

Università degli Studi di Torino

Dipartimento di informatica



Tesi di Laurea Triennale in Informatica

**Progettazione e realizzazione di un
modulo di gamification per la
piattaforma FirstLife**

Relatore:

Prof: Claudio Schifanella

Candidato:

Luca Modica

*Let's make everything better with
the lessons we've learned through
many hours of game-play in
our lives!*

- Chou Yu-Kai, Actionable Gamification:
Beyond Points, Badges, and Leaderboard

Abstract

FirstLife è una piattaforma che ha l’obiettivo di creare un ambiente collaborativo fra cittadini, così come coordinare iniziative e progetti fra stakeholder e supportare processi di gestione di beni comuni urbani. Unisce funzioni di un social network e la possibilità per gli utenti di aggiungere contenuti georiferiti per valorizzare il proprio territorio locale, le proprie idee a riguardo, le azioni svolte da una comunità e iniziative in ambienti reali. *FirstLife* è disponibile sia come piattaforma generalizzata (il social network civico), sia as-a-service per progetti esterni. Nel secondo caso potrà essere personalizzata in base alle specifiche esigenze, variabili al tipo di utente o organizzazione che ne richiede ed il loro obiettivo.

Soprattutto nell’ultimo decennio si è consolidato in molti ambiti l’utilizzo di una tecnica detta *gamification*: si tratta dell’applicazione di elementi ispirati da videogiochi (punti, livelli, badge o ancora classifiche) in contesti non ludici, arricchendo l’esperienza in servizi ed organizzazioni. In questo modo è possibile influenzare e direzionare il comportamento delle persone per un loro coinvolgimento attivo in attività, al fine di raggiungere determinati obiettivi come l’incremento delle vendite di un’impresa. I contesti di applicazione sono davvero vasti: miglioramento della produttività di un’impiegato in azienda, dell’apprendimento e della motivazione in campagne di crowdsourcing sono solamente alcuni.

Lo scopo principale della tesi è quello di integrare in *FirstLife* un nuovo componente, che permetta di introdurre ed implementare dinamiche e meccanismi di gamification, in maniera altamente personalizzabile. Arricchendo l’esperienza e i servizi della piattaforma, si potranno incentivare determinate azioni desiderate e formare così utenti attivi anche in processi complessi, come la collaborazione con altri cittadini o la creazione di nuovi contenuti od iniziative.

Il primo passo è stato studiare gli elementi di Gamification da rappresentare ed utilizzare nel modulo, il loro funzionamento e i meccanismi che regolano l’ambiente: dalle regole per cui assegnare reward agli utenti che svolgono una certa attività, all’organizzazione di ricompense simboliche come i badge. L’approccio nello studio di tali aspetti deriva dal componente di gamification del progetto CO3. Esso ha lo scopo di valutare gli impatti di tecnologie innovative in ambiente cittadino: blockchain, AR o ancora strumenti di democrazia interattiva.

Nel passo successivo questi elementi sono stati effettivamente implementati: è stato utilizzato il database NoSQL MongoDB per la memorizzazione di dati e scritta la logica di backend con NestJs, framework basato sull’ambiente server-side Node.js. Il risultato è stato l’esposizione delle RESTful API per gestire gli elementi di Gamification ed attivare del meccanismo per assegnare le ricompense, unite allo scheduling di cron-jobs per gestire eventi ricorrenti (come le competizioni).

Infine, sono state sviluppate 2 web-app grazie al popolare framework frontend Angular. Il primo applicativo è per utenti amministratori in *FirstLife*, dove potranno creare gli elementi per la propria strategia di Gamification e monitorarne l’andamento. In una seconda app invece ciascun utente potrà visualizzare il proprio profilo con livello raggiunto, ricompense guadagnate e posizione in competizioni attive. L’interfaccia delle web-app sono basate sulla libreria di componenti grafici Angular Material.

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Indice

1 FirstLife	5
1.1 Il modello concettuale	5
1.1.1 Modello Utente	6
1.1.2 Modello di Entità	6
1.2 La piattaforma	7
2 Gamification	11
2.1 Che cos'è la Gamification?	11
2.2 Framework di progettazione della Gamification	15
2.2.1 MDA framework	16
2.2.2 Octalysis framework	17
2.3 Esempi di Gamification	19
2.3.1 Duolingo 	20
2.3.2 Waze 	22
2.4 Gamification in FirstLife	23
3 Tecnologie utilizzate	27
3.1 MongoDB	27
3.2 NestJS	29
3.2.1 Elementi principali	29
3.2.2 Mongoose	34
3.2.3 Repository Pattern	35
3.2.4 Class validator/transformer	36
3.2.5 Cron jobs	37
3.3 Postman e Swagger	38
3.4 Angular	39
3.4.1 Componenti, direttive e data binding	40
3.4.2 Angular material	43
3.4.3 RxJS	44
4 Progettazione	47
4.1 Approccio iniziale	49
4.2 Elementi del modulo	49
4.2.1 Elementi introduttivi	49
4.2.2 Regole	52
4.2.3 Badge	56
4.2.4 Profili ACA	62
4.2.5 Profili utente	63
4.2.6 Log di assegnamento	64

4.2.7	Classifiche	65
4.3	Meccanismi del modulo	66
4.3.1	Meccanismo Principale	67
4.3.2	Task settimanali e mensili	71
4.3.3	API degli elementi	72
5	Gamification Engine	75
5.1	Struttura ed elementi generali del backend	76
5.1.1	Cartella common	77
5.1.2	Cartella config	81
5.1.3	Cartella database	82
5.2	Moduli degli elementi	83
5.2.1	Rules Module	90
5.2.2	Badges Module	92
5.2.3	Aca Profiles Module	95
5.2.4	User Profiles Module	96
5.2.5	Logs Assignment Module	99
5.2.6	Rankings Module	102
5.3	Moduli dei meccanismi	104
5.3.1	App Module	104
5.3.2	Tasks Module	107
5.4	Risultati finali	110
6	Interfacce grafiche	113
6.1	Struttura ed elementi delle applicazioni web	113
6.1.1	Modelli	114
6.1.2	Service	115
6.1.3	Componenti	116
6.2	Dashboard amministratore	117
6.2.1	Schermata delle regole	117
6.2.2	Schermata dei badge	120
6.2.3	Schermata dei log di assegnamento	125
6.2.4	Generatore di classifiche	127
6.2.5	Monitoraggio statistiche in aree	128
6.3	Dashboard utente	128
6.3.1	Schermata principale	129
6.3.2	Schemata badge e obiettivi	131
6.3.3	View delle competizioni attive	134
7	Conclusioni	137

Capitolo 1

FirstLife

FirstLife è un social network civico che ha lo scopo di creare un ambiente per la collaborazione fra cittadini, coordinare iniziative di stakeholder a livello locale e supportare i processi di gestione di beni comuni urbani. La piattaforma consente di fare community mapping digitale grazie ai contenuti georiferiti aggiunti dagli utenti (anche detti Volunteered Geographic Information, VGI); unite alle funzionalità social degli stessi contenuti, permettono di valorizzare, coordinare e monitorare le azioni svolte dalla comunità, fornendo supporto ad ambienti reali come iniziative, progetti e molto altro ancora [3, 16].

Si hanno pertanto molteplici utilizzi, in base al tipo di utente o organizzazione ed il proprio obiettivo:

- *i cittadini e le cittadine* possono condividere storie, idee e progetti per la comunità;
- *le scuole* hanno la possibilità di promuovere con maggior efficacia l'educazione civica e digitale;
- *gli enti di ricerca, del terzo settore ed istituzioni* hanno a disposizione uno strumento per meglio indagare sul territorio locale e questioni di interesse o attività ad alta partecipazione.

La piattaforma inoltre non è solo rappresentata dal singolo social network, ma anche come servizio per progetti esterni (detti anche *progetti branch* o *verticalizzazioni*). FirstLife pertanto è sia disponibile come servizio generale con tutte le sue funzionalità, sia è possibile personalizzarla per gli scopi specifici di iniziative esterne.

1.1 Il modello concettuale

Sin dall'inizio la progettazione e sviluppo di FirstLife ha avuto un *approccio user-centered*, mediante il coinvolgimento di gruppi eterogenei di stakeholder che hanno portato all'emersione e successiva analisi dei requisiti utente. Il risultato di questo processo partecipativo ha portato alla definizione dei 2 modelli sul quale si basa quello concettuale della piattaforma: il *modello utente e di entità*.

1.1.1 Modello Utente

Il modello utente è basato sul seguente concetto: un utente, all'interno di un contesto urbano, può assumere differenti ruoli a far parte di più reti nello stesso territorio. Sono state individuate principalmente 3 reti al quale un cittadino può far parte:

- *professional network*: include le relazioni lavorative;
- *territorial network*: formata dalle relazioni più personali o politiche, create ad esempio partecipando attivamente in processi decisionali o formazione di opinioni;
- *community network*: rappresentata da membri in gruppi locali od organizzazioni non-profit.

1.1.2 Modello di Entità

Ispirati anche dal contesto urbano/geografico che riguarda il progetto, il modello rappresenta gli elementi basati sulle azioni sociali effettuate dagli utenti (partecipazione in gruppi, condivisione di notizie, scrittura di storie, ...), raggruppate in base al tipo di concetto che vogliamo rappresentare. In FirstLife esistono principalmente 5 tipi di entità:

- *Places*: edifici o spazi dedicati perlopiù ad eventi pubblici o attività di comunità, piuttosto che punti di interesse dal punto di vista socio-culturale (dei luoghi storici, ad esempio);
- *Events*: rappresentano un qualsiasi tipo di evento pubblico di ogni dimensione, da esibizioni a festival, ma anche iniziative a livello di singolo quartiere residenziale;
- *News*: azioni di condivisione di avvisi, annunci o anche notizie a livello locale;
- *Extra*: legati soprattutto a dove vive un utente, possono essere storie, testimonial o progetti;
- *Groups*: insiemi di utenti che effettuano azioni di coordinamento e organizzazione, oltre a poter scambiare informazioni tra loro in tempo reale.

Le entità sopra descritte sono dette *di prim'ordine*: ognuna possiede un insieme di proprietà essenziali (un nome, una descrizione, l'utente che l'ha creata e l'ultimo che l'ha aggiornata) unita a caratteristiche peculiari per ogni tipo, come l'organizzatore di un evento o la data di pubblicazione di una notizia. Le entità di prim'ordine possono essere arricchite da altre chiamate *di secondo ordine*, di scala minore e più legate al singolo utente e alla sua personale prospettiva (come un commento, una foto o anche un sondaggio).

Le entità di prim'ordine possono anche essere connesse e creare relazioni fra di loro, potendo così rappresentare concetti più complessi come storie appartenenti a gruppi o eventi riguardanti determinati posti (Figura 1.1).

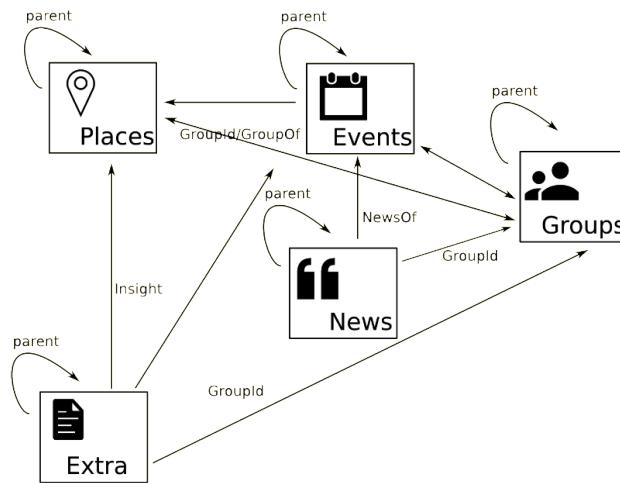


Figura 1.1: entità di prim'ordine e le loro possibili relazioni.

Infine, le entità descritte sopra possono essere organizzate in una gerarchia o ancora raggruppate: insieme possono rappresentare un' iniziativa o progetto di un singolo o più stakeholder, piuttosto che un criterio utente (come i luoghi della sua infanzia). Un aggregatore di entità viene anche detto *Tema* o *PlacesList*.

1.2 La piattaforma

FirstLife è stata sviluppata come un' applicazione web, disponibile per PC e smartphone¹. La sua interfaccia grafica è suddivisa in 2 parti: *una mappa interattiva ed un "wall" di contenuti*, ovvero un elenco di entità di prim'ordine visualizzate in riquadri con layout a griglia (Figura 1.2).

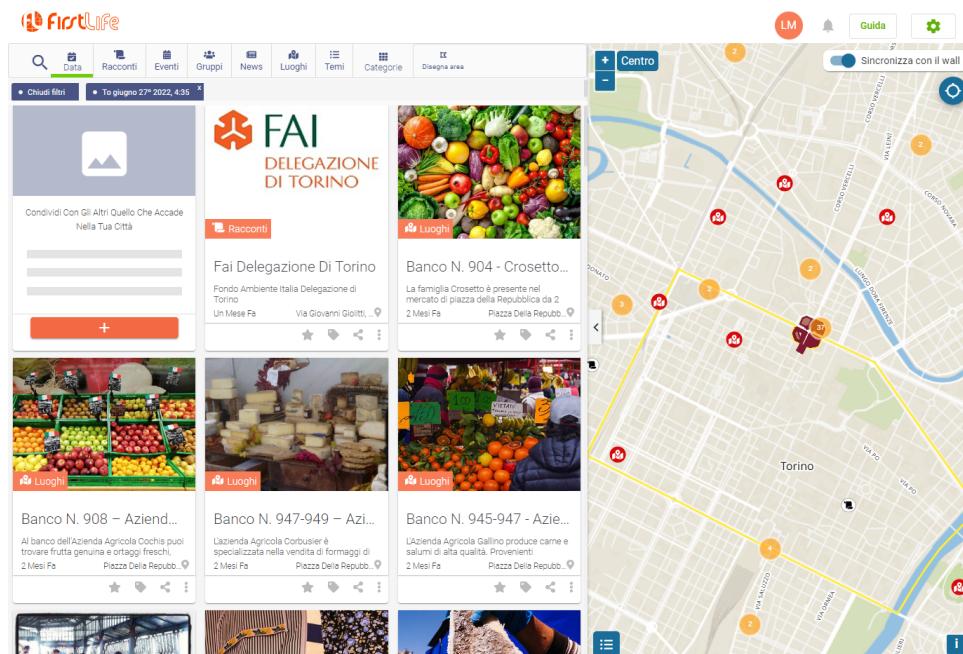


Figura 1.2: Interfaccia grafica di FirstLife.

¹Qui è possibile provare la demo della piattaforma: <https://cittat15minuti.firstlife.org/>.

Cliccando su uno dei riquadri si possono visualizzare i dettagli di tale entità (sue proprietà, sotto-entità o entità di secondo ordine, come commenti o foto allegate), mentre la mappa viene aggiornata in tempo reale con la sua posizione precisa. La mappa infatti rappresenta un filtro a livello geografico dei contenuti presenti nel "wall": in base all'area che si seleziona, dove viene fatto uno zoom o ancora che si delimita (è possibile disegnare un poligono per delimitare una certa zona), l'elenco di entità viene aggiornato di conseguenza. Come anche mostrato nella precedente figura, altri filtri possibili sono:

- *per intervallo temporale* ove le entità risultano valide (come la data di un evento);
- *in base al contenuto stesso*: uno dei tipi di entità di prim'ordine, categorie, sue proprietà specifiche o ancora un tag definito sul momento.

Cliccando sull'apposito pulsante, si possono aggiungere nuove entità sulla piattaforma. Viene aperta una schermata dove è possibile inserire tutti i suoi dati: tipo di entità di prim'ordine, nome, indirizzo e altre proprietà che dipendono da tipo scelto in precedenza. Inoltre, sia in fase di creazione che in seguito, gli utenti possono arricchire le entità appena create con altre correlate (come una notizia riguardo un evento specifico) ed entità di secondo ordine (Figura 1.3).

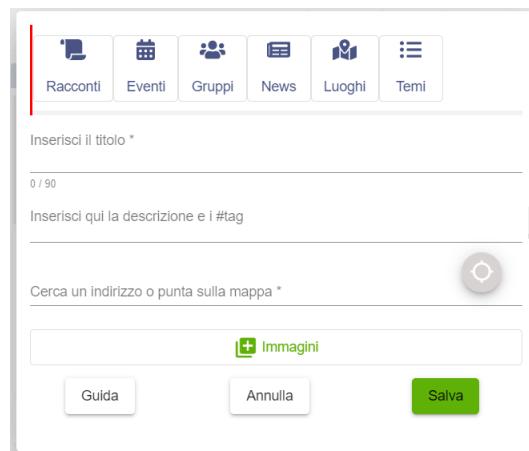


Figura 1.3: Schermata di creazione di un'entità.

Si può notare dalla schermata di creazione che è presente, oltre ai 5 tipi di entità di prim'ordine, l'aggregatore di entità "Temi". Un tema viene visualizzato come le altre entità sulla piattaforma. Visualizzando i dettagli di un tema si possono creare nuove entità al suo interno (Figura 1.4).

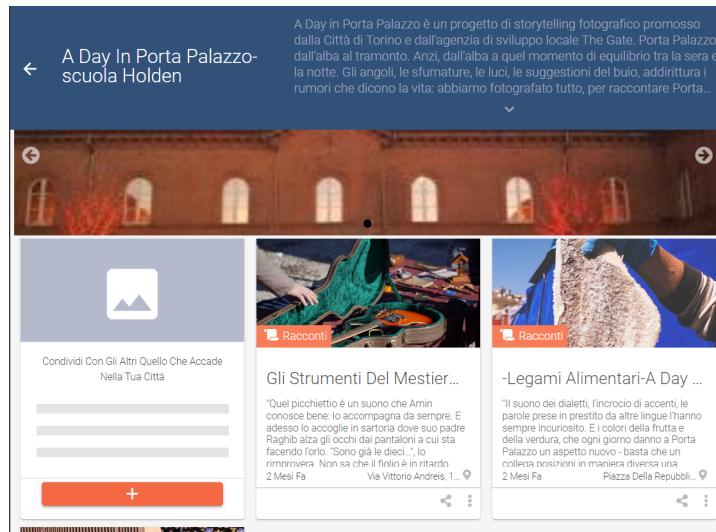


Figura 1.4: Esempio di tema.

Come detto in precedenza, FirstLife è utilizzato come servizio in progetti esterni. Qui di seguito sono riportati esempi di applicazioni in 3 differenti contesti.

1. *MiraMap*². Il progetto, localizzato nel quartiere di Mirafiori di Torino, ha l'obiettivo di facilitare ed innovare le comunicazioni tra cittadini ed enti pubblici della zona. FirstLife viene utilizzata in questo contesto per permettere ai cittadini di segnalare problemi a livello locale grazie ad un sistema a ticket (condivisi fra utenti ed autorità locali) o inviare proposte di miglioramento del territorio.
2. *ConsegnaTo*³. Iniziativa nata durante l'emergenza COVID, Consegnato ha fatto utilizzo di FirstLife per permettere a gestori di imprese e cittadini di condividere informazioni (come orari di apertura di negozi o listini di prodotti) e coordinarsi grazie alla mappa interattiva (ad esempio, per le consegne a domicilio).
3. *EduLife*⁴. In collaborazione con gruppi di ricerca e scuole superiori, il fine di EduLife è quello di utilizzare l'applicazione web per attività di educazione civica e digitale, dando in particolare voce ad iniziative e proposte degli adolescenti.

L'obiettivo della tesi è quello di implementare un nuovo modulo in FirstLife che permetta di introdurre e progettare elementi e strategie di Gamification. In questo modo si potrà agire sulla motivazione degli utenti che utilizzano la piattaforma, arricchendo l'esperienza con dinamiche e meccaniche di gioco per incentivarli ad effettuare determinate azioni desiderate. Questo anche interagendo e cooperando con altri cittadini e organizzazioni, o ancora in aree geografiche specifiche. Il sistema permetterà da una parte ad utenti amministratori di creare la propria strategia di Gamification, creando elementi ad-hoc in modo altamente personalizzabile; dall'altra, ciascun utente potrà visualizzare i propri progressi e ricompense in un'interfaccia apposita.

²Sito web del progetto: <https://www.miramap.it/>

³sito web della piattaforma: <https://torinocitylove.firstlife.org/>

⁴Sito web della piattaforma: <https://edu.beta.firstlife.org/>

Capitolo 2

Gamification

2.1 Che cos'è la Gamification?

La rapida crescita dell'industria videoludica e del consumo di questo tipo di media è ormai evidente da decenni. Il fenomeno, inoltre, viene anche accompagnato dalla proliferazione di software e piattaforme che attingono ispirazione dai videogiochi: a questo trend ci si riferisce con il termine "gamification". La gamification introduce l'idea di **applicare elementi e tecniche ispirati da videogiochi in contesti non ludici, con il fine di motivare ed incrementare l'attività utente**: si tratta di un concetto che ha suscitato molto interesse nel campo dell'interaction design e nel digital marketing, proprio per le sue potenzialità e risultati notevoli già raggiunti in termini di engagement [6].

Il termine "gamification" si è originato nel 2008 dall'industria dei digital media, e ampiamente riconosciuto intorno al 2010. Tuttavia la storia di questo settore, anche dal punto di vista accademico e di ricerca, è costellata da numerose definizioni e dibattiti principalmente per 2 motivi.

- I designer di esperienze videoludiche utilizzano spesso una vasta gamma di tecniche per realizzarle, portando quindi ad utilizzare termini molto differenti fra di loro.
- Quando si parla di un'esperienza "gamificata", spesso il mondo accademico presenta 2 principali idee. La prima si basa sul fatto che l'incremento dell'adozione dei videogiochi ed elementi di questi ultimi ha cambiato e formato la vita di tutti i giorni e le interazioni fra le persone. La seconda è maggiormente specifica, considerando i contenuti videoludici più come forma di intrattenimento che utilità: essi sono in grado di produrre determinati stati ed emozioni, al fine di motivare gli utenti e mantenerli attivi all'interno di una piattaforma con grande intensità e per molto tempo.

I primi veri tentativi di applicazione di questa idea risale addirittura negli anni Ottanta, con il design di un'avventura testuale chiamata *Adventures*: si tratta di una delle prime esperienze online dove le persone potevano interagire fra di loro, attraverso una UI che pur essendo sotto forma di solo testo risultava incredibilmente piacevole. Da allora, ricercatori nel campo della HCI (Human-Computer Interaction) hanno cominciato ad effettuare ricerche su come i giochi potessero avere un scopo, utilizzando le loro peculiarità per risolvere determinati problemi (classificare immagini rappresenta una delle tante applicazioni).

Il loro studio, dal punto di vista della progettazione e tipo di esperienza, hanno portato ad una serie di risultati di importante rilevanza. È cominciato ad emergere un primo concetto concreto di "gamification", ossia osservare un videogioco con uno scopo differente dall'intrattenimento casalingo. Questo è stato accompagnato dalla nascita di nuovi contenuti che si ispirano a quest'ultima definizione: *serious games*, videogiochi che hanno uno scopo che va oltre a quello di intrattenimento (piattaforme educative, simulazioni per addestramento militare o di interventi chirurgici sono solo alcuni casi); *pervasive games*, un genere che va ad espandere l'esperienza videoludica in nuovi contesti, situazioni o anche posti (un esempio eclatante è rappresentato dai videogiochi basati su geo-localizzazione).

Come è possibile immaginare e citato in precedenza, l'aumento del tipo di contenuti relativi al mondo dei videogiochi non ha fatto altro che generare ulteriore confusione nell'ambito della "gamification" stessa. Allo scopo di chiarire meglio il concetto, esso verrà analizzato in diversi punti, basandosi sulla definizione data inizialmente ("applicare elementi e tecniche ispirati da videogiochi in contesti non ludici").

- Innanzitutto, è importante precisare che la gamification non è inherente al concetto di *giocare*, bensì a quello di *gioco*. Mentre il giocare si riferisce ad un insieme di comportamenti (generalmente liberi), i giochi rappresentano un insieme più specifico di essi, comprendendo una forma del giocare sotto determinate regole che generano una competizione, al fine di raggiungere un obiettivo. È bene precisare che i giocattoli fanno parte del concetto di "giocare", poiché anch'essi non sono limitati da criteri da rispettare.

Alla luce di questo, la gamification è sinonimo del cosiddetto *gameful design*, ossia della creazione di un'esperienza con la nozione di "gioco". Al contempo differisce invece dal *playful design*, che invece riprende la nozione di "giocare".

- Si ricorda che il gameful design utilizza solo singoli **elementi** caratteristici da videogiochi, piuttosto che descrivere un'intera esperienza videoludica. Sulla base di questo, i *serious games* non rappresentano un caso di gamification: per quanto abbiano uno scopo che divaghi da uno di intrattenimento essi rappresentano un'intera esperienza videoludica, e non una che utilizza solo sue parti.

Vi è un vasto insieme di singoli elementi utilizzabili per "gamificare" una piattaforma: avatar, punti, badge, ambienti tridimensionali, competizioni e molto altro ancora, in crescita proporzionale a quello dell'industria videoludica. Il come sono utilizzati ed il loro significato verranno approfonditi nel corso del capitolo.

- Degli elementi presi in considerazione, è fondamentale tenere conto della loro **progettazione**. Seppur presi da contenuti videoludici, essi devono essere creati sulla base della piattaforma per cui vogliamo creare un certa esperienza e dei suoi scopi, e non considerando i soli videogiochi. Inoltre, tali elementi vengono suddivisi sotto molteplici livelli di astrazione, che vanno da quelli inerenti a pattern per il design di interfacce, fino a modelli concettuali per componenti di gioco e metodi di game design più concreti.
- Come per i *serious games*, l'esperienza "gamificata" è realizzata per scopi che vanno oltre il puro divertimento. In altre parole, come detto in precedenza

vengono impiegati in **contesti non ludici**, sfruttando la piacevole interfaccia di un videogioco.

Grazie anche alle sue enormi potenzialità in termini di engagement, i contesti di applicazione sono davvero numerosi: da esperienze per l'allenamento fisico, a quelle dedicate all'educazione. Gli ambiti di utilizzo possono inoltre essere suddivisi in 3 categorie.

- Si dicono *esterni* quei contesti che sono al di fuori di un'azienda, e non nella sua organizzazione interna. In altre parole, si parla della gamification applicata per un miglior engagement degli utenti, aumentare vendite o ancora potenziare il marketing.
- Seguono i contesti detti *interni*, ossia all'interno di un'attività. I fini di gamification, in questo caso, vertono soprattutto su processi di risorse umane o miglioramento della produttività dei propri lavoratori. In questa categoria è anche inclusa la cosiddetta attività di "crowdsourcing": si tratta di dividere un grande problema in parti ridotte, coinvolgendo la propria utenza per risolverli.
- Si parla infine anche di ambiti dove si desidera un *cambiamento di comportamenti* da parte delle persone. Sono generalmente contesti come quelli della salute, del fitness o ancora dell'educazione: la gamification svolge il ruolo di aiutare gli utenti nel trovare la motivazione per svolgere attività in tali contesti.

Al fine di sintetizzare e ben fissare la definizione di "gamification" e dei concetti relativi visti sopra, essi sono ordinati in un diagramma bidimensionale (Figura 2.1). Mentre la prima dimensione distingue elementi relativi al "gioco" da quelli inerenti al "giocare", la seconda si concentra nel dividere esperienze che utilizzando parti di una nozione (come singoli elementi a cui si ispirano) da quelle che le rappresentano interamente.

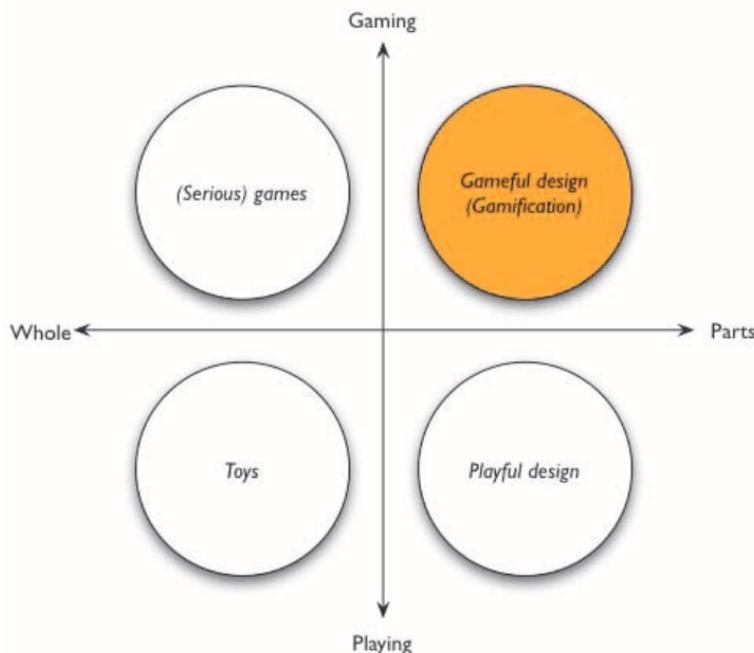


Figura 2.1: Gamification in relazione a concetti affini.

Indipendentemente dal contesto a cui si applicano, la forma più utilizzata di gamification è quella di **sistema di reward**: a seconda delle azioni che effettua un utente sulla piattaforma vengono assegnate ricompense, al fine di dare loro una sensazione di soddisfazione e che le loro decisioni hanno realmente un impatto significativo sul sistema. Più in particolare, un aspetto fondamentale nel game design (e quindi anche nella gamification) è proprio quello della motivazione dei giocatori: senza di essa, gli utenti non sarebbero più interessati a continuare un gioco, ovvero ad effettuare determinate azioni desiderate su una piattaforma. Possiamo distinguere 2 tipologie di motivazione:

- *intrinseca*, il quale riflette un interesse personale o divertimento nel fare un'attività;
- *estrinseca*, ovvero guidata da fattori esterni all'utente (ricevere un compenso economico o avere paura di una perdita).

Entrambi i tipi di motivazione sono molto importanti in un sistema "gamificato": è bene infatti trovare un bilanciamento fra *ricompense tangibili* che l'utente si aspetta dopo aver raggiunto un obiettivo (motivazione estrinseca), e *feedback informativi* (talvolta anche inaspettati) dati al giocatore per un evento di gioco (motivazione intrinseca).

Attualmente, esiste un ampia varietà di elementi e meccaniche di gioco che è possibile applicare per raggiungere gli obiettivi di gamification, sempre in termini di motivazione e per mantenere il più possibile la propria utenza sulla piattaforma. Di seguito vengono elencati i più utilizzati e la loro utilità.

- L'elemento più elementare e conosciuto è quello dei **punti** (o punteggi). Si tratta unità metriche, che permettono ad un progettista di gamification di tenere traccia dell'attività utente ed effettuare eventuali aggiustamenti all'esperienza creata. Esistono molteplici tipi di punti, in base al loro fine nel sistema:
 - *esperienza*: ricompensa senza scadenza, accumulabile e che mostra i progressi di un giocatore;
 - *karma*: tipologia progettata con lo scopo di ricompensare altri utenti, in modo simile ad un sistema di votazione;
 - *reputazione*: si tratta di un punteggio basato sull'omonimo concetto. Più ne vengono accumulati da un utente, maggiore è la fiducia che può ricevere da altri giocatori.
- Seguono i **livelli**, utilizzati per marcare sia il punto del gioco in cui sono arrivati, che per indicare una maggior difficoltà, e quindi sfida. Dal punto di vista grafico sono solitamente accompagnati da una barra di progresso, per dare feedback sul livello successivo da raggiungere.
- Le **classifiche** introducono il fattore competitivo (o di ranking) in un'esperienza di gamification: si tratta di un elenco di nomi di giocatori, ordinato in base ad un criterio (punti guadagnati o foto scattate sono solo alcuni esempi). Questo elemento può essere ulteriormente modellato, suddividendo le classifiche localmente (quindi in base alla posizione geografica) o socialmente (mostrare i soli amici e persone seguite).

- Sono utilizzati i **badge** per rappresentare la fine di un'attività con successo o un obiettivo raggiunto, grazie agli sforzi ed interazioni di uno o più persone nel sistema. Si tratta di ricompense simboliche che possono essere collezionate e anche mostrate ad altri giocatori come status sociale. In alcune esperienze di gamification, questo elemento è usato anche come sistema di livellamento, oppure generati casualmente per incrementare il fattore di sorpresa e quindi di interesse da parte degli utenti.
- Vengono proposte inoltre **sfide** e **quest** per direzionare le azioni di un utente verso un obiettivo specifico sulla piattaforma. L'idea è quello di mantenere i giocatore in una costante situazione di sfida, cercando di minimizzare lo stress e la ripetitività dell'esperienza.

Gli elementi sopra descritti contribuiscono al funzionamento dei 2 processi principali in un sistema di gamification. Il primo viene detto **onboarding**, ovvero l'introduzione di un nuovo giocatore nel sistema: è bene in questo passaggio rendere l'esperienza il più piacevole possibile (talvolta anche più semplice), senza riempire l'utente con troppe informazioni o farlo sentire confuso. L'onboarding è seguito dal processo di **engagement loop**: si tratta della parte dove si agisce maggiormente sulla motivazione dell'utente, cercando di rinnovare continuamente il suo interesse e mantenerlo nella piattaforma. I social network sono un ottimo esempio di engagement loop ben progettato: le notifiche che sono ricevute da un utente quando è menzionato in un contenuto sono pensate proprio per far tornare ad interagire con il sistema.

Infine, al giorno d'oggi le tecniche di gamification giocano un ruolo importantissimo in servizi geo-localizzati. Grazie anche all'esponenziale diffusione di dispositivi mobili dotati di GPS (come gli smartphone), grandi corporazioni come Google e Foursquare hanno creato business incentrati nella raccolta di dati geografici, il tutto presentato all'utenza come una esperienza videoludica con costante engagement. Le piattaforme considerate in questo contesto sono dette *LBS* (*Location Based Service*), le cui funzionalità permettono di:

- raccogliere *VGI* (Volunteered Geographic Information), ossia dati geografici generati e creati volontariamente dagli utenti;
- risolvere *problemi di crowdsourcing*, sfruttando quindi la propria community per distribuire e risolvere problemi e compiti specifici.

Applicare la gamification nel raccoglimento di dati geografici migliora drasticamente l'esperienza in sistemi di questo tipo [15].

2.2 Framework di progettazione della Gamification

Analogamente alla progettazione di un videogioco, anche il design di una strategia di gamification richiede un approccio ben strutturato. Lo scopo principale, attraverso gli elementi di ispirazione videoludica scelti, è di massimizzare i processi di *onboarding* (introduzione dell'utenza nell'esperienza) e di *engagement loop* (rinnovare interesse e motivazione per continuare ad usare la piattaforma). Nel corso del tempo, sono stati proposti numerosi framework e modelli su cui poter basare la propria progettazione, suddivisi in 2 categorie:

- *function-focused*: design incentrato soprattutto su meccaniche e funzionalità di gioco, solitamente più efficiente;
- *human-focused*: l'attenzione nella progettazione si sposta maggiormente verso gli utenti, insieme a cosa effettivamente incide sulle loro motivazioni, emozioni ed ambizioni. Risulta un metodo più efficace, poiché si ottimizza quegli aspetti che mantengono il giocatore su una piattaforma e rendono la sua esperienza piacevole.

Qui di seguito saranno descritti 2 dei framework più popolari ed usati nella gamification, entrambi più inclini ad un design human-focused.

2.2.1 MDA framework

MDA (Mechanics, Dynamics and Aesthetics), nato come un approccio formale per studiare i videogiochi e le loro fasi di realizzazione, è un framework che suddivide la progettazione di un'esperienza di gioco (e quindi anche di gamification) in differenti livelli di astrazione, considerando sia il punto di vista del creatore di contenuti (il *designer*) che i consumatori di essi (il *player*) [10]. Più in particolare, produzione ed utilizzo di questo genere di contenuti è suddiviso in 3 parti.

- Il primo livello di astrazione è quello delle **meccaniche**, il quale descrivono le componentistiche di un gioco a livello di rappresentazione dati e funzionamento. L'esempio cardine in questo contesto è dato da *regole* da rispettare, ossia delle meccaniche che guidano le azioni e i comportamenti dei giocatori. Seguono singoli elementi già descritti fra quelli più comunemente utilizzati in un'esperienza gamificata: punteggi, classifiche, badge e molto altro ancora, tutti con l'obiettivo di descrivere il contesto di gioco.
- Le meccaniche servono a supportare quelle che sono le **dinamiche** di gioco, ovvero il comportamento dei singoli elementi mentre il gioco stesso è in corso e in base sia ad input dati da giocatori, che le interazioni fra questi ultimi. Fra gli esempi più significativi, comuni dinamiche di gioco possono essere:
 - *competizioni*, create grazie ad elementi come la pressione data dal tempo o dallo scalare una classifica per punti guadagnati;
 - *collaborazioni* fra giocatori, realizzata mediante obiettivi in comune raggiunti svolgendo attività altrimenti difficili in solitaria e commemorati con badge unici nel loro genere.

Insieme, le dinamiche possono essere descritte anche come il "sistema" del gioco: si comporta in modo differente a seconda degli input degli utenti, mentre restituisce a questi ultimi feedback come una schermata di vittoria o una nuova sfida da affrontare.

- Come ultimo livello di astrazione troviamo la componente "**aesthetic**", che descrive le risposte emotive dei giocatori quando interagiscono con il sistema. In dettaglio, si occupa di specificare cosa effettivamente rende un gioco "divertente" ed "emozionante", a cosa ogni utente attribuisce valore e lo spinge a continuare ad interagire. I campi dove indagare questi aspetti sono davvero

numerosi: spaziano dall'aspetto narrativo del contenuto, al senso di scoperta ed esplorazione di una nuova piattaforma, o ancora ad elementi social e competitivi.

Meccaniche e dinamiche servono per creare *esperienze aesthetic*: mediante questi, obiettivi cooperativi e gare, vengono generate emozioni (ad esempio, quando si vince o perde) determinanti nello stabilire per un giocatore se un'esperienza "gamificata" continui ad essere interessante.

Come anticipato all'inizio, uno degli aspetti più interessanti e fondamentali del framework MDA è il poter rappresentare sia la prospettiva dei progettisti di esperienze videoludiche, che il giocatore che ne usufruisce. In base al punto di vista preso in considerazione, la percezione dei 3 livelli di astrazione descritti sopra cambia, come mostrato in Figura 2.2.

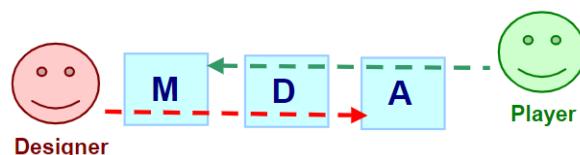


Figura 2.2: Prospettive del MDA framework, dal designer e dal giocatore.

L'ordine scelto per descrivere gli elementi ha seguito la prospettiva del designer: inizialmente vengono scelti gli elementi e le meccaniche che si desiderano nella propria esperienza di gamification, per poi implementarle al fine di realizzare le dinamiche di gioco e studiare quali sono le emozioni che si desidera suscitare. Nel senso opposto troviamo invece l'utente, il quale percepisce inizialmente determinate emozioni (parte *aesthetic*) nate da dinamiche basate su elementi e meccaniche di gioco.

Infine, un ultimo aspetto importante dell'MDA è quello di seguire un *approccio iterativo* nella progettazione. Questo permette, dopo aver analizzato i risultati dell'esperienza di gamification, di perfezionare e bilanciare progressivamente il sistema creato.

2.2.2 Octalysis framework

Octalysis, a differenza del precedente MDA, è un framework che si concentra prevalentemente sulla human-centered design: la progettazione di elementi e dinamiche si basano quindi sul comportamento della propria utenza, così come gli aspetti che guidano maggiormente la loro motivazione.

Il modello è stato creato dal ricercatore Yu-kai Chou, il quale ha impiegato anni nello studio delle strategie impiegate per rendere un videogioco divertente. Il risultato della ricerca è che questo tipo di contenuti risulta piacevole per uno o più aspetti specifici detti *Core Drives*, il quale incentivano un utente in un certo modo a svolgere un'attività. Octalysis si compone di 8 *Core Drives*: rappresentano e definiscono tecniche sempre a fini di motivazione, ciascuno in modo e per cause differenti (Figura 2.3) [13]. In altre parole, qualsiasi attività svolta da un utente è dovuta ad uno o più Core Drives rappresentati sotto.



Figura 2.3: Struttura dell'Octalysis framework.

1. **Epic Meaning & Calling** rappresenta il *Core Drive* dove il giocatore crede che la sua attività sia per una causa maggiore di lui stesso, come se fosse un "prescelto". Con questo in mente, si sente in dovere di spendere il suo tempo per aiutare e creare contenuti per l'intera comunità di cui fa parte, ad esempio quando si contribuisce su Wikipedia o in altri progetti open-source.
2. Il Core Drive **Development & Accomplishment** guida l'utente attraverso la sensazione di soddisfazione: questo grazie al fare progressi, sviluppare nuove capacità o ancora superare sfide. Si tratta anche dell'aspetto più conosciuto ed implementato di gamification, attraverso elementi di gioco quali punteggi, badge o ancora classifiche.
3. **Empowerment of Creativity & Feedback** coinvolge la parte "creativa" di un'esperienza: in altre parole, mantenere l'engagement con l'utente dando la possibilità di immaginare nuovi concetti e provare differenti combinazioni in un ambiente già familiare, inviando loro un responso per le loro ispirazioni. Un esempio che rappresenta alla perfezione il Core Drive sono i mattoncini Lego, il quale aprono ad infinite costruzioni possibili in base a come vengono composti.
4. Grazie al Core Drive **Ownership & Possession** gli utenti vengono motivati dalla sensazione di possedere qualcosa, portandoli a continuare ad effettuare azioni per possedere ancora di più. L'idea principale di questo punto è quello della "ricchezza", mettendo a disposizione ad esempio monete virtuali accumulabili e spendibili. Questi possono poi essere spesi per acquistare elementi per collezione (come dei francobolli) o personalizzare il proprio profilo ed avatar (spendendo molto tempo e sentendo quindi una maggior proprietà ed importanza nei loro confronti).
5. **Social Influence & Relatedness** incorpora l'elemento "social" in una strategia di gamification: tutoraggio da altri giocatori, compagnia, o ancora sensazioni di competizione o invidia nei confronti di altri persone. Ciò che motiva

qualcuno in questo ad essere attivo è, ad esempio, impegnarsi per raggiungere il livello di qualcun' altro, od ottenere nuove abilità esclusive da mostrare ai propri amici. A questo, si aggiunge l'essere attratti da elementi a noi familiari: vedere un posto a noi caro o un prodotto che rimanda alla nostra infanzia genera sensazione di nostalgia, e quindi la probabilità di effettuare un'azione desiderata come un acquisto.

6. **Scarcity & Impatience** incentiva l'utente a voler qualcosa perché non può ottenerla. Una delle tecniche più comuni per rendere questa sensazione è anche detta *Appointment Dynamics*: al giocatore viene visualizzata una ricompensa, ma che potrà ottenere solo dopo un certo numero di ore. Questo induce al pensare a questo per un lungo periodo, proprio perché non si può ricevere ciò che si desidera immediatamente.
7. Il Core Drive **Unpredictability & Curiosity** si occupa della parte di imprevedibilità di un evento che può accadere nel sistema, motivando gli utenti ad usare una piattaforma per la curiosità di sapere cosa succederà dopo. Si tratta dello stesso principio che spinge le persone nel continuare a leggere libri o guardare serie TV, ma anche il motivo dietro la dipendenza dal gioco d'azzardo. Un'azienda utilizza solitamente questo Core Drive per migliorare l'engagement con la propria utenza grazie a programmi di lotteria.
8. Infine, **Loss & Avoidance** incentiva l'attività da parte dei giocatori per evitare una perdita, o in generale che succeda qualcosa di negativo. Gli esempi possono spaziare dall'aver paura di perdere il lavoro che è stato fatto, fino a quella di perdere invece opportunità limitate che non potranno ricevere mai più, a meno che non venga svolta subito un'azione.

Nonostante ogni Core Drive descrive il motivo che ha spinto una persona a fare qualcosa, non è necessario che siano presenti tutti e 8 all'interno di una strategia di gamification. Questo perché, in base all'azienda o al contesto dove viene applicato Octalysis, ci si può concentrare maggiormente su un punto del framework piuttosto che un altro: il suo utilizzo principale è quello di associare per ciascun Core Drive le meccaniche relative che si desidera utilizzare. Secondo quest'ultimo passaggio e quanto sono effettivamente potenti le stesse meccaniche, sarà possibile vedere su quali aspetti della motivazione si concentra la piattaforma, e su quali invece in misura minore o nulla.

2.3 Esempi di Gamification

Come già anticipato riguardo ai contesti di applicazione di tecniche di gamification, il suo utilizzo al giorno d'oggi è davvero esteso e conta molteplici casi di enorme successo. Questo non solo per migliorare le prestazioni di un'impresa (dal punto di vista di engagement e vendite), ma anche per cambiare le abitudini delle persone e motivarle verso attività a cui si è interessati: iniziare un corso per imparare qualcosa di nuovo o ricordarsi di svolgere attività fisica sono 2 di tantissimi casi.

Per dimostrare ancora una volta le potenzialità di un'esperienza "gamificata", di seguito vengono descritti 2 esempi di applicazioni che fanno uso di queste tecniche.

2.3.1 Duolingo

Duolingo è una popolare piattaforma educativa utilizzata per imparare nuove lingue, in modo unico nel suo genere. Si tratta di un app gratuita con servizio in abbonamento (*Duolingo Plus*), il quale permette di rimuovere pubblicità e ottenere altri benefit. Si tratta di un successo di scala internazionale: nel 2021 ha contato ben oltre 500 milioni di utenti e 40 milioni attivi mensilmente, offrendo oltre circa 98 corsi che insegnano 39 lingue differenti.

Uno dei motivi del raggiungimento di questi numeri è proprio grazie a tecniche di gamification. In particolare, Duolingo ha basato la sua strategia sul framework Octalysis, utilizzando tutti e 8 i Core Drives messi a disposizione per motivare l'utente in modi differenti. Qui sotto vengono analizzati i punti sfruttati principalmente, mediante una grande varietà di elementi e tecniche videoludiche.

- Uno dei Core Drives più in evidenza, come molte altre piattaforme "gamificate", è il Core Drive 2 (**Development & Accomplishment**). Duolingo, anche con colori vivaci ed illustrazioni giocose, celebra ogni singolo obiettivo raggiunto dall'utente con tono entusiasta e positivo: che sia per una lezione completata o scalare una classifica, la piattaforma assicura di essere soddisfatti per qualsiasi azione svolta (Figura 2.4).

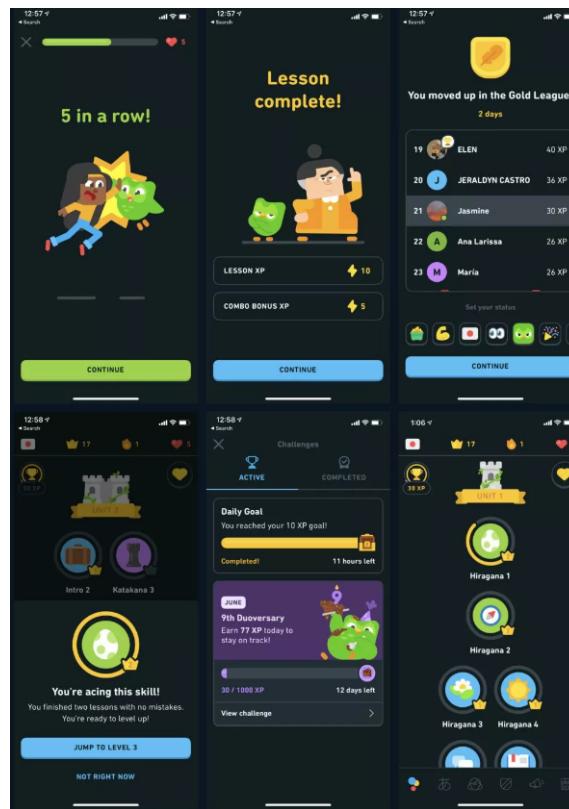


Figura 2.4: Schermate dell'app Duolingo con elementi del core drive 2 di Octalysis.

Le schermate nella figura sopra mostrano numerosi elementi di gamification:

- *Classifiche e leghe*: promuovono o retrocedono ad un determinato livello ed innescano certe emozioni di risposta dall'utente. Infatti, mentre nel primo

caso il giocatore si sente soddisfatto del successo ottenuto, scendere da una graduatoria lo spingerà invece a lavorare più duramente durante le lezioni.

- *Punti esperienza*: dati solitamente per ogni lezione completata, è uno dei modi più semplici per ricompensare la propria utenza.
- *Barre di progresso*: sempre come illustra la Figura 2.4, Duolingo mostra sia per ogni argomento di una lingua che i badge mensili una barra di progresso. Viene riempita nel primo caso man mano che vengono completate le lezioni, e nel secondo con i punti esperienza. In entrambi i casi, il fine è quello di rendere una sensazione di progresso per il proprio lavoro svolto.
- *Serie*: in questo contesto, si tratta di un contatore di giorni con associata una piccola fiamma. Esso è incrementato per ogni giorno di fila in cui l’utente svolge almeno una lezione. Inoltre, dopo un certo numero di giornate (come una settimana), l’utente viene ricompensato per incentivare la propria serie.
- Un altro punto molto utilizzato di Octalysis è quello di **Social Influence & Relatedness**. Duolingo usa funzionalità simili a quelle nei social network, per incoraggiare gli utenti a connettersi fra di loro: aggiungendosi come amici, confrontandosi per progressi settimanali o ancora superandosi in classifiche. Come descritto in precedenza, meccanismi di questo tipo incentivano a rimanere sulla piattaforma per continuare ad interagire con altri giocatori (Figura 2.5).

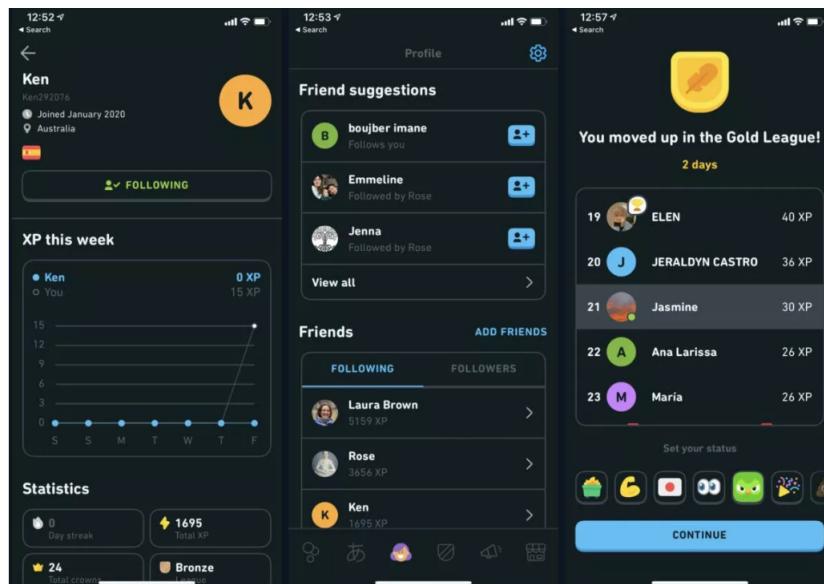


Figura 2.5: Funzionalità social di Duolingo.

- Infine, un Core Drive molto importante nella strategia di Duolingo è quello di **Loss & Avoidance**: motivare tramite la paura di perdere qualcosa induce a prendere decisioni nell’immediato. La meccanica principale che implementa questo principio è quella delle serie di giorni, descritta precedentemente: poiché quest’ultima simboleggia gli sforzi di un utente nell’imparare nuove lingue, rovinare la serie (soprattutto se lunga) verrebbe percepito come un evento “disastroso”. Durante una giornata, se non si è svolta ancora nessuna attivi-

tà, l'app si occupa di avvisare della perdita della serie mediante una notifica (Figura 2.6).



Figura 2.6: Notifica per avvisare gli utenti della perdita imminente.

Se un serie venisse interrotta, Duolingo mette infine a disposizione una scelta: pagare un ammontare per ripristinarla. Nella maggior parte dei casi, per un utente l'acquisto ne varrà la pena, data l'importanza da lui attribuita ai progressi altrimenti rovinati.

Tutti gli elementi fin'ora descritti contribuiscono a rendere Duolingo ciò che è oggi: una piattaforma di e-learning con elevato engagement e una costante sfida, feedback chiari e che soprattutto sia divertente [2].

2.3.2 Waze ☺

Waze, un'altra celebre realtà attuale, è un'app che tramite GPS mostra preziose informazioni stradali in tempo reale: aggiornamenti del traffico, autovelox, posti di blocco e in generale tutto ciò a cui si dovrebbe porre attenzione mentre si guida. Uno degli aspetti più interessanti è che questi dati provengono dagli input di tutti gli utenti della piattaforma, dalla velocità del loro veicolo a segnalazioni di un possibile pericolo su strada. Si tratta in particolare di un *LBS* (Location Based Service): le informazioni vengono raccolte in forma di *crowdsourcing*, ed utilizzati per migliorare la piattaforma ed offrire un miglior servizio.

La gamification entra in gioco proprio per incentivare le persone alla guida ad essere attivi collaboratori. Gli utenti possono accumulare punti per le contribuzioni aggiunte o per i chilometri di strada percorsa, posizionandosi in classifiche anche geo-localizzate e salendo di livello (Figura 2.7).

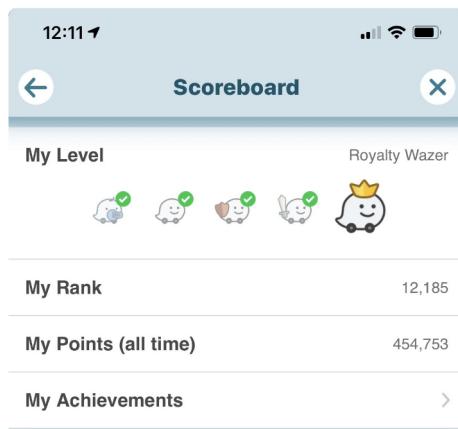


Figura 2.7: Schermata dei livelli, punteggio e posizione in classifica.

Oltre alla senso di progressione, aumentare di grado in Waze permette ad un giocatore di accedere a funzioni esclusive, come poter modificare informazioni sulla composizione della mappa. L'app inoltre cura in modo meticoloso le icone associate ad ogni livello, evidenziando l'aspetto *aesthetic* di tale componente (per ciascuna figura, viene associato un certo valore ed importanza).

Come in Duolingo, un altro aspetto molto importante dell'esperienza "gamificata" è la sua componente social. All'interno della sua mappa, l'applicazione mostra gli altri utenti con posizione approssimativa e sotto forma di un avatar da loro scelto, fra i molti disponibili (Figura 2.8).

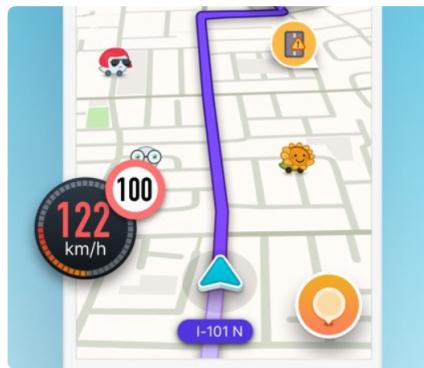


Figura 2.8: Visualizzazione della mappa, con avatar di altri utenti della piattaforma.

Unita a questa funzionalità, gli utenti possono non solo interagire fra di loro, ma connettere social network esterni come Facebook per condividere più facilmente la propria posizione.

Waze, per concludere, utilizza tecniche di gamification per rendere più efficiente la raccolta di dati geografici ed utilizzarli per aumentare efficacia ed efficienza del proprio servizio. Unito ad un modello "freemium" con pubblicità, la piattaforma è diventata un grande successo commerciale ed è stata capace di motivare le persone nei loro momenti più occupati, come durante la guida [12].

2.4 Gamification in FirstLife

Nel corso dell'elaborato di tesi, quello che verrà illustrato e spiegato nel dettaglio è l'implementazione di un nuovo componente di FirstLife, detto **modulo di gamification**. Grazie ad esso, verranno introdotti anche nel social network civico elementi e quindi dinamiche di ispirazione videoludica: lo scopo è di incoraggiare i cittadini della piattaforma ad esplorarla, collaborare fra loro per arricchire contenuti (mediante foto, video e molto altro ancora) e diventare attivi creatori di iniziative ed eventi per il proprio territorio locale. Più in dettaglio, si tratta come nel caso di Waze in un'applicazione LBS: ponendo come sfida quella di mappare e aggiungere valore per la propria zona, le entità create dagli utenti rappresentano VGI raccolti e visualizzati nella mappa per raggiungere lo scopo.

Per permettere di realizzare una prima versione di esperienza "gamificata", si è scelto di basarsi sul framework di progettazione MDA.

- Il primo livello di astrazione delle *meccaniche* vede fra i principali elementi le regole. Come all'interno di un videogioco, l'idea è quella di imporre una serie di

condizioni che, se soddisfatte dall'azione del giocatore, ricompensa quest'ultimo con punteggi, di natura differente a seconda del contesto. Questi elementi sono accompagnati da badge, ricompense simboliche che richiedono un certo ammontare di punti o badge pregressi per essere ottenuti; classifiche, lista di nomi di cittadini ordinata secondo un punteggio da loro accumulato.

- Le *dinamiche* che derivano dal livello precedente sono molteplici. È prevista una forma di sfide da affrontare, singolarmente o in collaborazione con altri cittadini: il giocatore o un insieme di essi effettueranno le azioni proposte dalle regole per ottenere punti, ed essere ricompensati con badge commemorativi dei propri sforzi. Queste sono anche affiancate da competizioni fra utenti: una volta che quest'ultima giunge al termine, in base alla situazione in classifica verranno assegnati i meritati trofei. È presente infine un sistema di livellamento, modellato sulla base della flessibile meccanica dei badge.
- Le dinamiche sopra strutturano l'ultimo livello di astrazione di MDA: la componente "*aesthetic*". Ogni badge è illustrato da un logo, il quale trasmette un certo significato e valore non solo per l'obiettivo per cui è stato vinto, ma anche per il contesto in cui è stato ottenuto (dove ciascun contesto, ad esempio, può essere rappresentato da una verticalizzazione in FirstLife). I motivi appena descritti sono applicati anche nel caso di sfide collaborative e competizioni fra cittadini, soprattutto per la risposta emotiva di questi ultimi. Il tutto, mentre si è immersi in queste attività, è accompagnato dal senso di scoperta dall'esplorazione della mappa della piattaforma e dei contenuti che l'arricchiscono.

Tutti questi elementi rappresentano dunque il modello concettuale di gamification in FirstLife, con il fine di migliorare l'engagement dell'utenza e la loro motivazione nel contribuire per il proprio territorio.

Grazie alle nozioni acquisite inerenti la gamification e le attività possibili sulla piattaforma, è stato strutturato un percorso ideale detto **Gamification Journey**. Il suo obiettivo è di portare ciascun utente dallo stato di cittadino "non attivo" a quello di attivo collaboratore e creatore di contenuti. Esso prevede una serie di step, ciascuno dei quali corrisponde ad attività ed azioni desiderate da stimolare, ricompensare e, quindi, da "gamificare" (Figura 2.9).

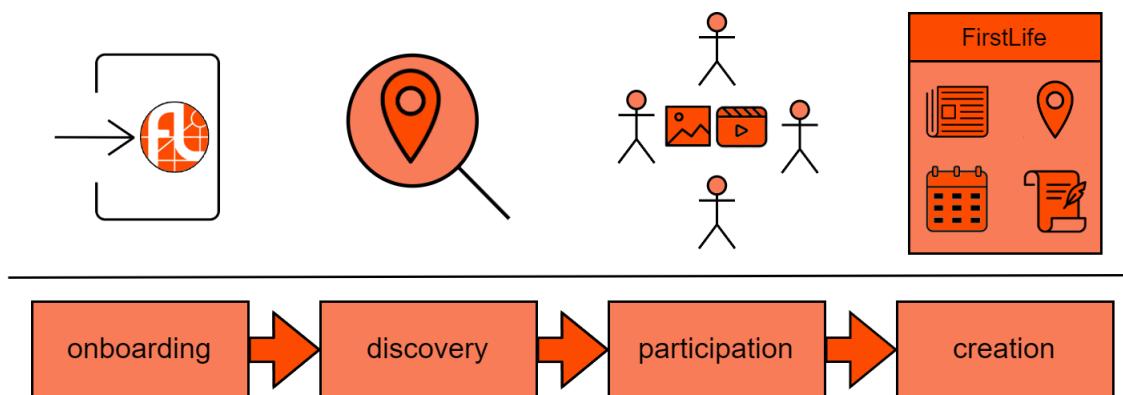


Figura 2.9: Percorso di un utente FirstLife in una strategia di gamification.

1. **Onboarding:** prima parte del percorso, rivolta ai nuovi utenti della piattaforma. Comprende tutte le azioni preliminari per iniziare ad usare FirstLife:

registrazione del proprio profilo, sua personalizzazione e completamento del tutorial.

2. **Discovery:** una volta entrato, nella seconda fase l'utente comincerà ad esplorare e adattarsi in FirstLife. Questo comprende la scoperta dei contenuti nel "wall", la selezione della propria zona sulla mappa o ancora filtrare le entità secondo criteri specifici. Queste operazioni andranno a formare progressivamente quelle che sono le preferenze del cittadino, comprese le tipologie di contenuti a cui è interessato contribuire.
3. **Partecipation:** fase che segue l'ambientazione dell'utente e che comprende una sua prima forma di partecipazione "attiva". Le azioni correlate riguardano soprattutto collaborazioni con altri cittadini e arricchimento di contenuti già esistenti: aggiunta di foto o altri contenuti multimediali ad entità di prim'ordine (come un evento o una notizia), partecipare ad un gruppo o ancora ad una discussione di comune interesse.
4. **Creation:** step finale del Gamification Journey, dove si può trovare il massimo livello di engagement sul social network civico. In questo punto, gli utenti effettuano le attività più impegnative e di valore in FirstLife: dal lancio e proposta di nuovi progetti a creazione di entità più complesse, come eventi, storie o ancora nuovi gruppi collaborativi.

Per mantenere questo livello di engagement con gli utenti sono anche previste le cosiddette *ricompense indirette*, ossia punti assegnati se altri cittadini interagiscono con entità create da quelli più attivi. In questo modo, essi percepiscono ancor di più l'importanza di ciò che hanno realizzato per la piattaforma.

A differenza della maggior parte delle piattaforme su cui è applicata la gamification, un aspetto particolare nel caso di FirstLife è che il fine della strategia può variare molto a seconda della verticalizzazione che ne fa utilizzo, pur mantenendo il percorso Gamification Journey appena illustrato. Si pensi a progetti esterni come quelli visti nel capitolo precedente: lo scopo in *Miromap* può essere quello di incoraggiare i cittadini nel segnalare attivamente eventuali problemi dell'omonimo quartiere; invece, l'esperienza "gamificata" in *EduLife* cerca di motivare gli adolescenti nel condividere le proprie considerazioni, proposte ed iniziative su tematiche di pubblica utilità, basandosi sul loro territorio locale di residenza.

Il modulo di gamification è stato progettato considerando la flessibilità di FirstLife. Si tratta non solo di un componente che ricompenserà gli utenti per determinate azioni, ma metterà anche a disposizione un'interfaccia dove gli amministratori della propria verticalizzazione potranno costruire la propria strategia, creando regole e badge su cui basare l'esperienza di ispirazione videoludica.

Capitolo 3

Tecnologie utilizzate

Per lo realizzazione del modulo di gamification, le tecnologie sono state scelte basandosi su 2 criteri. Il primo rappresenta un'attinenza alle best practises nello sviluppo software, grazie soprattutto all'utilizzo di framework per un maggior supporto nel lavoro, dal debugging alla documentazione del codice. Il secondo invece si concentra nel massimizzare la scalabilità e la manutenibilità del progetto, al fine di una maggior flessibilità in correzione di bug ed implementazioni future di nuove funzionalità.

Come descritto qui di seguito, le tecnologie impiegate possono inoltre essere suddivise sulla base dell'area di impiego e sviluppo.

- **Persistenza dati:** concetto riferito alla sopravvivenza e salvataggio dei dati anche al termine dell'accesso da parte del programma, in supporti non volatili come un file-system o un database.
- **Back-end:** componente che include la logica di business di un software, così come la gestione, la manipolazione e l'elaborazione dei suoi dati. Esempi di compiti di backend possono essere il salvataggio di record all'interno di un database o l'esposizione di *Application Programming Interfaces (APIs)* per poter accedere ai servizi che offre la piattaforma.
- **Front-end:** parte del sistema dedicata alla presentazione dei dati e all'interazione con l'utente. In quest'area sono compresi tutti gli elementi che compongono l'interfaccia grafica di un programma, in aggiunta ormai sempre più spesso ad elaborazioni dati intermedie per migliorare la *user experience (UX)*.

3.1 MongoDB

MongoDB è un database non-relazionale document-oriented, progettato per offrire elevate prestazioni, scalabilità ed affidabilità nello sviluppo di un software. La scelta della piattaforma per la persistenza dei dati è dipesa dai seguenti motivi [11].

- A differenza di una base di dati relazionale i record di dati in MongoDB sono rappresentati da **documenti** con struttura chiave-valore, organizzati all'interno di **collezioni** (equivalente semantico di "tabelle" in un RDBMS). I documenti in particolare sono memorizzati sotto forma di *oggetti BSON*, rappresentazione binaria di oggetti JSON in grado non solo di contenere più tipi di dati, ma anche di essere più "leggeri" nella trasmissione. Ogni documento è identificato da un oggetto ObjectId, un tipo valore composto da 12bytes efficiente da generare.

```

1  {
2    _id: ObjectId("5099803df3f4948bd2f98391"),
3    name: { first: "Alan", last: "Turing" },
4    birth: new Date('Jun 23, 1912'),
5    death: new Date('Jun 07, 1954'),
6    contribs: ["Turing machine", "Turing test", "Turingery"],
7    views: NumberLong(1250000)
8  }

```

Listing 1: Esempio di documento in MongoDB.

Dalle caratteristiche descritte sopra derivano molti vantaggi rispetto ad un database SQL. Il primo è l'assenza di uno schema fisso, anche per documenti in una stessa collezione, dando quindi una grande flessibilità nella modellazione dei dati. Segue la possibilità di poter rappresentare strutture complesse (come alberi) e ridurre la necessità di query costose (come le operazioni di join), grazie a tipi di oggetti nestati ed array messi a disposizione dai documenti BSON (Figura 3.1).

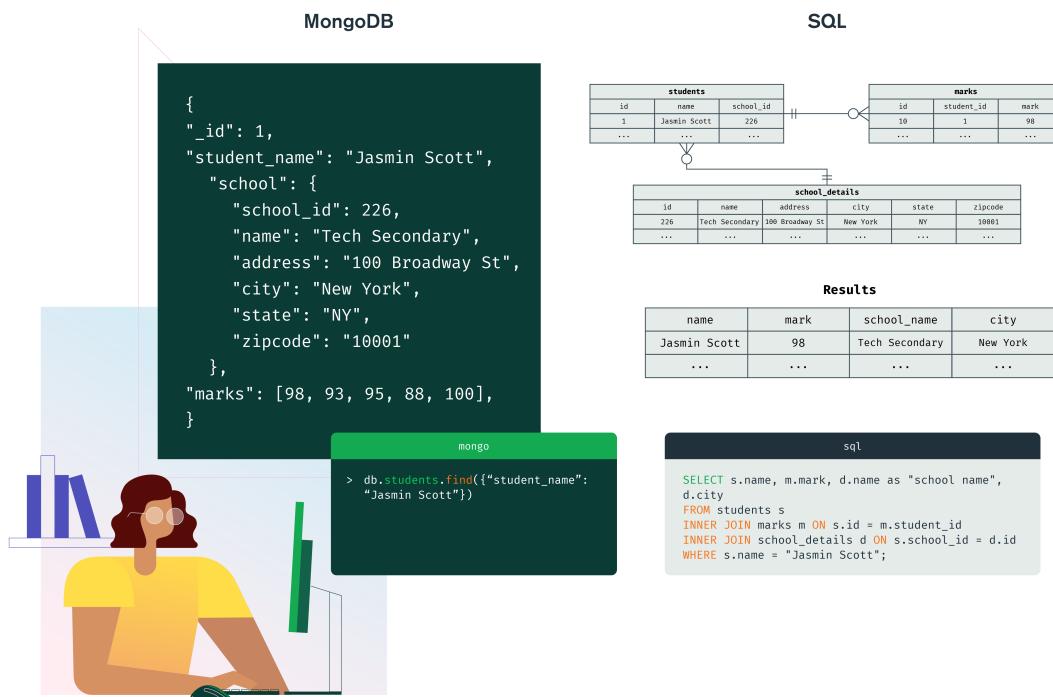


Figura 3.1: Database document-oriented e relazionale a confronto.

- Segue la semplificazione di accesso e gestione dei dati grazie alle *Query API*. Si tratta di una serie di metodi che supportano tutte le operazioni di lettura e scrittura del database (si parla di operazioni CRUD, Create, Read, Update, Delete), anche su più documenti. Un'interessante operazione messa a disposizione è quella di *Data Aggregation* (mediante la funzione `aggregate()`): permette elaborazioni complesse di documenti, anche da collezioni diverse, suddivise in sotto-operazioni dette *stage*.

- MongoDB offre un' elevata affidabilità e ridondanza mediante i *replica set*, ovvero una serie di processi che mantengono lo stesso set di dati. Unita ad una buona scalabilità orizzontale distribuendo gli stessi dati su più macchine (questo metodo è anche detto *sharding*).

Infine, un' ulteriore caratteristica fondamentale è il supporto alle transazioni. Mediante le sue *Transaction API*, permette quindi di poter eseguire operazioni multi-dокументo (anche se appartengono a collezioni diverse) come un'unica operazione atomica, che rispettino le proprietà logiche ACID (Atomicity, Consistency, Isolation, Durability).

3.2 NestJS

NestJS (o Nest) è un framework per applicazioni server-side (back-end) in ambiente Node.js. Il suo principale obiettivo, con l'aumentare di framework e librerie Javascript negli ultimi anni, è di standardizzare l'architettura software fornendone una out-of-the-box, anche basata su moduli. Questo permette non solo lo sviluppo di applicazioni efficienti, ma anche scalabili, facilmente manutenibili e altamente testabili.

Il framework è scritto e supporta il linguaggio Typescript¹ (pur mantenendolo anche per Javascript), si basa sul popolare framework Express² ed è platform-agnostic. Permette in altre parole di sviluppare componenti riutilizzabili in più tipi di applicazioni e contesti.

Nest fa utilizzo di classi e decorator, per definire un elemento e aggiungere i metadata necessari per essere riconosciuto e mappato. Questo in aggiunta al design pattern *Dependency Injection (DI)*, che permette di usare funzionalità delegate in determinate classi in altre parti dell'applicativo, come altre classi o moduli. Il fine è di incrementare flessibilità e modularità nello sviluppo [14].

3.2.1 Elementi principali

Controllers

Il primo elemento fondamentale che si incontra nel framework è quello di controller: ciascuno di essi è responsabile della gestione di specifiche richieste HTTP da parte dei client e di fornire loro una response. Per definire un controller viene utilizzato il decorator `@Controller()`, dove è possibile specificare come parametro il prefisso route³. Seguono `@Get()`, `@Post()` e così per tutti i tipi di richieste HTTP: apposti sui metodi della classe controller, sono i decorator utilizzati per definire gli endpoint delle API.

¹Versione di Javascript con supporto alla tipizzazione statica e strutture avanzate, come interfacce od enumerazioni.

²Framework Node.js molto minimale e con un vasto insieme di librerie a disposizione.

³Una "route" è un endpoint mappato da Nest, in modo da poter determinare da quale API deve essere soddisfatta una richiesta.

```

1 import { Controller, Get, Post } from '@nestjs/common';
2
3 @Controller('cats')
4 export class CatsController {
5   @Post()
6   create(): string {
7     return 'This action adds a new cat';
8   }
9
10  @Get()
11  findAll(): string {
12    return 'This action returns all cats';
13  }
14}

```

Listing 2: Esempio di controller.

In aggiunta ai decorator visti sopra, Nest ne mette a disposizione altri per programmare un endpoint ed il suo comportamento in tutti i suoi aspetti: attributi dell'header, parametri di query e di path, status code della response e molto altro ancora.

Providers

I provider sono classi che possono essere "iniettate" in altre, sfruttando la Dependency Injection: l'obiettivo è fare in modo che, in classi come un controller, si possano delegare compiti complessi ad altri elementi dell'applicazione. I provider più importanti in Nest sono i *services*, responsabili dell'ottenimento di dati da un database o della loro gestione.

```

1 import { Injectable } from '@nestjs/common';
2 import { Cat } from './interfaces/cat.interface';
3
4 @Injectable()
5 export class CatsService {
6   private readonly cats: Cat[] = [];
7
8   create(cat: Cat) {
9     this.cats.push(cat);
10  }
11
12  findAll(): Cat[] {
13    return this.cats;
14  }
15}

```

Listing 3: Esempio di service.

Nell'esempio, sul service è apposto il decorator `@Injectable()`. `@Injectable()` indica che una determinata classe può essere "iniettata" in un'altra seguendo il design DI.

Moduli

Come detto in precedenza, uno dei punti di forza di Nest è la possibilità di poter organizzare il proprio applicativo in moduli. Un modulo è definito usando il decorator `@Module()`: all'interno sono descritte informazioni fondamentali come i controller instanziati nel modulo, i suoi provider e ancora i suoi import ed export.

```

1 import { Module } from '@nestjs/common';
2 import { CatsController } from './cats.controller';
3 import { CatsService } from './cats.service';
4
5 @Module({
6   controllers: [CatsController],
7   providers: [CatsService],
8 })
9 export class CatsModule {}
```

Listing 4: Esempio di modulo.

Alla creazione di una nuovo progetto, il framework genererà in automatico un modulo detto *root module* (o *modulo root*). Ogni applicazione Nest deve possedere almeno un root module: è responsabile di avvio e costruzione di relazioni e dipendenze, tra gli altri moduli e classi create.

Exception filters

Nest di default fornisce un sistema di gestione di eccezioni, in grado di restituire un response user-friendly all'utente in caso di errore. Questo in aggiunta ad un insieme di classi per gestire gli errori HTTP più comuni, da `BadRequestException` a `ForbiddenException`.

Tuttavia, nello sviluppo software, si può avere la necessità di gestire le eccezioni in modo personalizzato. Gli exception filters vanno incontro a questa esigenza: si tratta di classi che permettono di prendere il controllo del flusso dati in caso di errori, aggiungendo logica business o ancora costruendo una risposta json custom. Alla classe di un exception filter viene aggiunto il decorator `@Catch()`, che può avere come parametro il tipo di eccezione che deve catturare. Nel caso non fosse specificato il parametro, verranno catturati tutti i tipi di eccezioni.

```

1 import {
2   ExceptionFilter, Catch,
3   ArgumentsHost, HttpException
4 } from '@nestjs/common';
5 import { Request, Response } from 'express';
6
7 @Catch(HttpException)
8 export class HttpExceptionFilter implements ExceptionFilter {
9   catch(exception: HttpException, host: ArgumentsHost) {
10    const ctx = host.switchToHttp();
11    const response = ctx.getResponse<Response>();
12    const request = ctx.getRequest<Request>();
13    const status = exception.getStatus();
14
15    response
16      .status(status)
17      .json({
18        statusCode: status,
19        timestamp: new Date().toISOString(),
20        path: request.url,
21      });
22  }
23 }

```

Listing 5: Esempio di exception filter.

In questo caso specifico la classe passa in `@Catch()` è `HttpException`, classe padre di tutte gli altri tipi di eccezioni: in altre parole, il filter catturerà di i tipi di exception HTTP. Il metodo `catch(exception, host)` è quello eseguito in caso di cattura di un eccezione, con la logica di business aggiunta dallo sviluppatore.

Per poter infine utilizzare l'exception filter nella nostra applicazione, è necessario utilizzare il decorator `@UseFilters()` (il processo viene anche detto *binding*). Viene passato come parametro un'istanza della classe filter a cui siamo interessati o la sua sola classe, sfruttando la DI. In base a dove verrà apposto il decorator, l'elemento sarà applicato a diversi livelli: il filter può essere valido a livello di singolo endpoint, di controller o globalmente in tutto il progetto. Il concetto di livello di applicazione è valido anche per tutti gli altri tipi di elementi che verranno visti qui di seguito.

Pipes

Le pipe sono classi che hanno principalmente 2 utilizzi:

- *trasformazione* di dati in input (applicando modifiche o facendo cast di tipo, come da intero a stringa);
- *validazione* di dati in input, lanciando un eccezione se non sono soddisfatti determinati criteri.

Come per i filter, anche in questo caso Nest mette a disposizione classi pipe di default che coprono molti use case, da `ValidationPipe` a `ParseEnumPipe`. È possibile in

ogni caso creare un propria pipe custom, estendendo l'interfaccia `PipeTransform<InputType, ConvertedType>`.

```

1 import {
2     PipeTransform, Injectable,
3     ArgumentMetadata, BadRequestException
4 } from '@nestjs/common';
5
6 @Injectable()
7 export class ParseIntPipe implements PipeTransform<string, number> {
8     transform(value: string, metadata: ArgumentMetadata): number {
9         const val = parseInt(value, 10);
10        if (isNaN(val)) {
11            throw new BadRequestException('Validation failed');
12        }
13        return val;
14    }
15 }
```

Listing 6: Esempio di pipe.

Nell'esempio, la stringa di input viene serializzato ad intero e viene validato, verificando sia un valore numerico.

Per utilizzare una pipe, nella maggior parte dei casi il binding viene fatto con il decoratore `@UsePipes()`.

Guards

Le guards rappresentano classi con una singola responsabilità, utilizzate per determinare se una richiesta debba essere gestita o meno. Seguendo questo principio, gli utilizzi più comuni di una guard sono quelli di autenticazione ed autorizzazione nel chiamare un endpoint. In Nest, può essere creata una propria guard implementando l'interfaccia `CanActivate`.

```

1 import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
2 import { Observable } from 'rxjs';
3
4 @Injectable()
5 export class AuthGuard implements CanActivate {
6     canActivate(
7         context: ExecutionContext,
8     ): boolean | Promise<boolean> | Observable<boolean> {
9         const request = context.switchToHttp().getRequest();
10        return validateRequest(request);
11    }
12 }
```

Listing 7: Esempio di guard.

Come mostrato nell'esempio sopra, all'interno del metodo `canActivate()` viene inserita la logica applicativa della guard. Ritornerà un oggetto di Promise od Observable con valore booleano, in base al quale verrà concessa o negata l'esecuzione del metodo di endpoint. Per poter utilizzare una guard, è necessario effettuare il binding con il decorator `@UseGuard()`.

3.2.2 Mongoose

Mongoose è una libreria utilizzata per poter interagire con un database MongoDB, dalla connessione ad esso alla gestione dati. Si tratta di uno dei metodi più popolari e supportati da Nest, tanto da sviluppare anche un package dedicato per il framework in questione.

Mongoose è considerato un Object Data Modelling (ODM), ovvero modella i dati sulla base di una classe specifica. La libreria sfrutta questo principio grazie classi dette *Schema*: ciascuna di esse mappa una collezione in MongoDB e definisce la struttura dei documenti al suo interno. Gli Schema vengono poi utilizzati per definire i *Models*, oggetti responsabili per le operazioni di lettura e scrittura nel database.

```

1 import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
2 import { Document } from 'mongoose';
3
4 export type CatDocument = Cat & Document;
5
6 @Schema()
7 export class Cat {
8     @Prop()
9     name: string;
10
11    @Prop()
12    age: number;
13
14    @Prop()
15    breed: string;
16 }
17
18 export const CatSchema = SchemaFactory.createForClass(Cat);

```

Listing 8: Esempio di Schema con Mongoose.

Come illustrato sopra, grazie al package dedicato di Mongoose (`@nestjs/mongoose`), è possibile utilizzare i tipici decorator di Nest: `@Schema()` è quello usato per marcare una classe come Schema, seguito da `@Prop()` per indicare gli attributi presenti nella struttura del documento nel database. Vengono infine creati `CatDocument`, ovvero il tipo effettivo nel Model, e `CatSchema`, lo Schema che verrà applicato nella collection di MongoDB.

Una volta che viene registrato lo Schema, è infine possibile "iniettare" il model della collezione con il decorator `@InjectModel()`.

3.2.3 Repository Pattern

Con l'incrementare della complessità di un'applicazione, centralizzare le funzionalità di accesso dati e disaccoppiare tecnologie di backend e database risulta essenziale. Queste necessità sono soddisfatte da un design pattern molto popolare nello sviluppo software, ovvero il Repository pattern. Il concetto si basa su una classe detta *classe repository*, che permette di astrarre il livello di accesso dati per separarlo dal resto della logica business [1].

In Nest le repository sono considerate come provider, con il Model della collezione di riferimento istanziato per DI. La strategia migliore è la creazione di una classe generica astratta, estesa per ogni repository necessaria (solitamente una per collection di MongoDB).

```

1 import { Model, FilterQuery, UpdateQuery} from 'mongoose';
2
3 export abstract class AbstractRepository<TDocument> {
4
5   constructor(protected readonly model: Model<TDocument>) {}
6
7   async create(document: TDocument): Promise<TDocument> {
8     return await new this.model(document).save();
9   }
10
11  async insertMany(documents: TDocument[]): Promise<TDocument[]> {
12    return await this.model.insertMany(documents);
13  }
14
15  async find(query: FilterQuery<TDocument>): Promise<TDocument[]> {
16    return await this.model.find(query);
17  }
18
19  async findOne(query: FilterQuery<TDocument>): Promise<TDocument> {
20    return await this.model.findOne(query);
21  }
22
23  async findOneAndUpdate(
24    query: FilterQuery<TDocument>,
25    update: UpdateQuery<TDocument>,
26  ): Promise<TDocument> {
27    return await this.model.findOneAndUpdate(query, update);
28  }
29
30  async deleteOne(query: FilterQuery<TDocument>): Promise<TDocument> {
31    return await this.model.deleteOne(query);
32  }
33}
```

Listing 9: Esempio di classe astratta per repository.

Per accedere infine effettivamente i dati, le repository che estendono `AbstractRepository` verranno inserite per DI nei service corrispondenti. Saranno infatti i service a richiamare i metodi della repository, nel momento in cui è necessario interagire con il database.

3.2.4 Class validator/transformer

I class validator e transformer sono 2 librerie utili nel gestire validazione, trasformazione e serializzazione di dati. I package che le implementano sono chiamati rispettivamente *class-validator* e *class-transformer*. Esse risultano ben integrate in Nest per l'utilizzo di alcuni suoi componenti analizzati sopra.

class-validator fa utilizzo di decorator per impostare la validazione, sfruttando il vantaggio delle pipe nel leggere i metadati dei valori elaborati. I decorator verranno applicate alle proprietà della classe che si desidera usare per gestire i dati in input⁴.

```

1 import { IsEmail, IsNotEmpty } from 'class-validator';
2
3 export class CreateUserDto {
4     @IsEmail()
5     email: string;
6
7     @IsNotEmpty()
8     password: string;
9 }
```

Listing 10: Esempio di classe con decorator di class-validator.

I decorator utilizzati nell'esempio, `@IsEmail()` e `@IsNotEmpty()`, verificano rispettivamente che la stringa sia una email e che non sia vuota.

class-transformer fa utilizzo dei decorator nella stessa modalità del package precedente, ma con l'obiettivo di trasformare e serializzare dati secondo un certo criterio.

```

1 import { Exclude, Transform, Type } from 'class-transformer';
2
3 export class UserEntity {
4
5     @Type(() => string)
6     id: number;
7
8     @Exclude()
9     password: string;
10
11    @Transform(({ value }) => value.name)
12    role: RoleEntity;
13 }
```

Listing 11: Esempio di classe con decorator di class-transformer.

⁴Questa tipologia di classi viene detta anche DTO (Data Transfer Object).

I decorator applicati nell'esempio sono descritti qui di seguito.

- `@Type()`: effettua il cast del tipo passato in input in quello definito nella funzione passata come parametro (in questo caso, il valore sarà convertito in stringa).
- `@Exclude()`: esclude una determinata proprietà in un oggetto. Questo risulta molto utile quando si vogliono nascondere dati sensibili o futili in una risposta HTTP, come la password di un utente.
- `@Transform()`: converte il valore della proprietà secondo la funzione passata come parametro. In questo caso specifico il tipo della proprietà era la classe `RoleEntity`: il decorator si occupa di convertirne il valore ritornando il solo campo "name".

3.2.5 Cron jobs

I cron jobs permettono di eseguire una porzione di codice periodicamente e ad intervalli prefissati. La terminologia deriva dai sistemi operativi Unix-like, che ne fanno utilizzo con il package `cron` per schdulare determinati compiti. La ricorrenza o intervalli di un cron job viene definita dal cosiddetto `crontab`: si tratta un pattern solitamente formato da 5 campi, organizzati secondo i criteri rappresentati qui sotto (Figure 3.2) [20].

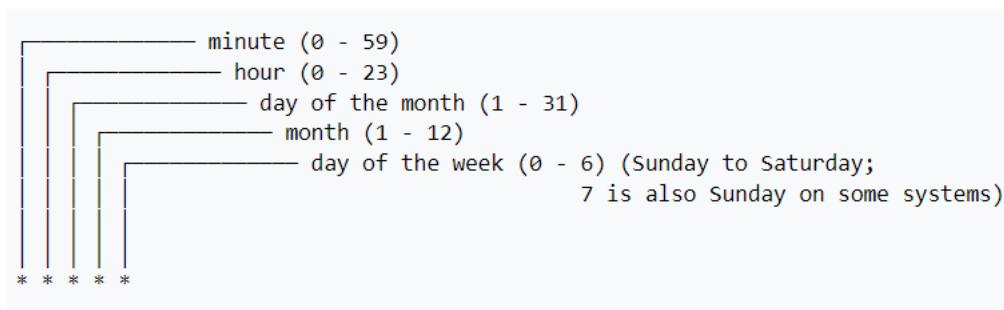


Figura 3.2: Struttura di un crontab.

In alcuni casi può essere presente un sesto campo, davanti i 5 illustrati sopra, che specifica i secondi di un cron job. Oltre a valori numerici è possibile inserire altri caratteri, come "-" per indicare intervalli o "/" per dichiarare una periodicità.

In NestJS, grazie al package `@nestjs/schedule`, è possibile creare cron jobs. Viene introdotto il decorator `@Cron()`, che applicato su un metodo permette di schedulare l'esecuzione del codice al suo interno. `@Cron()` richiede come parametro un crontab, per indicare quando e ogni quanto il job deve essere eseguito.

```

1 import { Injectable } from '@nestjs/common';
2 import { Cron } from '@nestjs/schedule';
3
4 @Injectable()
5 export class TasksService {
6
7   @Cron('0 30 11 * * 1-5')
8   handleCron() {
9     console.log('Called from Monday to Sunday at 11:30:00am');
10  }
11}

```

Listing 12: Esempio service che fa utilizzo di un cron-job.

3.3 Postman e Swagger

A supporto dello sviluppo backend con NestJS, sono stati utilizzati i tool Postman e Swagger.

Postman è un software per costruire, testare ed usare API. Offre diverse funzionalità per coprire tutte le fasi della loro ciclo di vita, dalla progettazione alla documentazione; questo insieme alla possibilità di misurare le loro performance e di condividere l'ambiente di lavoro, per collaborare nella propria organizzazione [18].

La feature più utilizzata di Postman è *API client*: permette di fare debug e test delle proprie API mediante un'interfaccia grafica, con supporto ai formati più utilizzati (come REST o SOAP). Oltre il riconoscimento automatico del linguaggio usato per le response, API client permette di definire una richiesta in tutti i suoi aspetti, da parametri di query a token per autenticazione (Figura 3.3).

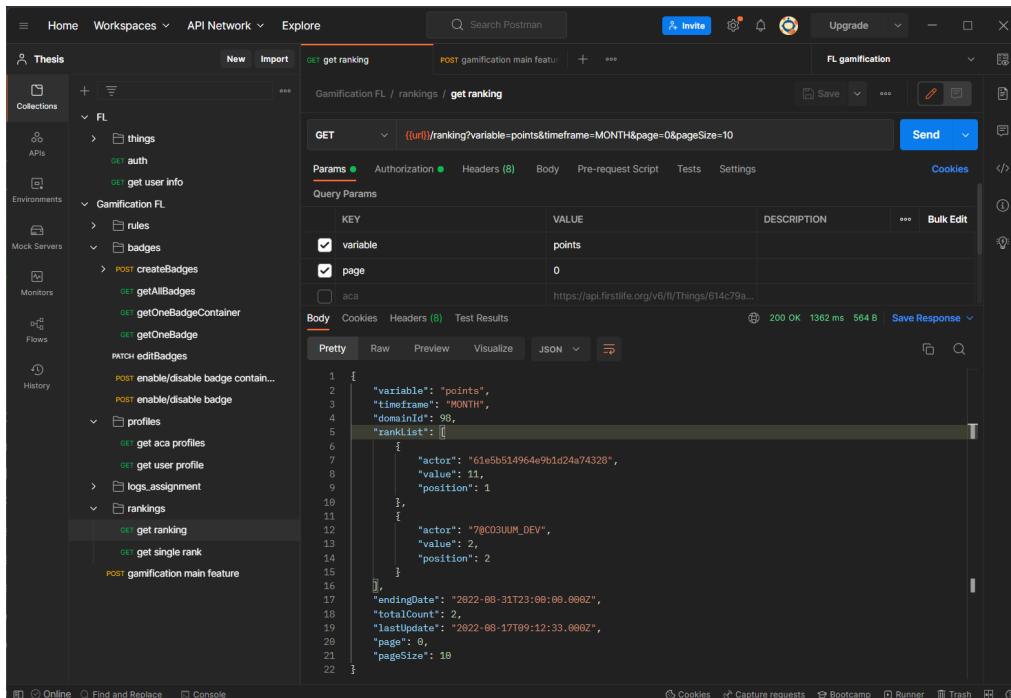


Figura 3.3: Schermata di API Client di Postman.

Come è possibile vedere sopra, le richieste possono essere salvate ed organizzate in cartelle dette *collection*.

Swagger è stato invece utilizzato per la documentazione di API, utilizzando JSON come formato di scambio dati. Si tratta di uno strumento che fa utilizzo dello standard *OpenAPI Specification (OAS)* sia per descrivere RESTful API in tutti i loro aspetti (dal tipo di endpoint agli attributi di query necessari), che generare la documentazione per queste ultime. È inoltre uno standard *language-agnostic*, ovvero indipendente dal linguaggio di programmazione usato per la logica delle API [19].

In Nest è possibile utilizzare Swagger grazie al suo package `@nestjs/swagger`: contiene un insieme di decorator per documentare sia gli endpoint sviluppati che i DTO usati come struttura del body di una richiesta. All'avvio di un applicativo NestJS, verrà generata la pagina di documentazione (Figura 3.4).

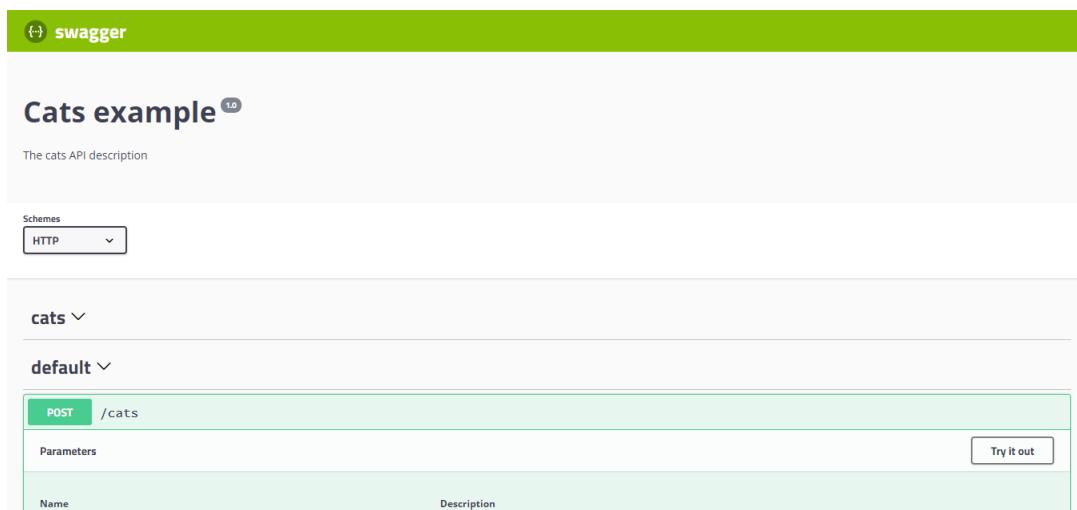


Figura 3.4: Esempio di documentazione generata da Swagger.

3.4 Angular

Angular è un framework front-end typescript, sviluppato da Google per la creazione di applicazioni web. Utilizzando il linguaggio typescript, HTML e CSS, permette la scrittura di SPA (Single Page Application) altamente performanti e scalabili, oltre che responsive [7].

L'architettura di NestJS è profondamente ispirata da Angular: infatti rappresentano insieme un ottima combinazione per lo sviluppo di un intero applicativo. Qui di seguito sono elencati i principali punti in comune.

- **Ampio utilizzo di decorator nelle classi.** Come nel framework backend, sono utilizzati per fornire metadati alle classi su cui sono applicati. In questo modo, Angular comprende il tipo delle classi stesse e come verranno utilizzate nel progetto.
- **Dependency Injection e services.** DI e di conseguenza provider risultano sempre fondamentali, per dare la possibilità a classi di delegare compiti particolarmente onerosi ad altre. Viene utilizzato il decorator `@Injectable()` per indicare che una classe può essere "iniettata".

I provider più importanti rimangono i services, creati per compiere task ben definiti. Use case comuni sono il recupero di dati tramite chiamate server, piuttosto che la validazione di input utente.

- **Moduli.** Anche in questo contesto, per raggiungere un' elevata modularità e scalabilità, un progetto può essere organizzato a moduli. Ciascuno di essi è definito utilizzando il decorator `@NgModule()`, il quale descrive dichiarazioni, provider, import ed export del modulo stesso.

Sempre in modo affine a Nest, ogni applicazione Angular deve avere almeno un modulo iniziale, detto *root module*: è incaricato dell'avvio dell'app, oltre che di risolvere relazioni e dipendenze fra classi e moduli creati.

3.4.1 Componenti, direttive e data binding

I **componenti** rappresentano i blocchi che compongono effettivamente l'applicazione Angular: la loro funzione è quella di definire e renderizzare una view, ovvero un insieme di elementi grafici a livello di interfaccia. Ciascuno di essi è formato dai seguenti elementi:

- *un template HTML* che definisce il come e la struttura della view visualizzata;
- *una classe typescript*, ovvero la parte interattiva del template, il quale può modificare gli elementi a schermo in base alla logica applicativa e i dati al suo interno;
- *un foglio di stile*, incaricato di definire l'aspetto della view (e quindi degli elementi HTML presenti).

Un' app deve contenere almeno un componente, ovvero il *componente di root*. A partire da questo, i componenti possono essere organizzati secondo una struttura gerarchica, che rappresenta il DOM (Document Object Model) della pagina web: ogni componente infatti è considerato come elemento individuale, permettendo di gestire e nascondere parti della UI singolarmente (Figura 3.5).

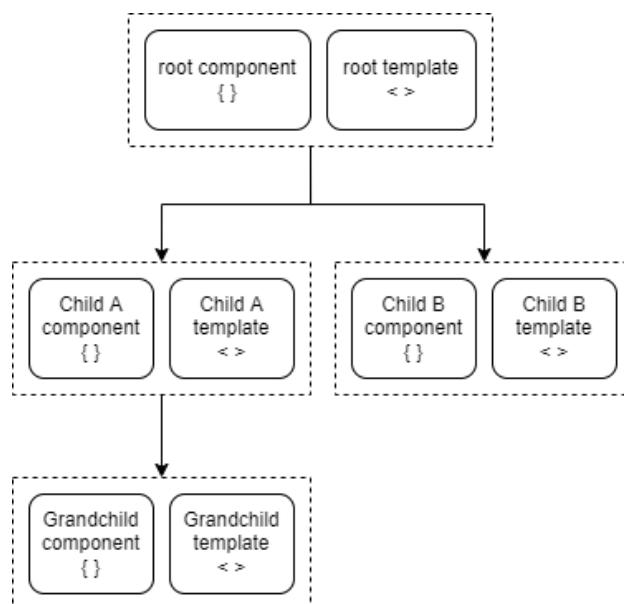


Figura 3.5: Esempio di gerarchia di componenti in Angular.

Per definire un componente, sulla classe typescript viene applicato il decorator `@Component()`, il quale fornisce template e metadati necessari per riconoscerlo.

```

1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'hero-list',
5   template: `
6     <h2>Hello World</h2>
7     <p>This is Hero component list!</p>
8   `,
9   styleUrls: ['./hero.component.scss'],
10  providers: [HeroService]
11 })
12 export class HeroListComponent implements OnInit {
13   constructor(
14     private readonly heroService: HeroService
15   ) {}
16
17   ngOnInit(): void {
18     /* code executed after component rendering */
19   }
20
21   /* All component business logic and data */
22 }
```

Listing 13: Esempio di componente.

Qui di seguito la descrizione dei campi definiti da `@Component()`.

- **selector**: selettori CSS, utilizzati da Angular per comprendere dove creare un'istanza del componente. Nel caso dell'esempio, quando un componente desidera utilizzare questo nel proprio template, dovrà utilizzare i tag `<hero-list>` `</hero-list>`.
- **template**: il template HTML del componente. Il campo può essere rimpiazzato da `templateUrl` per indicare che proviene da un file HTML esterno.
- **styleUrls**: url di tutti i fogli di stile utilizzati. È possibile utilizzare diversi formati: CSS, SCSS o ancora SASS.
- **providers**: elenco di tutti i provider di cui il componente necessita nella propria logica applicativa. I service stessi verranno poi "iniettati" per Dependency Injection.

All'interno di un template, oltre alla normale sintassi HTML, viene introdotta la cosiddetta *sintassi di template*: essa modifica la struttura di una pagina sulla base della logica e stato dell'applicazione, oltre ai dati del DOM. Il processo che permette questo viene detto **data-binding**: si tratta di un meccanismo che connette e coordina parti di un template con parti della classe di un componente, sulla base della sintassi di template utilizzata.

```
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
<button type="button" (click)="selectHero(hero)">
  {{hero.name}}
</button>
```

Listing 14: Template HTML che fa utilizzo della sintassi di template.

L'esempio sopra, utilizzando la sintassi di template, illustra i 3 tipi principali di data-binding.

- *Property binding*: permette di passare un valore come proprietà specifica da un componente padre ad un elemento sottostante. L'esempio passa `selectedHero` come valore alla proprietà `hero` del componente `HeroDetail`, utilizzando la sintassi `[hero]="selectedHero"`.
- *Event binding*: permette di richiamare un metodo del componente connesso a seguito di un evento. Il codice di esempio seguendo questo principio chiama il metodo `selectHero(hero)` quando il pulsante verrà cliccato, utilizzando la sintassi `(click)="selectHero(hero)"`.
- *Interpolation*: mostra il valore della proprietà del componente connesso in un punto specifico del template HTML. Nell'esempio, viene mostrato il valore di `hero.name` entro `<button>`, utilizzando la sintassi `{{hero.name}}`.

In Angular, un ulteriore elemento di dinamicità dei template è dato dalle **direttive**: permettono di modificare il DOM in base a determinate istruzioni date in input. Esistono principalmente 2 tipi di direttive: *strutturali e di attributo*. Le direttive strutturali permettono di aggiungere, rimuovere o sostituire elementi di una view seguendo una specifica logica applicativa, mentre quelle di attributo ne modificano l'aspetto e il comportamento.

Il codice presentato qui sotto illustra un esempio di utilizzo delle direttive più utilizzate nel framework. Come sarà possibile notare, una direttiva viene passata ad un elemento HTML come un suo attributo.

```
<li *ngFor="let hero of heroes"></li>
<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
<input type="text" id="hero-name" [(ngModel)]="hero.name">
```

Listing 15: Template HTML che fa utilizzo di direttive.

- `*ngFor`: direttiva strutturale che permette di stampare tanti elementi dove inserito quanti sono i valori nella lista passatagli come valore, come un'iterazione. Nell'esempio sopra, quando la view verrà renderizzata, verranno stampati tanti elementi `` quanti sono gli elementi nella lista `heroes`.
- `*ngIf`: altro caso di direttiva strutturale, che dà la possibilità di mostrare l'elemento HTML in cui è inserito solo se soddisfatta la condizione passatagli come valore. In questo caso, l'elemento `<hero-detail>` verrà mostrato nella view solo se la proprietà `selectedHero` del componente esiste.

- `[(ngModel)]`: si tratta di una direttiva di attributo che introduce un quarto tipo di data-binding, detto *two-way data binding*. Inserito solitamente in un elemento `<input>`, rappresenta una combinazione fra property e event binding: mentre una proprietà del componente viene passata ad un componente del template, la stessa è modificata allo stesso tempo da eventi generati dall'utente, come un suo input.

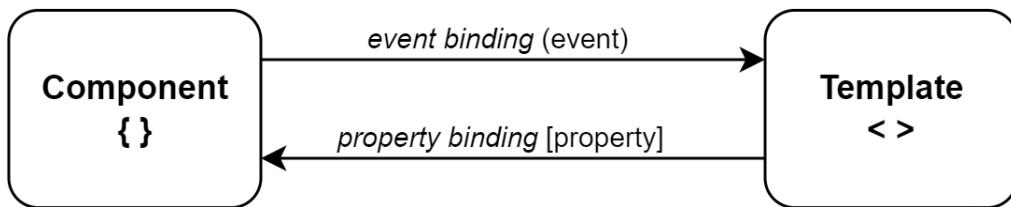


Figura 3.6: Schema di funzionamento del two-way data binding.

Nell'esempio sopra, il campo `name` della proprietà `hero` del componente viene modificata man mano che l'utente digita nell'input testuale.

3.4.2 Angular material

Angular Material è un'ampia libreria di componenti UI per Angular, altamente personalizzabili e responsive. Sono basati sulle specifiche del *Material Design*, una serie di linee guida per la progettazione di elementi grafici: il loro focus riguarda le superfici del mondo reale, incluso il come reagiscono alla luce e proiettano l'ombra, per dare un feedback il più naturale e piacevole possibile all'utente. I componenti di Angular Material coprono una vasta gamma di necessità di una UI: posizionamento ed organizzazione di contenuti (card e grafici), navigazione di una UI (tabs e barre di navigazione) o ancora comunicazione di informazioni (dialog e banner) [8, 9].

Di seguito è mostrato la dichiarazione di uno dei tipi di pulsante in Angular Material. Per poter utilizzare un componente, è necessario importare il suo modulo relativo nel modulo root dell'applicazione: nel seguente caso, è necessario importare il modulo `MatButtonModule`.

```

<button mat-raised-button color="primary">
  Angular Material button
</button>
  
```



Template HTML e view di un bottone in Angular Material.

Come è possibile vedere, il componente è stato introdotto aggiungendo l'attributo `mat-raised-button` al tag HTML `<button>`. Alcuni componenti della libreria vengono inseriti come un componente in Angular, ovvero usando un selector specifico come tag nel template: un layout a griglia, ad esempio, viene dichiarato mediante il tag `<mat-grid-list>`.

Da notare infine l'attributo `color`, responsabile del colore blu del pulsante. Si tratta di uno degli attributi specifici del componente della libreria: ogni componente ha un set attributi a cui è possibile assegnare un valore, al fine di poterlo personalizzare in base alle proprie esigenze.

3.4.3 RxJS

RxJS (Reactive Extension for JavaScript) è una libreria javascript per la programmazione basata su eventi ed asincrona. Combinando i pattern iterable⁵ ed observable⁶ con la programmazione funzionale, permette gestire gli eventi come una sequenza di essi, sempre in modo asincrono [17].

Il concetto principale di RxJS viene detto *Observable*: si tratta di un tipo che implementata l'idea di rappresentare collezioni richiamabile di valori o eventi futuri, provenienti ad esempio da interfaccia utente piuttosto che da API response. Al concetto di Observable seguono altri elementi essenziali nella gestione di eventi asincrona, fra cui:

- *Subscription*: rappresenta l'esecuzione di un Observable, determinando eventualmente le istruzioni da eseguire sugli oggetti della sua collezione.
- *Operators*: funzioni pure che permettono di operare su collezioni con il paradigma della programmazione funzionale. Esempi di Operators sono funzioni come `map()` e `filter()`.
- *Subject*: anche considerato un emettitore di eventi, rappresenta un modo per trasmettere valori o eventi a più observer.

Utilizzare RxJS nel contesto della programmazione ad eventi asincrona porta diversi vantaggi. Utilizzando gli operators (e quindi funzioni pure) si riducono possibilità di errori e migliora il controllo del flusso dati di un Observable, oltre che poter modificare i suoi dati.

```

1 import { fromEvent, throttleTime, map, scan } from 'rxjs';
2
3 fromEvent(document, 'click')
4   .pipe(
5     throttleTime(1000),
6     map((event) => event.clientX),
7     scan((count, clientX) => count + clientX, 0)
8   )
9   .subscribe((count) => console.log(count));

```

Listing 16: Esempio di utilizzo della libreria RxJS.

⁵Pattern dove un oggetto, detto iterator, attraversa una collezione di elementi ed accede a ciascuno di essi. L'obiettivo software è di disaccoppiare l'algoritmo dell' iterator dalla collezione.

⁶Popolare pattern nella programmazione ad eventi dove è presente un oggetto, detto *subject*, che mantiene una lista oggetti detti *Observer*. Quando il subject cambia stato, notificherà tutti gli observer mediante un metodo.

L'esempio sopra somma il valore della coordinata X del mouse ogni volta che avviene un click. La funzione `fromEvent()` si occupa della creazione dell'Observable dovuto all'evento; segue la chiamata del suo metodo `pipe()`, che permette di applicare sequenzialmente una serie di operators per trasformare e gestire i valori del flusso. Infine, grazie a `subscribe()`, i valori dell' Observable vengono utilizzati per eseguire determinate istruzioni, come la loro stampa a console di log.

Capitolo 4

Progettazione

L’obiettivo dell’elaborato è quello di implementare ed integrare in FirstLife elementi e strategie di gamification. In particolare, questi ultimi devono essere personalizzabili per ciascun progetto branch esistente in modo individuale, per una maggior flessibilità nell’utilizzo. Sulla base di questo, il sistema deve permettere principalmente 2 attività.

- Gli utenti amministratori di una verticalizzazione devono poter creare e gestire i propri elementi di gamification e monitorarne l’andamento sulla piattaforma. Gli elementi considerati in questo contesto sono i seguenti.
 - Dei *criteri* secondo cui un utente, se in certo istante svolge una determinata azione, gli vengano assegnate determinate ricompense, come dei punti. Questo potrebbe anche contemplare un’area geografica specifica, piuttosto che imporre un limite di volte per cui si possa essere ricompensati per la stessa attività.
 - Dei *badge*, ovvero ricompense simboliche da assegnare al raggiungimento un certo obiettivo: aver aggiunto un ammontare di foto piuttosto che aver creato un numero di iniziative entro un territorio, sono solo alcuni esempi. Questa componente potrebbe essere modellata in diversi modi, in base a ciò che vogliamo rappresentare: un singolo badge per obiettivi evocativi, ovvero per azioni che ad esempio sono svolte una sola volta (registrazione nell’app); una serie di badge assegnati progressivamente all’attività utente (aggiunta di sempre più video); badge per obiettivi raggiunti grazie alla collaborazione di più persone.
 - Un *sistema di livellamento*, per dare la sensazione di progresso a coloro che utilizzano la piattaforma grazie alle loro azioni e contributi.
 - *Classifiche* generate su un criterio specifico (come su chi ha aggiunto più foto), ed utilizzate per implementare *competizioni* in FirstLife. Al termine di una competizione, all’utente viene assegnato un trofeo, paragonabile dal suo punto di vista sempre ad una reward simbolica.

Per il monitoraggio dei componenti sopra citati, si intende invece la cronologia degli assegnamenti di ricompense, piuttosto che la loro distribuzione in una zona geografica con il passare del tempo. In questo modo, è possibile visualizzare l’andamento di tali elementi e agire di conseguenza, come modificarli, disattivarli o ancora crearne di nuovi.

- Per quanto riguarda tutti gli utenti, offrire un meccanismo che si occupi dell'assegnamento delle ricompense a seguito di un determinata azione, se esistenti. Questo in aggiunta ad un'interfaccia grafica dove poter visualizzare il proprio livello, i badge raccolti o ancora la propria posizione in competizioni attive.

Il lavoro per realizzare il modulo di gamification e soddisfare i requisiti richiesti, è stato suddiviso in 3 step:

1. analisi e studio degli elementi e dei meccanismi di gamification necessari;
2. implementazione e sviluppo della logica applicativa di backend secondo il passo precedente;
3. realizzazione delle interfacce grafiche che permettano ad utenti ed amministratori di svolgere tutte le possibili azioni.

Il risultato previsto dall'elaborato, in relazione alla piattaforma FirstLife, è mostrato qui sotto.

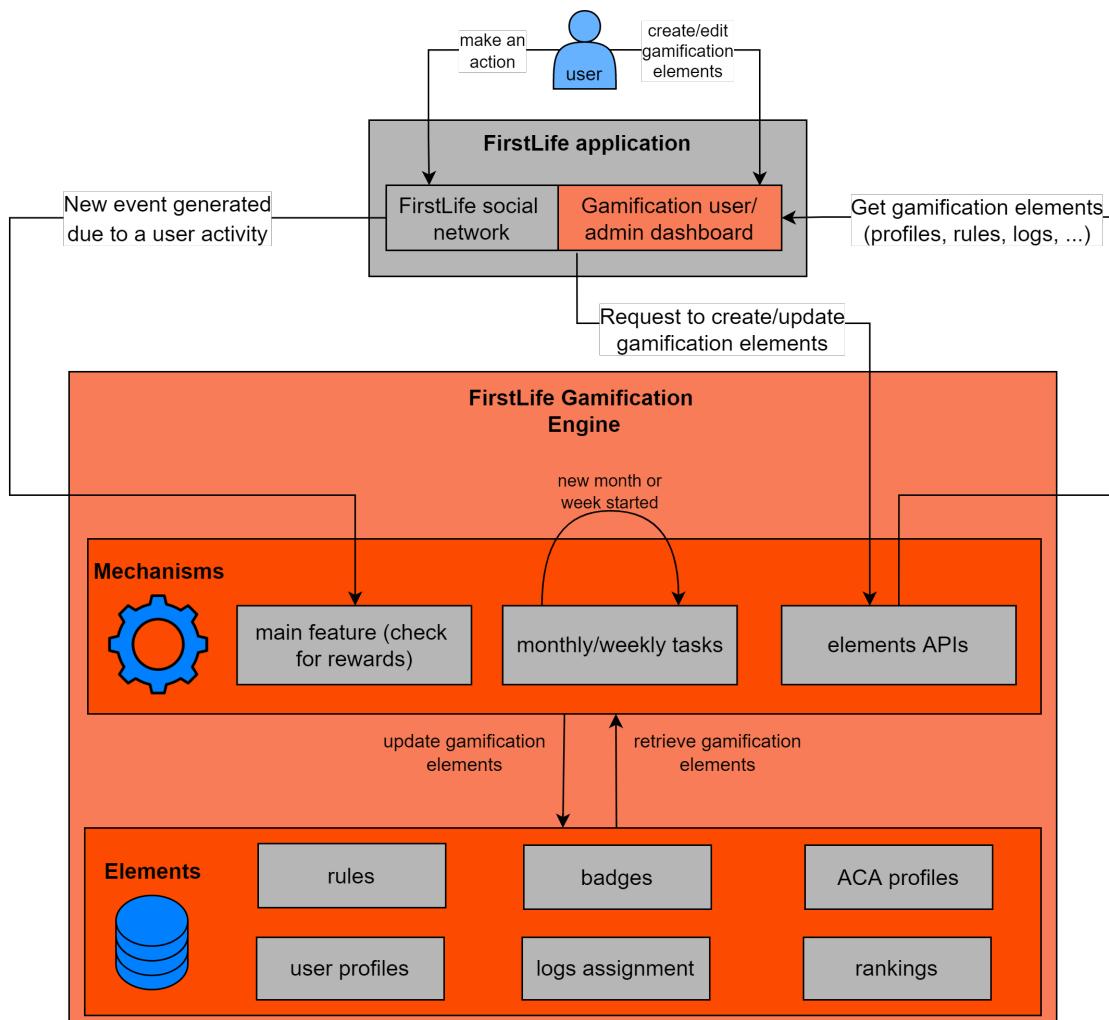


Figura 4.1: Struttura e principali interazioni con il modulo di gamification, integrato in FirstLife. I componenti nei riquadri arancioni sono quelli implementati nel progetto di tesi.

Questo capitolo si occupa della prima fase del progetto: viene svolta l’analisi preliminare per realizzare il gamification engine, il gestore degli aspetti di gamification di FirstLife. Come sintetizzato in Figura 4.1, lo studio è suddiviso in *elementi* del modulo e i relativi *meccanismi* che offrono le funzionalità necessarie.

4.1 Approccio iniziale

Inizialmente, per comprendere come modellare elementi e meccanismi del modulo, nello studio iniziale sono state analizzate le specifiche del *Gamification Layer del progetto CO3 (Co-create, Co-produce, Co-manage)*. Si tratta di un progetto finanziato dall’Unione Europea e che fa utilizzo di FirstLife come mappa interattiva: il suo obiettivo consiste nel valutare rischi e benefici di tecnologie innovative in ambiente cittadino. Fra quelle utilizzate sono presenti:

- *blockchain*, per gestione di artefatti finanziari;
- *realtà aumentata (AR)*, come approccio alternativo nell’interazione con oggetti reali;
- *strumenti di democrazia interattiva*, per prendere decisioni a livello locale;
- *tecniche di gamification* organizzate appunto nel *Gamification Layer*, per migliorare l’engagement con i cittadini e supportare determinate azioni da loro svolte [4].

Pertanto, lo studio dei componenti di gamification di CO3 ha portato ad un loro adattamento e modellazione nell’ambito di FirstLife. Questo non solo considerando la struttura della piattaforma, ma soprattutto che elementi e meccanismi siano adattabili ad ogni verticalizzazione che ne farà utilizzo.

4.2 Elementi del modulo

4.2.1 Elementi introduttivi

Prima di descrivere le nuove entità introdotte grazie al modulo di gamification, verranno presentati alcuni concetti introduttivi a fine di comprensione. Nelle spiegazioni, ci si riferirà diverse volte ad elementi del modello di entità riguardo FirstLife.

ACA

Le *ACA* (*Augmented Commoning Area*) sono una tipologia speciale di entità che rappresentano un’area geografica delimitata in quella di un progetto esterno che fa uso di FirstLife (Figura 4.2).

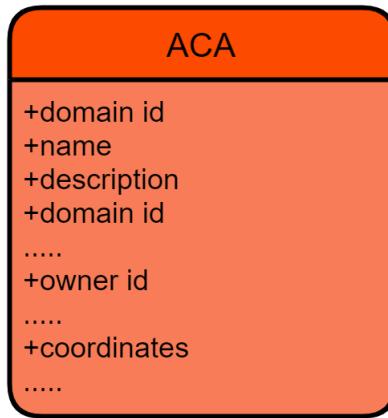


Figura 4.2: Proprietà principali di un entità ACA.

L'attributo `domain id` rappresenta l'id numerico della verticalizzazione di appartenenza dell'ACA, creata dall'utente indicato da `owner id` entro specifiche coordinate descritte da `coordinates`. Alcuni attributi sono stati omessi, poiché fuorvianti dalla trattazione della tesi.

Quando un componente di gamification farà riferimento ad essa, verrà associato l'URL REST API relativo per ottenere le sue informazioni (avrà il seguente formato: `<FirstLife API>/<id dell' ACA>`).

Nel modulo verrà considerata inoltre un' area particolare definita “ACA pilot”. Non rappresenta un ACA realmente esistente (quindi, non possiede un URL da cui ottenere le sue caratteristiche) e sarà identificata da un nome simbolico. La sua utilità è quella di rappresentare la zona di un intero dominio: infatti ne è presente una per verticalizzazione FirstLife.

Come vedremo in seguito, un ACA pilot può essere riferita da un componente di gamification, come nel caso dei badge cooperativi globali.

Utenti

Nel modulo di gamification, gli utenti (denominati in questo contesto anche *actor* o *attori*) sono identificati da un id nelle entità in cui compaiono (il loro profilo, all'interno di un log di assegnamento di ricompense o ancora nelle classifiche).

A partire da un'attività svolta in FirstLife, possiamo distinguere 2 ruoli che un utente può assumere.

- **actor:** l'utente protagonista che ha svolto l'azione.
- **reference owner:** se nell'attività l'attore interagisce con un'entità, le info inerenti a quest'ultima vengono registrate come riferimento nel log dell'attività stesso (o evento di attività, struttura nella prossima sezione). Un utente assume il ruolo di *reference owner* quando e' proprietario di un riferimento a cui un'azione è collegata.

La distinzione di questi 2 ruoli è importante, poiché il gamification engine può prevedere ricompense sia per l'autore dell'azione (aggiungere un commento sotto un post), che per il proprietario di un' entità coinvolta nell'azione (il post sotto cui è stato lasciato il commento). A livello di Gamification, la decisione è stata presa per mantenere attivo l'incentivo anche verso l'utenza più attiva della piattaforma, ovvero quella creatrice di contenuti.

Eventi di attività

Un evento di attività può rappresentare una qualsiasi azione che un attore possa effettuare, dall'aggiunta di un commento sotto una news alla creazione di un'iniziativa: in FirstLife, si tratta del log di un'attività svolta. A differenza degli Events descritti precedentemente, essi non sono entità.

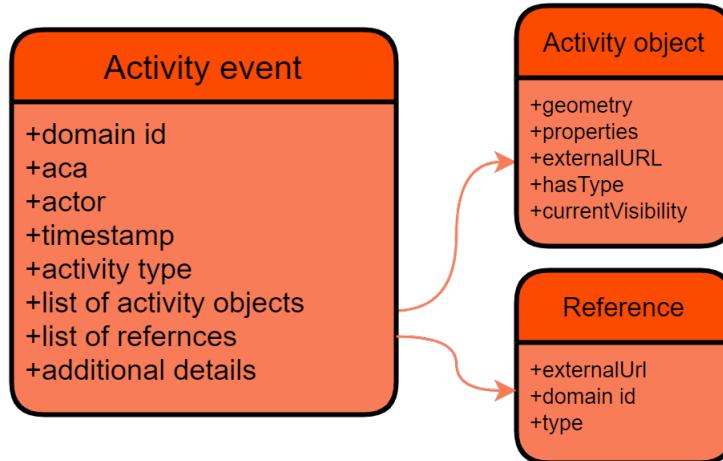


Figura 4.3: Proprietà di un evento legato all'azione di un utente in FirstLife.

Fra gli attributi elencati in Figura 4.3, i primi sono dedicati al luogo dove è stata svolta un'azione: si tratta di `domain id`, sempre indicante la verticalizzazione di riferimento, eventualmente dettagliata da `aca`, il quale precisa una zona geografica. Seguono le informazioni inerenti chi ha effettuato l'attività (`actor`), il timestamp di quando (`timestamp`) e la sua tipologia (`activity type`).

Occorre inoltre prestare attenzione alle liste di `activity objects` e `references`:

- **activity objects** contiene informazioni aggiuntive sulle entità create dall'azione dell'utente. Sono comprese il suo tipo (indicato da `hasType`), piuttosto che caratteristiche aggiuntive per quel tipo di entità (indicato da `properties`; l'entità può ad esempio essere di tipo “Commento” e possedere caratteristiche come il testo relativo).
- Le **references** indicano altre entità già esistenti, con cui il protagonista dell'azione ha interagito. In questo contesto, gli utenti proprietari di quelle entità rappresentano i reference owner. Solitamente, si tratta di entità di prim'ordine.

Gli eventi di attività rappresentano il punto di partenza del meccanismo di Gamification: a partire da essi, in base alle loro proprietà, vengono distribuite nuove reward ai giocatori (Figura 4.4).

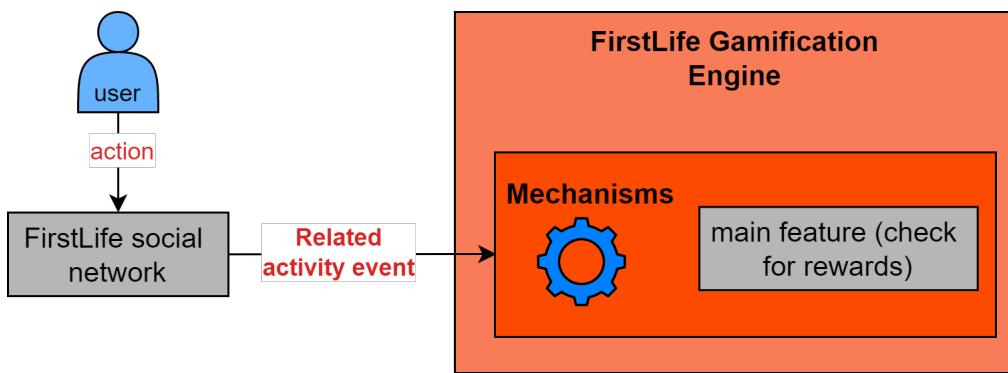


Figura 4.4: Posizione dell'evento di attività nello schema del progetto.

L'evento di attività è generato da FirstLife nel momento in cui un utente effettua un'azione sulla piattaforma. Viene ricevuto dal gamification engine il quale, grazie al suo meccanismo principale, verifica se dei criteri sono soddisfatti da tale attività per assegnare ricompense. Il meccanismo principale sarà approfondito nella sezione successiva, inerente i meccanismi del modulo.

Variabili

Le variabili (o punteggi) sono unità contabili, per tenere traccia di attività e stabilire un sistema di score sulla piattaforma. Si tratta di quelle ricompense accumulate dagli utenti svolgendo determinate azioni.

Alla creazione degli elementi di gamification, diverse variabili vengono "inventate" (ovvero, viene assegnato loro un nome), per riconoscere quelle che sono azioni significative in FirstLife in quella determinata strategia. In base al nome assegnato, è possibile distinguere 2 categorie.

- *Variabili che fanno riferimento ad un'attività che si può effettuare sulla piattaforma.* Un esempio in questa categoria può essere un punteggio chiamato "photos_added": verrà incrementato in base al numero di foto che sono state aggiunte.
- *Variabili che hanno un certo significato simbolico, associando valore all'attività utente.* A questa categoria appartengono tutti quei punteggi che non tengono rigorosamente traccia di un numero di contributi aggiuntivi, come il numero di video o di iniziative. Gli esempi più immediati sono punteggi che possono essere chiamati "points", piuttosto che "kudos", o ancora "credits": in tutti i casi elencati, sono variabili con l'obiettivo di far percepire del valore all'utente. Quest'ultimo potrebbe guadagnare un certo numero di "points" aggiungendo una qualsiasi contribuzione.

4.2.2 Regole

Una regola rappresenta un insieme di criteri e ricompense (per ricompense, in questo caso ci si riferisce ai punteggi). Quando un utente effettua un'azione per cui una regola è applicabile (ovvero, l'evento generato soddisfa i criteri richiesti), guadagnerà le ricompense specificate.

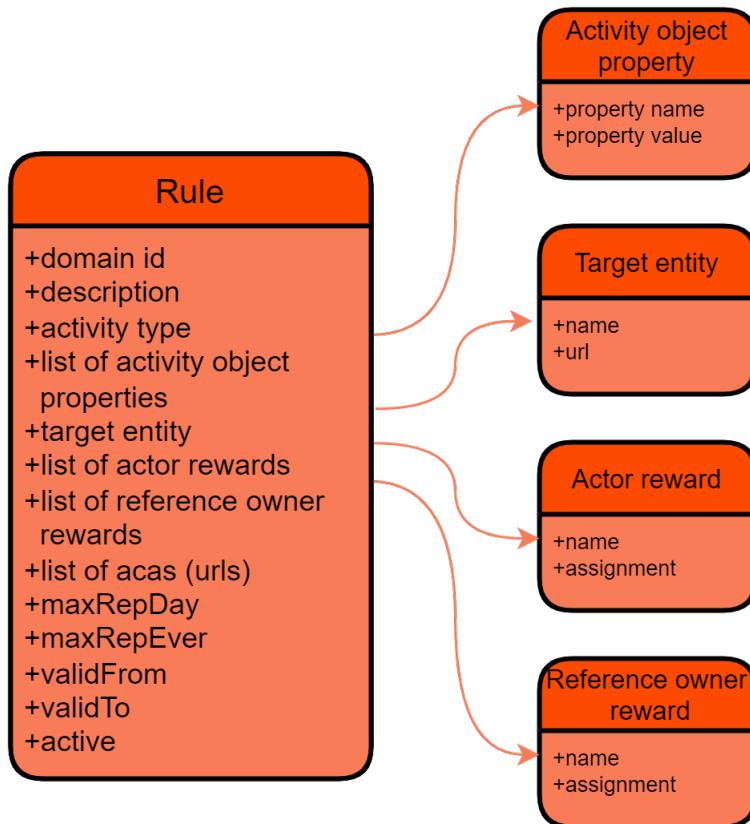


Figura 4.5: Proprietà di una regola di gamification.

Come mostrato nella Figura 4.5, una regola è rappresentata da un insieme ben definito di informazioni.

- Innanzitutto, **domain id** e **activity type** rappresentano i campi principali, più generali e necessari di una regola. Indicano rispettivamente il dominio di FirstLife ove valida e per quale tipo di attività deve essere applicata.
- Può essere associata una breve descrizione (**description**), che idealmente riassume i criteri per cui una regola viene appunto applicata. Risulta utile nel suggerire agli utenti l’azione da svolgere per ottenere le relative ricompense, a livello di UI.
- Per aggiungere una maggior granularità, è possibile definire una **lista di activity object properties**: si tratta delle proprietà che devono possedere le entità create da un’azione. Ad esempio, può essere richiesto come tipo di entità creata una foto, e che questa debba possedere il tag di un certo argomento.
- Un altro criterio di applicazione di una regola può essere per azioni di utenti svolte interagendo con un’entità, come un post sotto cui viene lasciato un commento. Questo viene indicato dalla **target entity**, associando il suo nome e il suo url REST API a fine di identificazione.

Nel meccanismo di gamification, una regola con entità target è applicabile ad un evento se quest’ultimo contiene l’url del target tra i suoi riferimenti. In altre parole, se un’attività consiste nell’aggiungere una foto al “punto di interesse 1”,

quest’ultimo rappresenta una sua entità di riferimento; se una regola ha come criterio di **entità target** “punto di interesse 1”, allora è soddisfatto dall’evento in questione.

- L’applicazione può essere limitata solo in determinate ACA, specificate nella **lista di ACA** apposita. Questo soprattutto al fine di incentivare l’attività degli utenti in certe zone geografiche.
- È possibile rendere una regola valida in un certo arco temporale (indicato da **validFrom** e **validTo**), limitare il suo numero di applicazioni quotidiane o in assoluto (indicati da **maxRepDay** e **maxRepEver**) o ancora disabilitarla manualmente (**active** ne indica lo stato).
- Infine, è specificato l’insieme di ricompense. Si tratta di punteggi, suddivise in base al ruolo dell’utente che le riceverà:
 - **Lista di actor reward**: assegnate all’utente protagonista dell’evento, che ha svolto l’azione;
 - **Lista di reference owner reward**: assegnate a quegli utenti proprietari di entità referenziate dall’evento compatibile.

Tutti i punteggi hanno un nome e una quantità numerica da assegnare (“5 kudos”).

Combinando i criteri elencati sopra, è possibile creare regole che descrivono azioni molto precise, anche in una determinata area; unite a punteggi ben studiati (che, come vedremo nel prossimo paragrafo, sono necessari per vincere nuovi badge) le regole sono il punto di partenza per raggiungere gli obiettivi posti dalla gamification, dall’on-boarding dell’utente in FirstLife alla partecipazione attiva in sue attività di crowdsourcing.

Di seguito sono illustrate mediante esempi le tipologie più comuni di regole di gamification, sulla base dei criteri specificati¹.

Regole di dominio

Le regole di dominio sono il tipo più generale di una regola in una verticalizzazione. Si concentrano sul tipo di attività dell’utente e sulle proprietà dell’entità creata da quest’ultima.

¹A fine di rappresentazione e miglior espressività, alcuni valori che saranno memorizzati e già descritti (come gli url delle entità) negli esempi saranno omessi e sostituiti da un nome simbolico.

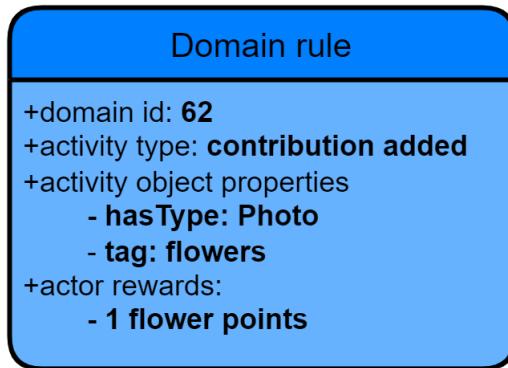


Figura 4.6: Esempio di regola di dominio.

La regola in Figura 4.6 verrà applicata a quegli utenti che, nel dominio 62, hanno aggiunto una foto con tag “flowers”, guadagnando 1 "flower points".

Regole per entità

Nelle regole per entità, l'attenzione si sposta maggiormente sull'interazione di un utente verso una specifica entità FirstLife.

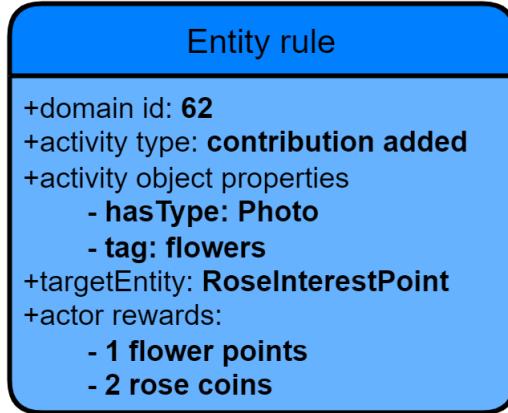


Figura 4.7: Esempio di regola per entità.

La regola sopra verrà applicata a quegli utenti che, nel dominio 62, hanno aggiunto una foto con tag “flowers” al punto di interesse “RoseInterestPoint”, guadagnando 1 "flower points" e 2 "rose coins".

Regole per area

Le regole per area sono focalizzate in azioni svolte in una o più aree specifiche. Le aree, come detto in precedenza, sono rappresentate della lista di ACA.

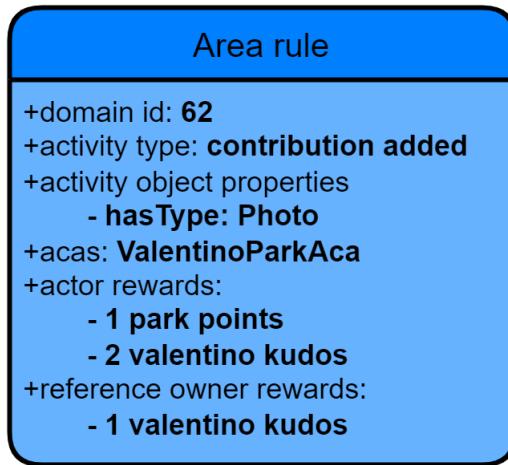


Figura 4.8: Esempio di regola per area.

La ricompense della regola verranno distribuite agli utenti che aggiungono una foto all'interno dell'ACA "ValentinoParkAca", guadagnando in particolare 1 "park points" e 2 "valentino kudos". Inoltre, verrà dato 1 "valentino kudos" anche agli utenti proprietari delle entità con cui ha interagito il protagonista dell'azione, come ad esempio un post.

4.2.3 Badge

I badge sono una ricompensa simbolica e rappresentativa per un player, che li ottiene e colleziona raggiungendo determinati obiettivi in FirstLife. Nel modulo di gamification, i badge sono organizzati all'interno di **container** che contengono informazioni di contesto: un nome che li rappresenti, il dominio dove sono attivi, un eventuale ACA a cui fanno riferimento e ancora il loro tipo (Figura 4.9).

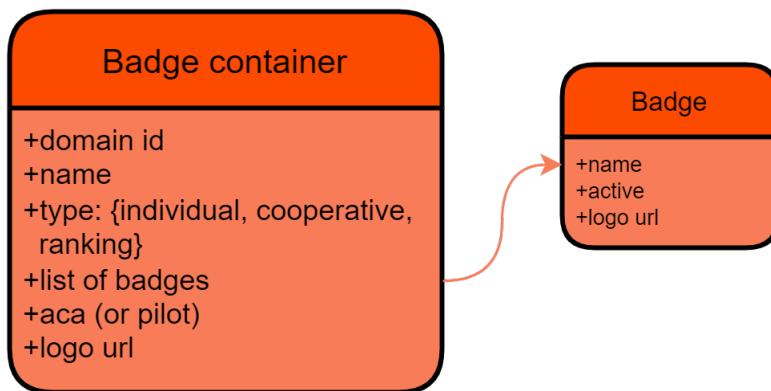


Figura 4.9: Struttura e proprietà generali dei badge, organizzati in container.

Sia un container che i badge all'interno possono avere un logo, rappresentante la parte "aesthetic" dell'elemento di Gamification: viene indicato con l'url dell'immagine adottata grazie all'attributo **logo url**. Inoltre, come per le regole, i singoli badge possono essere disattivati, con **active** che ne indica il loro stato; quando tutti i badge di un container sono disattivati, anche quest'ultimo viene considerato come tale. Infine, il campo **type** indica uno dei 3 tipi di badge modellati all'interno del modulo, come vedremo più avanti nel paragrafo.

In base a come vengono ottenuti, i singoli badge si distinguono in 2 modi.

- **badge step**, vinti soddisfando un serie di precondizioni (Figura 4.10). Possono essere:

- avere una certa quantità di variabili ottenute da regole di gamification, come 10 "points" o 5 "photos added";
- aver già ottenuto determinati badge. Per i badge step all'interno di uno stesso container, ciascuno di essi è vincolato dal possedere quello precedente, in base all'ordine con cui sono stati inseriti in creazione. Ad esempio, se sono stati inseriti in ordine i badge "bronze", "silver" e "gold", allora per ottenere il badge "gold" sarà necessario possedere prima "silver", che a sua volta richiederà di avere "bronze".

Nel definire l'elenco dei badge necessari per ottenerne un altro, è possibile specificarne uno anche appartenente ad un altro container.



Figura 4.10: Struttura di un badge step.

L'attributo `precondition` contiene le liste di precondizioni descritte sopra: la lista di `variable thresholds` per le soglie di punteggi da raggiungere, e la lista di `badges required` per i badge necessari per ottenerlo.

- **badge position**, ottenuti in base ad una posizione specificata raggiunta in una classifica; se la posizione specificata ad esempio è “1”, questo significa che verrà assegnato al giocatore nel primo piazzamento. Come mostrato nella Figura 4.11, tale posizione è indicata dal campo `position in ranking`.



Figura 4.11: Struttura di un badge position.

I badge step, i badge position e i container che li contengono sono strutturati per modellare 3 tipologie di badge, come illustrato sotto.

Badge individuali

I badge individuali sono ottenuti dai giocatori in base alle loro reward e badge accumulati individualmente, in un'intera verticalizzazione o anche entro un' ACA specifica. Si tratta di una lista di badge step inseriti in un container di tipo *individual* (Figura 4.12).



Figura 4.12: Struttura di un container di badge individuali.

Essendo una ricompensa strettamente legata alle azioni del singolo utente, appena vengono aggiunti nuovi badge individuali, il modulo di gamification verificherà se esistono giocatori che possono già ottenerli. Se assegnati, verranno generati i log di assegnamento relativi.

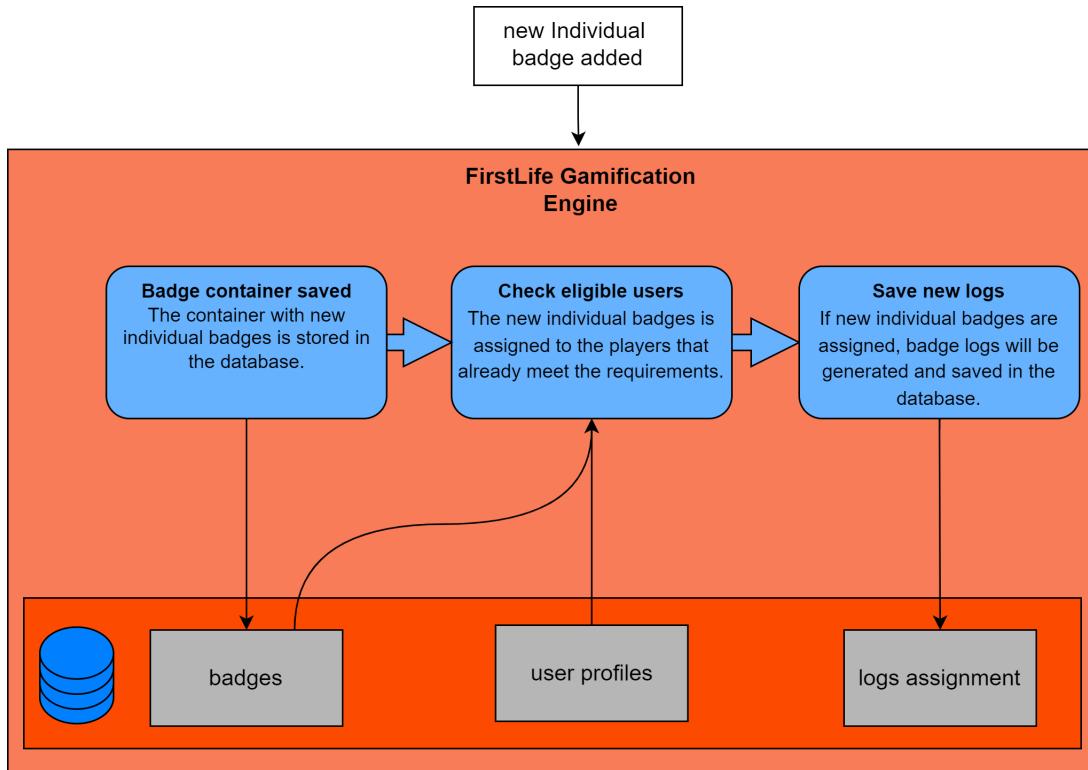


Figura 4.13: Meccanismo per verificare se nuovi badge individuali possano già essere assegnati a giocatori.

Infine, i badge individuali vengono anche utilizzati per il rappresentare il *sistema di livellamento di un giocatore*. I badge dotati di questa caratteristica vengono anche detti **Badge dei livelli**. Sarà la proprietà del container `levelsContainer` ad indicare se i badge al suo interno siano considerati come tali, nell'ordine in cui sono stati inseriti.

Ogni verticalizzazione in FirstLife può avere un proprio sistema di livellamento, tuttavia al loro interno è consentito un solo container di badge individuali con questa caratteristica.

Badge cooperativi

I badge cooperativi vengono ottenuti dai player grazie alle reward accumulate collettivamente, in un'area specifica. Lo scopo di questa tipologia di reward è di rappresentare sfide collaborative periodiche fra più cittadini o stakeholder, aumentando così interazione ed engagement nella piattaforma.

Si tratta di una lista di badge step inseriti in un container di tipo *cooperative* (Figura 4.14).



Figura 4.14: Struttura di un container di badge cooperativi.

A differenza della tipologia precedente, i badge cooperativi possiedono una validità temporale, indicata dalla proprietà **timeframe** del loro container: settimanale, o mensile. Il criterio per cui assegnarli ai giocatori sarà proprio quello di controllare i loro punteggi e badge cooperativi ottenuti nel periodo e nell'ACA specificata, complessivamente. A patto di non essere disabilitati, gli utenti possono ottenere nuovamente uno stesso badge cooperativo una volta al mese/settimana.

Per i badge cooperativi, inoltre, è possibile non indicare nel loro container l'ACA dove devono essere accumulati punteggi, o specificare l'ACA pilot. In questo caso particolare vengono definiti **badge cooperativi globali**: sono ottenuti in base alle ricompense accumulate dai giocatori in un'intera verticalizzazione, sempre entro un intervallo temporale.

Badge competitivi

I badge competitivi (o di ranking) vengono assegnati ai player in base alla loro posizione nella classifica per una determinata variabile, ad esempio per numero di foto caricate o di punti guadagnati. È il tipo di ricompensa che meglio rappresenta le competizioni (anche queste periodiche) nel modulo di gamification, in particolare i trofei ottenuto dopo quest'ultima.

Si tratta di una lista di badge position inseriti in un container di tipo *ranking* (Figura 4.15).



Figura 4.15: Struttura di un container di badge competitivi.

La classifica di riferimento per i badge competitivi è quella generata sulla base della variabile (indicato da **variable**), **timeframe** ed eventualmente ACA specificati dal container.

Il loro assegnamento avviene sempre all'inizio del periodo indicato (settimanale o mensile):

1. viene calcolata la classifica della settimana/mese appena trascorso;
2. ciascun badge position verrà assegnato all'utente nella posizione corrispondente;
3. per ogni badge assegnato in questo modo, viene generato un log di assegnamento.

Come per i badge cooperativi, anche quelli competitivi hanno un periodo di validità, sempre corrispondente a quello del timeframe indicato. Al termine di quest'ultimo, verrà effettuato il calcolo della leaderboard per il nuovo assegnamento.

I passaggi sopra indicati fanno parte dei *task settimanali e mensili* svolti dal modulo di gamification, approfonditi ulteriormente nel corso della trattazione di questo capitolo.

Struttura complessiva

Nello schema che segue (Figura 4.16) viene sintetizzata la struttura complessiva delle 3 tipologie di badge vista finora, gestita dal gamification engine.

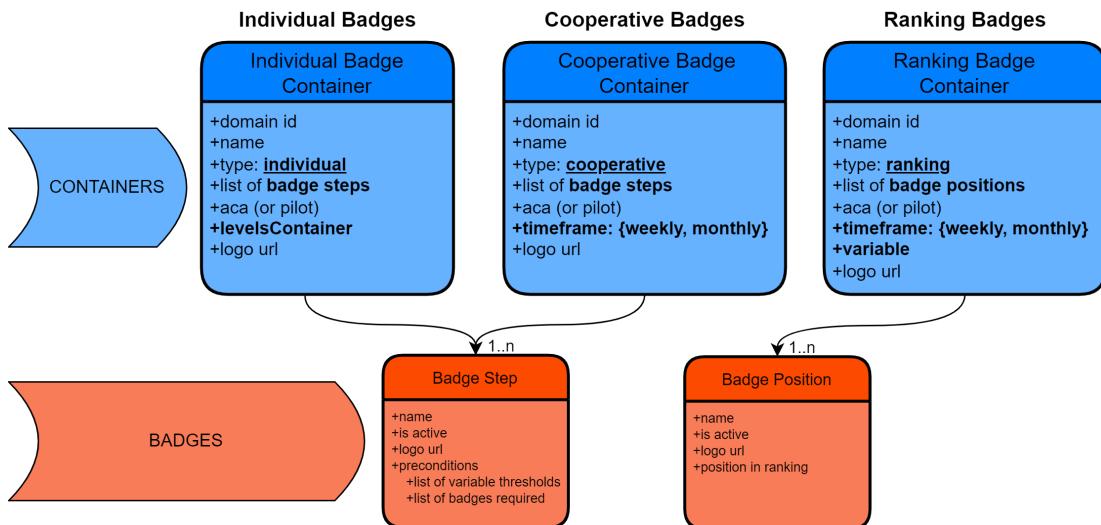


Figura 4.16: Struttura complessiva delle 3 tipologie di badge gestite.

4.2.4 Profili ACA

Un profilo ACA include tutte le reward e i badge cooperativi accumulati dagli utenti in un'area (Figura 4.17).

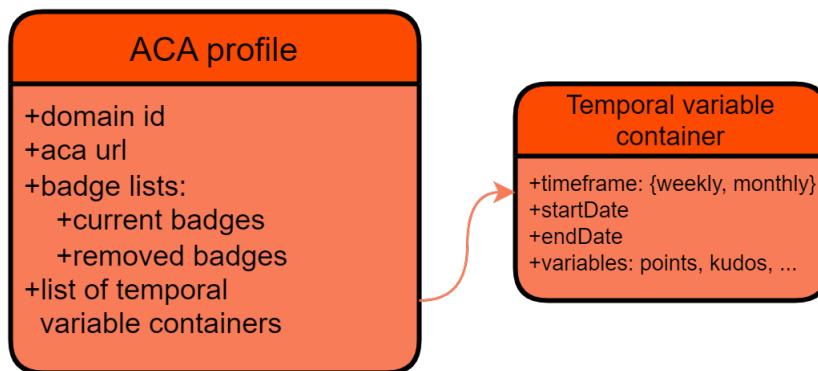


Figura 4.17: Proprietà principali del profilo di un ACA.

Il profilo è riconoscibile principalmente dall'url dell'area e dal domain id del dominio cui appartiene. I badge guadagnati grazie ai punteggi accumulati dagli utenti sono organizzati nel campo **badge lists**, il quale li suddivide a sua volta in:

- lista di **current badges**, ossia i badge cooperativi validi in quel momento;
- lista di **removed badges**, spostati dalla lista precedente per la scadenza del loro timeframe mensile o settimanale. L'idea di badge rimossi in questo modo è quella di rappresentare ricompese simboliche ottenute in un evento passato.

Tale organizzazione, come vedremo nel prossimo paragrafo, sarà la stessa utilizzata nei profili degli stessi giocatori.

Allo stesso modo, i punteggi guadagnati dagli attori sono nei cosiddetti **temporal variable container**: ciascuno contiene le variabili di una determinata settimana o mese. **startDate** ed **endDate** comprendono il periodo della validità del contenitore.

I dati memorizzati nei profili ACA risultano utili nel verificare se un badge cooperativo sia assegnabile a tutti i giocatori che hanno contribuito in quell'area, sempre entro il timeframe indicato.

Il profilo pilot

Il *profilo pilot* rappresenta un profilo ACA speciale, presente uno per dominio di FirstLife.

La sua utilità è di contenere le reward vinte da tutti i giocatori di una verticalizzazione (tutti i punteggi accumulati o ad esempio i badge cooperativi globali). Essendo le variabili sempre organizzate in una lista di **temporal variable container**, il profilo pilot contiene inoltre le informazioni necessarie per monitorare la propria strategia di gamification, settimana per settimana o mese per mese.

4.2.5 Profili utente

I profili utente contengono tutte le reward vinte da ciascun player e le loro info inerenti la gamification: valori delle variabili accumulate da regole applicate, badge collezionati ed il proprio livello (Figura 4.18²). Questo limitatamente in una verticalizzazione FirstLife: un utente può avere un profilo per ognuna di esse.

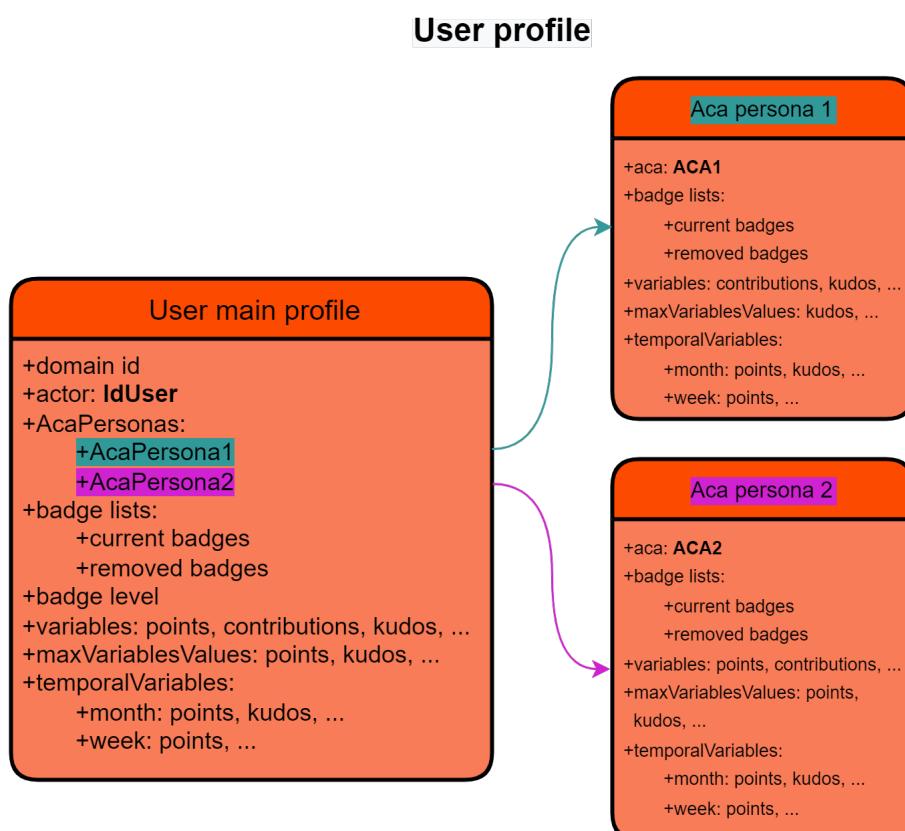


Figura 4.18: Rappresentazione e proprietà principali del profilo di un utente.

²La struttura implementata è leggermente diversa unicamente da un punto di vista tecnico rispetto a quella in schema, per una migliore leggibilità e comprensione. Ad esempio, a livello di DB le **temporalVariables** sono organizzate in **temporal variable containers**, come nei profili ACA.

Come illustrato nella figura sopra, il profilo è strutturato ad albero.

- Il nodo radice contiene le informazioni principali (come l'id dell'utente in **actor**) e quelle inerenti la verticalizzazione in generale, al di fuori quindi da aree specifiche: si tratta di punteggi accumulati non in ACA, badge individuali e competitivi (suddivisi in correnti e rimossi come nei profili ACA) e il livello del player (**badge level**), rappresentato da un badge del container del suo dominio.

In base al loro fine, sono presenti le seguenti proprietà per rappresentare le variabili:

- **variables**: tutte le variabili guadagnate dal giocatore, sin dalla creazione del profilo.
- **maxVariablesValues**: dei punteggi memorizzati nella proprietà precedente, ne vengono riportate quelle che si desidera mettere in risalto, eventualmente anche a livello di interfaccia utente.

Nel modulo i nomi di queste variabili sono stabilite a priori. Ad esempio, fra quelle che tengono conto del numero di foto aggiunte o altri tipi di contributi, si potrebbe essere interessati ad evidenziare il valore delle ricompense “points” e “kudos”.

- **temporalVariables**: viene tenuta traccia delle variabili accumulate nella settimana e mese corrente. In base a questa proprietà viene valutato l'assegnamento di badge con timeframe, come i badge cooperativi o competitivi.
- I nodi figli rappresentano invece le ricompense ottenute in determinate aree (un nodo per ciascuna di esse). Sono anche detti *ACA persona*.

La loro struttura rimane la stessa del nodo radice, escludendo proprietà specifiche del giocatore come il suo livello o l'id del dominio. In essi i badge memorizzati sono quelli con area associata, come ad esempio i badge cooperativi.

4.2.6 Log di assegnamento

I log di assegnamento tengono traccia delle reward ottenute da un giocatore, da punteggi da regole a badge, ed eventualmente in quale ACA sono stati ottenuti (Figura 4.19).

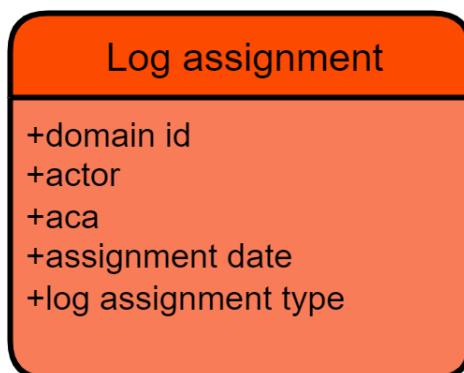


Figura 4.19: Proprietà generali di un log di assegnamento.

In base alla reward di cui se ne registra un assegnamento, i log si dividono in 2 tipologie, indicata da **log assignment type**.

Log di regola

I log di regola vengono generati ogni qualvolta una regola è applicabile all'azione di un giocatore (Figura 4.20).

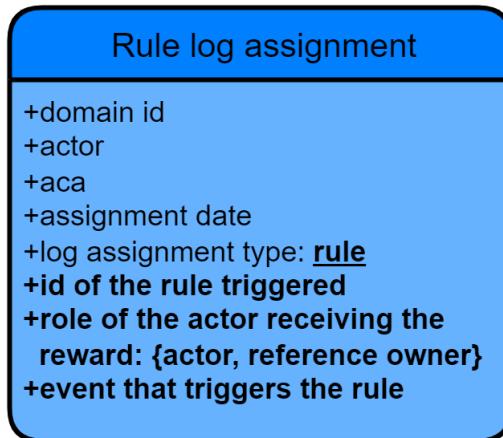


Figura 4.20: Proprietà di un log per l'assegnamento delle reward di una regola.

Oltre all'evento di attività che ha fatto attivare la regola e l'id di quest'ultima, viene registrato anche il ruolo dell'utente nel contesto. In questo modo viene distinto verso chi sono destinate le reward, tra l'attore che ha effettuato l'azione ("actor") e il proprietario di un entità con cui l'attore protagonista ha interagito ("reference owner").

Log di badge

I log di badge vengono generati ad ogni assegnamento di badge ad un giocatore, sia esso individuale, cooperativo o competitivo (Figura 4.21).



Figura 4.21: Proprietà di un log per l'assegnamento di un badge.

4.2.7 Classifiche

Una classifica rappresenta la graduatoria dei giocatori per una variabile da loro accumulata, come il numero di foto aggiunte piuttosto che i punti guadagnati (Figura 4.22).

Come detto in precedenza, sulla base di questo elemento vengono distribuiti i badge competitivi, alla fine di una settimana o mese.

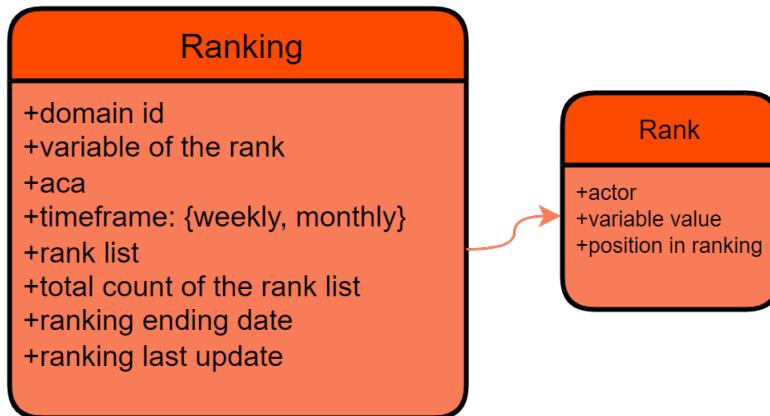


Figura 4.22: Proprietà di una classifica.

Una classifica viene calcolata tenendo conto di:

- variabile (indicata da `variable of the rank`);
- periodo riferito da `timeframe` (settimanale o mensile);
- eventuale ACA dove gli utenti hanno accumulato la variabile specificata (se non presente, è sempre considerata l'intera verticalizzazione indicata da `domain id`).

Gli utenti che partecipano all'interno della classifica sono memorizzati in `rank list`. Per ciascuno di essi viene indicato l'id dell'utente in questione con il suo valore del punteggio e la sua posizione nella leaderboard. Per ragioni di ottimizzazione, a livello di database, una classifica già memorizzata viene aggiornata solo dopo un certo periodo (che è possibile indicare nel gamification engine) dall'ultima richiesta. La data di ultimo aggiornamento è indicata nel campo `ranking last update`.

Ulteriori campi memorizzati in una classifica sono il numero totale di utenti che sono presenti (`total count of the rank list`) e la data di fine della leaderboard stessa (`ranking ending date`). Per "data di fine" in particolare ci si riferisce alla sua validità, nel momento in cui ad esempio la classifica sia stata calcolata per l'assegnazione di badge competitivi. Essa coincide con la data di fine del `timeframe` indicato (quindi, la fine di un mese o di una settimana).

4.3 Meccanismi del modulo

Tutti gli elementi sopra descritti contribuiscono alla logica del modulo di gamification e alle funzionalità offerte: assegnamento di ricompense agli utenti, reset mensile e settimanale dei punteggi e altro ancora.

A rendere disponibili tali funzionalità sono principalmente 3 meccanismi, descritti ed analizzati qui di seguito.

4.3.1 Meccanismo Principale

Il meccanismo principale di gamification, dato in input un evento generato dall'attività di un utente, permette di verificare se per esso possano essere distribuite nuove ricompense, siano esse in forma di punteggi o badge.

Quando il gamification engine riceve un evento per una nuova attività utente, sono svolti i seguenti passi schematizzati in Figura 4.23.

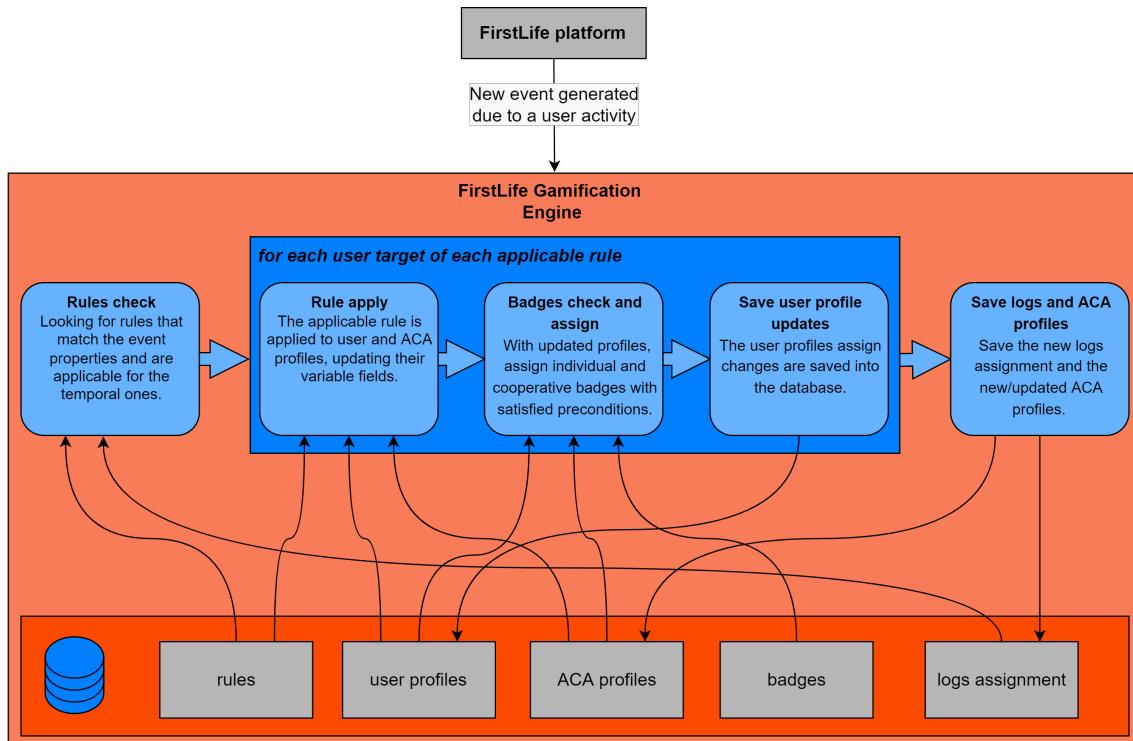


Figura 4.23: Schema del meccanismo principale del modulo di gamification.

Rules check

Il primo passo del meccanismo è proprio quello di controllare se, per l'evento di attività dato in input, esistano delle regole che siano soddisfatte da esso e che possano essere applicabili. In altri termini, vengono effettuati 2 distinti sottopassaggi.

1. Innanzitutto, per ogni regola esistente, viene verificato se i suoi criteri siano soddisfatti dall'attività. Come menzionato in precedenza, questo vuol dire che le proprietà dell'evento devono contenere i valori previsti dai criteri definiti. La tabella mostrata di seguito illustra per campo degli elementi come viene effettuato il matching.

Proprietà della regola da soddisfare	Proprietà dell'evento che deve soddisfarla
domain id	domain id
activity type	activity type
lista di object properties	Almeno una delle entità nella lista di activity object contiene le proprietà
target entity	Almeno un'entità nella lista di

	references che contenga lo stesso url in externalUrl, essendone il riferimento
lista di aca	aca (il valore deve essere nella lista)
validFrom e validTo	timestamp (la data deve essere nell'intervallo specificato dai 2 attributi della regola)

Tabella di matching degli attributi di una regola.

È da ricordare che, ad eccezione di domain id e activity type, una regola non deve obbligatoriamente avere tutte le proprietà definite in tabella. Per essere considerata dal meccanismo, è necessario e sufficiente che l'evento di attività soddisfi quelle presenti.

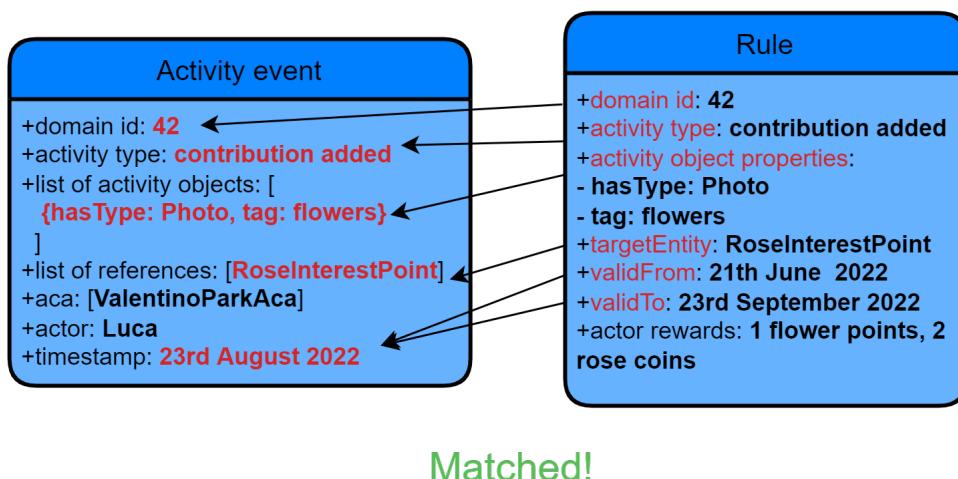


Figura 4.24: Esempio regola soddisfatta da un evento di attività.

Nell'esempio sono evidenziate in rosso le proprietà richieste della regola, e i valori di quelle dell'evento che le soddisfano poiché eguali o, nel caso degli attributi temporali, nell'intervallo richiesto.

2. Delle regole che hanno passato il passo precedente, ora deve essere verificato che siano assegnabili. In altri termini devono essere controllati, fra i campi definiti, di ripetitività (`maxRepEver` e `maxRepDay`).

Il numero di ripetizioni massime giornaliere e in assoluto viene verificato grazie ai log di assegnamento: per ciascuna regola considerata vengono contati il numero di assegnamenti dovuti ad essa, opportunamente solo nella giornata corrente a seconda dei 2 casi.

Le sole regole che superano le verifiche vengono considerate come *applicabili*. Se non vengono individuate regole applicabili all'evento di attività, gli step successivi non verranno eseguiti e l'algoritmo del meccanismo principale termina. Non trovare regole applicabili implica nessun nuovo punteggio da aggiornare per i giocatori e nessun cambiamento al loro stato: non risulta pertanto necessario effettuare controlli per nuovi badge, e nemmeno salvare nuove modifiche.

Prima di passare oltre, sono necessarie alcune ultime precisazioni. Innanzitutto, come anche sintetizzato in Figura 4.23, l'insieme delle regole applicabili ottenute

verrà considerata una alla volta: questo per motivi tecnici e maggior coerenza per la successiva generazione di log di assegnamento. Per ragioni molto simili alle precedenti, essendo che per ciascuna regola possono essere presenti più profili target (l'actor e i reference owner), verranno anche loro presi singolarmente. In altri termini, i 3 passi successivi del meccanismo principale verranno eseguiti un profilo alla volta.

Rule assign

In questo passo, la regola considerata verrà applicata al target utente corrente e i profili ACA (il profilo pilot più quello dell'area dell'attività, se presente). I punteggi specificati in ciascuna regola come ricompense possono essere distribuiti ai profili coinvolti, aggiornandone i loro valori.

Il primi profilo le cui variabili vengono aggiornate sarà quello utente, distinguendo per ogni regola le ricompense per l'actor da quelle del reference owner: in base al fatto che l'azione sia svolta in un ACA o no, l'aggiornamento avviene nel nodo radice del profilo o nell'ACA persona corrispondente. L'incremento delle variabili avviene nei campi rispettivi:

- nel campo **variables**, contenente tutti i punteggi accumulati fin'ora;
- nel campo **maxVariablesValues**, se fra le ricompense è compresa una di quelle messe in risalto;
- nel campo **temporalVariables**, che tiene traccia degli stessi punteggi guadagnati nella settimana e mese corrente. In particolare, come già precisato a proposito della struttura della proprietà, verranno aggiornati i **temporal variable containers** corrispondenti, ossia quelli la cui data di inizio e fine comprendano quella corrente.

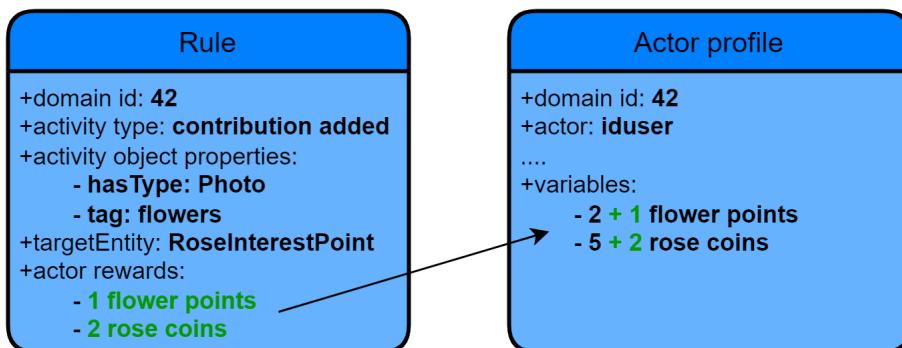


Figura 4.25: Esempio di assegnamento dei punteggi di una regola.

La procedura di aggiornamento variabili avviene nello stesso modo anche nei profili ACA contemplati, nei loro rispettivi **temporal variable containers**.

Il risultato di questo step del meccanismo è una struttura dati contenente le variabili modificate (distinte fra quelle aggiornate nei profili utente e in quelli ACA) e un log di regola per ogni profilo aggiornato, con distinzione di ruolo. Verranno utilizzate nel passaggio successivo.

Badges check and assign

Terminata l'assegnazione dei nuovi punteggi, verrà verificato se al giocatore possono essere assegnati nuovi badge (fra quelli esistenti): si considereranno quelli che contengono, tra le loro precondizioni, le variabili modificate dalla regola corrente. La scelta è motivata dal fatto che la precedente operazione rappresenta l'unico cambiamento all'interno dei profili.

La verifica procede nel seguente ordine:

1. I primi badge ad essere controllati sono i badge individuali. Per ognuno della tipologia, vengono considerati i punteggi totali del giocatore (o solo in un ACA, se specificata nel badge container) e i badge da lui già ottenuti. Se le variabili dell'attore superano le soglie richieste e possiede i badge necessari, allora il badge controllato viene inserito in quelli correnti del nodo corrispondente del profilo, registrandone la data di assegnamento.
2. Successivamente si procede per i badge cooperativi. Verranno considerati solo quelli all'interno di verticalizzazione ed ACA dell'evento di attività: le loro precondizioni sono verificate come descritto sopra, ma con i punteggi cumulativi e badge dei rispettivi profili ACA. Se le precondizioni di un badge cooperativo vengono soddisfatte, quest'ultimo non viene solo assegnato fra quelli correnti del profilo dell'area, ma anche nelle ACA persona di tutti i profili che hanno svolto almeno un'azione in quell'ACA. Oltre alla data di assegnamento è registrata quella di scadenza, corrispondente alla fine del periodo specificato dal badge appena ottenuto.

Lo step termina con la generazione di un log di badge per ogni assegnamento compiuto.

Save user profile updates

Assegnate le ricompense per l'attività svolta in FirstLife, il profilo dell'utente corrente viene salvato nel database con le nuove modifiche.

In seguito, si procede con il prossimo target. Vengono effettuati nuovamente i passaggi di **"Rule assign"**, **"Badges check and assign"** e **"Save user profile updates"**, per ogni target di ogni regola applicabile.

Save logs and ACA profiles

Il meccanismo principale, una volta terminati i target, procede registrando le modifiche dei profili ACA coinvolti, in aggiunta ai log di regola e di badge generati. Mette infine a disposizione una serie di informazioni, inerenti alle operazioni svolte in tutti i suoi passi:

- numero di assegnamenti da regole;
- numero di badge assegnati;
- domain id della verticalizzazione considerata;
- url dell'ACA considerata (se l'azione non è stata svolta in un ACA, viene specificato il nome dell'ACA pilot).

4.3.2 Task settimanali e mensili

Il modulo di gamification, per offrire determinate funzionalità, necessita che alcune operazioni siano svolte periodicamente ed ad un intervallo preciso. Questo è possibile grazie al meccanismo dei task settimanali e mensili (abbreviato anche in *task*): all'inizio di un periodo fra i 2 specificati, il gamification engine attiva il task corrispondente, incaricato di effettuare una certa sequenza di passi.

Gli step e le interazioni con gli elementi sono schematizzati in Figura 4.26.

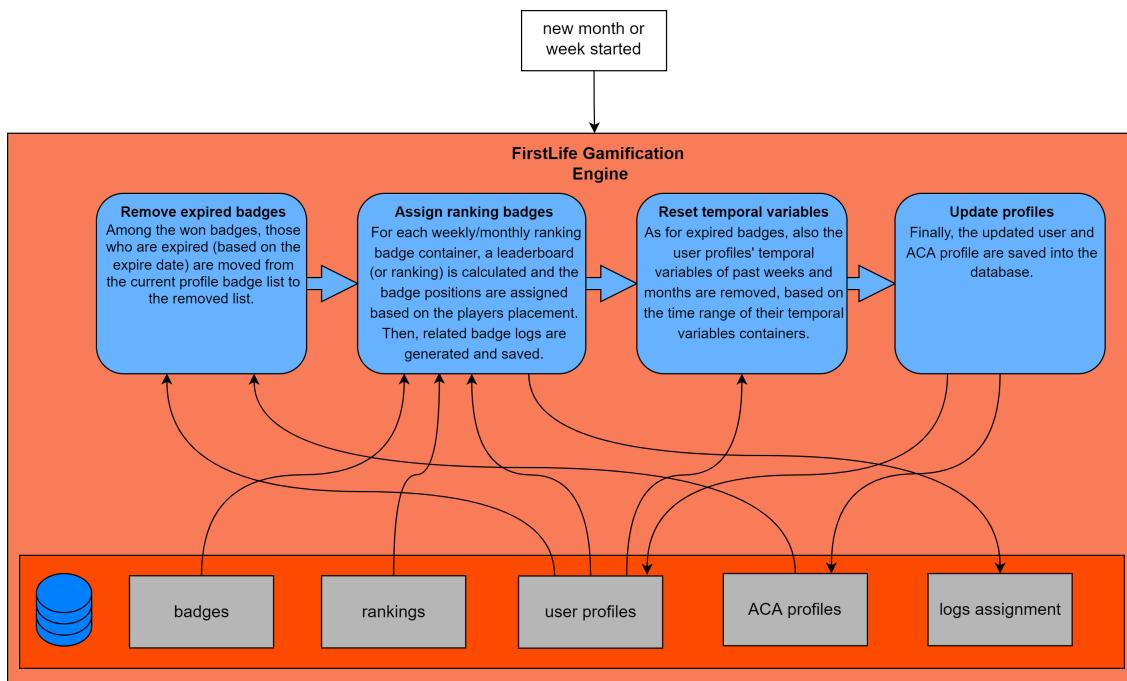


Figura 4.26: Schema dei task settimanali/mensili.

Remove expired badges

La prima azione svolta dal task è quella di rimuovere i badge scaduti dai profili ACA ed utente. Per quest'azione, si intende spostare i badge correnti (la cui data di scadenza sia quindi inferiore a quella corrente), nella lista dei rimossi.

A fine di precisazione, si ricorda che per "badge rimossi" non si intende tolto ad un utente, vengono bensì considerati come vinti in un periodo passato (mesi o settimane fa). Gli elementi considerati in questo step sono quelli dotati di timeframe.

Assign ranking badges

Il passo successivo implementa effettivamente il fattore "competizioni" del modulo di gamification: si sta parlando dell'assegnamento dei badge competitivi.

Vengono considerati inizialmente i contenitori di badge di ranking e, in base al timeframe del task in esecuzione, quelli settimanali o mensili. Ciascun container possiede tutte le informazioni di una competizione effettiva: il suo periodo (timeframe) di validità, domain id della verticalizzazione ove attiva, eventuale ACA dove si è svolta e punteggio su cui basare la gara. Sulla base di tutti questi campi viene calcolata una classifica relativa. L'assegnamento dei badge position nel container, ovvero i trofei della competizione, avviene proprio sulla base del ranking calcolato: è

usato l'attributo `position in ranking` di entrambi i badge e i rank per distribuire nel modo corretto (Figura 4.27).

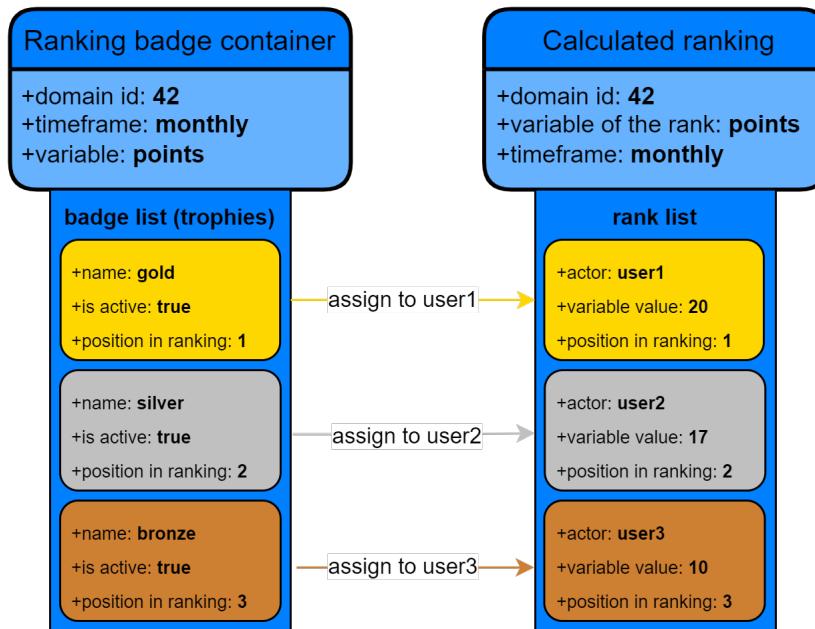


Figura 4.27: Esempio di assegnamento di badge competitivi.

La data di scadenza dei badge competitivi, in questo caso, corrisponde al periodo della successiva competizione nello stesso contesto: se il trofeo è stato ricevuto ad esempio per la gara di agosto, la data di scadenza corrisponde alla fine di settembre.

Come nell'assegnamento del meccanismo principale, per ognuno di essi viene generato un log di badge. Sono registrati nel database una volta terminata la distribuzione di badge, per ogni container.

Reset temporal variables

Successivamente, il meccanismo provvede a resettare le variabili temporali degli utenti, che hanno accumulato nei mesi o settimane precedenti all'attuale. L'operazione, in particolare, effettua nei profili rispettivi un controllo sui `temporal variable container`: quelli che hanno una data di fine (indicato dall'attributo `endDate`, come mostrato a proposito dei profili ACA) passata alla corrente, verranno rimossi.

Il procedimento, tuttavia, esclude i profili ACA. Questo per motivo statistico, come verrà mostrato a proposito delle interfacce utente.

Update profiles

I profili utente ed ACA aggiornati vengono infine memorizzati, terminando il ciclo del meccanismo di un task.

4.3.3 API degli elementi

Le API degli elementi rappresentano il meccanismo per creare, modificare e ottenere informazioni sugli elementi di gamification, tra quelli esistenti. Esse sono in forma di RESTful API, esposte ed utilizzate dalle interfacce grafiche per visualizzazione

e gestione. Questa componente verrà approfondita meglio nel prossimo capitolo, dedicato alla loro implementazione.

Capitolo 5

Gamification Engine

La seconda fase del progetto di tesi riguarda l'implementazione della logica di backend, sulla base dell'analisi nel capitolo precedente. Il database scelto per la persistenza dei dati è MongoDB, di tipo noSQL, con il quale sono state create 6 collezioni:

- `rules` (dati delle regole);
- `badges` (dati dei contenitori dei badge);
- `aca_profiles` (dati dei profili ACA);
- `user_profiles` (dati dei profili utente);
- `logs_assignment` (dati dei log di assegnamento);
- `rankings` (dati delle classifiche calcolate).

L'applicazione server-side, invece, è stata sviluppata con il framework NestJS e scritta in linguaggio Typescript. Sono state principalmente utilizzate, inoltre, le seguenti librerie.

- **@nestjs/mongoose** per l'interazione con il database, oltre che definire la struttura (*schema*) per ogni documento delle collezioni MongoDB.
- **class-validator** per la validazione dei dati nelle richieste HTTP per le API (valori in body o anche parametri di query). In altre parole, il suo principale scopo è la strutturazione dei DTO (Data Transfer Object) necessari con il formato corretto.
- **class-trasformer** utilizzato nelle classi per la serializzazione di dati in input ed output. I casi d'uso più comuni sono il cast del tipo di un attributo, la trasformazione di un dato secondo un certo criterio o ancora la definizione di un valore di default, nel caso un campo non sia definito da input.
- **@nestjs/schedule**, che ha permesso la creazione di cron-jobs per l'esecuzione periodica di istruzioni.
- **@nestjs/swagger** per la documentazione di backend, API e DTO secondo lo standard *OpenAPI*.

Per tutte le librerie citate sopra sono utilizzate le tipiche funzioni decorator di Nest: ciascuno aggiunge i dati necessari per uno scopo ben preciso (esempi possono essere un attributo dello schema di un documento MongoDB, piuttosto che la descrizione di una API per la documentazione Swagger).

Il risultato atteso di questa fase è l'esposizione delle API necessarie per la gestione degli elementi del modulo (per la precisione, le *API degli elementi*), in aggiunta alla logica del meccanismo principale e dei task settimanali e mensili.

5.1 Struttura ed elementi generali del backend

Nella strutturazione del progetto, viene fatto riferimento all'architettura a moduli tipica del framework. La decisione ha sia il fine di massimizzare l'organizzazione dei file di codice, che la scalabilità dell'applicazione stessa.

Qui di seguito viene mostrata la suddivisione generale delle directory. È presente il file `app.module.ts`, contenente le informazioni del *modulo root* di Nest.

```
FirsLife-Gamification-Engine/
  └── src/
    ├── common/
    │   └── ...
    ├── config/
    │   └── ...
    ├── database/
    │   └── ...
    ├── rules/
    │   └── ...
    ├── badges/
    │   └── ...
    ├── aca-profiles/
    │   └── ...
    ├── user-profiles/
    │   └── ...
    ├── logs-assignment/
    │   └── ...
    ├── rankings/
    │   └── ...
    ├── tasks/
    │   └── ...
    └── app.module.ts
      └── ...
```

Nel corso della trattazione, per strutture di directory o in porzioni di codice, alcune parti possono essere contrassegnati con "...". Questo perché o saranno visti in seguito, oppure non risultano fondamentali nella spiegazione. Inoltre, essendo il progetto di considerevoli dimensioni, nelle sezioni a seguire verranno evidenziati solo alcuni aspetti: l'obiettivo è comunque modo garantire la comprensione di ciascun componente, così come la sua importanza in questa fase dell'elaborato di tesi.

In base all'organizzazione illustrata sopra, nel capitolo la logica applicativa è divisa in 3 categorie:

- elementi generali;
- moduli degli elementi;
- moduli dei meccanismi.

Gli *elementi generali* sono rappresentati dalle prime 3 cartelle elencate: `common`, `config` e `database`. Ognuna di esse contiene componenti per il corretto funzionamento dell'app, da funzioni di supporto a parametri di configurazione.

5.1.1 Cartella common

La cartella `common` accoglie logica, strutture dati e funzioni di utilità globali nel resto dell'applicazione. Sono utilizzate diverse tipologie di elementi, come mostrata nell'ulteriore organizzazione della directory.

```
common/
└── dto/
    └── ...
    └── enums/
        └── ...
    └── filters/
        └── ...
    └── guards/
        └── ...
    └── pipes/
        └── ...
    └── schemas/
        └── ...
    └── utilities/
        └── ...
```

dto

Data Transfer Object comuni negli altri moduli, come stessi parametri di query utilizzati in più API.

```
1 import { ApiProperty } from '@nestjs/swagger';
2 import { Type } from 'class-transformer';
3 import { IsNumber, IsOptional } from 'class-validator';
4
5 export class TimeRangeQueryDto {
6     @IsNumber()
7     @Type(() => Number)
8     @IsOptional()
9     @ApiProperty({ required: false })
10    startTime: number;
11
12    @IsNumber()
13    @Type(() => Number)
```

```

14     @IsOptional()
15     @ApiProperty({ required: false })
16     endTime: number;
17 }

```

Listing 17: Uno dei DTO comuni utilizzati nel progetto.

Il DTO sopra è dedicato a parametri di query, in particolare a range temporali. Da notare il decorator della libreria `@nestjs/swagger` `@ApiProperty()`, utilizzato per la documentazione degli attributi di questi tipi di classi. L'opzione `{required: false}` indica un parametro opzionale, in una chiamata API che lo accetti come campo.

enums

Si intendono tutte le enumerazioni impiegate per nomi standard, piuttosto che messaggi di errore predefiniti. Di seguito viene mostrata l'enumerazione che elenca tutti i possibili timeframe.

```

1  export enum Timeframe {
2      WEEK = 'WEEK',
3      MONTH = 'MONTH',
4  }

```

Listing 18: Enumerazione che elenca tutti i timeframe utilizzati.

filters

Contiene gli exception filter personalizzati di Nest, creati per questo contesto. È stato creato un solo exception filter all'interno del progetto: ha il compito di intercettare tutte le possibili eccezioni e formattare una risposta di errore nel modo corretto e facilmente leggibile.

```

1 import {
2     ExceptionFilter, Catch, ArgumentsHost, HttpException,
3 } from '@nestjs/common';
4 import { Response } from 'express';
5 import { ResponseType } from '../eunms';
6
7 @Catch()
8 export class ExceptionsFilter implements ExceptionFilter {
9     catch(exception: HttpException, host: ArgumentsHost) {
10         const ctx = host.switchToHttp();
11         const response = ctx.getResponse<Response>();
12         const status = exception.getStatus() ? exception.getStatus() : 500;
13         const res = exception.getResponse()
14             ? exception.getResponse()
15             : {
16                 error: ResponseType.INTERNAL_SERVER_ERROR,
17                 message: exception.toString(),

```

```

18     };
19
20     response.status(status).json(res);
21 }
22 }
```

Listing 19: Exception filter del Gamification Engine.

Come mostrato nel codice, nel caso in cui non venga riconosciuta l'eccezione (il metodo `exception.getStatus` non esiste nell'oggetto `Response`), verrà restituito un errore di tipo "Interval Server Error". Questo in aggiunta alla descrizione dell'eccezione sconosciuta.

guards

Raggruppa tutte le guards sviluppate. Nell'app sono presenti 5 guards, ciascuna delle quali determina l'esecuzione di un endpoint sulla base di una certa condizione.

- **ContentTypeGuard**: controlla che, all'interno di una richiesta HTTP, il suo body sia di tipo json. Viene utilizzata soprattutto in endpoint per la creazione di elementi di gamification: i campi del nuovo oggetto è rappresentato da un oggetto json. In caso il tipo di contenuto non venisse rispettato, viene lanciata un'eccezione di tipo `UnsupportedMediaType`.
- **FLAuthGuard**: guard per l'autenticazione di un utente, che prova a comunicare con una delle API del gamification engine. L'autenticazione è eseguita mediante *Bearer Token*: in altre parole, dato il token di sicurezza, viene effettuata una chiamata al server di FirstLife per validare il token stesso ed accertarsi che sia stato generato per un utente della piattaforma. In caso la validazione non vada a buon fine, o il token risulti scaduto, viene lanciata un'eccezione di tipo `Unauthorized`.
- **DomainIdGuard**: si assicura sia presente nell'header di una richiesta il campo "DOMAIN_ID", dedicato appunto all'id di una verticalizzazione in FirstLife. Il valore dell'id solitamente deriva dall'utente autenticato grazie alla guard precedente. Nel caso non sia presente "DOMAIN_ID", viene lanciata un'eccezione di tipo `Forbidden`.
- **FLAdminGuard**: a seguito dell'autenticazione, questa guard si occupa di controllare il ruolo dell'utente in API dedicate agli amministratori FirstLife. Esempi di API protette in questo modo sono ad esempio quelle di creazione e modifica di regole piuttosto che badge. Se il ruolo non è quello previsto, viene lanciata un'eccezione di tipo `Forbidden`.
- **GamificationActiveGuard**: utilizzata per disattivare l'intero Gamification Engine manualmente, per motivi ad esempio di manutenzione per correggere bug nell'applicativo. In particolare, la guard effettua un controllo su un parametro booleano del file di configurazione del progetto (verrà illustrato a breve): in base al suo valore, le chiamate a tutti gli endpoint saranno bloccate.

```

1 import {
2     CanActivate, ExecutionContext,
3     Injectable, UnsupportedMediaTypeException,
4 } from '@nestjs/common';
5 import { Observable } from 'rxjs';
6 import { HeaderParams, ResponseMessage, ResponseType } from '../eunms';
7
8 @Injectable()
9 export class ContentTypeGuard implements CanActivate {
10     canActivate(
11         context: ExecutionContext,
12     ): boolean | Promise<boolean> | Observable<boolean> {
13         const request = context.switchToHttp().getRequest();
14         const contentType: string = request.headers[HeaderParams.CONTENT_TYPE];
15
16         if (contentType === 'application/json') return true;
17         throw new UnsupportedMediaTypeException(
18             ResponseType.WRONG_CONTENT_TYPE_ERROR,
19             ResponseMessage.WRONG_CONTENT_TYPE,
20         );
21     }
22 }

```

Listing 20: Codice della guard per controllare che il tipo di contenuto in body sia json, in una richiesta HTTP (`content-type.guard.ts`).

pipes

Contiene tutte le pipes utilizzate globalmente nel progetto, per serializzazioni o validazioni con una logica specifica. È stata realizzata a questo proposito la pipe `ParseObjectIdPipe`, che si occupa di controllare che una stringa in input sia un `ObjectId` valido.

schemas

In `schemas`, sono presenti classi utilizzate in *schema mongoose* realizzate nel progetto. Mongoose, infatti, è in grado di riconoscere anche campi di tipo complesso, come una classe scritta appositamente. Questo purché, nelle classi realizzate a questo scopo, sia presente il decorator `@Prop()` per ogni attributo necessario.

```

1 import // ...
2
3 export class TemporalVariableContainer {
4     @Prop(Timeframe)
5     timeframe: Timeframe;
6
7     @Prop()
8     start: Date;
9

```

```

10  @Prop()
11  end: Date;
12
13  @Prop({ type: Map, of: Number })
14  variables: Map<string, number>;
15
16  constructor() {
17    this.variables = new Map();
18  }
19
20  // ...
21 }

```

Listing 21: Classe per i temporal variable container.

La classe sopra viene riferita come tipo nelle classi schema che ne fanno uso, come quello dedicato ai profili ACA. Da notare che, per tipi maggiormente complessi, `@Prop()` ne richiede le specifiche come parametro (`@Prop({ type: Map, of: Number })`).

utilities

Infine, `utilities` rappresenta una libreria di funzioni. Ne sono state sviluppate soprattutto per gestire casi particolari con 2 oggetti Javascript, `Date` e `Map`.

5.1.2 Cartella config

La cartella `config` contiene la parte di configurazione dell'applicazione Nest. Il compito è diviso in 2 file distinti.

config.ts

Il file `config.ts` definisce i parametri di funzionamento del progetto. Si tratta di attributi di una costante, di cui alcuni raggruppati in base all'aspetto di cui si occupano.

```

1  export const config = () => ({
2    port: parseInt(process.env.PORT) || 42069,
3    is_dev: process.env.IS_DEV === 'true' || false,
4    default_domain_id: parseInt(process.env.DEFAULT_DOMAIN_ID) || 62,
5    check_expiration_acc_token:
6      process.env.CHECK_EXPIRATION_ACC_TOKEN === 'true' || false,
7    fl_auth_api: 'https://secure.firstlife.org/validate',
8
9    http: { timeout: 5000 },
10
11   mongo: { uri: process.env.MONGODB_URI },
12
13   gamification: {
14     active: process.env.GAMIFICATION_ACTIVE === 'true' || true,

```

```

15     verticalization_profile: process.env.VERTICALIZATION_PROFILE || 'pilot',
16     main_variables: process.env.MAIN_VARIABLES.split(' '),
17     ranking_caching_time: 50000,
18   },
19 });

```

Listing 22: File di configurazione del progetto.

Come si può notare dal codice, alcuni campi prendono valori da variabili d'ambiente ("process.env.value_name"). Il file .env che contiene tali contenuti è stato escluso dalla trattazione: l'obiettivo principale in questo contesto è presentare in modo chiaro i parametri di configurazione del Gamification Engine.

I primi attributi elencati si concentrano sulle configurazioni iniziali dell'applicativo. `port` definisce la porta di ascolto del server, mentre `is_dev` se l'app è stata avviata in ambiente di sviluppo o produzione. In base a quest'ultimo valore, verranno utilizzati a fine di testing anche quelli di `default_domain_id` e `check_expiration_acc_token`: rispettivamente, indicano un domain id di default e se sia necessario controllare la scadenza di un token di sicurezza. Invece, `f1_auth_api` indica l'url dell'API FirstLife per validare i token.

Le 2 categorie successive riguardano l'aspetto inerente le chiamate HTTP da parte dell'applicazione (`http`) e l'interazione con il database (`mongo`). Mentre la prima indica il tempo massimo per ricevere una risposta (in millisecondi), nel secondo caso viene definito lo uri del db a cui connettersi.

Infine, l'ultima sezione è relativa a degli aspetti di default della gamification:

- `active`: indica se il Gamification Engine è attivo.
- `verticalization_profile`: il nome dei profili ACA pilot (se non specificato, vengono appunto chiamati "pilot").
- `main_variables`: variabili principali tra quelle accumulate nei profili di un utente (i loro valori sono specificati nel loro attributo `maxVariablesValues`).
- `ranking_caching_time`: il tempo massimo (in millisecondi) di permanenza nel database di una classifica calcolata. Nel caso venga richiesta la stessa classifica dopo lo scadere del tempo, essa viene generata nuovamente.

swagger.config.ts

`swagger.config.ts` sposta invece l'attenzione alla configurazione della pagina di swagger generata. Sono presenti i campi più comuni della documentazione: dalla descrizione dello scopo delle API sviluppate, ai tag con cui sono suddivise.

5.1.3 Cartella database

L'ultimo tra gli elementi generali del backend è la cartella `database`. Contiene al suo interno codice di utilità e classi generali per l'interazione con base di dati.

```

database/
└── collection-name.enum.ts
└── abstract.schema.ts

```

```

└── abstract.repository.ts
    └── collection-name.enum.ts

```

collection-name.enum.ts

Enumerazione contenente i nomi delle collezioni di MongoDB (quelle elencate all'inizio del capitolo): si farà riferimento ad essa per la loro creazione. Quest'ultima operazione è svolta dai moduli degli elementi, che saranno approfonditi successivamente.

abstract.schema.ts

Classe generale estesa per ogni schema esistente nel progetto. Al suo interno contiene l'attributo identificativo per ogni documento, ovvero un *ObjectId*.

```

1 import { Prop, Schema } from '@nestjs/mongoose';
2 import { Types } from 'mongoose';
3
4 @Schema({ versionKey: false })
5 export class Abstract {
6     @Prop({ type: Types.ObjectId, required: true })
7     _id: Types.ObjectId;
8 }

```

Listing 23: Classe Schema Abstract.

abstract.repository.ts

Infine, **abstract.repository.ts** mette a disposizione la classe astratta estesa da tutte le repository necessarie (chiamata **AbstractRepository**), una per collezione gestita. Si tratta, in altri termini, del punto di partenza per realizzare l'omonimo design pattern. Ciascun metodo definito al suo interno rappresenta un'operazione svolta nel database: creazione di un nuovo documento, ottenimento di uno di essi dato un id o ancora la sua cancellazione.

5.2 Moduli degli elementi

Proseguendo con l'analisi del Gamification Engine, la seconda categoria considerata è quella dei *moduli degli elementi*, dove per "modulo" si intende il componente effettivo di NestJS. Ciascuno di essi ha il duplice obiettivo di:

- esporre le API per uno specifico elemento di gamification (l'insieme delle API offerte da tutti i moduli compongono le *API degli elementi*);
- offrire la logica applicativa relativa a quest'ultimo.

Dal punto di vista della struttura dell'applicativo, ciascun modulo è contenuto in uno delle seguenti cartelle, i cui nomi richiamano gli elementi descritti nel capitolo precedente: **rules**, **badges**, **aca-profiles**, **user-profiles**, **logs-assignments** e **rankings**.

Le cartelle appena elencate, a loro volta, condividono la struttura illustrata sotto. In ciascun caso specifico potrebbero contenere file aggiuntivi, in base alla necessità del singolo modulo. Degli esempi possono essere ulteriori pipe per validazioni o classi di appoggio.

```
<folder-name>/
  └── dto/
    └── ...
  └── schemas/
    └── <folder-name>.schema.ts
  └── <folder-name>.module.ts
  └── <folder-name>.controller.ts
  └── <folder-name>.service.ts
  └── <folder-name>.repository.ts
```

Dove `<folder-name>` va sostituito con il nome delle cartelle elencate in questo contesto.

Cartelle `dto` e `schemas`

In modo analogo alle omonime cartelle in `common`:

- `dto` contiene i Data Transfer Objects utilizzati specificatamente per un modulo;
- `schemas` ha al suo interno le classi utilizzate da *Mongoose* come Schema o parte di esso.

Per quanto riguarda i DTOs, essi hanno prevalentemente utilizzo in casi di validazione, soprattutto in parti di una richiesta HTTP. L'esempio mostrato di seguito mostra il DTO utilizzato per la creazione di una regola: definisce in questo caso il formato che deve possedere l'oggetto json nel corpo della richiesta, con i vari attributi dell'elemento considerato. Si ricorda che, nel caso una validazione non vada a buon fine, si ritornerà all'utente un errore di tipo `BadRequest`.

```
1 import //...
2
3 //...
4
5 export class CreateRuleDto {
6   @IsString()
7   @IsOptional()
8   @ApiPropertyOptional()
9   description: string;
10
11  @IsEnum(ActivityType)
12  @IsNotEmpty()
13  @ApiProperty()
14  activityType: ActivityType;
15
16  @ArrayNotEmpty()
17  @IsString({ each: true })
```

```

18  @IsOptional()
19  @ApiPropertyOptional()
20  acas: string[];
21
22  @IsNotEmptyObject()
23  @ValidateNested({ each: true })
24  @Type(() => RewardDto)
25  @ApiProperty({ isArray: true, type: () => RewardDto })
26  rewards: RewardDto;
27
28  @ArrayNotEmpty()
29  @ValidateNested({ each: true })
30  @Type(() => ObjectPropertyDto)
31  @IsOptional()
32  @ApiPropertyOptional()
33  activityObjectProperties: ObjectPropertyDto[];
34
35  @IsNotEmptyObject()
36  @ValidateNested()
37  // @ ...
38  targetEntity: TargetEntityDto;
39
40  // @ ...
41  maxRepDay: number;
42  // @ ...
43  maxRepEver: number;
44  // @ ...
45  validFrom: Date;
46  // @ ...
47  validTo: Date;
48 }

```

Listing 24: DTO per la creazione di una regola.

Ogni attributo dell'oggetto può aver specificato 3 tipi di decorator.

- Decorator di validazione dalla libreria *class-validator*. Da notare nell'esempio sopra casi come `@ValidateNested()`: si tratta di un decorator che permette di validare anche oggetti annidati, come per `ObjectPropertyDto` piuttosto che `TargetEntityDto`.
- Decorator per trasformazione o cast ad altri tipi di dato, provenienti dalla libreria *class-transformer*.
- Decorator per documentazione dei parametri di API, grazie alla libreria `@nestjs/swagger`.

Per le classi nella cartella `schemas`, invece, ne esiste almeno una per la struttura dei documenti in una collezione specifica; tale classe è definita nel file `<folder-name>.schema.ts`. Il codice qui sotto rappresenta la classe Schema per

i documenti della collezione MongoDB `rules`: in altre parole, la struttura utilizzata per memorizzare ogni regola di gamification.

```

1 import //...
2
3 @Schema({ versionKey: false, collection: CollectionsName.RULES })
4 export class Rule extends DomainIdObject {
5   @Prop()
6   description: string;
7
8   @Prop({ required: true })
9   activityType: ActivityType;
10
11  @Prop({ default: undefined })
12  acas: string[];
13
14  @Prop({ type: Reward, required: true })
15  rewards: Reward;
16
17  @Prop({ default: undefined })
18  activityObjectProperties: ObjectProperty[];
19
20  @Prop({ type: TargetEntity, default: undefined })
21  targetEntity: TargetEntity;
22
23  @Prop()
24  maxRepDay: number;
25  @Prop()
26  maxRepEver: number;
27  @Prop()
28  validFrom: Date;
29  @Prop()
30  validTo: Date;
31  @Prop({ default: true })
32  active: boolean;
33 }

```

Listing 25: Classe Schema per la collezione di regole.

Oltre al decorator `@Prop()`, per elencare gli attributi nella struttura ed un suo eventuale valore predefinito, particolare importanza viene assunta da `@Schema()`. `@Schema()` indica che la classe è effettivamente una struttura per documenti della base di dati, indicando fra i parametri anche il nome della collezione a cui si fa riferimento (`collection: CollectionsName.RULES`). In ultimo, ogni classe utilizzata per lo scopo descritto estende `DomainIdObject`: si tratta della classe contenente l'attributo `domainId` (per indicare la verticalizzazione FirstLife) ed estende a sua volta la classe `Abstract`, per l'*ObjectId* relativo.

<folder-name>.module.ts

File con le informazioni principali di un singolo modulo: le importazioni/esportazioni da e ad altri moduli, piuttosto che le classi *controller* e *provider* al suo interno. Di seguito è mostrato il codice nel caso del modulo dedicato alle classifiche.

```

1 import //...
2
3 @Module({
4   imports: [
5     MongooseModule.forFeature([{ name: Ranking.name, schema: RankingSchema }]),
6     HttpModule.register({}),
7     UserProfilesModule,
8   ],
9   controllers: [RankingController],
10  providers: [RankingService, RankingsRepository],
11 })
12 export class RankingModule {}
```

Listing 26: Classe modulo per quello dedicato alle classifiche.

Fra le importazioni specificate, tutti i moduli condividono le seguenti.

- **MongooseModule**, per la registrazione della classe Schema per una collezione di MongoDB. L'operazione avviene con il metodo `forFeature()`: viene specificato un nome per la struttura (`name: Ranking.name`) e la classe della struttura stessa (`schema: RankingSchema`).
- **HttpModule**, il quale mediante il suo metodo `register()` permette al modulo di effettuare chiamate HTTP. Un esempio è la chiamata al server di FirstLife per l'autenticazione di un utente.

<folder-name>.controller.ts

Contiene il codice della classe controller. Ciascun metodo all'interno di essa definisce una API: per ciascuno modulo verranno elencate tutte quelle realizzate. Qui di seguito è illustrata la classe dedicata alle API per gestire i badge.

```

1 import //...
2
3 @ApiTags(swaggerConfig.tags.badges)
4 @Controller()
5 export class BadgesController {
6   constructor(private readonly badgesService: BadgesService) { }
7
8   // ...
9
10  @Post('badgeContainers')
11  @ApiOperation({
12    summary: 'Create a new badge container. ' + swaggerConfig.admin_warning_msg,
13  })
```

```

14  @ApiBearerAuth()
15  @UseGuards(FLAuthGuard, FLAdminGuard, DomainIdGuard, ContentTypeGuard)
16  createBadges(
17    @Req() req: Request,
18    @Body(BadgeContainerValidationPipe) newBadges: CreateBadgeDto,
19  ) {
20    const domainId = Number.parseInt(
21      <string>req.headers[HeaderParams.DOMAIN_ID],
22    );
23    return this.badgesService.createBadges(newBadges, domainId);
24  }
25
26  // ...
27 }

```

Listing 27: Classe controller ed uno dei metodi API per i badge.

Nelle classi realizzate, tutti controller di tutti i moduli degli elementi presentano dei punti in comune. Innanzitutto, sempre a fine di documentazione, entrano in gioco le funzioni decorator di `@nestjs/swagger`:

- `@ApiTags()`, per raggruppare le API del controller sotto un certo tag (nell'esempio, `swaggerConfig.tags.badges` rappresenta il tag per le operazioni dedicate ai badge);
- `@ApiOperation()`, il quale descrive ciascuna di esse (si possono specificare anche esempi di valori nei parametri di chiamata);
- `@ApiBearerAuth()`, per indicare che una API è protetta dalla validazione di un token di sicurezza.

Inoltre, nel costruttore di ciascuna classe, è "iniettato" per Dependency Injection una istanza della classe service corrispondente (`BadgesService` nel caso sopra), in modo da poter accedere alla logica di business necessaria per elaborare risposte alle richieste ricevute.

Si ricorda infine l'utilizzo del decorator `@UseGuards()` per l'applicazione delle *guards* necessarie. Sempre nel caso mostrato sopra, `@UseGuards(FLAuthGuard, FLAdminGuard, DomainIdGuard, ContentTypeGuard)` è applicata a livello del singolo metodo `createBadges()`; le *guards* specificate verranno eseguite in sequenza, nell'ordine scritto.

`<folder-name>.service.ts`

File contenente la classe del provider service, responsabile della logica applicativa di un elemento specifico del Gamification Engine. La classe service mostrata sotto, dedicata ai log di assegnamento, evidenzia un aspetto ancora una volta in comune con tutti gli altri service scritti nel progetto: la possibilità di utilizzare la DI per usare metodi di service anche provenienti da altri moduli. `LogsAssignmentService` presenta inoltre un caso molto particolare a questo proposito, ovvero l'utilizzo del-

la funzione di Nest `forwardRef()` per risolvere problemi di *Circular Dependency* dell'applicazione¹.

```

1 import // ...
2
3 @Injectable()
4 export class LogsAssignmentService {
5   constructor(
6     private readonly logsAssignmentRepository: LogsAssignmentRepository,
7     @Inject(forwardRef(() => BadgesService))
8     private readonly badgesService: BadgesService,
9   ) {}
10
11  async save(logs: (LogRule | LogBadge) [], dbSession?: ClientSession) {
12    if (logs.length > 0) {
13      await this.logsAssignmentRepository.insertMany(logs, dbSession);
14    }
15  }
16
17  // ...
18}
```

Listing 28: Classe service ed uno dei metodi per i log di assegnamento.

In ultimo, è necessaria un'osservazione inerente ai metodi in classi di questa tipologia: nella maggior parte dei casi, essi avranno un parametro facoltativo di tipo `ClientSession`. Utilizzato anche in metodi di classi repository, serve per indicare che le operazioni svolte siano in una singola transazione MongoDB. La necessità di questo deriva da operazioni, svolte nel backend, contenenti più scritture in più collezioni diverse da svolgere in modo atomico.

`<folder-name>.repository.ts`

Infine, l'ultimo file comune a tutti i moduli degli elementi è quello contenente la classe dedicata al provider repository: mette a disposizione i metodi per accedere alle *QueryAPI* di MongoDB, al fine di poter interagire con la base di dati. Ogni classe repository estende quella astratta `AbstractRepository`, specificando il tipo di documento per cui svolgere le operazioni nel database. Il codice di seguito ne mostra un esempio, per interagire in questo caso con la collezione dedicata ai profili utente.

```

1 import // ...
2
3 @Injectable()
4 export class UserProfileRepository extends AbstractRepository<UserProfileDocument> {
5   protected readonly logger = new Logger(UserProfileRepository.name);
6
7   constructor(
```

¹Per "Circular Dependency", o appunto "dipendenza circolare", si intende quando 2 o più entità dipendono reciprocamente per funzionare in modo corretto.

```

8     @InjectModel(MainPersona.name)
9     private userProfileModel: Model<UserProfileDocument>,
10    @InjectConnection() protected connection: Connection,
11  ) { super(userProfileModel, connection); }
12 }
```

Listing 29: Classe repository per i profili utente.

Come previsto dall'estensione di `AbstractRepository`, la classe specificata come tipo di documento è `UserProfileDocument`. In ogni repository sono inoltre instanziati 2 elementi per Dependency Injection:

- l'oggetto di tipo `Model` della libreria *Mongoose*, il quale mette a disposizione le QueryAPI richiamate nei metodi della repository;
- un oggetto di tipo `Connection`, responsabile invece di creare sessioni per transazioni in MongoDB.

5.2.1 Rules Module

Il *Rules Module* è il modulo inherente le regole di gamification, contenuto nell'omonima cartella `rules`. Per la gestione di questo elemento, il suo controller mette a disposizione le seguenti API:

- ***Creazione di una nuova regola*** (`POST api/v1/gamification/rules`). Le informazioni sulla nuova istanza vengono passate in formato JSON, nel corpo della richiesta. Queste vengono validate grazie alla classe DTO `CreateRuleDto`, ed in seguito attraverso la pipe apposita `ParseCreateRuleDtoPipe` per verifiche maggiormente complesse (ad esempio, accertarsi che il campo `validFrom` sia una data precedente a quella eventualmente definita in `validTo`).

L'unica informazione a fare eccezione è il *domain id*: verrà considerato quello dell'amministratore che ha richiesto la creazione della regola, con la guard `DomainIdGuard` ad accertarsi che tale valore esista. Il metodo con cui viene ottenuto l'id della verticalizzazione per una richiesta, quindi dall'utente che si è autenticato, è applicato anche a tutte le altre API degli elementi.

- ***Ottenimento di tutte le regole di una verticalizzazione*** (`GET api/v1/gamification/rules`). Le lista di regole ottenuta in questo modo può essere ulteriormente filtrata, mediante 3 parametri di query:
 - `actorReward`: verranno considerate solo quelle regole che, fra le ricompense dedicate al protagonista di un azione, sia presente quella specificata dal parametro;
 - `referenceOwnerReward`: analogo al caso precedente, ma considerando invece i punteggi per i *reference owner*;
 - `aca`: vengono ottenute le sole regole che, nella lista di ACA ove valide, contengono quella specificata.
- ***Ottenimento di una singola regola, dato il suo id*** (`GET api/v1/gamification/rules/<idRule>`).

- *Abilitazione/disabilitazione una regola specifica* (POST api/v1/gamification/rules/<idRule>). Il criterio per cui abilitare o disabilitare la regola viene determinato dall'attributo booleano di query enable.

Fra i metodi messi a disposizione dal suo provider service, oltre che per gestire le API descritte sopra è presente una funzione molto importante per il Gamification Engine: `getApplicableRules()`. Tale metodo, avendo come parametro un evento di attività, è quello utilizzato dal meccanismo principale per avere le regole applicabili.

```

1  async getApplicableRules(event: ActivityEvent, dbSession?: ClientSession) {
2      const now = new Date(event.timestamp);
3      const eventRefsUrls = event.references.map((r) => r.externalUrl);
4
5      const rules: Rule[] = (
6          await this.rulesRepository.aggregate(
7              [
8                  // first general checks
9                  { $match: {
10                      domainId: event.domainId,
11                      activityType: event.activityType,
12                      active: true,
13                      $or: [{ acas: { $in: [event.aca] } }, { acas: { $exists: false } }],
14                  } },
15                  // target entity check
16                  { $match: {
17                      $or: [
18                          { 'targetEntity.url': { $in: eventRefsUrls } },
19                          { targetEntity: { $exists: false } },
20                      ],
21                  } },
22                  // validFrom check
23                  { $match: {
24                      $or: [{ validFrom: { $lt: now } }, { validFrom: { $exists: false } }],
25                  } },
26                  // validTo check
27                  { $match: {
28                      $or: [{ validTo: { $gte: now } }, { validTo: { $exists: false } }],
29                  } },
30              ],
31              dbSession,
32          )
33      ).filter((r) =>
34          this.isRulePropsInEventProps(
35              r.activityObjectProperties,
36              event.activityObjects.map((obj) => obj.properties),
37          ),
38      );
39      // filter applicable rules found to checks if it's assignable
40      return await filterAsync(
```

```

41     rules,
42     async (rule) => await this.logsAssignmentService.canRuleBeAssigned(
43       event.domainId, event.actor, event.aca,
44       now, rule, dbSession,
45     ),
46   );
47 }

```

Listing 30: Funzione per ottenere regole applicabili da un evento di attività.

Dopo aver creato gli oggetti necessari, è il metodo della classe repository `aggregate()` ad eseguire la query necessaria e specificata come parametro, verso la collezione di regole del database. Vengono in seguito controllate le proprietà dell'oggetto creato dall'azione (mediante `isRulePropsInEventProps()`) e, sempre per ogni regola, i loro attributi di ripetitività. Quest'ultima verifica è effettuata chiamando un metodo del service dedicato ai log di assegnamento, `canRuleBeAssigned()`: sarà approfondita nel modulo dedicato.

5.2.2 Badges Module

Il *Badges Module* è quel modulo che si occupa degli elementi dei badge, con un'attenta distinzione fra le varie tipologie: individuali, cooperativi e competitivi. I suoi contenuti si trovano nella cartella `badges`, mentre le API esposte dal controller sono le seguenti:

- *Creazione di un nuovo contenitore di badge, contenente nuovi badge* (`POST api/v1/gamification/badgeContainers`). Le informazioni sulla nuova istanza vengono passate come nella creazione di una regola, ovvero tramite oggetto JSON nel corpo di una richiesta. Vi è tuttavia una differenza fondamentale: in questo caso, in base al tipo di contenitore, è necessario effettuare una validazione differente. Bisogna considerare inoltre che la validazione è esclusiva, rendendo pertanto il procedimento più complesso: in altre parole, creando ad esempio un container di badge individuali, bisogna impedire che venga validato un campo come `timeframe`, presente invece nelle altre tipologie. La soluzione è stata, oltre ad aver creato un DTO per ogni tipo di container, l'introduzione della pipe `BadgeContainerValidationPipe`: si occupa di validazione e serializzazione dell'oggetto in input, utilizzando uno dei 3 DTO sulla base del valore del campo `type` specificato.

Inerente alle 3 tipologie, si cita il fatto che è stato svolto un procedimento simile anche per quanto riguarda gli Schema per Mongoose. Sono state realizzate, ed in seguito specificate in `BadgesModule`, 3 differenti classi a questo scopo.

- *Ottenimento di tutti i contenitori di badge di una verticalizzazione* (`GET api/v1/gamification/badgeContainers`). È possibile filtrare ulteriormente la lista grazie a 2 parametri di query:

- `type`, per ottenere tutti i contenitori di badge di un certo tipo;
- `aca`, filtrando i container per un'area specifica ove è possibile vincere i badge che contengono.

- *Ottenimento di un singolo badge, dato il suo id* (GET api/v1/gamification/badges/<idBadge>).
- *Ottenimento di un singolo contenitore di badge, dato il suo id* (GET api/v1/gamification/badgeContainers/<idBadgeContainers>).
- *Abilitare/disabilitare un badge specifico* (POST api/v1/gamification/badges/<idBadge>). Mediante il parametro booleano enable viene indicata l'operazione di abilitazione o disabilitazione.
- *Abilitare/disabilitare un contenitore di badge specifico* (POST api/v1/gamification/badgeContainers/<idBadgeContainers>). Il funzionamento mediante parametro di query risulta uguale all'API precedente. Quando un contenitore di badge è disabilitato si intende, in particolare, che tutti i badge al suo interno siano nel medesimo stato.
- *Modificare uno specifico contenitore di badge e i badge al suo interno* (PATCH api/v1/gamification/badgeContainers/<idBadgeContainer>). Al fine di effettuare l'operazione, verrà passato l'oggetto JSON dell'elemento nel corpo della richiesta. La modifiche permesse sono le seguenti:
 - cambio di nome del contenitore e/o dei badge al suo interno;
 - sostituzione del logo del contenitore e/o dei badge al suo interno;
 - aggiunta di un nuovo badge, senza però modificare l'ordine degli elementi già esistenti. In altri termini, può essere aggiunto un badge solo in fondo alla lista, per introdurre ad esempio un nuovo livello della ricompensa simbolica (un badge che richieda di aggiungere ancora più foto).

Spostando l'attenzione alla classe service del modulo, sono definiti al suo interno 3 metodi utilizzati da altri componenti dell'app ed importanti per il loro funzionamento. La prima è rappresentata da `getUserLevel()`: è utilizzata dal modulo dei profili utente per calcolare il livello di un certo giocatore, considerando i badge dei livelli della verticalizzazione specificata. Seguono `getApplicableIndividualBadges()` e `getApplicableCooperativeBadges()`, le 2 funzioni alla base dello step "*Badges check and assign*" del meccanismo principale: restituiscono rispettivamente i badge individuali e cooperativi assegnabili in quella situazione, considerando i parametri passati ad esse.

```

1  async getApplicableCooperativeBadges(
2    acaProfile: AcaProfile, timeframe: Timeframe,
3    modifiedVariablesAca: string[] , dbSession?: ClientSession,
4  ) {
5    if (modifiedVariablesAca.length === 0) return [];
6
7    const gottenBadges: Types.ObjectId[] = acaProfile.badges.currentBadges.map(
8      (assignedB) => assignedB.idBadge,
9    );
10   const excludedBadges = acaProfile.badges.currentBadges
11     .filter((assignedB) =>
12       timeframe === Timeframe.MONTH

```

```

13     ? new Date(assigndB.expiredDate).getMonth() !== new Date().getMonth()
14     : timeframe === Timeframe.WEEK
15     ? getWeek(new Date(assigndB.expiredDate), { weekStartsOn: 1 }) !==
16       getWeek(new Date(), { weekStartsOn: 1 })
17     : false,
18   )
19   .map((assigndB) => assigndB.idBadge);
20
21 const varsQuery = this.buildVarsThresholdPreconditions(
22   acaProfile.getTemporalVariables(timeframe, new Date()),
23   modifiedVariablesAca,
24 );
25
26 return await this.badgesRepository.aggregate(
27 [
28   // domainId and container type
29   { $match: {
30     $and: [
31       { domainId: acaProfile.domainId },
32       { type: BadgeContainerType.COOPERATIVE },
33       { $or: [{ aca: { $exists: false } }, { aca: acaProfile.url }] },
34     ],
35   }},
36   // deconstruction based on the badge lists
37   { $unwind: { path: '$badgeList' } },
38   // badge preconditions, vars preconditions and if it's active
39   { $match: {
40     $and: [
41       { 'badgeList.id': { $nin: excludedBadges } },
42       { timeframe: timeframe },
43       { $or: varsQuery },
44       { $or: [
45         { 'badgeList.preconditions.badgeList': { $exists: false } },
46         { 'badgeList.preconditions.badgeList': { $size: 0 } },
47         { 'badgeList.preconditions.badgeList': { $in: gottenBadges } },
48       ] },
49       { 'badgeList.active': true },
50     ],
51   }},
52   ],
53   dbSession,
54 ].map((c) => BadgeStep.fromDb(c.badgeList));
55 }

```

Listing 31: Funzione per ottenere la lista di badge cooperativi assegnabili.

Come visualizzato nel codice sopra, verificare per badge cooperativi che possono essere distribuiti richiede 3 parametri, che definiscono in modo completo la situazione corrente:

- **acaProfile**: il profilo cooperativo considerato. Si tratta del target a cui poi verranno eventualmente assegnati nuovi badge cooperativi. Vengono controllati proprio quelli che possono essere vinti nell'ACA corrispondente (e i badge cooperativi globali), considerando i punteggi accumulati da tutti i giocatori in quell'area.
- **timeframe**: il periodo temporale a cui si è interessati effettuare la verifica. In base al suo valore, viene selezionato il **temporal variable container** corretto del profilo.
- **modifiedVariablesAca**: le variabili che sono state modificate dallo step precedente del meccanismo principale, ovvero l'assegnamento delle ricompense delle regole. Il valore di questo parametro rappresenta il cambiamento di stato (o di gioco) del profilo; risulterebbe futile verificare nuovi badge per valori di punteggi rimasti immutati.

L'operazione effettiva di controllo viene infine eseguita dal metodo **aggregate(query)** della repository corrispondente (**badgesRepository**). Un'ultima osservazione inerente alla variabile **excludedBadges**: posta all'interno della query di ricerca, essa fa in modo che non venga riassegnato uno stesso badge già ottenuto nel mese o settimana corrente.

5.2.3 Aca Profiles Module

Per *Aca Profiles Module*, contenuto in **aca-profiles**, si intende il modulo dedicato ai profili cooperativi e la loro gestione, come creazione e aggiornamento. La API che espone il suo controller permette di **ottenere l'elenco dei profili ACA esistenti** (GET `api/v1/gamification/profile/aca`), opportunamente personalizzato sulla base dei possibili attributi di query:

- **aca**: se questo campo specifica l'url dell'area interessata, verrà restituito il solo profilo relativo;
- **startTime** ed **endTime**: definiscono un intervallo temporale per filtrare i **temporal variable containers** di ciascun profilo ottenuto, potendo così visualizzare i punteggi guadagnati in un determinato periodo.

La classe qui sotto rappresenta il DTO per gli attributi appena descritti.

```

1 import //...
2
3 export class AcaProfileQueryDto extends TimeRangeQueryDto {
4   @IsString()
5   @IsOptional()
6   @ApiPropertyOptional()
7   aca: string;
8 }
```

Listing 32: DTO per i parametri di query.

Dove i campi **startTime** ed **endTime** derivano dall'estensione della classe **TimeRangeQueryDto**.

5.2.4 User Profiles Module

In modo analogo ai moduli precedenti, *User Profiles Module* si occupa dei profili utente e si trova nella cartella `user-profiles`. La singola API esposta, in aggiunta a 3 parametri di query, permette di ***ottenere il profilo dell'utente richiedente*** (GET `api/v1/gamification/profile/user`), con specificità necessaria. I primi 2 parametri sono `actor` e `aca`: indicano, rispettivamente, l'id di un attore di cui si desidera il profilo (andrà quindi a sostituire quello dell'utente autenticato), e l'area a cui si è interessati visualizzare le ricompense.

Infine, mediante l'ultimo campo `viewMode`, la REST API offre 3 modalità di visualizzazione per il profilo di un utente, in termini di struttura del file json in risposta.

- **HIERARCHY:** verrà considerato *il sotto-albero del profilo, a partire dal nodo specificato*. In altri termini:
 - se specificato il solo giocatore il nodo sarà quello radice, restituendo l'intero albero del profilo;
 - se invece viene precisata anche un ACA verrà allora considerato il nodo figlio (se esiste), restituendo il sotto-albero relativo a quest'ultimo.

```

1
2   "acaPersonas": [
3     {
4       "badges": { ...
5       },
6       "aca": "https://api.firstlife.org/v6/fl/Things/614c79a06f9f21fbc3521925",
7       "temporalVariables": { ...
8       },
9       "variables": { ...
10      },
11       "maxVariablesValues": {
12         "points": 5
13       }
14     },
15     { ...
16   },
17   ],
18   "variables": {
19     "comment_added": 5,
20     "points": 10
21   },
22   "maxVariablesValues": {
23     "points": 10
24   },
25   "temporalVariables": {
26     "WEEK": {},
27     "MONTH": {
28       "comment_added": 5,
29       "points": 10
30     }
31   },
32   "badges": {
33     "currentBadges": [
34       {
35         "expiredDate": null,
36         "timeframe": null,
37         "idBadge": "62c98da8ef76d72bb8cd25ad",
38         "assignedDate": "2022-07-02T15:09:46.000Z"
39       }
40     ],
41     "removedBadges": []
42   },
43   "viewMode": "HIERARCHY",
44   "actor": "61e5b514964e9b1d24a74328"
45 }
```

Figura 5.1: Profilo in modalità di visualizzazione HIERARCHY.

Essendo stato specificato il solo id dell’utente (presente in `actor`), l’esempio mostra l’intero albero del profilo. `acaPersonas` contiene gli ACA persona, uno per ciascuna area ove il giocatore ha ottenuto ricompense.

- **SINGLE:** verrà selezionato *un singolo nodo dell’albero del profilo*, ovvero:
 - se specificato il solo utente, verrà mostrato il solo nodo radice;
 - se è presente in aggiunta anche un’ACA specifica, verrà allora restituito il singolo nodo figlio inherente quell’area, se presente.

```

1
2   "aca": "https://api.firstlife.org/v6/fl/Things/614c79a06f9f21fb3521925",
3   "badges": {
4     "currentBadges": [],
5     "removedBadges": []
6   },
7   "variables": {
8     "photos_added": 1,
9     "points": 5,
10    "comment_added": 1
11  },
12  "maxVariablesValues": {
13    "points": 5
14  },
15  "temporalVariables": {
16    "WEEK": {},
17    "MONTH": {
18      "photos_added": 1,
19      "points": 5,
20      "comment_added": 1
21    }
22  },
23  "viewMode": "SINGLE",
24  "actor": "61e5b514964e9b1d24a74328"
25

```

Figura 5.2: Profilo in modalità di visualizzazione SINGLE.

L’esempio illustrato mostra il singolo nodo figlio delle reward vinte dal giocatore in una determinata ACA, specificata nel parametro di query `aca`.

- **AGGREGATE:** mostra in modo aggregato tutte informazioni inerenti il giocatore, tali che *le variabili e i badge vinti siano la somma di quelli ottenuti nel nodo considerato, e nel suo sotto-albero*. In particolare:
 - se specificato il solo utente, allora verrà restituita l’aggregazione di tutte le sue reward, anche quelle in aree specifiche;
 - se viene aggiunta anche la specifica di un’area, il risultato sarà pressoché simile alla visualizzazione in SINGLE mode: verrà mostrato il solo ACA persona relativo.

```

1   "variables": {
2     "comment_added": 6,
3     "points": 15,
4     "photos_added": 1
5   },
6   "variablesMaxValue": {
7     "points": 15
8   },
9   "badges": {
10    "currentBadges": [
11      {
12        "expiredDate": null,
13        "timeframe": null,
14        "idBadge": "62c98da8ef76d72bb8cd25ad",
15        "assignedDate": "2022-07-02T15:09:46.000Z"
16      }
17    ],
18    "removedBadges": []
19  },
20  "temporalVariables": {
21    "WEEK": {},
22    "MONTH": {
23      "comment_added": 6,
24      "points": 15,
25      "photos_added": 1
26    }
27  },
28  "viewMode": "AGGREGATE",
29  "actor": "61e5b514964e9b1d24a74328"
30}
31

```

Figura 5.3: Profilo in modalità di visualizzazione AGGREGATE.

L'esempio mostra la somma di tutte le ricompense ottenute, anche quelle all'interno di aree.

Passando alla parte del service del modulo, uno dei suoi metodi degni di nota è `buildUserProfileResponse()`. Visualizzato qui sotto, il suo compito è proprio quello di costruire la visualizzazione di un profilo utente per essere restituito in risposta, sulla base del valore di `viewMode`.

```

1 buildUserProfileResponse(profile: UserProfile, viewMode: ViewMode) {
2   const res: any = {};
3   if (profile instanceof AcaPersona) res.aca = profile.aca;
4
5   switch (viewMode) {
6     case ViewMode.HIERARCHY:
7       if (profile instanceof MainPersona)
8         res.acaPersonas = MainPersona.acaPersonasNormalized(profile.acaPersonas);
9       this.setProfileMaps(profile, res);
10      res.badges = profile.badges;
11      break;
12
13    case ViewMode.SINGLE:
14      res.badges = profile.badges;
15      this.setProfileMaps(profile, res);
16      break;
17
18    case ViewMode.AGGREGATE:

```

```

19     res.variables = Object.fromEntries(profile.getFullVariables());
20     res.variablesMaxValue = Object.fromEntries(profile.getFullMaxVariables());
21     res.badges = profile.getFullBadges();
22     res.temporalVariables = UserProfile.temporalVariablesNormalized(
23         <Map<string, Map<string, number>>>profile.getFullTemporalVariables(),
24     );
25     break;
26 }
27
28 if (profile instanceof MainPersona) res.level = profile.badgeLevel;
29 return res;
30 }
```

Listing 33: Funzione per costruire la visualizzazione del profilo utente, in base a quella richiesta.

Essenzialmente, in base alla modalità di visualizzazione, la funzione provvedere nel costruire le mappe di valori (inerenti ad esempio ai punteggi ottenuti) per far fronte alla richiesta.

Prima della fase di costruzione (ovvero, prima del costrutto di `switch`), se è relativo ad un ACA persona, verrà specificato l'url dell'area corrispondente (`res.aca = profile.aca`): si tratta del caso in cui l'attributo di query `aca` è stato definito.

In seguito alla costruzione, invece, se non è stata indicata alcun ACA e si considera il nodo radice, all'oggetto JSON di risposta viene aggiunta anche l'informazione sul livello del giocatore. Quest'ultimo valore viene calcolato prima della chiamata a `buildUserProfileResponse()`, mediante il metodo `getUserLevel()` del service di `BadgesModule`.

5.2.5 Logs Assignment Module

Si tratta del modulo dedicato alla gestione dei log di assegnamento, sia per ricompense da regole che di badge: il suo contenuto è presente in `logs-assignment`. Come descritto nel capitolo precedente, le 2 tipologie di elemento possiedono una struttura differente tra di loro, portando ad attuare una strategia analoga a *Badges Module* a livello di database: è stata creata cioè una classe Schema per ogni tipo esistente, in modo da essere riconosciute da *Mongoose*.

Il controller di *Logs Assignment Module* espone ai client 2 API:

- **Ottenimento dei log di assegnamento dell'utente autenticato** (GET `api/v1/gamification/logs`). Come molte API già descritte, sono disponibili diversi parametri di query come criterio di ulteriore filtro.
 - `startTime` e `endTime`, per considerare i log entro un certo intervallo temporale.
 - `aca`: la lista viene filtrata per quei log generati nell'area specificata.
 - `variable`: in relazione ai log di regola, vengono restituiti solo quelli che hanno assegnato la variable indicata.
 - `rules` e `badges`: si tratta di 2 campi booleani. In base al loro loro valore, indicano rispettivamente se includere nella lista i log di regola e quelli

di badge. Come valore predefinito è posto il valore "true", considerando pertanto tale tipologia.

- **resolve**: campo booleano che, se posto a "true", espande i dettagli di ciascun log di assegnamento. In particolare:

- * nel caso dei log di regola vengono mostrati i dettagli della regola applicata, oltre a quelli dell'evento di attività che ha scaturito l'applicazione;
 - * per quanto riguarda i log di badge, vengono aggiunte tutte le informazioni sul badge assegnato al giocatore.

Se invece il valore `resolve` è "false", saranno indicati i soli id degli elementi citati sopra (Figura 5.4). Quest'ultimo rappresenta inoltre il valore predefinito.

```
[{"  
  "_id": "63075c49cbb84e2e12e9903a",  
  "idBadge": "62d6704a1d301a5ddcf25769",  
  "domainId": 98,  
  "actor": "61e5b514964e9b1d24a74328",  
  "aca": "https://api.firstlife.org/v6/fl/Things/614c79a06f9f21fbc3521925",  
  "date": 1661381999999  
},  
  
{  
  "_id": "63075c49cbb84e2e12e9903a",  
  "domainId": 98,  
  "actor": "61e5b514964e9b1d24a74328",  
  "aca": "https://api.firstlife.org/v6/fl/Things/614c79a06f9f21fbc3521925",  
  "date": 1661381999999,  
  "badge": {  
    "id": "62d6704a1d301a5ddcf25769",  
    "active": true,  
    "name": "master",  
    "position": 1,  
    "containerReference": {  
      "id": "62d6704a1d301a5ddcf25769",  
      "name": "Gold monthly rush",  
      "logo": "",  
      "type": "ranking",  
      "aca": "https://api.firstlife.org/v6/fl/Things/614c79a06f9f21fbc3521925",  
      "domainId": 98  
    }  
  },  
  "aca": "https://api.firstlife.org/v6/fl/Things/614c79a06f9f21fbc3521925",  
  "date": 1661381999999  
}]
```

Figura 5.4: Differenza fra log "non risolto" (a sinistra) e "risolto" (a destra).

- **page** e **pageSize**: nel corso del funzionamento del Gamification Engine e dell’assegnamento delle ricompense, vengono man mano generati sempre più log relativi. Questo porta ad averne una corposa quantità, e ritornare tutta la lista risulterebbe un’operazione tutt’altro che efficiente.

I 2 parametri di query in questione giocano un ruolo chiave nel risolvere il problema: essi permettono infatti di ritornare una risposta "paginata", ovvero di suddividere i log di assegnamento in blocchi e restituirne un singolo. Mentre `page` indica il numero di blocco da restituire (rispettando l'ordine nel database), `pageSize` specifica invece quanti log dovrà contenere quest'ultimo.

- **Ottenimento di tutti i log di assegnamento** (GET `api/v1/gamification/admin/logs`). La chiamata di questa API è limitata ad un utente amministratore, poiché restituisce i log di tutti gli utenti (si ricorda essere sempre limitato ad una verticalizzazione, in questo caso a quella dell’utente admin chiamante). I parametri di query ed il loro funzionamento rimangono gli stessi della API precedente, con l’aggiunta di un campo **actor** che permette di ottenere i log di un attore specifico.

Come già citato nel metodo `getApplicableRules()` di *Rules Module*, una delle funzioni più importanti invece del service di *Logs Assignment Module* è `canRuleBeAssigned()`. Essa permette, data in input una regola, se questa possa essere assegnabile in termini di ripetitività massimi (assoluti e/o giornalieri, sulla base dei campi definiti). Il conteggio dei log per questo scopo si basa sull'utente a cui applicare la regola, oltre eventualmente ad un'area.

```

1  async canRuleBeAssigned(
2    domainId: number, actor: string, aca: string,
3    date: Date, rule: Rule, dbSession?: ClientSession,
4  ) {
5    let ever = true;
6    let day = true;
7
8    const matchStage: PipelineStage = {
9      $match: { $and: [
10        { domainId: domainId },
11        { actor: actor },
12        { 'rule._id': rule._id },
13        aca ? { aca: aca } : { aca: { $exists: false } },
14      ] }
15    };
16
17    const groupSumStage: PipelineStage = {
18      $group: { _id: '$rule.id', count: { $sum: 1 } },
19    };
20
21    // checks for max overall repetition
22    if (rule.maxRepEver) {
23      const countEvEver = await this.logsAssignmentRepository.aggregate(
24        [matchStage, groupSumStage], dbSession);
25      ever = countEvEver[0] ? countEvEver[0].count < rule.maxRepEver : ever;
26    }
27
28    // checks for max daily repetition
29    if (rule.maxRepDay) {
30      const startHourDate = new Date(date);
31      startHourDate.setHours(0);
32      startHourDate.setMinutes(0);
33      startHourDate.setSeconds(0);
34      const endHourDate = new Date(date);
35      endHourDate.setHours(23);
36      endHourDate.setMinutes(59);
37      endHourDate.setSeconds(59);
38
39      matchStage.$match.$and.push({ date: { $gte: startHourDate, $lt: endHourDate } });
40      const countEvDay = await this.logsAssignmentRepository.aggregate(
41        [matchStage, groupSumStage], dbSession);
42      day = countEvDay[0] ? countEvDay[0].count < rule.maxRepDay : day;
43    }
44
45    return ever && day;
46  }

```

Listing 34: Funzione per verificare se una regola sia assegnabile per ripetitività.

Il codice di `canRuleBeAssigned()` mette in evidenza un ulteriore vantaggio di MongoDB, inerente in particolare alle sue *Query API*: essendo la stessa query un oggetto javascript, è possibile dichiarare variabili che ne contengano parti e comporla man mano prima di eseguirla, secondo anche determinate condizioni. In questo caso, ad esempio, sono presenti inizialmente le variabili `matchStage` e `groupSumStage`: parti di query comuni ad entrambe le operazioni da svolgere, contengono rispettivamente i controlli iniziali sui log (come id dell'utente e l'ACA) e la logica di conteggio di tutti i documenti che rispettino i criteri. Nel caso in cui sia necessaria anche la verifica sulle ripetizioni giornaliere, a `matchStage` viene aggiunta un'ulteriore parte di query, necessaria a contare i log limitatamente nella giornata specificata (`{date: { $gte: startHourDate, $lt: endHourDate }}`).

5.2.6 Rankings Module

Infine, tra i moduli degli elementi è presente *Rankings Module*, il cui compito è quello di gestire e generare gli elementi delle classifiche. Contenuto nella cartella `rankings`, e API esposte dal suo controller sono elencate qui sotto.

- **Ottenimento di una classifica per un punteggio specifico** (GET `api/v1/gamification/ranking`). I parametri di query presenti rispecchiano i criteri secondo cui una classifica viene calcolata, sulla base di quelli descritti nel capitolo di progettazione:

- `variable`, ovvero il nome del punteggio per cui gli utenti competono;
- `timeframe`, il periodo temporale preso in considerazione (tra settimanale o mensile, il periodo predefinito è il primo menzionato);
- `aca`, riferito all'area geografica di riferimento (si ricorda che se non specificato, verrà considerata l'intera verticalizzazione).

In modo più specifico alla risposta in formato JSON, come nei log di assegnamento sono presenti i campi `page` e `pageSize`. Con il trascorrere del tempo la quantità di utenti in competizione in FirstLife cresce: risulta pertanto essenziale, anche in questo caso, dare la possibilità di ottenere la classifica in modo "paginato" per mantenere un certo livello di efficienza.

- **Ottenimento della posizione di un utente in una classifica specifica** (GET `api/v1/gamification/rank`). Grazie a questa REST API, si offre la possibilità di ottenere il solo stato di un utente all'interno di una graduatoria, senza dover ritornare quest'ultima nella sua interezza (Figura 5.5).

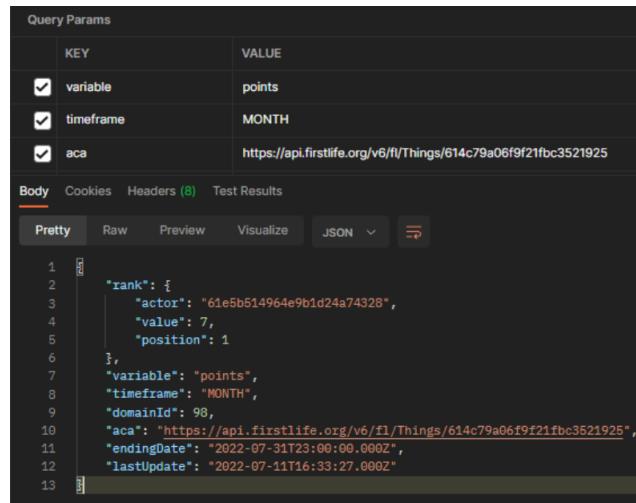


Figura 5.5: Esempio di stato di un giocatore all'interno di una classifica, calcolata con i parametri di query sopra.

L'esempio mostra lo stato di un utente in una classifica mensile della variabile “points”, all'interno dell'area specificata dal campo aca.

In relazione alle funzionalità del controller, i metodi nel service del modulo si occupano principalmente della parte di calcolo. In particolare, il metodo `calculateRanking()` visualizzato di seguito effettua la creazione effettiva dell'elemento, sulla base principalmente dei parametri passati: id della verticalizzazione e parametri di query contenuti in un oggetto di tipo `RankingsQueryDto`.

```

1  private async calculateRanking(domainId: number, query: RankingsQueryDto) {
2    const ranking = new Ranking(
3      domainId, query.aca, query.variable,
4      query.timeframe, new Date(),
5    );
6
7    const userProfiles = await this.userProfilesService.getAllMainProfiles(
8      domainId,
9    );
10   userProfiles.forEach((p) => {
11     let varValue = 0;
12     const persona = p.getPersona(query.aca);
13     if (persona) {
14       varValue = <number>(
15         persona.getFullTemporalVariables(query.timeframe, new Date())
16           .get(query.variable)
17       );
18     }
19     ranking.addRank(p.actor, varValue ? varValue : 0);
20   });
21
22   ranking.calculatePosition();
23   ranking.totalCount = ranking.rankList.length;

```

```

24     return ranking;
25 }

```

Listing 35: Funzione di calcolo della classifica.

Dopo aver creato l’oggetto di tipo `Ranking`, il quale contiene tutte le informazioni della classifica (compresa la sua data di termine e di ultimo aggiornamento), si procede alla sua generazione effettiva. Viene ottenuta la lista dei profili utente del `domainId` indicato, grazie al metodo `getAllMainProfiles()` del service di `User Profiles Module`. In seguito, per ciascuno dei profili, è aggiunta nella classifica il valore della variabile specificata dalla graduatoria, eventualmente filtrata per quelle accumulate in un ACA. Infine, sulla base di tali valori, il metodo `calculatePosition()` genera le posizioni in classifica, per ciascun giocatore.

5.3 Moduli dei meccanismi

I moduli dei meccanismi rappresenta l’ultima categoria della logica applicativa: essa comprende i moduli Nest che implementano gli altri 2 meccanismi del Gamification Engine, ovvero il meccanismo principale e i task settimanali e mensili. A differenza dei moduli degli elementi, essi possiedono una struttura diversa fra di loro: questo naturalmente ad eccezione del provider service e della classe `Module`, essenziali per un componente di questa tipologia.

I moduli sviluppati a questo scopo sono 2: `App Module` e `Tasks Module`.

5.3.1 App Module

In ogni applicazione NestJS, per `App Module` si intende innanzitutto il *modulo root* del progetto. Si occupa cioè di tutta la parte di avvio del backend: dall’inizializzazione delle dipendenze degli altri moduli, fino alla mappatura delle API di tutti controller presenti. Qui sotto sono elencati i file di `App Module`, il quale si trovano proprio nella cartella radice del progetto.

```

FirsLife-Gamification-Engine/
├── ...
└── app.controller.ts
    ├── app.module.ts
    ├── app.service.ts
    └── main.ts

```

Oltre alle normali importazioni ed esportazioni che prevede ogni modulo, il file `.module` (`app.module.ts`) possiede responsabilità maggiori da questo punto di vista. Sono presenti in altri termini anche impostazioni ed import necessari a tutti gli altri moduli realizzati: inizializzazione delle variabili di configurazione (quelle presenti in `config.ts`), indicazione a *Mongoose* dello uri del database a cui connettersi, o ancora la registrazione di `ScheduleModule` (dalla libreria `@nestjs/schedule`) per poter utilizzare i cron-jobs. Il file `main.ts` esegue invece la parte di "bootstrapping" effettiva: vengono indicati tutti gli elementi di Nest usati globalmente (fra pipe, guard o ancora exception filter), impostati i dati di Swagger, ed infine creata un’istanza dell’applicazione.

Il motivo per cui App Module viene citato in questo contesto è per il suo utilizzo nei meccanismi del Gamification Engine: si tratta del modulo che implementa la logica applicativa del meccanismo principale. Tale funzionalità è offerta dal backend mediante API (`POST /api/v1/gamification`), esposta dalla classe controller corrispondente (`app.controller.ts`): viene chiamata dalla piattaforma FirstLife a seguito dell'azione di un utente, inserendo l'evento di attività come oggetto JSON nel corpo della richiesta. L'effettiva verifica ed assegnamento delle ricompense è svolta infine da `checkEvent()`, metodo principale della classe service del modulo (`app.service.ts`).

```

1  async checkEvent(event: ActivityEvent) {
2      const gameResult: GameResult = new GameResult();
3      const coopProfileCache: AcaProfile[] = [];
4
5      const dbSession = await this.dbConnection.startSession();
6      await dbSession.withTransaction(async () => {
7          const rules: Rule[] = await this.ruleService.getApplicableRules(event, dbSession);
8
9          // if applicable rules is found, we retrieve the cooperative profiles
10         let acaPilotProfile = null;
11         let acaProfile = null;
12         if (rules && rules.length > 0) {
13             acaPilotProfile = await this.acaProfilesService.getPilot(
14                 event.domainId, dbSession);
15             coopProfileCache.push(acaPilotProfile);
16             if (event.aca) {
17                 acaProfile = await this.acaProfilesService.getOneAcaProfile(
18                     event.domainId, event.aca, true, dbSession);
19                 coopProfileCache.push(acaProfile);
20             }
21         }
22
23         for (const rule of rules) {
24             // rule target creation
25             const ruleTargets = await this.getRuleTargets(event, rule.rewards);
26             for (const target of ruleTargets) {
27                 const mainPersona = await this.userProfilesService.getUserMainProfile(
28                     event.domainId, target.actor, dbSession);
29                 const ruleResult = this.applyRule(
30                     rule, mainPersona, coopProfileCache,
31                     event.aca, target.role, event,
32                 );
33                 gameResult.logsRule.push(ruleResult.logAssignment);
34
35                 // check badges
36                 const logCoopBadges = this.checkExistingCooperativeBadges(
37                     mainPersona, acaProfile, event.timestamp);
38                 const logPilotBadges = this.checkExistingCooperativeBadges(
39                     mainPersona, acaPilotProfile, event.timestamp);

```

```

40     const logNewBadges = await this.checkBadges(
41         mainPersona, coopProfileCache, ruleResult,
42         event.timestamp, false, dbSession,
43     );
44     gameResult.logsBadges = gameResult.logsBadges.concat(
45         logCoopBadges, logPilotBadges, logNewBadges);
46
47     await this.userProfilesService.save(mainPersona, dbSession);
48 }
49 }
50
51 // save/update profiles and logs in db
52 await this.acaProfilesService.updateProfiles(coopProfileCache, dbSession);
53 await this.saveLogs(gameResult.logsRule, dbSession);
54 await this.saveLogs(gameResult.logsBadges, dbSession);
55 );
56 await dbSession.endSession();
57
58 return {
59     ruleAssignments: gameResult.logsRule.length,
60     badgeAssigned: gameResult.logsBadges.length,
61     assignmentsDate: new Date(event.timestamp),
62     domainId: event.domainId,
63     aca: event.aca ? event.aca : this.pilotName,
64 };
65 }

```

Listing 36: Funzione per verifica e distribuzione delle ricompense, data l'attività di un utente.

Prima di procedere con la logica effettiva del meccanismo, la funzione inizializza 3 oggetti molto importanti per il suo funzionamento. I primi 2 hanno finalità di mantenimento dati: `gameResult` conterrà i log che verranno generati, mentre `coopProfileCache` la lista dei profili ACA, che verranno man mano aggiornati con eventuali nuove ricompense. L'ultimo oggetto, `dbSession`, introduce per la prima volta il concetto di transazione nel Gamification Engine: il metodo in questione effettuerà diverse operazioni di scrittura e su più collezioni, rendendo quindi necessario trasformarle come un unico procedimento atomico. Viene inizialmente creata una sessione con `startSession()`, seguita dalla transazione vera e propria grazie a `withTransaction(codice da rendere atomico)`. `dbSession` verrà poi passato ad ogni metodo che interagisce con la base di dati, per indicare le operazioni che faranno parte della transazione stessa.

Con i primi 3 oggetti inizializzati, il meccanismo principale inizia il suo corso:

- Il metodo `getApplicableRules()` del service di `Rules Module` si occupa della fase di **Rules check**. Vengono restituite le regole applicabili e, se presenti, registrati i profili cooperativi coinvolti in `coopProfileCache`. In seguito, sono generati i target utente con `getRuleTargets()` per le 3 fasi successive, regola per regola.

2. `applyRule()` provvede ad assegnare i punteggi della regola sia al target corrente che ai profili ACA, come previsto da **Rule assign**. I log di regola generati in questo modo sono salvati in `gameResult`.
3. Si procede di seguito con la fase **Badges check and assign**, ulteriormente strutturata nei seguenti 2 sotto-punti:
 - (a) Nel caso in cui l'utente target abbia effettuato l'azione in un area per la prima volta, il metodo `checkExistingCooperativeBadges()` si occupa di assegnare i badge cooperativi correnti già guadagnati dagli altri giocatori. Questa azione avviene sia a livello di verticalizzazione (considerando il profilo ACA pilot), che l'eventuale ACA dell'evento di attività.
 - (b) Viene svolta la verifica per nuovi badge individuali e cooperativi grazie a `checkBadges()`, considerando nei requisiti delle ricompense le sole variabili modificate da **Rule assign**. Per ogni badge distribuito, sono generati i log di badge relativi, registrati nell'oggetto `gameResult`.
4. La funzione `save()`, chiamata dal service di **User Profiles Module**, procede con lo step di **Save user profile updates**. Vengono salvate nel database le nuove ricompense nel profilo dell'utente target.
5. Terminati i punti 3, 4 e 5 per ogni target di ogni regola applicabile, il meccanismo principale può infine svolgere la fase di **Save logs and ACA profiles**. I profili ACA vengono salvati nel db mediante `updateProfiles()`, metodo del modulo dedicato a tali tipi di elementi. Lo stesso procedimento avviene per tutti i log di assegnamento creati, grazie a `saveLogs()`. In conclusione del ciclo del meccanismo, la transazione e la sessione vengono concluse, e le informazioni sull'elaborazione restituite come risposta API.

In `AppService`, classe service di `App Module`, si ricorda infine anche al presenza di metodi incaricati di gestire altre meccaniche di gamification. Un esempio in questo ambito è la funzione `calculateRankingBadges()`, per l'assegnamento di badge competitivi nella varie competizioni.

5.3.2 Tasks Module

`Tasks Module` è il modulo incaricato di gestire i task settimanali e mensili. Si tratta dell'ultimo realizzato, andando pertanto a concludere la trattazione non solo dei moduli dei meccanismi, ma anche dell'intero Gamification Engine. Esso è contenuto nella cartella `tasks`, presentando una struttura più minimale rispetto agli elementi precedentemente descritti.

```
tasks/
└── tasks.module.ts
└── tasks.service.ts
```

Definiti gli import necessari in `tasks.module.ts`, la logica applicativa in questione si trova nel provider service, in `tasks.service.ts`. Il meccanismo considerato prevede una periodicità sulla base di un periodo specifico, settimanale o mensile: per soddisfare tale requisito il miglior approccio risulta la creazione di cron-jobs, grazie

alla libreria `@nestjs/schedule` e il suo decorator `@Cron(crontab)`. Qui di seguito, i metodi del service che adempiono proprio a questo scopo.

```

1  @Cron('0 0 3 1 * *')
2  async monthlyTasks() {
3      this.logger.log('MONTHLY TASK STARTED');
4      await this.timeframeTasks(Timeframe.MONTH);
5      this.logger.log('MONTHLY TASK ENDED');
6  }
7
8  @Cron('0 0 2 * * 1')
9  async weeklyTasks() {
10     this.logger.log('WEEKLY TASK STARTED');
11     await this.timeframeTasks(Timeframe.WEEK);
12     this.logger.log('WEEKLY TASK ENDED');
13 }
14
15 private async timeframeTasks(timeframe: Timeframe) {
16     const dbSession = await this.dbConnection.startSession();
17
18     await dbSession.withTransaction(async () => {
19         this.logger.log('Removing expired badges from user profiles... ');
20         const mainPersonaCache = await this.userProfilesService
21             .removeAllExpiredBadges(dbSession);
22
23         this.logger.log('Removing expired badges from Aca profiles... ');
24         const acaProfileCache = await this.acaProfilesService
25             .resetCooperativeBadges(dbSession);
26
27         this.logger.log(
28             `Calculating and assigning ranking badges of the
29             ${timeframe.toLowerCase()}... `);
30         await this.appService.calculateRankingBadges(
31             timeframe, mainPersonaCache, endOfDayYesterday(), dbSession);
32
33         this.logger.log('Resetting monthly and weekly temporal variables... ');
34         this.userProfilesService.resetTemporalVariables(mainPersonaCache);
35
36         this.logger.log('Saving user and aca profiles changes... ');
37         await this.userProfilesService.updateProfiles(mainPersonaCache, dbSession);
38         await this.acaProfilesService.updateProfiles(acaProfileCache, dbSession);
39     });
40     await dbSession.endSession();
41 }
```

Listing 37: Funzioni per i task settimanali e mensili.

I primi 2 metodi presentano i 2 cron-jobs realizzati nel progetto:

- `monthlyTasks()` si riferisce al task mensile, il quale viene eseguito ogni inizio del mese alle 3 del mattino;

- `weeklyTasks()` è inerente invece al task settimanale, eseguito ogni inizio della settimana al 2 del mattino.

Il motivo del discostamento delle ore è per evitare conflitti fra i 2 cron-jobs, poiché l'inizio di una settimana potrebbe coincidere con quella di un mese. Come è possibile sempre notare dal codice, `monthlyTasks()` e `weeklyTasks()` eseguono entrambe il metodo `timeframeTasks(timeframe)`, passando il proprio periodo come parametro. `timeframeTasks(timeframe)` contiene ed esegue il codice del meccanismo dei task settimanali e mensili, seguendo l'ordine progettato nel capitolo precedente. Come nel meccanismo principale, la presenza di multiple scritture ha portato alla creazione di una transazione MongoDB.

1. Definita la transazione con `withTransaction(codice da rendere atomico)`, il metodo esegue al suo interno `removeAllExpiredBadges()`, funzione del service di **User Profiles Module**. Il suo compito, secondo quanto previsto dalla fase di **Remove expired badges**, è quello di spostare i badge correnti scaduti dei profili utente nella loro lista dei rimossi. Allo stesso modo, `resetCooperativeBadges()` di **AcaProfilesService** (il service del modulo dedicato ai profili ACA), esegue la stessa operazione per i profili cooperativi. Profili utente ed ACA così modificati sono inseriti rispettivamente in `mainPersonaCache` e `acaProfileCache`: si tratta di 2 variabili di comodo per il loro futuro salvataggio effettivo in database.
2. Il prossimo passo è l'implementazione della parte "competitiva" del Gamification Engine, come già stato descritto nello step di **Assign ranking badges**. `timeframeTasks(timeframe)` chiama il metodo `calculateRankingBadges()` al fine di effettuare, per ogni contenitore di badge competitivo esistente, i seguenti sotto-passaggi.
 - (a) Viene calcolata una classifica sulla base degli attributi del container corrente: id della verticalizzazione, periodo indicato dai cron-jobs in esecuzione e variabile su cui basare la graduatoria.
 - (b) Per ogni badge nel contenitore, essi sono assegnati ai giocatori nelle posizioni in classifica corrispondenti. Viene generato, inoltre, un log di badge per ogni assegnamento fatto in questo modo.

Una volta terminata l'assegnazione di trofei (quindi, i badge di ranking), i log generati sono salvati nel database. Si nota in ultimo che la funzione `calculateRankingBadges()` viene chiamata dal service di **App Module**.

3. Segue la fase di **Reset temporal variables**, dove vengono di fatto azzerati i punteggi accumulati dagli utenti nei mesi e settimane passate. Da un punto di vista maggiormente tecnico, vengono eliminati i `temporal variable container` la cui data di fine sia inferiore a quella attuale.
4. Infine, in **Update profiles**, tutte le modifiche fatte finora ai profili sono registrate nella base di dati. Il metodo `updateProfiles()` è messo a disposizione a questo proposito, sia dal service del modulo dei profili utente, che quello dei profili cooperativi.

5.4 Risultati finali

In questo capitolo sono state descritte le principali caratteristiche del Gamification Engine: dai suoi elementi generali, fino alle funzionalità di ogni suo modulo. L'applicativo server-side così sviluppato ha permesso di raggiungere i seguenti 3 obiettivi.

1. L'esposizione, per ogni elemento del modulo di gamification, delle API necessarie alla loro gestione.
2. L'attivazione del meccanismo principale, offerto anch'esso come API, per verificare l'assegnamento di ricompense data l'azione di un utente in FirstLife.
3. La gestione di eventi periodici, come delle competizioni o il reset di punteggi mensili e settimanali, realizzata schedulando cron-jobs.

All'avvio dell'applicazione, il framework effettua una serie di operazioni per rendere attivi i servizi appena elencati. Qui sotto è mostrato il log di avvio inerente a questo passaggio, generato a livello di terminale (Figura 5.6).

```
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [NestFactory] Starting Nest application...
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] MongooseModule dependencies initialized +56ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] HttpModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] HttpModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] HttpModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] HttpModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] HttpModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] HttpModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] HttpModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] DiscoveryModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] ConfigHostModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] ConfigModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] ConfigModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] ScheduleModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] HttpModule dependencies initialized +10ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] MongooseCoreModule dependencies initialized +367ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] MongooseModule dependencies initialized +5ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] MongoModule dependencies initialized +6ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] MongooseModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] MongooseModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] MongoModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] MongoModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] TasksModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] AcaProfilesModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] RulesModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] LogsAssignmentModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] BadgesModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] UserProfilesModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] RankingModule dependencies initialized +1ms
[Nest] 18584 - 09/12/2022, 12:11:58 PM   LOG [InstanceLoader] AppModule dependencies initialized +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RoutesResolver] AppController {/api/v1/gamification}: +463ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification, POST} route +2ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RoutesResolver] RulesController {/api/v1/gamification/rules}: +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/rules, POST} route +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/rules, GET} route +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/rules/:id, GET} route +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/rules/:id, POST} route +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RoutesResolver] LogsAssignmentController {/api/v1/gamification}: +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/logs, GET} route +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/admin/logs, GET} route +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RoutesResolver] BadgesController {/api/v1/gamification}: +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/badgeContainers, POST} route +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/badgeContainers, GET} route +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/badgeContainers/:id, GET} route +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/badgeContainers/:id, PATCH} route +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/badgeContainers/:id, POST} route +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/badges/:id, POST} route +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RoutesResolver] UserProfilesController {/api/v1/gamification/profile/user}: +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/profile/user, GET} route +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RoutesResolver] AcaProfilesController {/api/v1/gamification/profile/aca}: +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/profile/aca, GET} route +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RoutesResolver] RankingController {/api/v1/gamification}: +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/ranking, GET} route +1ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [RouterExplorer] Mapped {/api/v1/gamification/rank, GET} route +0ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [NestApplication] Nest application successfully started +18ms
[Nest] 18584 - 09/12/2022, 12:11:59 PM   LOG [NestApplication] Please see the docs on: http://[::1]:42069/api
```

Figura 5.6: Log di avvio dell'applicazione Nest.

Nest utilizza una serie di oggetti per svolgere tutte le operazioni necessarie:

1. `NestFactory` provvede, in primo luogo, a creare l'istanza dell'app;
2. `InstanceLoader` si occupa in seguito inizializzare le dipendenze di ogni modulo presente, sia quelli da librerie che quelli creati nel progetto;
3. `RoutesResolver` e `RoutesExplorer` identificano rispettivamente le classi controller esistenti e loro metodi usati come endpoint, per mapparli correttamente;
4. infine `NestApplication`, l'istanza creata all'inizio del processo, indica che l'app è stata avviata con successo e il link alla pagina Swagger.

Inerente alla pagina di documentazione, essa apparirà come mostrato di seguito (Figura 5.7). All'interno sarà possibile non solo visualizzare l'elenco delle API e delle loro specifiche, ma anche effettuare chiamate per ciascuna di esse a fini di test.

The screenshot shows a Swagger UI interface for the 'FirstLife Gamification Engine APIs'. At the top, there's a dark header bar with the 'Swagger' logo and the text 'Supported by SMARTBEAR'. Below it, the main title is 'FirstLife Gamification Engine APIs' with a '1.0' version indicator and an 'OAS3' badge. A sub-header below the title says 'The APIs documentation for the Gamification Engine of FirstLife.' On the right side of the header, there's a 'Authorize' button with a lock icon. The main content area is divided into sections: 'Gamification APIs' and 'Rules APIs'. Under 'Gamification APIs', there's a single POST method for '/api/v1/gamification'. Under 'Rules APIs', there are three methods: a POST for '/api/v1/gamification/rules', a GET for '/api/v1/gamification/rules', and a GET for '/api/v1/gamification/rules/{id}'. Each method entry includes a description and a lock icon indicating security status.

Figura 5.7: Pagina di documentazione delle API del progetto, generata da Swagger.

Capitolo 6

Interfacce grafiche

Il terzo ed ultimo step del progetto di tesi è dedicato alle interfacce grafiche del modulo di gamification. Sono state sviluppate 2 applicazioni web utilizzando il framework Angular, ciascuna con un target di utenza differente:

- **Dashboard amministratore**, indirizzata agli utenti admin in una verticalizzazione FirstLife. Al suo interno, potranno creare i componenti per la propria strategia e monitorarne l'andamento.
- **Dashboard utente**, realizzata invece per tutti gli utenti della piattaforma. In essa, ciascuno potrà visualizzare il proprio profilo di giocatore e le ricompense guadagnate, limitatamente al progetto branch ove registrato.

In modo analogo alle specifiche progettuali, la creazione delle 2 interfacce ha utilizzato come punto di partenza quelle per il progetto europeo CO3, opportunamente adattate e migliorate per il contesto di FirstLife.¹

Da un punto di vista maggiormente grafico, per costruire le UI sono stati utilizzati componenti della libreria *Angular Material*. Utilizzando le linee guida di design di tali componenti, il fine è quello di rendere le interfacce il più intuitive e funzionali possibili.

Per gestire invece flussi di dati da mostrare a livello di view, la libreria *rxjs* ha giocato un ruolo chiave nella loro gestione, per evitare soprattutto inconsistenze dati.

6.1 Struttura ed elementi delle applicazioni web

Per mantenere un certo livello di omogeneità nello sviluppo di entrambi i progetti, le web-app condividono la stessa struttura da un punto di vista di organizzazione di file: si è data la priorità nel suddividere questi ultimi in base alla tipologia di elemento.

La struttura appena introdotta è mostrata qui di seguito. È presente immediatamente il file `app.module.ts`, con le principali informazioni del *modulo root* della singola applicazione Angular. Questo in aggiunta al *componente di root* previsto dal framework, con i 3 file che formano ogni componente:

¹Link alle repository: <https://gitlab.com/ontomap/otm-gamification-frontend> per la dashboard amministratore, <https://gitlab.com/ontomap/otm-gamification-user-dashboard> per la view utente.

- un template HTML, `app.component.html`;
- una classe typescript, `app.component.ts`;
- un foglio di stile, `app.component.css`.

```

app/
└── common/
    └── ...
└── model/
    └── ...
└── services/
    └── ...
└── components/
    └── ...
└── app.component.css
└── app.component.html
└── app.component.ts
└── app.module.ts

```

Come nel caso del Gamification Engine, possiamo notare la presenza della cartella `common`, contenente codice di utilità globale nel progetto. Esempi sono enumerazioni o ancora metodi di supporto.

6.1.1 Modelli

I modelli sono classi che rappresentano concetti manipolati dalle 2 applicazioni: in base al contesto ne sono presenti alcune per elementi di gamification, e altre per FirstLife. Contenute nella cartella `model`, sono utilizzate per mappare dati in arrivo dal Gamification Engine in oggetti, più usufruibili dagli elementi Angular. Ogni classe modello è contraddistinta da 3 parti:

- gli *attributi* dell'oggetto stesso, il quale rappresentano i dati dell'entità;
- *metodi* per gestire le sue informazioni, o per restituirli in un certo formato;
- una *funzione di mapping*, che serializza dati JSON passati come parametro in un oggetto di classe del modello. In alcuni casi limite questa parte è esclusa, ad esempio per classi molto semplificate o utilizzate internamente nel progetto.

Qui sotto è mostrato il modello delle regole di gamification, per la dashboard dedicata agli amministratori. Esistono classi di questo tipo che possono avere un'ulteriore versione nell'altro progetto: questo in base ai dati che ciascuna applicazione necessita di mostrare.

```

1  export class Rule {
2      _id: string;
3      description: string;
4      activityType: string;
5      rewards: { [role: string]: Variable[] };
6      activityObjectProperties: ObjectProperty[];

```

```

7   targetEntity: TargetEntity;
8   active: boolean;
9   acas: string[];
10  validFrom: number; validTo: number;
11  maxRepDay: number; maxRepEver: number;
12
13  constructor() { this._id = uuid(); }
14
15  static fromJson(json: any): Rule {
16    const rule = new Rule();
17    rule._id = json._id;
18    rule.description = json.description ? json.description : null;
19    rule.activityType = json.activityType;
20    rule.active = json.active;
21    rule.acas = json.acas;
22    rule.targetEntity = json.targetEntity;
23
24    if (json.rewards) {
25      //...
26    }
27    //...
28
29    return rule;
30  }
31  //...
32}

```

Listing 38: Classe model per le regole di gamification, nel dashboard amministratore.

In aggiunta ad attributi e costruttore della classe, è presente il metodo statico `fromJson(json data)`, il quale rappresenta l'effettiva funzione di mapping. È passato l'oggetto generico `json` come parametro e, creata una nuova istanza del modello, viene effettuato l'assegnamento dei valori dei campi (per attributi opzionali viene anche verificata la loro esistenza, come `description`). Terminato il processo, la funzione restituisce l'istanza creata inizialmente.

6.1.2 Service

Proseguendo con l'analisi della struttura, la cartella `services` contiene tutti i provider service di un'applicazione. In questo contesto, si occupano di comunicare con le API del Gamification Engine o del server di FirstLife, al fine di ottenere le informazioni necessarie.

```

1 @Injectable({providedIn: 'root'})
2 export class BadgeService {
3   constructor(private http: HttpClient) {}
4
5   getBadgesAll(): Observable<BadgeContainer<Badge>[]> {
6     return this.http.get<BadgeContainer<Badge>[]>(
7       environment.FL_GAMIFICATION_INSTANCE + '/badgeContainers', {

```

```

8     headers: { Authorization: 'Bearer ' + environment.USER_AUTH.access_token}
9   })
10    .pipe(
11      map((response: any) => {
12        const bcList = [];
13        response.forEach(bc => bcList.push(BadgeContainer.fromJson(bc)));
14        return bcList;
15      })
16    );
17  }
18 }

```

Listing 39: Uno dei provider service della dashboard utente.

La classe service sopra è utilizzata per interrogare le API dedicate ai badge. In particolare, il metodo `getBadgesAll()` ritornerà la lista dei contenitori di badge di una certa verticalizzazione, seguendo determinati passaggi.

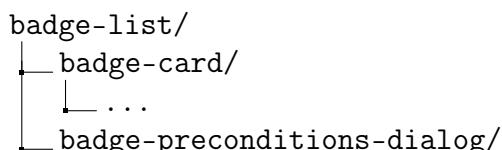
- Innanzitutto, comunica verso la API desiderata mediante `HttpClient`, oggetto utilizzato per le chiamate HTTP.
- La risposta dell'endpoint viene incapsulata in un `Observable`, proveniente dalla libreria `rxjs`. Utilizzando gli operatori di quest'ultima (in questo caso, `pipe` e `map`), le informazioni all'interno dell'Observable sono mappate secondo la classe modello `BadgeContainer`.
- `getBadgesAll()` ritorna infine un oggetto sempre di tipo Observable, ma con all'interno i dati mappati dal punto precedente (il tipo corretto, in questo caso, sarà quindi `Observable<BadgeContainer<Badge>[]>`). Un elemento Angular che vuole accedere a tali informazioni utilizzerà la metodo di `rxjs subscribe()`.

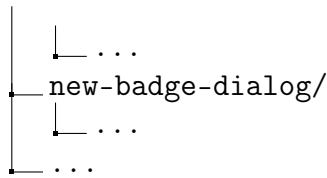
6.1.3 Componenti

Infine, nella cartella `components` sono definiti tutti i componenti di una web-app. Ciascun componente ha la propria cartella dedicata, con i 3 file che tipicamente li caratterizzano.

Sfruttando la gerarchia dei componenti per realizzare l'interfaccia grafica, `components` segue a sua volta la stessa organizzazione come directory. In altri termini, se la view di un componente padre è formata da altri 3 componenti figli (o, in ogni caso, sono integrati con le sue funzionalità), le cartelle di queste ultime saranno in quella del componente padre.

Viene preso come esempio la struttura dei file di un componente della dashboard amministratore, `badge-list`: esso utilizza al suo interno i componenti `badge-card`, `badge-preconditions-dialog` e `new-badge-dialog`, dando vita alla seguente organizzazione in cartelle.





6.2 Dashboard amministratore

La dashboard amministratore è la prima web-app sviluppata nel modulo. Si tratta dello strumento di partenza per creare l'effettiva strategia di gamification: gli amministratori ne fanno utilizzo per realizzare la propria, nella verticalizzazione di loro competenza. Per raggiungere questo scopo, e guidare gli utenti nelle 4 fasi di gamification (*Onboarding, Discovery, Partecipate* e *Create*), l'applicazione dispone di 5 schermate, suddivise in base all'utilizzo.

- Le prime 2 riguardano la **creazione degli elementi**: si tratta di progettazione e gestione di regole e badge necessari al contesto.
- Le altre 3 view che seguono spostano l'attenzione al **monitoraggio della strategia**. Vengono visualizzati aspetti come i log di assegnamento, o ancora l'andamento dei punteggi nelle ACA di una verticalizzazione.

6.2.1 Schermata delle regole

La prima schermata della dashboard riguarda gestione delle regole di gamification. Gli amministratori possono crearne di nuove, oppure visualizzare e disattivare quelle già esistenti (Figura 6.1).

FirstLife gamification admin dashboard (domain id: 98)							
Rules	Badges	Logs	Rankings	ACA Stats	Activity Type	Variable	Concept
+ Create new Rule							
Activity Type	Active	Rewards	Activity Object Properties	Target Entity	ACAs	Max Repetitions	Time Validity
contribution_added	<input type="checkbox"/>	Actor: +1 photos_added Actor: +2 points Reference Owner: +1 points	hasType: Photo				
contribution_added	<input checked="" type="checkbox"/>	Actor: +1 comment_added	hasType: Comment		Show List	Ever: 20 times	
object_created	<input checked="" type="checkbox"/>	Actor: +1 activities_created Actor: +5 points Actor: +5 xp	hasType: Event				
participation_added	<input checked="" type="checkbox"/>	Actor: +1 participations	hasType: Participation				
contribution_added	<input checked="" type="checkbox"/>	Actor: +1 videos	hasType: Video				
contribution_added	<input checked="" type="checkbox"/>	Actor: +1 comment_added Actor: +1 points	hasType: Comment				
token_created	<input checked="" type="checkbox"/>	Actor: +2 gold			Show List	Day: 5 times	
contribution_added	<input checked="" type="checkbox"/>	Actor: +1 photos_added Actor: +3 alberto_points Actor: +2 flaneur_koins Reference Owner: +1 alberto_points Reference Owner: +1 flaneur_koins	hasType: Photo	Gruppo Flaneur	Show List	Day: 20 times	
contribution_added	<input checked="" type="checkbox"/>	Actor: +1 videos Actor: +5 points Reference Owner: +3 popularity_points	hasType: Video		Show List	Day: 2 times	

Figura 6.1: Schermata delle regole.

Come è possibile vedere sopra, l'interfaccia organizza l'elenco di regole in una tabella, in particolare nel componente di Angular Material `<mat-table>`. La lista

può essere filtrata grazie ai campi nella parte superiore della schermata: tipo di attività di applicazione (**Activity Type**), punteggio che è possibile ottenere dalla regola (**Variable**) e concetto, come foto o video, che devono essere creati dall’azione (**Concept**).

A partire dalla colonna più a sinistra, sono mostrate le seguenti proprietà degli elementi presenti.

- *Descrizione e tipo di attività* da svolgere. Per il primo attributo menzionato, è necessario cliccare sul pulsante "i" per visualizzare la descrizione nel dettaglio (Figura 6.2).

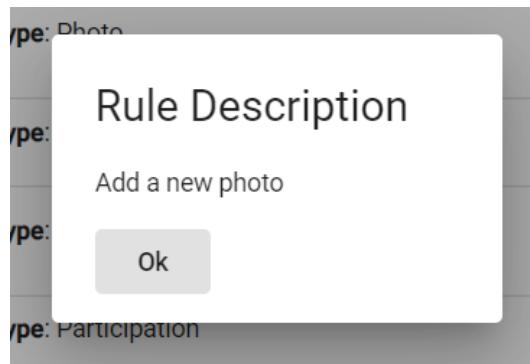


Figura 6.2: Schermata con la descrizione di una regola.

- *Stato della regola*. Il componente scelto per rappresentare questa informazione è una checkbox: un amministratore può spuntare la casella per attivare la regola, e viceversa per disattivarla.
- *ricompense ottenute* in punteggi. Come si può notare nell’omonima colonna, l’incremento della variabile prevista dalla regola è preceduta da un ruolo utente, corrispondente a chi verrà assegnata ("Actor" o "Reference Owner").
- *Oggetto che deve essere creato dall’azione*. Oltre alla tipologia dell’entità (indicata da "hasType"), sono eventualmente elencati anche altri attributi necessari (come un tag che deve possedere una foto).
- *Entità con cui si deve interagire*. Viene visualizzato sotto forma di link con il nome dell’entità stessa, il quale porta alla view dell’elemento in FirstLife.
- *Lista di ACA* ove la regola è valida. Siccome possono essere molteplici, l’elenco è raccolto in elemento espandibile, detto "accordion" (<mat-accordion> in Angular Material). Cliccando in "Show List" viene mostrata la lista sotto forma di link, ciascuno dei quali indirizza all’area di riferimento in FirstLife.
- *Numeri di applicazioni massime* (in assoluto e giornaliere) e *intervallo temporale* di validità. Sono specificate, rispettivamente, nelle colonne **Max Repetitions** e **Time Validity**.

Creazione di una nuova regola

Cliccando sul pulsante in alto a sinistra dell'interfaccia ("+ Create new rule"), l'amministratore può creare una nuova regola nella propria verticalizzazione. Viene aperta una schermata dedicata, come illustrato progressivamente in Figura 6.3.

(a)
(b)
(c)

Figura 6.3: Schermate per la creazione di una regola.

Nella prima parte della procedura (Figura 6.3a), sono mostrati gli input per le informazioni più generali dell'elemento. Nel caso si abbia bisogno di una linea guida, l'admin ha la possibilità di cercare un template da cui partire, selezionandolo dalla tendina **Templates**. Il primo campo che è possibile inserire è quello della descrizione, il quale ha un duplice scopo:

- aiutare l'amministratore nel riassumere la regola stessa, ed il suo effetto;
- suggerire agli utenti l'azione da svolgere, per ricevere le ricompense che verranno indicate. Questo aspetto verrà approfondito a proposito della dashboard utente.

Sottostante, sono elencati campi che definiscono i primi criteri di una regola. Il più generale è il tipo di attività (**Activity Type**), inserito in un input particolare detto *autocomplete* (<mat-autocomplete> in Angular Material): grazie a questo componente, la web-app effettua una ricerca sulle tipologie di azioni possibili in FirstLife, mostrate poi all'utente in base all'input da lui digitato. Allo stesso modo, un amministratore può ricercare l'entità target, scrivendo nel campo **Target Entity** (Figura 6.4).

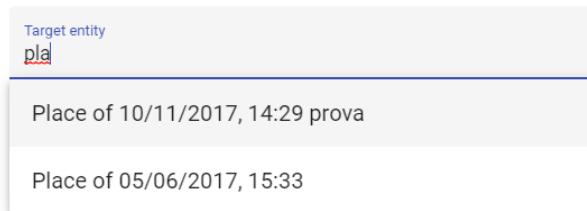


Figura 6.4: Esempio di utilizzo di un input autocomplete.

Dove le entità mostrate nell'esempio sopra, sono limitate a quelle nella verticalizzazione di riferimento. Di seguito, per indicare le aree ove la regola deve essere valida (**ACAs**), esse vengono selezionate fra quelle disponibili in un menù a tendina.

Nella seconda parte della creazione (Figura 6.3b), vengono specificate eventuali proprietà che l'utente crea con la propria azione (**Activity Object**), e i punteggi con cui ricompensare sia quest'ultimo, che possibili *reference owner*. Nel primo caso, dopo aver specificato il tipo di oggetto, è possibile aggiungere sue proprietà specifiche. Nell'immagine di esempio è stato selezionato il tipo **Places**: il tipo di oggetto possiede un attributo chiamato **isInCity**, per indicare che quel posto deve essere aggiunto in un area cittadina. Passando invece alle ricompense, l'amministratore aggiunge per i 2 ruoli i nomi delle variabili, con i rispettivi valori numerici assegnati ai giocatori.

Infine, la Figura 6.3c mostra gli ultimi criteri che possono essere definiti: quelli temporali (**Temporal Properties**). Possono essere specificate le date di inizio e fine di validità della regola, unite al numero massimo di applicazioni in assoluto e giornaliero.

Una volta che sono stati inseriti criteri e ricompense necessari, il procedimento di creazione mostra una schermata dove revisionare ulteriormente gli inserimenti (Figura 6.5). Cliccando su "Create Rule", la regola viene effettivamente registrata e aggiunta alla lista.

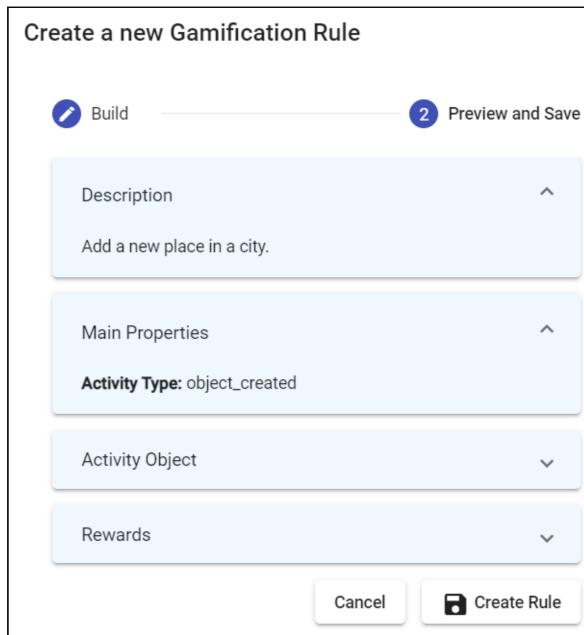


Figura 6.5: Schermata di revisione della regola da creare.

6.2.2 Schermata dei badge

La seconda schermata dell'applicazione, come la precedente, rientra in quelle di creazione di elementi di gamification. In particolare, si tratta della view riguardante tutti i badge esistenti nella verticalizzazione, più il sistema di livellamento di quest'ultima. Come è possibile osservare nella Figura 6.6, i badge sono raggruppati sulla base del container che li contiene.

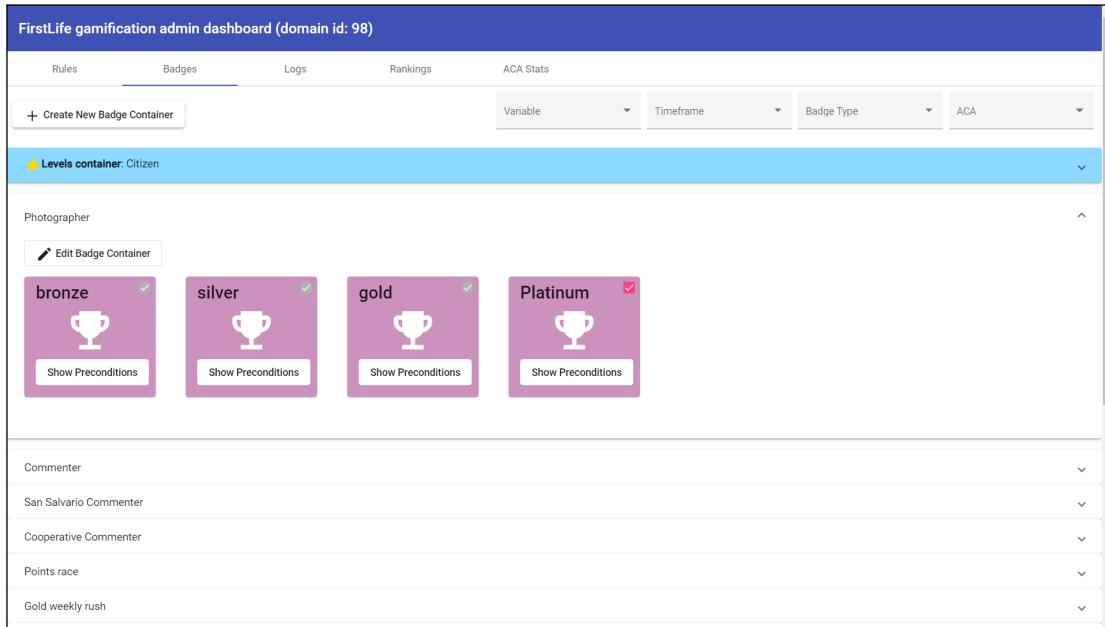


Figura 6.6: Schermata dei badge.

L’interfaccia, quindi, risulta nell’elenco dei contenitori di badge presenti. Per ognuno di essi, è mostrato il loro nome e sono rappresentati con un *accordion* come componente grafico: cliccando su di esso è possibile espanderlo e vedere tutti i badge che sono contenuti (come nel caso sopra del container "Photographer"), oltre che il pulsante per l’operazione di modifica.

Un singolo badge, a sua volta, è composto dal componente grafico di Angular Material *card* (`<mat-card>` come tag in HTML). Si occupa di racchiudere le 4 principali informazioni della ricompensa simbolica.

- Innanzitutto, è posto in evidenza il nome del badge stesso.
- Segue subito accanto una checkbox, che ne rappresenta il suo stato. Spuntandola è possibile abilitarlo, e viceversa. Un aspetto particolare che si può notare sempre nell’esempio del container "Photographer", è il fatto che le caselle in alcune card siano disattivate: questo per impedire casi di inconsistenza, come disattivare un badge che è nelle precondizioni di un altro che è attivo.
- Al centro del componente grafico è esposto il logo del badge, ovvero la parte "aesthetic" dell’elemento che fa associare un certo valore ai giocatori che puntano ad ottenerlo. Nel caso non sia specificato, viene mostrata invece l’icona di un trofeo.
- Infine, nella parte bassa della *card* è disposto un pulsante, che porta alla visualizzazione delle precondizioni ("Show precondition"). Queste ultime sono strutturate come un albero, con informazioni che possono variare anche in base alla tipologia del badge (Figura 6.7).

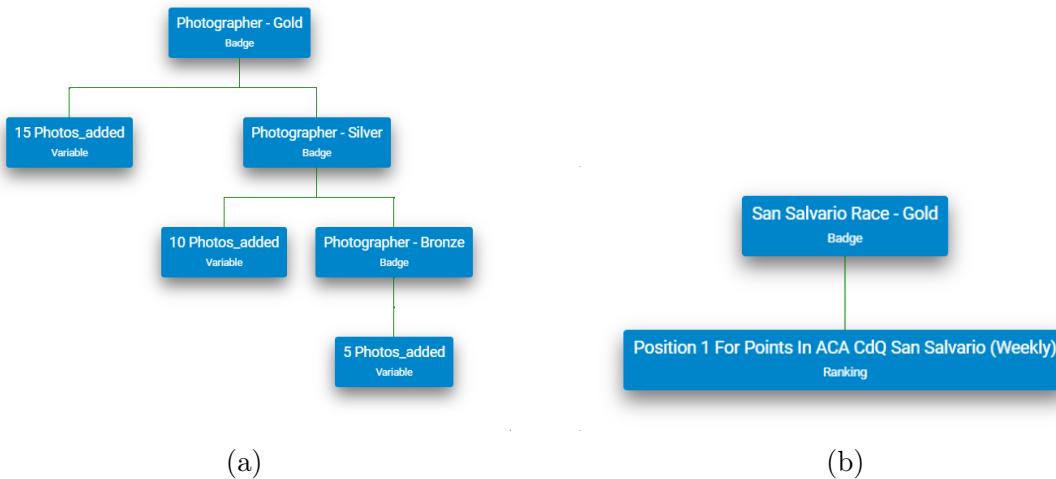


Figura 6.7: Visualizzazioni di alberi di precondizioni per i badge.

- L’albero in Figura 6.7a rappresenta le precondizioni del badge individuale **Photographer** (nome del contenitore) - **Gold** (nome del singolo badge). Sono mostrate le soglie di punteggi da raggiungere (**15 photos_added**) e i badge che devono essere posseduti (il badge **Photographer - Silver**). Nel secondo caso, vengono elencate ricorsivamente le precondizioni dei badge necessari ad ottenere quello originariamente menzionato.
- Invece, nella struttura a fianco (Figura 6.7b), è mostrato un tipico albero di precondizioni di un badge competitivo. L’esempio indica che per ottenere il trofeo **San Salvario Race - Gold**, un giocatore settimanalmente (**weekly**) deve raggiungere la prima posizione (**Position 1**) per **points**, guadagnati nell’ACA CdQ **San Salvario**.

La schermata dei badge in Figura 6.6 si conclude con la sua parte superiore. Come per la lista delle regole, anche quella dei contenitori di badge può essere filtrata secondo criteri differenti.

- Punteggio necessario per ottenere i badge (**Variable**). Un esempio può essere selezionare la variabile "photos_added", per visualizzare tutti i contenitori che possiedono badge vinti per foto aggiunte.
- Il periodo per cui un badge può essere vinto periodicamente (**Timeframe**). È possibile filtrare badge che sono ottenuti settimanalmente o mensilmente.
- Tipologia di badge, fra i 3 esistenti nel modulo di gamification: individuale, cooperativo o competitivo.
- Un area specifica ove il badge viene ottenuto (**ACA**).

Infine, è presente un contenitore di badge fissato in alto rispetto a tutti gli altri, con sfondo azzurro (**Levels Container**). Espandendo quest’ultimo, è possibile vedere quelli che sono i *badge dei livelli*, ovvero il sistema di livellamento della verticalizzazione in FirstLife.

Creazione di un nuovo badge container

Cliccando sul pulsante "+ Create New Badge Container", si apre una schermata dove un amministratore può creare un nuovo contenitore, con nuovi badge all'interno. Il procedimento è suddiviso in 3 step, ben evidenziati dalla UI grazie al componente grafico *stepper* (<mat-horizontal-stepper> in Angular Material).

La prima fase è dedicata alle informazioni generali dei badge, ovvero quelle del loro contenitore (Figura 6.8). Viene inserito il tipo di badge, il nome del contenitore ed un eventuale area, su cui basare le precondizioni da raggiungere. In base alla tipologia inserita vengono richiesti ulteriori campi, ovvero:

- nel caso dei badge cooperativi, anche la periodicità con cui vengono ottenuti Timeframe);
- per i badge competitivi, la periodicità e la variabile su cui basare la competizione (Timeframe e Variable).

The screenshot shows the first step of the badge container creation process. At the top, there are three numbered tabs: 1. Badge Container, 2. Badges, and 3. Save. The first tab is active. Below the tabs, there are two dropdown menus: 'Choose Type' set to 'Cooperative' and 'Timeframe' set to 'Weekly'. A text input field 'Name *' contains 'Cooperative scouts'. A dropdown menu 'ACA' is set to 'ACA'. A 'Next' button is at the bottom.

Figura 6.8: Inserimento informazioni generali sul contenitore.

Nell'esempio sopra, ai futuri nuovi badge cooperativi non è stata specificata alcun ACA. Ci troviamo quindi in un caso di badge cooperativi globali: i punteggi considerati per le precondizioni sono quelli di tutti i giocatori, in tutta la verticalizzazione.

Lo step successivo permette di creare i singoli badge del container in modo sequenziale, ciascuno con una serie informazioni che possono essere definite (Figura 6.9). Per "sequenziale" si intende che, cliccando il pulsante "+ Add Badge" per aggiungere un nuovo elemento, quest'ultimo avrà come precondizione di badge predefinita quello precedente, creando così un sistema di progressione interno.

The screenshot shows the second step of the badge creation process. At the top, there are three numbered tabs: 1. Badge Container, 2. Badges, and 3. Save. The second tab is active. A button '+ Add Badge' is visible. A badge named 'Novice' is shown with a trophy icon. Its name is 'Novice'. Under 'Variable Preconditions', there is a dropdown 'Variable *' set to 'places_added' and a dropdown 'Threshold *' set to '20'. Under 'Badge Preconditions', there is a dropdown 'Choose Badge' with an 'Add' button next to it. A preview box shows '20 places_added'.

Figura 6.9: Creazione dei badge del container.

I dati più generali specificati sono il nome della ricompensa simbolica ("Novice", nel caso della figura sopra) e il logo con cui viene rappresentato. Quest'ultimo viene realizzato cliccando sul pulsante "Choose image", che permette di caricare un'immagine dal proprio dispositivo per questo scopo. Le altre informazioni inerenti ciascun badge variano nuovamente in base al tipo, indicato nella fase precedente di creazione.

- Se la loro tipologia è individuale o cooperativa, la struttura di riferimento è quella di un *badge step*. I dati da specificare, quindi, saranno quelli delle loro precondizioni, suddivisi in soglie di variabili da raggiungere, e badge da possedere. Nell'esempio in Figura 6.9, vengono richiesti "20 photos_added".
- Per l'ultimo tipo rimanente (il competitivo), gli elementi creati rientrano nel caso dei *badge position*. Al posto di precondizioni, viene indicata la posizione da raggiungere in classifica per ottenere il trofeo. In questo contesto, quindi, il concetto di "sequenziale" descritto in precedenza viene ignorato.

Come per la creazione di nuove regole, nell'ultima parte del procedimento si può revisionare l'elenco di badge realizzati (Figura 6.10). L'amministratore clicca infine il pulsante "Save Badge Container" per registrare effettivamente il nuovo elemento.

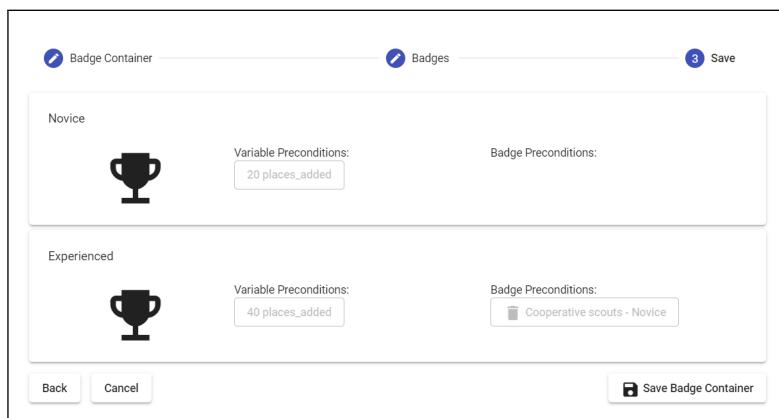


Figura 6.10: Revisione dei badge creati.

Modifica di un badge container esistente

A proposito della schermata dei badge, è stato menzionato per ogni contenitore la presenza di un pulsante che permetta di modificare quest'ultimo. Viene aperta una schermata del tutto analoga a quella di creazione, al fine di cambiare alcuni aspetti sia del contenitore, che degli elementi al suo interno. In particolare, come già stato descritto nella API dedicata del Gamification Engine, nella UI è possibile:

- modificare il nome del container;
- rinominare o sostituire il logo dei singoli badge al suo interno;
- aggiungere un nuovo badge in fondo all'elenco di quelli già presenti. Dal punto di vista di badge individuali o cooperativi, un elemento creato in questo modo rappresenta un nuovo "livello" da raggiungere nel contesto: ad esempio, creare un badge che chieda un numero di points maggiore dei precedenti.

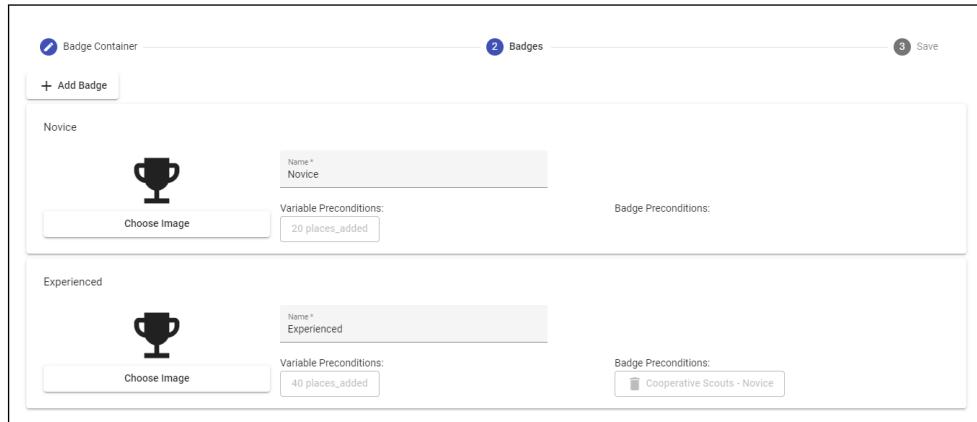


Figura 6.11: Schermata di modifica dei badge, o per crearne di nuovi.

La Figura 6.11 evidenzia inoltre che, per quei campi non modificabili dei badge, gli input risultano disattivati (in questo caso, le precondizioni).

6.2.3 Schermata dei log di assegnamento

La schermata dei log di assegnamento sposta l'attenzione sulla parte di monitoraggio della dashboard. Sono mostrati i log generati per la distribuzione di ricompense, opportunamente organizzati in una tabella. La Figura 6.12 evidenza come, in base alla tipologia dell'elemento, siano utilizzati 2 colori differenti per distinzione: azzurro per i log di regola, viola per i log di assegnamento.

FirstLife gamification admin dashboard (domain id: 98)							
Rules	Badges	Logs	Rankings	ACA Stats			
Download Data					Log Type	Date range	Actor
Actor	Date	Log Type	ACA	Rule Reward	Badge	Activity	
61e5b514964e9b1d24a74328	9/29/22, 2:21 PM	Rule Triggered	Alberto Aca	+ 1 activities_created + 5 points + 5 xp	Rule Info	Go to Activity	
61e5b514964e9b1d24a74328	9/29/22, 2:21 PM	Rule Triggered		+ 1 photos_added + 2 points	Rule Info	Go to Activity	
61e5b514964e9b1d24a74328	9/29/22, 2:21 PM	Rule Triggered	Più Spazio Quattro	+ 1 participations	Rule Info	Go to Activity	
7@CO3UUM_DEV	9/29/22, 2:21 PM	Rule Triggered		+ 1 points	Rule Info	Go to Activity	
61e5b514964e9b1d24a74328	9/29/22, 2:21 PM	Badge Assigned			Citizen - level 1		
61e5b514964e9b1d24a74328	9/29/22, 2:21 PM	Rule Triggered	Più Spazio Quattro	+ 1 activities_created + 5 points + 5 xp	Rule Info	Go to Activity	
7@CO3UUM_DEV	9/23/22, 12:21 PM	Badge Assigned			Citizen - level 0		
61e5b514964e9b1d24a74328	9/23/22, 12:21 PM	Rule Triggered	Più Spazio Quattro	+ 1 videos	Rule Info	Go to Activity	
7@CO3UUM_DEV	9/23/22, 12:21 PM	Rule Triggered	Più Spazio Quattro	+ 3 popularity_points	Rule Info	Go to Activity	
61e5b514964e9b1d24a74328	9/23/22, 12:21 PM	Rule Triggered	Più Spazio Quattro	+ 1 videos	Rule Info	Go to Activity	

Figura 6.12: Schermata dei log di assegnamento.

La parte superiore dell'interfaccia è divisa in 2 parti. A sinistra è presente il pulsante "Download Data", che permette di scaricare un file con tutti i log in formato JSON, a fine ad esempio di analisi da parte dell'amministratore. Dalla parte opposta, invece, sono disponibili diversi criteri per filtrare l'elenco sottostante: è possibile

visualizzare log di un solo tipo (**Log Type**), entro un certo intervallo temporale (**Date Range**), generati in una certa area (**ACA**) o ancora da un utente specifico (**Actor**).

La tabella in schermata visualizza i seguenti dati dei log di assegnamento.

- Nelle prime 3 colonne è possibile vedere i dati principali dell'elemento: id dell'attore coinvolto (**Actor**), data di creazione (**Date**) e tipo di log (**Log Type**), fra quello di regola ("Rule Triggered") e di badge ("Badge Assigned").
- Segue l'informazione relativa all'area dove tale log è stato creato (**ACA**), con link di riferimento a quest'ultima.
- Le ultime 3 colonne sono valorizzate in base al tipo di log di assegnamento.
 - Se si tratta di un log di regola, saranno definite le colonne **Rule Reward** e **Activity**. Nel primo caso sono elencate le ricompense distribuite (distinguendo quelle del ruolo dell'utente coinvolto nel contesto), in aggiunta ad un pulsante ("Rule Info") che permette di visualizzare le info della regola attivata (Figura 6.13).

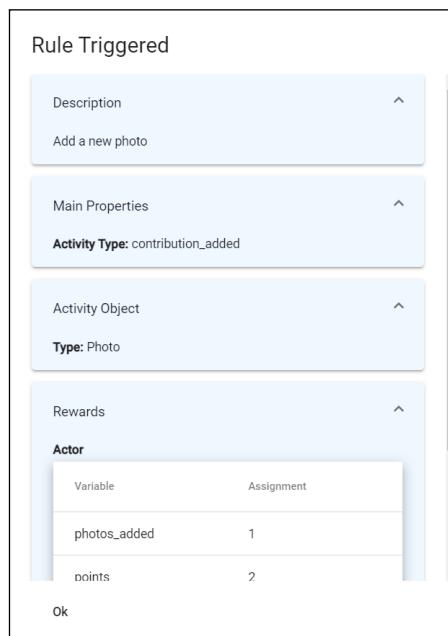


Figura 6.13: Info sulla regola attivata, da un log di assegnamento.

Per la colonna **Activity**, è messo a disposizione il link all'attività a cui è stata applicata la regola.

- Nel caso di un log di assegnazione di badge, invece, viene valorizzata la colonna **Badge** con il nome di quest'ultimo.

Infine, nella parte in basso a destra della UI, è possibile indicare quanti elementi devono essere visualizzati a schermo, e usare dei pulsanti direzionali per scorrere le pagine dei log. Queste 2 funzionalità sono disponibili in tutte le tabelle create con l'omonimo componente di Angular Material.

6.2.4 Generatore di classifiche

Sempre a fine di monitoraggio della propria strategia, la schermata del generatore di classifiche permette la visualizzazione dell'omonimo elemento, sulla base dei parametri specificati dall'amministratore. Nel particolare, serve per analizzare il grado di competizione tra gli utenti, nei confronti di un certo punteggio: di conseguenza si può valutare, ad esempio, l'aggiunta di nuove gare creando nuovi badge di ranking.

Position	Actor	Score
	61e5b514964e9b1d24a74328	17 points
	7@CO3UUM_DEV	1 points

Figura 6.14: Schermata del generatore di classifiche.

L'elemento della classifica, mostrato ancora una volta in forma tabellare (Figura 6.14), è generato in base ai criteri posti nella parte superiore della schermata.

- Fra i parametri di creazione di una graduatoria, il principale è la *variabile* per cui gli utenti competono. L'amministratore può digitare il nome del punteggio di suo interesse nel campo **Ranking Variable**, per poi caricarne la classifica relativa premendo il pulsante "Load variable ranking".
- Segue l'*ACA* in cui la competizione è circoscritta, scegliendone una dal menù a tendina dedicato (*ACA*). In caso non sia specificata l'area, la classifica considera l'intera verticalizzazione.
- Infine, la generazione si basa sul *periodo temporale* della competizione (*Timeframe*). Le opzioni disponibili sono tra una classifica settimanale ("week") o, come in Figura 6.14, mensile ("month").

6.2.5 Monitoraggio statistiche in aree

L'ultima schermata per monitorare la strategia, e anche della dashboard amministratore, permette di osservare l'andamento in aree della verticalizzazione. L'interfaccia mette a disposizione un grafico, il quale illustra la variazione di accumulo delle variabili nel tempo (Figura 6.15). Il fine principale di questa funzionalità è statistica, oltre che di analisi delle attività principali degli utenti: se, ad esempio, viene distribuito 1 `comment_points` ad ogni commento aggiunto, l'amministratore può tenere traccia di quanti di questi contenuti siano stati creati, ed eventualmente modificare la propria strategia di conseguenza.

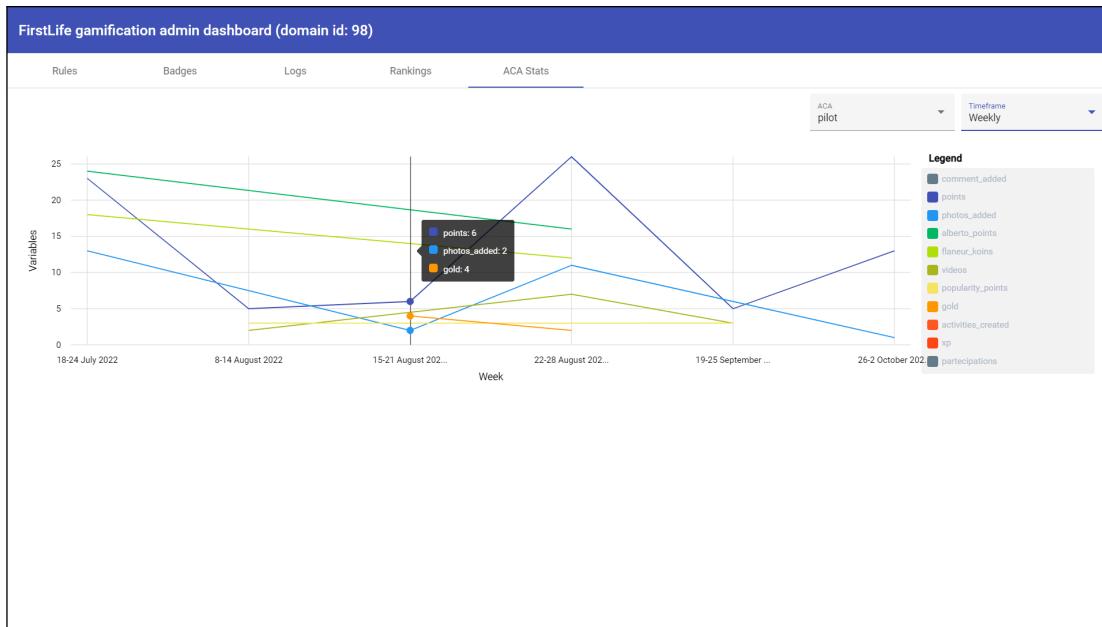


Figura 6.15: Schermata per visualizzare le statistiche in un area.

In modo analogo alle schermate già analizzate della web-app, nella parte superiore vengono indicati i filtri per ottenere i dati desiderati. È possibile selezionare l'ACA di cui interessano tali informazioni (ACA) e il periodo temporale con cui suddividere il grafico, fra settimanale e mensile (Timeframe). Se come area è specificato "pilot", allora viene considerata l'intera verticalizzazione.

Il diagramma nell'immagine sopra mostra i punteggi accumulati nell'area "pilot", con precisione settimanale. Passando il cursore sopra un singolo intervallo, si può osservare la quantità di variabili accumulata in quel periodo. Per ciascuna variabile è infine associato un colore, come mostrato nella legenda relativa.

6.3 Dashboard utente

La dashboard utente rappresenta la seconda applicazione web sviluppata nel progetto di tesi, il quale completa il modulo creato insieme al Gamification Engine e alla dashboard amministratore. Essa mette a disposizione di tutta l'utenza FirstLife una visione delle informazioni di gamification, dalle ricompense guadagnate agli obiettivi raggiunti, il tutto grazie alla loro attività svolta nella piattaforma.

I dati di gioco mostrati sono relativi alla verticalizzazione in cui l'utente ha effettuato l'accesso in quel momento: può aver guadagnato punteggi o badge in più

contesti, ciascuno corrispondente ad un progetto esterno. Anche a questo fine, la web-app è suddivisa in 3 schermate:

- **schermata principale** (o schermata *home*);
- **schermata dei badge e degli obiettivi;**
- **view delle competizioni attive.**

6.3.1 Schermata principale

La schermata principale è il punto di partenza della dashboard, oltre ad essere la prima schermata che vedranno gli utenti all'apertura dell'app. La sua interfaccia riassume i dati principali del profilo di un giocatore: il suo livello, i punteggi in evidenza e ancora la cronologia delle ricompense guadagnate (Figura 6.16).

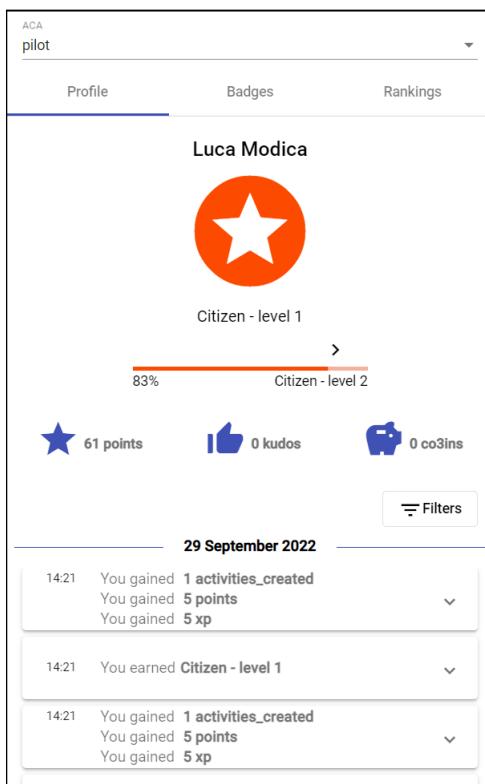


Figura 6.16: Schermata home della dashboard utente.

L'immagine, come quelle che saranno presenti nelle altre 2 schermate, mostra la UI in modalità mobile. Grazie alla libreria di componenti grafici Angular Material, è stato possibile rendere la dashboard responsive, ovvero adattabile a più dimensioni di schermo.

Un ulteriore punto in comune fra 3 parti dell'app, è la parte superiore dell'applicazione: si tratta di una tendina dove è possibile selezionare l'ACA di interesse, filtrando così le informazioni per quella sola area selezionata (Figura 6.17). Nel caso sopra è stato selezionato "pilot", corrispondente all'intera verticalizzazione: saranno quindi mostrati tutti i dati relativi a quest'ultima.



Figura 6.17: Selezione dell'area nella dashboard utente.

Tornando agli elementi in Figura 6.16, la schermata home è suddivisa del seguente modo.

- Nella parte più in alto, è presente *il nome dell'utente ed il suo livello attuale*. Il logo con la stella in sfondo arancione è quello predefinito dell'app, nel caso l'amministratore non ne abbia specificato uno nella dashboard a lui dedicata (nella sezione specifica per i badge dei livelli). La parte di livellamento è inoltre accompagnata dalla barra di progressione al livello successivo (**Citizen - level 2** nel caso sopra) ed un pulsante ">". Al click di quest'ultimo, vengono mostrati i requisiti di avanzamento.
- Subito sottostante, un utente può visualizzare le *variabili in evidenza*. Nell'esempio, sono mostrati i punteggi "points", "kudos" e "co3ins", ciascuno con la propria icona rappresentativa.
- La parte rimanente e sottostante della schermata, è dedicata alla *cronologia di assegnamento delle ricompense* all'utente, ovvero la lista di tutti i log generati per quest'ultimo. La motivazione dietro questa parte è quella di dare al giocatore una visione chiara e trasparente di ciò che accade dal punto di vista della gamification, come indicare per quale determinata azione sono stati distribuiti determinati punteggi, e quando.

Dal punto di vista interattivo, si può infatti scorrere la lista degli assegnamenti, raggruppati per data e con specificato per ciascuno l'ora della generazione (gli assegnamenti nell'immagine, ad esempio, sono stati generati il 29 settembre 2022 alle 14:21). Espandendo le informazioni un log, il giocatore può vedere l'attività per cui ha ottenuto i punteggi (per i log di regola), o nome e tipo di badge ricevuto (per i log di badge),

In ultimo, è anche possibile filtrare i log di assegnamento. Cliccando sul relativo pulsante "**filters**", si apre una finestra dove poter inserire un intervallo temporale, piuttosto il nome di un punteggio per sapere quando e quante volte è stato percepito dall'utente (Figura 6.18).

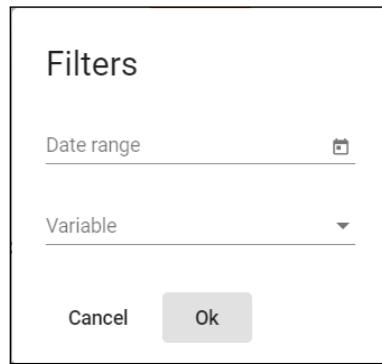


Figura 6.18: Filtri per la cronologia degli assegnamenti.

6.3.2 Schemata badge e obiettivi

Nella seconda schermata dell'app, ogni utente FirstLife visualizza i badge ottenuti grazie alla sua attività nella piattaforma, e gli obiettivi da raggiungere per ottenerne di nuovi. Le ricompense simboliche sono rappresentate dal componente *card* di Angular Material ed organizzate in un layout a griglia (Figura 6.19).

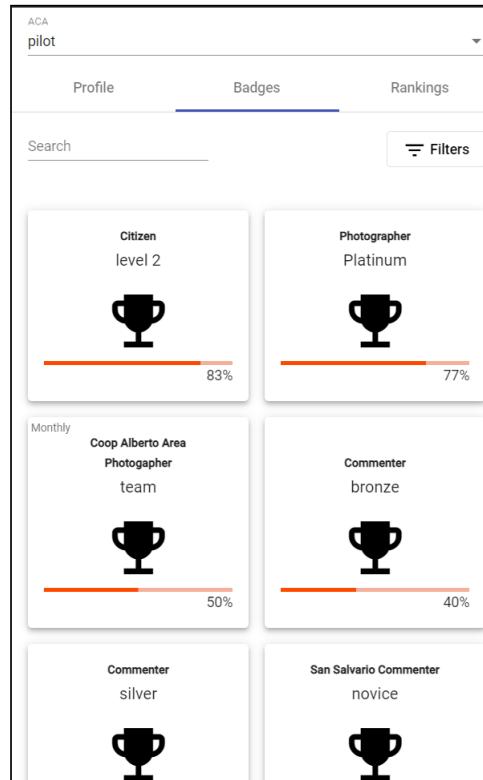


Figura 6.19: Schermata dei badge ed obiettivi da raggiungere.

La parte superiore della UI, è utilizzata per trovare e controllare le informazioni desiderate. È possibile cercare un badge per nome nell'input dedicato ("Search"), e filtrare la lista sottostante in base ai criteri mostrati in Figura 6.20, accessibili premendo "Filters".

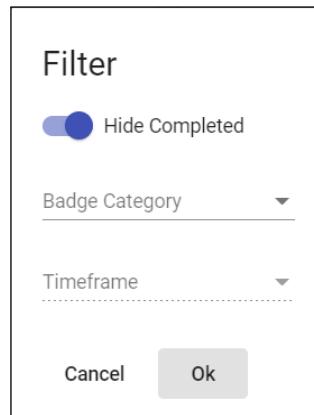


Figura 6.20: Opzioni e filtri disponibili per la schermata dei badge.

Come opzione predefinita, i badge già collezionati saranno nascosti dalla lista: questo al fine di spostare l'attenzione del giocatore verso gli obiettivi ancora da raggiungere. Se un utente desidera vedere i badge vinti, può cliccare sull'interruttore "Hide Completed".

Gli altri 2 campi permettono, invece, di filtrare in base a proprietà dell'elemento di gamification. Può essere indicato uno dei 3 tipi esistenti ("Badge Category") e, se si tratta di cooperativi o competitivi, anche la loro periodicità di ottenimento, tra ricompense settimanali o mensili ("Timeframe").

Spostando l'attenzione all'effettivo elenco di badge, viene seguito il seguente ordine di visualizzazione.

1. I primi ad essere mostrati, a meno di non essere nascosti dall'opzione "Hide Completed", sono i **badge già ottenuti dal giocatore**. Essi rappresentano gli status e gli obiettivi che ha raggiunto attualmente in FirstLife. Tali ricompense si contraddistinguono a livello di interfaccia da una *card* con sfondo dorato, e data di ottenimento (Figura 6.21).



Figura 6.21: View di un badge ottenuto.

L'esempio sopra si riferisce ad un badge di tipo individuale, **Photographer - gold**. In caso di ricompense che possono essere vinte periodicamente, come quelle cooperative o di ranking, viene specificata anche la settimana e mese in cui il giocatore è stato premiato.

Si può infine notare la presenza, al centro dell'elemento, dell'icona di un trofeo: come nella dashboard amministratore, si tratta del logo predefinito di un badge,

in caso non sia specificato. L'utente amministratore, per curarne al meglio la parte "aesthetic" e di valore espressivo, potrà caricare i loghi nella sua web-app dedicata.

2. Seguono i **badge che devono essere ottenuti**, ovvero gli obiettivi attualmente da raggiungere. Un loro elenco è mostrato nella schermata in Figura 6.19: si caratterizzano da uno sfondo bianco e, nella parte sottostante della loro *card*, da una barra di progresso, calcolata in base ai requisiti già superati dal giocatore. Fanno eccezione i trofei competitivi, il cui ottenimento è sempre determinato alla fine di una settimana o mese.

Cliccando sul riquadro di un badge, viene aperta una finestra che approfondisce in dettaglio le rispettive precondizioni da soddisfare. Ancora una volta, il come e le informazioni mostrate variano dalla tipologia della ricompensa simbolica (Figura 6.22).

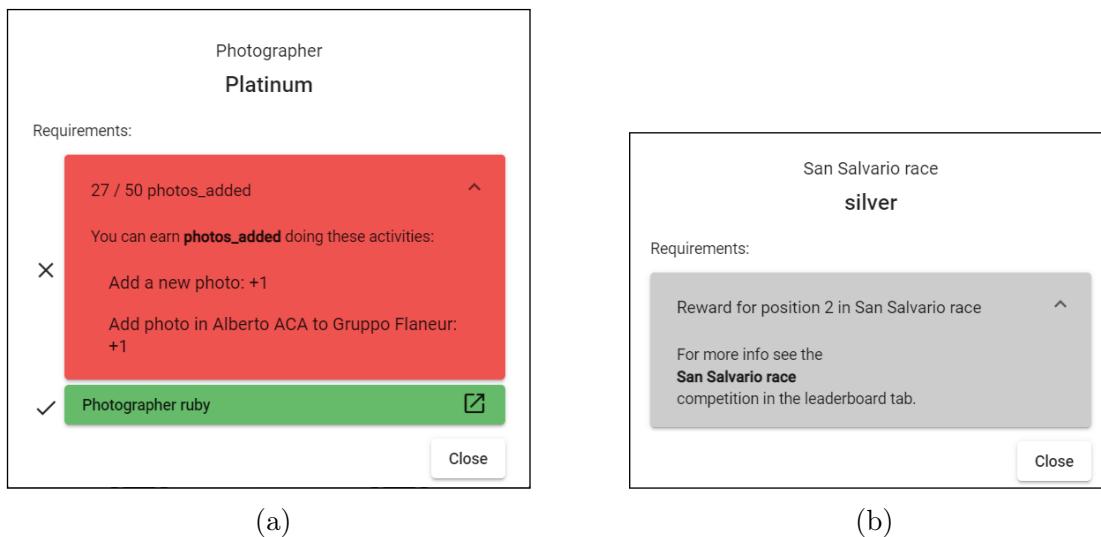


Figura 6.22: View delle precondizioni, per badge individuali e cooperativi (a), e competitivi (b).

- La view in Figura 6.22a presenta l'interfaccia per badge in formato *badge step*, quindi con una serie di precondizioni da soddisfare per ottenerli: in altri termini, è dedicata ai tipi individuale e cooperativa. Può presentare precondizioni inerenti ad soglie di variabili da raggiungere (27/50 `photos_added`), o badge da possedere.

Nel primo caso, espandendo l'*accordion* come nell'esempio, la web-app si occupa di caricare e suggerire all'utente una serie di azioni da svolgere in FirstLife, al fine di guadagnare la variabile specifica. In questo contesto, viene suggerito di aggiungere una nuova foto ("Add a new photo") per guadagnare un "`photos_added`" ("+1"). I suggerimenti di azioni, in particolare, sono generati sulla base delle regole di gamification attualmente attive in quella verticalizzazione, e che permettono all'utente di ottenere quel punteggio.

Inerente precondizioni di badge invece è messo a disposizione un link per accedere alle loro, di view delle precondizioni. Sempre nella figura di riferimento, cliccando sul badge da possedere ("Photographer ruby"), vie-

ne aperta in sovrapposizione la schermata dei suoi requisiti, con le stesse caratteristiche descritte fin'ora.

- La Figura 6.22b mostra la UI della stessa finestra, ma per i badge competitivi. Al posto delle precondizioni viste prima, viene indicata la competizione a cui fa riferimento, oltre alla posizione da raggiungere in essa per ottenere il trofeo. Se un utente vuole ricevere il badge "San Salvario race - silver" dell'esempio, deve arrivare alla seconda posizione della competizione "San Salvario race". Per avere maggiori dettagli della gara relativa, è inoltre suggerito di consultare la schermata delle competizioni, il quale sarà descritta a breve in questo capitolo.

3. La lista, in una sezione separata dal resto, si conclude con i **badge ottenuti in un evento passato**, come in mesi o settimane già trascorse. I dati visualizzati sono gli stessi dei badge vinti, ma con uno sfondo grigio (Figura 6.23).



Figura 6.23: Lista dei badge ottenuti in eventi passati.

6.3.3 View delle competizioni attive

L'ultima schermata della dashboard utente è dedicata alla parte competitiva del modulo di gamification. Al suo interno, l'utente può visualizzare tutte le competizioni attive, la sua posizione in esse, la periodicità della gara (mensile o settimanale) e ancora la data di termine (Figura 6.24). Si ricorda che ogni competizione corrisponde ad un contenitore attivo di badge di ranking, un elemento tale da assegnare trofei alla fine del suo periodo specifico.

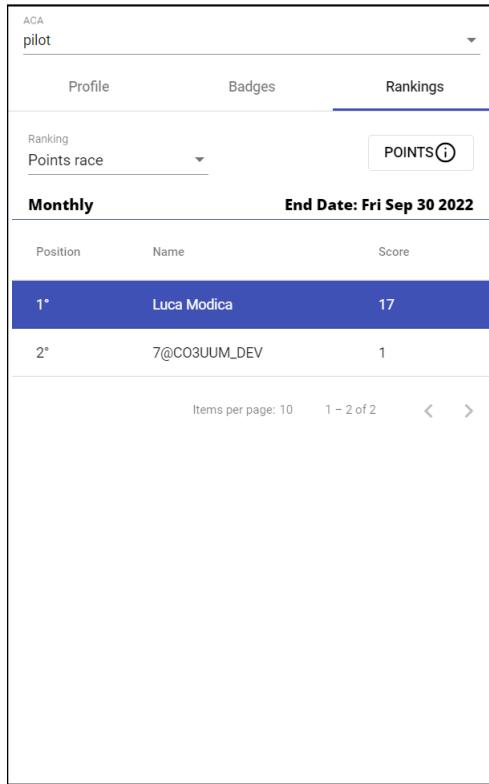


Figura 6.24: Schermata con le competizioni attive.

Nella parte superiore, gli utenti possono selezionare innanzitutto la competizione di cui consultarne le informazioni ("Ranking"). Segue subito di fianco un pulsante, con il nome della variabile su cui è basata la classifica: analogamente alle precondizioni dei badge, premendo quest'ultimo verranno mostrati in una nuova finestra suggerimenti di attività da svolgere, che permettono di guadagnare ulteriore punteggio (Figura 6.25).

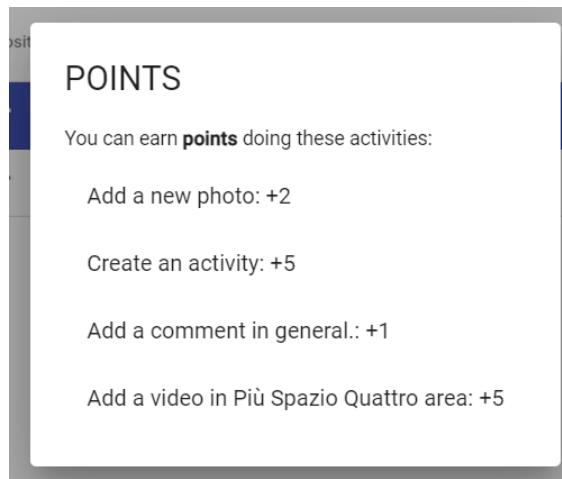


Figura 6.25: Finestra con suggerimenti per guadagnare la variabile.

Infine, la parte sottostante riassume tutti i dati della gara indicata: periodicità ("Monthly"), data di termine ("Fri Sep 30 2022") e la classifica stessa con gli utenti in competizione fra loro, organizzata con il componente della tabella di Angular Material.

Capitolo 7

Conclusioni

In questo elaborato, è stato sviluppato un modulo per implementare strategie di gamification nella piattaforma FirstLife. Utilizzando il social network civico come servizio, ogni progetto esterno ora ha disposizione uno strumento per creare elementi e meccaniche di ispirazione videoludica. Questi ultimi, seguendo l'approccio formale del framework MDA (Mechanics, Dynamics and Aesthetics), contribuiscono alla progettazione di dinamiche di gioco con determinate regole e obiettivi da raggiungere, come competizioni o collaborazioni fra cittadini. Il fine è di fidelizzare la propria utenza e migliorare l'engagement con essa, incentivando ad effettuare determinate azioni che contribuiscono all'obiettivo della verticalizzazione. È seguito un percorso di 4 fasi, che accompagna l'utente dall'essere registrato in FirstLife fino a diventare un attivo creatore di contenuti ed iniziative.

Il prossimo passo è quello di testare il sistema creato e valutarne l'impatto nella piattaforma, oltre ad osservare come gli utenti interagiscono e si comportano nei confronti di un'esperienza arricchita con le nuove meccaniche. L'architettura altamente flessibile e scalabile del modulo, data dalle tecnologie impiegate, permette non solo di adattarsi ad eventuali emersioni di nuovi requisiti, ma anche di implementare in modo relativamente agevole ulteriori elementi di gamification. Le possibilità sono davvero vaste: come negli ultimi decenni l'industria videoludica ha fatto molti progressi, allo stesso modo il numero e la tipologia di elementi applicabili in altri contesti è aumentato drasticamente. Qui di seguito, per dar prova delle potenzialità, sono descritti 2 possibili nuovi componenti.

- *Introduzione di una "happy hour"*, ossia di una determinata fascia oraria dove gli utenti possono ricevere ricompense uniche, piuttosto che raddoppiare i punteggi guadagnati. Questo aiuta ad incrementare l'attività dei cittadini in determinate parti della giornata, e quindi a portare l'attenzione in eventi con tale cadenza.
- Una delle particolarità dell'applicazione Waze, come visto nell'esempio dedicato, è di *mostrare nella propria mappa informazioni che innescino curiosità a chi la utilizza*: allerte di posti di blocco, informazioni su ingorghi stradali, o ancora una posizione molto approssimativa degli utenti, visualizzati con il loro avatar. Il possibile sviluppo futuro è quello di portare concetti molto simili anche in FirstLife. Ogni cittadino, grazie a cosmetici guadagnati partecipando a determinate attività, potrà creare il proprio avatar, personalizzarlo e renderlo visibile come foto profilo. In aggiunta, potranno apparire in alcuni momenti di utilizzo informazioni sull'attività degli utenti nella mappa, talvolta anche di

coloro che si seguono sulla piattaforma: un esempio può essere l'aggiunta di nuovi eventi in un'area specifica.

Da un punto di vista maggiormente tecnologico, infine, si propone come obiettivo futuro anche quello di *integrare il modulo di gamification con il progetto CommonsHood*. Sviluppato dall'Università di Torino, si tratta wallet-app basata su Blockchain, il quale fornisce alle comunità strumenti di inclusione finanziaria e di supporto alle economie locali [5]. Introducendo tecniche ed elementi già descritti di gamification, l'idea è quella di permettere agli utenti di guadagnare ciò che CommonsHood mette a disposizione, sempre grazie alla loro attività in FirstLife. Alcuni casi d'uso sono token per buoni sconto in negozi nelle proprie vicinanze, crypto valute spendibili su determinati beni di valore, o ancora Token Non Fungibili (NFT) che possono essere sia messi in vendita, che semplicemente collezionati dal giocatore. Dare la possibilità di accumulare ricompense anche spendibili ed elementi collezionabili unici come gli NFT, contribuirebbero molto all'efficacia della strategia progettata per la verticalizzazione: gli utenti percepiscono il guadagno di qualcosa di maggior valore, e quindi percepiscono di maggior valore anche i loro contributi nel social network civico per il proprio territorio.

Bibliografia

- [1] Royi Benita. Implementing a Generic Repository Pattern Using NestJS, 2022. <https://betterprogramming.pub/implementing-a-generic-repository-pattern-using-nestjs-fb4db1b61cce>, accessed 14/8/2022.
- [2] Jasmine Bilham. Case Study: How Duolingo Utilises Gamification To Increase User Interest - Raw.Studio, 2022. <https://raw.studio/blog/how-duolingo-utilises-gamification/>, accessed 21/9/2022.
- [3] Guido Boella, Alessia Calafiore, Elena Grassi, Amon Rapp, Luigi Sanasi, and Claudio Schifanella. Firstlife: Combining social networking and vgi to create an urban coordination and collaboration platform. *IEEE Access*, 7:63230–63246, 2019.
- [4] CO3. CO3 - CO3, Digital Disruptive Technologies to Co-create, Co-produce and Co-manage Open Public Services along with Citizens, aims at assessing the benefits and risks of disruptive technologies., 2022. <https://www.projectco3.eu/it/>, accessed 8/9/2022.
- [5] CommonsHood. CommonsHood - the internet of values, 2022. <https://www.commonshood.eu/>, accessed 22/9/2022.
- [6] Sebastian Deterding, Rilla Khaled, Lennart Nacke, and Dan Dixon. Gamification: Toward a definition. In *CHI 2011 Gamification Workshop Proceedings*, pages 12–15, 01 2011.
- [7] Google. Angular - Introduction to the Angular Docs, 2022. <https://angular.io/docs>, accessed 17/8/2022.
- [8] Google. Angular Material UI component library, 2022. <https://material.angular.io/>, accessed 17/8/2022.
- [9] Google. Introduction - Material Design, 2022. <https://material.io/design/introduction>, accessed 18/8/2022.
- [10] Robin Hunicke, Marc Leblanc, and Robert Zubek. Mda: A formal approach to game design and game research. *AAAI Workshop - Technical Report*, 1, 01 2004.
- [11] MongoDB Inc. What is MongoDB?, 2022. <https://www.mongodb.com/docs/manual/>, accessed 13/8/2022.

- [12] Sheldon Laframboise. Waze: A Beautiful Marriage of Tech, Gamification & Product Design | LinkedIn, 2022. <https://www.linkedin.com/pulse/waze-beautiful-marriage-tech-gamification-product-sheldon-laframboise/>, accessed 21/9/2022.
- [13] Erik Mechelen. Octalysis: Complete Gamification Framework - Yukai Chou, 2022. <https://yukaichou.com/gamification-examples/octalysis-complete-gamification-framework/>, accessed 21/9/2022.
- [14] NestJS. Documentation | NestJS - A progressive Node.js framework, 2022. <https://docs.nestjs.com/>, accessed 14/8/2022.
- [15] Dražen Odobašić, Damir Medak, and Mario Miler. Gamification of geographic data collection. In *Creating the GISociety – Conference Proceedings*, 07 2013.
- [16] University of Turin. FirstLife, 2022. <https://www.firstlife.org/>, accessed 3/8/2022.
- [17] Matthew Podwysocki. RxJS - Introduction, 2022. <https://rxjs.dev/guide/overview>, accessed 17/8/2022.
- [18] Inc. Postman. Postman API Platform, 2022. <https://www.postman.com/product/what-is-postman/>, accessed 17/8/2022.
- [19] SmartBear Software. OpenAPI Specification - Version 3.0.3 | Swagger, 2022. <https://swagger.io/specification/>, accessed 17/8/2022.
- [20] Wikipedia contributors. Cron — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Cron&oldid=1094795012>, 2022. [Online; accessed 16-August-2022].

E come ogni videogioco, quelli che non possono mancare sono i titoli di coda. È uno spazio per ringraziare quelle persone che mi sono state accanto in questo periodo decisamente difficile, ma che mi ha fatto crescere come persona.

La prima persona che mi sento in dovere di ringraziare è il mio relatore, il professore Claudio Schifanella. Lo ringrazio per la pazienza che ha avuto di fronte ai miei dubbi, ma soprattutto per avermi aperto a nuove opportunità e campi di ricerca di cui ignoravo l'esistenza. Sono una di quelle persone che usa abitualmente Duolingo per guadagnare punti esperienza e scalare classifiche, e non avrei mai pensato esistesse uno studio così approfondito dietro tutto questo. La gamification è un argomento davvero meraviglioso, e dopo il mio lavoro di tesi credo fermamente che un suo buon utilizzo possa migliorare le nostre vite.

Vorrei poi ringraziare Luigi Sanasi, uno degli sviluppatori principali di FirstLife. Se c'è un altro aspetto che per me è stato molto importante in questo periodo, questo è sicuramente l'esperienza acquisita con nuove tecnologie, e come utilizzarle al meglio. Luigi mi ha aiutato molto in questo, dandomi disponibilità anche per i problemi più banali.

Ringrazio mia sorella Giulia. Nell'istante in cui sto scrivendo, non ho assolutamente parole per descrivere non solo ciò che mi ha dato in questi mesi, ma da sempre. Ti voglio un mondo di bene e sono orgoglioso dei risultati che hai già raggiunto, da ambiziosa quale sei so che punterai davvero in alto.

Ringrazio Nicolò per esser stato uno dei compagni migliori di questa avventura; Nausica, per avermi sempre ascoltato e aiutato nel momento del bisogno; entrambi, per aver condiviso insieme momenti meravigliosi. Siete davvero, delle persone meravigliose.

Ringrazio Elisa, un'amica davvero d'oro. Un'amica che, in un modo o nell'altro, è sempre riuscita a tirarmi su il morale, ad ascoltarmi in momenti di crisi, il tutto con una pazienza a cui non potrei mai arrivare.

Ringrazio Marta, una persona speciale. Una persona che, seppur distanti, mi ha tenuto compagnia anche nelle parti più difficili del mio percorso.

Infine, ringrazio tutti coloro che hanno reso questo periodo impegnativo più piacevole, ricco e anche più divertente. Siete numerosissimi, e questo mi rende felice a fronte del nuovo traguardo raggiunto, del nuovo livello completato, del nuovo trofeo conquistato.

Grazie.