

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Corso di
Applicazioni e Servizi Web
Docenti: Silvia Mirri, Roberto Girau

TRADING VIRTUAL GOODS

Autori:

Matteo Brocca · 1005681
Alan Mancini · 1005481
Mattia Achilli · 979641

Settembre 2021

Indice

1	Introduzione	2
2	Requisiti	3
3	Design	4
3.1	Metodologie di sviluppo	4
3.2	Archittettura del sistema	6
3.3	Interfaccia utente	8
3.3.1	Target Users	8
3.3.2	Mockups/Prototipo	10
4	Tecnologie	11
4.1	MERN Stack	11
4.2	Sviluppo	13
4.3	BackEnd	13
4.4	FrontEnd	17
4.5	Blockchain	18
5	Codice	19
5.1	BackEnd	19
5.2	FrontEnd	24
6	Test	26
6.1	Codice	26
6.2	User Experience	29
7	Deployment	30
8	Conclusioni	31

1 Introduzione

Realizzazione di un portale web per le aste in tempo reale di NFT (Non-Fungible Token) per lo scambio di proprietà digitali.

Un NFT è un'unità di dati archiviata su un registro digitale, chiamato blockchain, che certifica l'univocità di un asset digitale quindi non intercambiabile. [1]

Gli NFT possono essere utilizzati per rappresentare elementi digitali il cui accesso non è limitato all'acquirente. Sebbene le copie di questi elementi digitali siano disponibili per chiunque, i Non-Fungible Token vengono tracciati su blockchain per fornire al proprietario una prova di proprietà separata dal copyright. [4]

L'arte digitale è stata uno dei primi casi d'uso per le NFT grazie della capacità della tecnologia blockchain di assicurare la firma e la proprietà uniche delle NFT.

L'opera d'arte digitale intitolata "Everydays - The First 5000 Days", dell'artista Mike Winkelmann, noto anche come Beeple, è stata venduta per 69,3 milioni di dollari nel 2021.[2]

2 Requisiti

Il progetto si pone l'obiettivo di realizzare un portale web accessibile a chiunque, all'interno del quale utenti registrati possono creare e scambiarsi NFT.

Le opere caricate in formato digitale dovranno essere le protagoniste del sito web quindi si dovrà realizzare un'estetica dal design minimalista ma allo stesso tempo funzionale. Considerando che almeno il 55% degli utenti naviga sul web con dispositivi mobile [3], si dovrà inoltre prestare attenzione nello realizzare un'interfaccia responsive che si adatti a tutte le situazioni.

Specifiche funzionali:

- Registrazione utenti con possibilità di definire l'account su blockchain
- Accesso utenti con email e password
- Gestione di diversi ruoli utente: *amministratore* ed *utente base*
- Possibilità di creare nuovi NFT con *titolo*, *descrizione*, *immagine*, *categoria* e *tags* salvando le informazioni necessarie su blockchain.
- Possibilità di creare una nuova asta associata all'NFT con *prezzo di partenza*, *data di scadenza* e *descrizione*.
- Home page dove vengono visualizzati di tutti gli NFT, in vendita e non, con possibilità di ricerca
- Pagina dettaglio NFT e nel caso ci fosse un'asta attiva, possibilità di partecipare e visualizzare lo storico delle puntate
- Profilo utente con lista di NFT creati e posseduti
- Multilingua

Per quanto riguarda la scelta della tecnologia di blockchain da utilizzare si dovrà fare una fase di studio per trovare quella più adatta alle esigenze tra quelle che supportano gli NFT e smart contract.

3 Design

Design dell'architettura del sistema e delle interfacce Utente

3.1 Metodologie di sviluppo

La prima fase di design si è concentrata nella definizione delle metodologie di sviluppo in team, in questo caso composto da 3 persone.

L'obiettivo è stato quello di lavorare in modalità **Agile**, scomponendo tutto il progetto in piccoli task, ognuno dei quali era scelto e preso in carico da un membro in accordo con gli altri a seconda delle priorità.

Settimanalmente è stato fatto il punto della situazione attraverso video conferenze su Google Meet, ogni membro del team era aggiornato sullo stato dei task, le difficoltà riscontrate, i problemi risolti e le decisioni prese. In questa fase veniva aggiornato lo stato dei task (**To Do**, **Doing**, **Done**), eventualmente aggiunti o modificati in base alle esigenze emerse e si proseguiva scegliendo nuovamente i prossimi step da realizzare per avanzare.

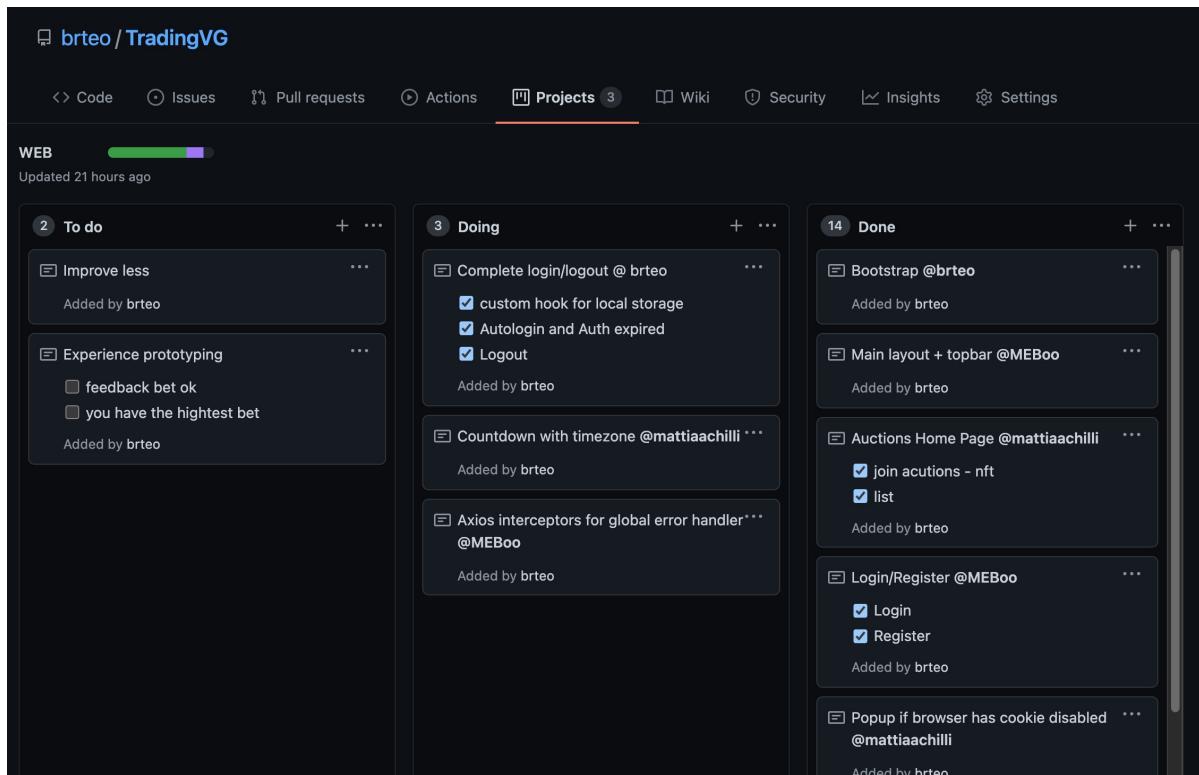


Figura 1: GitHub Projects

In questo modo siamo stati in grado di valutare progressivamente tutte le caratteristiche implementative ed accogliere i cambiamenti per tempo dal momento che buona parte delle tecnologie si sono studiate durante la realizzazione del progetto.

Volendo utilizzare un approccio **User Centered Design**, durante l'intera fase di design e sviluppo, ci si è focalizzati sui bisogni degli utenti al fine di creare un servizio che soddisfi gli utilizzatori. In mancanza di utenti finali reali, durante le riunioni settimanali, il team valutava le funzionalità e l'interfaccia che stava prendendo forma immedesimandosi nelle **Personas** definite inizialmente [3.3.1].

Come strumento software a supporto si è deciso di utilizzare GitHub Projects all'interno del repository del progetto.

Infine abbiamo deciso fin dall'inizio di utilizzare il metodo **TDD** (Test Drive development) per quanto riguarda lo sviluppo della RestApi, facendo in modo che tutti gli endpoint della RestApi siano opportunamente testati fin dall'inizio.

3.2 Archittettura del sistema

Dopo la definizione delle metodologie di sviluppo, ci siamo concentrati nella realizzazione dell'architettura del sistema. Abbiamo immaginato che il portale web sia in futuro distribuito in moderne piattaforme cloud a microservizi.

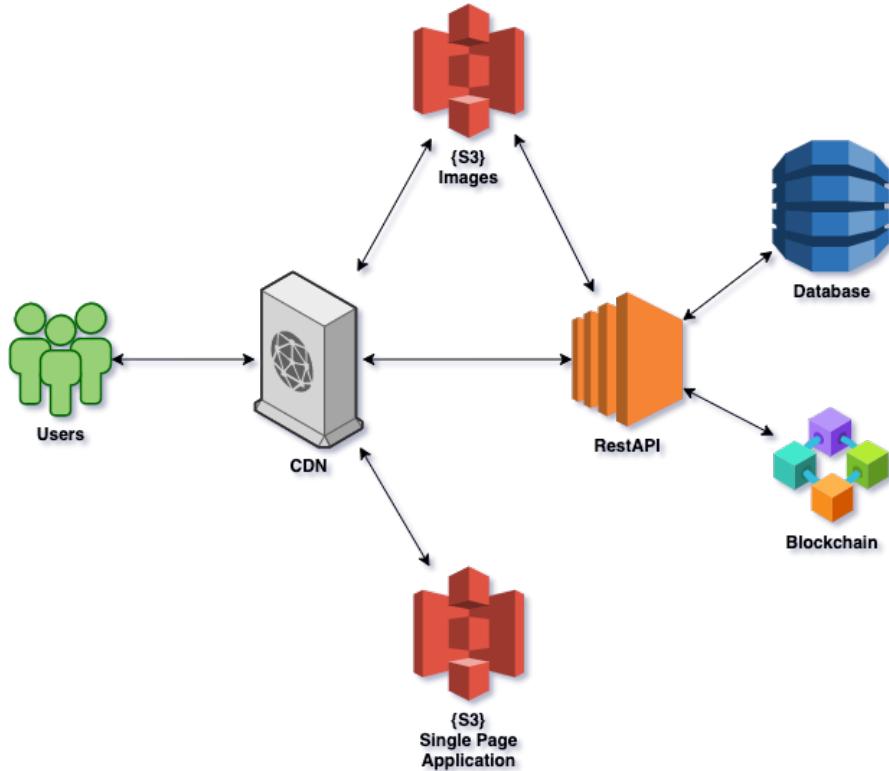


Figura 2: Architettura del sistema

Gli utenti che navigheranno all'indirizzo web del portale accederanno alle risorse attraverso una **CDN** (Content Delivery Network), una piattaforma di server altamente distribuita che aiuta a minimizzare il ritardo nel caricamento dei contenuti.

Il FrontEnd sarà una **SPA** (Single Page Application) ed in quanto tale potrà essere caricato su un Bucket **S3**. Le SPA sono applicazioni composte da una singola pagina non renderizzata lato server e contenente tutto il codice necessario (HTML, JavaScript e CSS) recuperato in un singolo caricamento al prima accesso alla pagina.

Le altre risorse che compongono il portale saranno caricate dinamicamente secondo le esigenze attraverso chiamate al BackEnd dove sarà realizzata una RestApi che mette a disposizione attraverso i suoi EndPoint le azioni di tipo CRUD (Create, Read, Update, Delete) utili a reperire e manipolare le informazioni salvate su **Database**.

In questo progetto specifico si dovrà connetere la RestApi anche ad una **Blockchain** per la creazione e gestione degli NFT. Le caratteristiche della Blockchain dovranno essere... quali Alan?

Infine le immagini saranno caricate su un'ulteriore Bucket **S3** per non appesantire la RestApi che deve restare sempre reattiva nello rispondere alle richieste degli utenti.

3.3 Interfaccia utente

Durante la fase di design abbiamo definito l’interfaccia utente e per tutta la durata del progetto l’abbiamo migliorata grazie ai meet settimanali dove ognuno di noi riportava i propri feedback, immedesimandosi nei target users definiti inizialmente, soprattutto nella fase finale quando il progetto prendeva sempre più forma e diventava un vero e proprio prototipo.

3.3.1 Target Users

Prima di partire con lo sviluppo, quindi nelle primissime fasi del progetto, abbiamo cercato di identificare i target users: il portale sarà utilizzato dagli appassionati di arte digitale, principalmente suddivisi tra **collezionisti**, **creatori di opere** o semplici **speculatori** che vogliono cavalcare il momento con l’obiettivo di arricchirsi.

Immaginiamo un target per lo più composto da giovani, compreso tra i 15 ed i 35 anni, ma non escludiamo anche persone più adulte appassionate di tecnologia. Con un target così giovane l’utilizzo di dispositivi mobile sarà sicuramente molto elevato, di conseguenza un’interfaccia responsive è fondamentale per permettere agli utenti di avere una User Experience appagante ed intuitiva.

Da questa riflessione iniziale sono state definite le seguenti **personas** utilizzate come riferimento per lo sviluppo dell’interfaccia.

Jhon

Jhon ha 30 anni ed è un collezionista appassionato di arte e di tecnologia. Potremmo considerarlo un ‘geek’, abita ancora con i genitori e la sua camera è piena di fumetti, edizioni limitate di videogiochi e film, console del passato ed almeno un poster di Star Wars.

Scenario d’uso: Jhon scopre il nuovo portale per lo scambio di NFT, sa già che cosa sono in quanto appassionato di tecnologia e vuole accedere quotidianamente per verificare la presenza di nuove opere di suo gusto così da partecipare alle aste per aggiudicarsene.

- Entra sul portale
- Controlla la homepage dove immagina di trovare le ultime novità
- Non soddisfatto effettua una ricerca sul tema che più lo appassiona
- Trova l’NFT che sogna e decide subito di effettuare una puntata
- La prima volta gli verrà richiesto di registrarsi ma lui non ha problemi
- Si registra

- Effettua la puntata

Tohoku

Tohoku ha 16 anni e le piace disegnare, lo fa di continuo, anche a scuola durante le lezioni che la annoiano. La ragazza non sa se farà l'università, non gli piace nessun lavoro, gli piace solo disegnare ma in questo periodo storico sa già che sarà difficile renderlo un lavoro.

Scenario d'uso: Tra gli amici di Tohoku gira voce di questi nuovi NFT, dei token digitali che scambiarsi opere digitali, ci sono persone che con i CryptoKitties hanno fatto un sacco di soldi! Tohoku sul treno di ritorno da scuola, tramite il suo smartphone cerca di informarsi e scopre il portale.

- Entra sul portale
- Naviga le pagine per farsi un'idea
- Vuole creare subito il suo NFT con l'ultimo disegno fatto a scuola anche se non sa niente della tecnologia
- Gli verrà richiesta la registrazione
- Crea il suo primo NFT

Mario

Mario ha 52 anni è un fotografo da una vita, ha viaggiato il mondo e lavora per qualche redazione importante. La fotografia è la sua passione ma a è al servizio di altri che lo pagano per fare reportage.

Scenario d'uso: Mario sa utilizzare il computer ed i social, visto anche il lavoro che svolge, ed un giorno legge degli NFT sul sole24ore. Ha subito una illuminazione: da quando è arrivato internet non ha mai saputo come proteggere le proprie foto pubblicate online, sul suo sito fatto dal nipote o sui social che detengono tutti i diritti, e finalmente pensa che questa nuova tecnologia può proteggere il suo copyright. Effettua subito delle ricerche sul suo iMac da 27 pollici e finisce sul portale.

- Entra sul portale
- Naviga le pagine per farsi un'idea
- Legge tutta la privacy e termini e condizioni d'uso del sito
- Si fida e vuole creare il suo NFT con la foto che più gli piace dell'ultimo anno ma che non aveva ancora pubblicato

- Gli verrà richiesta la registrazione
- Crea il suo primo NFT

dark02

Dark02 è il suo nickname, non sappiamo il suo nome e la sua età, sappiamo solo che è giovane ed un appassionato di tecnologia, naviga tutti i forum e cerca sempre il modo di cavalcare qualche moda per fare soldi, ne ha fatti tanti con i Bitcoin e CryptoKitties.

Scenario d'uso: Dark02 viene a conoscenza del nuovo portale in poco tempo, ha le sue fonti e subito si interessa è tra i primi a registrarsi. Viene spesso sul portale, vuole comprare partecipando alle aste con l'augurio che prendano valore, ma allo stesso tempo vuole creare i suoi NFT, riciclando illustrazioni fatte dalla ex ragazza.

- Entra sul portale
- Accede perchè già registrato
- Controlla la homepage
- Trova subito quello che vuole ed entra nel dettaglio
- Partecipa all'asta
- Cerca altri NFT dello stesso tipo
- Partecipa anche a questi
- Infine crea il suo NFT, na ha giò creati tanti.

3.3.2 Mockups/Prototipo

Immagini e flussi principali

4 Tecnologie

Tecnologie adottate e motivazioni.

4.1 MERN Stack

La scelta tecnologica per questo progetto, dopo la fase di design dell'archittettura è ricatuta sullo stack **MERN**, acronimo di **MongoDB**, **Express**, **React**, **NodeJS**.

Questo significa che la RestApi è un'applicazione **NodeJS** che, grazie alla programmazione Asincrona in linguaggio JavaScript, la rende estremamente reattiva alle richieste degli utenti rispetto altre tecnologie server-side come PHP ed ASP. Inoltre la natura Single Thread del linguaggio ci ha alleggerito dal compito di gestire la concorrenza, soprattutto in vista della gestione delle puntate delle aste che potrebbero avvenire da più utenti e molto ravvicinate prima della scadeza.

Il backend è supporto del pacchetto **Express**, un leggero e minimalista framework che mette a disposizione tutti gli strumenti necessari per una gestione delle richieste HTTP, così da pensare solo alla logica dell'applicazione.

I dati sono salvati su MongoDB, un database non relazionale molto reattivo ed adatto per veloci operazioni di Input/Output, altro tema molto importante per quanto riguarda le puntate dell'asta.

Dal momento che il FrontEnd è una SPA, la scelta poteva ricadere tra i tre framework più diffusi attualmente: React, Angular e Vue. La scelta per questo progetto è stata **React**, una libreria JavaScript per creare SPA fortemente incentrata sui componenti, per i seguenti motivi:

- Consideriamo Angular prolioso, genera una grande quantità di file come Java vincolando molto lo sviluppatore a rispettare la sua struttura. Inoltre ha una curva di Apprendimento maggiore rispetto gli altri.
- Vue è una buona soluzione ma più giovane rispetto gli altri due framework e quindi attualmente ha meno supporto e pacchetti rispetto a React (11M di download settimanali per React contro i 2M di Vue come da Figura 3 e 4). Questa è una considerazione importante da sviluppatore in quanto se ci si dovesse imbattere in qualche problema risulta più complesso trovare soluzioni online.
- A livello soggettivo apprezziamo maggiormente il metodo di lavoro di React.
- Come da Figura 5 React continua ad essere molto apprezzato dalla community, anche a distanza di anni.

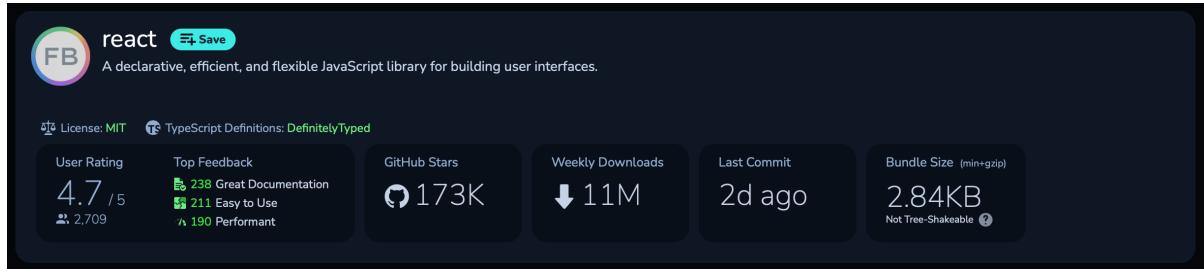


Figura 3: Statistiche React su [openbase.com](#)

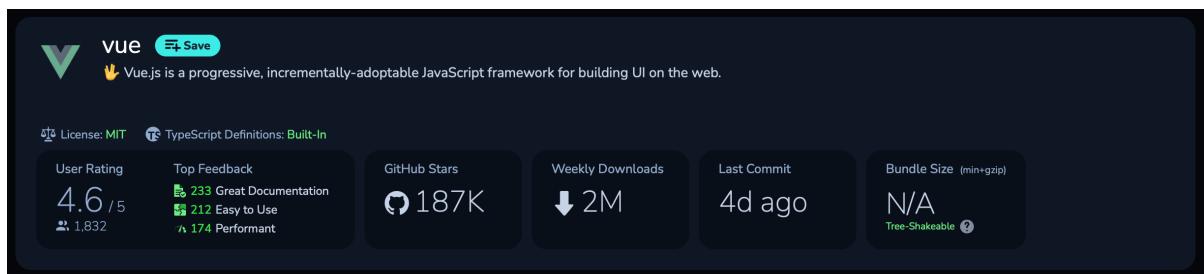


Figura 4: Statistiche Vue su [openbase.com](#)

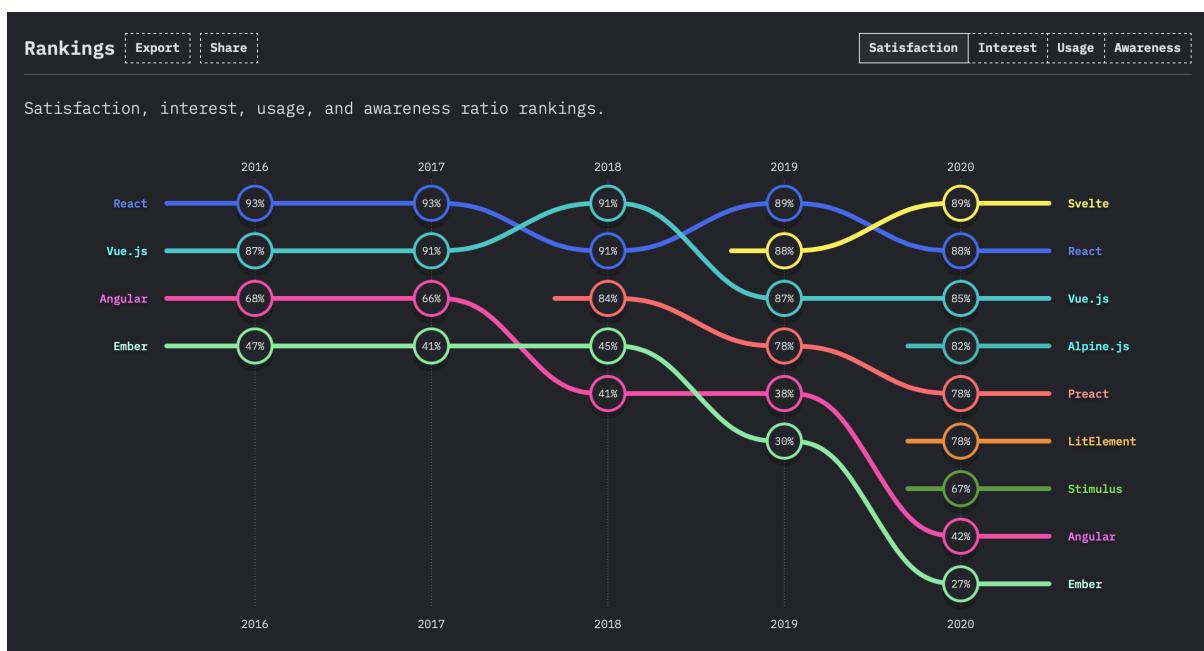


Figura 5: Comparativa Framework [2020.stateofjs.com](#)

4.2 Sviluppo

Sviluppando in locale, abbiamo ricreato l'archittettura del sistema attraverso **Docker Compose**, ignorando la CDN in quanto non necessaria durante lo sviluppo.

Attraverso Docker Compose si sono costruiti i seguenti container:

- **mongo**: database MongoDB versione 5.0.1-focal
- **api**: server contenente la RestApi in NodeJS
- **web**: server per la generazione del FrontEnd in React durante la fase di Sviluppo. In caso di Release sarà necessaria lanciare il comando build per generare tutti i file statici HTML, Javascript e CSS da caricare su S3
- **blockchain**: container generato ad hoc e pubblicato su DockerHub per lavorare con la blockchain EOS
- **localstack**: container per emulare i servizi di AWS (Amazon Web Services) del quale abbiamo utilizzato solo S3 per il caricamento delle immagini.

Lavorando su GitHub in team, si è deciso di utilizzare tutti **VScode** per scrivere codice, supportati dai pacchetti **ESlint** e **Prettier** per permetterci di formattare allo stesso modo il codice e guidarci nelle "best practice" della programmazione secondo l'estensione di **AirBnb**.

4.3 BackEnd

Pacchetti utilizzati a supporto per lo sviluppo della RestApi:

- **ajv**: validazione tramite JSON schema dei parametri e body inviati tramite le richieste alla RestApi
- **aws-sdk**: necessaria per interagire con S3 per il salvataggio delle immagini
- **bcryptjs**: libreria di criptazione utilizzata per salvare la password sul database
- **cookie-parser**: estensione di express per poter agevolmente estrarre i cookie dalle richieste, utile per l'autenticazione.
- **cors**: estensione di express per gestire il Cross-Origin Resource Sharing e proteggere la RestApi
- **crypto-js**: libreria di criptazione utilizzata per le chiavi pubbliche su blockchain
- **ejs**: linguaggio embedded javascript per la generazione di template html di email

- [email-templates](#): gestione dei template delle email
- [eosjs](#): libreria per comunicare con la blockchain
- [jsonwebtoken](#): generazione e validazione di JSON Web Token utilizzati per l'autenticazione
- [moment](#): libreria per la gestione delle date
- [mongoose](#): object modelling per il database MongoDB
- [nodemailer](#): libreria per l'invio delle email
- [passport](#): middleware per l'implementazione di vari sistemi di autenticazione dei quali abbiamo utilizzato "passport-local" per il login con credenziali salvate su database e "passport-jwt" per la validazione del JSON Web Token
- [socket.io](#): socket server per la comunicazione in tempo reale utilizzato per le puntate
- [sprintf-js](#): utility per aggiungere la funzionalità sprintf a NodeJs
- [uuid](#): libreria di generazione chiavi con standar uuid, utilizzata per la generazione dei refresh token

Al fine di testare le email inviate dalla RestApi, abbiamo utilizzato il servizio [MailTrap](#) che intercetta tutte le email inviate e le mostra in una casella di posta dove abbiamo potuto verificare la corretta visualizzazione sui vari dispositivi ed eventuali errori di compatibilità tra client (Firure 7 e 6).

Per quanto riguarda i test realizzati sul BackEnd si è deciso di optare per il pacchetto [Jest](#), supportato da [Supertest](#) e [Mongodb-memory-server](#). Supertest è un pacchetto utile per testare le chiamate ad una RestApi mentre Mongodb-memory-server permette di virtualizzare in memoria il database durante la fase di test direttamente, evitando così di effettuare i test nel database presente sul container docker utilizzato da noi sviluppatori per provare l'applicativo in locale ed inizializzato tramite seed.

Welcome to Trading Virtual Goods

[Email](#) [Delete](#) [More](#)

From: TradingVG <noreply@tradingvg.com>
To: <tom@meblabs.com>

2021-07-22 10:01, 470 KB
[Attachments \(2\)](#)

Show Headers

HTML

HTML Source

Text

Raw

Spam Analysis

HTML Check 18

Tech Info



Welcome to Trading Virtual Goods

Hi tom@meblabs.com,

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas ut luctus risus.

Cras ac nisl ultricies, gravida nunc ut, mollis metus. Fusce lacinia tempor nulla.

[CREATE YOUR NFT](#)



[Contact Us](#) / [Privacy Policy](#) / [Unsubscribe](#) / [Terms](#)

© TradingVG - All Rights Reserved

15



Figura 6: Visualizzazione email su MailTrap

Welcome to Trading Virtual Goods

From: TradingVG <noreply@tradingvg.com>
To: <tom@mblelabs.com>

2021-07-22 10:01, 470 KB
Attachments (2)

Show Headers

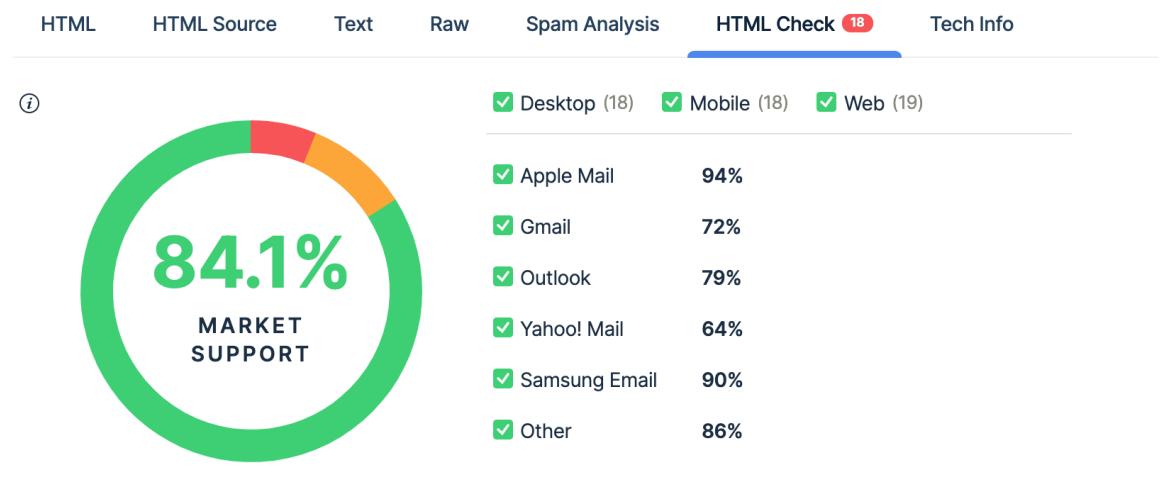


Figura 7: Compatibilità email su MailTrap

4.4 FrontEnd

Il progetto del FrontEnd è stato creato con [Create React App](#).

Pacchetti utilizzati a supporto del FrontEnd:

- [ant](#): componenti della User Interface
- [aws-sdk](#): necessaria per interagire con S3 per il salvataggio delle immagini
- [axios](#): per gestire le chiamate HTTP alla RestApi
- [craco](#): plugin della Create React App per permettere l'utilizzo di LESS come transpiler dei CSS (Non abbiamo utilizzato SCSS perchè Ant è basato su LESS)
- [i18next](#): libreria per la gestione del multilingua e che fornisce un apposito Hook per React (useTranslation)
- [moment](#): libreria gestione delle date
- [react-router-dom](#): plugin di React per la gestione dei link
- [socket.io-client](#): socket client per la comunicazione in tempo reale utilizzato per le puntate
- [sprintf-js](#): utility per aggiungere la funzionalità sprintf a NodeJs

4.5 Blockchain

Scelto EOS... perchè Alan?

5 Codice

Durante la fase iniziale di sviluppo ci siamo preoccupati di realizzare delle funzionalità lato BackEnd e FrontEnd che poi ci permettessero di standardizzare il metodo di lavoro e velocizzare la scrittura del codice, concentrando così sulla logica dell'applicativo.

Di seguito sono riportati alcuni gli aspetti che riteniamo più rilevanti lato codice.

5.1 BackEnd

Uniformate le risposte di express

Per prima cosa abbiamo cercato di uniformare tutte le risposte di Express per fare in modo di gestire agevolmente gli errori, creando un middleware che si occupasse di rispondere effettivamente alle chiamate.

In questo modo, all'interno dei vari controllers che gestiscono le chiamate ci siamo permessi di utilizzare una funzione che generasse la corretta risposta impostando i giusti parametri tra cui lo status code.

Alcuni esempi:

```
// invio errore generale con status code 500
next(ServerError());

// invio errore risorsa non trovata con status code 404
next(NotFound());

// invio dati richiesti in formato Json con status code 200
next(SendData({...}));
```

Autenticazione

Abbiamo realizzato da zero il processo di autenticazione con l'ausilio dei pacchetti **passport** e **jwt**.

Durante la fase di login, vengono inviate le credenziali email e password. Se le credenziali sono corrette vengono generati due JSON Web Token:

- **authToken**: JWT contenente nel payload l'identificativo utente su database (durata 5 minuti)
- **refreshToken**: JWT contenente nel payload il refresh token generato tramite uuid e salvato sul database in relazione all'utente (durata 30 giorni)

Per cercare la massima sicurezza ed evitare che il client salvi questi token riservati all'interno del browser, in posti accessibili, abbiamo deciso di utilizzare i cookie http-only: cookie settati dal server e che il browser storicizza in uno spazio di memoria non accessibile dall'esterno.

Una volta che il client è in possesso del authToker, questo viene inviato per ogni richiesta al server che, tramite un middleware creato ad hoc, verifica la correttezza nelle route degli endpoint che richiedono l'accesso per essere richiamati e, in caso di errore, viene automaticamente restituito l'errore 401 (*Unauthorized*) evitando di coinvolgere il controller ed interrompendo la richiesta.

```
// router.get('/check', isAuthenticated, check);
```

Quando l'authToken scade, il client può richiedere automaticamente un nuovo token di autenticazione utilizzando il refreshToken. Ogni utente può avere più di un refreshToken attivo in quanto è previsto che possa fare accesso da più dispositivi e quindi sono salvati su database sotto forma di array.

Infine è stato aggiunto un ulteriore controllo di sicurezza: nel caso venga inviato un refreshToken non più valido, l'utente viene bloccato in quanto è stato probabilmente soggetto di furto del token. Infatti l'unico modo per non avere un refreshToken valido è che qualcuno lo abbia utilizzato in precedenza e che quindi è stato rimosso sul database. Di conseguenza, se avviene una richiesta di questo tipo, significa che un malintenzionato ha rubato il token e provato ad utilizzarlo per ottenere accesso dopo che è stato già utilizzato e quindi invalidato dall'utente o viceversa, in entrambi i casi l'utente viene bloccato.

Role-Based Access Control

Sono stati realizzati ad hoc i Role-Based Access Control (RBAC) per gestire i permessi delle azioni CRUD della RestApi. Abbiamo previsto che esistano due tipi di utente: **admin** e **user**.

Ogni ruolo ha un file Json che descrive per ogni risorsa la relativa policy, definendo per ogni azione di tipo CRUD se ha il permesso di manipolare o richiedere qualsiasi informazioni ("any") o soltanto le proprie ("own").

```
// esempio ruolo "admin" per le risorse "users"
{
  "users": {
    "create:own": ["*"],
    "read:own": ["*"],
    "update:own": ["*"],
    "delete:own": ["*"]
  }
}
```

```
// esempio ruolo "user" per le risorse "users"
{
  "users": {
    "read:own": ["*"],
    "update:own": ["*", "!active"]
    // can't change active field
  }
}
```

A questo punto ci è bastato creare un middleware di controllo da inserire nelle richieste che necessitano di verifica del ruolo per essere eseguite. Questo middleware è sempre aggiunto in successione a quello dell'autenticazione in quanto la verifica del ruolo necessita sicuramente di un'autenticazione che valida l'utente che effettua la richiesta e, in caso di errore, viene automaticamente restituito l'errore 403 (*Forbidden*) evitando di coinvolgere il controller ed interrompendo la richiesta.

```
router.route('/')
  .get(isAuth, rbac('users', 'read:any'), controller.get);
```

Plugins Mongoose

Per rendere il lavoro sui controller più automatizzato possibile, abbiamo realizzato due plugin per Mongoose: **SoftDelete** e **dbFileds**.

Il plugin **SoftDelete** può essere aggiunto ad un Model del database per definire che la cancellazione di un documento di quella collezione è di tipo logica e non reale. Durante la progettazione del database, abbiamo deciso di applicare questa tecnica, su alcune collezioni del database, che ci permette di cancellare le risorse definendo i campi aggiuntivi *deleted* e *deletedAt*. In questo modo possiamo mantenere lo storico su database e ripristinare eventuali errori umani di cancellazione dati in quanto la risorsa non viene effettivamente cancellata.

```
const softDelete = require('../helpers/softDelete');
const { Schema } = mongoose;

const schema = Schema({
  // schema
});
schema.plugin(softDelete);
```

Per evitare di ricordare di aggiungere in ogni query sul database il filtro `"deleted: false"`, è stato automaticamente inserito in tutte le query del modello attraverso il plugin a meno che non sia esplicitamente definito dallo sviluppatore.

Per quanto riguarda il plugin **dbFileds**, ci siamo accorti durante lo sviluppo di volere un automatismo per definire quali campi sono pubblici e che quindi possono essere inviati

dalle risposte della RestApi. Ad esempio il campo *password* sulla collezione degli utenti non deve mai ritornare come risposta.

Di conseguenza abbiamo creato un plugin che ci permettesse di definire una o più liste di campi pubblici da poter selezionare durante le query:

```
const softDelete = require('../helpers/softDelete');
const { Schema } = mongoose;

const schema = Schema({
    // schema
});

schema.plugin(dbFields, {
    public: ['_id', 'nickname', 'pic', 'role', 'lang'],
    profile: ['_id', 'nickname', 'pic', 'header', 'bio'],
    cp: ['_id', 'email', 'account', 'nickname', 'name', 'lastname',
        'pic', 'lang', 'role', 'active']
});

```

Aggiunta di una nuova puntata sull'asta

Particolare attenzione è stata riposta all'aggiunta di una nuova puntata su un'asta attiva.

Nella progettazione del database, avevamo deciso di lasciare delle informazioni ri-dondanti all'interno dell'asta (collezione *auction*), salvando le ultime 10 puntate andate a buon fine. Ovviamente ognuna di queste puntate viene salvata anche nella relativa collezione *bets* con il riferimento all'asta.

Il problema principale era però l'eventuale concorrenza data da più richieste contemporaneamente che si sarebbe presentato qualora avessimo eseguito due query in successione:

- una per la verifica del prezzo della puntata, che deve essere maggiore di quello attuale
- una per l'effettivo aggiornamento della risorsa

Anche se JavaScript è un linguaggio asincrono single thread, non abbiamo la garanzia che ogni callback delle query abbia la giusta sequenza. Potrebbe capitare che viene lanciato due volte il controllo sul prezzo, da due client diversi, prima di inserire effettivamente la puntata e quindi permettere a due utenti diversi di effettuare anche la stessa puntata.

Per risolvere tale problema, abbiamo approfondito la documentazione di Mongoose dove abbiamo scoperto che l'operazione **findOneAndUpdate** ci viene garantita come atomica.

```
const auction = await Auction.findOneAndUpdate(
  { _id: auctionId, price: { $lt: betPrice } },
  {
    price: betPrice,
    $push: {
      lastBets: {
        $each: [newBets],
        $position: 0,
        $slice: 10
      }
    },
    { new: true }
  ).exec();
```

5.2 FrontEnd

Context

Tipicamente in React, i dati vengono passati dall'alto verso il basso (da genitore a figlio) tramite props, ma per alcune informazioni avevamo la necessità di renderle disponibili a tutti i componenti.

Per fare questo React ci permette di creare un *contesto* ed utilizzarlo tramite l'hook **useContext**. Quando un elemento del contesto viene modificato, tutti i componenti al suo interno vengono verificati ed eventualmente aggiornati, come avviene normalmente per lo stato o le props.

Abbiamo quindi realizzato un contesto che racchiude l'intera struttura del sito web dove condividere a tutti i componenti le seguenti informazioni/funzioni:

- **logged**: informazioni sull'utente che ha effettuato l'accesso al sito, se non presente significa che l'utente è un ospite
- **setLogged** funzione per settare l'utente collegato una volta che ha effettuato l'accesso
- **isMobile**: identifica se il dispositivo di visualizzazione è di tipo smartphone oppure no
- **handleLogout**: funzione effettuare il logout dell'utente
- **showLogin**: identifica se la popup di accesso/registrazione è visibile
- **setShowLogin**: funzione per l'apertura della popup di accesso/registrazione

Nel momento in cui il contesto viene inizializzato ed visualizzato sulla pagina, tramite l'hook **useEffect** viene effettuato l'autologin: se l'utente aveva già navigato sul sito ed aveva effettuato l'accesso senza eseguire il logout, allora probabilmente avrà un refreshToken (durata di 30 giorni) ancora impostato sui cookie http-only, di conseguenza viene automaticamente provato a richiedere un'autenticazione e, nel caso l'operazione vada a buon fine, l'utente si ritroverà loggato senza dover effettuare nuovamente l'accesso.

Inoltre, all'interno del contesto, abbiamo sfruttato gli *interceptors* di axios per intercettare automaticamente tutte le chiamate http di tutto il FrontEnd. Grazie a questa funzionalità abbiamo gestito la seguente casistica: se l'utente è loggato e riceve l'errore 401 che può trovarsi nella situazione che l'authToken è scaduto (la sua durata è di 5 minuti). In questo caso, viene automaticamente provata a richiedere una nuova autorizzazione tramite il refreshToken. Se questa va a buon fine allora viene nuovamente effettuata la chiamata originaria evitandoci di prevedere questa casistica in ogni chiamata http fatta dal FrontEnd.

Custom hook

Avendo la necessità di salvare alcune informazioni utili al sistema sul localstorage del browser, abbiamo deciso di creare un Custom Hook che ci permettesse di salvare le informazioni ed utilizzarle come lo stato all'interno dei componenti che lo necessitano.

```
const useLocalStorage = (key, initialValue) => {
  const [value, setValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (err) {
      return initialValue;
    }
  });

  const setStoredValue = newValue => {
    try {
      const valueToStore = newValue instanceof Function ? newValue(
        value) : newValue;
      setValue(valueToStore);
      window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) { }
  };

  return [value, setStoredValue];
};
```

6 Test

6.1 Codice

Lato codice è stata testata la RestApi. Di eseguito alcuni screenshot di esempio.

File	%Stmts	%Branch	%Funcs	%Lines
All files	86.95	77.46	76.52	86.95
controllers	85.98	72.29	84.09	85.98
auctions.js	95.7	76.92	87.5	95.7
auth.js	92.02	75	100	92.02
categories.js	100	66.67	100	100
nfts.js	70.09	74.36	71.43	70.09
profile.js	66.25	69.23	50	66.25
s3.js	100	100	100	100
search.js	50	100	0	50
tags.js	97.37	65.63	100	97.37
users.js	100	72	100	100
db	65.08	71.43	75	65.08
config.js	100	100	100	100
connect-test.js	94.59	83.33	100	94.59
connect.js	0	0	0	0
emails	81.16	77.78	75	81.16
index.js	81.16	77.78	75	81.16
helpers	75.37	87.95	67.35	75.37
auth.js	100	100	100	100
dbFields.js	68.75	83.33	75	68.75
eosjs.js	61.54	100	0	61.54
passport.js	96.4	86.49	100	96.4
rbac.js	86.36	55.56	100	86.36
response.js	100	100	84	100
s3.js	38.46	100	0	38.46
softDelete.js	81.25	100	66.67	81.25
middlewares	100	89.58	100	100
isAuth.js	100	92.31	100	100
lowercase.js	100	75	100	100
passport.js	100	100	100	100
rbac.js	100	100	100	100
response.js	100	80	100	100
trimmer.js	100	100	100	100
validator.js	100	87.5	100	100
models	95.12	50	50	95.12
auction.js	100	100	100	100
bet.js	100	100	100	100
category.js	100	100	100	100
nft.js	91.75	100	0	91.75
tag.js	100	100	100	100
user.js	92.47	50	66.67	92.47
routes	100	100	100	100
auctions.js	100	100	100	100
auth.js	100	100	100	100
categories.js	100	100	100	100
nfts.js	100	100	100	100
profile.js	100	100	100	100
s3.js	100	100	100	100
search.js	100	100	100	100
tags.js	100	100	100	100
users.js	100	100	100	100
schema	100	100	100	100
auction.js	100	100	100	100
auth.js	100	100	100	100
bet.js	100	100	100	100
category.js	100	100	100	100
nft.js	100	100	100	100
objectId.js	100	100	100	100
s3.js	100	100	100	100
tag.js	100	100	100	100
user.js	100	100	100	100
Test Suites:	9 passed	9 total		
Tests:	159 passed	159 total		
Snapshots:	0 total			
Time:	61.693 s			

Figura 8: Test coverage della RestApi

```

PASS  specs/auth.test.js (6.821 s)
POST /auth/login
  ✓ Missing credentials (191 ms)
  ✓ Invalid email (117 ms)
  ✓ Missing password (95 ms)
  ✓ Wrong email (100 ms)
  ✓ Wrong password (182 ms)
  ✓ Inactive account (103 ms)
  ✓ Deleted account (103 ms)
  ✓ Login successfully (194 ms)
GET /auth/check
  ✓ Check with valid token should be OK (190 ms)
  ✓ Check without token should be Unauthorized (121 ms)
  ✓ Check with invalid token should be Unauthorized (18 ms)
GET /auth/email/:email
  ✓ Check if email exist should be OK (166 ms)
  ✓ Check if email exist should be NotFound (9 ms)
  ✓ Check if deleted users email exist should be NotFound (127 ms)
  ✓ Check without email should be NotFound (15 ms)
  ✓ Check with incorrect email should be ValidationError (11 ms)
POST /auth/register
  ✓ Register new user with email and password should be OK and response with auth token + refresh token (139 ms)
  ✓ Register new user with email that already exist should be EmailAlreadyExists (98 ms)
  ✓ Register new user with nickname that already exist should be NicknameAlreadyExists (97 ms)
  ✓ Register new user with account that already exist should be AccountAlreadyExists (99 ms)
  ✓ Register new user with invalid account should be ValidationError (6 ms)
  ✓ Register new user without email should be MissingRequiredParameter (7 ms)
  ✓ Register new user without password should be MissingRequiredParameter (6 ms)
  ✓ Register new user with incorrect email should be ValidationError (7 ms)
  ✓ Register new user with not send lang filed has registered as "en" (106 ms)
  ✓ Register new user with supported lang "it" (108 ms)
  ✓ Register new user with not supported lang has registered as "en" (103 ms)
GET /auth/rt
  ✓ Get new auth with valid refresh token should be OK (215 ms)
  ✓ Get new auth with invalid refresh should be Unauthorized (100 ms)
  ✓ Get new auth with expired refresh should be Unauthorized (211 ms)
  ✓ Get new auth with valid refresh but already used token should be remove all refreshToken and set authReset (301 ms)
GET /auth/logout
  ✓ Destroy refresh token on logout and clear http only cookie (198 ms)

```

Figura 9: Test dell'endpoint /auth per l'autenticazione

```

PASS  specs/auctions.test.js (15.602 s)
Role: admin
  GET /auctions
    ✓ Get all and should contains an auction (659 ms)
    ✓ Get any specific auctionId (237 ms)
    ✓ Get with wrong auctionId should not be done (214 ms)
    ✓ Get deleted auction should not be found (327 ms)
  POST /auctions
    ✓ A new auction should be added (276 ms)
    ✓ A wrong auction should not be added (215 ms)
    ✓ An auction with inexistent nft should not be added (211 ms)
  PATCH /auctions
    ✓ Auction data should be changed (215 ms)
    ✓ Update with wrong data should not be done (204 ms)
    ✓ Update with wrong auctionId should not be done (243 ms)
    ✓ Update deleted auction should not be found (352 ms)
  DELETE /auctions
    ✓ Auction data should be deleted (216 ms)
    ✓ Any auctions should be deleted (361 ms)
    ✓ Soft deleted: after delete auction with GET does not return (205 ms)
Role: user
  GET /auctions
    ✓ Get all auctions should be Permitted (200 ms)
    ✓ Get a auctionId should be done (201 ms)
  POST /auctions
    ✓ Add new auction should be Permitted (205 ms)
  PATCH /auctions
    ✓ Update your own should be Permitted (212 ms)
    ✓ Update auctions of others users should be Forbidden (222 ms)
  DELETE /auctions
    ✓ Delete your own should be Permitted (234 ms)
    ✓ Delete auctions of others users should be Forbidden (224 ms)
Bets
  GET /auctions/:id/bets
    ✓ Get auction with invalid auctionID should be ValidationError (210 ms)
    ✓ Get auction with not existent auctionID should be NotFound (197 ms)
    ✓ Get auction bets should be empty (210 ms)
  PUT /auctions/:id/bets
    ✓ Add a new bet without auth should be Unauthorized (212 ms)
    ✓ Add a new bet with invalid auctionID should be ValidationError (228 ms)
    ✓ Add a new bet with not existent auctionID should be NotFound (214 ms)
    ✓ Add a new bet without price should be a MissingRequiredParameters (209 ms)
    ✓ Add a new bet with invalid price should be a ValidationError (197 ms)
    ✓ Add a new bet with price less or equal to current price should be a ValidationError (214 ms)
    ✓ Add a new bets should be ok (292 ms)
    ✓ Add a new bets should be update last bets on auction (345 ms)

```

Figura 10: Test dell'endpoint /auctions delle aste

6.2 User Experience

Fatto provare il prototipo sono emerse correzioni

7 Deployment

Il progetto è condiviso su un repository GitHub al seguente link:

<https://github.com/brteo/TradingVG/>

Clonare il progetto sul proprio dispositivo e seguire la indicazione.

Requisiti

- node and npm
- docker compose

Installazione

1. installare i pacchetti npm all'interno della cartella "api"

```
||  npm i
```

2. installare i pacchetti npm all'interno della cartella "web"

```
||  npm i --legacy-peer-deps
    # add flag --legacy-peer-deps with npm version > 7.0
```

3. avviare docker dalla cartella principale

```
||  docker compose up -d
```

4. una volta che docker è avviato, inizializzare i seed del database all'interno della cartella "api"

```
||  npm run seed
```

8 Conclusioni

Conclusioni

Riferimenti bibliografici

- [1] Sam Dean. \$69 million for digital art? the nft craze explained. *Los Angeles Times*, 2021.
- [2] Jacob Kastrenakes. Beeple sold an nft for \$69 million. *The Verge*, 2021.
- [3] StatCounter. Desktop vs mobile vs tablet market share worldwide. *StatCounter*, 2021.
- [4] Wikipedia. Non-fungible token. *Wikipedia*, 2021.