



# Specifica Tecnica

## Informazioni sul documento

<b>Titolo documento</b>	Specifiche Tecniche
<b>Versione attuale</b>	v1.0.0
<b>Data versione attuale</b>	2012/01/31
<b>Data creazione</b>	2012/01/17
<b>Redazione</b>	Antonio Pretto Andrea Zironda Luca Guerra Luca Lorenzini Giacomo Lorigiola Umberto Dall'Est
<b>Revisione</b>	Stefano Faoro
<b>Approvazione</b>	
<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
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 . . . . .	10
2.3 Stili architetturali - 3-Tier . . . . .	10
2.3.1 Presentation . . . . .	11
2.3.2 Resource Oriented Architecture . . . . .	12
2.3.3 Logic . . . . .	12
2.3.4 Data . . . . .	13
2.4 Design patterns . . . . .	14
2.4.1 Model-View-Controller . . . . .	14
2.4.2 Active Record . . . . .	14
2.4.3 Decorator . . . . .	14
2.4.4 Observer . . . . .	14
2.4.5 Singleton . . . . .	14
2.5 Tecnologie . . . . .	14
2.5.1 Lato server . . . . .	15
2.5.2 Lato Desktop . . . . .	15
2.5.3 Lato Mobile . . . . .	15
2.6 Librerie Aggiuntive . . . . .	16
2.7 Implementazione . . . . .	16
2.7.1 Sottosistema Server: Webservice . . . . .	16
2.7.2 Sottosistema Desktop: Tool di notifica . . . . .	17
2.7.3 Sottosistema Mobile: Mobile client . . . . .	17
2.7.4 Web client . . . . .	18
<b>3 Sottosistema Server</b>	<b>19</b>
3.1 Modalità di descrizione . . . . .	19
3.2 Diagramma delle classi . . . . .	19
3.2.1 HttpResponse . . . . .	19
3.2.2 Resource . . . . .	20
3.2.3 Diagramma ad oggetti . . . . .	23
3.3 Diagrammi di sequenza . . . . .	24
3.3.1 Resource.create . . . . .	24

3.3.2	Resource.read . . . . .	25
3.3.3	Resource.delete . . . . .	26
3.3.4	Resource.update . . . . .	27
3.3.5	Resource.index . . . . .	28
3.3.6	Resource.new . . . . .	29
3.3.7	Resource.edit . . . . .	30
3.3.8	Richieste non-RESTful . . . . .	30
3.4	Lato Server - Descrizione tecnica . . . . .	30
3.5	Modello . . . . .	30
3.5.1	ActiveModel . . . . .	31
3.5.2	ActiveRecord . . . . .	31
3.6	Controllo . . . . .	31
3.6.1	ActionDispatch . . . . .	31
3.6.2	ActionController . . . . .	31
3.7	Vista . . . . .	32
3.7.1	Viste non HTML . . . . .	32
3.7.2	Viste HTML . . . . .	32
3.7.3	Layouts . . . . .	33
3.8	Implementazione REST . . . . .	34
3.8.1	Struttura di ogni risorsa . . . . .	34
<b>4</b>	<b>Sottosistema Desktop</b>	<b>36</b>
4.1	Diagramma dei componenti . . . . .	36
4.2	Diagramma delle classi . . . . .	38
4.3	Diagramma delle attività . . . . .	40
4.4	Diagrammi di sequenza . . . . .	40
<b>5</b>	<b>Sottosistema Mobile</b>	<b>43</b>
5.1	Componenti del sottosistema . . . . .	43
5.1.1	Diagramma dei componenti . . . . .	43
5.1.2	Descrizioni componenti . . . . .	44
5.2	Diagramma delle classi . . . . .	46
5.3	Diagramma delle attività . . . . .	49
5.4	Diagrammi di sequenza . . . . .	50
<b>6</b>	<b>Tracciamento relazioni componenti - requisiti</b>	<b>53</b>
6.1	Sottosistema Server . . . . .	53
6.2	Sottosistema Desktop . . . . .	55
6.3	Sottosistema Mobile . . . . .	55
<b>7</b>	<b>Stime di fattibilità e di bisogno di risorse</b>	<b>57</b>
7.1	Modalità decisionali . . . . .	57
7.2	Conoscenze tecniche . . . . .	57

# Elenco delle tabelle

2.1	HTTP/URI/CRUD	12
6.1	Server, tracciamento requisiti-classi	53
6.2	Desktop, tracciamento componenti-requisiti	55
6.3	Desktop, tracciamento requisiti-componenti	55
6.4	Mobile, tracciamento componenti-classi-requisiti	55
6.5	Mobile, tracciamento requisiti-componenti-classi	56

# Elenco delle figure

2.1 Client-Server . . . . .	9
2.2 3-tier . . . . .	11
2.3 Struttura generale del sistema . . . . .	16
2.4 Sottosistema Server . . . . .	17
2.5 Sottosistema Desktop . . . . .	17
2.6 Sottosistema Mobile . . . . .	18
3.1 Server - HttpResponse . . . . .	19
3.2 Server - Resource . . . . .	20
3.3 Server - Diagramma ad oggetti . . . . .	23
3.4 Server - Resource.create . . . . .	24
3.5 Server - Resource.read . . . . .	25
3.6 Server - Resource.delete . . . . .	26
3.7 Server - Resource.update . . . . .	27
3.8 Server - Resource.index . . . . .	28
3.9 Server - Resource.new . . . . .	29
3.10 Server - Resource.edit . . . . .	30
4.1 Desktop, diagramma dei componenti . . . . .	36
4.2 Desktop, componente Manager . . . . .	37
4.3 Desktop, componente Account . . . . .	37
4.4 Desktop, componente Client . . . . .	37
4.5 Desktop, diagramma delle classi . . . . .	38
4.6 Desktop, diagramma delle attività . . . . .	40
4.7 Desktop, diagramma di sequenza: login . . . . .	41
4.8 Desktop, diagramma di sequenza: logout . . . . .	42
5.1 Mobile, diagramma dei componenti . . . . .	43
5.2 Mobile, componente Appcore . . . . .	44
5.3 Mobile, componente Logic . . . . .	44
5.4 Mobile, componente ServerCommunication . . . . .	44
5.5 Mobile, componente C2DM . . . . .	45
5.6 Mobile, componente QRScan . . . . .	45
5.7 Mobile, diagramma delle classi . . . . .	46
5.8 Mobile, diagramma delle attività . . . . .	49
5.9 Login effettuato con successo . . . . .	50
5.10 Recupera nuova quest segnalata ma non presente nel dispositivo . . . . .	51
5.11 Recupera e salva quest nel dispositivo . . . . .	52
5.12 Richiesta di una classifica . . . . .	52

# 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.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:  
<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>

## 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. Per il diagramma ad oggetti a figura 3.3, 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.

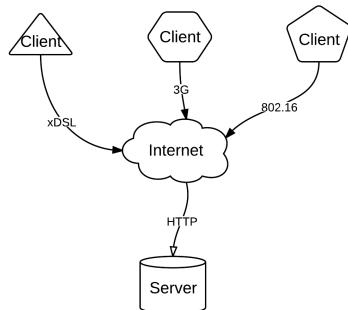


Figura 2.1: Client-Server

#### 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 computer serventi 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.

## Hosting esterno

L'appoggio a una struttura esterna renderebbe il problema più gestibile e delegherebbe a queste le responsabilità di garantire certe tipologie di servizi. Dall'altra parte non si avrebbe il completo controllo sulle implementazioni e la libertà di scegliere tutto l'hardware/software.

## Mantenimento presso il fornitore

Il mantenimento di server da parte del fornitore necessita la presenza di personale adeguato e di strutture costose. Questo garantirebbe la libertà di gestire le problematiche in casa quindi più rapidamente.

## Comunicazione

Le comunicazioni tra client e server possono avvenire su qualsiasi canale supporti il protocollo HTTP e garantisca la banda sufficiente ad un buon uso del software.

## HTTP

HTTP è il protocollo di base per la fruizione di contenuti web. Essendo suo supporto universale, non valutiamo possa creare dipendenze anche verso nuovi dispositivi. Esso è l'unico mezzo disponibile all'esterno tramite il quale i dispositivi client comunicano con i server.

### 2.2.3 Interfaccia

Secondo l'architettura 3-Tier<sup>9</sup> descritta nella sezione dedicata, 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"

## 2.3 Stili architetturali - 3-Tier

La logica applicativa è distinta in tre strati che comunicano tra di loro tramite interfacce, come indicato in figura 2.2 . La differenza di questo approccio rispetto a MVC<sup>9</sup>, oltre al dominio applicativo di più alto livello, è che lo strato presentation, vedendo solamente l'interfaccia dello strato logic, non interagisce mai direttamente con lo strato data: ogni comunicazione passa necessariamente tramite lo strato intermedio.

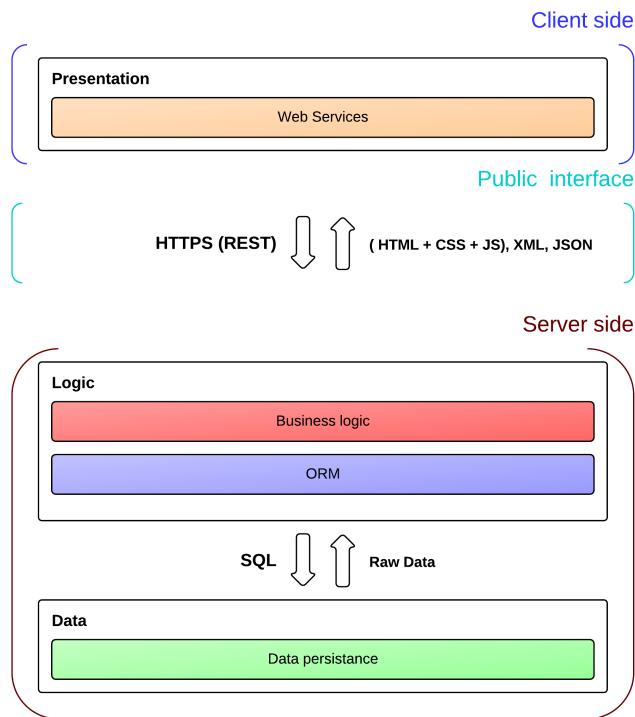


Figura 2.2: 3-tier

### 2.3.1 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.

### 2.3.2 Resource Oriented Architecture

Representational state transfer<sup>g</sup>, di seguito REST, è uno stile architettonico che permette la modellazione dei *business models* tramite *resources* (risorse). Le risorse sono rese accessibili tramite protocollo HTTP sottoforma 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 otteniamo:

- 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
- Stateless 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, mappa tramite uri non solamente i metodi CRUD ma anche le interfacce web necessarie a lanciarli (azioni index, show, edit).

Tabella 2.1: 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

### 2.3.3 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 descisivo 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 pressochè uniforme per ogni risorsa. Questo determina la facilità di estensione, mantenimento, sviluppo minimizzando la dipendenza dalle competenze dei singoli programmatore.

### 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>.

### 2.3.4 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.

## 2.4 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.

### 2.4.1 Model-View-Controller

Il pattern architettonale 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.

### 2.4.2 Active Record

Il pattern è già incluso nelle librerie offerte dal framework RoR, tramite subclassing della classe ActiveRecord. Si tratta di incapsulare 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.

### 2.4.3 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.

### 2.4.4 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.

### 2.4.5 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.

## 2.5 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.

### 2.5.1 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>g</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.

### 2.5.2 Lato Desktop

#### C++

Per la realizzazione del gestore delle notifiche lato Desktop si è deciso di utilizzare il linguaggio di programmazione Object Oriented<sup>g</sup> C++. La scelta di tale linguaggio è dovuta ad un buon bilanciamento tra portabilità ed efficienza.

- vantaggi:  
efficienza del linguaggio.
- svantaggi:  
gestione della memoria a cura del programmatore.

#### 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.

### 2.5.3 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>g</sup>, al fine di permettere alla quasi totalità dei dispositivi di ricevere notifiche push. Questo tipo di notifiche è reso possibile attraverso C2DM<sup>g</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.

## 2.6 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.

## 2.7 Implementazione

In figura 2.3 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.

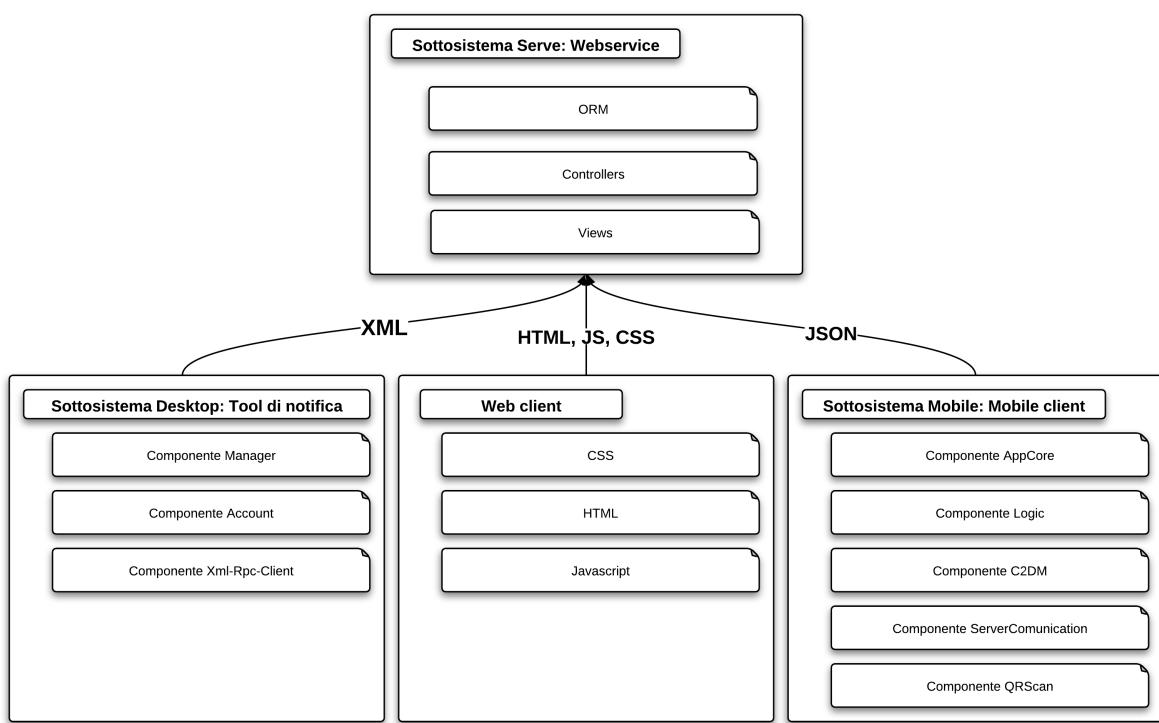


Figura 2.3: Struttura generale del sistema

### 2.7.1 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.

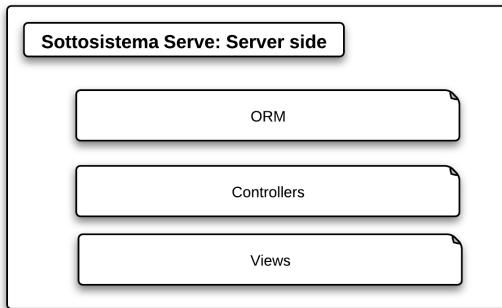


Figura 2.4: Sottosistema Server

### 2.7.2 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 2.5.

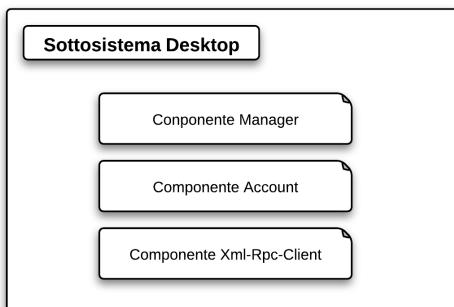


Figura 2.5: Sottosistema Desktop

### 2.7.3 Sottosistema Mobile: Mobile client

La parte dell'applicazione dedicata alla ricezione delle notifiche è indipendente dalle altre: una volta ricevuta una notifica viene aperta una connessione con il sottosistema server. Ad occuparsi di questa fase è il componente C2DM.

L'architettura dell'applicazione Android è basata sul design pattern MVC, i componenti in gioco sono AppCore per la View, Logic per il Controller e ServerCommunication per il Model. In questo caso il Model è remoto, cioè viene visto come una chiamata di ServerCommunication verso il sottosistema Server.

Il componente QRScan è dedicato alla lettura di un QR-Code, necessario per lo svolgimento delle quest speciali per la classe di utenti Mobile.

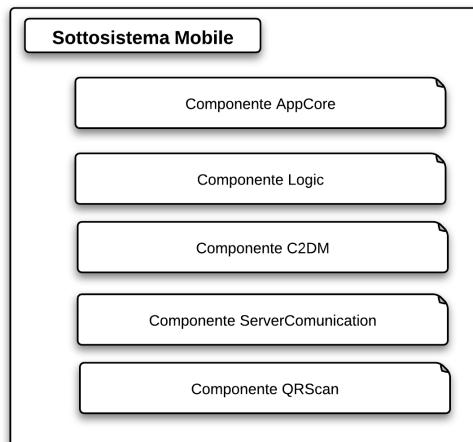


Figura 2.6: Sottosistema Mobile

#### 2.7.4 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

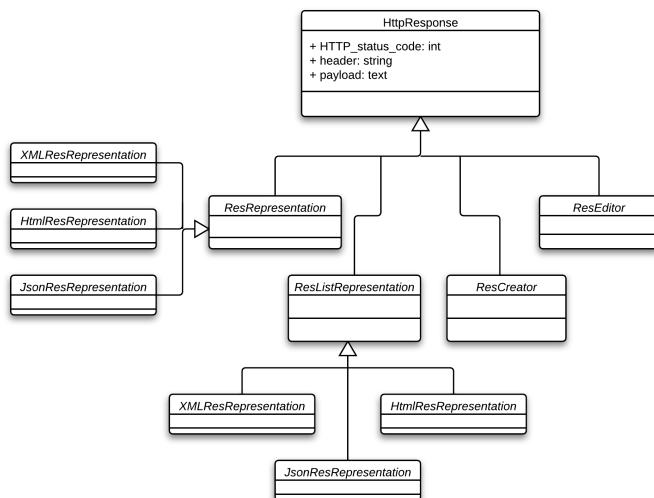


Figura 3.1: Server - HttpResponse

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 contenete una vista HTML con la quale è possibile creare una risorsa. Anche in questo caso, genericamente è una form.
- Classe ResListRepresentation: è una HttpResponse contente lista di istanze di una generica risorsa. Da essa ereditano le classe XMLResListRepresentation, HtmlResListRepresentation, JsonResListRepresentation le quali incapsulano la lista in, rispettivamente, XML, HTML, JSON.
- Classe ResRepresentation: è una HttpResponse contenete una vista HTML che visualizza una risorsa. Da essa ereditano le classe XMLResRepresentation, HtmlResRepresentation, JsonResRepresentation le quali incapsulano la risorsa in, rispettivamente, XML, HTML, JSON.

### 3.2.2 Resource

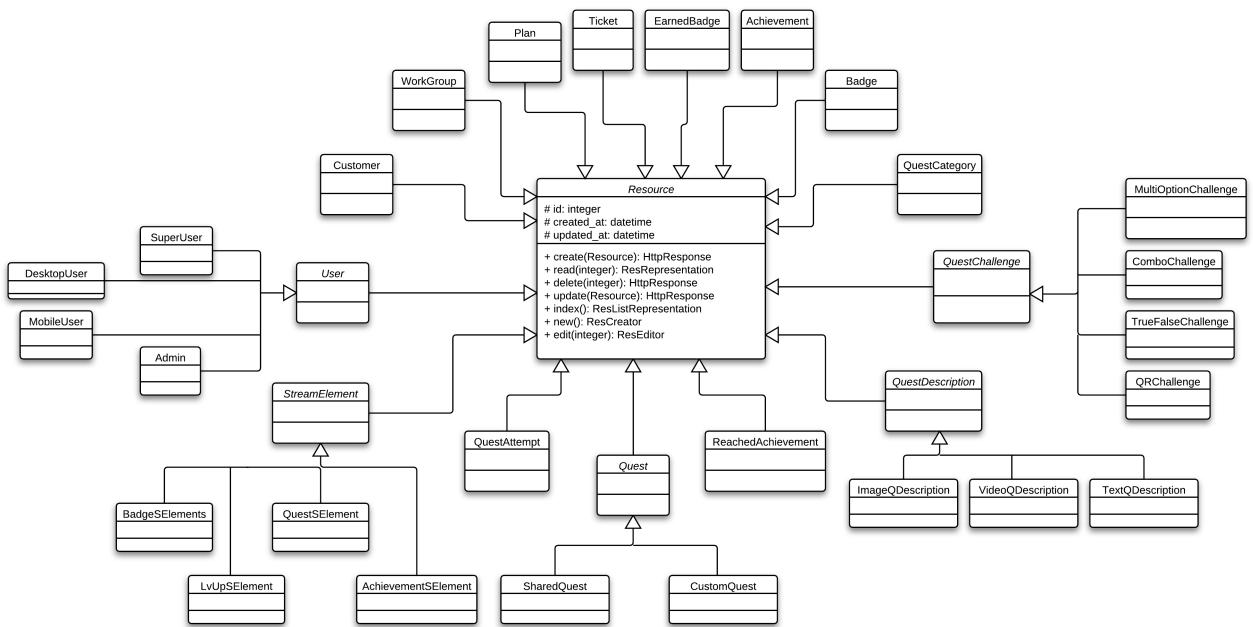


Figura 3.2: Server - Resource

Questo diagramma vuole mostrare come ogni classe logica disponibile nel webservice sia in realtà una risorsa. 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 `HttpServletResponse` 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 PMAX 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 consegue un determinato Achievement.

### 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.

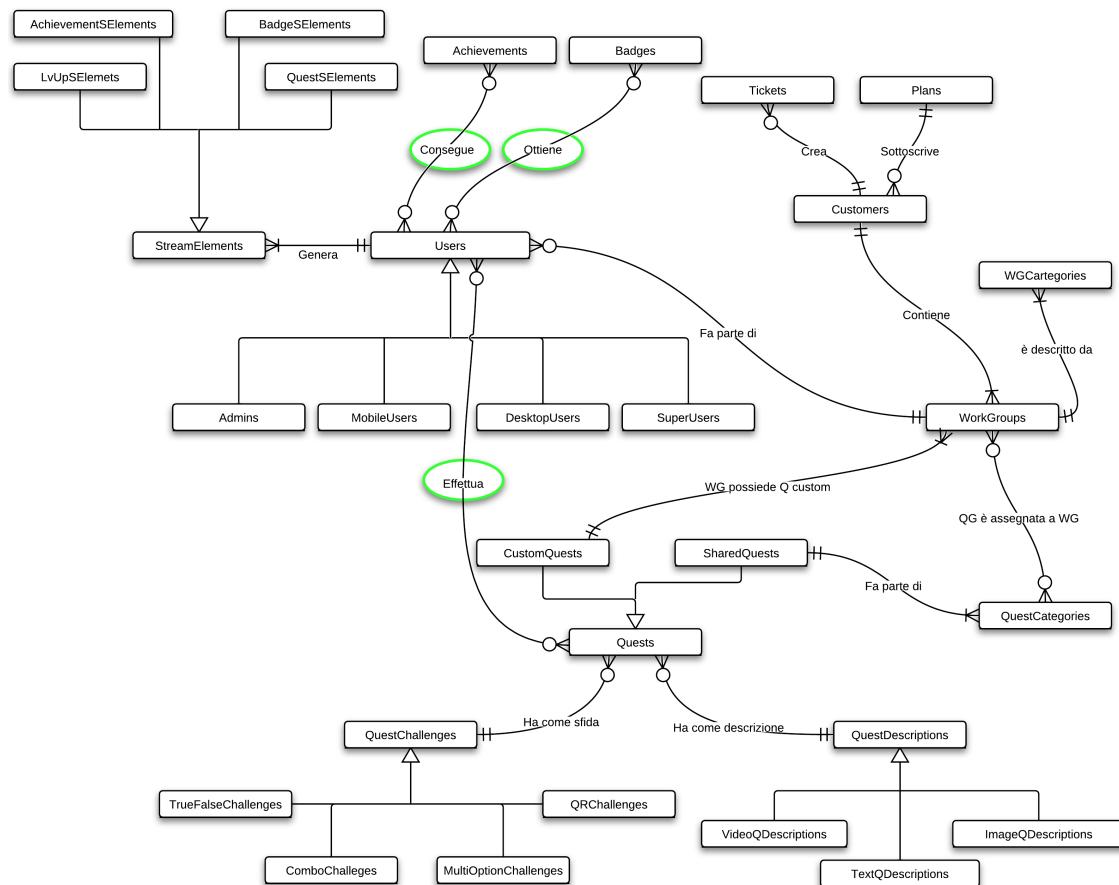


Figura 3.3: Server - Diagramma ad oggetti

Da notare il fatto che dalle relazioni N:M nascano 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 realizzazione 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 crei una nuova istanza di una risorsa.

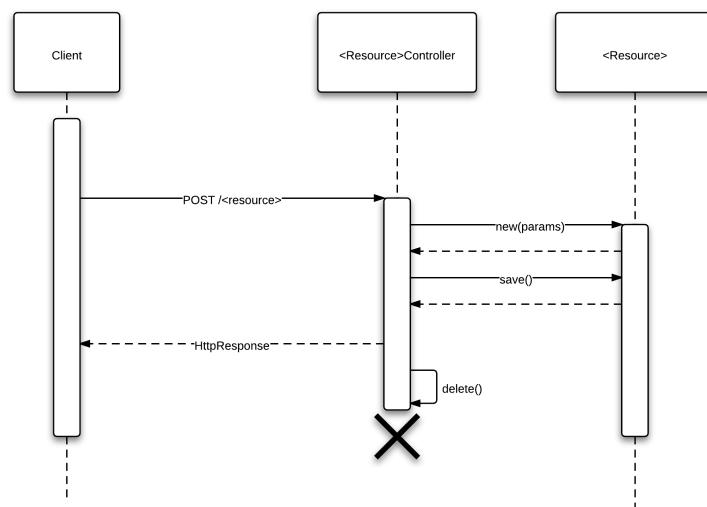


Figura 3.4: Server - Resource.create

### 3.3.2 Resource.read

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta per la lettura di una specifica risorsa, la cerchi nel modello e la restituisca al client mediante una ResRepresentation. Se la risorsa specificata non esiste, il webservice ritorna una HttpResponseMessage con un codice di stato rappresentante un errore.

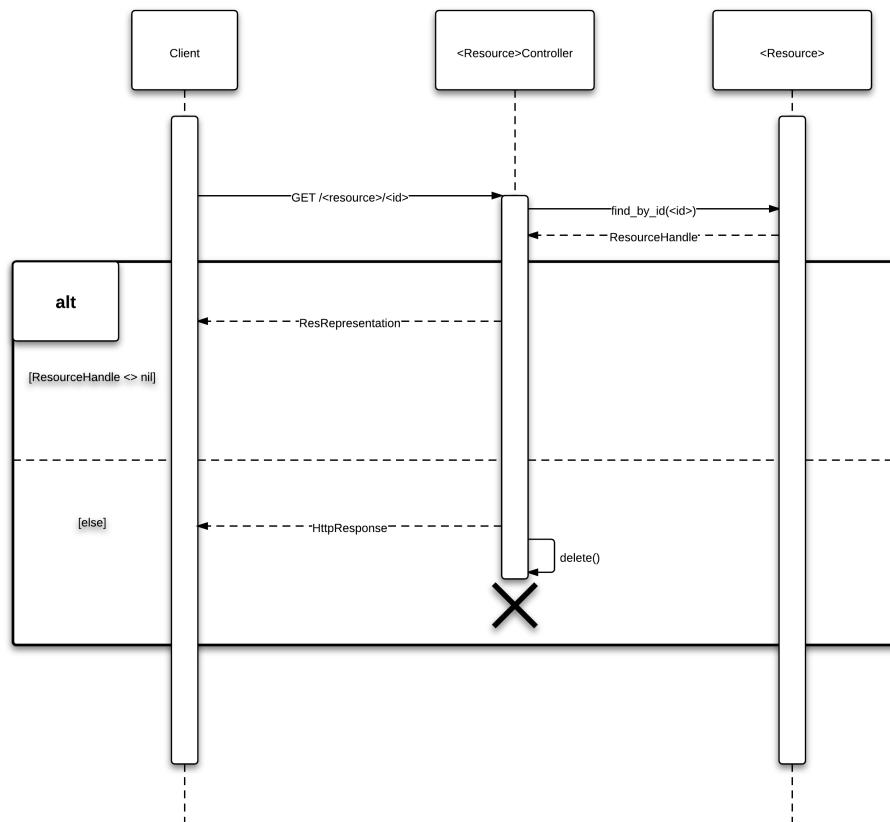


Figura 3.5: Server - Resource.read

### 3.3.3 Resource.delete

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta per la cancellazione di una specifica risorsa. La HttpResponse sarà caratterizzata dall'esito dell'operazione (se la risorsa specificata non esiste, avrà un codice di stato rappresentante un errore).

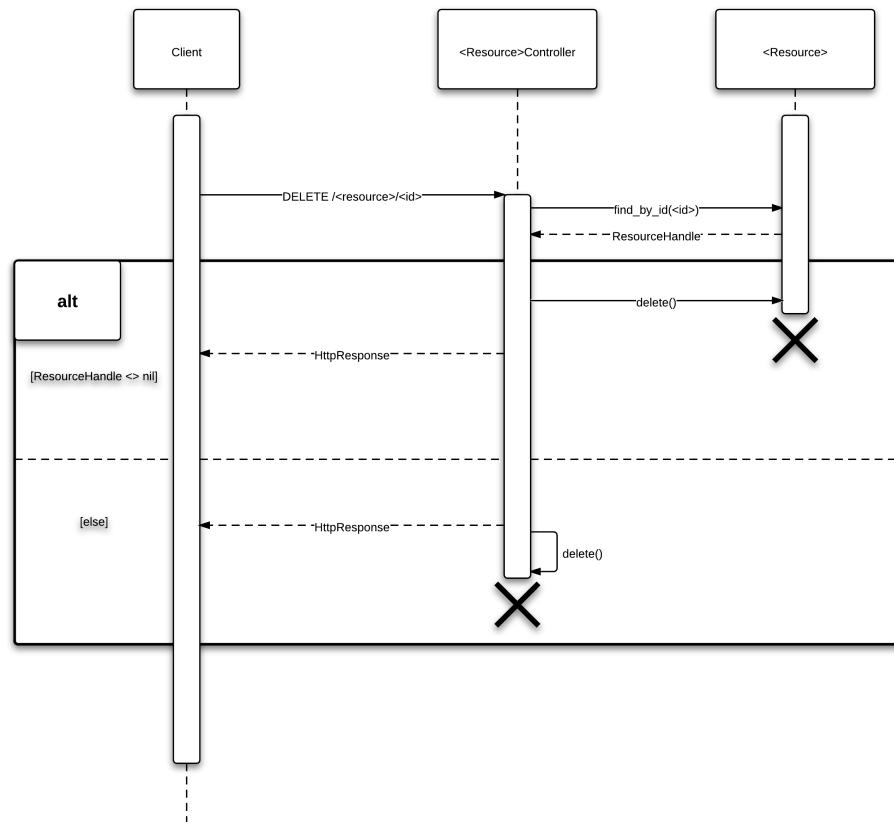


Figura 3.6: Server - Resource.delete

### 3.3.4 Resource.update

Il seguente diagramma di sequenza illustra come il webservice riceva una richiesta per l'update di una specifica risorsa. La `HttpResponse` sarà caratterizzata dall'esito dell'operazione (se la risorsa specificata non esiste, avrà un codice di stato rappresentante un errore).

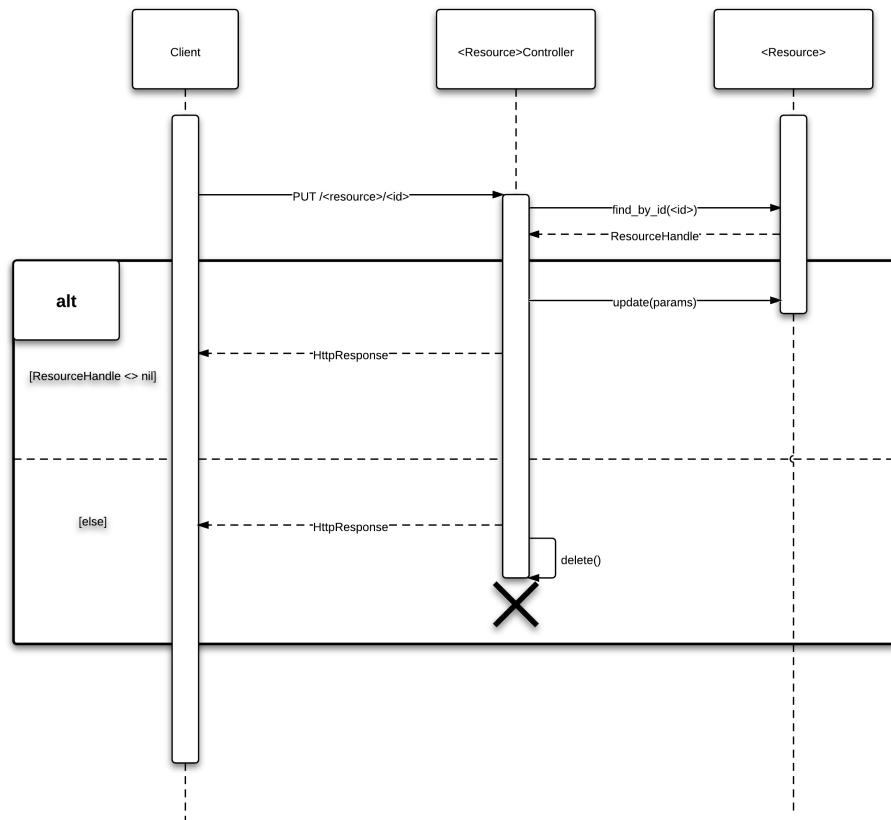


Figura 3.7: Server - Resource.update

### 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 webservice risponde al client con una ResListRepresentation.

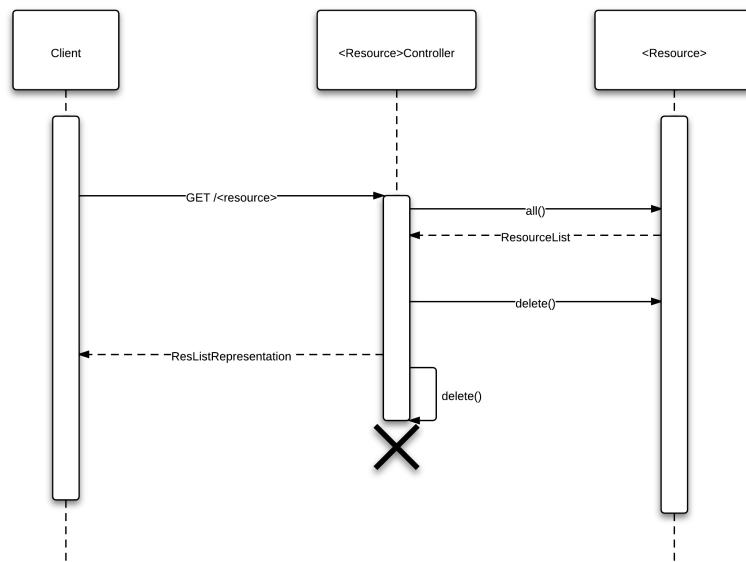


Figura 3.8: Server - Resource.index

### 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, e risponda con il relativo ResCreator.

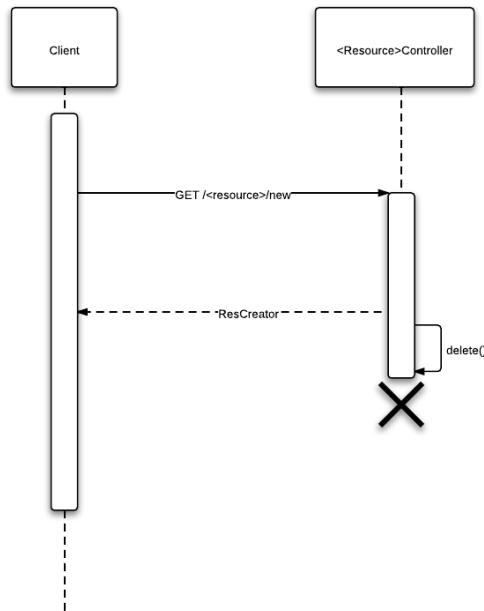


Figura 3.9: Server - Resource.new

### 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, e risponda con il relativo ResEditor.

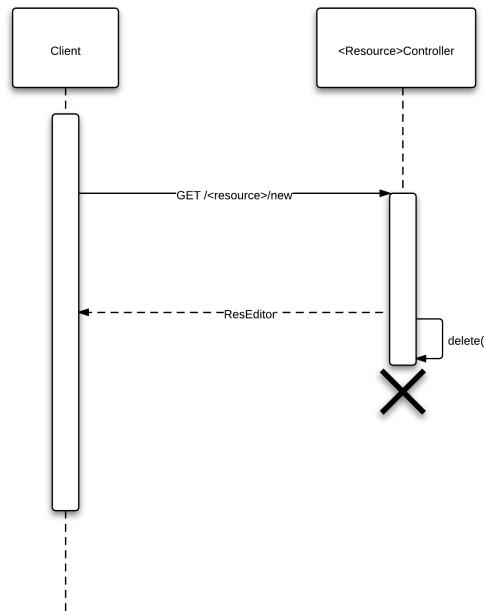


Figura 3.10: Server - Resource.edit

### 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.

### 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 microgestione 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 preelaborazione 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 *NomeEntità*Decorator
- 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

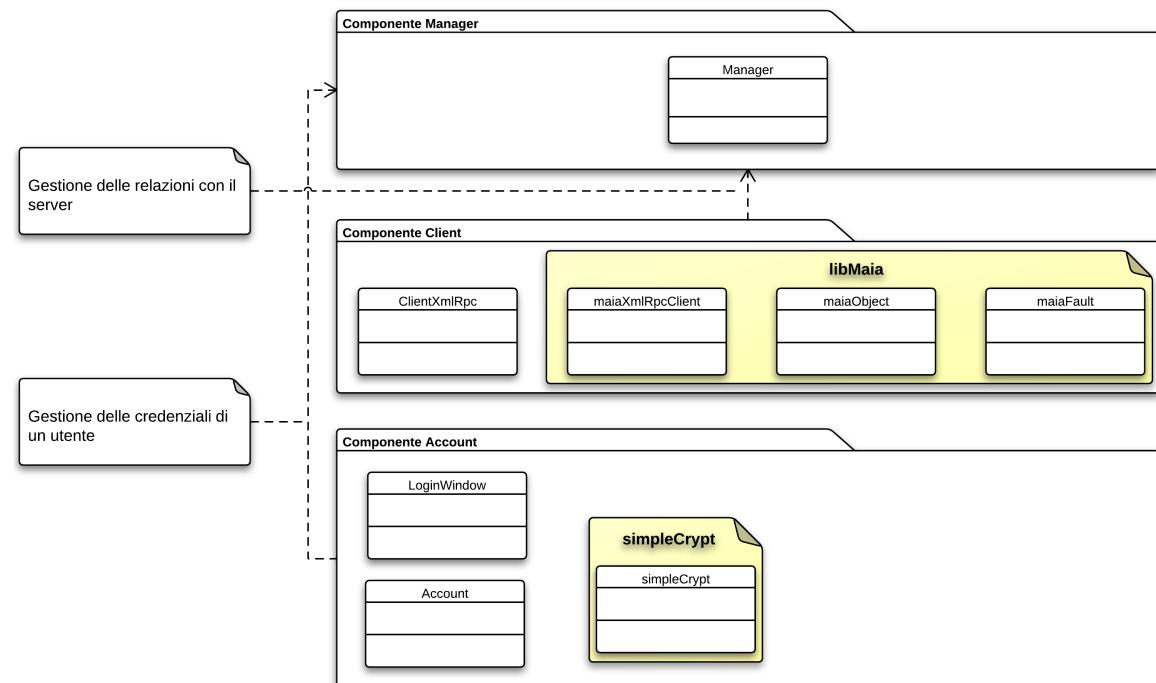


Figura 4.1: Desktop, diagramma dei componenti

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.

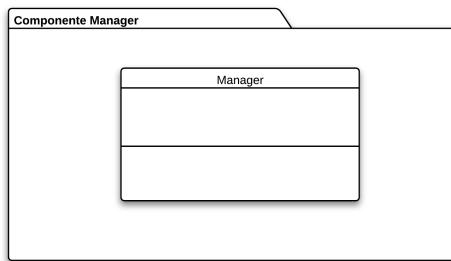


Figura 4.2: Desktop, componente Manager

- **Componente Account:**

ha il compito di gestire 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. Inoltre si occupa del salvataggio di tali credenziali e del criptazione della password, qualora l'utente decida di rimanere loggato al sistema.

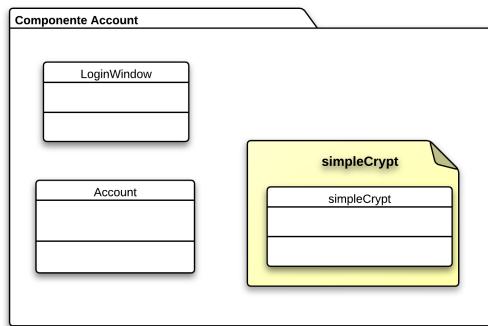


Figura 4.3: Desktop, componente Account

- **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.

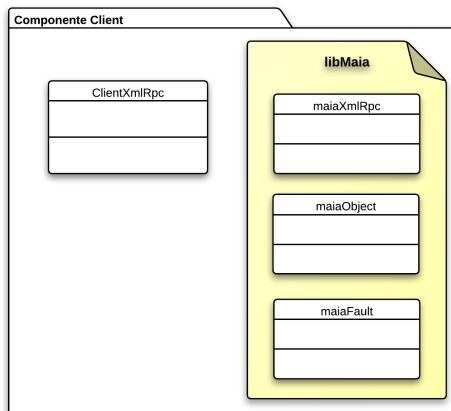


Figura 4.4: Desktop, componente Client

## 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"*.

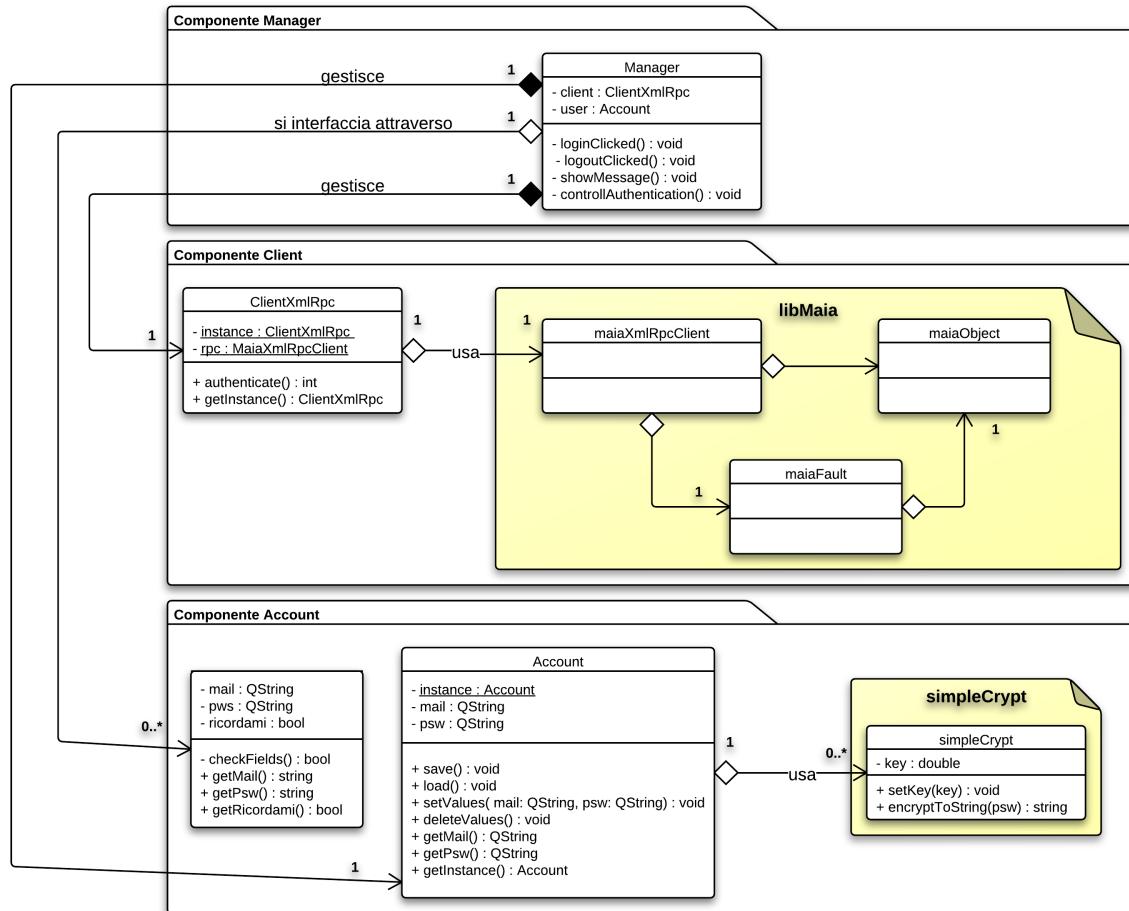


Figura 4.5: Desktop, diagramma delle classi

### 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 login, il logout con la cancellazione del file contenente le credenziali di accesso se queste erano state precedentemente salvate, e la chiusura 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'accesso in futuro. Tale classe farà inoltre un primo controllo sulla correttezza dei valori inseriti.

- **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 ad un precedente accesso su scelta dell'utente, 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 decrittaggio riceve la stringa criptata e ne restituisce la rispettiva stringa decriptata.

## Componente Client

- **Classe ClientXmlRpc:**

Classe singleton 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 ClientXmlRpc 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 4.6 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 possibili corrispondono al login e il logout al/dal sistema Server.

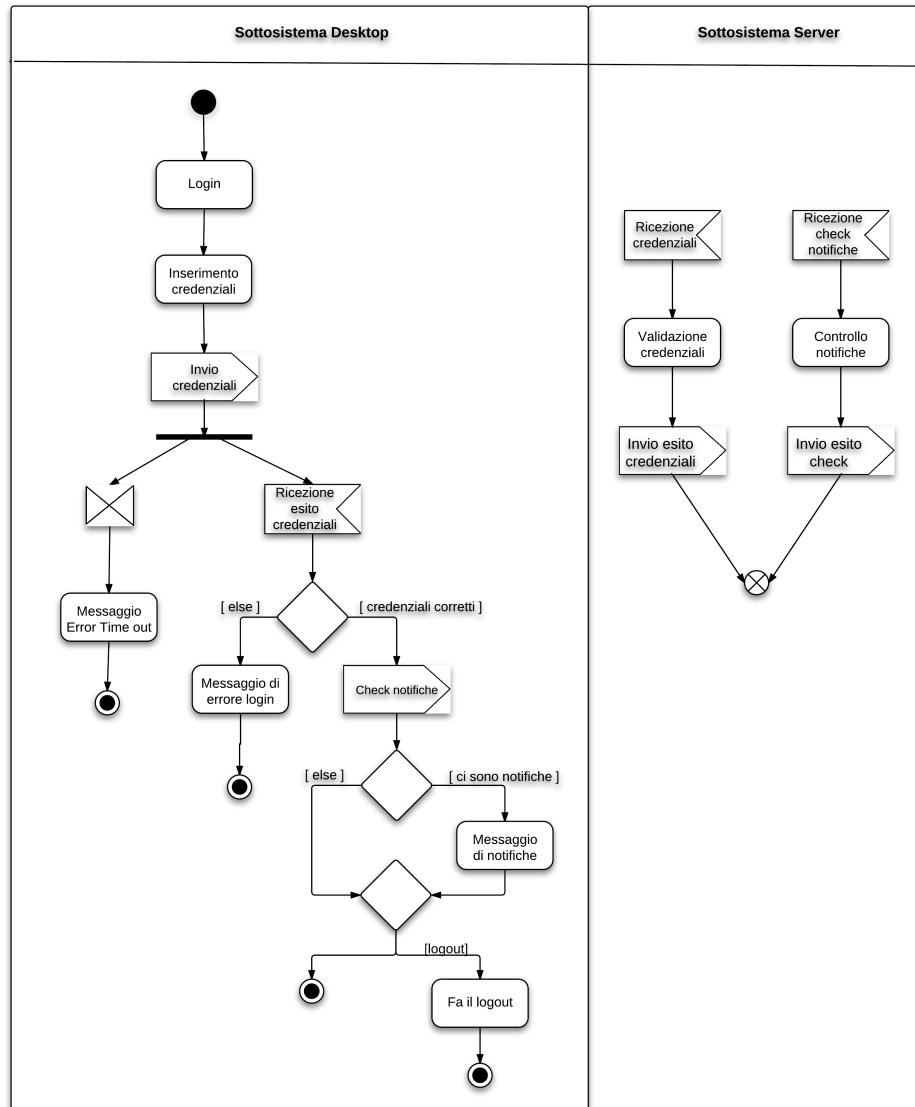


Figura 4.6: Desktop, diagramma delle attività

### 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 4.7 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 per l'inserimento delle credenziali, e successivamente attraverso il ClientXmlRpc viene richiesto il login al server che invierà una risposta indicando l'avvenuta o meno del login e quindi la correttezza delle credenziali.

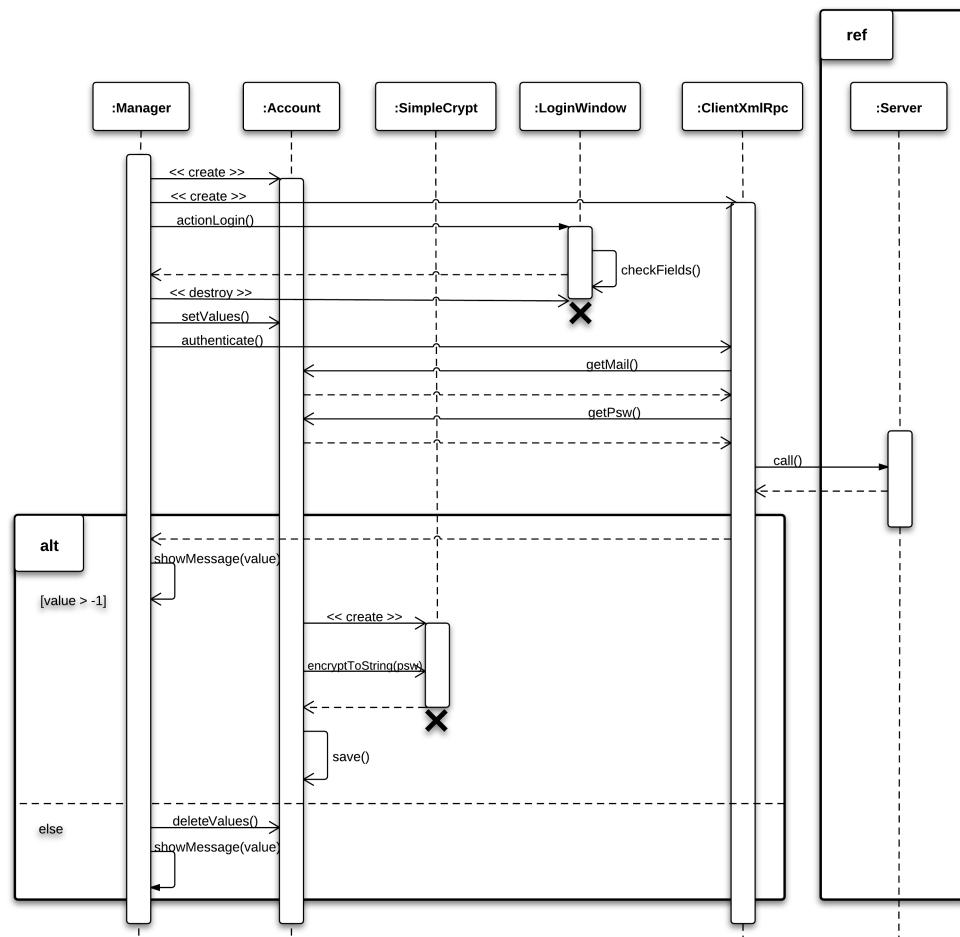


Figura 4.7: Desktop, diagramma di sequenza: login

**Logout:**

In figura 4.8 sono rappresentate le operazioni per il logout di un Desktop-user al sistema PMAC. Vengono cancellati le credenziali dall'Account e viene visualizzato un messaggio di avvenuto logout.

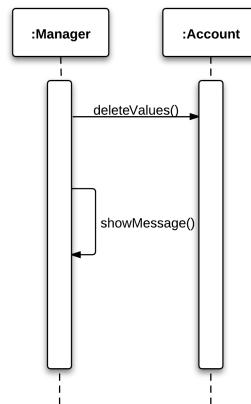


Figura 4.8: Desktop, diagramma di sequenza: logout

## 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

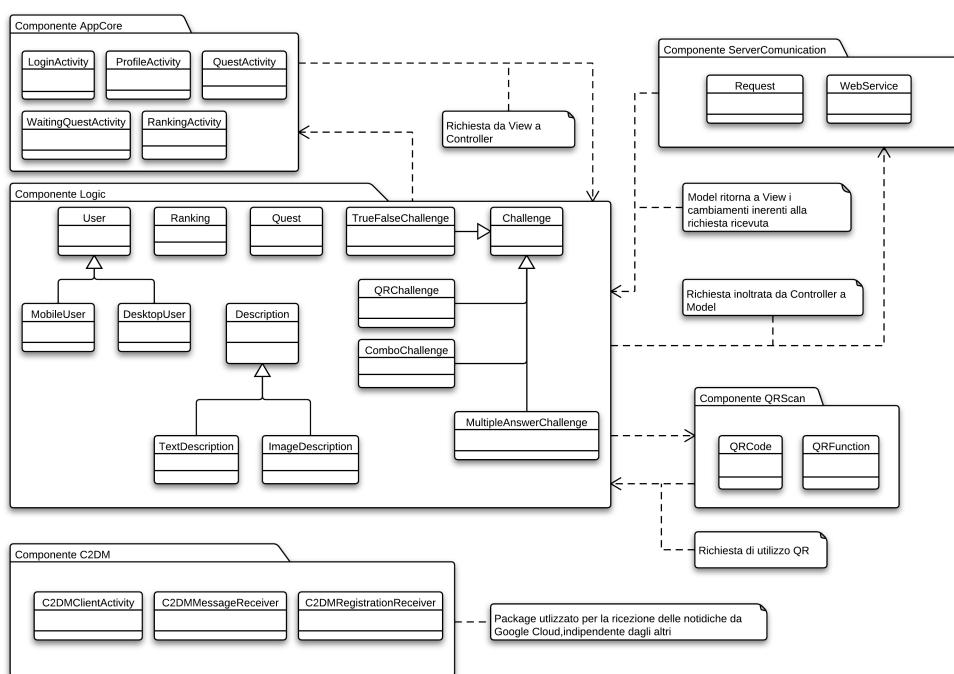


Figura 5.1: Mobile, diagramma dei componenti

### 5.1.2 Descrizioni componenti

- **Componente AppCore:**

Corrisponde al componente View del pattern MVC, riceve le richieste dall'utente e notifica le operazioni da eseguire al Controller.

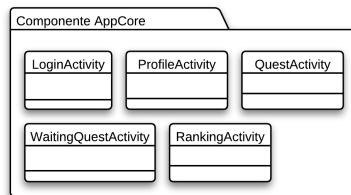


Figura 5.2: Mobile, componente Appcore

- **Componente Logic:**

Corrisponde al componente Controller del pattern MVC, inoltra le richieste giunte dalla View (AppCore) al Model (ServerCommunication), restituendo i risultati della chiamata alla View.

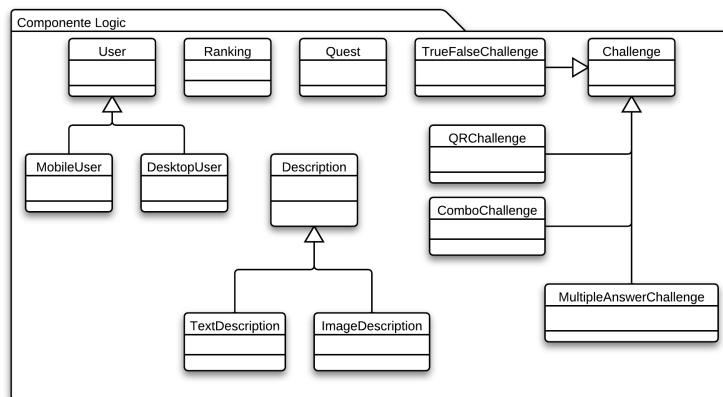


Figura 5.3: Mobile, componente Logic

- **Componente ServerCommunication:**

Corrisponde al componente Model del pattern MVC, riceve la richiesta dal Controller (Logic) e restituisce il risultato alla View (AppCore).

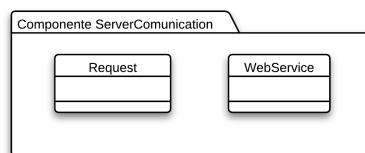


Figura 5.4: Mobile, componente ServerCommunication

- **Componente C2DM:**

Indipendente dagli altri componenti, ha il solo scopo di ricevere le notifiche da Google Cloud.

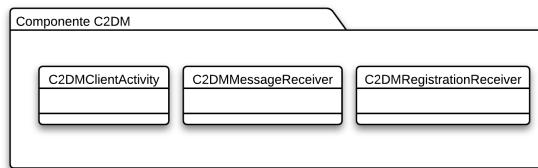


Figura 5.5: Mobile, componente C2DM

- **Componente QRScan:**

Permette la scansione di QR code, verrà utilizzato dalla sottoclass QRChallenge

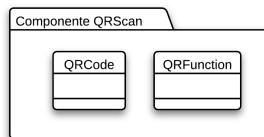


Figura 5.6: Mobile, componente QRScan

## 5.2 Diagramma delle classi

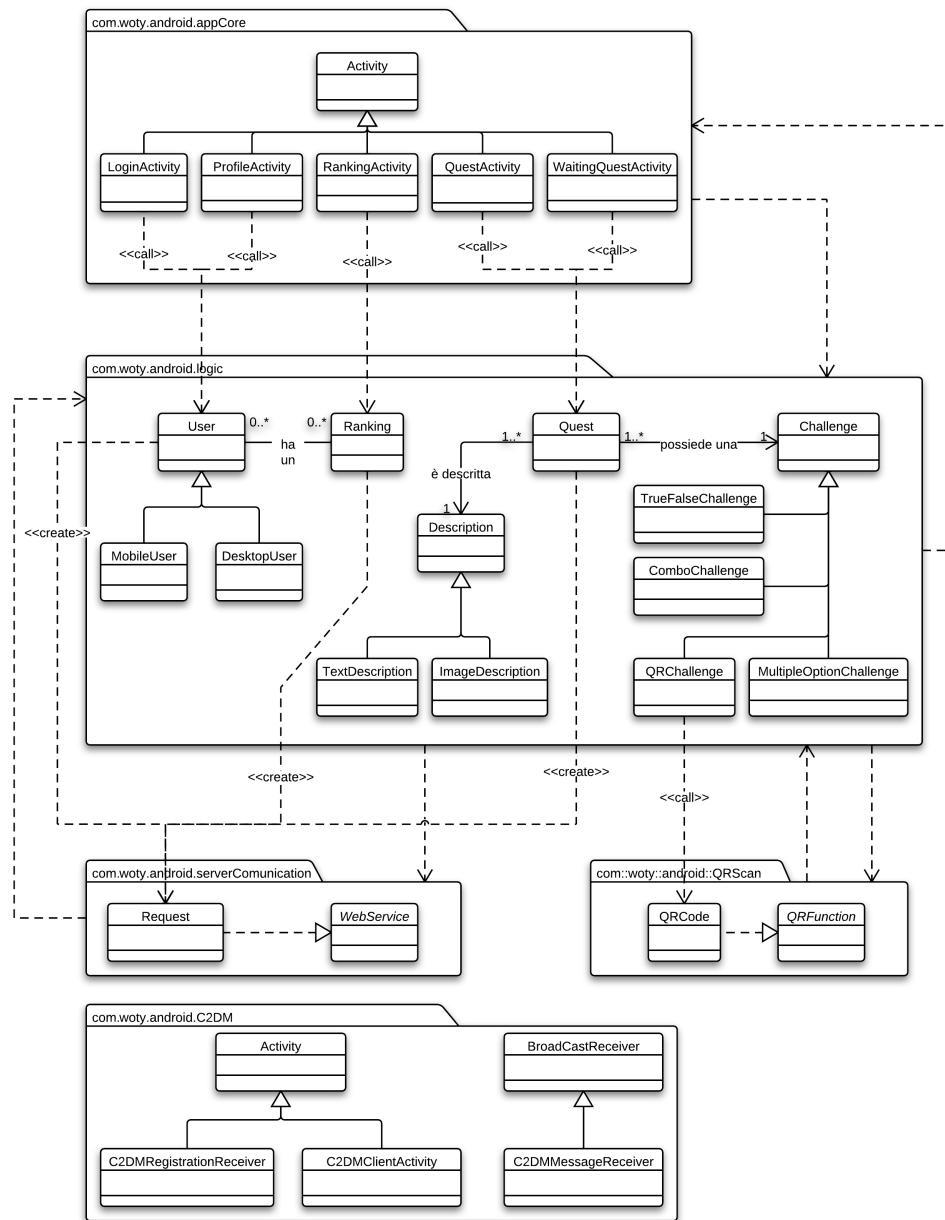


Figura 5.7: Mobile, diagramma delle classi

### Componente AppCore

- **Classe LoginActivity:**

Identifica il componente responsabile delle gestione del login da parte di un User al sistema. Dopo aver accettato la e-mail e la password dello User tramite interfaccia web è incaricato di verificare l'esistenza dello User e la validità delle credenziali tramite chiamata al controller.

- **Classe ProfileActivity:**

Identifica la classe responsabile della gestione e della visualizzazione del profilo di un User. Quando viene richiesta la visualizzazione o la modifica di uno profilo utente questo

componente è incaricato di passare i nuovi dati modificati dalla vista al controller o di richiedere al controller i dati del profilo da visualizzare.

- **Classe QuestActivity:**

Identifica la classe responsabile della gestione delle quest, dovrà quindi interagire con il controller per ottenere le quest da visualizzare e per comunicare le risposte date dal User.

- **Classe WaitingQuestActivity:**

Identifica la classe responsabile della gestione della coda delle quest che attendono di essere svolte.

- **Classe RankingActivity:**

Identifica la classe responsabile della gestione delle classifiche e deve interagire con il controller per ottenere le informazioni per creare le classifiche da mostrare all'utente.

## Componente Logic

- **Classe User:**

Classe che identifica uno User del nostro sistema. Fornisce i metodi per la creazione delle richieste per l'autenticazione e per la gestione del profilo verso il server ed ha una relazione molti a molti con la classe responsabile delle classifiche (Ranking).

- **Classe MobileUser:**

Classe che rappresenta un Mobile-user ed eredita dalla classe User. Offre le funzionalità di User e quelle specifiche per gli User in mobilità.

- **Classe DesktopUser:**

Classe che rappresenta un Desktop-user ed eredita dalla classe User. Offre le funzionalità di User e quelle specifiche per gli User che hanno postazione fissa.

- **Classe Ranking:**

Classe che rappresenta le classifiche della Gamification. Fornisce i metodi per effettuare le richieste al server dei dati da restituire alla View per la creazione delle classifiche. Ha una relazione molti a molti con la classe User.

- **Classe Quest:**

Classe che nel nostro sistema rappresenta le quest. Ha una relazione uno a uno con le classi Description e Challenge. Fornisce i metodi per effettuare le richieste al server per ottenere le quest elaborate dal PMAC Administrator.

- **Classe Challenge:**

Classe che identifica la "sfida" che una quest offre e ha una relazione uno a molti con la classe Quest. È ereditata dalle classi TrueFalseChallenge, QRChallenge, ComboChallenge e MultipleOptionChallenge.

- **Classe TrueFalseChallenge:**

Classe che eredita da Challenge ed identifica la sfida "vero o falso".

- **Classe QRChallenge:**

Classe che eredita da Challenge ed identifica la sfida da eseguire tramite scansione del QRCode.

- **Classe MultipleOptionChallenge:**

Classe che eredita da Challenge ed identifica la sfida a risposta multipla.

- **Classe ComboChallenge:**

Classe che eredita da Challenge ed identifica la sfida a risposta chiusa, quindi dove una sola tra le proposte proposte è corretta.

- **Classe Description:**

Classe che identifica la descrizione di una quest e ha una relazione uno a molti con la classe Quest. È ereditata dalle classi TextDescription e ImageDesctiption.

- **Classe TextDescription:**

Classe che identifica una descrizione testuale di una quest e appunto eredita dalla classe Description.

- **Classe ImageDescription:**

Classe che identifica una descrizione con immagine di una quest e appunto eredita dalla classe Description.

## Componente ServerComunication

- **Classe Request:**

Classe che rappresenta una richiesta da inviare al server ed eredita da WebService. È utilizzata dalle classi Quest, Ranking e User appunto per fare richieste al server web.

- **Interfaccia WebService:**

Classe che rappresenta il servizio web per le comunicazioni con il server.

## Componente C2DM

- **Classe C2DMClientActivity:**

Classe che gestisce, nel client mobile, il sistema C2DM per la ricezione delle notifiche.

- **Classe MessageReceiver:**

Classe che gestisce la ricezione dei messaggi push.

- **Classe RegistrationReceiver:**

Classe che gestisce la ricezione dei messaggi di avvenuta registrazione ai server Google per la ricezione delle notifiche.

## Componente QRScan

- **Classe QRCode:**

Classe che gestisce la scansione dei QRCode, rendendo disponibili al componente Logic le informazioni presenti. Per il suo corretto funzionamento estenderà l'interfaccia QRFunction,

che fornirà i metodi specializzati per svolgere tale compito.

- **Interfaccia QRFunction:**

Interfaccia contenente ulteriori funzionalità per l'implementazione della lettura di un QR Code. Comprenderà librerie aggiuntive e sarà implementata dalla classe QRCode.

### 5.3 Diagramma delle attività

In figura 5.8 viene fornito il diagramma delle attività per l'applicativo mobile, dove l'utente ha a disposizione una navigazione quasi completa da qualsiasi sezione dell'applicazione si trovi grazie ad un menù di navigazione.

Le azioni principali che l'utente può compiere con l'applicativo mobile sono la visualizzazione e risoluzione di quest, la richiesta di nuove quest, la visualizzazione delle classifiche e dei profili degli utenti e la visualizzazione e modifica del profilo personale.

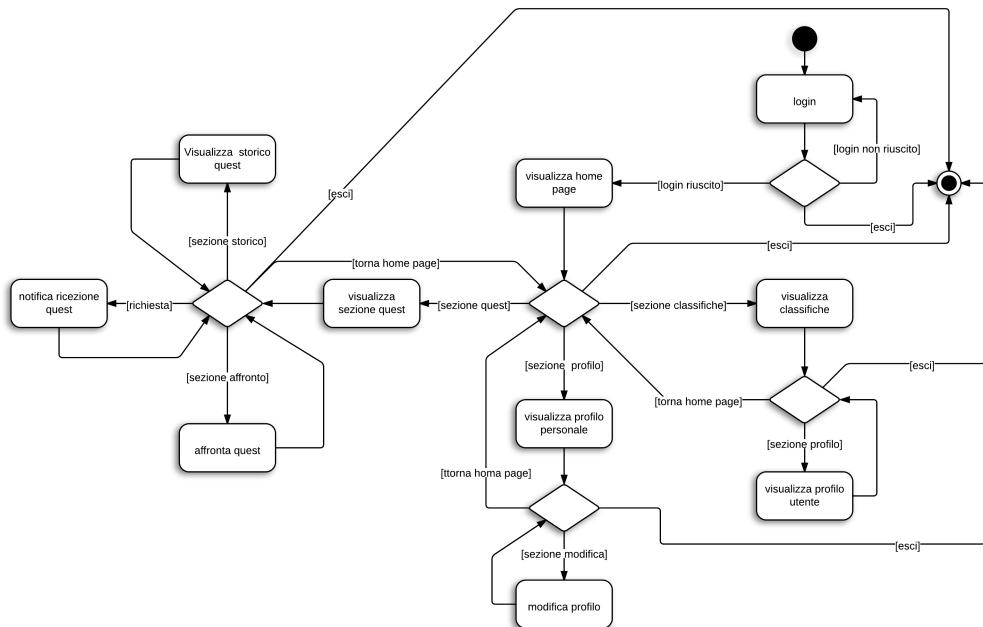


Figura 5.8: Mobile, diagramma delle attività

## 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. LoginActivity tenta di recuperare un oggetto di tipo User con i dati inseriti dall'utente, la stessa classe User si occuperà di verificarne la correttezza dialogando con la classe Request, e in caso di riscontro positivo restituirà un oggetto User completo.

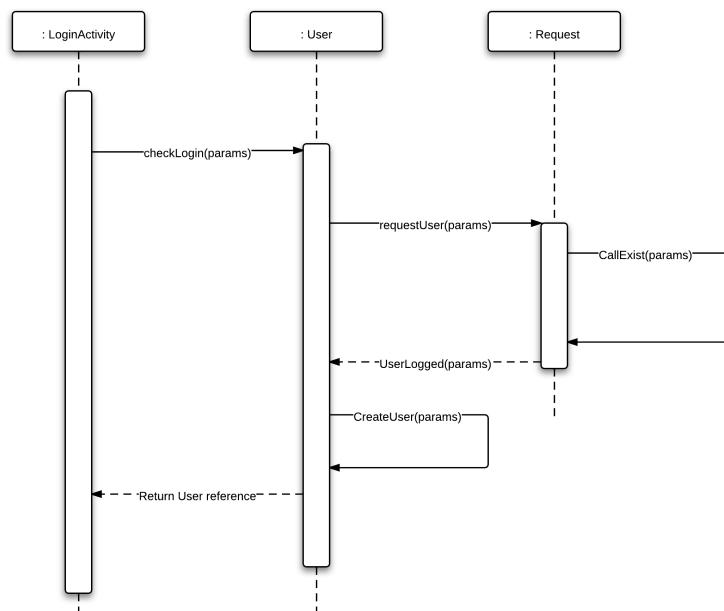


Figura 5.9: Login effettuato con successo

**Visualizza nuova 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 controllerà se per l'utente corrente esistono quest non salvate nel dispositivo; in caso affermativo, ne farà richiesta al server. WaitingQuestActivity richiede alla classe Quest una particolare quest non ancora salvata in locale; la classe Quest interrogherà la classe Request che ritornerà l'oggetto richiesto.

La richiesta da Quest a Request e la successiva risposta, in realtà, nascondono un passaggio più complesso, che verrà comunque affrontato nel successivo diagramma di sequenza (Figura 5.11).

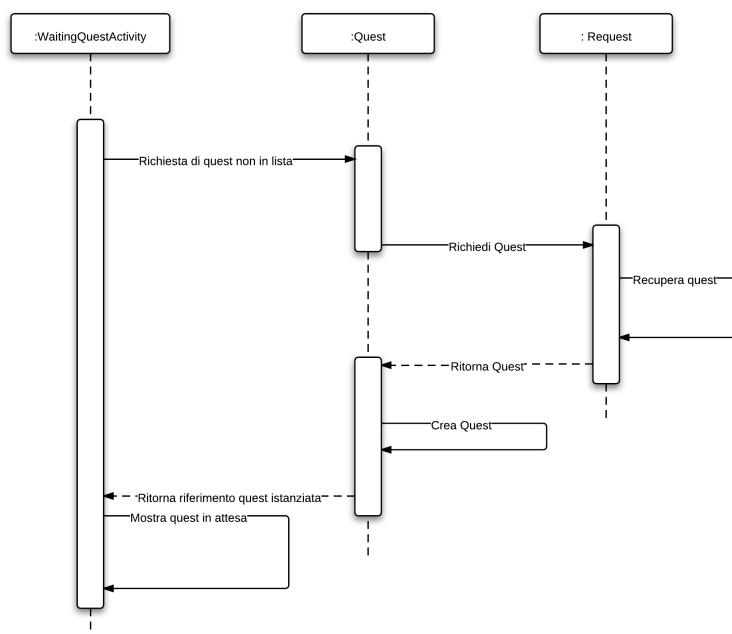


Figura 5.10: Recupera nuova quest segnalata ma non presente nel dispositivo

**Salva una quest nel dispositivo:**

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 alla classe Request, 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.

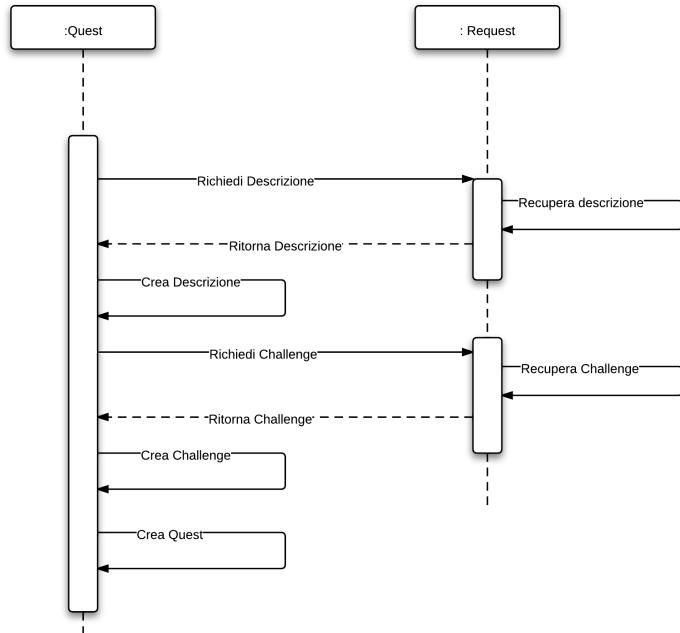


Figura 5.11: Recupera e salva quest nel dispositivo

**Diagramma di sequenza Richiedi una classifica:**

Nel momento in cui un utente volesse visualizzare una particolare classifica dal pannello RankingActivity, lo stesso tenterà di istanziare un nuovo oggetto di tipo Ranking con i parametri richiesti. La classe Ranking si occuperà di inoltrare la precedente richiesta alla classe Request che ritornerà l'oggetto completo, rendendolo così disponibile a RankingActivity.

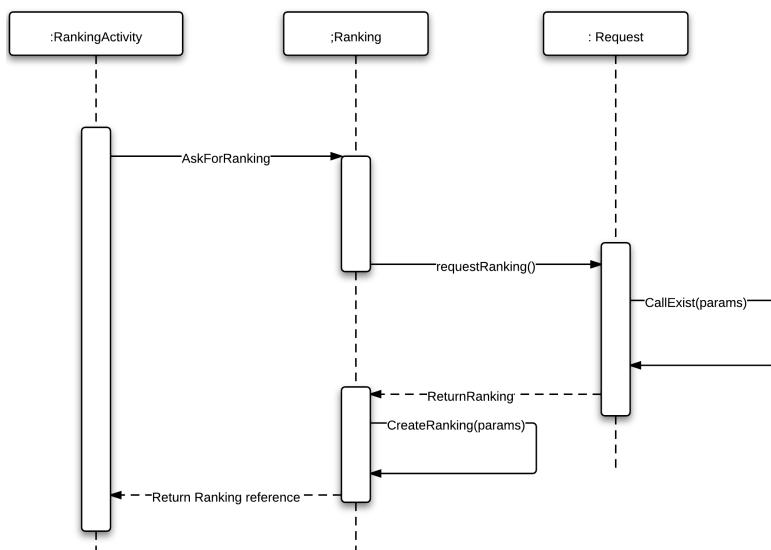


Figura 5.12: Richiesta di una classifica

## 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.pdf*, mentre per le specifiche indicazioni di ogni requisito si veda il documento *AnalisiDeiRequisiti.pdf*.

### 6.1 Sottosistema Server

Tabella 6.1: Server, tracciamento requisiti-classi

REQUISITO	CLASSE
Fo-1	Admin, Workgroup
Fo-2	Admin, Customer
Fo-2	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
Fo-3	Admin, Quest, QuestCategory, QuestChallenge, QuestDescription
Fo-3.1	Admin, Quest

(Continua alla pagina successiva)

*(Continua dalla pagina precedente)*

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
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

*(Continua alla pagina successiva)*

(Continua dalla pagina precedente)

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 6.2: Desktop, tracciamento componenti-requisiti

COMPONENTE	REQUISITO
Manager	Fo-7
Account	Fo-8
Client	Vo-2 Io-3 Io-5

Tabella 6.3: Desktop, tracciamento requisiti-componenti

REQUISITO	COMPONENTE
Fo-7	Manager
Fo-8	Manager
Vo-2	Client
Io-3	
Io-5	

## 6.3 Sottosistema Mobile

Tabella 6.4: Mobile, tracciamento componenti-classi-requisiti

Componente	Classe	Requisito
AppCore	LoginActivity, ProfileActivity, RankingActivity, QuestActivity, WainingQuestActivity	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
Logic	User, Ranking, Quest, Description, Challenge	Fo-6, Fo-10, Fo-10.1, Fo-10.1.1, Fo-10.1.2, Fo-10.1.3
ServerComunication	WebService	Fo-8.4
QRScan	QRFunction	Fo-5
C2DM	C2DMMassageReceiver, C2DMRegistrationReceiver, C2DMClientActivity	Fo-8, Io-5

Tabella 6.5: Mobile, tracciamento requisiti-componenti-classi

Requisito	Componente	Classe
Fo-5	QRScan	QRFunction
Fo-6	Logic	Quest
Fo-7	AppCore	LoginActivity
Fo-8	C2DM	C2DMMMessageReceiver
Fo-8.1	AppCore	QuestActivity
Fo-8.2	AppCore	WainingQuestActivity
Fo-8.3	AppCore	WainingQuestActivity
Fo-8.4	ServerCommunication	WebService
Fo-9	AppCore	RankingActivity
Fo-10	Logic	User
Fo-10.1	Logic	User
Fo-10.1.1	Logic	User
Fo-10.1.2	Logic	User
Fo-10.1.3	Logic	User
Fo-10.2	AppCore	ProfileActivity
Fo-10.3	AppCore	ProfileActivity
Fo-10.3.1	AppCore	ProfileActivity
Fo-10.3.2	AppCore	ProfileActivity
Fo-10.3.3	AppCore	ProfileActivity
Fo-10.3.4	AppCore	ProfileActivity
Fo-10.3.5	AppCore	ProfileActivity
Fo-10.4	AppCore	ProfileActivity
Fd-1	AppCore	ProfileActivity
Fd-2	AppCore	QuestActivity
Io-4	AppCore AppCore AppCore AppCore AppCore	LoginActivity ProfileActivity RankingActivity QuestActivity WainingQuestActivity
Io-5	C2DM	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.