

UniBurger

Applicazioni e Servizi Web

Matteo Cavalluzzo - 0000884883 {matteo.cavalluzzo@studio.unibo.it}
Luca Giorgetti - 0000884181 {luca.giorgetti7@studio.unibo.it}

17 Dicembre 2020



Indice

1	Introduzione	1
2	Requisiti	3
2.1	Customer	3
2.2	Admin	4
2.3	Rider	4
3	Design	5
3.1	Catalogo Prodotti	5
3.2	Autenticazione	7
3.3	Area riservata al customer	8
3.3.1	Carrello	8
3.3.2	Schermata completamento ordine	9
3.3.3	Lista ordini	10
3.4	Area riservata all'admin	10
3.4.1	Storico ordini	10
3.4.2	Ordini live	11
3.4.3	Gestione rider	12
3.4.4	Timetable	13
3.4.5	Schermate per l'aggiornamento del catalogo	14
3.5	Area riservata al rider	15
3.6	Caricamento e gestione degli errori	15
3.6.1	Schermate di caricamento	15
3.6.2	Schermate di errore	16
3.6.3	Alert di notifica	17
4	Tecnologie	19
4.1	Stack MERN	19
4.2	MongoDB	19
4.3	Express.js	19
4.4	React	20
4.4.1	Redux	20

4.5	Node.js	22
4.6	Socket.io	22
4.7	JSON Web Token	22
4.7.1	Funzionamento di dettaglio	23
4.8	Strumenti utilizzati	23
4.8.1	Postman	23
4.8.2	Trello	24
4.8.3	Docker	24
4.8.4	GitHub	24
4.8.5	Balsamiq	25
5	Codice	27
5.1	Client	27
5.1.1	Redux	28
5.1.2	View	31
5.2	Server	32
5.2.1	Models	33
5.3	Funzionalità di gestione degli ordini realtime	35
6	Test	37
6.1	Client	37
6.2	Server	38
6.3	Test con gli utenti	40
7	Deployment	41
7.1	Rilascio, installazione e messa in funzione	41
7.1.1	Credenziali di prova degli utenti	42
8	Conclusioni	43

Capitolo 1

Introduzione

Secondo uno studio di www.foodaffairs.it:

"Il digital food delivery continua a crescere in Italia, rappresentando tra il 20% e 25% dell'intero settore del domicilio"

Di fronte ad un settore in crescita, anche a causa della diffusione del virus Covid-19, con la conseguente chiusura dei negozi fisici, ci ha stupito che solamente un quarto delle attività abbia scelto di organizzare il servizio a domicilio sfruttando una piattaforma digitale. Il passaggio al digitale porterebbe innumerevoli vantaggi sia ai clienti che ai gestori del negozio.

I clienti potrebbero consultare ancora più facilmente l'elenco dei prodotti disponibili, creare l'ordine in maniera agevole e scegliere l'orario di consegna più congeniale a loro. Allo stesso tempo sgraverebbe il negoziante dalla creazione dell'ordine (basta pensare al tempo che l'operatore impiega al telefono per l'ordinazione) alla gestione del pagamento (tracciato ed immediato), permettendogli di utilizzare queste risorse in maniera diversa.

Pensando anche al contesto in cui viviamo, un paese di 12000 abitanti, pochissimi negozi scelgono di affidarsi alle compagnie di delivery più blasonate, probabilmente per via del numero di ordini effettuati in ciascuna serata non al pari di quelli delle grandi città. Anche per questo abbiamo ritenuto che una soluzione semplice, in grado di coprire tutte le fasi dell'ordine, dall'ordinazione alla consegna potesse essere una scelta piuttosto diffusa.

Nel nostro caso, abbiamo adattato l'applicazione al contesto di un negozio di panini. Tuttavia, con poche modifiche e personalizzazioni, la soluzione potrebbe essere adattata a qualsiasi negozio che volesse passare al digital food delivery senza sostenere una spesa pari a quella di una soluzione personalizzata.

Capitolo 2

Requisiti

Si desidera realizzare una applicazione web per un negozio di vendita di alimenti a domicilio. L'applicazione dovrà supportare 3 tipologie diverse di utenti, ciascuno dei quali avrà a disposizione diverse informazioni e diversi servizi. Gli utenti per fare operazioni dovranno autenticarsi con username e password, e dovranno restare autenticati anche alla chiusura e riapertura del browser.

- **Customer**

È il cliente del negozio, che tramite l'applicazione può ordinare cibo a domicilio.

- **Admin**

Il gestore dell'applicazione, ha funzionalità riservate solamente a lui.

- **Rider**

E uno dei fattorini del negozio ed ha il compito di consegnare a casa gli ordini fatti dai clienti.

L'applicazione deve essere fruibile sia da dispositivi mobile che da desktop e deve mantenere lo stato dell'ordine durante tutte le fasi di elaborazione e consegna.

2.1 Customer

Le funzionalità che sono richieste per il ruolo di Customer sono:

- Registrazione e Login tramite email e password.
- Visualizzazione dei prodotti accendendovi tramite la selezione di categorie di prodotto, anche quando non ha eseguito il login.

- Aggiunta dei prodotti al carrello e visualizzazione dei prodotti aggiunti.
- Conclusione dell'ordine tramite l'inserimento dei dati di consegna.
- Selezione modalità di pagamento(online o in contanti).
- Selezione dell'orario di consegna in base agli orari di apertura del negozio.
- Non deve essere possibile fare un ordine se non sono presenti slot di tempo selezionabili in giornata.
- Visualizzazione dello stato dello stato dell'ordine (in lavorazione, in consegna, consegnato).

2.2 Admin

Oltre alla visualizzazione delle categorie e dei prodotti, gli amministratori dell'applicazione (possono essere anche più di uno), dopo essersi autenticati con email e password devono poter compiere operazioni riservate:

- Gestione dell'anagrafica (creazione, modifica ed eliminazione) di prodotti e categorie.
- Gestione ed evasione degli ordini mediante il cambiamento di stato, aggiornato aggiornato in tempo reale.
- Assegnazione di un ordine ad un fattorino.
- Creazione di utenze per i fattorini.
- Gestione degli orari di attività del servizio con la possibilità di selezionare anche i giorni di chiusura.
- Visualizzazione dello storico degli ordini ricevuti e del loro stato.

2.3 Rider

Il fattorino, una volta che ha eseguito il login con username e password, avrà accesso ad una schermata, aggiornata in tempo reale, tramite la quale sarà in grado di:

- Accesso alle informazioni di consegna e della tipologia.
- Cambiamento dello stato dell'ordine una volta effettuata la consegna effettuata.

Capitolo 3

Design

Il processo di design della nostra applicazione si è basato su approccio mobile first.

Inizialmente abbiamo definito le funzionalità che il nostro sistema doveva supportare, e grazie all'utilizzo del tool Balsamiq abbiamo progettato i mockup delle schermate.

Solamente in un secondo momento, dopo aver capito quali componenti sarebbero serviti nelle schermate abbiamo scelto un tema che ci permettesse di ottenere questi componenti in maniera più efficiente possibile. La scelta nel nostro caso è ricaduta su Material UI Theme.

Quando lo sviluppo dell'applicazione web era a buon punto, abbiamo scelto font e colori, in base a come risultavano più leggibili e di impatto.

3.1 Catalogo Prodotti

Composta dalla serie di schermate che permettono di visionare il catalogo prodotti.

- Lista categorie mostra la lista delle categorie
- Dettaglio categoria mostra tutti i prodotti associati a quella categoria. Se l'utente è loggato avrà anche la possibilità di aggiungere un prodotto direttamente al carrello tramite l'icona apposita.
- Pagina di dettaglio prodotto mostra i dettagli dell'alimento selezionato con titolo, descrizione, lista ingredienti e prezzo. In caso di utente loggato è anche presente il pulsante di aggiunta al carrello, affiancato dal selettore della quantità.

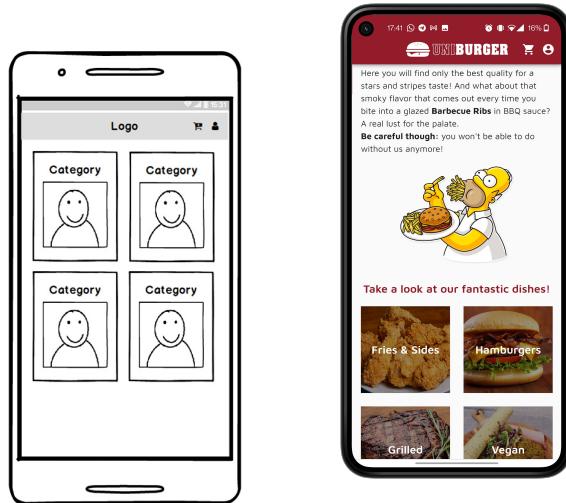


Figura 3.1: Homepage utente / lista categorie

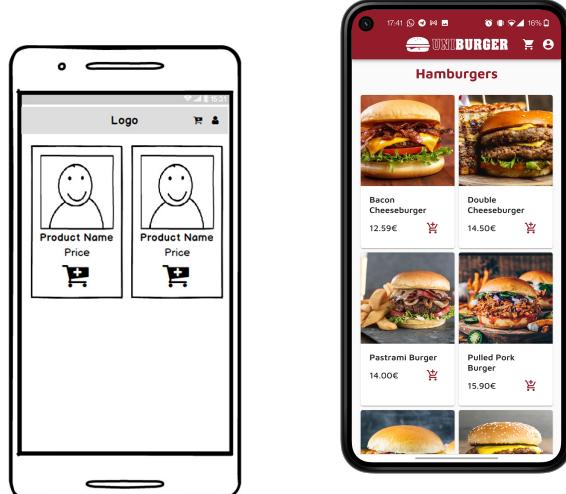


Figura 3.2: Prodotti per categoria

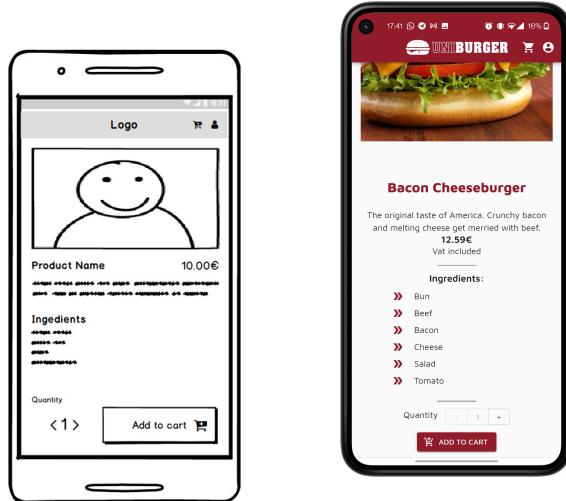


Figura 3.3: Dettaglio prodotto

3.2 Autenticazione

Sono presenti una schermata di registrazione con cui l'utente consumer può creare il suo account, e una di login con cui qualsiasi tipo di utente, indipendentemente dal ruolo, potrà accedere al sistema.

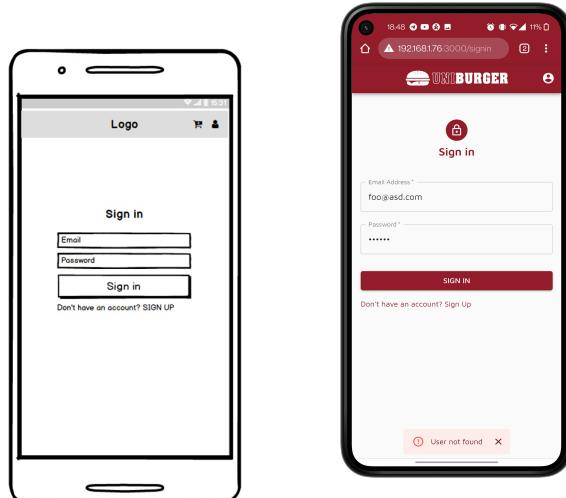


Figura 3.4: Login

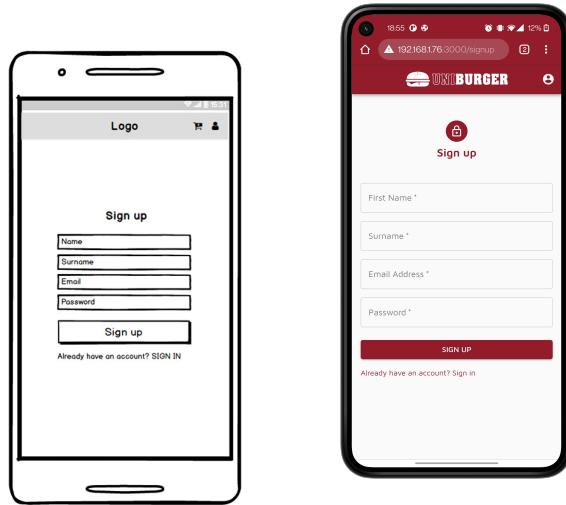


Figura 3.5: Registrazione

3.3 Area riservata al customer

L'utente customer, ovvero colui che utilizza l'applicazione per effettuare acquisti, ha accesso alla sezione catalogo prodotti e a degli ambienti dedicati ad effettuare gli ordini e a controllare lo stato degli stessi.

3.3.1 Carrello

Ambiente che mostra tutti i prodotti che sono stati aggiunti dal cliente, con a fianco le relative quantità ed in basso il prezzo totale.

Interagendo su ciascun elemento di prodotto l'utente ha la possibilità di modificare velocemente la quantità o di rimuovere il prodotto dal carrello. Questa schermata è disponibile solamente qualora l'utente sia loggato, insieme all'indicazione della quantità dei prodotti all'interno dell'icona del carrello posta sulla Toolbar.

Una volta controllati i prodotti, l'utente può passare al successivo step di completamento ordine cliccando sull'apposito tasto.

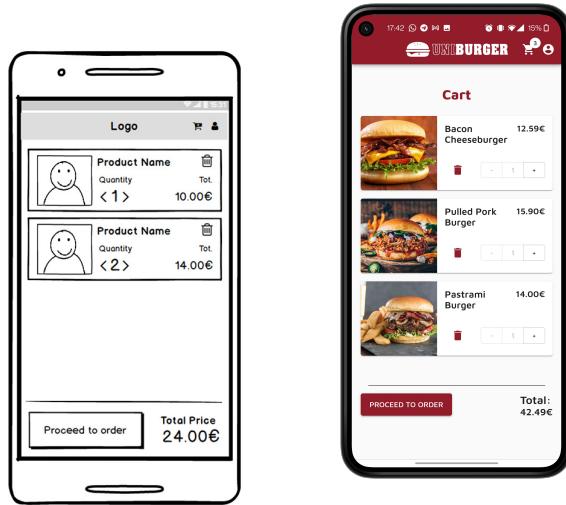


Figura 3.6: Carrello

3.3.2 Schermata completamento ordine

Schermata accessibile solamente provenendo del carrello, riepiloga il totale da pagare ed è caratterizzata da una form da compilare per poter completare l'ordine.

Una volta completato l'ordine viene visualizzata una dialog che chiede permette di scegliere all'utente se tornare in home o navigare nell'ambiente di lista ordini.

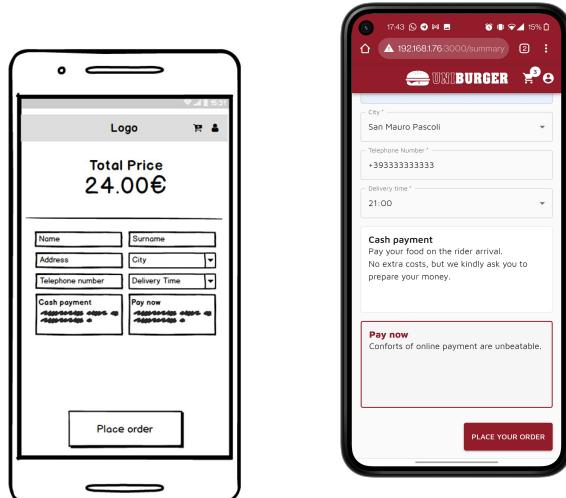


Figura 3.7: Completamento ordine

3.3.3 Lista ordini

Visualizza lo storico degli ordini effettuati. Ciascun elemento mostra i dettagli dell'ordine e il suo stato. È poi presente un elemento di paginazione che permette di navigare velocemente tra gli ordini, senza dover visualizzare una lista infinita.

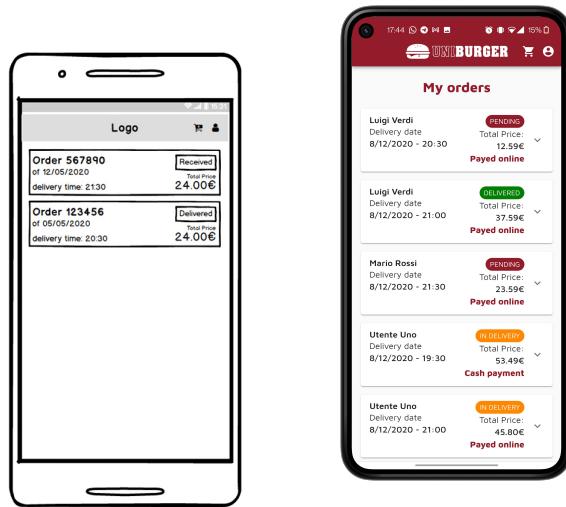


Figura 3.8: Lista ordini effettuati

3.4 Area riservata all'admin

Gli ambienti descritti di seguito sono accessibili solamente ad utenti di tipo admin.

3.4.1 Storico ordini

Come il layout analogo alla schermata di lista ordini visibile all'utente, in questo caso la lista ordini visualizzata riguarda tutti gli utenti.

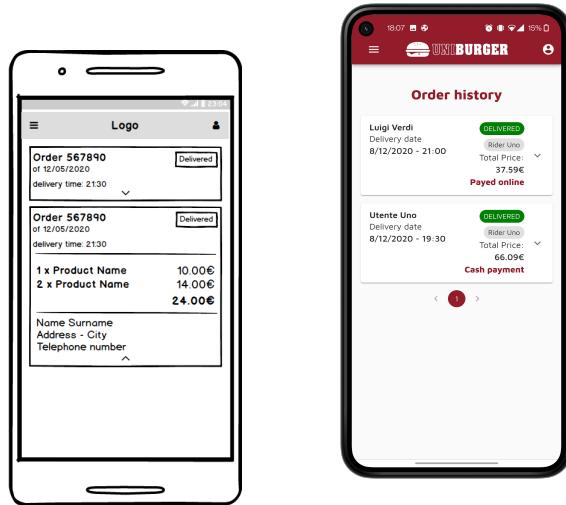


Figura 3.9: Storico ordini

3.4.2 Ordini live

Anche qui il layout è analogo ai precedenti, con la differenza che in questa schermata gli ordini vengono aggiornati in tempo reale, senza la necessità di dover ricaricare la pagina ogni volta. Oltre a ciò è anche possibile cambiare lo stato dell'ordine assegnandolo ad uno dei fattorini disponibili, selezionabile attraverso la dialog dedicata.

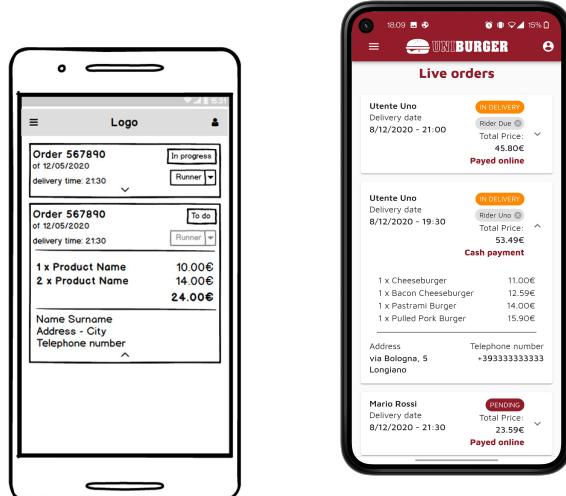


Figura 3.10: Ordini live

3.4.3 Gestione rider

Ambiente che permette all'admin di gestire la flotta di rider presenti: crearene di nuovi o eliminare quelli già esistenti.

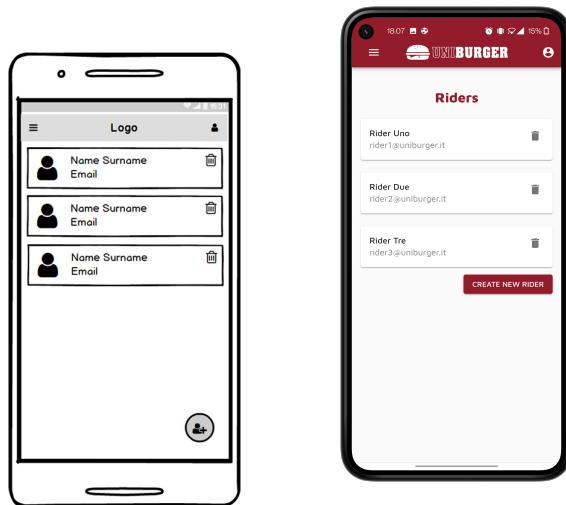


Figura 3.11: Gestione rider

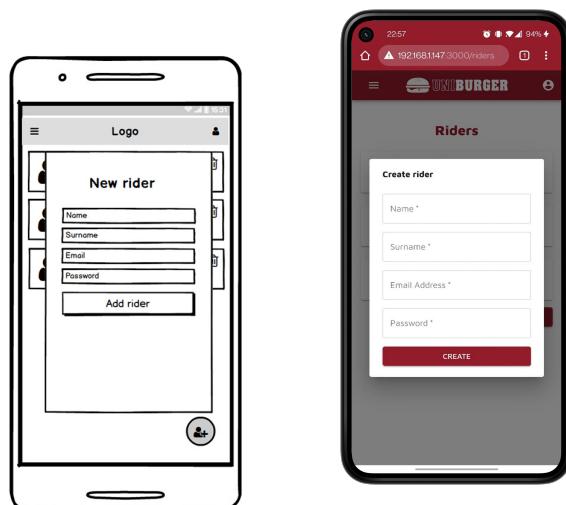


Figura 3.12: Creazione rider

3.4.4 Timetable

Permette di modificare gli orari di consegna dell'attività. Per ogni giorno della settimana è possibile abilitare la consegna a pranzo e/o a cena. Una volta abilitato il pasto è poi possibile modificare gli orari di inizio e fine servizio. Questi orari saranno poi consultabili dall'utente sulla schermata di completamento ordine.

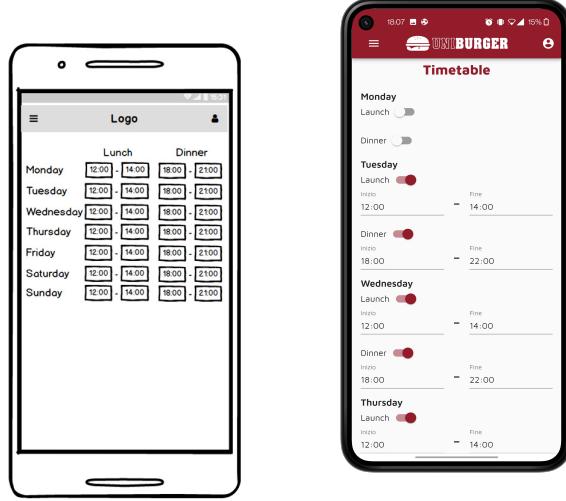


Figura 3.13: Timetable

3.4.5 Schermate per l'aggiornamento del catalogo

L'utente di tipo admin è l'unico che può operare sul catalogo, aggiungendo, modificando o rimuovendo elementi.

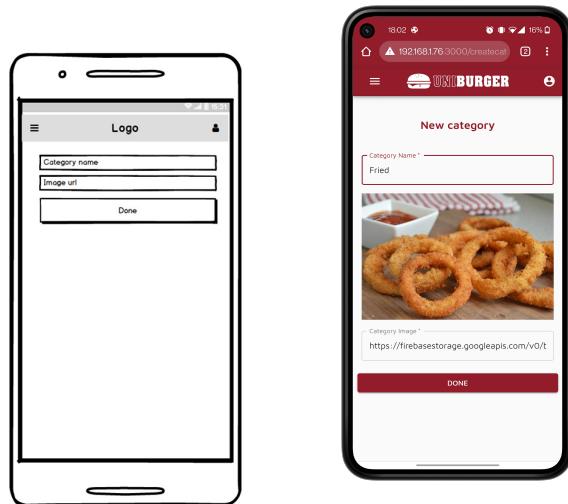


Figura 3.14: Creazione categoria

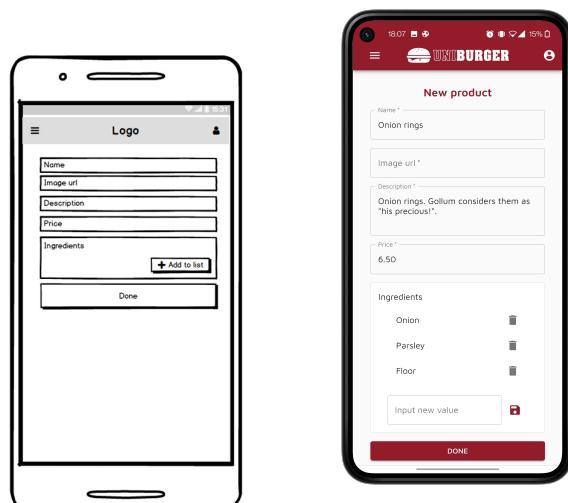


Figura 3.15: Creazione e modifica prodotto

3.5 Area riservata al rider

L'unica schermata accessibile dal rider è la lista di ordini presi in carico, che viene aggiornata real time senza dover ricaricare la pagina. L'unica azione che può compiere è impostare l'ordine in stato consegnato.

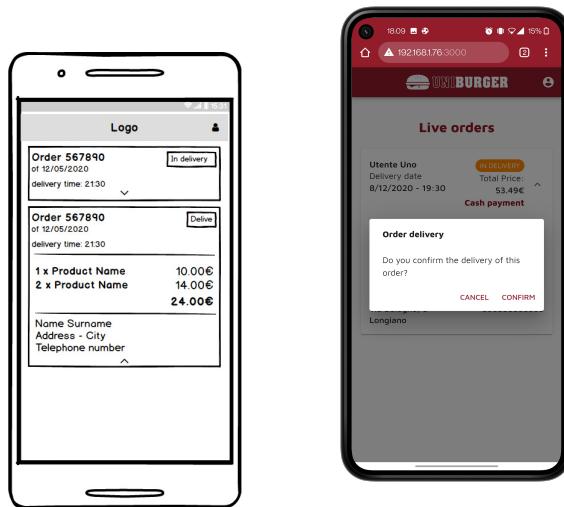


Figura 3.16: Schermata di gestione ordini del rider

3.6 Caricamento e gestione degli errori

Sono poi presenti degli elementi generali utilizzati in vari ambienti dell'applicazione, che informano l'utente circa il caricamento delle varie schermate o errori di caricamento o di risoluzione di alcune azioni.

3.6.1 Schermate di caricamento

Sono dei semplici loader che vengono visualizzati nel tempo che intercorre tra la richiesta di dati e il loro arrivo.



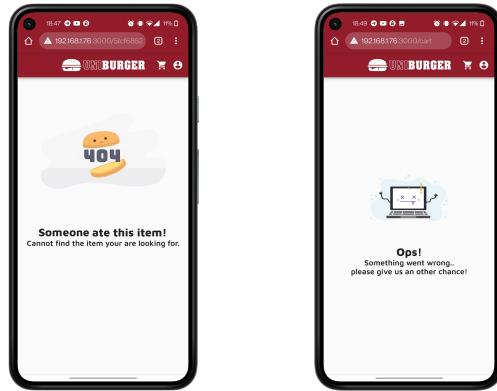
Figura 3.17: Loader

3.6.2 Schermate di errore

In caso di errori che impediscono la visualizzazione corretta di una schermata, vengono visualizzate delle schermate informatiche che avvisano l'utente che qualcosa è andato storto.

All'interno dell'app ne abbiamo usate 2 per gestire i principali casi di errore:

- Errore di **elemento non trovato** (figura 3.18a) visualizzato quando l'elemento richiesto per popolare una schermata non viene trovato (solitamente associato ad un errore 404 lato server). Nel caso specifico è stato utilizzato nelle varie schermate del catalogo nel caso in cui una categoria o un prodotto non venissero trovati.
- Errore **generico** (figura 3.18b) visualizzato in corrispondenza di un errore meno chiaro, come mancanza di connettività, mancata risposta del server.



(a) Errore di elemento non trovato (b) Errore generico

Figura 3.18: Schermate di errore

3.6.3 Alert di notifica

Componenti utilizzati per mostrare un breve messaggio informativo all’utente, attirando la sua attenzione ma senza interrompere il suo task in corso. Posso essere sia informativi che di errore. Esempi di utilizzo sono in corrispondenza di registrazione o aggiunta al carrello di un prodotto avvenuti con successo, errore di login a causa di credenziali non valide e altri casi simili.



Figura 3.19: Alert

Capitolo 4

Tecnologie

4.1 Stack MERN

Per la struttura del progetto abbiamo adottato lo stack MERN, che ci ha permesso di sviluppare in simultanea due applicazioni distinte: il client ed il server. Il client sfrutta la libreria Javascript React, il server invece è sviluppato con Node.js, con Express.js utilizzato per il routing e MongoDB utilizzo per il salvataggio dei dati.

4.2 MongoDB

Il database alla base dello stack MERN è MongoDB, un DBMS non relazionale (NoSQL) open-souce.

Rispetto ai database relazionali, non ha una struttura basata sulle tabelle, ma sfrutta la modellazione a documenti, che sono strutturati in stile json. Questa caratteristica dei database NoSQL permette di modificare in maniera semplice la struttura dei dati anche durante la fase dello sviluppo del software, cosa che con i database relazionali risultava piuttosto pesante.

4.3 Express.js

Express è un framework per applicazioni web Node.js che fornisce una serie di funzioni avanzate per le applicazioni web e per dispositivi mobili.

Agevola la creazione di api di tipo restful, fornendo metodi di utilità HTTP e middleware. All'interno dell'app è stato infatti utilizzato per la gestione del sistema di routing e per la parte di autenticazione delle rotte utilizzando middleware dedicati.

4.4 React

Una delle librerie Javascript più usate per la creazione di interfacce utente. A differenza di altri framework come Angular, non dispone di default di funzionalità avanzate per la gestione dello stato o della navigazione, abbiamo quindi fatto uso di librerie esterne quali:

- Redux per la gestione dello stato.
- React Router per la gestione della navigazione.

4.4.1 Redux

Abbiamo scelto di utilizzare questa libreria per gestire lo stato dell'app in maniera più pulita e centralizzata. Redux è una libreria per la gestione dello stato che può essere collegata con qualsiasi libreria Javascript, non solo React. Tuttavia, lavora molto bene con React a causa della sua natura funzionale. Con React, ogni componente ha uno stato locale che è accessibile dall'interno del componente e che può essere passato come *prop* ai figli, questo stato viene utilizzato sia per memorizzare lo stato dell'interfaccia utente (elementi dell'interfaccia o input di form compilati) ma anche lo stato dell'applicazione (come ad esempio i dati recuperati da un server, lo stato di accesso dell'utente ecc.).

La memorizzazione di queste informazioni risulta facilmente gestibile quando si dispone di un'applicazione con pochi componenti, quando invece questo numero aumenta, aumentano anche i livelli nella gerarchia dei componenti e la gestione dello stato potrebbe diventare problematica.

Redux permette di separare lo stato dell'applicazione da React. Introduce il concetto di store, lo stato globale dell'applicazione, che viene osservato da tutti i componenti, ogni volta che lo store viene aggiornato i componenti si aggiornano di conseguenza, questo permette di creare dei componenti molto più semplici.

Il funzionamento interno di redux si basa sui seguenti concetti fondamentali:

- *Store*

un grande oggetto Javascript immutabile che contiene lo stato corrente dell'applicazione. Lo store fornisce lo stato all'applicazione, e ogni volta che lo store viene aggiornato, la vista che lo asserva viene ridisegnata. Immutabile vuol dire che lo stato dello store non può mai essere modificato, vengono invece create nuovi versioni dello store a partire dalle

precedenti.

Ogni parte dell'applicazione decide di osservare una o più parti dello store, disegnano la view di conseguenza.

- *Action/Action Creators*

rappresentano le azioni, dei semplici oggetti Javascript che inviano delle informazioni verso lo store. Sono l'unica fonte di informazioni dello store. Ogni azione deve avere una proprietà di tipo che indica ciò che deve essere fatto, e opzionalmente una parte d dati con le informazioni necessarie a trasformare lo stato dell'applicazione. Gli Action Creators sono invece azioni che a seguito di attività asincrone (come chiamate ad api) ritorneranno altre azioni.

- *Reducers*

sono i componenti responsabili della risoluzione delle azioni. Sono implementati da delle funzioni pure che dato lo stato corrente e l'azione in input determinano il nuovo stato dello store.

Tutti questi componenti messi assieme permettono di astrarre la gestione della logica di aggiornamento dello stato dell'applicazione al di fuori dei componenti view di React. Questo consente non solo di avere del codice più pulito e testabile, come mostreremo nei capitoli successivi, ma anche di facilitare interventi di modifica futura, l'aspetto grafico dell'applicazione potrebbe essere del tutto variato senza mettere mano alla parte di gestione dello stato.

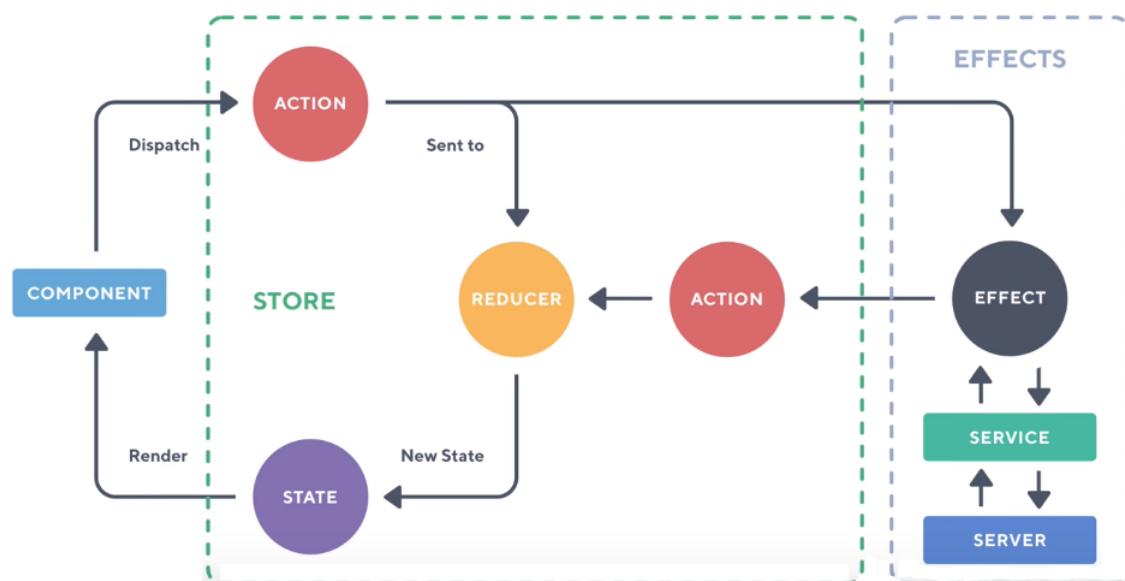


Figura 4.1: Redux

4.5 Node.js

Il server dell'applicazione è basato su Node.js, una piattaforma software cross-platform. Un vantaggio notevole ottenuto da questo motore runtime è quello derivante dal fatto che sfrutta un modello event-driven non bloccante, che lo rende reattivo ed efficiente. Un ulteriore vantaggio deriva dal fatto che, essendo open-souce, dispone di una grande mole di plugin installabili tramite npm, che permettono di personalizzarne l'utilizzo in base alle proprie esigenze, senza installare librerie superflue, che non farebbero altro che appesantire la piattaforma.

4.6 Socket.io

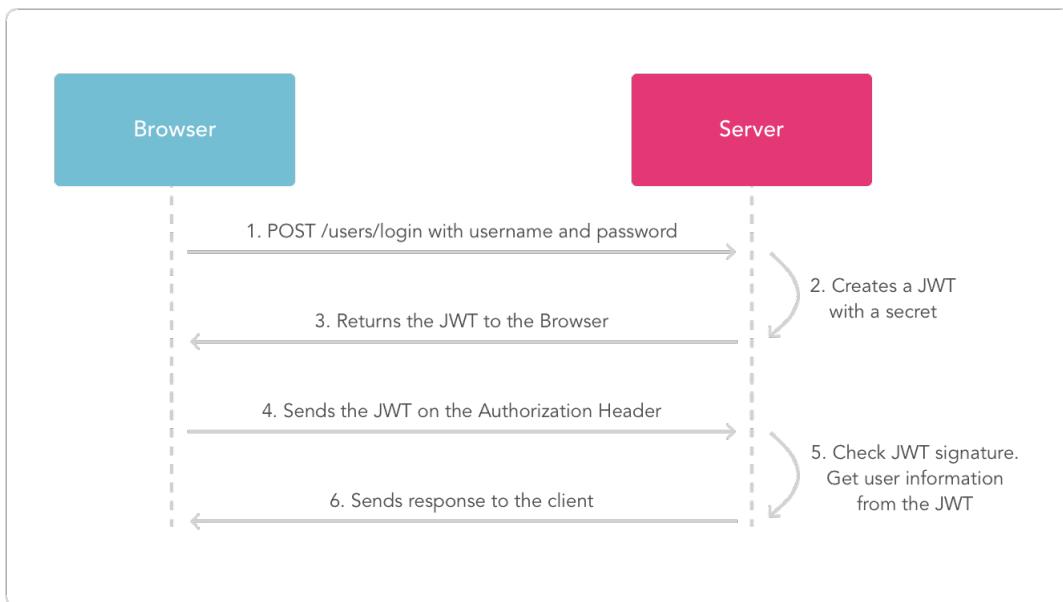
Una libreria Javascript che permette di creare applicazioni web che comunicano in tempo reale. È formata da una parte lato client che gira sul browser e una lato server per Node.js che insieme creano un canale bidirezionale per la scambio di informazioni realtime. All'interno dell'app è stato utilizzato per realizzare la funzionalità di ordini realtime, che permette all'admin di ricevere gli aggiornamenti degli ordini in arrivo e consegnati e al rider di ricevere gli ordini da consegnare in tempo reale, senza la necessità di dover ricaricare la pagina.

4.7 JSON Web Token

JSON Web Token è uno standard che definisce un modo sicuro di comunicare trasmettendo informazioni tra le parti come oggetti JSON. Queste informazioni vengono firmate (e posso essere quindi verificate) sia usando un segreto che una coppia di chiavi.

JSON Web Token può essere utilizzato per scambiare informazioni in maniera sicura o, come nel nostro caso per scopi di autorizzazione: successivamente al login dell'utente, tutte le successive richieste includeranno il JWT, permettendo all'utente di accedere alle risorse consentite con quel token.

4.7.1 Funzionamento di dettaglio



Durante l'autenticazione, quando l'utente si è loggato con successo al sistema usando le sue credenziali, gli viene ritornato come risposta un JSON Web Token.

Per accedere a una risorsa protetta, l'utente invia il JWT con la richiesta, tipicamente nel campo Authorization header, usando lo Schema bearer.

In questo modo il server può proteggere determinate rotte o variare il comportamento in base alla validità del token o al suo contenuto. Nel caso della nostra applicazione, il contenuto permette, oltre che ad identificare un utente, anche di verificarne il ruolo (consumer, admin o rider) adattando quindi le risposte in base a ciò.

4.8 Strumenti utilizzati

4.8.1 Postman

Postman è un piattaforma per lo sviluppo collaborativo ed il testing delle API. Ci è stato particolarmente utile per testare i servizi esposti dal server sviluppato con Express, condividendo una libreria di chiamate REST.

Con Postman ci è stato possibile verificare in maniera piuttosto rapida lo stato ed il payload di risposta delle chiamate. E' stata molto utile la funzio-

nalità di poter inserire all'interno dell'autenticazione il Bearer Token, così da poter chiamare anche i servizi riservati agli utenti autenticati.

4.8.2 Trello

Per la gestione del progetto, in particolar modo per tenere sotto controllo lo stato dello sviluppo dell'applicazione, abbiamo utilizzato una bacheca di Trello.

Trello è uno strumento online per la gestione dei progetti e dei task personali. Una volta definiti i task iniziali, modificati ed aggiornati anche durante tutto lo sviluppo, ci ha permesso di assegnare i vari requisiti allo sviluppatore a cui era stato assegnato.

Nella kanban board abbiamo utilizzato 5 stati: *To Do, Doing, Done, Bug, Improvement*, in cui lo specifico task veniva spostato ad ogni fase dello sviluppo. La possibilità di apporre etichette sui task ci ha permesso di dividere quelli relativi al server, quelli client e quelli che invece coinvolgevano entrambe le parti.

Molto utile è stata la possibilità di inserire in un task una checklist di cose da fare, in modo da poterle smarcare una ad una, senza rischiare di perderersi tra le cose da fare.

4.8.3 Docker

Per il delivery della nostra applicazione abbiamo utilizzato Docker. Docker è un progetto open-source per l'automatizzazione il deployment di applicazioni all'interno di *containers*, contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux.

Utilizzando Docker Compose per eseguire più container sotto una stessa rete, ci è stato possibile eseguire il deployment della nostra applicazione eseguendo un solo comando.

Oltre alle *images* create con la build dei nostri sorgenti, ci ha permesso di utilizzare quelle già disponibili su DockerHub, da una parte facendoci risparmiare lo sforzo di doverle implementare ex-novo, dall'altra fornendoci una immagine già testata ed utilizzata da tantissimi applicativi, e per tanto con buona sicurezza funzionante.

4.8.4 GitHub

GitHub è un servizio di hosting per progetti software, recentemente acquistato da Microsoft. Abbiamo utilizzato GitHub perché è quello più diffuso nella community degli sviluppatori ed è quello che conosciamo meglio.

Tra le varie funzionalità messe a disposizione da GitHub, abbiamo utilizzato in maniera particolare le GitHub Actions per l'esecuzione dei test automatici.

4.8.5 Balsamiq

Per la fase iniziale di design dell'applicazione abbiamo sfruttato Balsamiq, una applicazione di wireframing piuttosto nota.

Balsamiq ci ha permesso di focalizzare la fase di design solamente sulle funzioni che la nostra applicazione web doveva supportare, astraendo dallo stile che avrebbe avuto, scelto in un secondo momento in base alle esigenze che ci siamo trovati ad affrontare.

Capitolo 5

Codice

5.1 Client

Il client è stato sviluppato utilizzando la libreria React affiancata da Redux.

Il codice è stato organizzato all'interno delle seguenti subdirectory:

- **src/components** contiene tutto ciò che riguarda la parte di view dell'applicazione: componenti react, elementi di navigazione, componenti material, file di stile .css e .scss
- **src/redux** contiene tutto ciò che riguarda la gestione dello stato e delle azioni effettuate dall'utente.
- **src/services** contiene tutta la parte di chiamate http fatte al server
- **src/common** contiene gli elementi comuni alle varie subdirectory, come costanti utilizzate lato view e store
- **src/test** contiene la parte di unit test

5.1.1 Redux

L'intero stato dell'applicazione è stato suddiviso in sottostati, divisi in base alle funzionalità ed elementi del sistema.

```
const rootReducer = combineReducers({
  user: userReducer,
  catalog: catalogReducer,
  cart: cartReducer,
  catalogOperations: catalogOperationsReducer,
  alert: alertReducer,
  adminData: adminReducer,
  orders: ordersReducer
})
```

Listato 5.1: Codice che combina i vari sotto-stati in un unico stato globale

- **user** parte di stato contenente le informazioni sull'utente loggato
- **catalog** parte di stato contenente le informazioni sui prodotti e categorie
- **cart** parte di stato contenente i dati dei prodotti aggiunti al carrello dall'utente, con relativi prezzi e quantità.
- **catalogOperations** parte di stato contenente le informazioni di editing effettuate dall'amministratore sul catalogo.
- **alert** parte di stato che contiene i messaggi informativi di successo o errore da mostrare all'utente.
- **adminData** parte di stato contenente di dati a solo uso dell'amministratore, come timetable e lista di rider
- **orders** parte di stato contenente le informazioni sia sullo storico ordini che su quelli realtime utilizzati da admin e rider.

Lo stato viene aggiornato per mezzo dei reducer, ognuno associato a uno dei sottostati. Ogni nuovo stato emesso viene calcolato a partire dallo stato attuale e dall'azione ricevuta in input.

In figura 5.2 un esempio di reducer utilizzato in app.

```
export const INITIAL_STATE = {
  categories: [],
  products: [],
  productDetails: null,
  loading: false,
  error: null
}

const catalogReducer = (state = INITIAL_STATE, action = {}) => {
  switch (action.type) {
    case CatalogActionType.FETCH_CATEGORIES_SUCCESS:
      return {
        ...state,
        categories: action.payload,
        loading: false,
        error: null
      }
    case CatalogActionType.FETCH_CATEGORIES_PENDING:
      return {
        ...state,
        loading: true,
        error: null
      }
    case CatalogActionType.FETCH_CATEGORIES_FAILED:
      return {
        ...state,
        loading: false,
        error: action.payload
      }
  }
}
```

Listato 5.2: Catalog reducer

In figura 5.3 un esempio di action creator usato in app, corrispondente all’azione di caricamento delle categorie dal server. Emette come prima cosa un’azione che notifica il caricamento in corso delle categorie (utilizzato lato ui per mostrare i loader di caricamento), successivamente una volta ottenuta risposta dal server, lancia un’azione di successo contenente i dati della risposta o un’azione di errore (usata poi per mostrare messaggi o schermate di errore all’utente).

```

export const fetchCategories = () => {
  return (dispatch, getState) => {
    if (getState().catalog.categories.length) return
    dispatch(fetchCategoriesPending())
    return catalogService.fetchCategories()
      .then(result =>
        dispatch(fetchCategoriesSuccess(result.data)))
      .catch(error =>
        dispatch(fetchCategoriesFailed(mapNetworkError(error))))
  }
}

const fetchCategoriesSuccess = (categories) => {
  return {
    type: CatalogActionType.FETCH_CATEGORIES_SUCCESS,
    payload: categories
  }
}

const fetchCategoriesFailed = (error) => {
  return {
    type: CatalogActionType.FETCH_CATEGORIES_FAILED,
    payload: error
  }
}

const fetchCategoriesPending = () => {
  return {
    type: CatalogActionType.FETCH_CATEGORIES_PENDING
  }
}

```

Listato 5.3: Catalog actions

Di default le azioni gestite dallo store sono dei semplici oggetti javascript, per poter supportare l'emissione di funzioni (come nel caso della *fetchCategories*) abbiamo fatto uso della libreria **redux-thunk**, che funge da middleware che individua le azioni emesse di tipo funzione eseguendole e riemettendo sullo store le azioni da loro generate.

5.1.2 View

Come accennato in precedenza, la parte di view dell'applicazione è mantenuta nel package `components`. È composta principalmente dai componenti React e dai loro fogli di stile. Un componente react può rappresentare una serie di schermate, una singola schermata, una sua porzione o un singolo elemento di view.

Ad ogni componente, in base alle necessità, gli vengono iniettate parti di stato o funzioni da eseguire in risposta ad azioni compiute dall'utente, come nell'esempio riportato in figura 5.4. La funzione `mapStateToProps` serve per estrarre le variabili necessarie al componente a partire dallo stato dello store. La funzione `mapDispatchToProps` invece serve per selezionare le funzioni che emetteranno azioni sullo store da far compiere al componente.

```
const mapDispatchToProps = dispatch => {
  return {
    completeOrder: (orderData) =>
      dispatch(completeOrder(orderData)),
    fetchTodayTimetable: () => dispatch(fetchTodayTimetable()),
    clearOrderData: () => dispatch(clearOrderData())
  }
}

const mapStateToProps = state => {
  return {
    total: state.cart.total,
    timeSlots: state.cart.timetable,
    orderCompleted: state.cart.orderCompleted,
    user: state.user
  }
}

export default withRouter(connect(mapStateToProps,
  mapDispatchToProps)(OrderSummaryPage))
```

Listato 5.4: Parte di export del componente di riepilogo ordine

5.2 Server

Il codice del server è stato realizzato utilizzando javascript e facendo uso delle librerie:

- **express** utilizzato per la realizzazione delle api e per la gestione delle varie rotte.
- **mongoose** per l'interazione con il database documentale mongodb
- **socket.io** per la realizzazione di funzionalità di interazione realtime tra client e server.
- **jsonwebtoken** per la generazione dei token utilizzati per l'autenticazione utente.
- **bcrypt** utilizzato per criptare le credenziali utente.

Il codice è poi stato organizzato nelle seguenti sotto directory principali a partire dalla cartella *src*:

- **models** contiene tutti gli schemi del database mongodb.
- **controllers** contiene tutte le funzioni invocate a seguito dell'invio di chiamate in corrispondenza delle rotte gestite dall'applicazione. Operano in lettura e scrittura sul database mongo e ritornano risposte
- **routes** contiene il codice responsabile della gestione dei vari endpoint, collegando ciascuno di essi al metodo appropriato presente in **controllers**
- **middlewares** funzioni che nella gestione della rotta, precedono l'invocazione della funzione sul controller. Quelli sviluppati hanno lo scopo di autenticare alcune delle rotte, risolvendo e verificando il token associato alla chiamata, facendo quindi procedere all'esecuzione del metodo del controller in caso di successo o ritornando immediatamente un errore 401 in caso negativo.
- **common** contiene costanti condivise.
- **test** contiene tutti i file di test, descritti in dettaglio nel capitolo successivo.

5.2.1 Models

Di seguito verrà mostrata la struttura dei principali schemi del db mongo. Su ogni schema è poi presente un campo `id` generato automaticamente da mongo, che permette di identificare ogni singola entità di una collezione.

```
{
  name: { type: String, required: true },
  image: { type: String, require: true },
}
```

Listato 5.5: Category schema

La categoria ha il solo scopo di raggruppare prodotti simili, ha quindi un nome e un'immagine.

```
{
  name: { type: String, required: true },
  categoryId: { type: mongoose.ObjectId, required: true },
  categoryName: { type: String, required: true },
  description: { type: String },
  image: { type: String },
  price: { type: Number, required: true },
  ingredients: [String]
}
```

Listato 5.6: Product schema

Il prodotto ha diversi campi descrittivi e alcuni campi che riportano informazioni della categoria a cui appartiene. Queste informazioni sono state aggiunte direttamente sul prodotto per rendere più veloci le operazioni di lettura. Queste informazioni vengono poi aggiornate ogni qualvolta la categoria viene modificata.

```
{
  name: { type: String, required: true, trim: true },
  surname: { type: String, required: true, trim: true },
  email: { type: String, required: true, lowercase: true },
  password: { type: String, required: true, minLength: 8 },
  tokens: [
    token: { type: String, required: true }
  ],
  cart: [
    productId: { type: mongoose.ObjectId, required: true },
    quantity: { type: Number, required: true },
    name: {
      type: String,
      required: true
    }
  ]
}
```

```

        type: String, required: true
    },
    image: { type: String },
    price: { type: Number }
],
role: { type: String, required: true, }
)
}

```

Listato 5.7: User schema

Lo schema User modella l'utente del sistema in tutti i suoi ruoli. Mantiene le credenziali criptate, i token di accesso e, nel caso dell'utente customer, il carrello.

Il carrello mantiene i riferimenti ai prodotti aggiunti, insieme ad alcune delle loro informazioni, duplicate all'interno di questo modello in modo simile a ciò che è stato fatto sul prodotto in riferimento alla categoria, vengono quindi aggiornate nel momento in cui il prodotto viene modificato dall'admin.

Evitare la modellazione di questo aspetto avrebbe costretto di fare ulteriori accessi al database per recuperare le informazioni di ogni prodotto presente a carrello, azione molto frequente durante l'utilizzo dell'applicazione da parte dell'utente. Viceversa, l'azione di modifica di un prodotto viene resa sicuramente più dispendiosa a causa di questa modellazione, ma giustificata dalla sua frequenza molto più bassa (le modifiche al catalogo prodotti vengono effettuate molto raramente).

```
{
  userFullName: { type: String, required: true, },
  totalPrice: { type: Number, required: true, },
  address: { type: String, required: true, },
  city: { type: String, required: true, },
  creationDate: { type: Date, required: true, },
  date: { type: Date, required: true, },
  time: { type: String, required: true, },
  userId: { type: String, required: true, },
  telephoneNumber: { type: String, required: true, },
  state: { type: String, required: true, },
  paymentType: { type: String, required: true, },
  rider: {
    id: { type: String },
    name: { type: String },
    surname: { type: String }
  },
  products: [
    name: { type: String, required: true, }
  ]
}
```

```

        price: { type: Number, required: true, },
        quantity: { type: Number, required: true, }
    }]
})

```

Listato 5.8: Order schema

Lo schema Order modella l'ordine effettuato dall'utente customer. Riporta tutte le informazioni riguardanti la vendita, lo stato di consegna, il rider responsabile per il trasporto e l'elenco prodotti acquistato (stavolta il dato non viene mantenuto aggiornato in quanto serve solo l'informazione presente al momento dell'acquisto).

5.3 Funzionalità di gestione degli ordini realtime

La funzionalità di aggiornamento degli ordini realtime permette all'utente admin di vedere gli ordini in arrivo e i loro cambi di stato aggiornati automaticamente senza il bisogno di ricaricare la pagina e permette all'utente rider di vedere allo stesso modo gli ordini da consegnare.

Lato client la funzionalità è gestita passivamente: ci si mette in ascolto in corrispondenza degli eventi di creazione e aggiornamento degli ordini e si emette l'azione corrispondente sullo store, aggiornando quindi lo stato e di conseguenza la schermata.

Tutte le operazioni sugli ordini vengono effettuate invece tramite chiamate http.

Il server, diversamente dal client, opera sulla socket in maniera attiva, emettendo sulla socket. Questa logica è gestita a partire dalle funzioni presenti in *orderController* appoggiandosi ai metodi della classe *LiveOrdersHandler* che include tutti i metodi necessari per operare sulla socket: il server esegue le operazioni di creazione o aggiornamento ordini a seguito delle chiamate http in entrata, successivamente, dopo aver aggiornato il db, comunica l'informazione ai client collegati alla socket in base al ruolo ruolo. Un rider quindi riceverà solo gli eventi di aggiornamento ordine che lo riguardano (ordine assegnato o rimosso) mentre l'admin riceverà qualsiasi tipo di informazione.

Capitolo 6

Test

Lo sviluppo dell'applicazione è stato affiancato da un'attività di test sia a livello di codice che con utenti esterni, non avendo dei committenti o dei clienti finali abbiamo approfittato del maggior tempo libero di familiari e amici per avere indicazioni aggiuntive su funzionamento e design dell'applicazione.

6.1 Client

Il testing del client è stato fatto nella parte più importante e centrale dell'applicazione: la parte relativa alla gestione dello stato e delle azioni. Come già spiegato in precedenza, il funzionamento di Redux si basa sul concetto di immutabilità e di funzioni pure.

Sono quindi stati realizzati test sia sui reducer che sugli action creator, di seguito alcuni esempi presi dal progetto.

```
const result = { products: [] }
const expectedActions = [
  { type: CartActionTypes.FETCH_CART_START },
  { type: CartActionTypes.FETCH_CART_SUCCESS, payload: result }
]
const store = mockStore(stateWithToken)
axios.get.mockImplementation(() => Promise.resolve({ data: result }))

return store.dispatch(fetchCart())
  .then(() => {
    expect(store.getActions()).toEqual(expectedActions)
  })
```

Listato 6.1: Test dell'azione di caricamento del carrello

La figura 6.1 mostra il codice che verifica le azioni emesse dall'action creator in caso di richiesta dei dati del carrello (effettuata con la `fetchCart()`) avvenuta con successo.

Si crea uno store fittizio con uno certo stato iniziale, si mocka la chiamata in maniera che ritorni un risultato prevedibile, e infine viene lanciata l'azione e verificate che le azioni corrispondenti create siano quelle attese.

```
const prevState = cartInitialState
const products = ['prod1', 'prod2']
const total = 20
expect(cartReducer(prevState, {
    type: CartActionTypes.FETCH_CART_SUCCESS,
    payload: { cartProducts: products, total: total }
})).toEqual({
    ...prevState,
    products,
    total,
    loading: false,
    error: null
})
```

Listato 6.2: Test dell'aggiornamento di stato in seguito al caricamento del carrello

La figura 6.2 mostra il codice di test del reducer associato al carrello, qui viene verificato che dato lo stato precedente dello store e l'azione in input contenente venga generato un nuovo stato con le informazioni aggiornate correttamente. Sul repo di Github è sono state poi abilitate le Github Actions in maniera da eseguire tutti test ad ogni push.

6.2 Server

Per quanto riguarda il testing lato server, ad essere testato invece è stato il funzionamento delle api esposte dal server. Per ogni servizio sono stati creati dei test automatici che, una volta eseguiti, verificano lo stato della risposta, i valori ritornati e gli effetti della chiamata sul database.

Per l'esecuzione dei test abbiamo utilizzato il package `MongoMemoryServer` che utilizzato da `Mongoose` permette di avviare un processo che emula una istanza temporanea di MongoDB, senza che i test vadano a sporcare la base dati reale.

Questo approccio, rispetto alla creazione di mock che simulassero le chiamate api, sin dall'inizio ha permesso di testare il funzionamento dei servizi senza

dover modificare null’altro. Nel caso dei mock infatti, al cambio di un servizio avremmo dovuto modificare anche il rispettivo mock, impiegando parecchio tempo.

```
const mongoose = require('mongoose');
const { MongoMemoryServer } = require('mongodb-memory-server');

const mongod = new MongoMemoryServer();

/**
 * Connect to the in-memory database.
 */
module.exports.connect = async () => {
    const uri = await mongod.getConnectionString();

    const mongooseOpts = {
        useNewUrlParser: true,
        autoReconnect: true,
        reconnectTries: Number.MAX_VALUE,
        reconnectInterval: 1000
    };

    await mongoose.connect(uri, mongooseOpts)
```

Listato 6.3: Connessione a *MongoMemoryServer* tramite *Mongoose*

Come visibile dal codice in figura 6.3, la connessione a *MongoMemoryServer* è del tutto simile a quella di MongoDB, ma l'url del database viene restituito dal processo mongod, il demone che si occuperà delle letture e scritture in memoria.

I test sono stati eseguiti grazie alla libreria *SuperTest* che permette di fare le chiamate alle api in maniera asincrona e alla loro risposta verificarne stato e dati ritornati.

I test possono essere scritti in maniera molto discorsiva, così da poter verificare anche al loro lancio che gli scenari testati e i risultati attesi siano corretti.

```
describe('User services', () => {
    it('Should signin user', async () => {
        const res = await request(app)
            .post('/auth/signin')
            .send({
                email: consumerData.email,
                password: consumerData.password
```

```

        })

    expect(res.statusCode).toEqual(200)
    expect(res.body).toHaveProperty('token')
    token = res.body.token
})

it('Should logout user', async () => {
  const res = await request(app)
    .post('/auth/logout')
    .set('Authorization', 'Bearer ' + token)

  expect(res.statusCode).toEqual(200)
})
}
)

```

Listato 6.4: Test servizi di sign-in e logout

Con il metodo `request()` creiamo la richiesta asincrona, mentre con `expect()` verifichiamo i valori attesi e quelli ricevuti in risposta. Se anche uno solo degli `expect()` fallisce, allora il test non viene considerato superato.

6.3 Test con gli utenti

Purtroppo non è stato possibile confrontarsi con attività reali. Tuttavia abbiamo eseguito i test con utenti fisici con ruoli diversi tra loro, per verificare il funzionamento del sistema in uno scenario simile a quello reale.

In particolar modo abbiamo verificato con attenzione l'effettiva ricezione dell'ordine ed il corretto aggiornamento dello stato di quest'ultimo (aggiornato realtime) effettuato dai fattorini e dagli amministratori.

I test sul campo sono stati fatti in rete locale con diversi tipi di device quali tablet, laptop, computer fissi e smartphone cercando di simulare l'utilizzo reale dell'applicazione

Capitolo 7

Deployment

7.1 Rilascio, installazione e messa in funzione

I sorgenti del progetto possono essere consultati presso gli indirizzi <https://github.com/MatteoCaval/progetto-web-backend> e <https://github.com/MatteoCaval/progetto-web-frontend>.

Abbiamo deciso di mantenerli su repository separate per avere maggior indipendenza tra i due applicativi (nel caso in futuro ad esempio si volesse sostituire il client web con un client di un altro tipo come un'applicazione nativa Android o iOS).

Sui due progetti sono poi stati configurati dei task automatici utilizzando **Github Actions** per la creazione e l'upload delle immagini docker direttamente su **Docker Hub** ad ogni push sul branch **master**.

Per la messa in funzione automatica dell'intero sistema è necessario effettuare la clone del seguente repository <https://github.com/MatteoCaval/uniburger.git> e lanciare i comando `docker-compose up`, l'applicazione potrà poi essere raggiunta da browser all'url `http://localhost:3000/`

Il file `docker-compose.yml` utilizzato da `docker-compose` si trova in un repository a parte rispetto a quelli del client e del server, in quanto sfrutta 4 immagini distinte.

- **server** - immagine generata dalla build del progetto server, caricata su Docker Hub.
- **client** - immagine generata dalla build del progetto client, anch'essa caricata su Docker Hub.

- **mongo** - immagine di MongoDB, a cui in composizione fa riferimento **server**, che in input prende l'URI del database contenuto in questa immagine.
- **mongo-seed** - immagine di una libreria già presente su Docker Hub che fa riferimento a mongo. Lanciata subito dopo la creazione di MongoDB, esegue il comando **mongo-restore** caricando nel database il dump del database presente nella cartella /uniburger

Nel caso fosse ancora live una versione funzionante del sito è poi disponibile alla pagina `139.59.137.102:3000`, generata utilizzando i servizi messi a disposizione dalla piattaforma <https://www.digitalocean.com/>, con cui a partire da una droplet abbiamo installato il sistema usando docker compose come descritto in precedenza.

7.1.1 Credenziali di prova degli utenti

Per accedere e testare il sistema con i vari tipi di utente si possono usare le credenziali dei seguenti utenti già registrati oltre a creare di nuovi:

- **Admin** admin@uniburger.it pw: admin
- **Customer** utente1@uniburger.it pw: utente - utente2@uniburger.it pw: utente
- **Rider** rider1@uniburger.it pw: rider - rider2@uniburger.it pw: rider

Capitolo 8

Conclusioni

Lo svolgimento di questo progetto è stata una bella sfida. Si trattava infatti della nostra prima esperienza nello sviluppo di una applicazione client-server con stack MERN, per noi nuovo praticamente in tutte e 4 le sue componenti.

Abbiamo sfrutto lo sviluppo di questo progetto per mettere in campo la maggior parte delle librerie che ci avevano incuriosito durante le lezioni o di cui avevamo sentito parlare. Alla fine si è rivelata una mossa vincente, perché abbiamo potuto approfondirne le funzionalità e i vantaggi e svantaggi che portavano con sé.

L'utilizzo di un database NoSQL come Mongo ci ha spinto a dover rivedere più volte la struttura dati, in particolar modo denormalizzando i dati, favorendo la semplicità delle query a costo di duplicare documenti. Per fare la scelta giusta abbiamo dovuto ragionare sui dati che per ogni servizio ci sarebbero serviti e della frequenza con cui avremmo letto un documento rispetto alle volte in cui avremmo dovuto aggiornarlo in ciascuna delle sue copie.

L'applicazione web che è venuta fuori è una app che, pure nella sua semplicità, con qualche modifica sarebbe pronta a passare in produzione ed essere utilizzata in uno scenario reale, gestendo caricamenti dovuti al delay della rete, autenticazione delle chiamate con Bearer Token e notifica dell'assegnazione degli ordini e delle consegne realtime.

Possiamo quindi ritenerci soddisfatti del lavoro svolto, con l'intento di proseguire nello sviluppo del progetto, sperando che un giorno possa veramente essere adottata da qualche negozio.

