Relazione progetto LuckyAuction

1 Obiettivi

L'obiettivo principale del progetto consiste nella realizzazione di un sistema distribuito di aste online, in cui il focus sarà riposto sull'analisi del problema, al fine di creare un'architettura modulare orientata ai micro-servizi. La soluzione finale dovrà essere in grado di accogliere eventuali ulteriori soluzioni di configurazione (a livello applicativo) ma, condividendo sempre la stessa logica di business (dominio).

Inoltre, durante le fasi di progettazione e sviluppo del sistema, l'obiettivo è anche quello di applicare le seguenti *metodologie*:

- **DDD (Domain Driven Design)**: al fine di modellare efficacemente il dominio;
- **DevOps**: al fine di automatizzare tutti gli aspetti più implementativi e operativi, legati alle fasi di build, test e deploy del sistema. Questo significa pianificare una strategia per gestire:
  - controllo automatico della qualità del software realizzato;
  - controllo automatico delle versioni del sistema;
  - meccanismi di integrazione continua (CI);
  - deploy automatizzato.

Infine, l'ultimo obiettivo, consiste nell'integrare le seguenti *tecnologie* all'interno del sistema:

- **Stile architetturale ReST (con framework Spring)**, impiegato per gestire l'interazione manuale dell'utente con il sistema (e.g. creare una nuova asta, rilanciare una nuova offerta, ecc.);
- **Modello ad attori (con framework Akka)**, impiegato per gestire l'interazione automatica dell'utente con le aste (i.e. le automazioni delle aste);
- **Socket (con framework socket.io)**, impiegate per consentire ai vari micro-servizi e client web di reagire a eventi in real time in modalità push, secondo quindi un modello publish/subscribe.

In definitiva, lo scopo è di realizzare un *sistema distribuito eterogeneo*, strutturato a *micro-servizi*, e in grado di funzionare correttamente con diverse tecnologie, grazie all'utilizzo di opportune *interfacce*.

1.1 Scenari di utilizzo

Tutti gli scenari di utilizzo del sistema sono incentrati sull'utilizzatore finale (utente), il quale avrà diverse possibilità d'interazione con esso. Come già accennato, trattandosi di un sistema di aste online, il principale scenario di utilizzo da parte di un utente sarà quello di poter interagire con le aste, avendo la possibilità di rilanciare un'offerta per un determinato bene.

In particolare, il sistema offrirà agli utenti una duplice modalità d'interazione con le aste:

- attraverso offerte rilanciate manualmente dall'utente;
- attraverso offerte automatizzate eseguite in autonomia dal sistema per conto dell'utente.

Gli utenti potranno seguire ciascuna asta attraverso modalità interattive

differenti, senza essere vincolati ad una sola modalità. Ciò significa che, prendendo come riferimento una specifica asta, se l'utente ha attivato l'automazione per essa, non sarà vincolato esclusivamente a quella modalità ma potrà comunque effettuare rilanci manuali in qualsiasi momento, qualora lo ritenesse opportuno. In entrambi i tipi d'interazione, è previsto un sistema centrale che controlla e monitora il corretto svolgimento delle singole aste.

Ulteriori dettagli relativi ad altri possibili scenari di utilizzo saranno approfonditi in fase di knowledge crunching, con gli opportuni diagrammi dei casi d'uso.

2 Domain Driven Design

In questo capitolo si andranno a descrivere tutte le diverse fasi che portano alla definizione dei requisiti di progetto, fino all'ottenimento dell'architettura definitiva del sistema, sia a livello logico sia a livello infrastrutturale.

In seguito alla volontà di svolgere una modellazione del problema incentrata sulla logica di business (dominio) e che sia disaccoppiata da una specifica tecnologia implementativa, si è adottato l'approccio del **Domain Driven Design** (DDD). Si tratta di un approccio finalizzato ad allineare l'implementazione (il codice) con la realtà del problema attraverso la creazione di una forte e solida base di conoscenza. Per raggiungere questo obiettivo è necessario passare attraverso alcuni step: all'inizio si mettono in campo tutte quelle tecniche che servono per **capire quello che si deve fare** e solo dopo si andranno a mettere in campo pattern e funzionalità per **capire come implementare** e manutenere al meglio ciò che si è progettato.

Il principale beneficio che si ottiene adottando questo tipo di approccio è quello di rendere molto più agevole e meno noiosa tutta la fase di manutenzione del software realizzato, grazie al forte collegamento che si va a creare tra il linguaggio dell'analisi e il linguaggio del codice.

2.1 Knowledge crunching

L'approccio orientato al Domain Driven Design prevede una fase di **knowledge crunching**. Essa è di fondamentale importanza per ottenere quanta più conoscenza possibile in relazione al dominio che si deve modellare. L'obiettivo è quello di definire in modo incrementale e iterativo lo **scope di progetto** (*problem space*), ovvero, tutte le funzionalità, i vincoli ed eventuali invarianti che il sistema dovrà rispettare, in modo tale da rimanere sempre

coerenti con gli obiettivi di progetto in fase di implementazione, senza divergere, mantenendo così una buona **integrità concettuale**. Pertanto, questa fase, prevede una serie di incontri/riunioni da svolgere assieme agli esperti del dominio, i quali hanno molta esperienza e conoscenza nell'ambito, al fine di eliminare qualsiasi tipo di ambiguità e fraintendimento tra le parti.

In questo progetto, noi studenti, oltre ad assumere il ruolo di sviluppatori software, assumeremo anche il ruolo di esperti del dominio, cercando di metterci nei panni dell'utilizzatore finale per svolgere un'accurata **analisi dei requisiti**.

Lo scope del progetto è stato definito grazie ad una serie di incontri che si sono susseguiti l'uno dopo l'altro. Ciascun incontro è stato sempre caratterizzato dalla produzione di nuova documentazione/schemi che si andava ad aggiungere a quella degli incontri precedenti e, al termine di ciascun incontro, si cercava sempre di dare forma e struttura a tutta la conoscenza che stava pian piano emergendo. Procedendo in questo modo, l'intera fase di knowledge crunching ha dato origine alla seguente documentazione: Casi d'uso, User Stories, Requirements Breakdown Structure (RBS) e Conditions of Satisfaction (CoS).

2.1.1 User Stories

Le **User Stories** sono state utilizzate come mezzo iniziale per mettersi sin da subito nei panni dell'utente finale, al fine di comprendere le sue esigenze. Infatti, le user stories hanno proprio come obiettivo quello di mettere l'utilizzatore finale del nostro prodotto (l'utente) al centro dell'attenzione durante il processo di sviluppo, contribuendo a **identificare** tutti i **casi d'uso** del presente progetto.

Inoltre, si è sempre cercato di associare una **CoS** per ciascuna user story, in modo da avere ben chiaro il contributo a livello di business che l'implementazione di quella user story avrebbe portato all'intero sistema. Questo ci ha aiutato a individuare il *business value* che ogni funzionalità avrebbe portato all'utente, focalizzandoci sempre di più su quale fosse il valore da portare e a chi lo si volesse portare. Data una specifica user story, se non emergeva una CoS che portasse con sé un vero e proprio business value, allora quella user story veniva scartata senza più essere presa in considerazione.

Infine, le user stories sono anche state un mezzo attraverso il quale far emergere diversi dibattiti all'interno del team, contribuendo a definire in modo specifico tutti i **requisiti funzionali** che il sistema avrebbe dovuto avere (RBS, vedi in seguito).

Le **CoS** sono dei criteri che servono per stabilire, a fine progetto, se il progetto può essere o meno considerato di successo. Le CoS possono riferirsi strettamente alle funzionalità del progetto, ma possono anche essere relative al budget, ai tempi e alle risorse impiegate per la realizzazione dello stesso. Per questo progetto, ci siamo limitati a definire i criteri di successo relativi alle funzionalità che il sistema dovrà avere.

Il fatto di definire le CoS ci ha aiutato a pensare in termini di business, facendo emergere quelle funzionalità che portassero con sé più business value rispetto ad altre, contribuendo a evidenziare le vere priorità del progetto. Inoltre, le CoS ci hanno anche permesso di definire il perimetro del progetto, guidando così l'intero processo decisionale presente e futuro.

Un problema che molto spesso emerge tra gli sviluppatori nella fase implementativa è l'inserimento di nuove feature che non rispettano l'integrità concettuale, uscendo dal perimetro del progetto. Il problema, molto spesso, è che le decisioni vengono prese individualmente e non in maniera congiunta con tutto il team, creando inconsapevolmente tutta una serie di problemi. Le CoS servono anche per porre rimedio a questo problema, impedendo ai membri del team di introdurre feature che non siano coerenti a livello di business, evitando così di far divergere il progetto, mantenendo sempre l'integrità concettuale per l'intero processo decisionale.

Di seguito, si elencano le diverse user stories individuate.

Titolo: (1) Sistema di autenticazione degli utenti
Descrizione: Come utente, vorrei che il sistema mettesse a disposizione un meccanismo sicuro di autenticazione, al fine di proteggere i miei dati relativi alle attività che svolgo all'interno del sistema stesso.
CoS: Offrire agli utenti un valido meccanismo di autenticazione.

Titolo: (2) Sistema di autorizzazione degli utenti
Descrizione: Come committente, in vista di eventuali cambiamenti futuri, vorrei predisporre il sistema in modo tale da poter assegnare dei ruoli specifici agli utenti. In futuro, vorrei offrire la possibilità agli utenti di accedere a servizi "premium" solo a coloro che sottoscriveranno un piano di abbonamento.
CoS: Predisporre il sistema in vista di sviluppi futuri, con la possibilità di assegnare uno o più ruoli agli utenti.

Titolo: (3) Sistema di aste con budget
Descrizione: Come committente, vorrei predisporre un sistema di aste online, che sia robusto a fronte di eventuali falsi offerenti, introducendo il concetto di budget. Ogni utente avrà quindi un proprio budget, che potrà ricaricare quando desidera oppure incrementare quando riuscirà a vendere un prodotto mediante

un’asta. Dunque, per un utente, avere budget sufficiente sarà condizione necessaria affinché possa effettuare nuovi rilanci. Inoltre, quando l’utente deciderà di effettuare un rilancio, la cifra da lui specificata dovrà essere “congelata”, rimuovendola temporaneamente dal budget in attesa di capire se riuscirà ad aggiudicarsi l’asta oppure no. Nel caso in cui sopraggiunga un'offerta più alta da parte di un altro utente, l’ammontare di denaro congelato tornerà ad essere disponibile nel budget del primo offerente; al contrario, se l’asta termina senza ulteriori offerte, il denaro congelato verrà effettivamente decurtato dal budget del primo offerente, generando una vera e propria transazione.
CoS: Gestione delle aste con budget utente, robusto ai falsi offerenti.
NOTA: Una volta eseguito, il rilancio di una nuova offerta non potrà più essere ritirato/annullato in futuro.

Titolo: (4) Gestione budget utente
Descrizione: Come utente vorrei poter aggiungere denaro al mio budget, in modo da poter rilanciare offerte alle aste. Inoltre, vorrei avere la possibilità di ritirare il denaro dal mio budget qualora non mi serva più, ad esempio perché non sono riuscito ad aggiudicarmi un’asta oppure perché l'ho vinta ad un prezzo inferiore della somma di denaro che avevo versato inizialmente.
CoS: Offrire all'utente piena libertà nella gestione del proprio budget.

Titolo: (5) Trasparenza dei movimenti
Descrizione: Come utente, vorrei avere ben chiari tutti i miei movimenti, sia in termini di ricariche e prelievi, sia di offerte correnti e di aste aggiudicate. In altre parole, vorrei vedere il mio saldo attuale con lo storico di tutti i movimenti che lo giustificano. Ad esempio, se, partendo da un budget pari a 0€, ricarico 500€ e partecipo a due aste rilanciando 100€ nella prima e 200€ nella seconda, mi aspetto di vedere il budget corrente a 200€, e tra i movimenti +500€, -100€ e -200€, ciascuno con il riferimento all’asta a cui l’offerta si riferisce. Se poi qualcuno supererà la mia offerta, ad esempio nella seconda asta, tra i miei movimenti dovrà scomparire il -200€ e il budget dovrà essere incrementato di 200€, tornando così a 400€.
CoS: Possibilità da parte dell’utente di visualizzare il proprio budget con annesso lo storico dei movimenti.
NOTA: La somma dello storico di tutti i singoli movimenti deve corrispondere al budget attuale.

Titolo: (6) Creazione asta
Descrizione: Come cliente, vorrei dare la possibilità agli utenti di creare un’asta per un prodotto, con la possibilità di scegliere la durata (da un minimo di un'ora a un massimo di sette giorni) e un prezzo di partenza.
CoS: Possibilità da parte dell’utente di creare nuove aste.

Titolo: (7) Durata estendibile di un’asta
Descrizione: Come committente, vorrei che i miei utenti, alla creazione dell’asta, abbiano la possibilità di scegliere se estendere o meno la sua durata, nel caso in cui non sia arrivata alcuna offerta. Inoltre, l’asta potrà essere estesa una sola volta e per un massimo di ulteriori sette giorni. Nel caso in cui non si dovessero registrare offerte nemmeno durante il periodo di estensione, l’asta verrà dichiarata conclusa.
CoS: Possibilità da parte dell'utente di estendere la durata di un’asta.

Titolo: (8) Rilancio dell'offerta per un’asta
Descrizione: Come cliente, vorrei che gli utenti possano rilanciare un’offerta per una certa asta a patto che la nuova superi di almeno 1€ l’offerta corrente e che il budget dell’utente offerente sia almeno sufficiente.
CoS: Possibilità da parte dell'utente di rilanciare l'offerta di un'asta.

Titolo: (9) Garantire equità di partecipazione tra gli utenti
Descrizione: Come committente, mi aspetto un sistema di aste equo che impedisca a persone furbe di aspettare lo scadere dell'asta per rilanciare una nuova offerta, rubando di fatto l’asta a chi se l’era quasi aggiudicata. Più precisamente, è vero, sì, che un’asta ha una certa durata, ma se qualcuno rilancia un'offerta negli ultimi trenta minuti (*ending zone*), gradirei che il sistema estenda in automatico l’asta di un certo tempo (e.g. un’ora), per dare il tempo anche ai vecchi offerenti di pensare se rilanciare o meno con una nuova offerta.
CoS: Garantire equità di partecipazione durante lo svolgimento di un'asta.

Titolo: (10) Automazione dei rilanci
Descrizione: Come committente, vorrei offrire la possibilità agli utenti di partecipare alle aste anche nel caso in cui non abbiano la possibilità di seguirle in tempo reale, impostando così una sorta di automazione in cui sarà il sistema a effettuare offerte ricorrenti al posto dell'utente. Questa funzionalità potrebbe essere utile, per esempio, nel caso in cui l'utente si trovi al lavoro nel momento di chiusura (previsto) dell'asta. In questo modo, attraverso una configurazione opportuna (i.e. prezzo limite, incremento di prezzo, durata, intervallo di tempo dopo il quale effettuare nuovi rilanci), l'utente potrà partecipare attivamente ad un'asta pur non seguendola in prima persona.
CoS: Possibilità da parte dell’utente di automatizzare i rilanci di un'asta.

Titolo: (11) Monitoraggio asta
Descrizione: Come utente, gradirei che ogni asta abbia una propria pagina dedicata in cui poter vedere alcuni dettagli. Per esempio, reputo essenziale la presenza di una tabella contenente tutte le offerte che hanno preso parte fino a quel momento. Inoltre, credo sia fondamentale poter visualizzare anche il

tempo rimanente al fine di ottimizzare la propria strategia per i successivi rilanci.
CoS: Possibilità da parte dell'utente di monitorare un'asta, avendo a disposizione tutte le informazioni utili.

Titolo: (12) Conclusione asta
Descrizione: Come committente, vorrei che al termine dell'asta venga trasferito automaticamente il denaro tra il venditore e il vincitore. La somma del budget dei due utenti, prima e dopo lo scambio di denaro, deve rimanere costante.
CoS: Automatizzare lo scambio di denaro tra venditore e acquirente ad asta conclusa.

Titolo: (13) Riepilogo vendita
Descrizione: Come committente, per le aste aggiudicate, vorrei che venisse generato un riepilogo dell'asta appena conclusa in formato PDF, contenente mail e recapiti telefonici del venditore e vincitore per consentire loro di mettersi in contatto e accordarsi per il ritiro a domicilio o per la spedizione del prodotto in oggetto. Il report non deve contenere informazioni sensibili, quali: password, numero della carta di credito e anagrafica.
CoS: Per le aste aggiudicate, generare un file pdf affinché venditore e vincitore possano mettersi in contatto per decidere come, quando e dove effettuare la consegna del prodotto in oggetto.

Titolo: (14) Filtrare le aste nella homepage
Descrizione: Come utente, vorrei poter filtrare le aste in base a categoria e stato dei prodotti, in modo da individuare più velocemente i prodotti di mio interesse.
CoS: Filtrare le aste per categoria e/o stato prodotto.

Titolo: (15) Ordinare le aste nella homepage
Descrizione: Come utente, al fine di focalizzare meglio l'attenzione su alcune aste, vorrei avere la possibilità di ordinare le aste in base ad alcuni criteri, quali: prezzo più alto, prezzo più basso, in scadenza, appena inserite.
CoS: Visualizzare le aste ordinate secondo un criterio specifico.

Titolo: (16) Gestione storico aste utente
Descrizione: Come utente mi piacerebbe trovare una sezione dedicata allo storico di tutte le aste a cui ho partecipato. Inoltre, sarebbe utile avere anche una sezione con le aste attualmente in corso a cui sto partecipando, in modo tale da poterle raggiungere facilmente. Infine, nel caso in cui decidessi di mettere all'asta alcuni articoli, desidererei avere a mia disposizione una sezione dedicata per le aste da me create, riuscendo cosi a monitorarle costantemente.

CoS: Possibilità da parte dell'utente di visualizzare lo storico di tutte le proprie aste.

2.1.2 Casi d'uso

Oltre alle user stories ci si è avvalsi anche dei diagrammi dei casi d'uso, uno strumento più formale che consente di avere una visione d'insieme più completa, fornendo al contempo un riepilogo grafico delle funzionalità precedentemente descritte.

Partiamo dall'analisi dei casi d'uso generali, rappresentanti quindi le situazioni più rilevanti dell'applicazione. Come osserviamo dalla figura 2.1, quasi tutte le operazioni richiedono che l'utente abbia effettuato il login (dipendenza di tipo *include*), a eccezione ovviamente del login stesso e del caso d'uso di ricerca delle aste. Poniamo un momento maggiore attenzione sui casi d'uso riguardanti il budget e i rilanci. Come si evince, oltre a richiedere la validazione dell'utente (i.e. validità del token), tutti e tre hanno una dipendenza di tipo *extend* da "Aggiorna budget"; ciò significa che quest'ultimo caso d'uso verrà richiamato opzionalmente, se vengono soddisfatte determinate condizioni. A questo livello di dettaglio non è rilevante sapere esattamente quali saranno, ma a intuizione, possiamo immaginare che se l'utente non avrà abbastanza budget per effettuare un prelievo, non ci sarà da effettuare nessun aggiornamento del budget, idem per quanto riguarda un rilancio. Infine, anche la ricarica potrebbe fallire per qualche ragione, ad esempio se l'utente non sta fornendo un token valido.

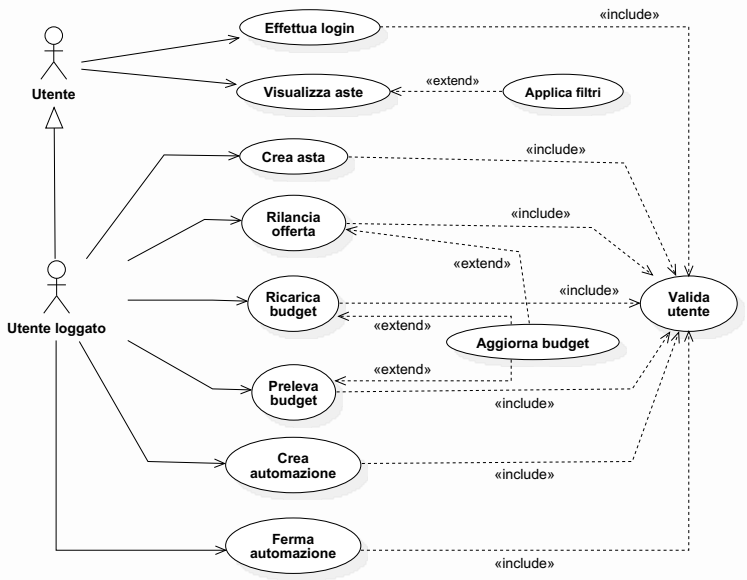


Figura 2.1 - Casi d'uso principali

Entriamo ora più nello specifico, mostrando i casi d'uso riguardanti lo svolgimento di un'asta (figura 2.2).

Iniziamo, per comodità, a leggere il grafico dalla parte in alto a sinistra, partendo quindi dall'offerente, a prescindere che sia l'utente fisico o l'automazione avviata dall'utente. L'azione principale che può effettuare un utente all'interno di un'asta è ovviamente quella di rilanciare un'offerta. A fronte di un valido rilancio, bisogna scongelare i soldi del vecchio offerente, ammesso che ci sia, congelare il budget dell'utente che sta attualmente rilanciando, ed eventualmente estendere l'asta se il rilancio avviene in *ending zone*. Ancora una volta tutte le dipendenze sono di tipo *extend*, dal momento che il nuovo rilancio potrebbe non essere valido e/o potrebbe non esserci un'offerta per l'asta in questione (quindi nessun budget da scongelare).

Come ultima situazione da analizzare rimane ora quella legata alla terminazione di un'asta. Passiamo quindi ora alla parte in basso a sinistra. Questa volta il caso d'uso di terminazione dell'asta viene invocato dal sistema, al momento opportuno. La procedura di terminazione asta varierà in base ad alcuni aspetti: se almeno un utente ha rilanciato per quell'asta, si avvia la procedura che scongela i soldi precedentemente congelati all'ultimo offerente (a questo punto il vincitore), e si effettua poi l'effettivo trasferimento di denaro al venditore. Altrimenti, se non ci sono offerte, il sistema verifica se è opportuno o meno estendere la durata dell'asta, ed eventualmente terminarla.

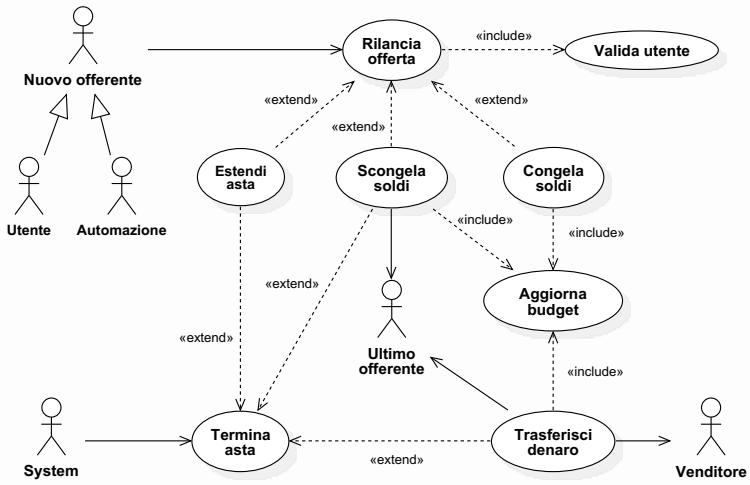


Figura 2.2 - Casi d'uso delle aste

Si precisa che nei casi d'uso appena descritti non viene mai tenuto in considerazione il fattore temporale, quindi non esiste un ordine con cui eseguire o leggere i casi d'uso; essi rappresentano semplicemente le diverse funzionalità che il sistema offre agli utenti. Una precisa descrizione cronologica degli eventi e dei diversi vincoli sarà discussa nelle fasi di analisi successive, mediante i diagrammi di attività e di sequenza.

2.1.3 Analisi dei requisiti

Sulla base dell'analisi svolta fino a questo momento e con lo scopo di definire chiaramente tutte le funzionalità che il sistema deve offrire, ci si è avvalsi della RBS (Requirements Breakdown Structure).

La **RBS** consiste in una definizione gerarchica di tutti i**requisiti funzionali** del sistema. Grazie alla sua struttura gerarchica, una RBS ha come vantaggio quello di far capire le dipendenze che ci sono tra i sotto-requisiti con i requisiti di più alto livello. Per realizzarla, si è partiti dalla definizione degli obiettivi di più alto livello (requisiti di primo livello), per poi creare livelli interni sempre più specifici.

NOTA: I numeri tra parentesi che si trovano alla destra dei requisiti di primo livello identificano le user stories dalle quali sono stati estrapolati.

1. Autenticare gli utenti (1)
   - 1.1 Registrare un nuovo utente all'interno del sistema
   - 1.2 Effettuare il login di un utente attraverso username e password
2. Autorizzare gli utenti (2)
   - 2.1 Possibilità di assegnare dei ruoli agli utenti
   - 2.2 Possibilità di configurare l'accesso a risorse specifiche in base al ruolo
3. Gestione budget utente (3, 4)
   - 3.1 Ricarica budget mediante carta
   - 3.2 Prelievo budget dall'account verso la carta
   - 3.3 Congelamento della corrispondente parte del budget quando si rilancia un'offerta
   - 3.4 Scongelamento della somma congelata qualora l'offerta venga superata
   - 3.5 Ad asta conclusa, trasferimento del budget tra acquirente e venditore
4. Storico dei movimenti (5)
   - 4.1 Budget attuale
   - 4.2 Ricariche effettuate
   - 4.3 Prelievi effettuati
   - 4.4 Congelamenti temporanei
   - 4.5 Addebiti permanenti a fronte di aste aggiudicate
5. Gestione di un'asta (6, 7, 8, 9, 11, 12, 13)
   - 5.1 Creare una nuova asta per uno specifico prodotto
   - 5.2 Possibilità di scelta se estendere o meno la durata di un'asta
   - 5.3 Estensione della durata dell'asta di un'ora se qualcuno fa un rilancio negli ultimi trenta minuti
   - 5.4 Visualizzare il tempo rimanente
   - 5.5 Visualizzare il prezzo attuale
   - 5.6 Visualizzare tutte le offerte effettuate dagli utenti

5.7 Possibilità di rilanciare l'offerta di almeno 1€ superiore al prezzo attuale
5.8 Terminare l'asta allo scadere del tempo
5.9 Al termine dell'asta, trasferimento di denaro tra venditore e vincitore
5.10 Al termine dell'asta, creare il report finale in formato PDF
6. Rilanci automatici (10)
6.1 Possibilità di configurare e attivare rilanci automatici per un'asta
6.2 Possibilità di disattivare i rilanci automatici
7. Homepage (14, 15)
7.1 Visualizzare tutte le aste in corso
7.2 Possibilità di filtrare le aste
7.3 Possibilità di ordinare le aste
8. Storico delle aste (16)
8.1 Aste create dall'utente
8.2 Aste concluse a cui l'utente ha partecipato
8.3 Aste in corso a cui l'utente sta partecipando

Infine, i **requisiti non funzionali** del sistema sono:

- Adozione di uno stile architetturale **ReST** (con framework **Spring**);
- Integrazione di un **modello ad attori** (con framework **Akka**) nella parte di automazione delle aste (i.e. rilanci automatici);
- Notifiche di eventi in modalità push mediante l'utilizzo di socket (con framework **socket.io**);
- Implementazione della **sicurezza** mediante **token JWT**;
- Persistenza dei dati su database **MongoDB** in cloud;
- Creazione di una **build Gradle multi progetto**;
- Realizzazione di due tipologie di test:
  - **Unit test** per i livelli di dominio, casi d'uso e persistenza;
  - **Integration test** per il livello del controller, al fine di testare la comunicazione tra i vari micro-servizi;
- Uso di **git-flow** come strumento di sviluppo;
- Uso di **TravisCI** per la Continuous Integration;
- Utilizzo di **Docker** per esemplificare il deployment mediante l'utilizzo di container.

2.1.4 Ubiquitous Language

Un aspetto fondamentale del Domain Driven Design è l'uso coerente di un linguaggio onnipresente nel sistema, ovvero un linguaggio comune che possa essere usato a partire dalle conversazioni con gli esperti del dominio, fino ad arrivare direttamente al codice. Definire un linguaggio comune è importante soprattutto nelle fasi di knowledge crunching e di analisi dei requisiti, al fine di

eliminare ambiguità semantiche, evitando quanto più possibile eventuali dubbi interpretativi in relazione ad alcuni termini adottati. Inoltre, questo porta ad abbassare drasticamente la probabilità di ritrovarsi nelle fasi più avanzate del progetto in cui ci si accorge che manca coerenza terminologica. Il linguaggio deve essere il più fine possibile e vicino a quello adottato in fase di analisi perché è in quella fase che, grazie soprattutto agli esperti di dominio, emergono i termini più importanti e centrali del problema. Inoltre, al fine di eliminare ogni ambiguità, non deve mai essere sottovalutato alcun aspetto del problema perché ogni termine adottato, anche in base al contesto in cui viene utilizzato, assume sempre un proprio significato e una propria importanza. Dunque, per aumentare notevolmente le probabilità di successo di un progetto è importante definire chiaramente un linguaggio ubiquo, senza mai dare nulla per scontato, riducendo così il rischio di fraintendere ciò che gli esperti del dominio intendono. Nel caso in cui non si è perfettamente allineati su uno stesso linguaggio, il rischio potrebbe essere quello di portare il progetto verso una direzione sbagliata

Nel presente progetto, il processo di definizione del linguaggio ubiquo si è evoluto man mano che cresceva la comprensione del dominio da parte del team. In particolare, sono state create alcune **mappe concettuali** contenenti tutta la terminologia necessaria per riuscire a modellare con successo l'intero dominio. Queste mappe le abbiamo trovate utili e allo stesso tempo intuitive perché forniscono una visione completa del dominio, riuscendo cosi a contestualizzare rapidamente ogni termine. In aggiunta, per ogni mappa, è stato prodotto un **glossario** contenente il significato di tutti i termini adottati, al fine di eliminare eventuali dubbi interpretativi.

In conclusione, la creazione delle mappe concettuali e del glossario, ci ha stimolato a ottenere una copertura terminologica che fosse in grado di raggiungere ogni aspetto del dominio. Inoltre, ha anche consentito di eliminare ambiguità presenti in alcuni casi d'uso, i quali erano soggetti a differenti interpretazioni. Per esempio, durante una seduta di confronto, ci si è accorti che stava emergendo ambiguità e indecisione sull'adozione di alcuni termini relativi alla durata di un'asta. Nello specifico, in alcune occasioni veniva adottato il termine "elapsed time", mentre altre volte si adottava il termine "duration". Oppure, ancora, veniva utilizzato il termine "duration" sia per indicare la durata iniziale dell'asta (specificata dall'utente al momento della creazione), sia per indicare la durata complessiva dell'asta. Quindi, ci si è resi conto sin da subito di quanto fosse importante definire un glossario, riuscendo cosi a dare il giusto significato ad ogni termine, ottenendo molta più chiarezza.

La figura 2.3 fornisce una divisione ad alto livello di tutti i macro-concetti che contribuiscono alla completa definizione del dominio.

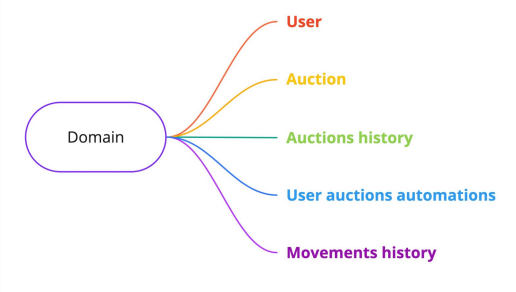
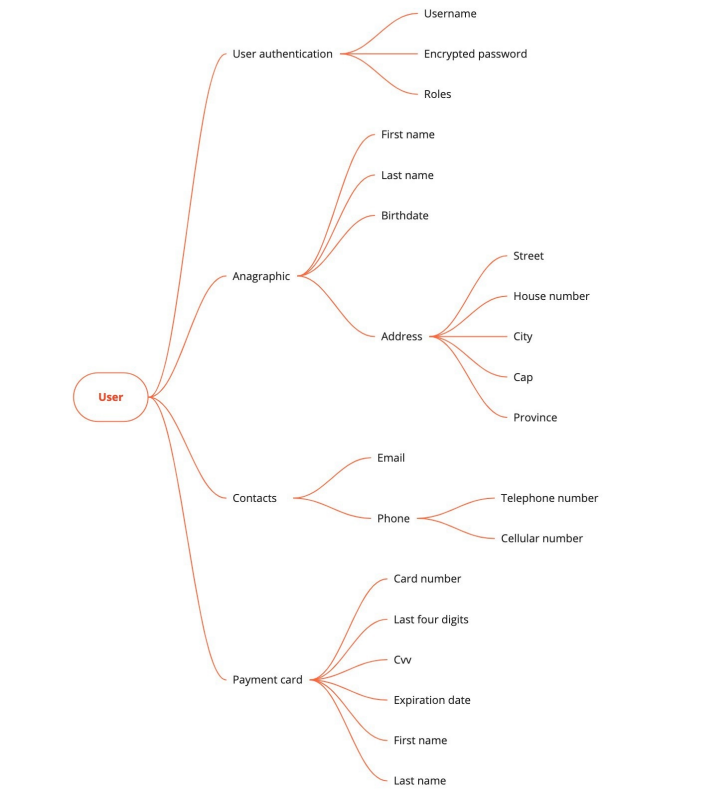
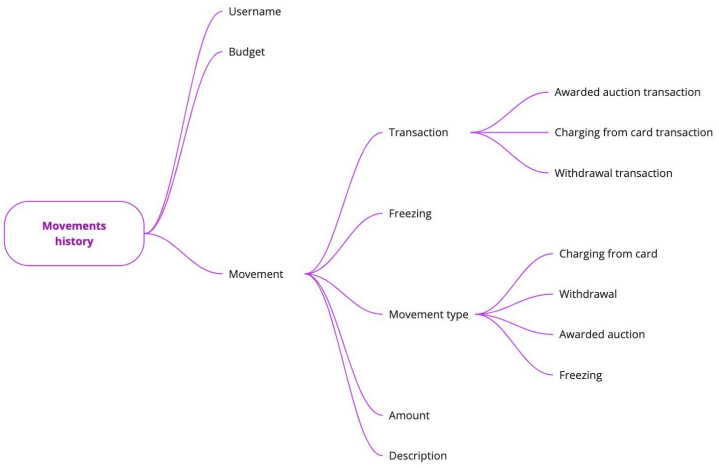
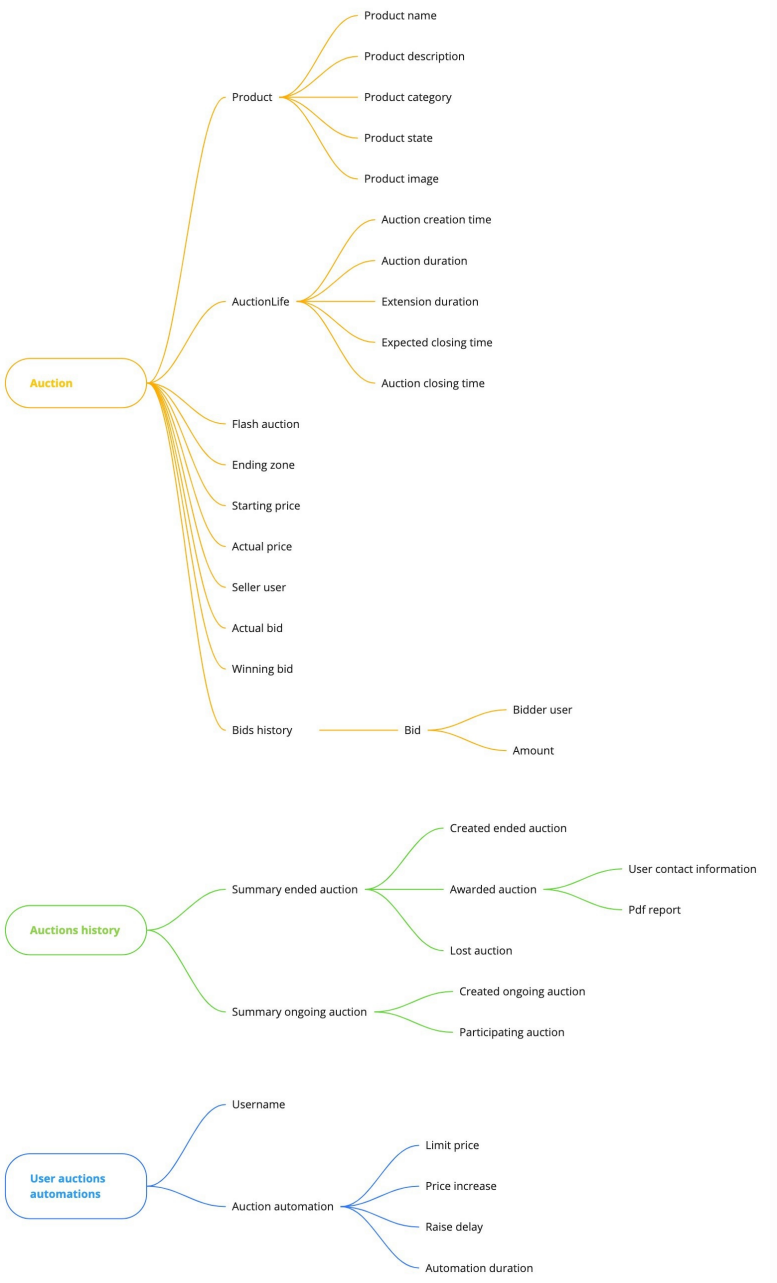


Figura 2.3 - Mappa concettuale di alto livello contenente tutti i macro-concetti del dominio.

Di seguito vengono mostrate le mappe concettuali relative a ciascun ambito del dominio, con il rispettivo glossario.


User
User authentication
Username
Encrypted password
Roles
Anagraphic
First name
Last name
Birthdate
Address
Street
House number
City
Cap
Province
Contacts
Email
Phone
Telephone number
Cellular number
Payment card
Card number
Last four digits
Cvv
Expiration date
First name
Last name


Movements history
Username
Budget
Movement
Transaction
Awarded auction transaction
Charging from card transaction
Withdrawal transaction
Freezing
Movement type
Charging from card
Withdrawal
Awarded auction
Freezing
Amount
Description



- **User**: persona fisica che avrà accesso al sistema e alle funzionalità messe a disposizione da esso.
- **Username**: stringa alfanumerica utilizzata da un utente per farsi identificare dal sistema.
- **User authentication**: dati associati all'utente affinché il sistema possa riconoscerne l'autenticità al momento del log-in (accesso al sistema). I dati associati all'utente sono: username, password criptata e ruoli.
- **Role**: ruoli che un utente ha all'interno del sistema (e.g. user, admin, ecc.).

- **Anagraphic**: dati anagrafici di un utente. I dati anagrafici comprendono: nome, cognome, data di nascita e indirizzo di residenza.
- **Contacts**: informazioni relative alle modalità di contatto di un utente. I contatti richiesti sono: email e numero di cellulare. Opzionalmente, viene richiesto un altro numero di telefono come contatto aggiuntivo.
- **Phone**: contatto telefonico di un utente. Può essere un numero di casa, un numero di cellulare, ecc.
- **Payment card**: carta di credito o debito valida. Al fine di accertare la validità della carta, sono richiesti i seguenti dati: numero della carta, cvv, data di scadenza, nome e cognome dell'intestatario.
- **Auction**: asta con aggiudicazione al maggiore offerente.
- **Product**: prodotto inteso come oggetto della vendita di una specifica asta. Un prodotto è formato dalle seguenti caratteristiche: nome, descrizione, categoria, stato e immagine.
- **AuctionLife**: concetto raffigurante informazioni circa la vita dell'asta, quali: tempo di creazione, durata, estensione della durata, tempo di fine atteso e tempo di chiusura.
- **Auction creation time**: istante di tempo in cui l'asta è stata creata.
- **Auction duration**: durata dell'asta, specificata dall'utente al momento della creazione. Può essere specificata in giorni e/o in ore. La durata minima è di un'ora, mentre quella massima di sette giorni. Ad asta conclusa, questo campo corrisponderà alla durata effettiva dell'asta.
- **Extension duration** (opzionale): specifica di quanto estendere l'asta nel caso in cui non si siano registrate offerte durate il regolare svolgimento. Minimo un'ora, massimo sette giorni.
- **Expected closing time**: instante di tempo (futuro) in cui ci si aspetta che l'asta termini. L'utente può direttamente specificare questo campo al posto della durata.
- **Auction closing time**: istante di tempo in cui l'asta è stata chiusa definitivamente dal sistema. Dopo la chiusura dell'asta non si potrà più partecipare in alcun modo. Tempo di fine, tempo di chiusura e tempo di terminazione sono sinonimi per riferirsi al medesimo termine.
- **Flash auction** (o asta flash, asta veloce): rappresenta un'asta che ha una durata iniziale al più di tre ore. Nel caso non sia specificato niente, si

intende sempre un'asta della durata maggiore di tre ore.

- **Ending zone**: fase conclusiva dell'asta, ovvero gli ultimi quindici minuti nel caso di un'asta flash, altrimenti gli ultimi trenta minuti. Questa zona è fondamentale per rinviare la fine dell'asta qualora qualcuno rilanci in ending zone, in modo da dare tempo sufficiente ad altre persone di rilanciare. Se si rilancia in ending zone in un'asta flash, l'asta viene estesa (i.e. posticipata) di trenta minuti, che diventano un'ora nel caso di aste standard.
- **Starting price**: prezzo iniziale di un'asta.
- **Seller user**: venditore di una specifica asta. Per ogni asta, ci sarà un solo e unico venditore.
- **Bid**: offerta relativa a una specifica asta. Un'offerta fa riferimento a una e una sola asta. Un'offerta è caratterizzata dell'istante di tempo di cui è stata effettuata, dal suo ammontare in euro e dallo username dell'utente offerente.
- **Bidder user**: utente offerente, associato ad una specifica offerta.
- **Actual bid**: offerta attuale di una specifica asta.
- **Actual price**: prezzo attuale di un'asta (i.e. valore dell'ultima offerta, se presente, oppure il prezzo iniziale).
- **Winning bid** offerta vincente di un'asta.
- **Bids history**: storico di tutte le offerte di un'asta.
- **Minimum price raise**: prezzo minimo di rilancio dell'offerta, i.e. di quanto deve essere superata l'ultima offerta, come minimo.
- **Auctions history**: storico di tutte le aste con cui l'utente ha interagito. Lo storico delle aste comprende: aste create in corso, aste create concluse, aste vinte, aste perse, aste a cui si sta partecipando.
- **Summary ended auction**: riepilogo di un asta conclusa.
- **Summary ongoing auction**: riepilogo di un'asta in corso di svolgimento.
- **Created ongoing auction**: asta creata precedentemente e non ancora conclusa.
- **Participating auction**: asta in corso a cui un utente sta partecipando, ovvero per cui ha effettuato almeno un rilancio o ha attivato

un'automazione.

- **Awarded auction**: asta vinta (aggiudicata).
- **Created ended auction**: asta creata precedentemente e che si è conclusa.
- **Lost auction**: asta persa. Si intende un'asta a cui l'utente ha partecipato, ma non è riuscito ad aggiudicarsela.
- **Pdf report**: resoconto in formato PDF di un'asta aggiudicata. Tale resoconto viene generato dinamicamente, al momento del bisogno, ed è comprensivo di: id dell'asta, nome del prodotto in oggetto, tempo di creazione, tempo di fine, prezzo iniziale, prezzo di fine, informazioni di contatto del venditore e informazioni di contatto del compratore (vincitore).
- **User contact information**: informazioni necessarie al fine di mettere in contatto venditore e vincitore per accordarsi sulle modalità di consegna del prodotto.
- **Auction automation**: automazione relativa ad un'asta avviata da parte di uno specifico utente. L'automazione è caratterizzata da una serie di proprietà che consentono di gestire i rilanci automatici che effettuerà il sistema per conto dell'utente. Le proprietà di cui ogni automazione è caratterizzata sono le seguenti: prezzo limite, incremento di prezzo, ritardo del rilancio delle offerte, durata dell'automazione.
- **Limit price**: prezzo limite di una specifica automazione. Tale prezzo sancisce il limite massimo oltre il quale, se raggiunto, l'automazione terminerà.
- **Price increase**: incremento di prezzo di ogni offerta successiva.
- **Raise delay**: intervallo di tempo (ritardo) da aspettare prima di effettuare un nuovo rilancio, a partire da quando viene ricevuta la notifica di nuovo rilancio.
- **Automation duration**: durata di una specifica automazione. Passato tale tempo l'automazione terminerà a prescindere.
- **User auctions automations**: automazioni attive di uno specifico utente.
- **Movements history**: storico di tutti i movimenti effettuati dall'utente a partire dal momento di creazione dell'account.
- **Amount**: cifra di denaro, espressa in euro.
- **Budget**: denaro disponibile di un utente, espresso in euro. Se un utente

non ha budget sufficiente, non potrà rilanciare offerte per le aste di suo interesse.

- **Movement**: azione volta ad aggiornare il budget utente, mediante un movimento di denaro. Con il termine movimento si comprendono sia le transazioni, sia i congelamenti.
- **Freezing**: congelamento (sottrazione) di una parte del budget utente. Questa azione viene eseguita automaticamente dal sistema quando un utente effettua un rilancio dell'offerta per una specifica asta.
- **Transaction**: spostamento di denaro relativo alle operazioni di prelievo, ricarica o a fronte di un'asta aggiudicata (vedi in seguito).
- **Awarded auction**: è un tipo di transazione che viene eseguita automaticamente dal sistema nel momento in cui un utente si aggiudica (vince) un'asta. L'operazione consiste nello spostare il denaro dall'account acquirente verso l'account venditore (quindi entrambi gli utenti vedranno questa transazione nel loro storico).
- **Charging from card**: è un tipo di transazione che ha luogo nel momento in cui un utente effettua una ricarica del proprio budget tramite la propria carta di credito/debito.
- **Withdrawal**: è un tipo di transazione. che avviene nel momento in cui un utente effettua un prelievo del proprio budget, versando lo specifico ammontare nella propria carta di credito/debito.

2.2 Context Maps

Dopo aver analizzato accuratamente i requisiti del progetto, si è deciso di suddividere lo spazio del problema in opportuni bounded context (contesti delimitati), i quali hanno permesso di individuare e isolare a livello logico diversi sotto-domini, agevolando cosi l'implementazione complessiva del dominio.

Sin dalle fasi iniziali di progettazione, si è fatto uso della **context map** per rappresentare la struttura del dominio. Nel corso dei primi incontri, la mappa ha subito diverse evoluzioni venendo costantemente aggiornata, al fine di riflettere sempre la situazione attuale. Infatti, lo scopo della context map è proprio quello di mostrare come sono integrati attualmente i bounded context, non come dovranno essere in futuro. Inoltre, l'obiettivo è anche quello di identificare le relazioni che intercorrono tra i vari bounded context e i relativi team di sviluppo.

La context map è mostrata in figura 2.4 e da come si può notare, sono presenti quattro bounded context:

- User (gestione utenti)
- Movement (gestione movimenti)
- Auction (gestione aste e storico aste)
- AuctionAutomation (gestione automazioni aste)

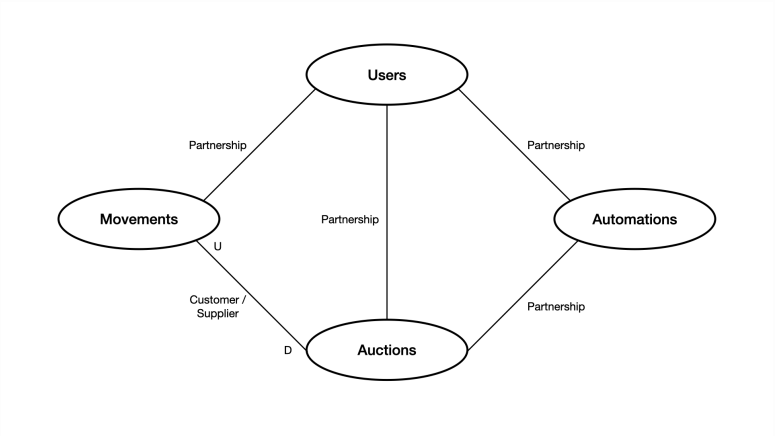


Figura 2.4 - Context map.

La relazione maggiormente presente tra i diversi bounded context è sicuramente quella di "**Partnership**". Questa relazione, tra due bounded context, indica che è presente una forte cooperazione sull'integrazione dei due contesti da parte dei rispettivi team di sviluppo, i quali lavorano e collaborano verso un obiettivo comune.

L'unica eccezione è fatta dalla relazione "**Customer/Supplier**" tra il contesto delle aste e quello dei movimenti. In questo caso, la relazione indica che il contesto a valle (aste) si basa sul contesto a monte (movimenti). Quindi, il contesto a valle (downstream) per integrarsi con il contesto a monte (upstream), deve conformarsi a esso.

Maggiori dettagli riguardo ciascun bounded context saranno forniti nella sezione successiva.

2.3 Pattern tattici

In seguito alla fase di knowledge crunching, dopo aver capito ciò che si deve realizzare, è possibile definire i diversi bounded context. Pertanto, in questa sezione verranno descritti i **pattern tattici** impiegati per definire il modello del dominio (**solution space**) dei diversi bounded context.

Sin dall'inizio della modellazione ci si è chiesti se avesse senso adottare gli **aggregati** all'interno dei bounded context. Il fatto di usare o meno gli aggregati è sicuramente una scelta che va discussa nelle fasi iniziali, perché se ci si accorge a progettazione già avanzata della necessità di introdurre gli aggregati, risulta molto complesso inserirli in quanto sarà necessaria una profonda riorganizzazione. Pertanto, pensando anche a eventuali sviluppi futuri, si è deciso di adottare questo pattern per una migliore organizzazione del dominio.

Ogni aggregato è composto da un insieme di **entity** e **value object** che tra loro condividono delle invarianti comuni, consentendo di mantenere una forte consistenza interna tra gli elementi che lo compongono. La consistenza viene garantita dalla presenza di una root, la quale funge da unico punto di accesso. Nelle figure mostrate successivamente per illustrare la modellazione di ogni bounded context, l'*aggregate root* viene individuata da un cerchio di colore grigio e una freccia entrante.

Oltre agli aggregati, si è fatto uso del **pattern factory**, il quale è stato implementato attraverso i pattern Factory e Builder della Gang of Four (GoF). L'idea è stata quella di demandare la creazione delle entità ad un elemento specifico in modo tale da garantire una creazione efficace dell'oggetto, permettendo di personalizzare gli oggetti a piacimento o creare una versione particolare a seconda delle esigenze, pur sempre rispettando le invarianti previste. La creazione di un oggetto all'interno del modello del dominio è un aspetto delicato, in quanto è un punto in cui si potrebbe creare inconsistenza. Dunque, l'utilizzo di questo pattern si è rivelato fondamentale durante lo sviluppo.

Infine, sono stati impiegati anche i **domain services** e i **domain events**, ma solo nel contesto relativo alle aste. Nella sezione dedicata verranno illustrate le opportune motivazioni in relazione a queste scelte.

Di seguito vengono illustrate le mappe relative ad ogni bounded context.

NOTA: Tutti i significati dei termini adottati all'interno delle figure possono essere consultati nella specifica sezione dedicata al linguaggio ubiquo.

2.3.1 User context

Il bounded context degli utenti è dedicato alla completa gestione dei dati personali di un utente (i.e. dati di autenticazione, dati di contatto, dati anagrafici, dati di pagamento). La figura 2.5 mostra la modellazione di questo contesto.

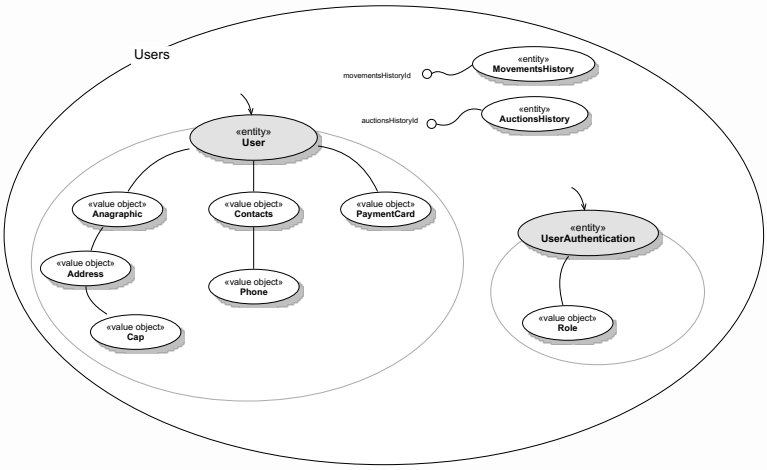


Figura 2.5 - Bounded context degli utenti.

In questo sotto-dominio sono presenti due aggregati: UserAuthentication e User. Il primo si occupa di gestire tutte le informazioni che consento all'utente di accedere al sistema in completa sicurezza. Mentre il secondo si occupa di gestire tutti i dati che riguardano l'utente in prima persona, quali: canali di comunicazione (contatti), anagrafica e dati di pagamento. Oltre a ciò, è presente anche una relazione verso il dominio dei movimenti e quello delle aste, per raggiungere le rispettive cronologie dell'utente.

Molto spesso quando si pensa al modello del dominio viene spontaneo pensare che si tratti di un grafo completamente connesso. In realtà, focalizzandoci sull'aggregato degli utenti e analizzandolo bene, ci si rende conto che un grafo non completamente connesso risulta essere molto più utile. Infatti, come possiamo notare dalla figura 2.5, il collegamento con il dominio dei movimenti e quello delle aste viene stabilito attraverso l'identità delle entità che sono coinvolte nel rispettivo aggregato. In particolare, anche se un utente ha molti movimenti a esso associati, nel modello del dominio non è importante il fatto che un utente abbia sempre con sé tutta la collezione dei propri movimenti. Nel progetto in questione, ogni volta che il dominio degli utenti viene idratato (recuperato dal database) non si è interessati anche ai movimenti; per questo motivo si è deciso di disaccoppiarli. Nel caso in cui si abbia la necessità di recuperarli, sarà sempre possibile farlo grazie all'identità delle entità. Discorso analogo anche per le aste.

2.3.2 Movement context

Il bounded context dei movimenti è dedicato alla modellazione delle transazioni e dei congelamenti; nonché alla completa gestione del budget utente. La sua modellazione è mostrata nella figura 2.6.

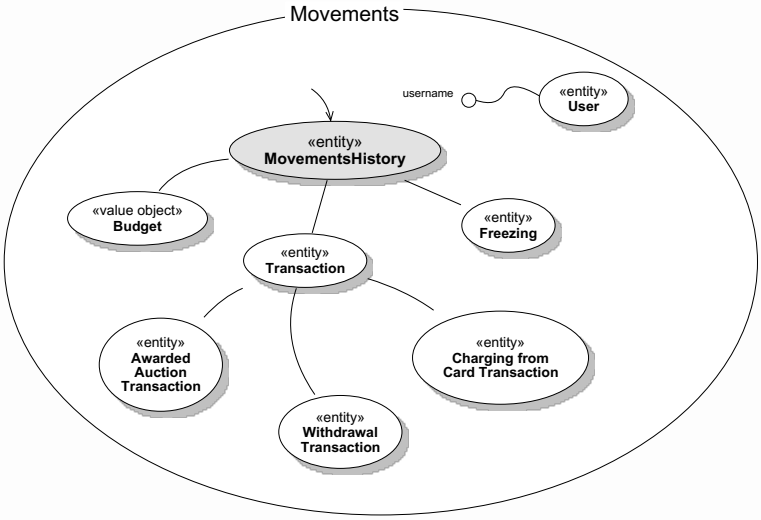


Figura 2.6 - Bounded context dei movimenti.

Questo bounded context è composto da un unico aggregato: MovementsHistory, finalizzato a modellare lo storico di tutti i movimenti di uno specifico utente. Questo aggregato è nato a fronte delle user stories (3), (4) e (5), le quali evidenziano la necessità da parte di un utente di visualizzare lo storico di tutti i propri movimenti. A tal proposito, sono state individuate quattro tipologie di entità differenti: Freezing, AwardedAuctionTransaction, ChargingFromCardTransaction e WithdrawalTransaction.

2.3.3 Auction context

Il bounded context delle aste si occupa di modellare sia il concetto di asta sia il concetto di storico delle aste. La sua modellazione è mostrata nella figura 2.7.

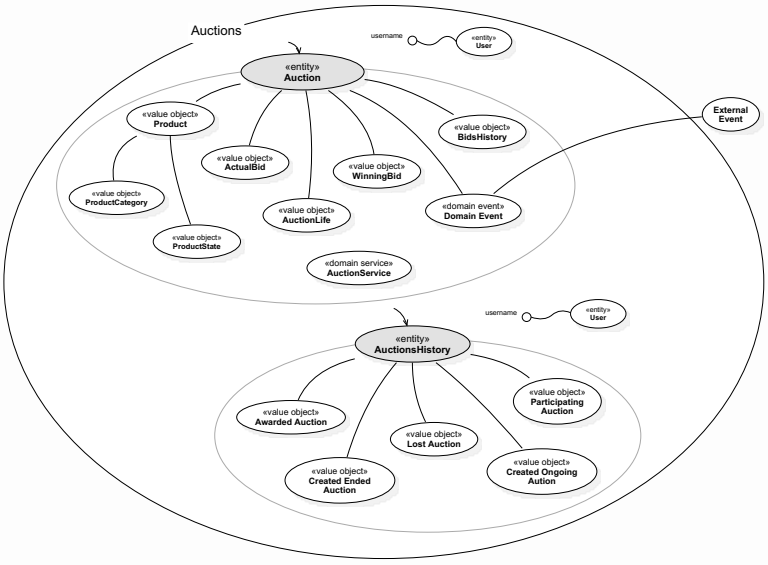


Figura 2.7 - Bounded context delle aste.

Come si può notare dalla figura, all'interno del bounded context sono stati individuati due aggregati: Auction e AuctionsHistory. Il primo definisce tutti i value object relativi a una specifica asta, quali: Product, AuctionLife, ActualBid, WinningBid e BidsHistory. Mentre il secondo definisce il concetto di storico delle aste, in particolare tutte le tipologie di aste che un utente può trovare nel proprio storico.

Oltre ai due aggregati, è stato introdotto anche un **domain service** per le aste. L'idea del domain service è nata dal bisogno di svolgere alcuni controlli che non dipendessero dall'identità dell'asta, ma che fossero stateless. Dunque, anche nell'ottica di mantenere l'entità dell'asta quanto più snella possibile dal punto di vista del comportamento, si è deciso di introdurre un servizio dedicato. In particolare, all'interno del servizio, sono state aggiunte le seguenti due funzioni.

```
public static boolean userCanRaise(String username, Auction auction);
public static boolean isValidBidAmount(Auction auction, Bid bid);
```

Infine, per reagire al verificarsi di determinati eventi all'interno del dominio delle aste, sono stati utilizzati i **domain events**. Essi rappresentano degli eventi che accadono all'interno del dominio, e che possono essere d'interesse per altre entità. Un esempio intuitivo può essere la terminazione di un'asta: quando ciò avviene, il controller delle aste deve avviare una serie di operazioni, come, prima fra tutte, lo scambio di denaro tra il vincitore e il venditore. A tale proposito sono quindi stati creati diversi tipi di eventi a cui (tipicamente il controller) ci si può registrare. La metodologia utilizzata per realizzare quanto appena detto consiste nel consueto publish/subscribe, dove cioè chi è interessato a un certo evento vi si sottoscrive (subscribe), in modo che quando il dominio genera (publish) quell'evento, verrà automaticamente notificato (quindi in modalità push).

2.3.4 AuctionAutomation context

Il presente bounded context modella tutti gli aspetti relativi all'automazione di un'asta. La sua modellazione è mostrata nella figura 2.8.

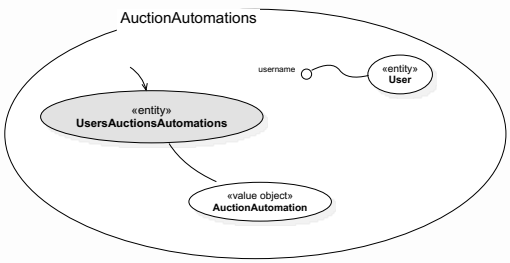


Figura 2.8 - Bounded context delle automazioni delle aste.

In questo caso è presente un unico aggregato: UsersAuctionsAutomations, finalizzato a tenere traccia di tutte le automazioni, attualmente attive, avviate dagli utenti. Ogni automazione è rappresentata dall'entità AuctionAutomation, la quale contiene tutti gli aspetti necessari al fine di governare l'automazione stessa.

2.4 Architettura

In questa sezione si andrà a esplorare l'architettura del sistema dal punto di vista logico. Inizialmente verrà descritta l'architettura di riferimento usata in ciascun bounded context, per poi dedicare maggiore attenzione alla struttura ad attori, e infine verrà mostrata e descritta l'architettura complessiva del sistema.

2.4.1 Clean architecture

L'architettura logica di ogni bounded contexts è stata progettata secondo la visione della **clean architecture**. Essa ha l'obiettivo di rendere il dominio indipendente dall'interfaccia utente, dalla persistenza dati e da eventuali framework applicativi, mantenendo sempre lo stesso *core business* dell'applicazione stabile nel tempo. Di conseguenza, questo consente al team (o all'azienda) di adattarsi alle mutevoli tecnologie. In particolare, con la presenza di librerie front-end sempre più numerose, l'utilizzo di un'architettura pulita e chiara può salvare l'intero team di sviluppo dal possibile sbaglio di accoppiare strettamente la logica aziendale con il livello di presentazione. In questo modo, è possibile rendere il sistema accessibile da parte di qualsiasi tecnologia, eliminando il rischio di dover riscrivere parti significative del sistema stesso. Inoltre, ciò, favorisce anche una migliore manutenibilità del sistema nel tempo.

La figura 2.9 mostra quindi l'architettura logica che si è deciso di adottare per strutturare al meglio ogni bounded context. Il livello delle entità rappresenta il nucleo interno di ogni contesto, ovvero, il dominio. Il core interno viene chiamato dai casi d'uso, che a loro volta vengono chiamati dal controller, il quale ha il compito di coordinare il singolo bounded context ed eventualmente la comunicazione con gli altri. Oltre a ciò, tra il controller e il livello più esterno dell'architettura, è presente anche il livello "adapter" che come suggerisce la parola stessa fa da ponte tra i livelli interni e il livello applicativo, permettendo quindi di passare dagli oggetti del dominio alla corrispondente versione da usare nel livello applicativo (e viceversa). Infine, nel livello più esterno sono presenti tutti i framework e gli strumenti necessari per l'esecuzione vera e propria dell'applicazione.

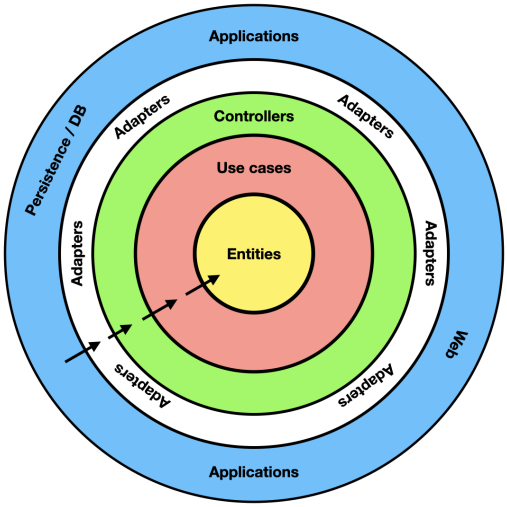


Figura 2.9 - Architettura logica di ogni bounded context.

La peculiarità di questa architettura è il fatto di posizionare il dominio al centro dell'architettura, affinché non dipenda da nessun altro livello. L'intero flusso dati all'interno dell'architettura deve necessariamente essere rivolto verso una unica direzione, ovvero, verso il dominio. Sono i livelli esterni che devono dipendere da quelli interni, non viceversa. Questa è l'unica invariante imposta dal Domain Driven Design a livello architetturale. Il fatto che il dominio non possa dipendere dai livelli architetturali, è una diretta conseguenza del fatto che in fase di analisi non si parla di architetture, quindi non si può pensare che il dominio dipenda da esse.

2.4.2 Modello ad attori

Come visto, oltre all'interazione manuale con un'asta, l'utente può avviare un'automazione che rilancia al posto suo. A tal fine si è deciso di adottare un modello di programmazione ad attori, che quindi merita di essere approfondito dato che presenta un'architettura a sé stante.

I tre componenti dell'architettura ad attori sono: **Master** (M), **Router** (R) e **UserActor** (A), raffigurati in figura 2.10 con le rispettive lettere. Vediamo di seguito più in dettaglio il funzionamento di ciascuno di questi, partendo dalle foglie per risalire fino alla radice.

L o *UserActor* rappresenta l'automazione vera e propria di un utente ed è riferita a una singola asta. Esso incapsula il comportamento desiderato dell'automazione, rilanciando quindi l'offerta dopo il tempo desiderato, incrementando il valore della quota desiderata. Inoltre, riceve dal router messaggi che rappresentano l'avvenuto rilancio di una bid per quell'asta e utilizza le tradizionali API REST per effettuare nuovi rilanci.

Il *Router* riceve dal master l'ordine di creare un nuovo attore (UserActor), a cui associare una particolare automazione. Si interfaccia con il server (inteso qui in generale come l'insieme di più microservizi) attraverso le **socket** per ricevere le bid, da inoltrare a tutti e i soli attori utente (A) interessati.

Infine, il *Master* ha una visione globale di tutti i router e del loro attuale carico di lavoro (i.e. *mpm* = messaggi per minuto), potendo così decidere quando è il caso di creare nuovi router e quando interromperli, secondo opportune politiche, utilizzate anche per decidere a quali router assegnare una nuova automazione. Anche il Master si interfaccia con il server mediante socket, dalla quale riceve l'ordine di creazione di una nuova automazione.

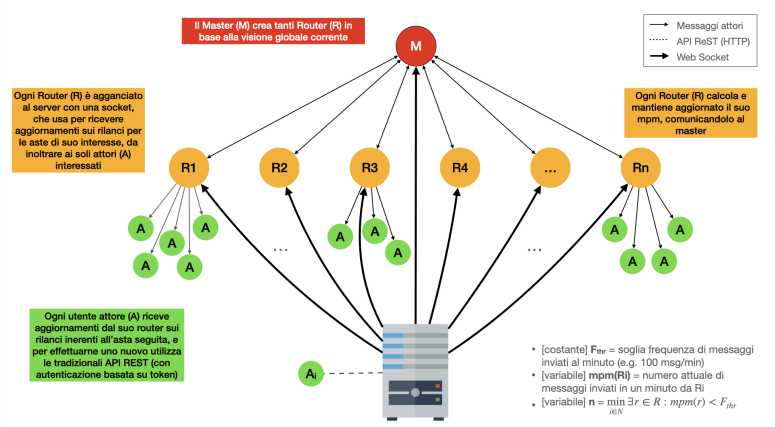


Figura 2.10 - Architettura ad attori.

2.4.3 Architettura complessiva

Il software realizzato, nel suo insieme, si basa su un'**architettura a micro-servizi**. Nello specifico, si è deciso di realizzare un micro-servizio per ogni bounded context e uno specifico ad attori per implementare le automazioni. La motivazione principale che ci ha spinto verso questa scelta è stata la possibilità di rendere il sistema modulabile e scalabile, anche in ottica di eventuali sviluppi futuri.

Nello specifico, è stato adottato il pattern architetturale **API Gateway**. Questo pattern consente di avere un unico punto di accesso al sistema, instradando le richieste dai client verso i giusti microservizi, nascosti all'utente. Infatti, come si può notare dalla figura 2.11, il gateway API si trova esattamente tra l'app client e i microservizi. Anche il gateway è un servizio a tutti gli effetti, ma, a differenza degli altri micro-servizi, è formato esclusivamente dal livello "applicazione" perché non definisce nuovi concetti a livello di business (dominio).

Inoltre, nel gateway viene gestita la sicurezza del sistema, dal momento che

questo è l'unico punto di accesso per gli utenti. Per ulteriori informazioni relativi alla sicurezza del sistema, si rimanda ai diagrammi di sequenza e alla sezione 4.4 dedicata ai dettagli implementativi.

Infine, è stato aggiunto un ulteriore servizio con lo scopo di fornire un semplice prototipo dell'interfaccia utente per dimostrare il funzionamento dell'intero sistema. Questo servizio è mostrato in figura 2.11 in basso a sinistra. Anch'esso, come il gateway, è formato esclusivamente dal livello "applicazione" perché funge esclusivamente da front-end per gestire l'interazione con l'utente.

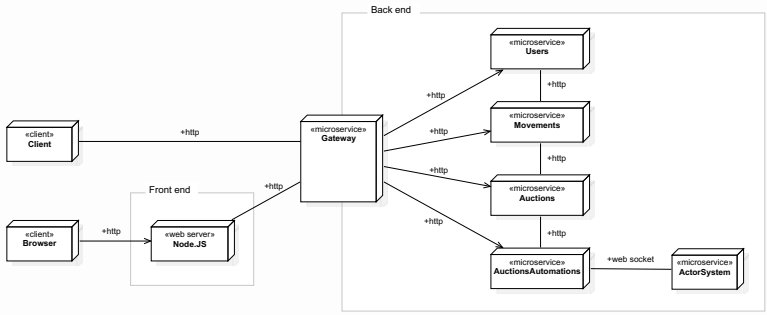


Figura 2.11 - Architettura finale del sistema.

2.5 Struttura

Di seguito analizziamo la modellazione delle diverse entità che compongono il dominio, dal punto di vista strutturale. Per farlo ci appoggeremo sui **diagrammi delle classi**, uno per ogni bounded context. Infine, mediante un **diagramma dei componenti**, mostreremo anche come sono state modularizzate le diverse entità.

Prima di addentrarci nell'analisi di ciascun diagramma delle classi, si precisa che questi sono stati svolti prima della scrittura del codice e senza pensare al linguaggio di programmazione specifico, pertanto nel diagramma non compariranno mai i famosi "getter" propri di Java (se non dove servono per realizzare template method), anche se poi nel codice saranno utilizzati questi per implementare i vari campi specificati nei diagrammi.

2.5.1 Utenti

Come visto, il bounded context degli utenti si compone di due entità: User e UserAuthentication, che infatti sono il cuore anche del diagramma delle classi mostrato in figura 2.12. Non si segnalano aspetti particolari, se non il fatto che come relazione di part-of si è scelto di utilizzare un'aggregazione per impedire che possano esistere utenti con informazioni mancanti.

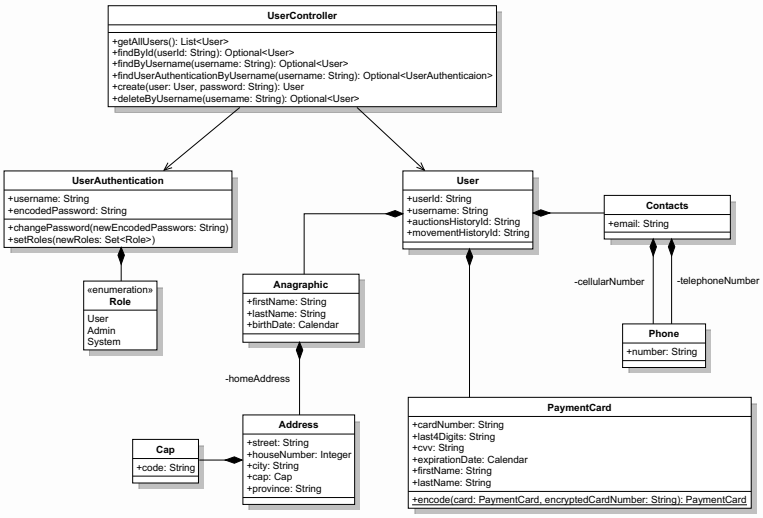


Figura 2.12 - Diagramma delle classi riferito alla modellazione degli utenti.

2.5.2 Movimenti

Dal momento che, come ricavato dalla fase di knowledge crunching, esistono vari tipi di movimenti che presentano molti aspetti comuni, si è pensato di creare una gerarchia di tutti i possibili movimenti, astraendo i concetti simili. Anche in questo caso, il diagramma in figura 2.13 è autoesplicativo. Si segnala solamente che, come evidenziano le relazioni di aggregazione, ogni MovementHistory contiene necessariamente il budget corrente dell'utente, le transazioni e i congelamenti (qualora ce ne siano).

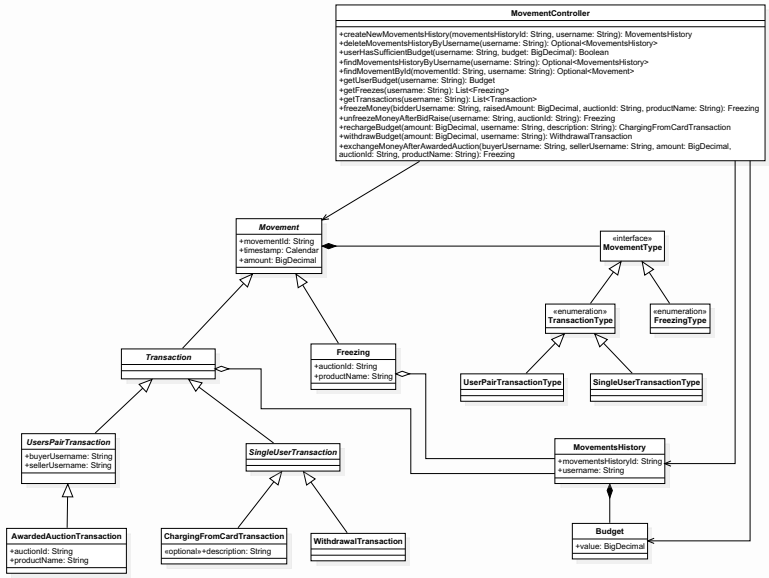


Figura 2.13 - Diagramma delle classi riferito alla modellazione dei movimenti degli utenti.

2.5.3 Aste

Il bounded context delle aste è sicuramente il cuore dell'intero sistema, nonché quello più complesso. Si è cercato di adottare una modellazione che sfruttasse ancora una volta le potenzialità offerte dalla programmazione a oggetti (quindi in particolare l'ereditarietà), astraendo quindi i concetti comuni, per poi specializzare le varie entità.

Come emerso in fase di analisi, si vuole avere, oltre al concetto di asta vera e propria, anche delle sintesi, contenenti solo gli aspetti strettamente necessari e di riepilogo. Pertanto, come mostrato in figura 2.14, è stata creata una classe astratta alla radice, contente solo l'id dell'asta, il prodotto e il venditore, in modo da modellare le classi sottostanti con gli altri aspetti strettamente necessari. Scendendo troviamo quindi una specializzazione per le aste in corso e poi due tipi di sommari: uno per le aste concluse e uno per le aste in corso.

Le aste effettivamente in corso vengono arricchite col concetto di vita dell'asta (AuctionLife), che gestisce tutte gli aspetti legati al tempo, oltre anche alle eventuali offerte. Infine, all'oggetto concreto Auction (l'asta vera e propria, completa), viene aggiunto un timer, in modo da tenere traccia della sua terminazione.

D'altro canto, i vari sommari delle aste non necessitano di mantenere memorizzate tutte le informazioni legate alla vita dell'asta, motivo per cui queste classi presentano meno campi e relazioni. Nelle sintesi delle aste terminate, in particolare in quelle create da un utente e in quelle aggiudicate, vengono aggiunti i contatti del venditore e del vincitore (nel caso delle aste create dall'utente queste relazioni sono opzionali perché non è detto che qualcuno si sia aggiudicato quella particolare asta conclusa). Tutti i sommari vengono poi racchiusi dentro la cronologia delle aste (AuctionsHistory), riferita a ciascun utente.

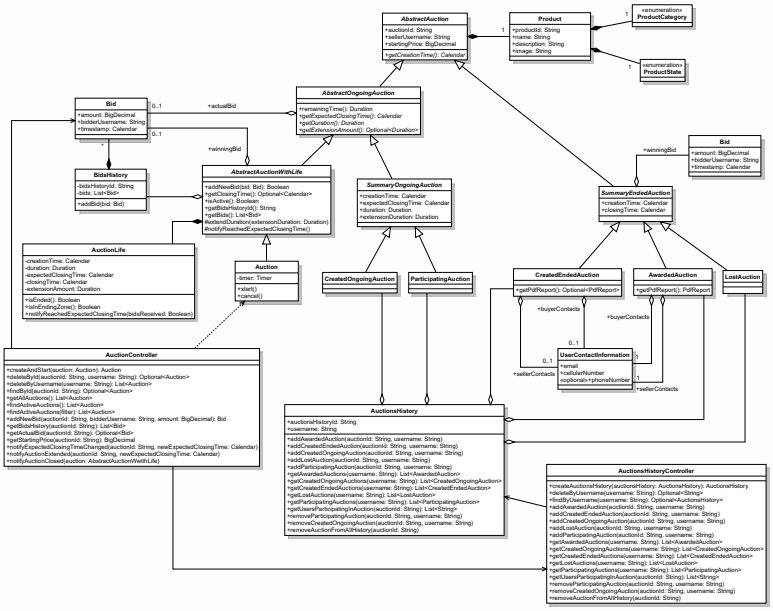


Figura 2.14 - Diagramma delle classi riferito alla modellazione delle aste.

2.5.4 Automazioni

Troviamo infine le automazioni, che come mostrato dal diagramma delle classi di figura 2.15 sono modellate semplicemente da due classi: UsersAuctionsAutomations contiene tutte le automazioni attualmente attive create da un utente e AuctionAutomation rappresenta la singola automazione, con le sue proprietà.

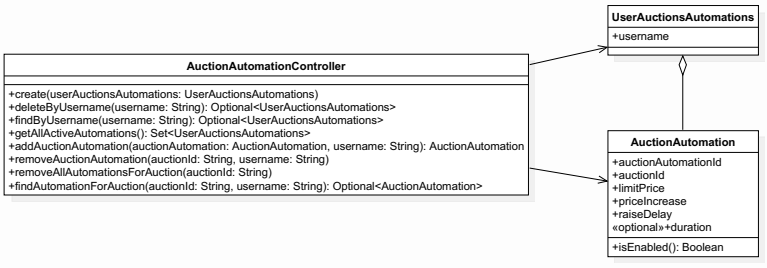


Figura 2.15 - Diagramma delle classi riferito alla modellazione delle automazioni.

2.5.5 Modularizzazione

Tutti i diagrammi delle classi mostrati fin qua corrispondono alla modellazione del dominio dell'intero sistema. Ciascun bounded context è poi organizzato secondo la visione della *clean architecture*, pertanto tale struttura deve essere mantenuta anche a livello di codice. A tal fine, si è scelto di creare un modulo per ciascun livello dell'architettura, al cui interno saranno poi presenti le classi

necessarie al suo corretto funzionamento. Per capire meglio quali siano i diversi moduli e quali le loro dipendenze, ci si avvale del **diagramma dei package** di figura 2.16. Questa organizzazione è stata applicata per ogni bounded context.

Il modulo **Domain** contiene quindi le classi mostrate in precedenza; **UseCase** i casi d'uso individuati in fase di knowledge crunching, i quali dipendono dal dominio; il **Controller** orchestra i diversi casi d'uso, al fine di raggiungere l'obiettivo desiderato; infine, più esternamente troviamo il livello applicativo e di persistenza, ovvero **SpringApplication** e **MongoRepository**, che si appoggiano sul modulo **Adapter** per la conversione da/verso il dominio. SpringApplication riceve le richieste HTTP e le inoltra al controller, motivo per cui dipende da quest'ultimo, mentre MongoRepository implementa i metodi dei repository definiti nei casi d'uso.

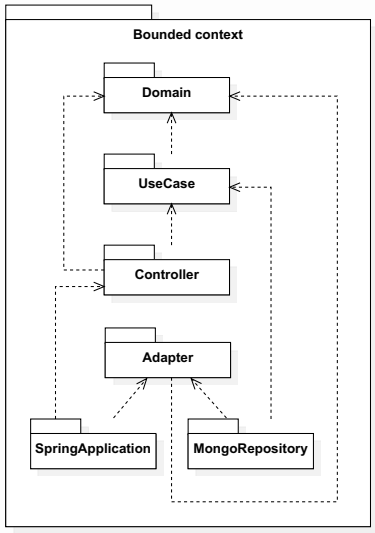


Figura 2.16 - Diagramma dei package rappresentante l'organizzazione adottata in ciascun bounded context.

2.6 Comportamento

Passiamo ora a considerare il comportamento delle diverse entità del sistema. Le entità cardine del progetto saranno naturalmente gli utenti che interagiscono con le aste, e le automazioni. Saranno dunque queste ultime due le entità di maggior rilevanza e dotate di un comportamento significativo e degno di essere approfondito. Utilizziamo pertanto i **diagrammi di attività** di UML, facilmente interpretabili anche dagli esperti del dominio, per mostrare in modo chiaro il comportamento del sistema.

2.6.1 Aste

A seguito di maggiori incontri, si è potuto formalizzare meglio il concetto di vita di un'asta, quindi tutto il processo che va dalla creazione dell'asta fino alla sua concreta terminazione ed eventuale proclamazione del vincitore. Analizziamo dunque il diagramma di attività di figura 2.17.

La prima attività da svolgere è naturalmente quella di creazione dell'asta, effettuata dal venditore stesso (che ricordiamo può essere qualsiasi utente). Una volta creata, l'asta ha immediatamente inizio e qualsiasi altro utente (o un'automazione) può effettuare un rilancio. Non appena un nuovo utente prova a rilanciare si controlla che il rilancio sia valido, ovvero che l'utente abbia effettivamente budget sufficiente e che non sia già lui stesso l'ultimo offerente (o il venditore). Se almeno una di queste condizioni non è verificata, il rilancio viene immediatamente rifiutato; in caso contrario si può procedere con le attività seguenti, che prevedono lo scongelamento del denaro del vecchio offerente e il congelamento di quello del nuovo offerente.

In aggiunta all'aspetto monetario, va tenuta in considerazione anche l'eventuale posticipazione della fine dell'asta, qualora il rilancio sia avvenuto in *ending zone*. Concluso quest'ultimo controllo, tutta la procedura associata all'evento "rilancio" può a tutti gli effetti dirsi conclusa. Essa si ripeterà allo stesso modo per ogni nuova offerta, fino a che il sistema non notificherà l'evento di timeout dell'asta.

Al verificarsi del timeout, il sistema controlla se l'asta ha un vincitore (i.e. l'ultimo utente che ha rilanciato): qualora ci sia, l'asta viene effettivamente dichiarata conclusa e si effettua lo scambio di denaro tra il vincitore e il venditore (avendo cura di scongelare prima il relativo budget del vincitore). In caso contrario, ovvero se nessuno ha rilanciato, si controlla se il venditore aveva predisposto un'eventuale estensione dell'asta: in caso affermativo, l'asta viene estesa del periodo di tempo desiderato dall'utente, altrimenti viene dichiarata conclusa, senza alcun vincitore. Ricordiamo che l'asta può essere estesa al più una sola volta, quindi dopo l'eventuale prima estensione viene posto a false il flag di estensione, in modo da evitare che venga estesa nuovamente in futuro.

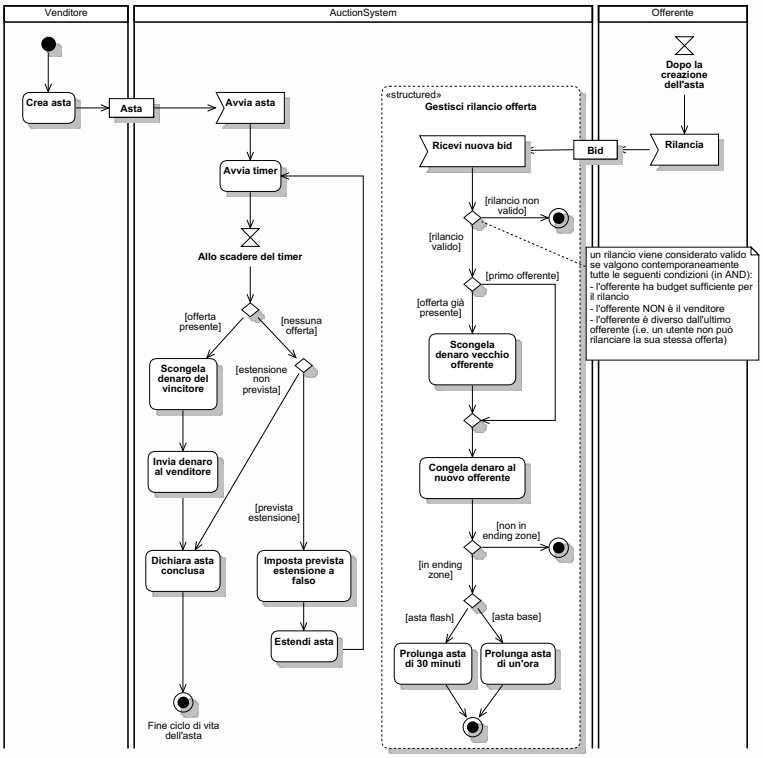


Figura 2.17 - Diagramma di attività riferito al ciclo di vita di un'asta. Da osservare che l'*Offerente* può essere sia l'utente in prima persona, ma anche un'automazione avviata da un utente per una determinata asta.

2.6.2 Automazioni aste

Passiamo a questo punto ad analizzare più in dettaglio il funzionamento di un'automazione, mediante il diagramma di attività di figura 2.18.

Come visto, un'automazione può essere creata da un utente, pertanto l'azione da cui tutto ha inizio è la creazione dell'automazione per una determinata asta, con i parametri specificati dall'utente. A questo punto il sistema avvia effettivamente l'automazione, la quale tenterà subito di fare il primo rilancio, per poi restare in attesa di nuovi rilanci per l'asta specificata. Analizziamo quindi le due macro attività "Rilancia bid" e "Gestisci rilancio bid".

La prima, come suggerisce il nome, comprende tutte le azioni volte a rilanciare una nuova bid (offerta). Dopo aver ottenuto il prezzo attuale dell'asta, ci si accerta che l'eventuale ultimo offerente non sia l'utente stesso (i.e. colui che ha avviato l'automazione), e solo in tal caso si procede a calcolare il valore della nuova offerta secondo l'incremento di prezzo desiderato dall'utente.

Se questo rientra ancora nei limiti, si procede a rilanciare l'offerta; viceversa l'automazione può terminare. In caso di errori durante il rilancio (tipicamente per via di budget insufficiente dell'utente), l'automazione termina, altrimenti si

continua a restare in attesa di nuove bid *di altri utenti*, il cui gestore dell'evento è rappresentato dalla macro attività "Gestisci rilancio bid".

Quest'ultima attività viene invocata non appena viene ricevuto un nuovo rilancio per conto di altri utenti, e controlla subito se il valore della bid da rilanciare eccede il limite di prezzo impostato dall'utente; il tal caso l'automazione può terminare. In caso contrario, si attende il tempo desiderato prima di effettuare il rilancio, tornando così direttamente nella prima attività (da notare che questa poi ricontrollerà il valore dell'offerta attuale, in quanto durante l'attesa potrebbe essere cambiata). Gli eventi (i.e. rilanci) che possono verificarsi durante il periodo di attesa vengono ignorati (i.e. non comportano l'esecuzione del gestore dell'evento), visto che al momento dell'effettivo rilancio si ricontrollerà il valore dell'ultima offerta.

Come ultima nota, si pone ora l'attenzione sull'interrupt presente in fondo: esso scatta nel momento in cui viene raggiunta l'ora di fine automazione, ponendo fine all'automazione. Ricordiamo infatti che tra i vari parametri dell'automazione, è possibile specificare, opzionalmente, la data e l'ora di fine, in modo che l'automazione termini in ogni caso raggiunto quel momento.

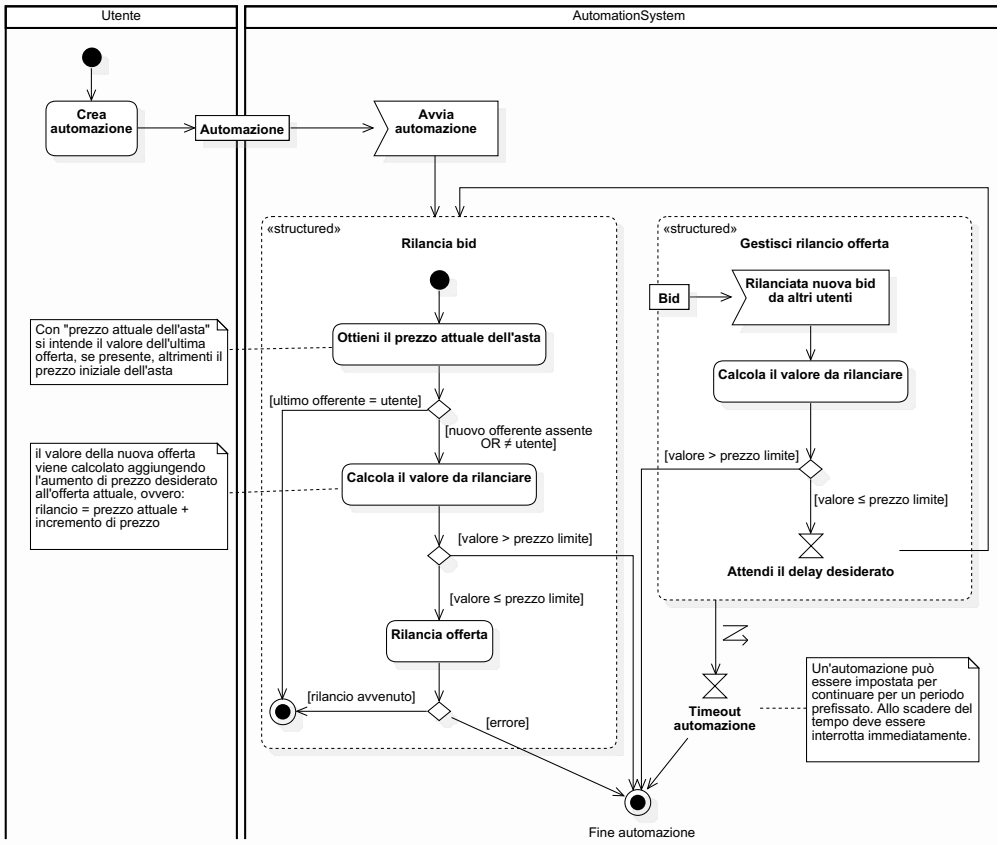


Figura 2.18 - Diagramma di attività riferito al ciclo di vita di un'automazione per un'asta.

2.7 Interazione

In questa sezione ci si concentrerà su come le diverse entità del sistema interagiscono tra loro, a partire dalle azioni intraprese dagli utenti. Per rappresentare le diverse interazioni verranno utilizzati i **diagrammi di sequenza** di UML, il cui scopo è proprio mostrare come comunicano le diverse entità del sistema e quali messaggi si scambiano a seconda delle circostanze.

2.7.1 Autenticazione e autorizzazione

La sicurezza è implementata nel Gateway, che è l'unica parte del sistema accessibile dagli utenti, ed è concretizzata nelle due fasi di autenticazione e autorizzazione. Partiamo dunque ad analizzare la prima, prendendo come riferimento il diagramma di sequenza di figura 2.19.

La richiesta che viene effettuata dall'utente sarà dunque quella di un tradizionale login, con username e password. Il Gateway andrà successivamente a richiedere a UserSystem le informazioni di **autenticazione** dell'utente (i.e. password criptata e ruoli). Se lo username esiste, il Gateway verifica che la password fornita dall'utente, opportunamente codificata, sia uguale alla password fornita da UserSystem. In tale caso vengono generati i token di accesso e di refresh da restituire all'utente, che potrà poi usarli per le successive richieste in cui è prevista l'autenticazione. Se invece la password non equivale a quella restituita da UserSystem, oppure non è stato trovato nessun utente con lo username specificato, la richiesta dell'utente termina con un errore di autenticazione.

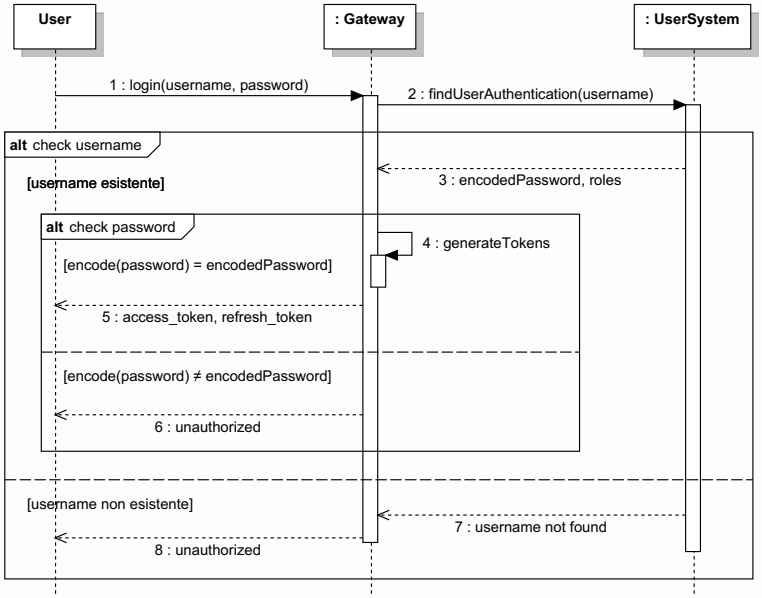


Figura 2.19 - Diagramma di sequenza riferito al login di un utente (i.e. autenticazione).

Passiamo a questo punto alla fase di **autorizzazione**, che consiste nel verificare se chi effettua una determinata richiesta ha i permessi sufficienti. Più precisamente, questo si traduce nel verificare che la richiesta contenga un token di accesso valido al fine di consentire all'utente l'accesso alla risorsa desiderata.

Consideriamo quindi il diagramma di figura 2.20. Come si nota, questa volta tutte le operazioni di verifica vengono effettuate direttamente dal Gateway, dato che tutte le informazioni necessarie sono contenute direttamente nel token. Non appena riceve la richiesta, il Gateway decodifica quindi il token, ottenendo così lo username e i ruoli dell'utente associato al token. Se il token è valido (i.e. se la decodifica è avvenuta correttamente e il token non è scaduto) e i ruoli sono sufficienti per accedere alla risorsa desiderata, il Gateway continua a processare la richiesta per conto dello username appena ottenuto. In caso contrario invece la richiesta termina immediatamente, con un errore di autorizzazione.

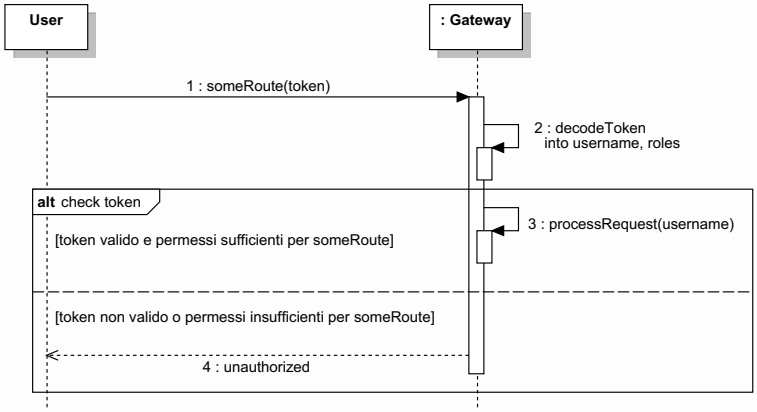


Figura 2.20 - Diagramma di sequenza riferito alla verifica del token (i.e. autorizzazione).

2.7.2 Aste

Iniziamo con l'analizzare il comportamento del sistema non appena viene ricevuta una richiesta di creazione asta. Prima di addentrarci nel funzionamento vero e proprio, elenchiamo brevemente quali sono e cosa rappresentano le diverse linee di vita (i.e. entità che interagiscono tra loro) che sono presenti nel diagramma di figura 2.21.

Le prime due, **Bidder** e **Seller**, rappresentano, rispettivamente, un generico utente "bidder", ovvero un utente che rilancia un'offerta per un'asta, e un generico utente "seller", cioè colui che vuole mettere all'asta un prodotto.

Si ricorda che ogni utente può essere sia venditore che acquirente, ovvero può creare aste ma può anche partecipare ad altrettante. Nel diagramma viene descritto ciò che avviene per un'asta prefissata: essa avrà un preciso venditore e tanti possibili offerenti che cercheranno di aggiudicarsi l'asta. Questa precisazione spiega il perché Bidder è considerato come classe, mentre Seller come istanza (è preceduto dai due punti).

Proseguendo troviamo altre linee di vita istanziate, ovvero:

- **Gateway**, che costituisce la porta d'ingresso di tutte le richieste effettuabili dagli utenti;
- **AuctionSystem**, la parte di sistema responsabile della gestione delle aste;
- **AuctionHistorySystem**, la parte di sistema responsabile della gestione della cronologia delle aste di ciascun utente;
- **MovementSystem**, la parte di sistema responsabile della gestione dei movimenti di ciascun utente;
- **Auction**, che corrisponde ad una specifica asta;
- **BidsSocket**, la socket su cui vengono notificati i rilanci delle bid;

- **AuctionSocket**, la socket su cui vengono notificati gli eventi relativi alle aste;
- **AutomationSocket**, la socket su cui vengono notificati gli eventi relativi alle automazioni.

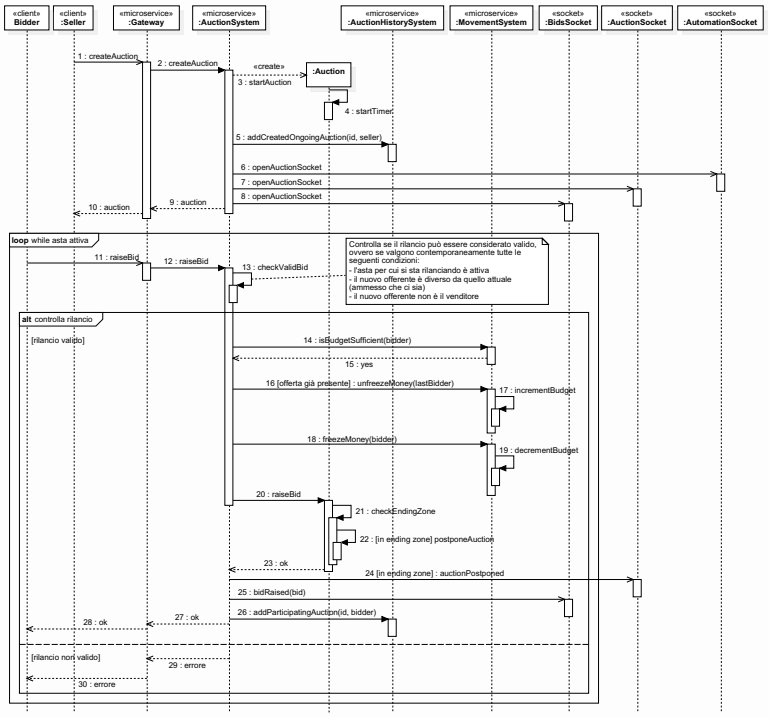


Figura 2.21 - Diagramma di sequenza riferito alla creazione e svolgimento di un'asta.

Fatte queste precisazioni, possiamo finalmente passare ad analizzare il diagramma di figura 2.21. Come visto in precedenza, ogni richiesta che l'utente vuole effettuare deve essere inviata al gateway, il quale poi la inoltra al microservizio corrispondente. Questo è ben visibile anche nel diagramma, dove ad esempio la prima richiesta che l'utente effettua per creare l'asta, è rivolta al gateway, il quale poi la inoltra al microservizio delle aste. Quest'ultimo creerà l'asta secondo i parametri specificati dall'utente e la aggiungerà nella cronologia di aste di quell'utente, nella categoria "aste create in corso", effettuando l'apposita richiesta ad AuctionHistorySystem. Al termine di tutto, la risposta di conferma (contenente l'asta appena creata) viene girata all'utente.

Da questo momento in poi e finché l'asta è attiva, gli utenti potranno rilanciare a piacimento, a patto di rispettare gli ormai vincoli noti. Nel caso di rilancio valido, se l'utente ha budget sufficiente, si procede a scongelare il budget del vecchio offerente (ammesso che ci sia) per poi procedere a congelare quello dell'offerente attuale. Da notare che ogni operazione di congelamento (o scongelamento) comporta anche l'incremento (o decremento) del budget. Fatte

queste operazioni preliminari, si può finalmente notificare il rilancio all'oggetto asta, il quale controlla se il rilancio sta avvenendo in *ending zone* e in tal caso posticipa la fine dell'asta, secondo le modalità discusse in precedenza. Come ultima operazione, si aggiunge l'asta attuale alla cronologia delle aste a cui l'utente (che ha appena rilanciato) sta partecipando. Se l'utente in futuro dovesse effettuare un nuovo rilancio, questo messaggio non avrà alcun effetto (i.e. il servizio che gestisce lo storico delle aste riconosce che quell'utente sta già partecipando all'asta specificata e quindi non esegue alcuna operazione). In caso l'offerta rilanciata non sia valida non verrà eseguita nessuna delle operazioni appena descritte e il gateway inoltrerà il messaggio di errore all'utente.

Passiamo a questo punto ad analizzare la fase conclusiva dell'asta, ovvero quando viene raggiunto il termine previsto. A seconda della circostanza e della configurazione dell'asta, possono verificarsi diverse situazioni. Prendiamo come riferimento la figura 2.22, raffigurante il diagramma di sequenza che va inteso come il continuo del precedente, con gli oggetti precedentemente istanziati. Le operazioni illustrate entrano in funzione non appena scatta l'evento di timer scaduto dell'asta: a questo punto, l'oggetto asta effettuerà operazioni differenti a seconda se sia presente o meno un'offerta.

Concentriamoci sul caso più semplice, ovvero quello senza offerte presenti. In questo caso si controlla se il venditore aveva impostato un'estensione dell'asta: in caso affermativo, l'asta semplicemente viene estesa, re-impostando il timer per la nuova fine attesa e notificando l'avvenuta estensione ad AuctionService. Altrimenti si notifica che l'asta è conclusa: a questo punto il pallino del gioco passa nelle mani di AuctionService che andrà a rimuovere l'asta dalle aste create in corso del venditore, per aggiungerla tra quelle create e concluse. Fatto ciò, l'oggetto asta può definitivamente essere distrutto.

Passiamo ora all'altro caso, ovvero quello in cui sia presente almeno un'offerta per l'asta. Durante la gestione di evento di timer scaduto, l'oggetto asta imposta la winning bid (i.e. l'offerta vincente), che corrisponde proprio con l'ultima offerta ricevuta. Fatto ciò, si può notificare ad AuctionService la conclusione dell'asta, che andrà a effettuare le operazioni conclusive. Questa volta però verranno effettuati diversi passaggi, al fine di aggiornare le history di tutti i partecipanti e di scambiare il denaro tra il vincitore e il venditore. Al termine di tutto l'oggetto asta può essere distrutto.

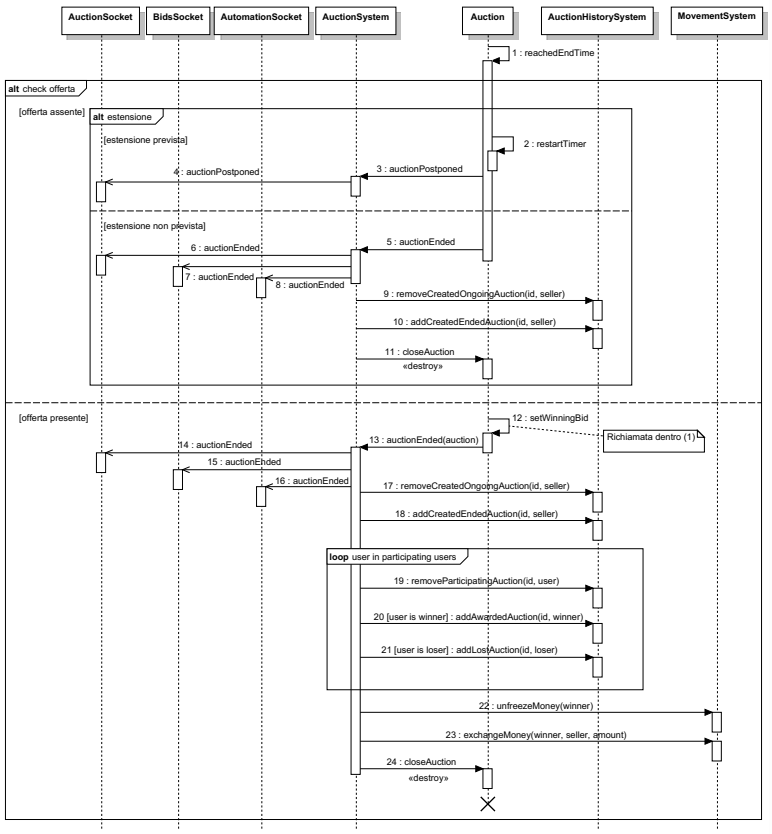


Figura 2.22 - Diagramma di sequenza riferito alla terminazione di un'asta.

2.7.3 Automazioni aste

Come mostrato nella sezione di architettura, il microservizio delle automazioni si compone di tre entità fondamentali, **Master**, **Router** e **UserActor**. Analizziamo quindi, mediante il diagramma di sequenza di figura 2.23, in che modo interagiscono tra loro e come si interfacciano col resto dell'applicazione.

Appena le prime entità Master e Router vengono create, si connettono alle apposite socket, per ricevere in modalità push le notifiche inerenti agli eventi d'interesse, in particolare:

- Master si registra sulla socket AutomationSocket per ricevere le richieste, da parte degli utenti, di creazione di una nuova automazione;
- Router si registra sulla socket BidsSocket e AutomationSocket, per ricevere, dalla prima, notifiche inerenti ai rilanci per i gruppi di aste d'interesse, e, dalla seconda, notifiche inerenti alla terminazione di una specifica automazione o di tutte le automazioni associate ad una determinata asta.

Fatte queste operazioni preliminari, il microservizio delle automazioni è pronto a soddisfare le richieste degli utenti. Infatti, non appena un utente richiede al

Gateway di creare una nuova automazione, questa viene inoltrata dapprima al microservizio delle automazioni, per poi arrivare all'attore (Akka) Master. Questo, seleziona dalla sua lista di router, quello che dovrà diventare responsabile della nuova automazione (la modalità di selezione verrà descritta nella sezione 4.3). Una volta selezionato il router, gli invia un messaggio contente l'automazione da creare. A questo punto il router crea un nuovo attore figlio, UserActor, incaricato di eseguire l'automazione dell'utente. Dopo questa operazione l'automazione è a tutti gli effetti attiva ed effettuerà rilanci automatici secondo le modalità impostate dall'utente.

Passiamo quindi ora ad analizzare cosa accade quando un altro utente effettua un rilancio per l'asta per la quale l'utente owner ha attivato l'automazione. Il microservizio delle aste invia sulla BidsSocket un messaggio contenente la bid e il riferimento all'asta a cui il rilancio è riferito. In questo modo, il router precedente riceve l'evento mediante la socket. Sfruttando la sua tabella di routing, seleziona tutte le automazioni attive per quella specifica asta, e invia agli UserActor corrispondenti un messaggio di offerta rilanciata (nel diagramma consideriamo solo l'automazione dell'utente owner, per semplicità). Quando lo UserActor riceve tale messaggio effettua i controlli descritti nella sezione del comportamento, per accertarsi se sia possibile effettuare un rilancio oppure no.

Concentriamoci ora sulla parte un po' più articolata del diagramma, ovvero il caso in cui non sia più possibile rilanciare (i blocchi contrassegnati dalla guardia "*no*", che sono equivalenti). La prima operazione che si effettua è l'invio del messaggio ad AutomationSystem per comunicargli che l'automazione dell'utente non può continuare, dopodiché comunica anche al router padre che la sua esecuzione è terminata. Fatto ciò, lo UserActor si autodistrugge. A questo punto però segnaliamo un aspetto interessante: come si evince, AutomationSystem invia sulla AutomationSocket il messaggio di terminazione, che quindi giungerà al router. Pertanto, il router padre dello UserActor appena terminato riceverà due volte lo stesso messaggio, da due sorgenti diverse. Ciò non crea nessun problema: il secondo che verrà ricevuto (a prescindere da quale sia), non comporterà l'esecuzione di nuove operazioni.

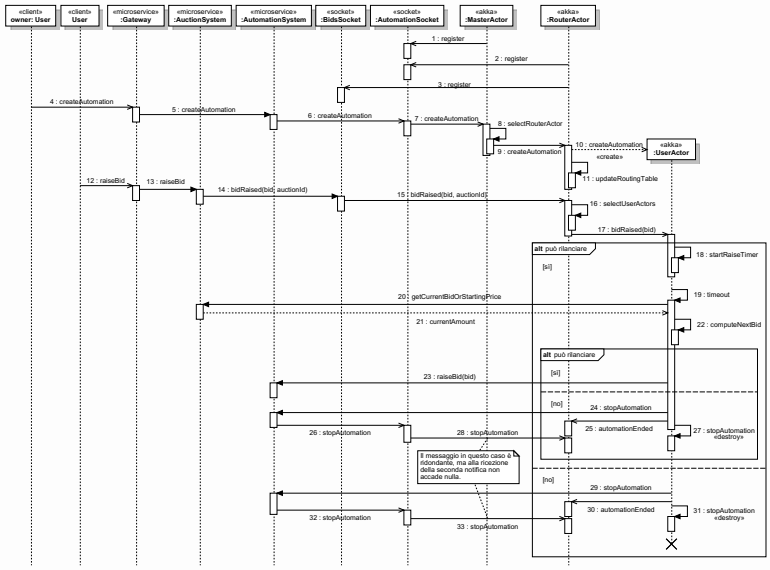


Figura 2.23 - Diagramma di sequenza raffigurante l'esecuzione di una automazione per un'asta.

Analizzati i casi in cui l'automazione termina in automatico a fronte di un rilancio, quindi a causa del superamento del prezzo limite, andiamo a considerare ora anche tutti gli altri casi che possono portare alla terminazione dell'automazione (figura 2.24).

Il primo è banalmente quello del raggiungimento della data/ora di fine impostata dall'utente (opzionale). Oppure, può essere l'utente stesso a richiedere la terminazione prematura dell'automazione, caso in cui il router riceverà il messaggio di terminazione dalla AutomationSocket, da inoltrare poi allo UserActor corrispondente. Infine, un'automazione termina necessariamente quando l'asta a essa associata termina.

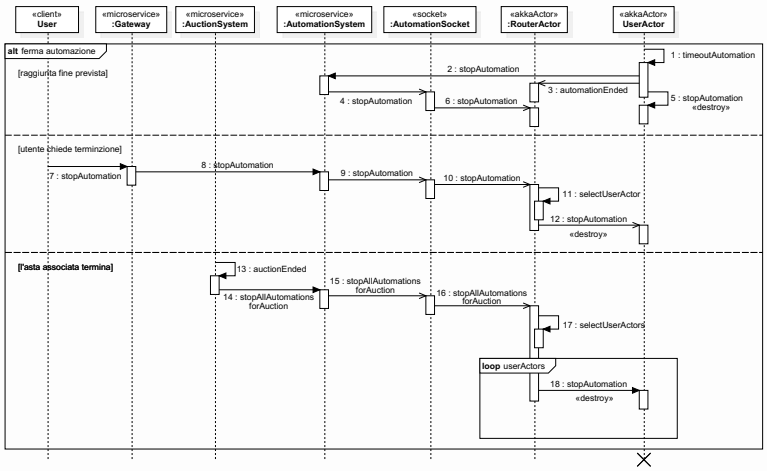


Figura 2.24 - Diagramma di sequenza raffigurante le modalità di conclusione di un'automazione.

2.7.4 Attori

Infine, prima di concludere l'intera sezione, diamo uno sguardo anche a come si coordinano le tre tipologie di attori, focalizzandoci quindi sugli aspetti più secondari, ma comunque importanti per il corretto funzionamento del sistema ad attori.

Come visto nella sezione dedicata all'architettura del modello ad attori, il master è colui che ha una visione completa sullo stato attuale dei diversi router, in particolare conosce il numero di messaggi inviati da ciascun router nell'ultimo minuto (mpm), potendo così sfruttare tale conoscenza all'occorrenza. Infatti, come mostrato dal diagramma di figura 2.25, non appena il master riceve il messaggio di creazione di una nuova automazione, deve selezionare un router che dovrà portare avanti la creazione e gestire l'attore figlio che impersonerà l'automazione vera e propria.

Per eseguire questa selezione si procede come primo tentativo a selezionare il router con il mpm più basso, tra i router attualmente attivi. Se non c'è nessun router attivo o il router selezionato presenta un mpm sopra la soglia, si procede nello stesso modo, però controllando la lista dei router messi in fase di cancellazione. Se anche in questo caso non viene trovato un router valido, si procede alla creazione di un nuovo router, che sarà dunque quello prescelto.

Quindi, a questo punto il master ha individuato il router e può inviargli il messaggio di creazione dell'automazione. Il router provvede ad aggiornare la propria tabella di routing inserendo la rotta per il nuovo UserActor di cui ha appena ordinato la creazione. Dopo di ciò, il flusso di operazioni da eseguire a fronte della richiesta di una nuova automazione può dichiararsi concluso e il sistema può correttamente funzionare come descritto nel precedente diagramma.

Passiamo ora ad analizzare cosa succede sempre "dietro le quinte" quando un attore termina. Oltre agli aspetti evidenziati in precedenza, è anche necessario rimuovere, dal router padre, la rotta per l'attore che è appena terminato e, se a questo punto la routing table è vuota, l'attore può terminare, in quanto non ha più figli in esecuzione. Prima di terminare, ha cura di comunicare al master la sua avvenuta terminazione, in modo tale che questo possa aggiornare la sua visione globale, rimuovendo definitivamente il router dalle sue liste.

Oltre a queste operazioni che avvengono a fronte di eventi esterni, ci sono due ulteriori attività in costante esecuzione in background, una lato router e una lato master. Partiamo dal primo: ciascun router, ogni minuto invia un messaggio al master, contente il valore corrente di mpm, in modo che

quest'ultimo possa sempre avere l'ultimo mpm di ogni router e intraprendere le giuste decisioni quando necessario. Lato master, invece, ogni mezz'ora si controlla se è il caso di liberare risorse, in particolare, se c'è qualche router sottoutilizzato, viene selezionato quello meno carico di lavoro e lo si sposta nella lista dei router in fase di cancellazione, in modo che da quel momento in poi non gli vengano assegnate nuove automazioni (ricordiamo che comunque quando poi servirà un nuovo router, se nessuno di quelli attualmente attivi è disponibile, il master potrà lo stesso selezionarne uno tra quelli in lista di cancellazione, come mostrato nella parte alta del diagramma).

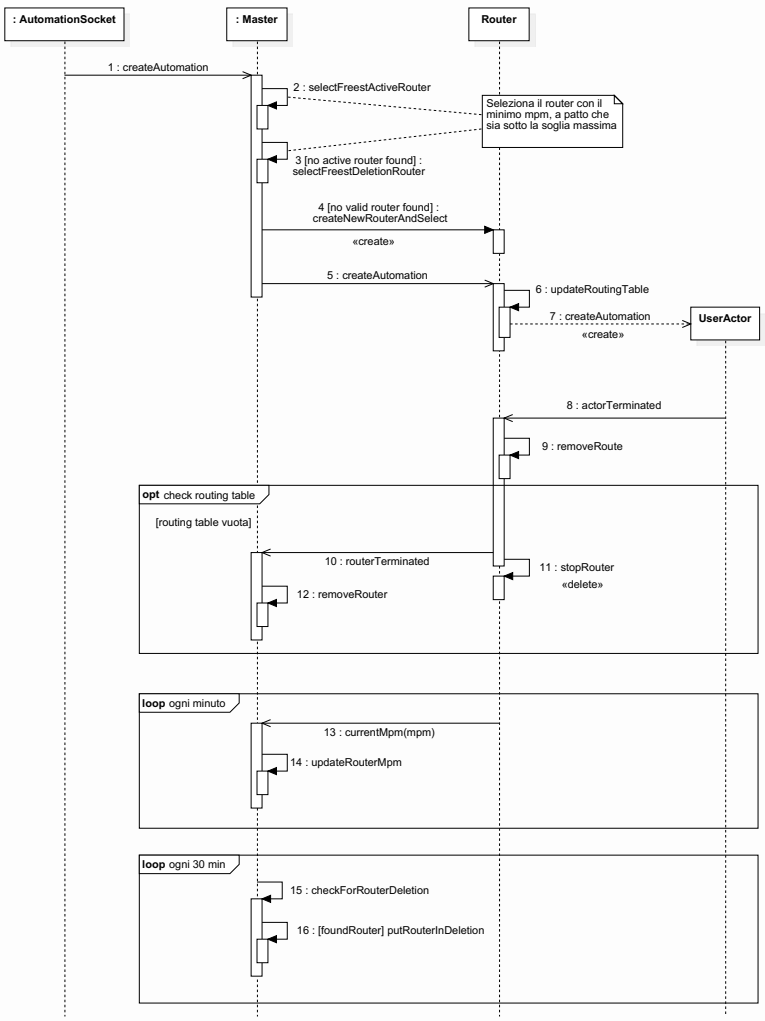


Figura 2.25 - Diagramma di sequenza raffigurante la gestione "dietro le quinte" degli attori.

3 DevOps

In questa sezione si analizzano le tecniche di **DevOps** impiegate al fine di

gestire al meglio tutte le diverse fasi di sviluppo del software, dall'implementazione al rilascio, manutenzioni incluse.

3.1 Licenza

Per questo progetto software si è deciso di utilizzare la licenza libera e opensource **Apache 2.0**. Come ogni licenza di software libero, essa, consente a chiunque di usare il software per qualsiasi scopo, distribuirlo, modificarlo e distribuirne anche versioni modificate. In particolare, la licenza Apache 2.0, non impone che versioni modificate del software vengano distribuite secondo i termini della stessa licenza o come software libero. Al contrario, è sufficiente specificare attraverso un'informativa che si è utilizzato software licenziato secondo i termini di questa licenza. Infatti, la motivazione principale che ha portato a questa scelta è dettata dal fatto che Apache 2.0 è una licenza che obbliga gli utenti a preservare l'informativa di diritto d'autore in essa contenuta. Questo implica che chiunque faccia uso di questo software sia obbligato a mantenere il riconoscimento di tutti gli autori che ne hanno preso parte fino a quel momento. Oltre a ciò, questa licenza offre anche la possibilità di integrarci una garanzia; per esempio, potremo chiedere una retribuzione mensile per garantire all'utente che il software continui a funzionare correttamente e se dovesse emergere un bug ci faremo a carico noi della sua risoluzione. Dunque, per tutte queste ragioni, la licenza Apache 2.0 è risultata essere quella più adatta al progetto, in quanto più strutturata rispetto alla licenza *MIT*.

La licenza è collocata nella root principale del progetto, dove sono presenti due file che ne attestano la validità:

- **LICENSE**: contiene i termini e le condizioni d'uso della licenza;
- **NOTICE**: contiene un'informativa testuale indicando i nomi degli sviluppatori, con alcuni dettagli relativi al software sviluppato.

Inoltre, all'interno di ogni file sorgente è stata aggiunta un'intestazione della licenza. Questa pratica non è obbligatoria affinché la licenza sia valida a tutti gli effetti, ma è consigliata, perché garantisce che se qualcuno visualizza uno dei file sorgenti del progetto in modo isolato dal resto, sarà comunque in grado d'identificare i termini di utilizzo previsti dalla licenza.

Infine, nel caso in cui il software o parte di esso venga redistribuito, oltre a includere il file NOTICE, nei file sorgenti deve essere preservata qualsiasi informativa di diritto d'autore e di brevetti presenti, ed in ogni file modificato deve essere aggiunta un'informativa specificando che il file è stato modificato.

Questo forza gli utenti che fanno un lavoro derivato a scrivere esplicitamente le modifiche che hanno fatto.

3.2 Git Workflow

Prima di cominciare con la scrittura del codice, tra i vari aspetti analizzati, si è discusso anche su quale modello adottare per facilitare lo sviluppo del progetto. Nello specifico, si è deciso di organizzare l'intero flusso di sviluppo secondo il **modello git-flow**. Si tratta di un modello di ramificazione (branching model) molto noto tra gli sviluppatori, il quale porta con sé numerosi vantaggi, tra cui:

- Definizione di responsabilità all'interno dei membri del team;
- Rapida identificazione di eventuali componenti del team che hanno causato rotture durante il processo di sviluppo;
- Ripristino agile della vecchia versione, nel caso in cui alcune nuove funzionalità non hanno funzionato come previsto;
- Disaccoppiamento dai rami principali dello sviluppo di nuove funzionalità;
- Possibilità di revisionare accuratamente il codice prima di unirlo al ramo principale;
- Efficace gestione dei rilasci e degli hot-fix;
- Mantenimento del repository organizzato e pulito.

In particolare, il modello git-flow prevede l'impiego di **cinque tipologie di rami**: master, hotfixes, release, develop, e feature.

Il **ramo master** è quello principale, quello da cui ha inizio il progetto, da cui poi si inizia con lo sviluppo effettivo. Esso ha il compito di memorizzare e di tracciare tutte le release ufficiali che vengono rilasciate nel tempo, ciascuna associata ad un *tag* che specifica la versione. Questo ramo riflette sempre uno stato pronto per la messa in produzione delle versioni in esso presenti.

Il **ramo develop** viene sempre considerato come un ramo principale, assieme al master, ma questo ha il compito di integrare tutte le varie funzionalità che vengono sviluppate nei differenti rami. Esso viene creato non appena si inizia con lo sviluppo delle prime funzionalità del sistema. Questo ramo ha il compito di riflettere sempre uno stato aggiornato con le ultime modifiche di sviluppo pronte per essere integrate nella prossima versione. Spesso questo ramo viene anche chiamato "ramo dell'integrazione", perché è proprio qui che vengono costruite tutte le build automatiche che integrano le modifiche effettuate nei vari rami. Quando il codice che è presente nel ramo develop diventerà stabile, pronto per essere rilasciato, tutte le modifiche verranno riunite al ramo master, passando per il ramo *release* (vedi dettagli in seguito). Ogni volta che le

modifiche vengono nuovamente unite al ramo master, viene generata una nuova versione.

Accanto ai rami principali master e develop, il flusso di lavoro prevede l'impiego di una varietà di **rami feature**, al fine di supportare lo sviluppo in parallelo delle diverse funzionalità previste dal sistema. Dunque, quando si ha la necessità di sviluppare una nuova funzionalità, si crea sempre un apposito ramo, avente l'obiettivo di portare a termine l'implementazione di quella specifica funzionalità. Esso viene nominato con un nome simile e vicino alla funzionalità che si desidera sviluppare. Ciascun ramo feature viene creato a partire dal ramo develop e non dal ramo master. Quando poi una certa funzionalità è completa, viene nuovamente unita al ramo develop. A differenza dei rami principali, questi rami hanno sempre una durata limitata, poiché prima o poi verranno uniti al ramo develop. Se lo sviluppo di una particolare funzionalità richiede del tempo, è bene, di tanto in tanto, fare dei *pull* dal ramo develop su quello della feature, in modo da evitare di divergere e di avere poi problemi durante il merge finale su develop (conflitti e/o complicanze in fase d'integrazione).

Una volta che nel ramo develop sono presenti abbastanza funzionalità per effettuare un rilascio, si crea un nuovo **ramo release**. La creazione di questo ramo avvia il successivo ciclo di rilascio, pertanto non sarà più possibile introdurre nuove funzionalità su di esso. In questo nuovo ramo si potranno effettuare solo delle correzioni di bug relative alle funzionalità già presenti, generare della documentazione o altre attività orientate esclusivamente al rilascio. Una volta che la release è ufficialmente pronta, può essere unita sia al ramo master, sia al ramo develop, in modo da proseguire sia con il rilascio ufficiale (ramo master), sia con l'aggiunta di ulteriori funzionalità (ramo develop). L'utilizzo di un ramo dedicato per preparare le versioni ufficiali consente ad alcuni membri del team di perfezionare la versione corrente, offrendo allo stesso tempo anche la possibilità ad altri membri del team di continuare a lavorare su nuove funzionalità (rami feature) per la versione successiva. Un aspetto a cui bisogna prestare attenzione è quello di non unire nuove funzionalità al ramo develop prima che la release non sia stata unita al ramo develop stesso. Questo è necessario affinché le future versioni contengano anche le eventuali correzioni di bug che sono state applicate durante la release precedente.

Il **ramo hotfixes** viene utilizzato per correggere rapidamente bug imprevisti nelle versioni in produzione (che sono state quindi rilasciate sul ramo master). Pertanto, questo ramo viene creato a partire dal ramo master. Non appena la correzione è completa, deve essere unita a entrambi i rami master e develop (o al ramo release della versione corrente), e su master, la nuova versione deve

essere contrassegnata con un nuovo numero di versione (si incrementa la PATCH). Avere un ramo di sviluppo dedicato solo per le correzioni di bug consente al team di sviluppo di affrontare e risolvere i problemi senza interrompere il resto del flusso di lavoro o attendere il prossimo ciclo di rilascio.

Nella figura 3.1 è possibile visualizzare in forma grafica l'intera gestione del flusso del modello git-flow appena descritto.

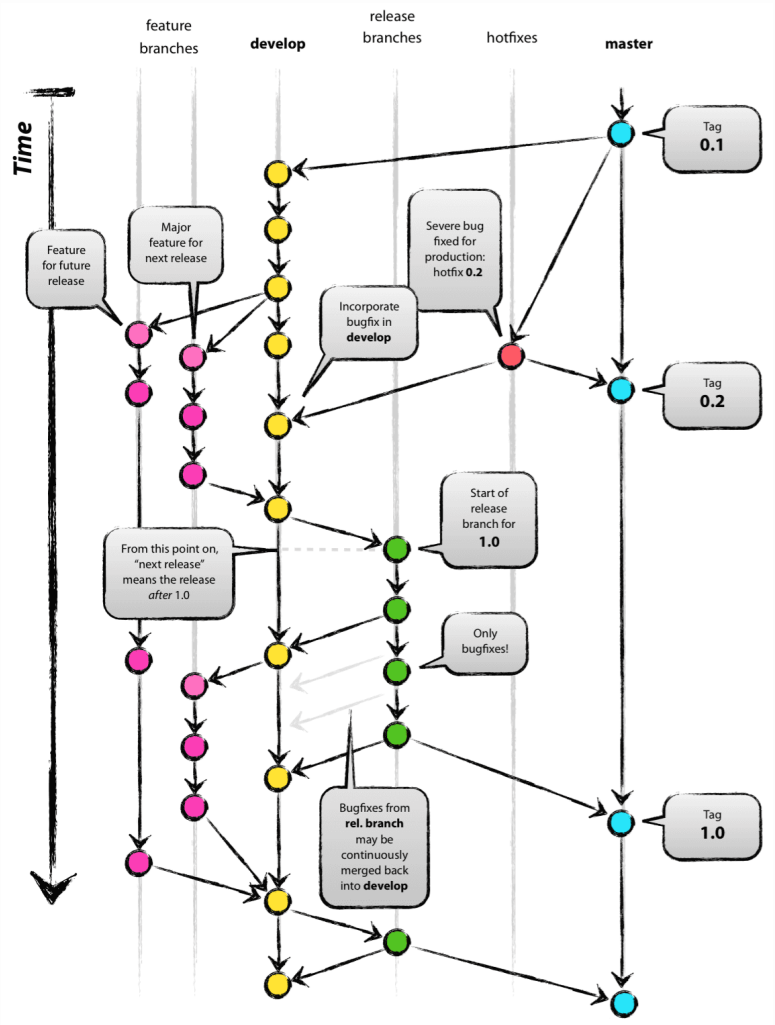


Figura 3.1 - Modello git-flow

3.3 Versioning

Il modello adottato per il controllo delle versioni è il **Semantic Versioning** (**SemVer**). Si è deciso di applicare questo tipo di schema numerico perché porta con sé numerosi vantaggi, tra cui:

- Possibilità di applicare informazioni aggiuntive: pre-release e/o build-

metadata;
- Possibilità di identificare particolari versioni sperimentali in fase beta;
- Evidenziare la differenza con la precedente versione;
- Possibilità di dedurre la versione dallo stato del DVCS;
- Possibilità di parziale automatizzazione;
- Flessibile e adattabile.

Il SemVer ha l'obiettivo di codificare i numeri di versione e la loro modifica, al fine di trasmettere all'utente finale un significato sul codice che troverà e su ciò che è stato modificato da una versione all'altra. Il formato di SemVer è il seguente: `X.Y.Z[-P][+B]`, in cui:

- `X`: Major;
- `Y`: Minor;
- `Z`: Patch;
- `P`: Pre-release;
- `B`: Build metadata.

In particolare, si è scelto 0.1.0 come valore per la prima versione. Il primo zero indica che si tratta di uno sviluppo iniziale con un'API pubblica instabile, in cui tutto può cambiare in qualsiasi momento, prevedendo dei rilasci molto frequenti (sviluppo sperimentale). I successivi rilasci, avranno il seguente schema di versione: `0.1.0-[branchName]-[YYYYMMGG]T[HHMMSS]`.

- `branchName`: indica il nome del branch da cui è stata pubblicata la versione. Il nome del branch rispecchia la funzionalità che si sta sviluppando;
- `YYYYMMGG`: indica l'anno (YYYY), il mese (MM) e il giorno (GG), in cui è stata pubblicata quella specifica versione;
- `HHMMSS`: indica le ore (HH), i minuti (MM) e i secondi (SS), in cui è stata pubblicata quella specifica versione.

Questo specifico schema consente di individuare a colpo d'occhio la funzionalità che si sta sviluppando in quella serie di rilasci, avendo anche un riferimento temporale costante sull'evoluzione del software nel tempo. Tutto questo processo è stato automatizzato, sfruttando i tag che vengono assegnati per le versioni stabili (vedi paragrafo successivo).

Poi, quando il software raggiungerà un buon grado di stabilità, verrà assegnata la versione 1.0.0, la quale fornisce una misura di quanto il software è cambiato per il cliente (utilizzatore finale). L'assegnazione delle successive versioni stabili che verranno rilasciate, compresa la 1.0.0, verranno assegnate sempre manualmente perché non è possibile automatizzare questo tipo di operazione.

Infine, nel caso in cui il software dovesse subire ulteriori espansioni e/o

modifiche, le successive versioni verranno assegnate secondo la seguente logica:

- Incrementare la PATCH solo quando si effettuano bug-fixing retro-compatibili. Con bug-fixing si intende la correzione di uno o più comportamenti errati;
- Incrementare la MINOR quando sussistono una o più delle seguenti condizioni:
  - introduzione nell'API pubblica di una nuova funzionalità compatibile con le versioni precedenti;
  - deprecazione di una funzionalità nell'API pubblica;
  - cambiamento di alcune dipendenze (e.g. cambio di libreria/e);
  - introduzione di nuove funzionalità o miglioramenti sostanziali all'interno del codice privato.
- Incrementare la MAJOR quando sussistono una o più delle seguenti condizioni:
  - introduzione di modifiche all'API pubblica che risultano incompatibili con le versioni precedenti;
  - introduzione di cambiamenti importanti e/o grandi estensioni.
  - modifica di un API pubblica;
  - assenza di retro-compatibilità;

NOTA: la versione della patch deve essere reimpostata a zero quando viene incrementata la versione minor. La versione della patch e della minor, devono essere reimpostate a zero quando viene incrementata la versione major.

3.3.1 Automazione del Semantic Versioning

Come accennato, per la maggior parte delle release il processo di assegnazione della versione è automatico, basandosi direttamente sulla storia del repository e sull'ultima versione stabile. A tal fine, è stato creato uno **script bash**, `version.sh`, il quale si occupa di calcolare la giusta versione ogni volta che viene avviata la fase di **deploy** su Travis CI. In particolare, lo script:

1. utilizza il comando `git describe` per ottenere l'ultimo tag (se non è presente utilizza 0.1.0 di default);
2. estrae la parte stabile del tag ottenuto nel punto precedente (i.e. la versione in forma X.Y.Z);
3. concatena alla parte stabile il nome del branch corrente (quello che ha generato la build su Travis CI) e il timestamp dell'ultimo commit, ottenendo cosi una stringa formattata nel seguente modo: `X.Y.Z-[branchName]-[YYYYMMGG]T[HHMMSS].`

3.4 Build automation

Come tool per automatizzare il processo di sviluppo si è deciso di utilizzare **Gradle**, in Kotlin. Esso consente di automatizzare gli aspetti più ripetitivi, sollevando gli sviluppatori dall'incarico, aumentando cosi la produttività dell'intero team. Vediamo di seguito quali tipi di automazioni e plugin si sono adottati.

3.4.1 Controllo di qualità

Per accertarsi che il codice scritto sia conforme agli standard di qualità desiderati, si sono adottati i seguenti tre plugin per Gradle:

- **Checkstyle**, applica le regole specificate all'interno del file `config/checkstyle/checkstyle.xml`, riguardanti lo stile, come ad esempio buone norme di spaziatura, indentazione di parentesi, nomi, documentazione su metodi non privati;
- **PMD**, applica le regole specificate all'interno del file `config/pmd/pmd.xml`, riguardanti aspetti più complessi, come il design delle classi, gestione delle eccezioni, utilizzo di interfacce al posto delle implementazioni (dove possibile), complessità dei metodi, e altri aspetti riguardanti lo stile di programmazione;
- **SpotBugs**, che come suggerisce il nome, va alla ricerca di bug comuni, elencati in questa pagina, con alcune piccole eccezioni specificate nel file `config/spotbugs/filters.xml`.

Tutti i plugin sono stati configurati per far fallire la build qualora almeno uno di questi rilevi un'anomalia nel codice. Inoltre, dentro la cartella `build/reports/{checkstyle, pmd, spotbugs}`, ogni plugin genera un report html dettagliato degli errori riscontrati durante il controllo.

Tutti i plugin appena citati inseriscono una loro dipendenza al task `check` del plugin *Java*, in modo che sia sufficiente lanciare tale task per effettuare tutti i controlli di qualità in automatico. Da segnalare che esso ha una dipendenza dal task *test*, quindi oltre a controllare la qualità del codice, verifica anche che tutti i test passino correttamente.

In aggiunta al task *check* è stato manualmente creato il task `qualityCheck`, che come suggerisce il nome, controlla che il codice, oltre a compilare, rispetti gli standard di qualità specificati nei plugin di cui sopra. L'unica differenza rispetto al task *check*, quindi, è che `qualityCheck` non esegue anche tutti i test, ma si limita al controllo di qualità.

3.4.2 Build

Per eseguire la build dei sorgenti è sufficiente utilizzare il task `build` (del plugin **Java**), che ha come dipendenza il task `check`, il quale, come visto, a sua volta dipende dai tre plugin per il controllo di qualità e dai test. In altre parole, ogniqualvolta si voglia eseguire la build, verranno prima eseguiti tutti i controlli di qualità e test, e solo se tutti passeranno si potrà procedere con l'effettiva build.

3.4.3 Test con coverage

Il plugin *Java* mette anche a disposizione il task `test`, che esegue tutti i test presenti all'interno della directory `test/java/`. Per essere sicuri che i test, oltre a passare, coprano buona parte del codice scritto e siano quindi una buona garanzia del corretto funzionamento del software, occorre introdurre un apposito plugin: **JaCoCo**. Esso analizza la coverage, ovvero la copertura dei test, intesa come analisi delle righe di codice che vengono eseguite durante il testing. Consideriamo, ad esempio, la seguente funzione:

```
public static Double divide(Double x, Double y) {
  if (y != 0) {
    return x / y;
  }
  return null;
}
```

Nello sviluppare il test, con la convinzione di aver effettivamente testato la funzione `divide`, può capitare che ci si limiti a scrivere le seguenti due righe di codice:

```
assertEquals(2, divide(4, 2));
assertEquals(1.5, divide(3, 2));
```

In realtà, però, non si prova a vedere cosa succede nel caso in cui il secondo numero sia uguale a zero. La coverage, in questo caso ci notificherebbe che la funzione `divide` è stata, sì, testata, ma solo parzialmente (più precisamente solo un ramo dell'`if`, su due).

Grazie al plugin JaCoCo è dunque possibile avere un quadro completo di tutto ciò che è stato correttamente testato e cosa no, in modo da provvedere a colmare eventuali mancanze.

Anche questo plugin è stato configurato per generare un report html all'interno della consueta cartella a lui dedicata, ovvero `build/reports/jacoco/test/html/`.

Il task da richiamare per analizzare la copertura dei test è `jacocoTestReport`, a cui è stata inserita la dipendenza da `test`, in modo da essere certi che il report faccia sempre rifermento all'ultima esecuzione dei test.

Oltre al task per generare il report, il plugin ne fornisce un secondo, `jacocoTestCoverageVerification`, il quale verifica che la **percentuale di copertura dei test** sia maggiore ad una soglia prefissata. Nel nostro caso, si è scelto **80%** come valore, in modo che il task fallisca se la coverage non raggiunga tale percentuale.

Passiamo ora a elencare e descrivere tutti i task realizzati al fine di automatizzare altri aspetti rilevanti.

3.4.4 Validazione del codice

Il task `validate` verifica che il codice sia valido, ovvero che compili, rispettando tutti gli standard di qualità, e che i test coprano almeno la percentuale desiderata (80%). Al termine viene anche generato il report (qualora siano state apportate modifiche a qualche sezione), sia in versione Markdown sia in PDF. Quindi, più precisamente, il task `validate` dipende da `runAllAndStop`, `check`, `jacocoTestCoverageVerification`, ed è finalizzato con `generateMdReport` e `generatePdfReport`. Si precisa che la dipendenza al task *runAllAndStop* è necessaria per avviare tutti i microservizi prima di eseguire i test, in modo che potranno correttamente essere eseguiti anche i test d'integrazione, che coinvolgono più microservizi. Per maggiori dettagli su tale task si rimanda al paragrafo seguente.

3.4.5 Esecuzione dei microservizi

Dal momento che l'applicazione è organizzata a microservizi, quando la si vuole eseguire nella sua interezza è necessario mandare in esecuzione ogni microservizio. Per automatizzare questa fase è stato creato il task `runAll`, che richiama lo script bash `application.sh` il quale fa partire in background tutti i diversi microservizi. I rispettivi PIDS (Process Identifiers) vengono esportati nella variabile LA_PIDS e anche su file, in modo da poterli stoppare successivamente quando lo si desidera. Infatti, in modo speculare, basterà richiamare il task `stopAll` per stoppare tutti i microservizi in esecuzione.

In aggiunta a questi due task, ne troviamo anche un terzo, utilizzato solo internamente, `runAllAndStop`, che manda in esecuzione tutti i microservizi e poi

li stoppa immediatamente, se non diversamente specificato. Questo è utile per eseguire i test d'integrazione del sistema (i.e. quelli relativi ai controller, che devono poter comunicare tra loro): il fatto di aver finalizzato il task con `stopAll` senza altre dipendenze intermedie a task che possono fallire, assicura che esso sarà sicuramente terminato con la `stopAll`. All'interno della *validate* sulla root è possibile osservare che il task `stopAll` deve essere eseguito solo successivamente alla *validate* (a prescindere da quale sarà il suo esito), in modo che complessivamente verrà richiamato il task *runAllAndStop*, il quale però non terminerà immediatamente come vorrebbe, ma sarà obbligato ad aspettare che la *validate* termini (eventualmente anche con errore), prima di poter invocare la *stopAll*.

Una plausibile alternativa a `runAllAndStop`, probabilmente più intuitiva, sarebbe potuta essere quella di specificare direttamente nella *validate* che deve dipendere da `runAll` e deve essere finalizzata con `stopAll`. Questo però, per come è attualmente progettato Gradle, porterebbe a un problema indesiderato: se qualche test fallisse, la *validate* non verrebbe mai finalizzata, dal momento che si interromperebbe prima e quindi la `stopAll` non verrebbe mai richiamata, senza più stoppare i microservizi in esecuzione.

In conclusione, si precisa che la *runAllAndStop* viene posta come dipendenza sia per il task *validate* che per il task *test* della root. In questo modo, prima di passare all'esecuzione dei vari test verranno sempre avviati tutti i microservizi, in modo da poter eseguire correttamente anche i test d'integrazione.

Chi desidera eseguire i test/check o la validazione non dovrà mai preoccuparsi di avviare manualmente i microservizi: sarà sufficiente lanciare semplicemente `./gradlew test` (o anche `./gradlew check`) o `./gradlew validate` e tutto funzionerà correttamente senza dover specificare altro.

3.4.6 Generazione del report

Il report finale viene generato in due versioni: Markdown e PDF. Sono quindi stati creati i due task `generateMdReport` e `generatePdfReport`, che hanno entrambi il compito di mettere assieme tutti i singoli file `.md` per formare un unico report finale. Più nel dettaglio, il primo task si occupa semplicemente di fondere tutti i file `.md` e di aggiungere il tag `TOC` a inizio file, che sarà poi interpretato come Table Of Contents (i.e. indice cliccabile) quando si andrà a visualizzare il report. Il secondo task, invece, utilizza il programma **pandoc** per generare il report in formato Latex/PDF a partire dai sorgenti `.md`.

3.4.7 Preparazione dei container

Come vedremo meglio nella sezione dedicata alla continuous integration, la fase di deployment di release stabili prevede la realizzazione dei *container*, all'interno dei quali ciascun micro-servizio andrà in esecuzione. Il task `prepareContainer` si occupa quindi di creare la cartella *container* per ciascun modulo applicativo, al cui interno verrà inserito il *Dockerfile* e tutti i file necessari per eseguire l'applicazione (i.e. il jar nel caso delle applicazioni Java e i sorgenti del client per quanto riguarda la web app in Node.js).

3.5 Continuous integration

La Continuous Integration (CI), come suggerisce il nome stesso è la tecnica che consiste nell'effettuare integrazioni frequenti di piccole porzioni di codice, piuttosto che effettuare un'unica grossa integrazione alla fine del ciclo di sviluppo. L'obiettivo è quindi di costruire un software in modo *incrementale*, aggiungendo di volta in volta piccole funzionalità che, almeno singolarmente, svolgono il loro compito nel modo corretto. Attraverso la CI si vuole fare in modo che una volta aggiunte nel sistema, le funzionalità lavorino nel modo corretto, interagendo regolarmente le une con le altre.

Nel progetto si utilizza **Travis CI** come piattaforma di Continuous Integration. Esso supporta quindi tutto il processo di sviluppo, andando ad eseguire, ad ogni nuovo push, le fasi di **build** e di **test** del sistema, fornendo immediatamente un riscontro sul successo o meno delle modifiche. In questo modo è subito possibile accorgersi di eventuali problemi o bug che possono essere sorti quando è stata introdotta una nuova funzionalità, potendoli così risolvere nell'immediato, senza che si accumulino tutti alla fine dell'intero processo di sviluppo. Al termine della fase di test, se tutto continua a funzionare nel modo corretto, è possibile anche effettuare il **deploy** dell'applicazione, sempre in modo del tutto automatico.

Vediamo ora più nel dettaglio come si è deciso di configurare la pipeline.

3.5.1 Matrice di job

Per garantire la portabilità del software sui principali sistemi operativi, ovvero Linux, Windows e MacOS, si vuole essere certi che esso funzioni correttamente su tali piattaforme, anche con versioni differenti di Java. A tale scopo, su Travis CI è possibile creare delle **matrici**, in cui si combinano tutte le versioni di Java desiderate con tutti i sistemi operativi specificati. Per ciascuna combinazione verranno eseguiti i consueti test, potendo così rilevare eventuali problemi di compatibilità e se possibile risolverli o suggerire subito di non utilizzare una determinata versione di Java che su un sistema operativo potrebbe non essere

del tutto compatibile con determinate funzionalità. Per realizzare quanto detto, è stata creata la seguente **matrice di job**:

	Java 16 @zulu	**Java 16 @adopt**	**Java 16 @openjdk**	**Java 17 @openjdk**	**Java 1 @amaz**
Linux	Check, Deploy, Containerization	Test	Test	Test	Test
Windows	Test	Test	Test	Test	Test
MacOS	Test	Test	Test	Test	Test

Ciascun incrocio rappresenta un diverso *job*, ovvero un processo automatizzato che clona il repository in un nuovo ambiente virtuale e poi porta avanti una serie di fasi, come compilare il codice ed eseguire i test. Come si nota, il job eseguito in tutti gli incroci, a eccezione del primo, è **Test**, che consiste nell'esecuzione di tutti i test che sono stati definiti (mediante il comando `./gradlew test`). Ulteriori job sono però stati inseriti sul sistema operativo e JDK di riferimento (Linux e Java 16), ovvero *Check*, *Deploy* e *Containerization*, dato che possono e/o devono essere eseguiti una sola volta. Vediamo ora più nel dettaglio:

- **Check** sostituisce *Test* e consiste nell'analisi della qualità del codice (mediante i plugin descritti in precedenza) e nella successiva esecuzione tradizionale dei test. Più precisamente, questo job utilizza il comando `./gradlew check` e dato che il task `check` di Gradle dipende dal task `test`, non ha senso eseguire il job *Test* qui. Si esegue il job Check una sola volta (in questo incrocio) dal momento che è più dispendioso e comunque non ha senso ripetere ogni volta il controllo di qualità del codice: se passa la prima volta significa che il codice è di qualità e quindi lo sarà sempre a prescindere dal sistema operativo e dalla versione di Java. In tutti gli altri incroci ci si può pertanto limitare ad eseguire solo i test tradizionali.
- **Deploy** si occupa di effettuare il deploy del software su *GitLab Releases*, generando il report in formato Markdown e PDF. Alla release viene assegnato il tag corrispondente alla versione che si sta rilasciando, generato secondo le modalità del *Semantic Versioning* trattato nella sezione 3.3. Dal momento che Travis non supporta in modo nativo il deploy su GitLab releases, è stato creato uno script bash che sfrutta le API di GitLab. Più in dettaglio, tale script è programmato per caricare i jar e i report nella release. Si segnala, però, che dal momento che GitLab nella versione free ha un limite molto basso per gli upload (10 MB), si è deciso di commentare la parte di codice relativa al caricamento dei jar, dato che altrimenti fallirebbe sistematicamente a causa del limite eccessivamente basso. Pertanto, nella

release su GitLab verrà caricato il consueto zip con tutti i sorgenti e i due report; nel caso di release stabili (i.e. sul ramo *main*), verranno anche allegati i container (come collegamenti alla pagina DockerHub, vedi punto seguente).
- **Containerization** si occupa di creare, tramite il *Dockerfile*, il container su cui l'applicazione girerà, e della pubblicazione su DockerHub, in modo che sia facilmente raggiungibile e utilizzabile. Questo job viene eseguito solo per i push effettuati sul ramo *master*, per evitare di pubblicare container di versioni ancora in via di sviluppo. Anche in questo caso, è stato creato uno script bash che effettua la build di tutti i container e li carica su DockerHub.

3.5.2 Pipeline

Travis CI offre la possibilità di raggruppare i job in **stage**, in modo che tutti i job dello stesso stage possano essere eseguiti in *parallelo*, senza quindi aspettare l'esito degli altri. Gli stage, a loro volta, vengono eseguiti in modo *sequenziale*, quindi uno stage non può iniziare fino a quando il precedente non ha terminato *con successo*. Uno stage si considera terminato con successo se e solo se tutti i job di cui si compone sono a loro volta terminati con successo. Alla luce di ciò, gli stage che compongono la nostra pipeline sono i seguenti:

1. **Check**,
2. **Test**,
3. **Deploy**,
4. **Containerization**.

Facendo riferimento ai job illustrati nella sezione precedente, si ha che il job Check viene eseguito nello stage Check, tutti i job di Test nello stage Test, Deploy nello stage di Deploy e Containerization nello stage Containerization. La situazione complessiva è pertanto la seguente:

- stage **Check** composto da 1 job Check;
- stage **Test** composto da 3x5 - 1 = 14 job Test;
- stage **Deploy** composto da 1 job Deploy;
- stage **Containerization** composto da 1 job Containerization.

Per quanto detto, si potrà procedere con il testing su tutte le configurazioni solo se lo stage di Check è terminato con successo, ovvero se il codice è di qualità e gira correttamente sull'architettura di riferimento. In modo analogo, si procede con la fase di deploy solo se i test sono passati su tutte le configurazioni specificate.

4 Dettagli implementativi

In questo capitolo verranno illustrati gli aspetti implementativi che, a nostro avviso, si sono rivelati più interessanti.

4.1 Specifica OpenAPI

Le funzionalità fornite dall'interfaccia RESTful del gateway, le quali rappresentano tutte e sole le rotte richiamabili dagli utenti, sono state descritte mediante le **specifiche OpenAPI**. Esse rappresentano uno standard che consente sia agli umani che alle macchine di capire quali sono le funzionalità offerte dal servizio, senza dover analizzare il codice sorgente. Inoltre, a partire dalle specifiche OpenAPI, si possono utilizzare tool che generano una vera e propria documentazione grafica, attraverso la quale l'utente può interagire eseguendo in tempo reale le diverse API.

Nel nostro caso, per rappresentare tali specifiche si è fatto uso di una libreria per Spring, *springdoc-openapi*, che a partire dal codice sorgente genera in automatico le rispettive specifiche OpenAPI, in formato *.yml*. Queste specifiche possono essere consultate anche attraverso un'interfaccia grafica interattiva, all'indirizzo `http://localhost:8080/swagger-ui/index.html`. Da qui è possibile visualizzare tutte le API che il gateway espone ed è anche possibile utilizzarle a tutti gli effetti, potendo personalizzare eventuali parametri. Dal momento che per la maggior parte delle API è necessario fornire il token di accesso, è presente un pulsante in alto a destra, Authorize, dal quale è possibile specificare il token da utilizzare nelle API dove è necessaria l'autenticazione (quelle contrassegnate da un lucchetto).

Oltre all'interfaccia grafica, è anche possibile visualizzare le vere e proprie specifiche OpenAPI in formato .yml, all'indirizzo `http://localhost:8080/v3/api-docs`. Queste sono poi state esportate su **SwaggerHub**, che è un servizio sul quale è possibile caricare le proprie specifiche OpenAPI e renderle così accessibili a chiunque. Da qui è sempre possibile visualizzare la stessa pagina del primo link. L'unica differenza è che in quest'ultimo caso la documentazione non è interattiva, dal momento che il gateway non è in esecuzione sul loro server. Per consultare le specifiche OpenApi del gateway su SwaggerHub senza doverlo mandare in esecuzione, è possibile raggiungere il seguente indirizzo https://app.swaggerhub.com/apis/magnanig/luckyauction/1.0.0.

4.2 Socket.io

Per l'invio di notifiche istantanee, come primi fra tutti i rilanci di un'offerta, si sono utilizzate le socket. Grazie alla loro modalità *push*, queste consentono a un utente che sta partecipando a un'asta di vedere in tempo reale le diverse offerte che arrivano, in modo da poter rilanciare prontamente, senza dover ricaricare continuamente la pagina (modalità pull).

Una nota libreria che consente di realizzare quanto appena detto è **socket.io**, che ha il vantaggio di avere un'integrazione su tantissimi ambienti di sviluppo differenti, permettendo così praticamente a qualsiasi applicazione di agganciarsi alla socket. La parte server delle socket è stata realizzata in Java e queste sono in esecuzione (i.e. in ascolto) sui microservizi delle aste e delle automazioni, dato che sono proprio questi ultimi che generano eventi da notificare immediatamente.

Sempre in Java, troviamo anche l'implementazione dei rispettivi client, dal momento che il microservizio degli attori deve ricevere le notifiche relative ai rilanci, in modo tale che gli attori possano a loro volta rilanciare prontamente in automatico. La versione client è poi anche implementata in Javascript, in modo che anche la pagina web possa aggiornarsi istantaneamente all'arrivo di una nuova offerta. Da questo possiamo proprio apprezzare l'eterogeneità di questa libreria, in grado di funzionare correttamente in ambienti completamente differenti.

Come già visto nei diagrammi discussi nelle precedenti sezioni, sono state implementate tre diverse socket:

- **BidsSocket**, dove vengono inviati i diversi rilanci effettuati dagli utenti;
- **AuctionSocket**, dove vengono inviate le notifiche riguardanti le aste (i.e. terminazione, posticipazione o estensione di un'asta);
- **AutomationSocket**, dove vengono inviate le notifiche riguardanti le automazioni (i.e. creazione o terminazione di un'automazione).

Al fine di organizzare in modo logico e ottimale le socket, sono stati creati due **namespace** per ogni socket. Il primo nella forma `/[socketName]` (e.g. `/auctions`), che consente di stare in ascolto di tutti gli eventi possibili (riguardo quella specifica socket) che si verificano *in tutte le aste*; il secondo, invece, nella forma `/[socketName]/[auctionId]` (e.g. `/auctions/12345`), che consente di stare in ascolto di tutti gli eventi (riguardo quella specifica socket) riferiti però alla *sola asta specificata*. Questa seconda opzione è particolarmente utile per il client web, dato che quando un utente è fermo su una pagina di un'asta vorrà ricevere gli eventi (e.g rilanci) riguardanti solo quell'asta. Questo consente

anche di evitare di utilizzare la rete con messaggi che sarebbero, per il 99% delle volte, scartati dal client.

La **BidsSocket** presenta in realtà una versione leggermente diversa del primo namespace, dal momento che le offerte possono essere potenzialmente un numero molto elevato. Utilizzando un'unica socket per ricevere tutte le bid di tutte le aste attualmente in corso, si potrebbero creare dei colli di bottiglia su quel namespace della socket e magari chi riceve tutti quei messaggi non è realmente interessato proprio a tutti. Basti pensare alla modellazione ad attori: se un router sta gestendo le automazioni per le aste con gli id 1, 2, 3, 4 e 5, sarà idealmente interessato a ricevere dalla socket solamente le bid per quelle cinque aste. Potrebbe allora utilizzare i namespace nella versione per la singola asta, ma questo potrebbe non scalare bene se un router si trovasse a gestire N aste, con N potenzialmente grande. La giusta soluzione spesso sta nel mezzo, ed è ciò che quindi è stato fatto.

Per evitare di avere una granularità troppo bassa (una sola socket che riceve tutte le bid indistintamente) o troppo alta (tante socket in cui ognuna riceve solo le bid di una sola asta) si è introdotto il concetto il **bucket**. Un bucket è un raggruppamento di aste e può essere determinato in modo univoco a partire dell'id di un'asta, ad esempio sommando i valori ASCII di ciascun carattere che compone l'id e poi facendo il modulo del numero di bucket che si vogliono creare. A questo punto ci si può registrare solo sui bucket che corrispondono alle aste desiderate, ricevendo possibilmente ancora più bid di quante effettivamente ne servono, ma comunque M volte in meno rispetto alla soluzione con la singola socket (dove M è il numero di bucket).

In conclusione, il primo namespace per la BidsSocket sarà nella forma `/bids/[bucket]`. Si precisa che bucket "0" è riservato e indica un namespace su cui vengono inviate regolarmente tutte le bid, quindi come se non ci fosse nessun bucket. Nell'implementazione attuale si utilizza solamente una socket con bucket "0" ma il sistema è comunque già stato progettato secondo quanto detto, in modo che se dovesse essere necessario scalare, si potrà fare molto rapidamente andando a definire in modo opportuno il metodo che mappa l'id dell'asta nel numero del bucket corrispondente (`SocketsConfig.getBidSocketId`, dentro il modulo `configuration`).

4.3 Programmazione ad attori con Akka

Per la programmazione ad attori si è utilizzato **Akka**, un framework che consente di realizzare applicazioni *concorrenti*, *distribuite* e con comunicazione basata su scambio di *messaggi asincroni* (la cui coda è automaticamente

gestita).

Nel progetto, la granularità di distribuzione è a livello di microservizio, quindi l'idea consiste, potenzialmente, nell'installare ed eseguire ciascun microservizio su una macchina diversa, eventualmente anche in località completamente distanti. Pertanto, il microservizio degli attori è stato considerato come un tutt'uno, ma nulla vieta di aumentare ulteriormente la granularità, qualora dovesse essere necessario scalare, andando quindi a creare una distribuzione anche a livello di singoli attori. Questo consentirebbe di avere ad esempio mille attori su una macchina a Milano, mille a Roma, mille a Parigi e così via, a seconda della necessità. Akka consente, attraverso un'opportuna configurazione degli indirizzi IP delle macchine disponibili, di realizzare quanto appena detto, scegliendo automaticamente a quale macchina assegnare il carico di lavoro in modo del tutto automatico e trasparente al codice di ciascun attore. Questa è la principale motivazione che ha portato all'utilizzo di questo framework, oltre sicuramente alla sua semplicità di utilizzo ed efficienza.

4.4 Sicurezza basata su token JWT

La sicurezza del sistema è implementata all'interno del gateway, unico punto di accesso al sistema. Avendo adottato il framework Spring a livello applicativo per creare ogni microservizio, si è deciso di utilizzare Spring Security per consentire agli utenti di effettuare un accesso protetto al sistema.

Spring Security è un framework di autenticazione e controllo degli accessi altamente personalizzabile. È lo standard de facto per la protezione delle applicazioni basate su Spring. Questo framework si concentra sulla fornitura sia di autenticazione che di autorizzazione alle applicazioni Java. L'autenticazione ha l'obiettivo di confermare la corretta identità di un utente, attraverso username e password. Mentre, l'autorizzazione si occupa di consentire o proibire l'accesso ad una risorsa da parte di un'utente.

Il fulcro della sicurezza è presente all'interno della classe `SecurityConfig`. All'inizio si specifica in che modo Spring Security deve riconoscere la validità o meno delle credenziali di un utente, non appena viene ricevuta una richiesta di login. Per fare ciò, occorre inviare una richiesta http al microservizio responsabile della gestione degli utenti. Successivamente, si va a specificare tutta la struttura della sicurezza, stabilendo per ogni rotta quali sono i permessi che ciascun utente deve possedere per accedere a quella specifica risorsa. Le uniche rotte per cui si è deciso di escludere l'intero processo di sicurezza sono le seguenti:

- "/api/auctions/active", restituisce tutte le aste attualmente in corso;
- "/api/users/signup", consente ad un utente di registrarsi al sistema;
- "/api/token/refresh", consente di ottenere nuovi token di accesso.

Inoltre, in Spring Security è possibile specificare una **catena di filtri** che ogni richiesta dovrà percorrere prima di giungere all'interno del sistema. In particolare, sono stati inseriti due filtri. Il primo filtro della catena è definito dalla classe `JwtTokenAuthorizationFilter`, il quale è responsabile della fase di autorizzazione. Mentre il secondo è definito dalla classe `JwtTokenAuthenticationFilter`, ed è responsabile della fase di autenticazione.

Il filtro **JwtTokenAuthorizationFilter** si occupa d'intercettare ogni richiesta, a eccezione delle rotte che sono state escluse e della richiesta di login. Questo filtro è finalizzato a controllare la validità del token fornito al momento della richiesta. Nel caso in cui il token non sia presente o non sia valido, la richiesta verrà rifiutata immediatamente. Altrimenti, se il token è valido e l'utente ha i permessi per accedere alla risorsa desiderata, la richiesta verrà inoltrata ai microservizi di competenza.

A differenza del precedente, il filtro **JwtTokenAuthenticationFilter** viene applicato solo alle richieste di login. In prima istanza si prova ad autenticare l'utente. In caso di credenziali sbagliate la richiesta termina immediatamente. Altrimenti, in caso di successo, nel corpo della risposta vengono inseriti due tipi di token:

- **Access token** (token di accesso): serve per accedere alle risorse desiderate;
- **Refresh token** (token di aggiornamento): serve per ottenere nuovi token di accesso, al fine di evitare login frequenti da parte dell'utente. Solitamente, il tempo di validità del token di aggiornamento è maggiore di quello del token di accesso.

Entrambi i token vengono firmati con l'algoritmo HS256 (HMAC con SHA-256) utilizzando una chiave segreta. Nello specifico, si utilizza la stessa chiave sia per generare la firma che per convalidarla (chiave simmetrica). Inoltre, in relazione al tempo di validità di ciascun token, è stato impostato ad un'ora per il token di accesso, mentre a due ore per il token di aggiornamento. Per ulteriori dettagli si rimanda all'implementazione della classe `JwtUtils`.

Infine, è importante sottolineare che il token di aggiornamento viene introdotto nella sicurezza dei sistemi al fine di bilanciare sicurezza e usabilità. Oltre a ciò, spesso viene creata una gestione ad-hoc anche per questo tipo di token per impedire che vengano utilizzati da parte di persone malintenzionate (aggressori). In questo progetto non ci si è spinti fino a questo livello in quanto esulava dagli obiettivi di progetto.

4.5 MongoDB

Come database si è scelto di utilizzare **MongoDB**, un noto database documentale che permette, mediante opportuni adapter (menzionati in precedenza), di passare dagli oggetti del dominio ai rispettivi Json da memorizzare, e viceversa. Inoltre, un ulteriore vantaggio dei database documentali è il fatto che, scegliendo la giusta modellazione, in base al carico di lavoro, si possono evitare i consueti join tra le tabelle dei tradizionali database relazionali. Infine, un aspetto interessante specifico di MongoDB è la possibilità di utilizzare un database in cloud in modo completamente gratuito, accessibile anche via codice e non solo dalla dashboard web.

Dunque, al fine di mantenere il sistema distribuito, non è stato creato un unico database centrale contenente tutte le collezioni (i.e. "tabelle"), ma ne è stato creato uno per ogni microservizio, ciascuno contenente solamente le collezioni utili a quello specifico microservizio (spesso solo una). Per i test è stato creato un ulteriore database dedicato, con le relative collezioni, in modo da non sporcare quelli ufficiali.

Per integrare MongoDB a Java è stata utilizzata la libreria ufficiale di MongoDB, che consente di lanciare ogni tipo di query direttamente via codice. Questa offre il vantaggio di effettuare in automatico ogni conversione da oggetti a documenti Json e viceversa (i.e. serializzazione e deserializzazione), a patto di specificare opportuni adapter, ovvero oggetti con costruttore vuoto e *getter* e *setter*. Si può quindi semplicemente chiedere di aggiungere un oggetto alla collezione e la libreria in automatico convertirà l'oggetto in Json e lo inserirà nella collezione desiderata. Il discorso vale ovviamente in entrambe le direzioni. Questo consente di non doversi preoccupare delle diverse conversioni e quindi di continuare a lavorare liberamente con il tradizionale paradigma a oggetti.

5 Autovalutazione e convalida

Vista la natura del progetto, sin dall'inizio è sembrato opportuno adottare una serie di strumenti che durante l'intero ciclo di sviluppo aiutassero a mantenere un buon livello di **qualità del codice** realizzato. A tal fine, come già specificato nella sezione relativa alla build automation, sono stati adottati i seguenti plugin: **CheckStyle**, **PMD** e **SpotBugs**. A ogni push, la pipeline della Continuous Integration verifica che il codice soddisfi sempre la qualità specificata nelle configurazioni di tali plugin, facendo fallire la build qualora ciò non avvenga (con il conseguente invio di una email).

Inoltre, per garantire l'**efficacia** dei risultati del progetto, sono stati sviluppati numerosi test. Nello specifico, sono stati realizzati due tipologie di test: **unit test** e **integration test**. I primi, sono stati impiegati per testare singolarmente i livelli di dominio, casi d'uso e persistenza (di ogni contesto). Mentre i secondi sono stati realizzati a livello di controller per testare la coordinazione dei vari componenti del singolo bounded context, nonché eventuali interazioni con altri micro-servizi.

Come già descritto nella sezione di build automation, a supporto della qualità dei test realizzati, è stato adottato il plugin JaCoCo il quale consente di analizzare la copertura dei test (**coverage**). Nel caso di questo progetto, si è deciso di impostare una copertura minima dell'80%.

I test totali realizzati nei diversi microservizi sono pari a 261, con una coverage media del 96.1%. La tabella seguente mostra un riepilogo dettagliato del numero di test effettuati in ciascun modulo (i.e. incrocio riga-colonna), riportando tra parentesi la relativa percentuale di coverage raggiunta.

	domain	**usecase**	**controller**	**mongo-repository**
users	35 (100%)	5 (100%)	4 (99%)	3 (89%)
movements	37 (100%)	6 (98%)	6 (99%)	7 (96%)
auctions	65 (96%)	22 (95%)	10 (86%)	34 (95%)
automations	7 (98%)	6 (100%)	8 (90%)	6 (97%)

Oltre a questi, sono presenti ulteriori 13 test relativi alle funzioni di utility presenti nel rispettivo modulo. Dunque, in totale sono stati realizzati 274 test.

Infine, grazie alle **Condition of Satisfaction** (CoS) definite durante la sessione di knowledge crunching, associate a tutti i diversi scenari di utilizzo, è possibile stabilire un criterio per la valutazione del software prodotto e la sua conformità ai requisiti. Nello specifico, mediante il prototipo di front-end realizzato è possibile verificare l'assolvimento di tutte le diverse CoS e quindi il raggiungimento degli obiettivi richiesti.

6 Istruzioni di deploy

Per eseguire il deploy si forniscono **due modalità**, in modo da poter scegliere quella più comoda. La prima fa uso dei **container Docker**, quindi per funzionare presuppone che quest'ultimo sia correttamente installato. La seconda, invece, non richiede nessun requisito aggiuntivo, se non **Java 16** (o versioni superiori).

Prima di procedere, si ricorda che l'applicazione è organizzata a microservizi, quindi il deploy consiste nel mettere in esecuzione tutti e sette i microservizi. Più precisamente, i microservizi in esecuzione saranno i seguenti (tra parentesi la libreria utilizzata a livello applicativo):

- Gateway (Spring)
- Utenti (Spring)
- Movimenti (Spring)
- Aste (Spring)
- Automazioni aste (Spring)
- Attori per eseguire le automazioni (Akka)
- Front-end (Node.js)

Al momento dell'esecuzione, è possibile scegliere se avviare l'applicazione nel modo tradizionale, quindi mandando in esecuzione tutti i microservizi come se fosse il **primo avvio**, oppure se avviarla in seguito ad una **interruzione del servizio**. Ad esempio, supponiamo che l'applicazione venga mandata in esecuzione la prima volta nel modo tradizionale (ex-novo), poi dopo una settimana ci sia un qualche problema che manda l'applicazione in down per un'ora. Durante questo lasso di tempo, nessuno può rilanciare per le aste dato che l'applicazione è in down. Quando poi si riavvia l'applicazione nel modo tradizionale, le aste continuerebbero come se nessuna interruzione ci fosse mai stata, quindi rispettando la loro data di fine prevista. Questo farebbe sì che se alla fine di un'asta mancavano 30 minuti quando c'è stata l'interruzione, una volta riavviata l'applicazione (dopo il down di un'ora), l'asta verrebbe immediatamente dichiarata conclusa, proclamando come vincitore l'ultimo offerente che c'era in quel momento. Questo non è il realtà un comportamento corretto, dato che magari se non ci fosse stato il down, qualcun'altro avrebbe voluto rilanciare.

Per risolvere questo inconveniente e gestire nel modo più equo possibile eventuali interruzioni del servizio, si è pensata ad una seconda modalità di avvio dell'applicazione. Più precisamente, quando la si vuole avviare, è possibile specificare, in modo opzionale, la data e l'ora in cui ha avuto inizio l'interruzione. Così facendo, le aste che erano in corso al momento del down recupereranno il tempo perso durante l'interruzione, quindi se ad esempio alla fine di un'asta mancavano 30 minuti e l'interruzione è durata un'ora, nel momento in cui il sistema viene riavviato si riprenderà da dove si era rimasti prima del down, quindi con i 30 minuti rimanenti. In questo modo nessuno verrà penalizzato.
Per ciascuna modalità di deploy saranno descritte entrambe le modalità di avvio.

NOTA: nel terminale su cui si esegue il deploy (a prescindere da quale delle due modalità si scelga) è fondamentale aver definito le variabili d'ambiente `MONGO_USERNAME` e `MONGO_PASSWORD`, in modo che l'applicazione possa connettersi al database MongoDB in cloud. Senza queste variabili, l'applicazione non funzionerà correttamente. Per fare ciò, eseguire, i seguenti comandi:

```
export MONGO_USERNAME="username"
export MONGO_PASSWORD="password"
```

Sostituire `username` e `password` con i valori ricevuti via email.

Si precisa infine che è stato creato un ramo `demo` nel quale i vincoli di durata minima di un'asta sono stati ridotti a tre minuti e l'ending zone e l'estensione di aste flash a un minuto. Questo consente di poter creare aste più brevi a fini di test. Si consiglia pertanto di effettuare il deploy a partire dal ramo `demo` per non essere costretti a creare aste che durino almeno un'ora.

6.1 Deploy mediante container

La soluzione più semplice per effettuare il deploy dell'intera applicazione (i.e. tutti i diversi microservizi) è quella che sfrutta i container **Docker**. Qui, ogni microservizio è in esecuzione su un container diverso, ed è pertanto necessario scaricare tutte le rispettive immagini da DockerHub e mandarle in esecuzione sulle giuste porte. Tutte queste operazioni sono state automatizzate nel file *docker-compose.yml*, che consente di impostare tutte le regole per il deploy dei diversi container sulle giuste porte. Pertanto, dopo aver avviato il demone di Docker ed essersi posizionati nella directory contenente il file `docker-compose.yml` (scaricabile anche da GitLab releases), per effettuare il deploy dell'intera applicazione sfruttando i container è sufficiente eseguire il seguente comando:

```
docker compose up
```

(in alcune versioni/sistemi operativi può essere necessario scriverlo nella forma `docker-compose up`).

Qualora si vogliano avviare i container in seguito ad una interruzione del servizio, per recuperare il tempo di down, si può eseguire il seguente comando:

```
INTERRUPTION_TIME="2022-01-01T12:30:00" docker compose up
```

impostando naturalmente INTERRUPTION_TIME con la giusta data e ora in cui ha avuto inizio l'interruzione (rispettando il formato).

Quando si desidera stoppare l'applicazione è sufficiente premere `CTRL+C`: questo termina tutti i container, che saranno eventualmente pronti per una nuova esecuzione con uno dei due comandi di cui sopra. Da precisare che la nuova esecuzione non comporterà un nuovo download delle immagini (ovviamente a patto che queste non siano state eliminate manualmente).

Questa soluzione con i container ha il grosso vantaggio che non è necessario avere alcun ambiente di sviluppo per poter eseguire i vari microservizi: tutto il necessario è dentro ogni container. Un piccolo contro, se vogliamo, è la dimensione di ogni immagine, circa 250 MB, che moltiplicato per 7 immagini corrisponde a un totale di 1,7 GB, che comunque non dovrebbe essere un grosso problema al giorno d'oggi. In ogni caso, se si vuole evitare un download di queste dimensioni, si può optare per la build manuale (vedi sezione seguente).

6.2 Build e deploy manuale

Per eseguire la build e il deploy del sistema è necessario avere installato Java 16 o superiori. Se il requisito è soddisfatto, si può procedere a eseguire i comandi successivi.

La prima azione da eseguire consiste nella **build** dell'intero sistema, mediante il seguente comando:

```
./gradlew assemble
```

(si suggerisce `assemble` e non `build` semplicemente per evitare di eseguire tutti i test prima di creare i jar).

Successivamente si può effettuare il deploy dell'intera applicazione sfruttando il `task` *runAll*, che manda in esecuzione in background tutti i microservizi. Dunque, il secondo comando da eseguire è:

```
./gradlew runAll
```

Questo in circa un minuto avvierà tutti i microservizi che saranno poi pronti all'uso (aspettare il messaggio di conferma). Tutti i messaggi di log saranno presenti all'interno della cartella `logs`, creata automaticamente.

Qualora si vogliano avviare i container in seguito ad una interruzione del

servizio, per recuperare il tempo di down, si può eseguire il seguente comando:

```
./gradlew runAll -PinterruptionTime="2022-01-01T12:30:00"
```

impostando naturalmente interruptionTime con la giusta data e ora in cui ha avuto inizio l'interruzione (rispettando il formato).

Quando lo si desidera, è possibile interrompere tutti i microservizi mediante il comando

```
./gradlew stopAll
```

Per le successive esecuzioni è sufficiente eseguire di nuovo il comando `./gradlew runAll`, a meno che non siano state effettuate modifiche al codice e si voglia quindi effettuare anche una nuova build (o "assemblaggio").

7 Esempi d'uso

In seguito all'avvio di tutti i microservizi (secondo una delle modalità illustrate nel capitolo di deploy), vedremo che è possibile interagire a tutti gli effetti con il sistema attraverso due modalità: con un browser web (opzione consigliata) oppure tramite il software Postman.

7.1 Browser

Grazie al servizio di front-end in ascolto sulla porta 3000 (Node.js) è possibile interagire con il sistema attraverso un prototipo d'interfaccia utente creato a scopo di test. Di seguito verranno illustrate le pagine raggiungibili.

NOTA: *è consigliato l'uso del browser Google Chrome per un'esperienza utente migliore*.

Homepage. In questa pagina sono presenti gli scenari d'uso relativi alle user stories 14 e 15. Digitando nella barra degli indirizzi il seguente indirizzo `http://localhost:3000/homepage/homepage.html` è possibile accedere alla pagina home, in cui vengono mostrate tutte le aste attualmente in corso. Per accedere a questa pagina non è necessario aver eseguito il login, in quanto si è deciso di dare a chiunque la possibilità di consultare le aste in corso. Si sottolinea che, nel caso in cui sia stato eseguito l'accesso con uno specifico

utente, la homepage permetterà la visualizzazione di tutte le aste in corso, a eccezione delle aste create dall'utente stesso.

Login. In questa pagina è presente lo scenario d'uso relativo alla user story 1. La pagina di login è raggiungibile tramite il seguente indirizzo `http://localhost:3000/login/login.html`. Nel caso si stia tentando di accedere ad una specifica pagina di cui non si hanno i permessi, si verrà reindirizzati automaticamente alla pagina di login. Attualmente, nel sistema sono registrati tre utenti con i seguenti username: "np", "gm" e "es". Per semplicità, le password di tutti e tre gli utenti equivalgono al rispettivo username. Quindi, per esempio, è possibile accedere con l'utente "es" inserendo le due lettere sia nel campo username sia nel campo password e poi cliccare sul bottone "Login".

Logout. Dopo aver effettuato il login con uno specifico utente, è possibile anche fare il logout posizionando il cursore del mouse sul bottone "Profilo" in alto a destra (nella barra del menù) e cliccare sul bottone "Logout" che comparirà.

Signup. La registrazione di un nuovo utente può essere fatta attraverso il seguente indirizzo `http://localhost:3000/signup/signup.html`. All'interno della pagina sarà obbligatorio riempire tutti i campi contrassegnati dal simbolo $*$ prima di effettuare la registrazione.

Profilo. La seguente pagina `http://localhost:3000/profile/profile.html` contiene un semplice riepilogo dei dati utente.

Gestione movimenti. In questa pagina sono presenti gli scenari d'uso relativi alle user stories 3, 4 e 5. Attraverso il seguente link `http://localhost:3000/movementsManagement/movementsManagement.html` è possibile accedere alla pagina di gestione dei movimenti in cui, oltre ai movimenti stessi, sarà presente anche il budget utente disponibile. Inoltre, all'interno di questa pagine sarà possibile ricaricare o prelevare il proprio budget.

Gestione aste. In questa pagina sono presenti gli scenari d'uso relativi alle user stories 13 e 16. All'interno della seguente pagina `http://localhost:3000/auctionsManagement/auctionsManagement.html` è possibile consultare l'intero storico delle aste utente. Inoltre, sempre da questa pagina, sarà possibile creare una nuova asta attraverso l'apposita form che comparirà cliccando sul bottone "Crea asta". Si precisa che, nel ramo di sviluppo *demo*, i vincoli per le aste flash sono stati ridotti, per consentire di testare rapidamente il comportamento del sistema. In particolare, la durata minima di un'asta è stata ridotta a 3 minuti, mentre l'ending zone e la durata di estensione sono pari a un minuto.

Asta. In questa pagina sono presenti gli scenari d'uso relativi alle user stories 6, 7, 8, 10 e 11. Al seguente indirizzo `http://localhost:3000/auction/auction.html?auctionId=AUCTION_ID` sarà possibile controllare i dettagli di una specifica asta, ma sarà anche possibile interagire con essa rilanciando una nuova offerta e/o creando una nuova automazione. Si sottolinea che nell'indirizzo, è necessario sostituire `AUCTION_ID` con l'id dell'asta d'interesse (questa pagina è comunque raggiungibile a partire dalla home, cliccando su un'asta). Oltre ad accedere alla pagina di un'asta attraverso l'indirizzo completo o dalla home, è possibile trovare il riferimento dell'asta da "Gestione aste", se questa rientra nello storico dell'utente. A quel punto basterà cliccare sull'asta per accedere alla sua pagina dedicata, senza dover digitare a mano l'URL nella barra degli indirizzi.

Nel caso in cui si desideri attivare un'automazione, è necessario compilare tutti i campi richiesti dalla specifica form, contrassegnati dal simbolo $*$. L'unico parametro opzionale è la durata dell'automazione, il quale, se non specificato, porterà l'automazione a terminare automaticamente al termine dell'asta oppure quando non sarà più possibile effettuare alcun rilancio entro i vincoli di denaro imposti. Dunque, appena viene creata, l'automazione ha immediatamente inizio, rispettando il comportamento descritto nella sezione 2.6 e nella sezione 2.7.

7.2 Postman

È possibile interagire con il sistema anche attraverso l'uso del software Postman inviando delle semplici richieste http direttamente al gateway, il quale è in ascolto sulla porta 8080. In questo caso non ci sarà un'interfaccia utente, pertanto l'interazione sarà inevitabilmente più macchinosa. Le richieste che possono essere rivolte al gateway sono definite nella sezione 4.1 in cui è illustrata la specifica OpenAPI.

8 Conclusioni

L'idea iniziale di questo progetto è sempre stata quella di voler creare un sistema distribuito di aste online dove gli utenti potessero interagire attraverso lo scambio di diverse offerte in relazione a uno specifico bene. Poi, attorno a questa idea, ha preso forma una visione che puntasse a modellare il concetto di asta a trecentosessanta gradi e che questo venisse fatto al meglio sfruttando i concetti di DevOps e di Domain Driven Design.

Inizialmente, prima di discutere in merito al problema da risolvere, all'interno del team sono state definite tutte le tecniche DevOps che avrebbero guidato l'intero sviluppo software e ci si è anche posti l'obiettivo di proporre una soluzione che fosse in grado relazionarsi con diverse tecnologie. Dunque, il piano era quello di svincolare le tecnologie implementative dalla logica di business.

In seguito alla definizione delle tecniche di DevOps, ha avuto luogo un'approfondita analisi di tutto il *problem space*, definendo i requisiti per capire esattamente cosa ci si aspettasse dal sistema finale. Al termine di questa fase, ha avuto inizio la modellazione del *solution space* con l'obiettivo di produrre un modello che fosse in grado di soddisfare tutti i requisiti emersi nella fase precedente.

La soluzione proposta si basa su un'architettura a *microservizi*, in cui ciascun bounded context è stato modellato secondo la visione della *clean architecture*, separando cosi la logica di business dal livello applicativo garantendo alta flessibilità tecnologica.

Il livello applicativo di ogni microservizio di back-end utilizza il framework Spring, fornendo un'interfaccia RESTful per relazionarsi con gli altri servizi. Il livello di persistenza è basato su MongoDB, in cloud. Per implementare le automazioni delle aste (i.e. rilanci automatici), è stato creato un ulteriore microservizio, basato sulla modellazione ad attori proposta dal framework Akka. Questo microservizio interagisce con gli altri, potendo rilanciare offerte in modo automatico per gli utenti che lo desiderano, rispettando i vincoli da loro desiderati.

Oltre ai microservizi di back-end, ne troviamo uno che funge da Gateway, che rappresenta quindi l'unico punto di accesso dall'esterno. Questo riceve le richieste da parte dei client e le inoltra al corretto microservizio. Ciò consente di mantenere un'architettura pulita e meglio manutenibile, senza che qualche modifica alle rotte interne richieda un aggiornamento dei client. Infatti, è sempre possibile modificare l'organizzazione a microservizi del back-end, cambiando ad esempio le rotte o il funzionamento, senza che i client siano mai influenzati da ciò: loro si interfacceranno sempre ed esclusivamente col Gateway e poi questo penserà a tutto (sicurezza compresa).

Infine, troviamo un microservizio che implementa il front-end, in Node.js, in modo da consentire una piacevole interazione con l'intero sistema da parte degli utenti.

Per la comunicazione tra i diversi microservizi si utilizzano richieste HTTP e, per ricevere eventi in tempo reale in modalità push, messaggi scambiati

attraverso socket.

Pertanto, considerando tutto il lavoro svolto, si può dire di aver raggiunto l'obiettivo iniziale soddisfando tutti i requisiti emersi in fase di analisi.

8.1 Lavori futuri

Nel corso dello sviluppo di questo progetto sono emerse molte idee in relazione ad alcuni aspetti interessanti che si sarebbero potuti introdurre nel sistema, ma che per mancanza di tempo non è stato possibile fare. Di seguito si elencano alcune di queste idee:

- **Gestire la fase di spedizione**. Introdurre una gestione efficace della spedizione all'interno del processo di vendita. Per esempio, ad asta conclusa, il versamento del denaro al venditore potrebbe avvenire solo dopo che è stato fornito un numero di tracking valido, il quale attesta che l'articolo è stato effettivamente spedito.
- **Distinguere tra saldo disponibile e contabile**. Il saldo contabile rispecchia tutte le transazioni che sono state effettivamente eseguite e quindi contabilizzate. Quello disponibile contiene tutte le operazioni, anche i congelamenti, i quali non sono ancora stati contabilizzati. Questo consentirebbe a ciascun utente una migliore analisi del proprio budget.
- **Restituire le aste a gruppi**. Questa idea è pensata per ottimizzare le performance di risposta del server nel caso in cui siano presenti molte aste in corso. Per esempio, quando un utente apre la pagina home del sito, in cui vengono mostrate tutte le aste in corso, anziché restituirle tutte in un'unica richiesta http si potrebbe pensare di restituirle per gruppi di trenta.
- **Introdurre il concetto di periodo nelle history**. Questa idea è pensata per ottimizzare le performance di risposta del server nel caso in cui siano presenti molti dati. Per esempio, quando un utente apre l'apposita pagina per visualizzare lo storico dei propri movimenti, anziché restituire all'utente tutti i movimenti in un'unica richiesta http, sarebbe più opportuno restituirgli tutti i movimenti dell'ultimo mese, o degli ultimi tre mesi. Discorso analogo per lo storico delle aste.
- **Sicurezza in tutti i microservizi**. Attualmente il meccanismo di sicurezza è presente solo all'interno del gateway, unico punto di accesso al sistema. In questo modo, se un "hacker" riuscisse a venire a conoscenza dell'indirizzo ip su cui risiedono i singoli microservizi e anche delle rispettive API, non ci sarebbe nulla che gli impedisca di eseguire qualsiasi

tipo di richiesta, andando a minare la sicurezza e l'integrità generale del sistema. Pertanto, si potrebbe integrare un meccanismo di sicurezza simile a quello analizzato in precedenza, anche all'interno di ogni singolo microservizio, anziché averlo solo nel gateway.

8.2 Conoscenze apprese

Lo sviluppo del progetto ci ha portato ad acquisire conoscenze su più fronti. Primo su tutti, percepiamo di aver acquisito la consapevolezza di saper affrontare lo sviluppo di un sistema complesso che sia completo di ogni sua parte. In più, percepiamo anche di avere acquisito una visione di sistema che sia molto più completa rispetto a prima. Vediamo più nel dettaglio gli aspetti di maggior rilievo.

DevOps. Abbiamo apprezzato i veri vantaggi che si possono ottenere attraverso l'impiego di tecniche di DevOps durante l'intero ciclo di sviluppo. All'inizio è servito un po' di tempo per creare i vari task, impostare i vari plug-in e collegare il tutto con TravisCI, però, sin dalle prime fasi implementative, ci si è resi conto che se non avessimo adottato tutti questi automatismi il prodotto finale sarebbe stato sicuramente di minor qualità e i tempi realizzativi sarebbero aumentati ulteriormente, ad esempio per ricercare bug non risolti tempestivamente. Inoltre, l'integrazione di nuove funzionalità sarebbe stata estremamente difficoltosa senza l'uso delle tecniche impiegate.

Di particolare utilità si è rivelata anche la fase di deploy automatizzata, che permette, ad ogni push valido (i.e. con tutti i test funzionanti), di generare immediatamente una release su GitLab, calcolando automaticamente la giusta versione e, nel caso di release stabili (quindi sul ramo *main*), di costruire tutti i container e rilasciarli su *DockerHub*, pronti per essere scaricati.

Domain Driven Design. Applicare questo approccio sin dall'inizio dello sviluppo ci ha portato sicuramente a esplorare e comprendere meglio l'intero spazio del problema, esaminando ogni singolo aspetto che avremmo dovuto includere nel sistema. Abbiamo imparato che questo approccio riesce a portare il team verso una soluzione più semplice, scomponendo di volta in volta il problema fino ad arrivare al fulcro della questione. Questo ci ha fatto capire quanto siano importanti tutte le fasi di analisi e quanto sia altrettanto importante eseguire ogni fase in modo rigoroso, senza mai dare nulla per scontato.

Architettura a microservizi. Prima d'ora non avevamo mai realizzato un sistema distribuito basato su microservizi. Eravamo già a conoscenza

dell'architettura a livello teorico, ma non avevamo mai avuto la possibilità di mettere poi in pratica le conoscenze apprese. Nel caso di questo progetto l'abbiamo fatto e ne siamo soddisfatti.

Test. La fase di test si è rilevata fondamentale durante lo sviluppo. Con questo progetto ci siamo davvero resi conto di quanto sia importante questa fase e di quanto sia importante svolgerla correttamente. Spesso, durante lo sviluppo del contesto di un'asta, capitava che quando si andava ad aggiungere una nuova funzionalità poi i test passati fallissero. Quindi, nel caso in cui non fossero stati aggiunti, molto probabilmente all'aumentare delle funzionalità sarebbero aumentati anche i bug nel sistema. Perciò, il fatto di aggiungere i test di pari passo con l'aggiunta delle diverse funzionalità si è rivelata una strategia vincente.

JSON Web Token (JWT). In questo progetto abbiamo anche avuto la possibilità di approcciarci alla sicurezza, affrontando i temi di autenticazione e autorizzazione baste sullo standard JWT, approfondendo più nel dettaglio il suo funzionamento.

Specifica OpenAPI. Il fatto di aver realizzato i diversi livelli applicativi con Spring, quindi secondo un approccio RESTful, ha permesso di approcciarsi alle specifiche OpenAPI, potendo apprezzare la loro semplicità ma allo stesso tempo le molteplici funzionalità che si ottengono a partire solamente da un file Json (o yml).

Docker. Lo svolgimento del progetto ha anche permesso di familiarizzare con l'uso dei container su cui eseguire ogni microservizio. Essi si sono rivelati uno strumento fondamentale per semplificare il processo di deploy su ogni macchina, dal momento che permettono di eseguire l'applicazione su qualsiasi computer, senza il bisogno di impostare alcun ambiente di sviluppo: tutto ciò che serve è gia presente all'interno del singolo container.

Docker Compose. Di particolare utilità è stato anche Docker Compose, un tool per definire ed eseguire applicazioni Docker composte da più container. Con un file YAML è possibile specificare in modo semplice di quali servizi (i.e. container) si compone l'intera applicazione. Successivamente, con un singolo comando è possibile creare ed eseguire tutti i diversi servizi, secondo le modalità specificate nel file (e.g. il mapping delle porte, le variabili d'ambiente ed eventuali dipendenze).

In conclusione, il team si ritiene soddisfatto degli obiettivi raggiunti e del prodotto realizzato.