Foodio

Web-app di delivery

Sistemi Distribuiti
LM - Ingegneria e Scienze Informatiche

Lorenzo Pastore
lorenzo.pastore6@studio.unibo.it

October 2023

Contents

1 Introduzione

Si vuole realizzare una web-app per la consegna a domicilio di beni, che metta in diretto contatto clienti, esercenti e rider.
Nello specifico, un utente dovrà poter effettuare ordini presso i negozi nelle vicinanze e avere aggiornamenti sullo stato dell'ordine. Per facilitare la gestione degli ordini agli esercenti, questi potranno modificare il menù, dichiararsi non disponibili alla vendita e consultare lo storico degli ordini, il tutto tramite un'UI intuitiva.
Infine, i rider avranno la possibilità di ricevere ordini in real-time, visualizzando in anticipo il compenso per la consegna proposta. Se questa sarà accettata, verrà attivata la modalità navigazione, tramite la quale il rider visualizzerà il percorso per arrivare al punto di pick-up prima e dal cliente poi.

2 Requisiti

Di seguito vengono illustrati i requisiti definiti in fase di progettazione, che hanno guidato lo sviluppo dell'applicazione.

2.1 Requisiti Funzionali

Tutti gli utenti devono avere accesso alle pagine di registrazione e login. Previa registrazione, gli utenti potranno effettuare accessi distinti a seconda del ruolo associato alle credenziali.

2.1.1 Utenti Clienti

I clienti saranno in grado di:

- visualizzare la lista degli esercenti nelle vicinanze con i relativi menu'
- effettuare un ordine presso uno degli esercenti, pagando con un credito virtuale
- visualizzare lo stato dell'ordine (da accettare/ in elaborazione / in consegna)
- visualizzare la posizione del rider su mappa

2.1.2 Utenti Esercenti

I negozianti potranno:

- accettare o rifiutare un ordine
- modificare nome e/o prezzo di un elemento del menu'
- rendersi non disponibili ad accettare ordini
- visualizzare lo storico degli ordini in homepage

2.1.3 Utenti Rider

I rider, invece, potranno:

- interrompere il turno di lavoro se non in fase di consegna
- accettare o rifiutare un ordine
- visualizzare in anticipo il compenso di un ordine in sospeso
- visualizzare su mappa punto di ritiro e di consegna

2.2 Requisiti non funzionali

Si desidera un sistema che:

- Sia sempre online, anche in caso di guasti
- Sia intuitivo, minimale e user-friendly
- Sia di facile installazione (deployment)

3 Progettazione

In fase di progettazione è stato scelto lo stack *MEAN* come stack tecnologico di riferimento, si è, perciò, utilizzato il database document-based *MongoDB* per l'archiviazione dei dati, *Node.js* col framework *Express* per lo sviluppo del back-end e delle api e *Angular 2+* per implementare l'interfaccia grafica.
Inoltre si è fatto uso anche di altre librerie e tool di rilevante importanza per la riuscita del progetto, come *Redis*, *Socket.io*, *Leaflet* e *Docker*, il cui impiego verrà approfondito successivamente.

Gli approcci architetturali impiegati sono il modello *REST* (Representational State Transfer), con particolare riferimento allo sviluppo delle REST Api, e il *publish/subscribe*, utilizzato nello sviluppo dei middleware di componenti indipendenti del sistema.

3.1 Entity Interaction

Di seguito vengono riportati i sequence diagram dell'interazione fra *Clienti*, *Esercenti* e *Rider* in fase di ordine, nel caso in cui la consegna venga rifiutata o accettata dal negoziante.

Nel caso in cui un ordine venga accettato, questo verrà proposto al primo rider presente in coda. Se quest'ultimo rifiuta l'incarico, dovrà essere fatto un check sui rider online, per poi proporre l'ordine al prossimo in coda.

Quando invece il negoziante rifiuta, l'ordine verrà annullato, notificando il cliente della mancata conferma.

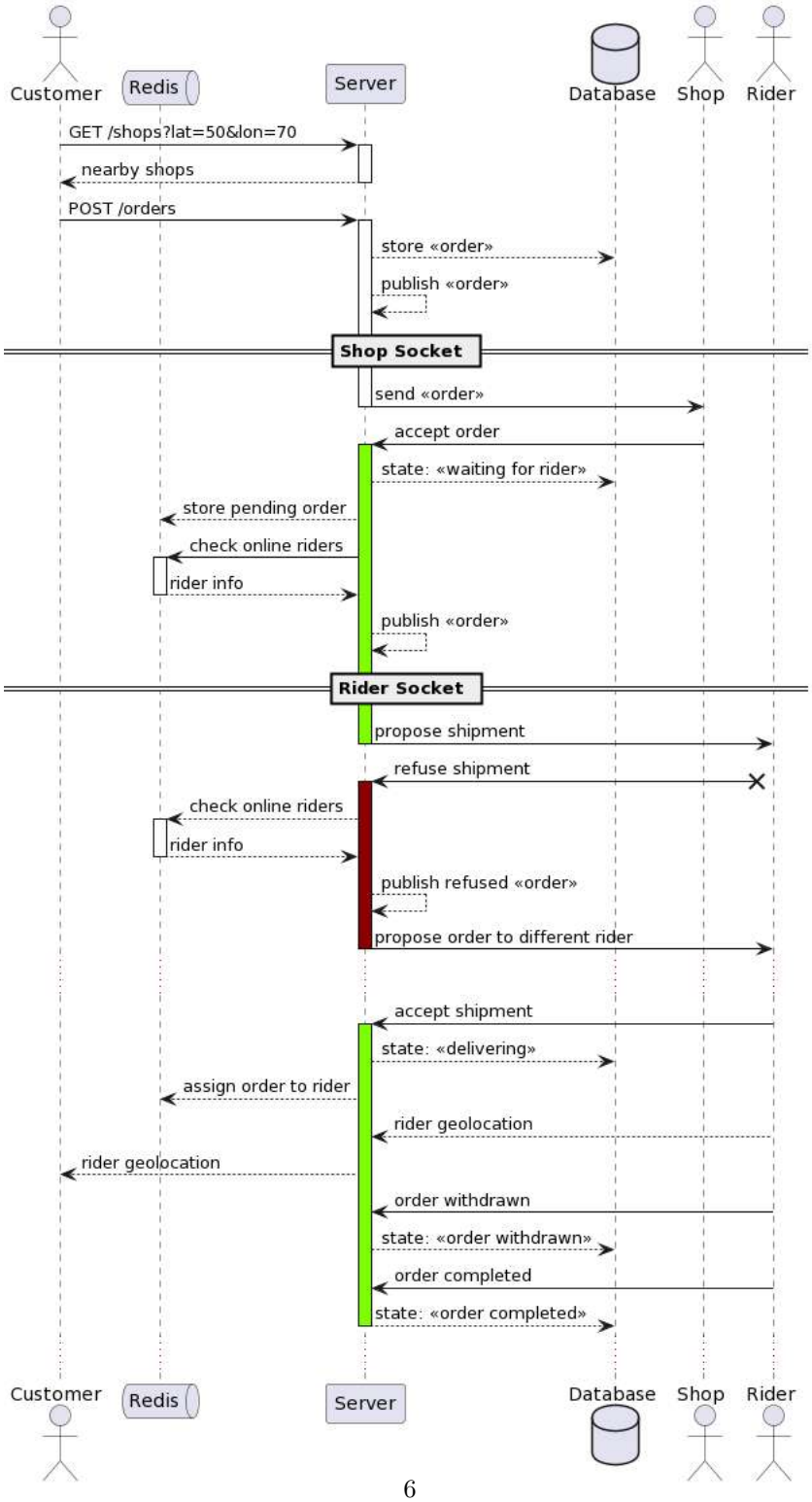

Figure 1: Successful delivery - Sequence Diagram

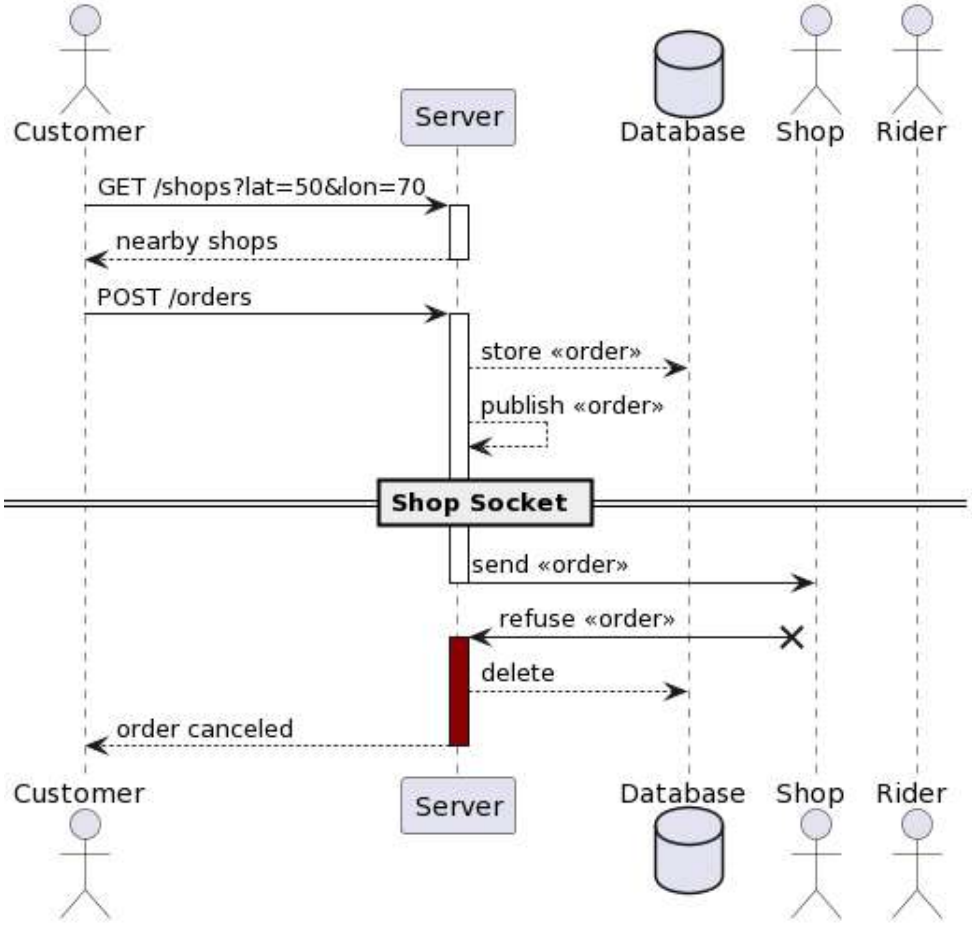

Figure 2: Delivery Refused - Sequence Diagram

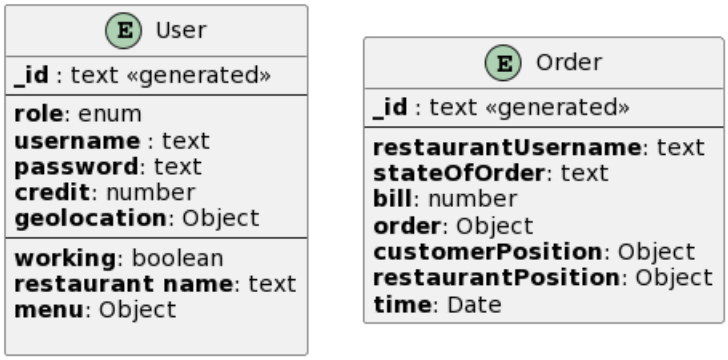

Figure 3: User and Order entities

3.2 Entities

Le entità individuate in fase di progettazione sono due:

- **User**: utenti registrati utilizzatori dell'applicazione, attribuibili a 3 diverse categorie:
  1. Cliente
  2. Ristoratore
  3. Rider
- **Order**: ordini effettuati da un cliente e presi in carico da un esercente e un rider

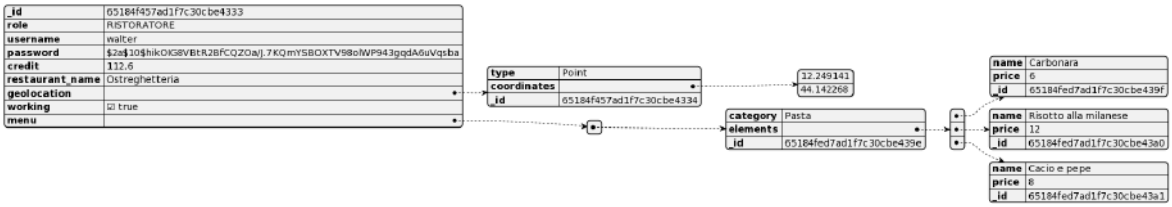

Figure 4: User data structure

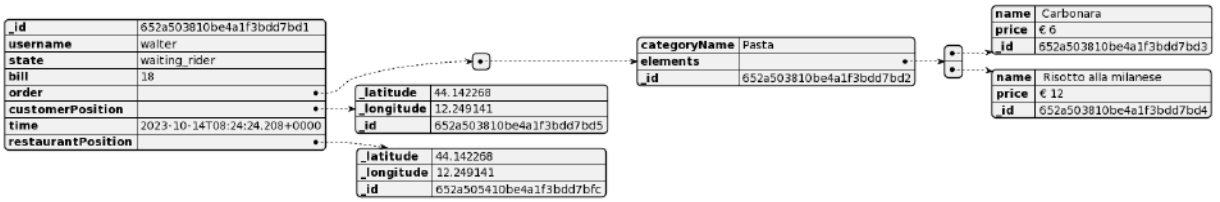

Figure 5: Order data structure

4 Implementation

4.1 Tecnologie Utilizzate

Dai sequence diagram di [3.1] si può evincere la scelta d'impiego di una tecnologia in particolare: il gran numero d'interazioni bidirezionali fra Client e Server ha reso necessario una libreria come *Socket.io*, sia per non appesantire il carico di lavoro, ma anche per garantire livelli di servizio efficienti e in real-time.

Un altro dettaglio tecnico che si evince dagli schemi è quello di servirsi di *Redis* come cache, nello specifico per storare informazioni temporanee come queue di rider online e ordini in corso, e anche da middleware servendosi del paradigma publish/subscribe, per consentire una comunicazione agnostica fra componenti software diverse come REST API e socket o anche fra *socket namespaces* differenti.

4.1.1 Node.js

Node.js è un runtime environment open-source, fondato sul motore Javascript V8 di Google Chrome. Node è un runtime single-threaded basato su eventi asincroni e operazioni I/O non bloccanti, consentendo, perciò, al server di ricevere migliaia di richieste, ideale per lo sviluppo di applicazioni web scalabili e con alto throughput.

4.1.2 Express.js

Express.js è un framework per Node.js, che rende lo sviluppo di api semplice e minimale. Grazie alle sue librerie per routing e middleware è ormai lo standard per la scrittura di endpoint HTTP in Node.

4.1.3 MongoDB

MongoDB è un database NoSQL document-based, che si contrappone alla classica struttura dei db basata sul modello relazionale. Questi ultimi non sono adatti per la maggior parte delle odierne web-app che richiedono bassa latenza, alto throughput e con frequenze di operazioni I/O elevate su quantità di dati non indifferenti. Si è quindi reso necessario un cambio di prospettiva, che consentisse la distribuzione dei db su più macchine, con un modello che valorizzasse le performance piuttosto che la consistenza.

4.1.4 Mongoose

Mongoose è una libreria per creare mapping fra oggetti che rappresenti la struttura dei dati da manipolare. Tramite istanze degli *Schema*, messe a disposizione da questa libreria, diventa immediato effettuare operazioni CRUD sui dati.

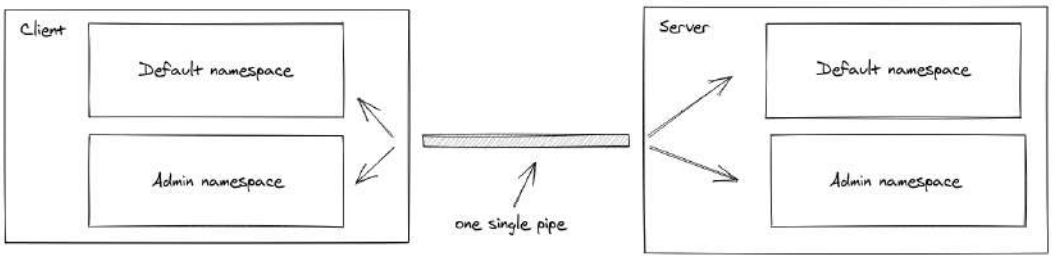

Figure 6: Socket.io - *namespaces*

4.1.5 Socket.io

Socket.io è una libreria che abilita una comunicazione bidirezionale a bassa latenza e event-based fra client e server, in un ambiente di sviluppo che faccia utilizzo del paradigma *Javascript everywhere*. Ciò consente, sfruttando HTTP long-polling e/o WebSocket come metodo di trasporto a basso livello, di avere una comunicazione in real-time fra Server e Client. Sfruttando i *namespaces* e le *room* di Socket.io, si è divisa la logica del backend per ogni entità, in modo da avere una relazione 1 a 1 fra tipo di utenti (cliente, ristoratore...) e logica applicativa.

4.1.6 Redis

Redis è un in-memory database open-source, ideale per implementare caching in memoria ad elevata disponibilità. Inoltre Redis mette a disposizione una libreria pub/sub (publish/subscriber).

4.1.7 Docker

Docker è una soluzione software che virtualizza parte del kernel Linux, in modo da creare pacchetti software vergini, minimali e con tutte le dipendenze necessarie, per facilitare la fase di deploy di applicazioni. Questi pacchetti, in gergo *container*, sono stati utilizzati per distribuire l'intera applicazione, sfruttando la funzionalità *docker compose*, costruendo le seguenti immagini:

- MongoDB
- Redis
- Backend
- Angular Frontend

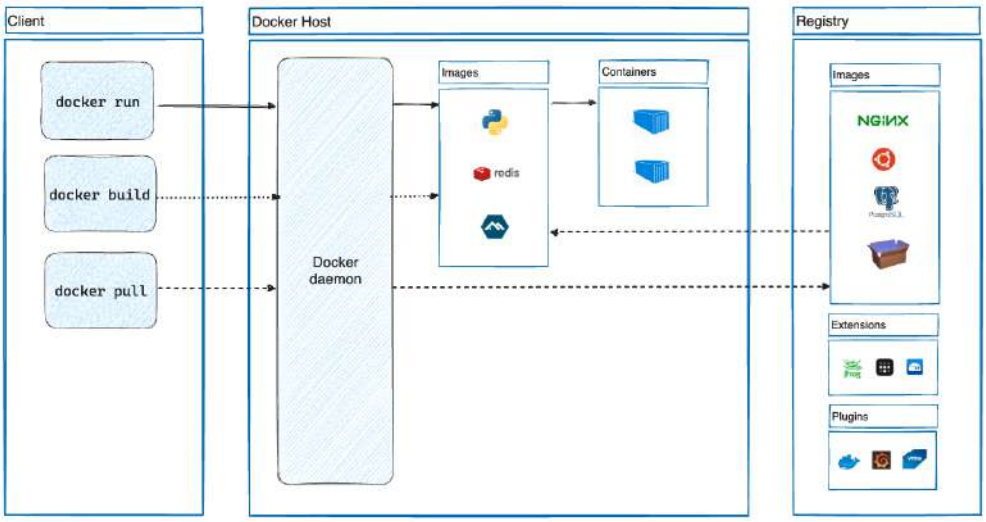

Figure 7: Docker Architecture

4.1.8 Angular e Leaflet

Per l'user interface, come tecnologia di riferimento, è stato scelto *Angular 2+* con Typescript, un framework per lo sviluppo di single-page application, abbinato ad *Angular Material*, una libreria che mette a disposizione varie componenti grafiche pronte all'uso e, infine, *Leaflet*, una libreria Javascript open-source per costruire mappe interattive.

4.2 Implementation Details

4.2.1 Endpoints

Lanciando l'applicazione è possibile consultare la documentazione Swagger, che illustra nel dettaglio tutti gli endpoint. Si vuole comunque descrivere il funzionamento della rotta *GET /shops*, poichè si ritiene non banale come è stata sviluppata questa funzionalità.

Figure 8: Users endpoints

Figure 9: Orders endpoints

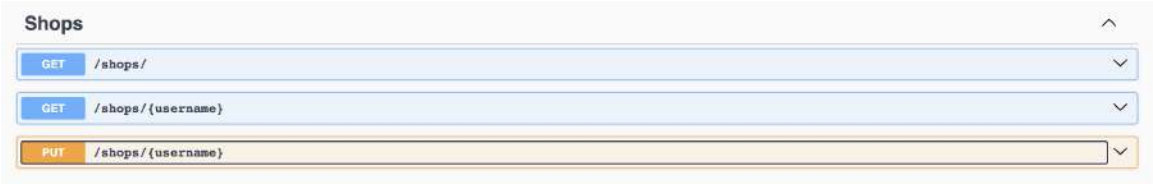

Figure 10: Shops endpoints

4.2.2 Retrieve nearby shops based on geoposition

Come descritto in precedenza, gli utenti, una volta effettuato l'accesso alla propria area riservata, devono poter visualizzare su una mappa tutti i ristoranti vicini alla loro posizione, disponibili alle consegne. Per far fronte a ciò si è deciso di utilizzare il formato GeoJSON nel salvataggio delle posizioni di tutti gli utenti. Il GeoJSON ha diverse geometrie disponibili, fra cui quella d'interesse per il nostro caso d'uso, ovvero il tipo **Point**.

```
const Coordinate = new Schema({
  type: {
    type: String,
    enum: ['Point']
  },
  coordinates: {
    type: [Number]
  }
})
```

Figure 11: GeoJSON - Point Geometry

Inoltre MongoDB mette a disposizione vari indici per poter ottimizzare le query più effettuate, ordinando i dati in un **B-Tree** in modo tale da ridurre il numero di accessi in memoria. Per il nostro caso d'uso, in cui la query con maggior impatto è la ricerca dei ristoranti più vicini alla posizione dell'utente, sono di particolare interesse due tipi di indici:

- **2d index**: effettua la query interpretando uno spazio geometrico piano e anche sferico, sebbene per quest'ultimo non sia propriamente indicato. Accetta coordinate nel formato *[lon, lat]*
- **2dsphere**: effettua la query interpretando uno spazio geometrico sferico. Accetta coordinate nel formato *[lon, lat]* e GeoJSON.

Figure 12: User Schema - Geolocation index 2dsphere

È stato creato un indice di tipo **2dsphere** nella struttura dati di MongoDB, preferendo performance e precisione dei dati ottenuti, sebbene ciò comporti una maggior quantità di memoria occupata nel database.

Infine è stato possibile costruire il controller dell'endpoint, dove viene eseguita la query che restituisce tutti i negozi nelle immediate vicinanze dell'utente, passando all'operatore **$near** *latitudine*, *longitudine* e *MAX_DISTANCE*, dove i primi due parametri sono le coordinate dell'utente, mentre l'ultimo vincola la massima distanza fra posizione del cliente e del ristoratore, fissata a *5 km*.

4.2.3 Users Geolocation

Le coordinate degli utenti sono state gestite in 2 modi differenti:

- **statico** per clienti e ristoratori, poichè in fase di consegna le loro posizioni è necessario che non abbiano variazioni significative;
- **dinamico** per i rider;

Per ottenere, quindi, le coordinate di clienti e ristoratori, viene richiesto in fase di registrazione *indirizzo*, *città* e *codice postale*, che vengono poi passati come query params all'api di geocoding messa a disposizione da *Geoapify*, per ricavare latitudine e longitudine da memorizzare sul db:

```
api.geoapify.com/v1/geocode/search?text=ViaPippo,1,47522,Cesena
```

```
exports.getAllShops = async (req, res, next) => {
  const { lat, lon } = req.query

  await User.find({
    geolocation: {
      $near: {
        $geometry: { type: 'Point', coordinates: [lon, lat] },
        $maxDistance: MAX_DISTANCE
      }
    }
  }) ...
    .where('role') ...
    .equals( val: 'RISTORATORE') ...
    .where('working') ...
    .equals( val: true) ...
    .select( arg: 'menu restaurant_name working geolocation') ...
    .then( onfulfilled: (result :... ) => res.json(result)) Promise<any>
    .catch((err) => next(err))
}
```

Figure 13: GET /shops?lat=num&lon=num

Mentre per i rider in fase di consegna, la posizione viene aggiornata ogni 2 secondi e condivisa all'utente tramite il canale socket.io *sharelocation*. Una volta completato l'ordine, la condivisione della posizione viene interrotta con la funzione *clearInterval()*, notificando gli altri utenti dell'avvenuta consegna:

```
    shareLocation(id: string) {
  this.intervalId = setInterval(() => {
    this.socket.emit('sharelocation', {id: id, coordinates: this.coordinates})
  }, 2000)
}

confirmDelivered() {
  this.currentOrder.state = "completed"
  this.onTravel = false
  clearInterval(this.intervalId)
  this.httpService.updateOrder(this.currentOrder).subscribe()
  this.socket.emit('orderCompleted', this.currentOrder._id)
  window.location.reload()
}
```

5 Self-assessment / Validation

5.1 Jest

Per testare il sistema è stata prodotta una classe di Test automatici servendosi del framework *Jest*. Per eseguire i test vengono creati un database e un'istanza di Express apposite, per partire da un ambiente vergine da poter buttar giù una volta completati i test. Tramite la libreria *axios* gli endpoint vengono richiamati con dei dati mockati, restituendo il risultato della validazione.

Figure 14: Jest Test

5.2 CI/CD Pipeline

Per garantire la bontà di quanto sviluppato, inoltre, è stata creata una pipeline su Gitlab con 3 stages principali:

- **Build**: lancia il comando *docker compose build*, che costruisce le immagini dei servizi **backend, angular, redis e mongodb**, dichiarati nel file *docker-compose.yml*
- **Test**: lancia i test Jest mostrati nel paragrafo precedente

- **Deploy**: lancia il comando *docker compose up -d* per verificare che il deploy dell'applicazione non restituisca errori

Questi stages vengono eseguiti in ambienti remoti totalmente vergini, detti *Runner*. Essendo i *Runner di Gitlab* container Docker a tutti gli effetti, è stata utilizzata l'immagine *docker:dind (Docker in Docker)* per essere in grado di usare la cli di Docker negli stages della pipeline.

6 Deployment Instructions

Per poter lanciare l'applicazione è sufficiente eseguire il seguente comando nella root del progetto:

```
docker compose up
```

La pagina d'accesso sarà disponibile al seguente link. Il backend sarà disponibile all'ascolto sulla porta 4000:4000 (locale:container), Redis su porta 6379:6379, mentre mongodb sulla porta di default 27017:27017.

All'inizializzazione di MongoDB viene restorato un dump, con al suo interno delle credenziali già fissate, per poter provare l'applicazione senza registrarsi. Le suddette credenziali sono elencate nel file *README.MD*.

7 Usage Examples

Una volta avviata l'applicazione sarà possibile effettuare il login, scegliendo il ruolo abbinato all'utente.

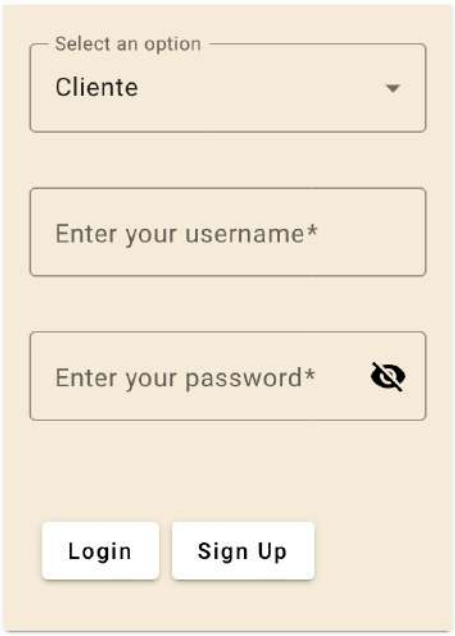

Figure 15: Login Page

Dalla dashboard del cliente è possibile visualizzare i menu cliccando sul tab del ristorante desiderato. Una volta selezionati gli item desiderati, è possibile inoltrare l'ordine all'esercente, se e solo se il proprio credito virtuale è sufficiente per pagare l'intero conto.

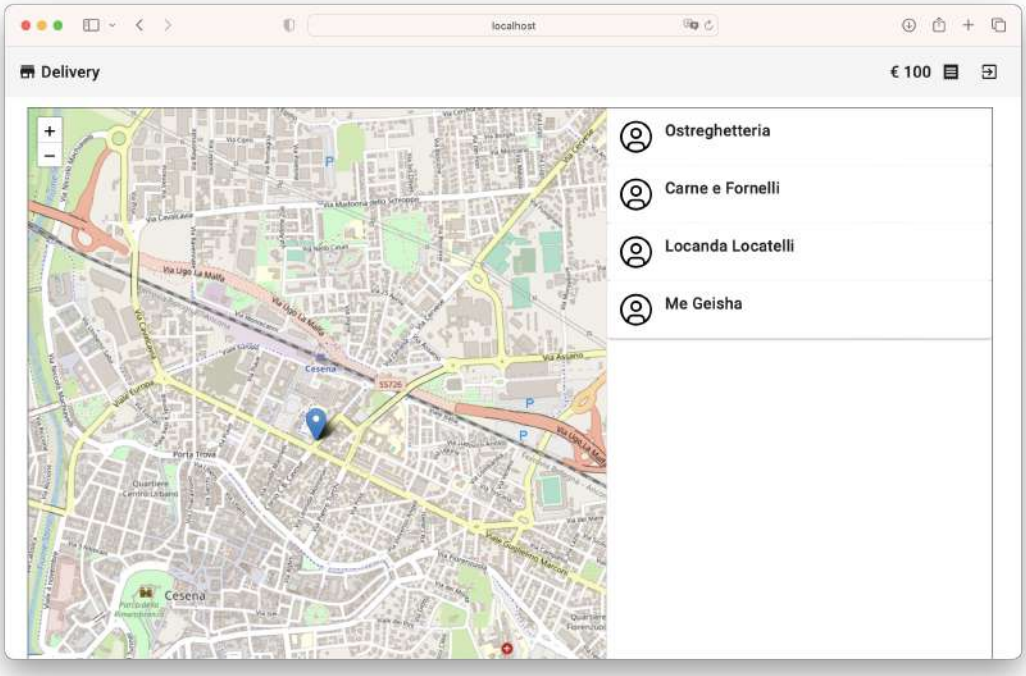

Figure 16: User Dashboard

Il ristoratore nella sua dashboard vedrà gli ordini in arrivo e il proprio menù presentato agli utenti. Nell'header della tabella dei menù sono posizionati un pulsante e un flag: il primo per modificare categorie ed elementi del menù, il secondo per dichiarsi disponibil ad accettare ordini.

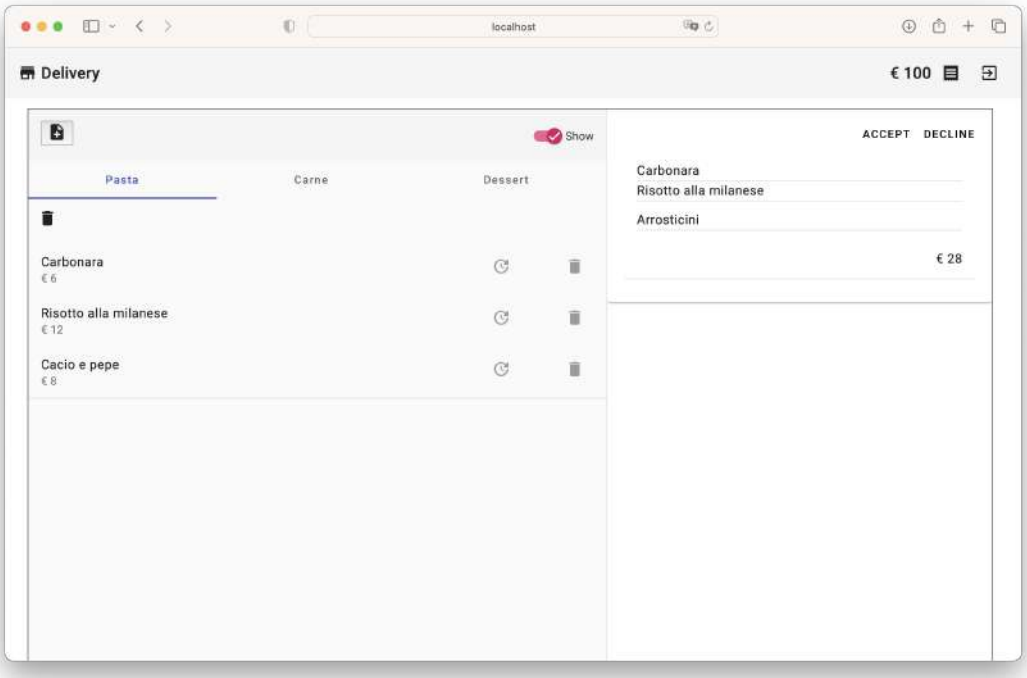

Figure 17: Shop Dashboard

Se sia ristoratore che rider accettano l'ordine, quest'ultimo potrà visualizzare i dettagli dell'ordine e verrà inizializzata la navigazione in real-time, guidando il rider prima presso il negozio e poi presso il cliente che ha effettuato l'ordine.

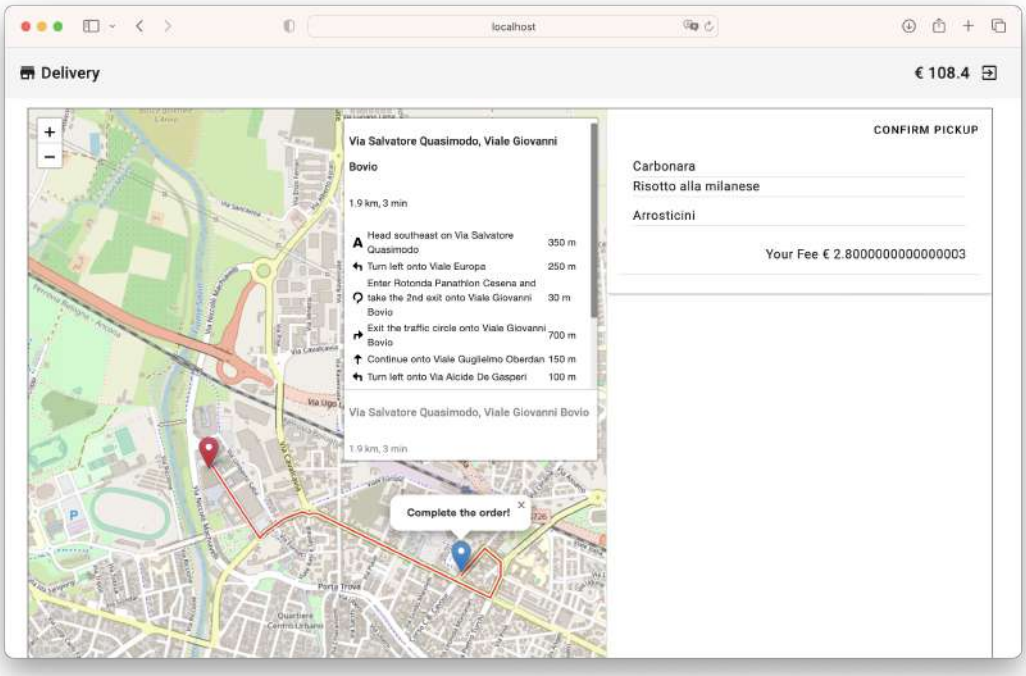

Figure 18: Rider Dashboard

Da notare che il flag per interrompere il turno di lavoro, è visibile solamente se il rider non ha delle consegne da completare.

Figure 19: Rider Flag

8 Conclusions

È stata sviluppata una web-app di delivery con una struttura **Client-Server** servendosi dello stack **MEAN**. La comunicazione avviene con HTTP, utilizzando le ReST Api per interagire con le risorse messe a disposizione dal web server e Socket.io per uno scambio di messaggi in real-time fra le parti.

8.1 Future Works

Possibili sviluppi futuri potrebbero essere:

i. Uno storico degli ordini per ogni utente attivo sulla piattaforma

ii. Un sistema di pagamento affidabile che si basi sugli esistenti circuiti bancari

iii. Implementare un'area riservata per la Customer Care, per poter supportare i clienti della piattaforma tramite chat

iv. Consentire agli utenti di cambiare l'indirizzo inserito in fase di registrazione e/o di memorizzare più indirizzi di consegna

8.2 What did we learned

Lo sviluppo di questo progetto mi ha permesso di entrare nel dettaglio di tecnologie innovative, come Docker, introdotte durante il corso, assimilandole al meglio. In aggiunta, è stato possibile consolidare le conoscenze riguardo tutte le fasi del ciclo di vita di un progetto, dalla progettazione al deploy.

Bibliography

[1] Angular. https://angular.io/docs/.

[2] Docker. https://redis.io/docs/getting-started/.

[3] Express.js. https://expressjs.com.

[4] Leaflet. https://leafletjs.com/reference.html.

[5] Mondodb. https://www.mongodb.com/docs/.

[6] Mongoose. https://mongoosejs.com/docs/guide.html.

[7] Node.js. https://nodejs.org/it/docs.

[8] Redis. https://redis.io/docs/getting-started/.

[9] Socket.io. https://socket.io/docs/v4/.

List of Figures