalla pagina precedente)

Fo-3	Admin Quest QuestCategory QuestChallenge QuestDescription
Fo-3.1	Admin Quest
Fo-3.1.1	MultiOptionChallenge ComboChallenge TrueFalseChallenge QRChallenge ImageQDescription VideoQDescription TextQDescription
Fo-3.1.2	Quest TextQDescription
Fo-3.1.3	Quest QuestChallenge
Fo-3.1.4	Quest QuestChallenge
Fo-3.1.5	Quest
Fo-3.1.6	Quest
Fo-3.1.7	Quest
Fo-3.2	Admin Quest Workgroup
Fo-3.3	Admin Quest
Fo-4	SuperUser
Fo-4.1	SuperUser
Fo-4.1.1	SuperUser MobileUser DesktopUser
Fo-4.1.2	User
Fo-4.1.3	User
Fo-4.1.4	User
Fo-4.1.5	User
Fo-4.1.6	User
Fo-4.1.7	User Workgroup
Fo-4.2	SuperUser User
Fo-4.2.1	SuperUser MobileUser DesktopUser
Fo-4.2.2	User
Fo-4.2.3	User
Fo-4.2.4	User
Fo-4.2.5	User
Fo-4.2.6	User
Fo-4.2.7	User Workgroup

(Continua alla pagina successiva)

(Continua dalla pagina precedente)

Fo-4.3	SuperUser User
Fo-4.4	SuperUser User
Fo-4.5	SuperUser
Fo-4.6	SuperUser Ticket
Fo-5	DesktopUser MultiOptionChallenge ComboChallenge TrueFalseChallenge
Fo-6	MobileUser MultiOptionChallenge ComboChallenge TrueFalseChallenge QRChallenge
Fo-9	User
Fo-10	User
Fo-10.1	User
Fo-10.1.1	User
Fo-10.1.2	User
Fo-10.1.3	User
Fo-10.2	User
Fo-10.3	User
Fo-10.3.1	User
Fo-10.3.2	User
Fo-10.3.3	User
Fo-10.3.4	User
Fo-10.3.5	User
Fo-10.4	User
Fd-1	User
Fd-2	User
Fz-2	Achievement Badge EarnedBadge

## 6.2 Sottosistema Desktop

Tabella 3: Desktop, tracciamento componenti-requisiti

Componente	Codice Requisito
Manager	Fo-7
Account	Fo-8
Client	Vo-2 Io-3 Io-5

Tabella 4: Desktop, tracciamento requisiti-componenti

Codice Requisito	Componente
Fo-7	Manager
Fo-8	Account

(Continua alla pagina successiva)

<i>(Continua dalla pagina precedente)</i>	
Vo-2	Client
Io-3	
Io-5	

### 6.3 Sottosistema Mobile

Tabella 5: Mobile, tracciamento componenti-classi-requisiti

Componente	Classe	Codice Requisito
View	Login Profile Ranking Quest PendingQuest	Fo-7 Fo-8.1, Fo-8.2, Fo-8.3 Fo-9 Fo-10.2, Fo-10.3, Fo-10.3.1, Fo-10.3.2, Fo-10.3.3, Fo-10.3.4, Fo-10.3.5, Fo-10.4 Fd-1 Fd-2 Io-4 Io-5
Model	User Rank Quest Description Challenge Resource	Fo-6 Fo-10, Fo-10.1, Fo-10.1.1, Fo-10.1.2, Fo-10.1.3, Fo-8.4
Presenter	C2DMMessagereceiver C2DMRegistrationReceiver	Fo-8 Io-5

Tabella 6: Mobile, tracciamento requisiti-componenti-classi

Codice Requisito	Componente	Classe
Fo-6	Model	Quest
Fo-7	View	Login
Fo-8	Presenter	C2DMMessagereceiver
Fo-8.1	View	SolveQuest
Fo-8.2	View	PendingQuest
Fo-8.3	View	PendingQuest
Fo-9	View	Rank
Fo-10	Presenter	UserP
Fo-10.1	Presenter	UserP
Fo-10.1.1	Presenter Model	UserP User
Fo-10.1.2	Presenter Model	UserP User
Fo-10.1.3	Presenter Model	UserP User
Fo-10.2	View	PlayerProfile
Fo-10.3	View	PlayerProfile
Fo-10.3.1	View	PlayerProfile
Fo-10.3.2	View	PlayerProfile
Fo-10.3.3	View	PlayerProfile

*(Continua alla pagina successiva)*

(Continua dalla pagina precedente)

Fo-10.3.4	View	PlayerProfile
Fo-10.3.5	View	PlayerProfile
Fo-10.4	View	PlayerProfile
Fd-1	View	PlayerProfile
Fd-2	View	PlayerProfile
Io-4	View	Login PlayerProfile Rank SolveQuest PendingQuest
Io-5	Presenter	C2DMMMessageReceiver

## 7 Stime di fattibilità e di bisogno di risorse

### 7.1 Modalità decisionali

Dall'analisi di progettazione architetturale svolta in merito al progetto, sono state avanzate più soluzioni tecniche a differenti problematiche. Si è quindi analizzato minuziosamente quali scelte effettuare, valutando ogni pro e contro.

Questa attività ha permesso di scegliere strumenti e tecnologie che il team ha ritenuto adeguate per ogni esigenza.

### 7.2 Conoscenze tecniche

Alcune tecnologie scelte erano conosciute da tutti i membri del gruppo, come HTML, Java, C++/Qt, per questo non vi saranno particolari difficoltà nelle implementazioni delle rispettive applicazioni strutturate con tali linguaggi.

Altre tecnologie, come Ruby e il framework Rails, oltre all'organizzazione delle applicazioni Android, hanno necessitato di un particolare approfondimento preventivo alla fase di progettazione.