Web Auction System

Matteo Venturi
matteo.venturi7@studio.unibo.it

Dicembre 2021

Indice

1 Goal

Lo scopo del progetto è quello di sviluppare un prototipo di sistema distribuito asincrono concorrente per la gestione in tempo reale di aste sul Web. L'intento è quello di implementare un sistema che possa sostenere un elevato numero di client concorrenti, che sia scalabile così da adattarlo al variare del traffico e del carico di lavoro richiesto e distribuito così da poter garantire una buona resistenza ai guasti e velocità di ripristino in caso di malfunzionamenti.

1.1 Scenari

Lo scenario tipico di utilizzo di questo sistema sono le aste online. Un utente desidera vendere al prezzo più alto possibile un bene o servizio e crea un'asta con durata limitata. I partecipanti all'asta sono gli offerenti e questi sono gli utenti principali del sistema. Gli offerenti concorrono ad aggiudicarsi l'asta con offerte a rialzo: vince l'asta l'offerente che alla scadenza risulta aver fatto l'offerta maggiore.

1.2 Politiche di validazione

Il collaudo della piattaforma viene effettuato per mezzo di test automatici, sia unitari che di integrazione, con percentuale di copertura test maggiore dell'80%. I test comprendono lo scenario classico di utilizzo nonché la verifica delle principali operazioni di base che la piattaforma offre.

2 Analisi dei requisiti

Il sistema permette la creazione di aste, caratterizzate da un titolo, una descrizione, la durata e il prezzo di partenza.
Un'asta inizia nel momento in cui viene creata e con durata minima di un giorno e massima di trenta giorni, prezzo di partenza maggiore uguale a zero e titolo e descrizione valorizzati. Ogni asta è caratterizzata in qualsiasi momento da una e una sola offerta massima o al più nessuna offerta.
Per ogni asta è possibile inserire più offerte, caratterizzate dall'ammontare, dalla data di offerta e dal riferimento all'utente che effettua l'offerta. Un'offerta è valida solo se il suo valore è maggiore uguale a zero e l'asta a cui si riferisce non è terminata. Non esiste un limite massimo di offerte che un offerente possa fare.

In base agli scenari e all'analisi dei requisiti, vengono identificati quattro casi d'uso principali (fig. 1):

- Venditore crea un'asta
- Venditore/Offerente visualizza l'offerta massima di un'asta
- Offerente inserisce fa un'offerta per un'asta
- Offerente visualizza le aste in corso

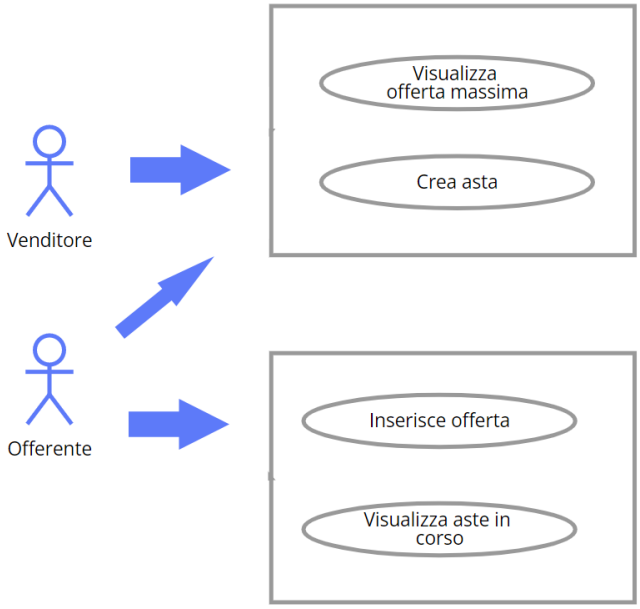

Figura 1: Casi d'uso

3 Design

Il sistema viene pensato a microservizi e distribuito, così da poter scalare all'aumentare del traffico richiesto.

Per garantire la massima velocità di gestione delle offerte il sistema utilizza una cache veloce come layer intermedio.
La presenza o meno del servizio di cache deve essere totalmente trasparente al client, questo richiede che il sistema implementi meccanismi di tolleranza al guasto della cache e garantisca la coerenza dei dati.

3.1 Struttura

Le entità software principali sono **Auction**, **Bid** e **Worker**.

L'entità **Auction** rappresenta la singola asta ed è caratterizzata dai seguenti campi:

- **Id**: identificativo univoco (guid)
- **Title**: titolo dell'asta
- **Description**: descrizione del bene o servizio in vendita
- **BasePrice**: prezzo di partenza
- **Start**: data e ora di inizio dell'asta

Figura 2: Diagramma delle classi

- **End**: data o ora di fine dell'asta
- **OwnerId**: identificativo univoco del proprietario del bene o servizio

L'entità **Bid** rappresenta la singola offerta fatta da un utente per un'asta ed è caratterizzata dai seguenti campi:

- **Id**: identificativo univoco (guid)
- **AuctionId**: identificativo univoco dell'asta a cui l'oferta è riferita
- **BidderId**: identificativo univoco dell'utente che ha fatto l'offerta
- **Value**: quantità di soldi offerta
- **When**: data e ora in cui è stata fatta l'offerta

La relazione che lega l'entità Auction e l'entità Bid è una relazione 1-n: un'asta può possedere n offerte e un'offerta è legata da una sola asta.(fig. 2)

L'entità **Worker** è un'entità di puro comportamento e interazione e quindi la sua struttura è trascurabile.

3.2 Comportamento

L'entità **Auction** si può trovare in due stati distinti: in corso e terminata. Quando data e ora dell'attributo End si trovano nel futuro allora l'asta è in corso, altrimenti l'asta è terminata. Lo stato dell'entità **Auction** non è storicizzato nell'entità va viene dedotto in tempo reale a ogni richiesta. Questo approccio assicura che la situazione di un'asta sia sempre coerente con la situazione reale e non crea disallineamenti.

L'entità **Bid** è sempre valida e non possiede uno stato interno. Sarà il sistema a individuare al bisogno quale sia l'offerta maggiore in un dato momento.

L'entità **Worker** è un'entità reattiva e che si occupa di storicizzare sul database le entità **Bid**.

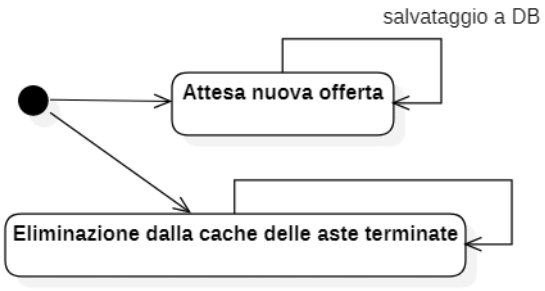

Figura 3: Diagramma di attività del worker

3.3 Interazione

Le entità **Auction** e **Bid** sono entità di storicizzazione dati: interagiscono con il database e con i client che utilizzano il sistema.

L'entità **Worker** interagisce unicamente con il database per poter portare a compimento il suo compito.

4 Implementazione

L'interfaccia Web è il punto di accesso al sistema per i client. Tramite i suoi endpoint i client possono interagire con le due risorse principali: aste e offerte.

La cache memorizza temporaneamente le aste in corso e le relative offerte, in modo da abbassare i tempi di latenza di lettura e scrittura nei casi di inserimento nuova offerta, lettura delle aste in corso, lettura delle offerte di un'asta in corso e lettura dell'offerta massima per un'asta in corso.
Nel caso la cache non fosse funzionante, il sistema continua a funzionare utilizzando come fonte il database.

Il database storicizza in modo permanente tutte le aste e relative offerte. La storicizzazione dell'asta avviene in modalità sincrona dato che la creazione dell'asta è un'operazione non critica e comunque necessaria per l'avvio dell'asta stessa. Altra cosa è la storicizzazione delle offerte che avviene in modalità asincrona.

Il worker è quell'entità perennemente in ascolto di nuove offerte e si occupa di storicizzarle in maniera asincrona.
Altro compito del worker è quello di eliminare periodicamente dalla cache le aste scadute in modo da mantenere la cache il più libera possibile e aggiornata.(fig. 3)

L'interfaccia Web permette in modo trasparente di reperire informazioni sia di aste in corso che di aste scadute: nel caso delle aste in corso le informazioni verranno caricate dalla cache, nel caso delle aste scadute saranno caricate dal database.

Tutte le entità del sistema interagiscono tra loro, direttamente e indirettamente, tramite comunicazioni di rete. Nello specifico, l'interfaccia Web interagisce sia con il worker in maniera indiretta tramite la cache, sia con il database per caricare le informazioni

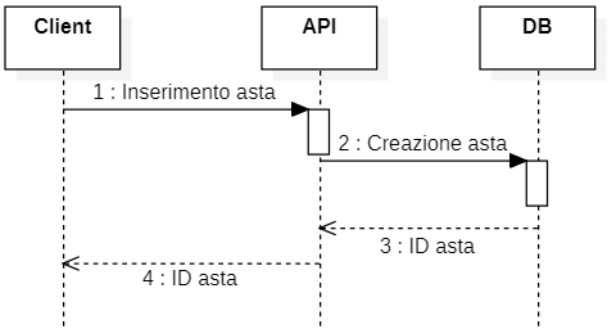

Figura 4: Diagramma di sequenza di creazione asta

sulle aste terminate e le loro offerte, sia con la cache per caricare le aste in corso e relative offerte.
Vengono riportate le interazioni principali: creazione nuova asta (fig. 4) e salvataggio nuova offerta (fig. 5)

Il codice sorgente viene memorizzato su repository Gitlab
https://gitlab.com/pika-lab/courses/ds/projects/ds-project-venturi-ay2122

Lo stack tecnologico scelto è .NET 5.0 con linguaggio C#, database PostgreSQL su modello relazionale e cache chiave-valore Redis. .NET 5.0 è un framework maturo, prestazionale e che può essere eseguito sui maggiori sistemi operativi moderni. Il linguaggio C# scelto offre notevoli vantaggi in termini di espressività e programmabilità. Con questa tecnologia saranno implementati tutti i componenti del sistema.

Il database PostgreSQL è un database ad oggetti che può essere utilizzato sia con modello relazionale classico sia con modello a oggetti. Viene scelto per la sua versatilità ed espandibilità.

Redis è genericamente un deposito di dati residente in memoria RAM. Viene scelto per la sua velocità, oltre che per le sue caratteristiche di scalabilità e resilienza.

L'architettura finale del sistema si compone di quattro parti principali identificate dalle API WEB di interfacciamento con i client, il database, la cache e il worker.(fig. 6)

Dal punto di vista dell'organizzazione del codice sorgente (fig. 7), la soluzione si articola in quattro progetti:

- WebAuction.API.FE: racchiude le API REST
- WebAuction.Worker: racchiude il worker che esegue le operazioni asincrone di storicizzazione dati
- WebAuction.Shared: racchiude codice comune come modelli, servizi e business logic
- WebAuction.Test: racchiude il codice dei test unitari e di integrazione

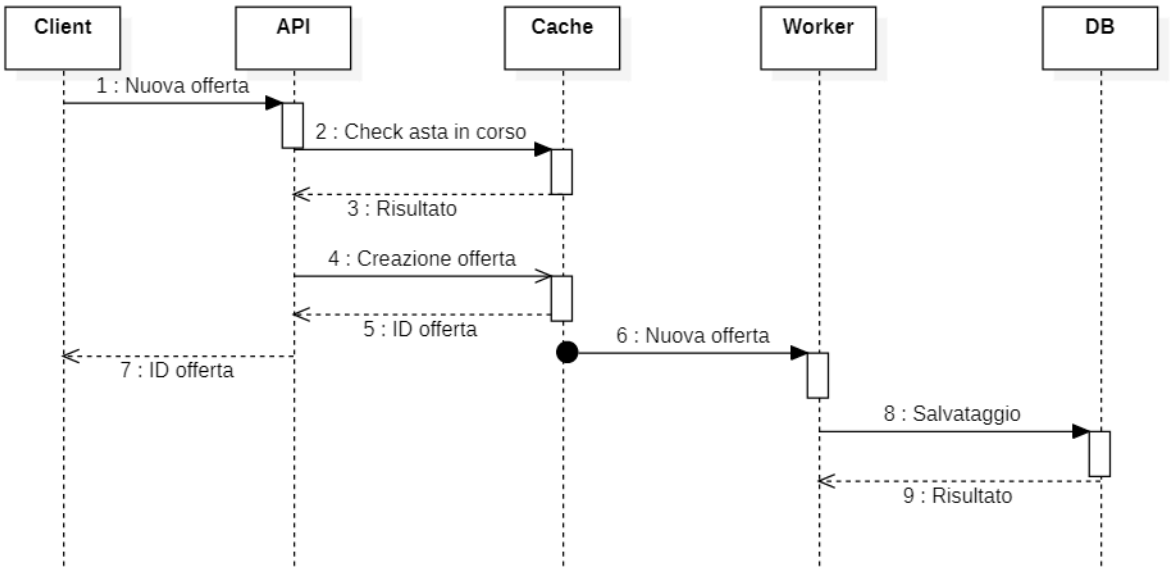

Figura 5: Diagramma di sequenza salvataggio offerta

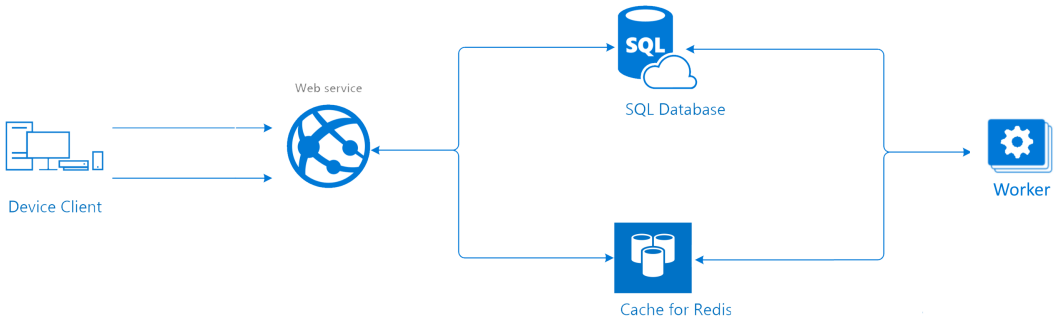

Figura 6: Struttura del sistema

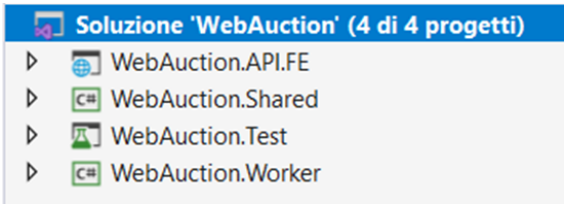

Figura 7: Organizzazione del codice

L'interfaccia Web è di tipo Restful ed è fruibile per mezzo di documentazione Swagger (fig. 8) all'indirizzo http://194.99.20.8:5950/swagger/index.html
Di seguito gli endpoint implementati:

- POST /auction Permette l'inserimento di una nuova asta. Questa operazione inserisce in maniera sincrona nel database le informazioni dell'asta e restituisce il risultato dell'operazione con l'identificativo associato all'asta.
- GET /auction/id Permette di reperire le informazioni complete dell'asta fornita in input. Questa operazione recupera unicamente i dati storicizzati a database. Per tale motivo può esistere un momento relativamente breve in cui la lista di offerte reperita con tale endpoint non combacia con la lista restituita dall'endpoint /bid/all/auctionId
- GET /auction/running Permette di avere la lista di tutte le aste in corso con relativi dati, a meno della lista delle offerte. Questa operazione legge dalla cache la lista delle aste in corso.
- POST /bid Permette l'inserimento di una nuova offerta ad un'asta in corso. Nel caso l'asta non esista o sia terminata allora viene restituito un errore, altrimenti viene restituito l'identificativo dell'offerta con la data e ora.
- GET /bid/all/auctionId Permette di ricevere la lista di tutte le offerte fatte per l'asta fornita in input. Questa operazione carica l'asta e le relative offerte dalla cache nel caso in cui l'asta sia in corso, altrimenti provvedere a caricare tutti i dati dal database. Al chiamante vengono restituiti i dati di un'asta, compresa la lista delle offerte fatte.
- GET /bid/highest/auctionId Permette di sapere l'offerta più alta per una data asta. Questa operazione legge direttamente dalla cache l'offerta maggiore per l'asta fornita in input.

5 Validazione

Viene implementato un progetto di test che racchiude una serie di test sia unitari che di integrazione. I test terminano con successo sia nel caso in cui tutti i servizi siano attivi

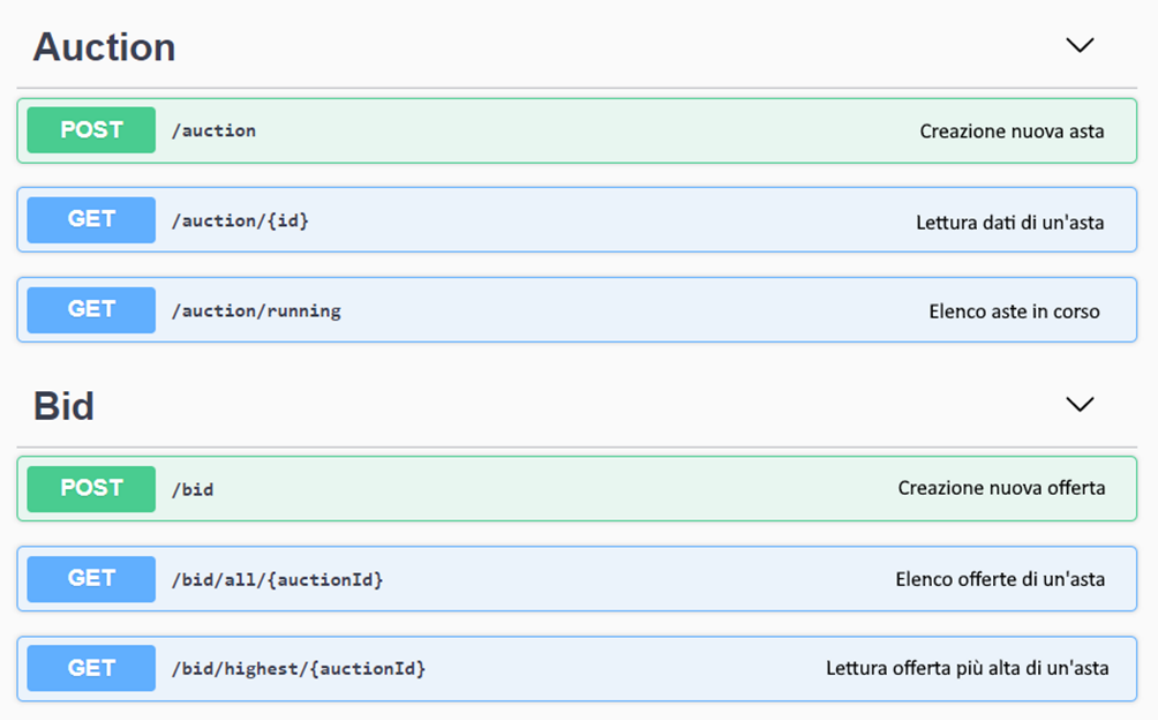

Figura 8: Documentazione Swagger

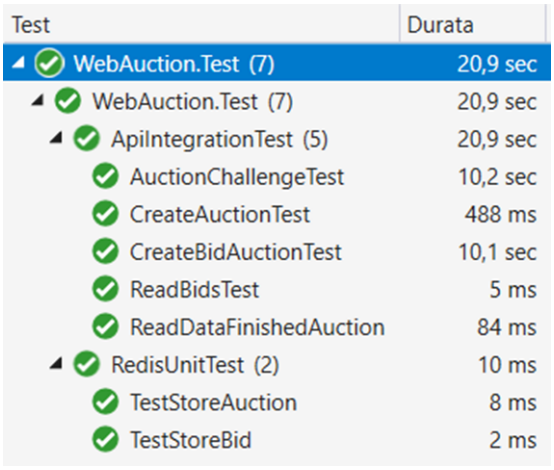

Figura 9: Test

e funzionanti, sia nel caso in cui il servizio di cache non sia funzionante. All'interno del progetto WebAuction.Test si possono trovare i seguenti test: (fig. 9)

- Creazione di una nuova asta
- Inserimento di una nuova offerta
- Inserimento offerte multiple (test di concorrenza)
- Lettura di tutte le offerte di un'asta
- Lettura delle aste in corso
- Lettura delle aste terminate
- Test di funzionalità cache per aste e offerte

La percentuale di copertura del codice raggiunge l'85%, quindi soddisfa ampiamente i requisiti di validazione. (fig. 10)

Durante lo sviluppo dei test si è reso necessario implementare un meccanismo di sincronizzazione tra cache e database. Il fatto che il servizio di cache potesse essere distrutto e ricreato a piacimento ha fatto emergere la necessità di un riallineamento tra database e cache ad ogni avvio del sistema. Pertanto, ogni qual volta le API si avviano, prima di iniziare a servire le richieste dei client, si assicurano che le aste in corso, e relative offerte fatte, siano memorizzate correttamente nella cache.

6 Deploy

La soluzione fa uso di Docker per il confinamento e gestione delle quattro entità che compongono la soluzione: viene utilizzato un container per il database PostgreSQL, per Redis, per le Web API REST e per il worker.

Gerarchia	Blocchi non analizzati	% blocchi non analizzati	Blocchi analizzati	% blocchi analizzati
matte_DESKTOP-TA1E8SL 2022-01-03 23_31_25.coverage	224	14,78%	1292	85,22%
webauction.api.fe.dll	121	21,53%	441	78,47%
webauction.shared.dll	49	17,82%	226	82,18%
webauction.test.dll	14	2,50%	546	97,50%
webauction.worker.dll	40	33,61%	79	66,39%

Figura 10: Copertura dei test

Per il database e la cache vengono utilizzati container ufficiali, già pronti all'uso e senza particolari configurazioni iniziali.

I container delle API REST e del worker vengono assemblati utilizzando lo strumento docker-build, basandosi sulle immagini ufficiali che forniscono l'ambiente Microsoft .NET 5.0

Infine, l'ambiente viene assemblato e configurato per mezzo dello strumento docker-compose che crea la sottorete dedicata e istanzia i container necessari che compongono la soluzione.

Da notare che la stessa procedura di preparazione dell'ambiente viene fatta anche per la fase di test, descritta in dettaglio più avanti.

Il processo di deploy viene configurato su Gitlab, repository scelto per ospitare il codice sorgente del progetto, per mezzo del file di configurazione .gitlab-ci.yml
Ad ogni push la configurazione di CI/CD prevede i seguenti passaggi:

- Impostazione di variabili di ambiente utili alla copilazione
- build del container worker
- build del container di front-end API REST
- preparazione dell'ambiente di test
- lancio dei test
- eliminazione dell'ambiente di test
- preparazione dell'ambiente di produzione
- pulizie finali

Ad ogni push sul repository, il server GitLab avvia il deploy sul server target sul quale è in esecuzione un GitLab Runner. Le fasi di avanzamento del deploy sono visibili direttamente su Gitlab. (fig. 11)

Per eseguire la soluzione in locale è necessario installare Visual Studio 2019, il framework .NET 5.0, Docker Desktop con istanziati i contaiener citati sopra. La soluzione è già configurata in modo da puntare in locale nel caso di avvio da Visual Studio. Se si desidera personalizzare i puntamenti è necessario configurare i file appsettings.json presenti in tutti i progetti della soluzione.

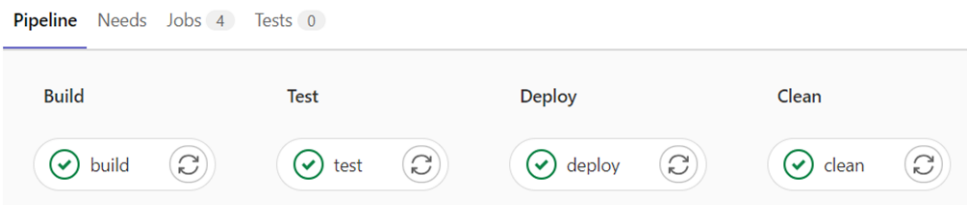

Figura 11: Fasi di deploy

7 Esempi di utilizzo

È possibile testare la soluzione per mezzo del front-end di documentazione Swagger all'indirizzo http://194.99.20.8:5950/swagger/index.html (fig. 12)
Cliccando su un'operazione è possibile compilare il modello della richiesta e avviare la richiesta.
Ad esempio, cliccando sulla POST nella sezione Auction è possibile inserire una nuova asta nel sistema. (fig. 13)

Un classico flusso di utilizzo è il seguente:

1. Creazione asta: POST /auction
2. Inserimento offerte: POST /bid
3. Lettura di tutte le offerte memorizzate nella cache: GET /bid/all/auctionId
4. Lettura di tutti i dati dell'asta (offerte comprese) memorizzati a DB: GET /auction/id
5. Verifica offerta più alta: GET /bid/highest/auctionId
6. Lettura di tutte le aste in corso: GET /auction/running

8 Conclusioni

Le scelte di design e implementative attuate portano allo sviluppo di un prototipo di back-end funzionante per la gestione delle aste online, seppur ancora con ampi margini di miglioramento e completamento.

L'implementazione del layer di cache garantisce basse latenze di risposta ai client.

L'utilizzo dell'architettura a microservizi e di Docker permettono di scalare la soluzione nei suoi componenti, soprattutto la parte di interfacciamento WEB e di cache, e di garantire un certo grado di resistenza ai guasti potendo attuare strategie di recovery sui singoli container.

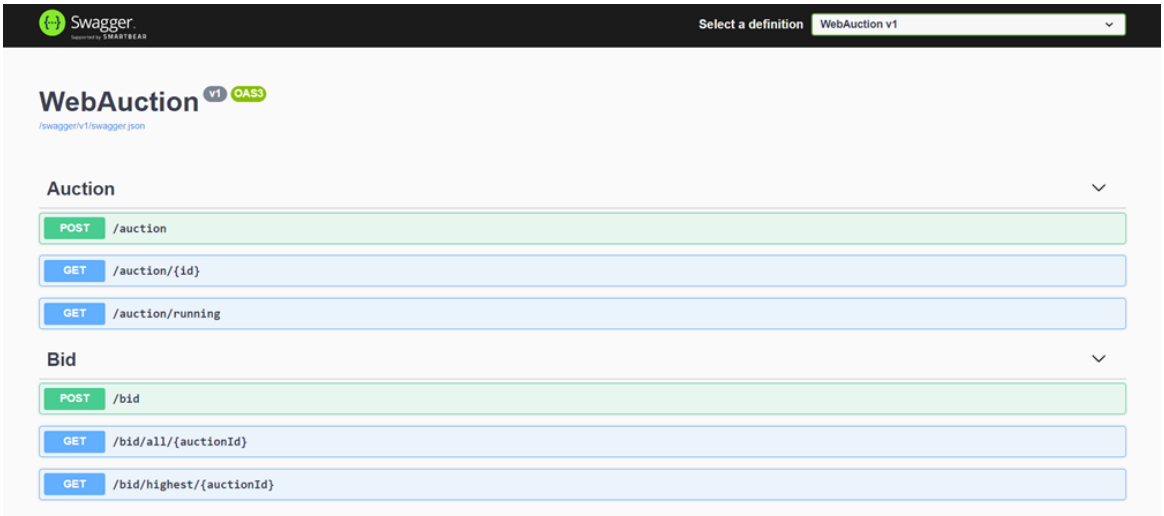

Figura 12: Endpoint implementati

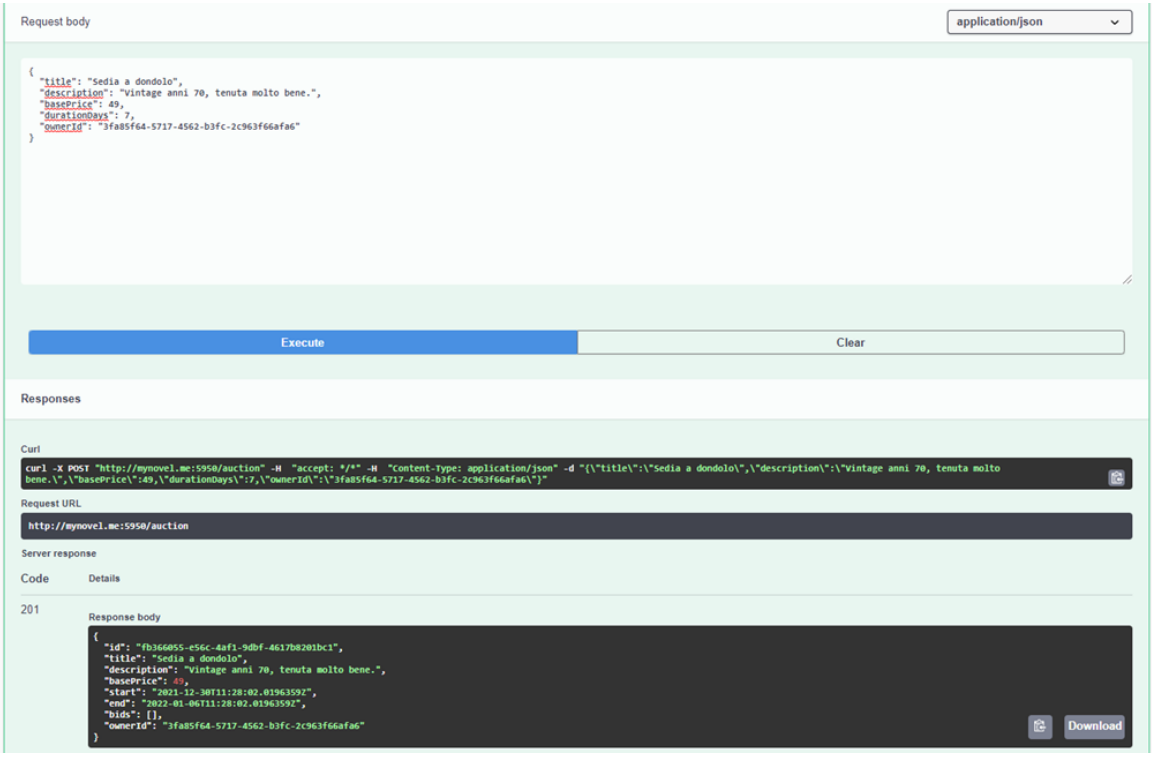

Figura 13: Richiesta inserimento nuova asta

8.1 Sviluppi futuri

Gli endpoint offerti dall'interfaccia Web API REST sono minimali e limitati. Uno sviluppo futuro è sicuramente quello di estenderli sia come numero per poter offrire maggiori funzionalità (es. utenti) sia come usabilità/performance (es. paginazione per le aste in corso).

Esistono anche punti dove l'implementazione può essere migliorata. Uno di questi è sicuramente la parte di comunicazione tra le Web API e il worker. Al momento viene impiegata una coda in Redis con pattern produttore/consumatore: le API inseriscono una nuova offerta, il worker legge l'offerta e la storicizza a database. Questo punto ha creato notevoli problematiche a livello implementativo causa difficoltà di gestione da parte di una prima libreria utilizzata. Una importante miglioria è sicuramente il refactor di questo punto introducendo l'utilizzo di RabbitMQ e una coda dedicata.

Altro sviluppo futuro sicuramente da implementare è la gestione della consistenza tra database e cache nel caso in cui la cache non sia raggiungibile. Ad oggi, nel caso in cui il servizio di cache diventasse improvvisamente irraggiungibile, il sistema riesce a sopperire a tale mancanza e a non perdere dati o restituire errori ai client andando ad operare direttamente sul database. Manca un meccanismo di riallineamento dati al momento del ritorno in funzione della cache. Tale meccanismo è essenziale per avere un sistema che offra dati coerenti anche in caso di guasti alla cache.

8.2 Concetti appresi

Questo progetto mi ha permesso di mettere in pratica aspetti e tecnologie viste durante il corso di Sistemi Distribuiti quali il design di un sistema distribuito, l'implementazione con strumenti quali Docker, la valutazione di tecnologie come RabbitMQ e i concetti di resilienza e tolleranza ai guasti tentando di gestire il guasto del servizio di cache.