Specifica Tecnica

RAMtastic6

29 maggio 2024

email: ramtastic6@gmail.com

Informazioni sul documento

Versione:	1.0.0
Redattori:	Zaupa R. Visentin S. Brotto D. Basso L. Tonietto F. Zambon M.
Verificatori:	Brotto D. Visentin S. Zambon M. Zaupa R. Tonietto F.
Destinatari:	Vardanega T., Cardin R., Imola Informatica
Uso:	Esterno

Registro dei Cambiamenti - Changelog

Versione	Data	Autore	Verificatore	Dettaglio
v 1.0.0	2024-05-29	Brotto D.	Brotto D.	Approvazione e validazione del documento
v 0.5.0	2024-05-28	Visentin S.	Tonietto F.	Modificata *architettura*$_G$ *websocket*$_G$ e aggiunta sezione database
v 0.4.6	2024-05-28	Brotto D.	Tonietto F.	Specificato modulo staff; aggiunti diagrammi *UML*$_G$ (con descrizioni) di craete_reservation, user, admin e notifications
v 0.4.5	2024-05-28	Zambon M.	Tonietto F.	Aggiunto e descritto l'*UML*$_G$ del sign-up-admin
v 0.4.4	2024-05-27	Zaupa R.	Tonietto F.	Descritto il modulo Notification del *backend*$_G$ e sistemato lo stato dei requisiti funzionali
v 0.4.3	2024-05-26	Zambon M.	Zaupa R.	Descritto il *Backend*$_G$ for *Frontend*$_G$ pattern, il Compound Components pattern e il Controlled inputs pattern (sezione 4.2.2)
v 0.4.2	2024-05-26	Visentin S.	Zaupa R.	Modificato il diagramma delle classi del *backend*$_G$
v 0.4.1	2024-05-26	Zambon M.	Zaupa R.	Aggiunti gli endpoints per User (sezione 4.1.1)
v 0.4.0	2024-05-25	Visentin S.	Zambon M.	Aggiunti gli endpoints per Restaurant, Reservation, Orders e Food (sezione 4.1.1)
v 0.3.3	2024-05-25	Tonietto F.	Zambon M.	Aggiunta la descrizione del Publisher-Subscriber pattern relativa al Websocket (4.2.3.1)
v 0.3.2	2024-05-25	Basso L.	Zambon M.	Aggiornata la sezione requisiti; inserite le sezioni riguardati i moduli backend Authentication, Menu, DaysOpen; inserito diagramma *UML*$_G$ del server socket (sezione 4.1.3); inseriti *UML*$_G$ pagina order e order checkout (sezione 4.1.2.5) con relativa descrizione.
v 0.3.1	2024-05-25	Visentin S.	Zambon M.	Stesura del Module pattern (sezione 4.2.1.5)
v 0.3.0	2024-05-24	Tonietto F.	Zambon M.	Aggiunto l'*UML*$_G$ del Login (*frontend*$_G$).
v 0.2.2	2024-05-24	Visentin S.	Zambon M.	Continuata la stesura della sezione 3.3, aggiuniti *Repository*$_G$ pattern e Controller-Service pattern
v 0.2.1	2024-05-24	Visentin S.	Zambon M.	Aggiunta sezione 3.3 (Esecuzione dei *test*$_G$), dependy injection e decorator pattern

v 0.2.0	2024-05-23	Tonietto F.	Visentin S.	Aggiunte le motivazioni relative alla scelta di Socket.io come libreria per l'implementazione del Websocket (2.1.6). Aggiunto l'uso di un pacchetto per scrivere frammenti di codice.
v 0.1.4	2024-05-22	Zaupa R.	Visentin S.	Soddisfatti ROF12 ROF13 ROF14 ROF19 ROF30 ROF31 ROF34
v 0.1.3	2024-05-22	Brotto D.	Visentin S.	Soddisfatto ROF57
v 0.1.2	2024-05-13	Basso L.	Visentin S.	Aggiunta alla sezione 4 il diagramma del *sistema*$_G$ di notifica.
v 0.1.1	2024-05-13	Brotto D.	Visentin S.	Aggiunta sezione 3
v 0.1.0	2024-05-13	Visentin S.	Brotto D.	Aggiungo schema ER del database
v 0.0.3	2024-05-13	Zaupa R.	Brotto D.	Prima stesura della sezione "Stato dei requisiti funzionali" e della sottosezione "*Architettura*$_G$ di *deployment*$_G$"
v 0.0.2	2024-05-07	Zaupa R.	Brotto D.	Prima stesura della sezione "*Tecnologie*$_G$"
v 0.0.1	2024-05-04	Zaupa R.	Brotto D.	Prima versione, stesura della sezione "Introduzione"

Indice

1 Introduzione **6**
1.1 Scopo del documento . . . 6
1.2 Scopo del prodotto . . . 6
1.3 Riferimenti . . . 6
1.3.1 Riferimenti normativi . . . 6
1.3.2 Riferimenti informativi . . . 6

2 Tecnologie **8**
2.1 Tecnologie implementative . . . 8
2.1.1 Next.js . . . 8
2.1.2 React . . . 8
2.1.3 Tailwind CSS . . . 8
2.1.4 Node.js . . . 8
2.1.5 Nest.js . . . 9
2.1.6 Socket.io . . . 9
2.2 Tecnologie per la persistenza dei dati . . . 9
2.2.1 PostgreSQL . . . 9
2.3 Tecnologie per il testing . . . 10
2.3.1 Jest . . . 10
2.4 Tecnologie per il deployment . . . 10
2.4.1 Docker . . . 10
2.4.2 Docker Compose . . . 10

3 Esecuzione del software **11**
3.1 Download del progetto . . . 11
3.2 Avvio dell'applicazione . . . 11
3.2.1 Docker (consigliato) . . . 11
3.2.2 NPM . . . 12
3.3 Esecuzione dei test . . . 13

4 Architettura **14**
4.1 Architettura logica . . . 14
4.1.1 Backend . . . 14
4.1.1.1 Authentication . . . 15
4.1.1.2 Daysopen . . . 16
4.1.1.3 Food . . . 16
4.1.1.4 Menu . . . 16
4.1.1.5 Notification . . . 16
4.1.1.6 Orders . . . 17
4.1.1.7 Reservation . . . 19
4.1.1.8 Restaurant . . . 22
4.1.1.9 Staff . . . 24
4.1.1.10 User . . . 24
4.1.2 Frontend . . . 25
4.1.2.1 Admin . . . 26
4.1.2.2 Create_reservation . . . 27
4.1.2.3 Login . . . 28
4.1.2.4 Notifications . . . 28

4.1.2.5 Order . . . 29
4.1.2.6 Sign_up . . . 29
4.1.2.7 Sign_up_admin . . . 30
4.1.2.8 User . . . 30
4.1.3 Websocket . . . 32
4.1.3.1 Componenti Principali . . . 32
4.1.4 Database . . . 33
4.2 Design Pattern utilizzati . . . 34
4.2.1 Backend . . . 34
4.2.1.1 Dependency Injection . . . 34
4.2.1.2 Decorator pattern . . . 35
4.2.1.3 Repository pattern . . . 37
4.2.1.4 Controller-Service pattern . . . 38
4.2.1.5 Module pattern . . . 39
4.2.2 Frontend . . . 40
4.2.2.1 Backend for frontend . . . 40
4.2.2.2 Compound Components . . . 42
4.2.2.3 Controlled inputs . . . 43
4.2.3 Websocket . . . 45
4.2.3.1 Publish-Subscribe pattern . . . 45
4.3 Architettura di deployment . . . 47
4.3.1 Vantaggi . . . 48

5 Stato dei requisiti funzionali **49**

1 Introduzione

1.1 Scopo del documento

Il presente documento ha lo scopo di descrivere in modo dettagliato le scelte progettuali effettuate dal gruppo RAMtastic6 per la realizzazione del *sistema*$_G$ *Easy Meal*$_G$ richiesto dall'azienda *Imola Informatica*$_G$. Viene compresa l'*architettura*$_G$ logica e l'*architettura*$_G$ di *deployment*$_G$ oltre che la lista delle *tecnologie*$_G$ utilizzate e i *design*$_G$ pattern adottati.
Inoltre, viene fornita una sezione relativa al tracciamento dei requisiti soddisfatti in linea con il documento di Analisi dei Requisiti.
Eventuali termini tecnici sono definiti all'interno del documento "Glossario Tecnico".

1.2 Scopo del prodotto

Il prodotto finale, realizzato tramite un'*Applicazione Web Responsive*$_G$, si propone di realizzare un *software*$_G$ innovativo volto a semplificate il *processo*$_G$ di *prenotazione*$_G$ e *ordinazione*$_G$ nei ristoranti, contribuendo a migliorare l'esperienza per clienti e ristoratori. In particolare, *Easy Meal*$_G$ dovrà consentire agli utenti di personalizzare gli ordini in base alle proprie preferenze, allergie ed esigenze alimentari; interagire direttamente con lo staff del ristorante attraverso una chat integrata; consentire di dividere il conto tra i partecipanti al tavolo.

1.3 Riferimenti

1.3.1 Riferimenti normativi

1. Presentazione del *capitolato*$_G$ d'appalto C3 - Progetto *Easy Meal*$_G$:
`https://www.math.unipd.it/~tullio/IS-1/2023/Progetto/C3.pdf`
(Consultato: 2024-05-13)

2. Regolamento del progetto didattico:
`https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/PD2.pdf`
(Consultato: 2024-05-13)

3. *Norme di Progetto*$_G$ v2.0.0.

1.3.2 Riferimenti informativi

1. Glossario v2.0.0;

2. Analisi dei Requisiti v3.0.0.

3. Lezione *"Progettazione e programmazione: diagrammi delle classi (UML*$_G$*)"* del corso di Ingegneria del *Software*$_G$:
`https://www.math.unipd.it/~rcardin/swea/2023/Diagrammi%20delle%20Classi.pdf`
(Consultato: 2024-05-13)

4. Lezione *"Progettazione: i pattern architetturali"* del corso di Ingegneria del *Software*$_G$:
`https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf`
(Consultato: 2024-05-13)

5. Lezione *"Progettazione: il pattern Dependency Injection"* del corso di Ingegneria del *Software*$_G$:
`https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Architetturali%`

20-%20Dependency%20Injection.pdf
(Consultato: 2024-05-13)

6. Lezione *"Progettazione software (T6)"* del corso di Ingegneria del *Software*$_G$:
`https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/T6.pdf`
(Consultato: 2024-05-13)

7. Lezione *"Progettazione: il pattern Model-View-Controller e derivati"* del corso di Ingegneria del *Software*$_G$:
`https://www.math.unipd.it/~rcardin/sweb/2022/L02.pdf`

8. Lezione *"Progettazione: i pattern creazionali (GoF)"* del corso di Ingegneria del *Software*$_G$:
`https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf`
(Consultato: 2024-05-13)

9. Lezione *"Progettazione: i pattern strutturali (GoF)"* del corso di Ingegneria del *Software*$_G$:
`https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Strutturali.pdf`
(Consultato: 2024-05-13)

10. Lezione *"Progettazione: i pattern di comportamento (GoF)"* del corso di Ingegneria del *Software*$_G$:
`https://www.math.unipd.it/~rcardin/swea/2021/Design%20Pattern%20Comportamentali_4x4.pdf`
(Consultato: 2024-05-13)

11. *Nest.js*$_G$ — Architectural Pattern, Controllers, Providers, and Modules:
https://medium.com/geekculture/nest-js-architectural-pattern-controllers-providers-and-modules-406d9b192a3a
(Consultato: 2024-05-29)

12. Topic-based Publish/Subscribe *design*$_G$ pattern implementation in OCaml — (Using socket programming)
https://medium.com/@aryangodara_19887/topic-based-publish-subscribe-design-pattern-implementa in-ocaml-using-socket-programming-ba536f0a3ef3
(Consultato: 2024-05-29)

13. *Repository*$_G$ pattern explained with Laravel and NestJS examples
https://draganatanasov.com/2023/01/15/repository-pattern-explained-with-laravel-and-nestjs-examples/
(Consultato: 2024-05-29)

14. Boosting Your NestJS Skills: Exploring the Module Pattern *Design*$_G$
`https://www.bymoji.com/blog/Boosting_Your_NestJS_Skills_Exploring_the_Module_Pattern_Design`
(Consultato: 2024-05-29)

15. *Backend*$_G$ for *Frontend*$_G$ (BFF) Pattern in System Designing
https://blog.bitsrc.io/backend-for-frontend-bff-pattern-in-system-designing-501a71df6bf7
(Consultato 2024-05-29)

16. *React*$_G$ *Design*$_G$ Patterns
https://refine.dev/blog/react-design-patterns
(Consultato 2024-05-29)

2 Tecnologie

In questa sezione viene presentata una panoramica degli strumenti e delle *tecnologie*$_G$ utilizzate per l'implementazione del *sistema*$_G$ denominato *Easy Meal*$_G$.

2.1 Tecnologie implementative

2.1.1 Next.js

Next.js è un *framework*$_G$ di sviluppo per applicazioni basato su *React*$_G$. La sua scelta è stata dettata dalla sua efficienza e dalla sua facilità d'uso. L'utilizzo di tale *framework*$_G$ permette di creare applicazioni veloci e performanti oltre che effettuare il rendering lato server. Offre una supporto integrato per TypeScript, funzionalità avanzate di realizzazione di *API*$_G$, gestione del routing e Server Action.

- **Versione scelta**: 14.0.4
- **Documentazione**: `https://nextjs.org/`
  (Consultato: 2024-05-13)
- **Linguaggio**: TypeScript, *JSX*$_G$, *JSON*$_G$.

2.1.2 React

React è una libreria JavaScript utilizzata per la creazione di interfacce utente dinamiche e reattive.

- **Versione scelta:** 18.0.0
- **Documentazione**: `https://reactjs.org/`
  (Consultato: 2024-05-13)
- **Linguaggio**: TypeScript, *JSX*$_G$, *JSON*$_G$.

2.1.3 Tailwind CSS

Tailwind *CSS*$_G$ è un *framework*$_G$ *CSS*$_G$ utilizzato per lo sviluppo di interfacce web lato utente offrendo classi predefinite per applicare lo stile agli elementi.

- **Versione scelta:** 3.4.1
- **Documentazione**: `https://tailwindcss.com/`
  (Consultato: 2024-05-13)
- **Linguaggio**: CSS

2.1.4 Node.js

Node.js è un *runtime system*$_G$ *open-source*$_G$ orientato agli eventi che esegue codice JavaScript *server-side*$_G$.

- **Versione scelta:** 21.0.0
- **Documentazione**: `https://nodejs.org/`
  (Consultato: 2024-05-13)
- **Linguaggio**: TypeScript, JSON

2.1.5 Nest.js

Nest.js è un *framework*$_G$ *typescript*$_G$ basato su *Node.js*$_G$ e progettato per sviluppare applicazioni *server-side*$_G$ efficienti, scalabili e manutenibili.

- **Versione scelta:** 10.0.0
- **Documentazione**: `https://nestjs.com/`
  (Consultato: 2024-05-13)
- **Linguaggio**: TypeScript, JSON

2.1.6 Socket.io

Socket.io è una libreria JavaScript che permette la comunicazione in tempo reale bidirezionale tra client e server tramite *websocket*$_G$, con supporto per la trasmissione di eventi. È progettata per facilitare lo sviluppo di applicazioni web interattive e collaborative, come chat in tempo reale, giochi multiplayer e applicazioni di monitoraggio in tempo reale.
In particolare, la scelta di utilizzare Socket.io è stata dettata da vari fattori. Innanzitutto, essendo giunta più in là nello sviluppo la necessità di utilizzare una libreria a parte per la realizzazione di alcuni requisiti obbligatori, ci si è orientati verso una libreria che fosse nativamente supportata dai *framework*$_G$ utilizzati per *Frontend*$_G$ e *Backend*$_G$. Altri aspetti che hanno portato il gruppo a scegliere la sua adozione sono stati l'ampia documentazione e la gestione efficace di connessioni, disconnessioni e riconoscimento degli eventi fornita dalla libreria.

- **Versione scelta:** 4.0.0
- **Documentazione**: `https://socket.io/docs/v4/`
  (Consultato: 2024-05-13)
- **Linguaggi**: JavaScript, TypeScript

2.2 Tecnologie per la persistenza dei dati

2.2.1 PostgreSQL

PostgreSQL è un *sistema*$_G$ di gestione di database relazionali *open-source*$_G$, noto per la sua affidabilità e robustezza.

- **Versione scelta:** 15.2
- **Documentazione**: `https://www.postgresql.org/`
  (Consultato: 2024-05-13)
- **Linguaggi**: SQL

2.3 Tecnologie per il testing

2.3.1 Jest

Jest è un *framework*$_G$ di *test*$_G$ del *software*$_G$ *open-source*$_G$ per JavaScript, sviluppato da Facebook. È progettato per essere semplice da utilizzare e offre un'esperienza di sviluppo fluida e piacevole per i *test*$_G$ unitari, di integrazione e di *sistema*$_G$.

- **Versione scelta:** 29.7.0
- **Documentazione**: `https://jestjs.io/docs/getting-started`
  (Consultato: 2024-05-13)
- **Linguaggi**: TypeScript, JavaScript

2.4 Tecnologie per il deployment

2.4.1 Docker

Docker è un *software*$_G$ utilizzato per effettuare il *deployment*$_G$ di un prodotto *software*$_G$. La sua caratteristica è di permettere la scalabilità e l'Isolamento delle applicazioni utilizzando la virtualizzazione a livello di *sistema*$_G$ operativo. Offre una discreta efficienza, portabilità e facilità d'uso.

- **Versione scelta**: latest
- **Documentazione**: `https://www.docker.com/`
  (consultato: 2024-05-13)

2.4.2 Docker Compose

Docker$_G$ Compose è uno *strumento*$_G$ per la definizione e l'esecuzione di applicazioni aventi più' container. Permette di gestire i container *Docker*$_G$ semplificando il *processo*$_G$ di *deployment*$_G$.

- **Versione scelta**: latest
- **Documentazione**: `https://docs.docker.com/compose/`
  (Consultato: 2024-05-13)

3 Esecuzione del software

3.1 Download del progetto

Clonare il *repository*$_G$ del progetto usando il comando:

```
git clone https://github.com/RAMtastic6/EasyMeal.git
cd EasyMeal
```

3.2 Avvio dell'applicazione

3.2.1 Docker (consigliato)

Questo progetto utilizza *Docker*$_G$ Compose per gestire l'avvio dei container *Docker*$_G$. Per avviare i container e utilizzare l'applicazione si seguando le seguenti istruzioni.

- **Avvio dei container**
  Per avviare i container utilizzando *Docker*$_G$ Compose, esegui il seguente comando nella directory del progetto:

  ```
  docker-compose up --build --watch
  ```

  Oppure per far partire i servizi senza che la console di comando attendi la chiusura si usi il comando:

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

  Una volta avviati i container, si potrà accedere all'applicazione utilizzando il browser o gli strumenti di sviluppo appropriati.
  Per esempio per poter accedere al progetto NextJS ci si colleghi al link:

  `http://localhost:3000/create_reservation`

  Si tenga presente che NextJS utilizza la porta 3000, NestJS 6969, Postgres utilizza 7070 e Socket 8000.

- **Modifica dei file Dockerfile e compose.yaml**
  Per applicare le modifiche e far partire il progetto si usi il comando:

  ```
  docker-compose up --build
  ```

3.2.2 NPM

Senza l'utilizzo di *Docker*$_G$ si devono installare manualmente le seguenti *tecnologie*$_G$:

- Node.js
- *npm*$_G$ (Node Package Manager)
- postgresSQL

Si ricorda di importare il dump del database nel proprio computer, per farlo si può usare pgAdmin.

Ora si eseguano le seguenti istruzioni:

1. Installare le dipendenze per il *backend*$_G$ *Nest.js*$_G$:

```
cd nest-js
npm install
```

2. Avviare il server *backend*$_G$ *Nest.js*$_G$:

```
npm run start:dev
```

3. Installare le dipendenze per il *frontend*$_G$ *Next.js*$_G$: Aprire una nuova shell lasciando la precedente in esecuzione

```
cd next-js
npm install
```

4. Avviare il server *frontend*$_G$ *Next.js*$_G$:

```
npm run dev
```

5. Installare le dipendenze per il progetto socket *Nest.js*$_G$:

```
cd websocket-server
npm install
```

6. Avviare il server socket *Nest.js*$_G$:

```
npm run start:dev
```

7. Accedere all'applicazione: Una volta avviati sia il server *backend*$_G$ *Nest.js*$_G$ che il server *frontend*$_G$ *Next.js*$_G$, è possibile accedere all'applicazione utilizzando il browser.
Si apra il browser e si vada all'indirizzo

   `$\textit{http}_G$://localhost:3000/create_reservation`

   per accedere alla pagina di creazione delle prenotazioni.

3.3 Esecuzione dei test

In questa sezione si elencano i comandi per eseguire i $\textit{test}_G$ tramite $\textit{jest}_G$ per tutti i servizi. Per poter eseguire i $\textit{test}_G$ in locale per i tre servizi si dovrà prima installare $\textit{Node.js}_G$ versione 21.x . $\textit{Node.js}_G$ installazione(consultato: 2024-05-27)
I comandi per eseguire i $\textit{test}_G$ dei vari servizi sono gli stessi, si deve solo cambiare la cartella dove vengono eseguiti i comandi:

- Per i $\textit{test}_G$ lato $\textit{backend}_G$:

```
cd nest-js
```

- Per i $\textit{test}_G$ lato $\textit{frontend}_G$:

```
cd next-js
```

- Per i $\textit{test}_G$ del $\textit{websocket}_G$:

```
cd websocket-server
```

Quindi, di seguito, vengono riportati i comandi per eseguire i $\textit{test}_G$:

1. Eseguire il comando, se non fatto in precedenza:

```
npm install
```

2. Eseguire il seguente script per l'esecuzione di tutti i $\textit{test}_G$ generando un report finale:

```
npm run $\textit{test}_G$:cov
```

3. Aprire il seguente file per visualizzare il report:

```
./coverage/coverage.txt
```

4 Architettura

4.1 Architettura logica

Il progetto EasyMeal è strutturato in tre componenti principali:

- **Backend**: Responsabile della gestione della *logica di business*$_G$ e della persistenza dei dati. Il *backend*$_G$ si occupa di stabilire e mantenere la connessione con il database, nonché di fornire i mezzi per salvare e accedere ai dati.
- **Frontend**: Destinato alla presentazione dei dati ottenuti dal *backend*$_G$, il *frontend*$_G$ consente all'utente di interagire con le funzionalità del progetto in modo intuitivo e user-friendly.
- **WebSocket Server**: Abilita la comunicazione in tempo reale tra i diversi client del *frontend*$_G$, permettendo l'aggiornamento istantaneo dei dati e migliorando l'interattività dell'applicazione.

4.1.1 Backend

Nel *backend*$_G$ si è scelto di utilizzare il *framework*$_G$ NestJS, che permette la realizzazione di un servizio *API*$_G$ REST. L'*architettura*$_G$ del *backend*$_G$ è suddivisa nei seguenti strati:

- **Controller Layer**: Si occupa del routing delle richieste. I controller fungono da intermediari tra il client e i servizi, gestendo la presentation layer e instradando le richieste del client ai servizi appropriati.
- **Service Layer**: Gestisce la *logica di business*$_G$ dell'applicazione. Questo strato è responsabile delle operazioni e delle regole di business, garantendo che tutte le operazioni siano eseguite correttamente.
- **Data Access Layer**: Implementato tramite *TypeORM*$_G$ (Object-Relational Mapping) che facilita l'interazione con il database realizzato tramite PostegreSQL. Questo strato gestisce la persistenza dei dati, fornendo i mezzi per salvare, recuperare e manipolare i dati nel database.

L'adozione di NestJS e la suddivisione in questi strati facilitano lo sviluppo, la *manutenzione*$_G$ e la scalabilità del *backend*$_G$ del progetto EasyMeal, conformandosi all'*architettura*$_G$ **multi-tier** a 3-tier.

Di seguito viene riportato il diagramma *UML*$_G$ del *backend*$_G$.

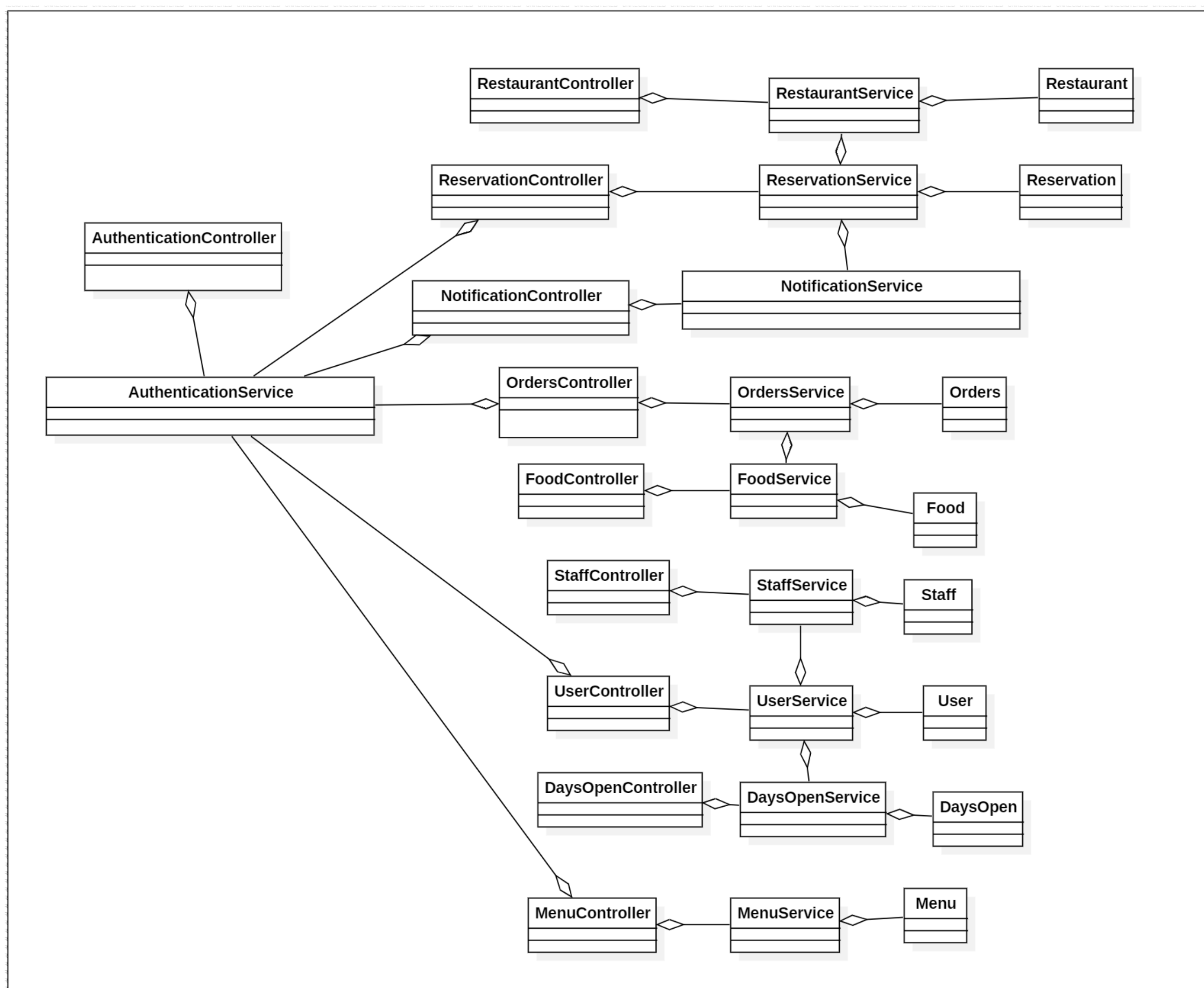

Figura 1: Diagramma delle classi del backend

Di seguito si elencano le *api*$_G$ fornite dal *backend*$_G$ raggruppate per moduli:

4.1.1.1 Authentication

signin

- **Endpoint**: /authentication/signin
- **Metodo**: POST
- **Parametri**: AuthenticationDTO
- **Descrizione**: questa funzione si occupa di prendere in input email e password; nel caso in cui le credenziali siano valide e già presenti all'interno del *sistema*$_G$, viene generato un token in formato JWT_G e ritornato; in caso contrario viene lanciata un'*eccezione*$_G$.

decodeToken :

- **Endpoint**: /authentication/decodeToken
- **Metodo**: POST
- **Parametri**: DecodeTokenDTO

- **Descrizione**: la funzione prende in input una stringa in formato JWT_G; nel caso la stringa non dovesse rispettare il formato, viene lanciata un'$eccezione_G$; in caso contrario, viene verificata la validità del token all'interno del $sistema_G$; nel caso in cui il token sia valido, viene ritornato un oggetto, il quale rappresenta l'id dell'utente e il ruolo di quest'ultimo all'interno del $sistema_G$.

4.1.1.2 Daysopen

create

- **Endpoint**: /daysopen
- **Metodo**: POST
- **Parametri**: CreateDaysOpenDTO
- **Descrizione**: la funzione permette di creare un elenco di giorni di apertura associati ad un determinato ristorante se presente all'interno del $sistema_G$.

4.1.1.3 Food

findOne :

- **Endpoint**: /food/:id
- **Metodo**: GET
- **Parametri**: id: number
- **Descrizione**: questa funzione ritorna il piatto, con i relativi ingredienti, associato all'id.

4.1.1.4 Menu

create

- **Endpoint**: /menu
- **Metodo**: POST
- **Parametri**: CreateMenuDto
- **Descrizione**: la funzione crea un menù (lista di piatti) associato ad un ristorante presente all'interno del $sistema_G$.

4.1.1.5 Notification

create :

- **Metodo**: POST
- **Parametri**: NotificationDto
- **Descrizione**: la funzione crea una notifica utilizzando un oggetto NotificationDto. Innanzitutto viene salvato un oggetto notification nel database, poi viene inviata una notifica al $websocket_G$ effettuando una richiesta POST all'url `http://socket:8000/notification/send`. Infine, viene restituito il messaggio del risultato dell'operazione di salvataggio della notifica.

findAllByUserId :

- **Endpoint**: /notification
- **Metodo**: POST
- **Parametri**: FindAllByUserIdDTO
- **Descrizione**: la funzione cerca tutte le notifiche corrispondenti a un determinato ID utente e restituisce un array di oggetti notifica contenenti le proprietà *id*, *title*, *message* e *status*.

updateStatus :

- **Endpoint**: /notification/update
- **Metodo**: POST
- **Parametri**: updateStatusDTO
- **Descrizione**: la funzione aggiorna lo stato di una notifica specificata tramite l'id. In breve, viene ricercata nel database una notifica corrispondente all'id fornito e se non viene trovata viene restituito null. Se la notifica viene trovata, viene impostato lo stato della notifica come *READ*. Infine, la notifica aggiornata viene salvata nel database e viene restituita.

findOne :

- **Metodo**: POST
- **Parametri**: id
- **Descrizione**: la funzione restituisce la notifica trovata nel database corrispondente all'id fornito.

4.1.1.6 Orders

create :

- **Endpoint**: /orders
- **Metodo**: POST
- **Parametri**: reservation_id: number, food_id: number, token: string
- **Descrizione**: questa funzione salva nel database una nuova *ordinazione*$_G$ associata alla *prenotazione*$_G$ con id uguale a reservation_id.

findAll :

- **Endpoint**: /orders
- **Metodo**: GET
- **Descrizione**: questa funzione ritorna tutti gli ordini presenti nel database.

findOne :

- **Endpoint**: /orders/findOne
- **Metodo**: POST
- **Parametri**: FindOneDTO
- **Descrizione**: questa funzione ritorna la *ordinazione*$_G$ con l'id corrispondente.

addQuantity :

- **Endpoint**: /orders/addQuantity
- **Metodo**: POST
- **Parametri**: AddQuantityDTO
- **Descrizione**: la funzione salva nel database la quantità del piatto appartenente all'*ordinazione*$_G$ appropriata.

updateIngredients :

- **Endpoint**: /orders/updateIngredients
- **Metodo**: POST
- **Parametri**: UpdateIngredientsDTO
- **Descrizione**: questa funzione aggiorna la lista degli ingredienti associati alla *ordinazione*$_G$.

remove :

- **Endpoint**: /orders/remove
- **Metodo**: POST
- **Parametri**: RemoveDTO
- **Descrizione**: questa funzione rimuove dal database l'*ordinazione*$_G$ dalla *prenotazione*$_G$ appropriata.

partialBill :

- **Endpoint**: /orders/partialBill
- **Metodo**: POST
- **Parametri**: PartialBillDTO
- **Descrizione**: questa funzione ritorna quanto un utente deve pagare col metodo "paga la tua parte".

romanBill :

- **Endpoint**: /orders/romanBill
- **Metodo**: POST
- **Parametri**: RomanBillDTO
- **Descrizione**: questa funzione ritorna quanto un utente deve pagare col metodo "pagamento alla romana".

totalBill :

- **Endpoint**: /orders/totalBill
- **Metodo**: POST
- **Parametri**: reservation_id: number
- **Descrizione**: questa funzione ritorna il totale da pagare di una *prenotazione*$_G$ con id associato.

checkOrdersPayStatus :

- **Endpoint**: /orders/checkOrdersPayStatus
- **Metodo**: POST
- **Parametri**: token: string, reservation_id
- **Descrizione**: questa funzione ritorna true se tutte le ordinazioni di un utente associate ad una *prenotazione*$_G$ sono state pagate, altrimenti ritorna *false*.

updateListOrders :

- **Endpoint**: /orders/updateListOrders
- **Metodo**: POST
- **Parametri**: UpdateListOrders
- **Descrizione**: questa funzione salva nel database la lista degli ingredienti delle ordinazioni di una *prenotazione*$_G$, confermando le ordinazioni.

4.1.1.7 Reservation

create :

- **Endpoint**: /reservation
- **Metodo**: POST
- **Parametri**: CreateReservationDto
- **Descrizione**: la funzione, attraverso i dati in input deve creare una nuova *prenotazione*$_G$, verificando che il ristorante non sia già pieno nel giorno della *prenotazione*$_G$, successivamente salvando nel database la nuova *prenotazione*$_G$ se avvenuta con successo.

addCustomer :

- **Endpoint**: /reservation/addCustomer
- **Metodo**: POST
- **Parametri**: AddCustomerDto
- **Descrizione**: questa funzione aggiunge, dopo la verifica del token, un utente ad una *prenotazione*$_G$ già creata in precedenza se ha ancora posti disponibili.

findAll :

- **Endpoint**: /reservation/
- **Metodo**: GET
- **Descrizione**: questa funzione restituisce la lista di tutte le prenotazioni

findOne :

- **Endpoint**: /reservation/:id
- **Metodo**: GET
- **Parametri**: id: number
- **Descrizione**: questa funzione ritorna la *prenotazione*$_G$ con l'id desiderato, oppure ritorna not found se non trovato.

getMenuWithOrdersQuantityByReservationId :

- **Endpoint**: /reservation/:id/orders
- **Metodo**: GET
- **Parametri**: id: number
- **Descrizione**: questa funzione ritorna il menu con la quantità degli ordini già effettuati della *prenotazione*$_G$ con id specificato.

getReservationsByRestaurantId :

- **Endpoint**: /reservation/restaurant/:restaurantId
- **Metodo**: GET
- **Parametri**: id: number
- **Descrizione**: questa funzione ritorna tutte le prenotazioni associate all'id del ristorante

getReservationsByUserId :

- **Endpoint**: /reservation/user/:userId
- **Metodo**: GET
- **Parametri**: id: number
- **Descrizione**: questa funzione ritorna tutte le *prenotazione*$_G$ associate all'user id.

verifyReservation :

- **Endpoint**: /reservation/verify
- **Metodo**: POST
- **Parametri**: verifyReservationDto
- **Descrizione**: questa funzione controlla, attraverso il token passato in input, se lo user appartiene ad una *prenotazione*$_G$.

getReservationByAdminId :

- **Endpoint**: /reservation/admin
- **Metodo**: POST
- **Parametri**: ReservationAdminDTO
- **Descrizione**: questa funzione ritorna tutte le prenotazioni del ristorante associato all'admin (amministratore del ristorante).

getUserOfReservation :

- **Endpoint**: /reservation/:id/user_token
- **Metodo**: POST
- **Parametri**: id: number, token: string
- **Descrizione**: questa funzione ritorna la *prenotazione*$_G$ con l'id desiderato se anche l'utente associato al token ne fa parte.

acceptReservation :

- **Endpoint**: /reservation/:id/accept
- **Metodo**: POST
- **Parametri**: id: number
- **Descrizione**: questa funzione salva nel database lo stato "accepted" nella *prenotazione*$_G$ con id appropriato.

rejectReservation :

- **Endpoint**: /reservation/:id/reject
- **Metodo**: POST
- **Parametri**: id: number
- **Descrizione**: questa funzione salva nel database lo stato "rejected" nella *prenotazione*$_G$ con id appropriato.

completeReservation :

- **Endpoint**: /reservation/:id/complete
- **Metodo**: POST
- **Parametri**: id: number
- **Descrizione**: questa funzione salva nel database lo stato "compleated" nella *prenotazione*$_G$ con id appropriato.

setPaymentMethod :

- **Endpoint**: /reservation/setPaymentMethod
- **Metodo**: POST
- **Parametri**: token: string, reservation_id: number, isRomanBill: boolean
- **Descrizione**: questa funzione salva nel database il metodo di pagamento della *prenotazione*$_G$ associata a reservation_id, il token viene utilizzato per verificare l'utente.

4.1.1.8 Restaurant

getFilteredRestaurants :

- **Endpoint**: /restaurant/filter
- **Metodo**: GET
- **Parametri**: query, currentPage: number, items_per_pege: number,
- **Descrizione**: questa funzione ritorna la lista dei ristoranti filtrati per nome, data, città e tipologia cucina

create :

- **Endpoint**: /restaurant
- **Metodo**: POST
- **Parametri**: RestaurantDTO
- **Descrizione**: questa funzione salva nel database un nuovo ristorante con i parametri in input.

findAll :

- **Endpoint**: /restaurant
- **Metodo**: GET
- **Descrizione**: questa funzione ritorna la lista di tutti i ristoranti presenti nel database.

findAllCuisines :

- **Endpoint**: /restaurant/cuisines
- **Metodo**: GET
- **Descrizione**: questa funzione ritorna tutte le tipologie di cucine presenti nel database.

findAllCities :

- **Endpoint**: /restaurant/cities
- **Metodo**: GET
- **Descrizione**: questa funzione ritorna la lista di tutte le città dei ristoranti prensenti nel database.

getNumberOfFilteredRestaurants :

- **Endpoint**: /restaurant/count
- **Metodo**: GET
- **Parametri**: query
- **Descrizione**: questa funzione ritorna il numero dei ristoranti disponibili con i filtri passati in input, ovvero: città, nome, data e tipologia cucina.

findOne :

- **Endpoint**: /restaurant/:id
- **Metodo**: GET
- **Parametri**: id: number
- **Descrizione**: questa funzione ritorna il ristorante con l'id associato.

getBookedTables :

- **Endpoint**: /restaurant/:id/booked-tables
- **Metodo**: GET
- **Parametri**: id: number, date: string
- **Descrizione**: questa funzione ritorna il numero di prenotazioni di un ristorante in un determinato giorno.

getRestaurantAndMenuByRestaurantId :

- **Endpoint**: /restaurant/filter
- **Metodo**: GET
- **Parametri**: id: number
- **Descrizione**: questa funzione ritorna il nuovo membro dello staff associato all'id con le informazioni del menu.

4.1.1.9 Staff

create :

- **Endpoint**: /staff
- **Metodo**: POST
- **Parametri**: StaffDto
- **Descrizione**: questa funzione salva nel database un nuovo membro dello staff con i parametri in input.

getRestaurantIdByAdminId :

- **Endpoint**: /staff/:id/restaurant
- **Metodo**: GET
- **Parametri**: restaurant_id: number
- **Descrizione**: questa funzione ritorna il ristorante associato all'admin (ovvero l'amministratore del ristorante).

4.1.1.10 User

create :

- **Endpoint**: /user/user
- **Metodo**: POST
- **Parametri**: UserDto
- **Descrizione**: questa funzione, attraverso i dati di input , crea un nuovo utente verificando che l'email dell'utente non sia già presente nel database.

createAdmin :

- **Endpoint**: /user/admin
- **Metodo**: POST
- **Parametri**: AdminDto
- **Descrizione**: questa funzione, attraverso i dati di input , crea un nuovo utente admin verificando che l'utente e il ristorante non siano già presenti nel database.

findOne :

- **Endpoint**: /user/:id
- **Metodo**: GET
- **Parametri**: id: number
- **Descrizione**: questa funzione ritorna l'utente con l'id desiderato, oppure ritorna not found se non trovato.

4.1.2 Frontend

Per il *frontend*$_G$, è stato utilizzato *Next.js*$_G$, un *framework*$_G$ basato su *React*$_G$. Questa scelta ha portato all'adozione della *React*$_G$ Architecture, che si basa sulla suddivisione della logica in vari componenti. Ogni componente gestisce una parte specifica della logica dell'applicazione, permettendo di creare pagine web utilizzando questi componenti. Inoltre, *Next.js*$_G$ supporta la *Server-Side Rendering* (*SSR*$_G$), che consente al server del *frontend*$_G$ di renderizzare le pagine prima di restituirle al client. Questo migliora le prestazioni iniziali del caricamento delle pagine e ottimizza l'indicizzazione sui motori di ricerca. La struttura delle cartelle del progetto *frontend*$_G$ è organizzata come segue:

```
src
 |- actions
 |- app
 |- components
```

Dove:

- **src**: È la cartella root del progetto *Next.js*$_G$.
- **actions**: Contiene tutte le Server Actions, che sono responsabili della gestione delle operazioni lato server, come il fetching dei dati e altre logiche *server-side*$_G$.
- **app**: Contiene le pagine renderizzate lato server (*SSR*$_G$), che vengono generate dal server e inviate al client.
- **components**: Contiene i componenti lato client, che gestiscono la logica e l'interfaccia utente sul lato client.

Questa *architettura*$_G$ offre diversi vantaggi:

- **Riutilizzabilità**: I componenti, essendo progettati per gestire logiche specifiche, possono essere facilmente riutilizzati in diverse parti dell'applicazione. Questo riduce la duplicazione del codice e facilita la *manutenzione*$_G$.
- **Manutenibilità**: La suddivisione in componenti rende il codice più organizzato e modulare. Ogni componente può essere sviluppato, testato e mantenuto in modo indipendente.
- **Composizione**: Le pagine vengono costruite componendo diversi componenti, rendendo semplice la costruzione di interfacce complesse a partire da parti più semplici e gestibili.

Per illustrare meglio l'*architettura*$_G$ adottata, vengono riportati i diagrammi delle classi per le pagine principali. Questi diagrammi mostrano come i componenti sono strutturati e come interagiscono tra loro per formare l'interfaccia utente. In sintesi, l'uso di *Next.js*$_G$ e della *React*$_G$ Architecture ha permesso di creare un *frontend*$_G$ modulare, riutilizzabile e facilmente mantenibile, migliorando l'efficienza dello sviluppo e la qualità del codice.

4.1.2.1 Admin

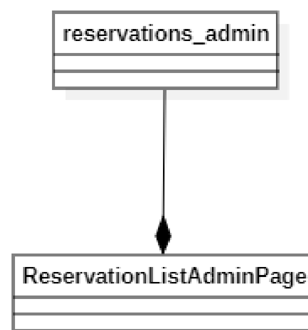

Figura 2: Diagramma UML pagina della lista prenotazione - Admin

Descrizione La pagina della lista prenotazioni per l'admin riporta tutte le prenotazioni che sono state fatte per il suo ristorante. Di ogni *prenotazione*$_G$ nella lista si visualizzano: id, numero di partecipanti, data, ora, un bottone per accedere alla singola *prenotazione*$_G$ e lo stato attuale in cui si trova la *prenotazione*$_G$ (in attesa di conferma, da pagare, accettata, rifiutata, e completata).

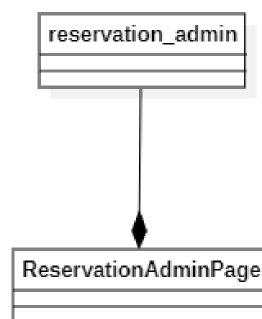

Figura 3: Diagramma UML pagina della singola prenotazione - Admin

Descrizione La pagina della singola *prenotazione*$_G$ permette all'admin di visualizzare i dati della *prenotazione*$_G$ con la possibilità di:

- accettare o rifiutare una nuova richiesta di *prenotazione*$_G$;
- visualizzare i pasti ordinati se la *prenotazione*$_G$ è stata accettata o completata;
- visualizzare il costo totale e/o parziale se lo ordinazioni sono state confermate.

4.1.2.2 Create_reservation

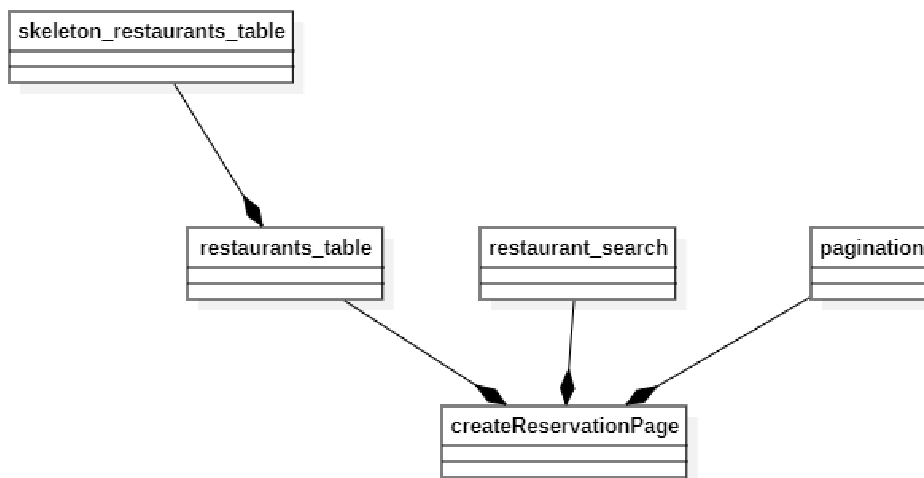

Figura 4: Diagramma UML pagina di prenotazione

Descrizione La pagina della *prenotazione*$_G$ permette all'utente di cercare un ristorante dove vuole effettuare una *prenotazione*$_G$, basandosi su nome, città, cucina e data. La ricerca tramite questi filtri permette di visualizzare i ristoranti che li soddisfano per poi procedere con la *prenotazione*$_G$ vera e propria.

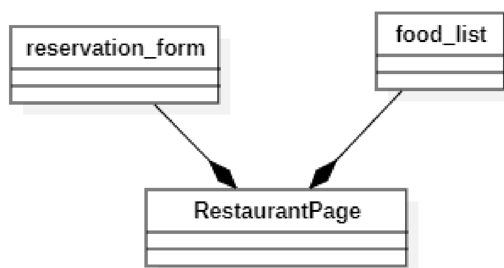

Figura 5: Diagramma UML pagina del ristorante

Descrizione La pagina del ristorante, dopo che è stato scelto il ristorante dalla lista di quelli presenti, permette di visualizzare diversi dati del ristorante scelto, tra cui indirizzo, città, cucina e il menù. Sempre nella stessa pagina vi è un form per la finalizzazione della *prenotazione*$_G$, in cui sono richiesti i dati: data, orario di arrivo e numero di partecipanti. Inseriti questi dati si può richiedere di effettuare una *prenotazione*$_G$, dopodiché l'utente verrà reindirizzata alla pagina reservation user.

4.1.2.3 Login

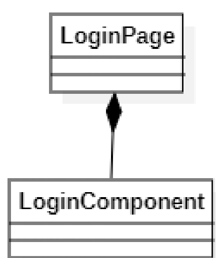

Figura 6: Diagramma UML Login

Descrizione La pagina login permette all'utente di inserire la propria email e password e accedere al sito se le credenziali corrispondono ad un account. Se il login ha successo l'utente sarà reindirizzato alla pagina create reservation.

4.1.2.4 Notifications

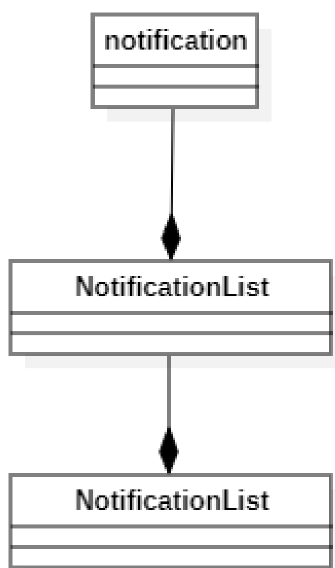

Figura 7: Diagramma UML pagina notifiche

Descrizione La pagina della lista delle notifiche permette di visualizzare la lista di tutte le notifiche ricevute in merito allo stato di una *prenotazione*$_G$ incluso il pagamento. La pagina è utilizzata sia per l'amministratore del ristorante che per l'utente base.

4.1.2.5 Order

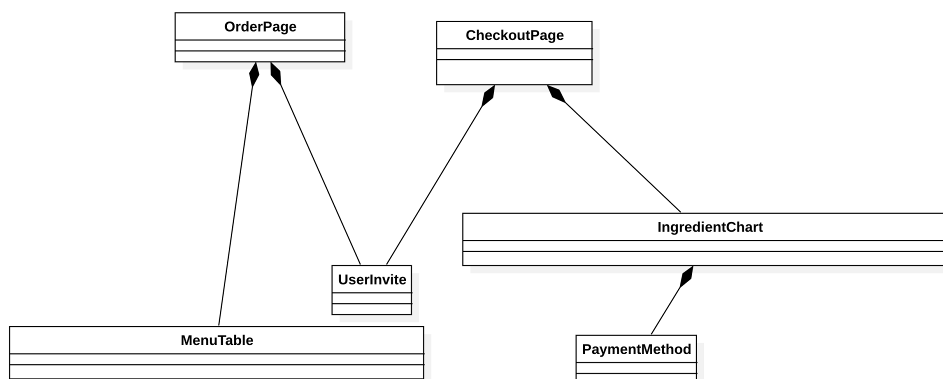

Figura 8: Diagramma UML pagina di ordinazione

OrderPage

Descrizione Durante la fase di *ordinazione*$_G$ collaborativa, si possono vedere le informazioni del ristorante e relativa *prenotazione*$_G$; questo è reso possibile dal componente OrderPage. Il componente MenuTable mostra una lista di piatti, ognuno dei quali ha a disposizione un contatore e due pulsanti per incrementare o decrementarne la quantità ordinata.

CheckoutPage

Descrizione Una volta finito il *processo*$_G$ di *ordinazione*$_G$, l'utente può accedere alla pagina di checkout, la quale contiene un riepilogo degli ordini e la possibilità di togliere degli ingredienti dai piatti ordinati; alla fine della pagina è inoltre possibile scegliere come suddividere il totale del conto tra gli utenti coinvolti nella *prenotazione*$_G$.

4.1.2.6 Sign_up

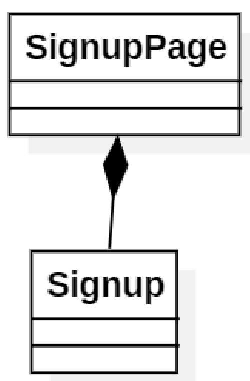

Figura 9: Diagramma UML pagina di sign up

Descrizione La pagina sign up permette all'utente di registrarsi come utente base all'interno del sito, inserendo una email non presente nel *sistema*$_G$, un nome, un cognome e la password. Se la registrazione avrà successo l'utente sarà reindirizzato al login.

4.1.2.7 Sign_up_admin

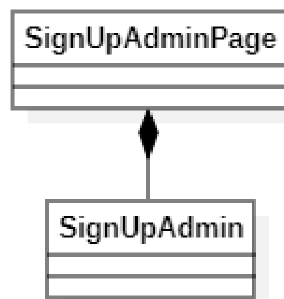

Figura 10: Diagramma UML pagina di sign up admin

Descrizione La pagina sign up admin permette all'utente di registrarsi come utente amministratore all'interno del sito, inserendo una email non presente nel *sistema*$_G$, un nome, un cognome, un nome per il ristorante, una città, un indirizzo, una descrizione, degli orari di apertura e di chiusura, un numero di coperti, un numero di telefono, una email per il ristorante, una tipologia di cucina, una password e la conferma della stessa. Se la registrazione avrà successo l'utente verrà reindirizzato al login.

4.1.2.8 User

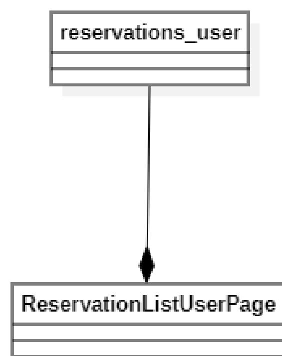

Figura 11: Diagramma UML pagina lista prenotazioni - utente base

Descrizione La pagina della lista prenotazioni per l'utente riporta tutte le prenotazioni che sono state fatte nei vari ristoranti. Di ogni *prenotazione*$_G$ nella lista si visualizzano: id, numero, di partecipanti, data, ora, un bottone per accedere alla singola *prenotazione*$_G$ e lo stato attuale in cui si trova la *prenotazione*$_G$ (in attesa di conferma, da pagare, accettata, rifiutata e completata).

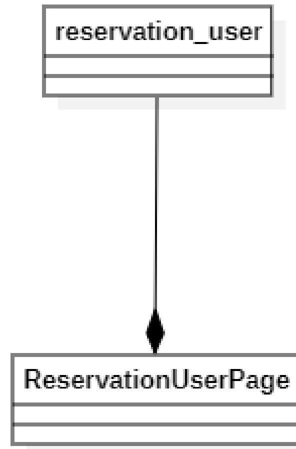

Figura 12: Diagramma UML pagina prenotazione specifica - utente base

Descrizione La pagina della singola *prenotazione*$_G$ permette all'utente di visualizzare i dati della *prenotazione*$_G$ con la possibilità di:

- vedere se la *prenotazione*$_G$ è stata accettata o rifiutata
- visualizzare i pasti ordinati se la *prenotazione*$_G$ è stata accettata o completata
- visualizzare il costo totale e/o parziale se lo ordinazioni sono state confermate

4.1.3 Websocket

Il *WebSocket*$_G$ server, implementato con NestJS, è progettato per gestire la comunicazione in tempo reale tra il server e i client. Questa *architettura*$_G$ permette di instaurare connessioni persistenti e bidirezionali, fornendo aggiornamenti immediati senza la necessità di interrogare continuamente il server.

4.1.3.1 Componenti Principali

L'*architettura*$_G$ logica del *WebSocket*$_G$ server si compone dei seguenti elementi principali:

- **Gateway *WebSocket*$_G$**: La porta di ingresso principale per le connessioni *WebSocket*$_G$. Questo componente gestisce la creazione, chiusura e gestione delle connessioni.
- **Eventi e Canali**: Gestione degli eventi e dei canali attraverso cui i messaggi vengono inviati e ricevuti. Ogni canale può essere visto come un argomento specifico a cui i client possono iscriversi.
- **Sottoscrizioni**: I client possono iscriversi a determinati eventi o canali, ricevendo notifiche ogni volta che un messaggio pertinente viene inviato.
- **Gestione delle Connessioni**: Mantenimento dello stato delle connessioni, inclusa la gestione di eventi di connessione e disconnessione, e l'associazione dei client a specifici canali.
- **Servizi e Controller**: Logica di business che elabora i messaggi in arrivo e genera risposte o eventi di notifica.

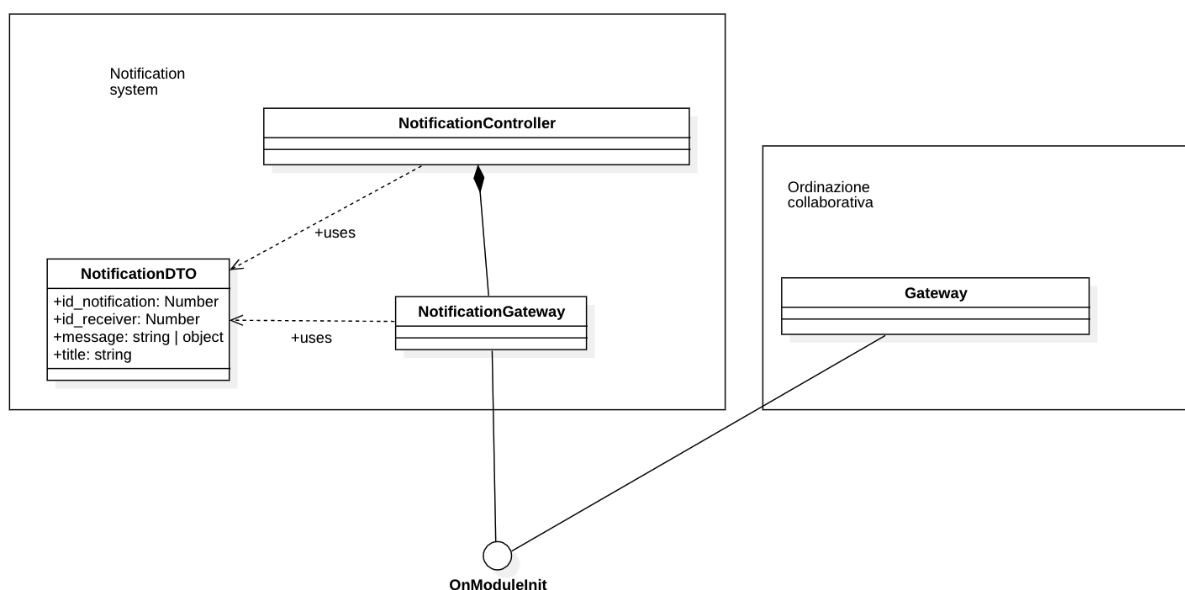

Figura 13: Diagramma UML server socket

4.1.4 Database

Il progetto utilizza PostgreSQL come *sistema*$_G$ di gestione del database relazionale (*RDBMS*$_G$). PostgreSQL è una scelta robusta e scalabile che offre numerose funzionalità avanzate, tra cui:

- **ACID Compliance**: Garantisce l'integrità dei dati attraverso le proprietà di Atomicità, Consistenza, Isolamento e Durabilità.
- **Supporto per Transazioni**: Consente l'esecuzione sicura e affidabile delle transazioni, fondamentale per applicazioni che richiedono coerenza nei dati.

Nel contesto del nostro progetto, PostgreSQL è utilizzato per memorizzare e gestire tutti i dati dell'applicazione, inclusi gli utenti, e le transazioni in tempo reale. La configurazione del database è gestita tramite un container *Docker*$_G$ dedicato, garantendo un ambiente di sviluppo e produzione coerente e facilmente replicabile.

Schema base di dati Di seguito viene riportato lo schema ER del database utilizzato

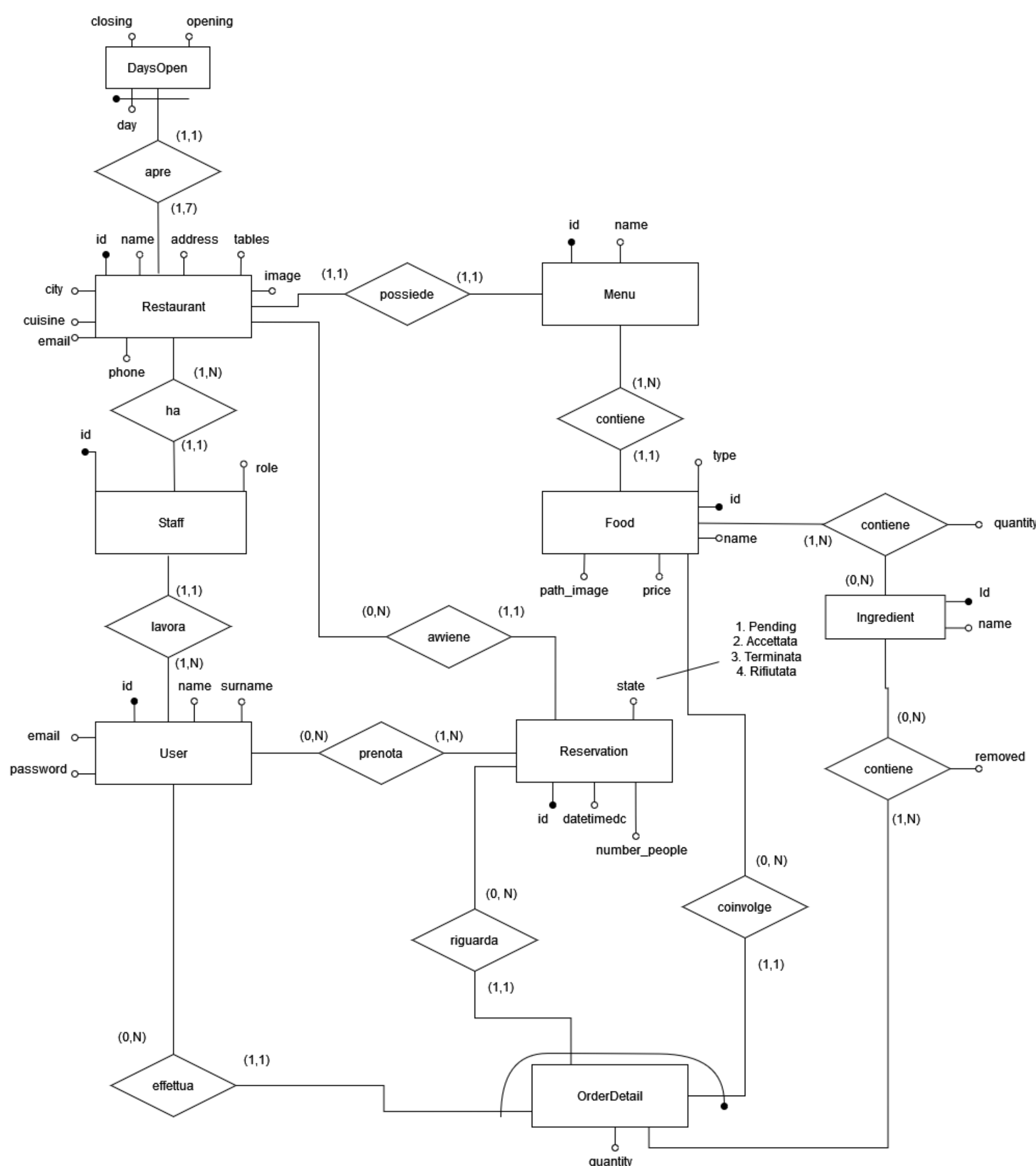

Figura 14: Schema ER

4.2 Design Pattern utilizzati

4.2.1 Backend

Nel *backend*$_G$ si sono utilizzati i pattern messi a disposizione dal *framework*$_G$ *Nest.js*$_G$.

4.2.1.1 Dependency Injection

Il pattern Dependency Injection ha l'obiettivo di minimizzare le dipendenze tra le classi, riducendone il grado di accoppiamento. Questo approccio migliora la modularità e la testabilità del codice. In NestJS, il pattern Dependency Injection è implementato utilizzando il decorator @Injectable(). Grazie a questo decorator, è possibile dichiarare le classi le cui dipendenze sono gestite e iniettate dal container di iniezione delle dipendenze di NestJS.

```
1 @Injectable()
2 export class AuthenticationService {
3   constructor(
4     private jwtService: JwtService,
5     private userService: UserService
6   ) { }
7 }
```

Listing 1: Esempio di utilizzo della Dependency Injection con il decorator @Injectable in un servizio di autenticazione

Il codice mostra l'implementazione del servizio di autenticazione. Il servizio AuthenticationService è decorato con @Injectable(), il che indica che può avere dipendenze iniettate. Il costruttore del servizio accetta due dipendenze:

- jwtService: un servizio fornito da NestJS per gestire la creazione e la verifica dei *JSON*$_G$ Web Token (*JWT*$_G$) utilizzati per l'autenticazione.
- userService: il servizio che gestisce la logica degli user. Questo servizio viene utilizzato per recuperare informazioni sull'utente necessarie per il *processo*$_G$ di autenticazione.

Questo esempio dimostra l'utilizzo del pattern Dependency Injection in NestJS, che semplifica la gestione delle dipendenze tra i vari componenti dell'applicazione.

```
1 @Controller('authentication')
2 export class AuthenticationController {
3   constructor(private readonly authenticationService: AuthenticationService) {
    }
4 }
```

Listing 2: Esempio di utilizzo della Dependency Injection per gestire il controller e il service

In questo esempio, viene definito un controller AuthenticationController che gestisce le richieste relative all'autenticazione dell'applicazione. La Dependency Injection viene utilizzata per iniettare il servizio AuthenticationService all'interno del costruttore del controller.
Questo approccio permette al controller di accedere direttamente al servizio AuthenticationService, senza la necessità di creare istanze o gestire manualmente le dipendenze. Ciò semplifica il codice e promuove una migliore organizzazione e manutenibilità dell'applicazione.

I vantaggi della Dependency Injection sono i seguenti:

- **Riduzione dell'Accoppiamento**: La DI riduce il grado di accoppiamento tra le classi, facilitando la sostituzione e la modifica dei componenti senza influenzare il resto del *sistema*$_G$.
- **Migliore Testabilità**: Le dipendenze possono essere facilmente simulate o sostituite durante i *test*$_G$, rendendo il codice più testabile.
- **Modularità**: La DI promuove una progettazione modulare del codice, dove i componenti sono autonomi e possono essere sviluppati, testati e mantenuti indipendentemente.
- **Manutenibilità**: Le applicazioni costruite con DI sono generalmente più facili da manutenere e scalare, poiché le dipendenze sono dichiarate e gestite dal *framework*$_G$.
- **Inversione del Controllo**: Con la DI, il controllo della creazione degli oggetti è spostato dal codice applicativo al *framework*$_G$, migliorando la gestione delle dipendenze.

4.2.1.2 Decorator pattern

In NestJS, il pattern dei decorator viene utilizzato per definire metadati e configurare le componenti dell'applicazione. Specificando i decorator prima di definire una classe, un metodo o una proprietà, è possibile aggiungere funzionalità al codice in modo dichiarativo. Di seguito è riportata una lista dei principali decorator utilizzati nel progetto, con una spiegazione dettagliata di ciascuno:

- **Module**: Il decorator `@Module` viene utilizzato per definire una classe come modulo. Un modulo in NestJS è una struttura che aggrega controller, provider e dipendenze da altri moduli. Serve a organizzare il codice in unità riutilizzabili e gestibili. Esempio:

```
@Module({
  controllers: [AuthenticationController],
  providers: [AuthenticationService],
  exports: [AuthenticationService],
  imports: [ JwtModule.register(), UserModule ]
})
export class AuthenticationModule {}
```

- **Controller**: Il decorator `@Controller` dichiara una classe come controller, il che significa che gestirà le richieste *HTTP*$_G$. I metodi all'interno della classe contrassegnata come `@Controller` possono essere associati a specifiche rotte *HTTP*$_G$. Esempio:

```
@Controller('authentication')
export class AuthenticationController { ... }
```

In questo caso viene passato al controller l'argomento `authentication`, in questo modo NestJS si occuperà di creare un *endpoint*$_G$ raggiungibile tramite il suffisso specificato.

- **Gestione delle richieste *HTTP*$_G$**: NestJS mette a disposizione diversi decorator per gestire parametri della rotta, corpo della richiesta, e per associare una rotta a una funzione. Alcuni esempi includono `@Get`, `@Post`, `@Body`, `@Param`, `@Query`, ecc. Questi decorator aiutano a mappare gli *endpoint*$_G$ *HTTP*$_G$ ai metodi del controller in modo chiaro e leggibile. Esempio:

```

@Controller('authentication')
export class AuthenticationController {
  constructor(private readonly authenticationService:
   AuthenticationService) { }

  @Post('signin')
  @HttpCode(200)
  async signin(@Body() dto: AuthenticationDto) { ... }

  @Post('decodeToken')
  @HttpCode(200)
  async decodeToken(@Body() body: DecodeTokenDTO) { ... }
}
```

- **Provider**: Una classe marcata come `@Injectable` può essere iniettata tramite il *sistema*$_G$ di Dependency Injection di NestJS. Questo significa che può essere utilizzata come dipendenza in altre classi. Esempio:

```
@Injectable()
export class AuthenticationService {
    \\logica del servizio
}
```

- **Validation**: NestJS permette di utilizzare dei decorator per la validazione dei dati in modo dichiarativo, tramite la libreria `class-validator`. Ad esempio, utilizzando `@IsString`, `@IsNumber`, ecc., sui campi di una classe per validare i dati in ingresso. Esempio:

```
export class AddCustomerDTO {
    @IsJWT()
    token: string;

    @IsNotEmpty()
    @IsNumber()
    reservation_id: number;
}
```

Quindi i vantaggi sono:

- **Leggibilità**: I decorator permettono di aggiungere metadati e comportamenti direttamente sulle classi e sui metodi, rendendo il codice più chiaro e immediato.
- **Modularità**: Consentono di suddividere l'applicazione in moduli ben definiti, ciascuno con controller, provider e dipendenze, migliorando l'organizzazione del codice.
- **Manutenibilità**: Grazie alla Dependency Injection gestita tramite decorator come `@Injectable`, il codice è più facilmente testabile e le dipendenze sono meglio gestite.
- **Validazione dei dati**: I decorator di validazione permettono di definire regole direttamente sui DTO, semplificando la validazione dei dati in ingresso e riducendo il codice boilerplate.

4.2.1.3 Repository pattern

Il *Repository Pattern* è un pattern di progettazione che facilita la separazione tra la *logica di business*$_G$ e l'accesso ai dati. In NestJS, l'implementazione del *Repository Pattern* permette di gestire in modo efficace le operazioni di persistenza dei dati, fornendo un'interfaccia comune per l'interazione con il database.

Il *Repository*$_G$ agisce come un'interfaccia tra la *logica di business*$_G$ e il *layer di persistenza*$_G$. Incapsula la logica necessaria per accedere ai dati e fornisce metodi per eseguire operazioni di *CRUD*$_G$ (Create, Read, Update, Delete). In questo modo, il *Repository*$_G$ funge da *Facade*, semplificando l'accesso ai dati e nascondendo i dettagli complessi dell'interazione con il database.

```
@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User)
    private userRepo: Repository<User>,
    ...
  ) { }
  // logica del service
}
```

Listing 3: Esempio di utilizzo del *Repository*$_G$ Pattern in un Service di NestJS

L'utilizzo del *Repository Pattern* offre i seguenti vantaggi:

- **Isolamento della logica di accesso ai dati**: Separando la logica di accesso ai dati dalla *logica di business*$_G$, il codice diventa più organizzato e mantenibile. Le modifiche al *layer di persistenza*$_G$ non influenzano direttamente la *logica di business*$_G$.
- **Testabilità**: I *Repository*$_G$ possono essere facilmente mockati durante i *test*$_G$, permettendo di isolare la *logica di business*$_G$ e di verificare il comportamento del codice senza dipendere da un database reale.
- **Astrazione del database**: Il *Repository*$_G$ nasconde i dettagli specifici del database e fornisce un'interfaccia unificata per l'accesso ai dati. Ciò semplifica il cambio del database sottostante senza dover modificare il codice dei Service che utilizzano il *Repository*$_G$.

In sintesi, il *Repository*$_G$ *Pattern* in NestJS permette di ottenere un'*architettura*$_G$ pulita e modulare, migliorando l'organizzazione del codice e facilitando la testabilità e la manutenibilità dell'applicazione. Agendo come *Facade*, il *Repository*$_G$ semplifica l'interazione con il *layer di persistenza*$_G$, nascondendo i dettagli complessi e fornendo un'interfaccia uniforme per l'accesso ai dati.

4.2.1.4 Controller-Service pattern

In NestJS, un *framework*$_G$ per applicazioni *Node.js*$_G$, si utilizza una suddivisione chiara tra la gestione delle richieste *HTTP*$_G$ e la *logica di business*$_G$. Questo approccio si basa sull'uso di due classi principali:

- **Controller**: Il Controller è responsabile della gestione delle richieste *HTTP*$_G$. Il suo compito principale è di associare le richieste ai metodi appropriati e di fungere da intermediario tra il client che effettua la richiesta e la *logica di business*$_G$.
- **Service**: Il Service contiene la *logica di business*$_G$ dell'applicazione ed è responsabile dell'interazione con il *layer di persistenza*$_G$ dei dati. I metodi definiti in un Service possono essere chiamati sia dai Controller sia da altri Service, facilitando così la separazione delle responsabilità.

Questo pattern di separazione offre diversi vantaggi:

- **Riusabilità** Separando la *logica di business*$_G$ dalla gestione delle richieste *HTTP*$_G$, i metodi definiti nei Service possono essere riutilizzati in diversi contesti. Ad esempio, un Service può essere utilizzato da più Controller o da altri Service, promuovendo una maggiore modularità e riusabilità del codice.
- **Testabilità**: La chiara divisione delle responsabilità facilita la scrittura di unit *test*$_G$. Poiché la *logica di business*$_G$ è isolata nei Service, è possibile *test*$_G$are questi in modo indipendente dai Controller, assicurando che ogni componente funzioni correttamente.
- **Manutenibilità**: La separazione tra Controller e Service rende il codice più mantenibile. Eventuali modifiche alla *logica di business*$_G$ nei Service non richiedono modifiche ai Controller, riducendo il rischio di introdurre errori e semplificando il *processo*$_G$ di aggiornamento e miglioramento dell'applicazione.

In sintesi, l'uso di Controller e Service in NestJS permette di mantenere il codice organizzato, modulare e facile da testare e mantenere, migliorando la qualità e la robustezza dell'applicazione.

4.2.1.5 Module pattern

Il *backend*$_G$ è organizzato in moduli, ciascuno dei quali può contenere controller, servizi, provider e la connessione al database. Di seguito è riportato un esempio che mostra come specificare un controller, un service, la connessione al database e altri servizi provenienti da moduli esterni:

```
@Module({
  controllers: [UserController],
  providers: [UserService],
  imports: [
    TypeOrmModule.forFeature([User]),
    StaffModule,
    RestaurantModule,
    DaysopenModule
  ],
  exports: [TypeOrmModule, UserService]
})
export class UserModule { }
```

Listing 4: Esempio di modulo con controller, service, connessione al database e altri servizi da moduli esterni

L'uso dei moduli nel *backend*$_G$ offre diversi vantaggi:

- **Modularità**: consente di suddividere l'applicazione in componenti indipendenti, migliorando l'organizzazione del codice e la manutenibilità.
- **Riusabilità**: i moduli possono essere facilmente riutilizzati in diverse parti dell'applicazione o in altri progetti.
- **Iniezione delle dipendenze**: facilita la gestione delle dipendenze tra i vari componenti, rendendo il codice più testabile e riducendo l'accoppiamento tra le classi.
- **Scalabilità**: permette di scalare l'applicazione aggiungendo nuovi moduli senza influenzare il resto del *sistema*$_G$.
- **Isolamento**: ogni modulo può essere sviluppato e testato indipendentemente, riducendo i possibili conflitti e bug.

4.2.2 Frontend

Nel *frontend*$_G$ si sono utilizzati i pattern messi a disposizione dal *framework*$_G$ *Next.js*$_G$.

4.2.2.1 Backend for frontend

Il Backends for Frontends (BFF) è un *design*$_G$ pattern utilizzato per creare *API*$_G$ specifiche per ogni tipo di client. Questo pattern prevede la creazione di uno strato di *backend*$_G$ dedicato che interagisce con il *frontend*$_G$, fornendo un'*API*$_G$ su misura per le esigenze specifiche del client. Di seguito vengono riportate le principali caratteristiche del BFF:

- ***API*$_G$ Specifiche per il Client**: Ogni tipo di client ha un proprio BFF che fornisce un'*API*$_G$ ottimizzata per le sue esigenze. Questo consente di adattare l'*API*$_G$ alle peculiarità del client, migliorando l'efficienza e l'esperienza utente.
- **Isolamento delle Logiche di Presentazione**: Il BFF si occupa della logica di presentazione, trasformando e aggregando i dati provenienti dai vari servizi *backend*$_G$ per fornire al *frontend*$_G$ solo ciò di cui ha bisogno in un formato conveniente.
- **Riduzione della Complessità del Frontend**: Spostando alcune logiche di business e aggregazione dei dati nel BFF, il *frontend*$_G$ diventa più semplice e focalizzato esclusivamente sulla presentazione e interazione con l'utente.
- **Decoupling tra *Frontend*$_G$ e Backend**: Il BFF funge da intermediario tra il frontend e i servizi *backend*$_G$, consentendo una maggiore indipendenza e flessibilità nella gestione delle modifiche. Ad esempio, cambiamenti nei servizi *backend*$_G$ possono essere gestiti nel BFF senza necessitare modifiche nel frontend.
- **Ottimizzazione delle Performance**: Il BFF può gestire la cache, la compressione dei dati e altre ottimizzazioni specifiche per migliorare le performance del client.

Di conseguenza i vantaggi sono:

- **Migliore Esperienza Utente**: L'*API*$_G$ personalizzata consente al *frontend*$_G$ di ottenere esattamente i dati necessari in un formato ottimizzato, migliorando le performance e la reattività dell'applicazione.
- **Sviluppo e *Manutenzione*$_G$ Separati**: Ogni BFF può essere sviluppato e mantenuto da team separati, permettendo una maggiore specializzazione e autonomia dei team di sviluppo.
- **Adattabilità e Flessibilità**: È possibile modificare e ottimizzare l'*API*$_G$ del BFF senza influire sugli altri client o sul *backend*$_G$ principale, facilitando l'adozione di nuove funzionalità e miglioramenti.
- **Sicurezza**: Il BFF può implementare misure di sicurezza specifiche per ogni tipo di client, migliorando la protezione dei dati e la gestione delle autorizzazioni.

```
'use server'
...
export async function validateSignUp(prevState: any, formData: FormData) {

  // Check if email is valid
  const email = formData.get('email')?.toString()
  if (!email) {
   return { message: 'Email is required' }
  }
  if (!String(email).includes('@')) {
   return { message: 'Email must be valid' }
  }
    ...

    ...
  // Create the customer
  const response = await createUser({
    email: email, name: firstName, surname: lastName, password: password
  });

  if (!response.ok) {
    const data = await response.json();
    if (data.message && Array.isArray(data.message))
      return { message: data.message.join(', ') };
    if (data.message)
      return { message: data.message };
    return { message: 'Registration failed' };
  }
  redirect('login?signup=success')
}
```

Listing 5: Esempio di *Backend*$_G$ for Frontend

Il pattern *Backend*$_G$ for *Frontend*$_G$ è particolarmente utile in contesti in cui diversi tipi di client hanno esigenze specifiche e richiedono ottimizzazioni particolari. Aiuta a migliorare la modularità, la manutenibilità e l'efficienza dell'*architettura*$_G$ complessiva, permettendo al tempo stesso un'evoluzione indipendente delle interfacce utente e dei servizi backend.

4.2.2.2 Compound Components

Il pattern Compound Components è un $design_G$ pattern utilizzato per creare componenti flessibili e riutilizzabili. Questo pattern consente ai componenti di collaborare tra loro in modo coeso, offrendo agli sviluppatori un modo potente per creare interfacce utente complesse e dinamiche. Di seguito vengono riportate le principali caratteristiche del Compound Components pattern:

- **Composizione Naturale**: I Compound Components sono costituiti da componenti più piccoli che lavorano insieme. Ad esempio, un componente di un modulo può essere composto da un ¡Form¿ principale e vari sottocomponenti come ¡Input¿, ¡Label¿, ¡Button¿, ecc.
- **Interazione Tra Componenti**: I componenti figli comunicano con il componente genitore attraverso il contesto (Context API_G) o altre forme di gestione dello stato condiviso. Questo consente ai figli di sapere del loro stato e delle loro azioni in relazione agli altri figli.
- **Flessibilità e Riutilizzo**: I Compound Components offrono una grande flessibilità poiché i componenti figli possono essere combinati in modi diversi, fornendo allo sviluppatore il controllo sulla struttura e il comportamento dell'interfaccia utente.
- **API_G Pulita**: L'API_G di un Compound Component è progettata per essere pulita e intuitiva. Gli sviluppatori possono utilizzare i componenti figli come se fossero parte di un'unica entità, senza preoccuparsi dei dettagli interni della loro implementazione.

I vantaggi del Compound Components pattern sono:

- **Maggiore Modularità**: I componenti sono altamente modulari e possono essere riutilizzati in diverse parti dell'applicazione.
- **Manutenibilità**: La separazione delle logiche dei componenti rende il codice più facile da mantenere e testare, poiché ogni componente ha una responsabilità ben definita.
- **Flessibilità di Configurazione**: Gli sviluppatori possono configurare e comporre i componenti in modi diversi senza dover modificare il codice del componente principale.

```
"use client";
...

export default function Header({ isLogin, isAdmin }: { isLogin: boolean,
   isAdmin: boolean }) {
  return (
   <header className="bg-orange-500">
    <div className="mx-auto max-w-screen-xxl px-4 sm:px-6 lg:px-8">
     ...

          ...
    </div>
   </header>
  );
}
```

Listing 6: Esempio di Compound Component

```
...
import Header from '../components/header'
...

const inter = Inter({ subsets: ['latin'] })

export const metadata: Metadata = {
  title: 'Easy Meal',
}

export default async function RootLayout({
  children,
}: {
  children: $\textit{React}_G$.$\textit{React}_G$Node
}) {
  const token = cookies().get("session")?.value;
  let isAdmin = false;
  if(token != null)
    isAdmin = (await verifySession()).role === "admin";
  return (
    <html lang="en">
      <body className={inter.className}>
        <NotificationProvider token={token}>
          <Header isLogin={token != null} isAdmin={isAdmin}/>
          {children}
        </NotificationProvider>
      </body>
    </html>
  )
}
```

Listing 7: Esempio dell'uso del component

4.2.2.3 Controlled inputs

Il pattern dei Controlled inputs è una tecnica utilizzata nelle applicazioni *frontend*$_G$ per gestire gli input degli utenti in modo prevedibile e controllato. Questo pattern si basa sull'associazione diretta dello stato di un componente con il valore degli input, fornendo così un controllo totale sugli input e sulle loro modifiche. Di seguito vengono riportate le principali caratteristiche del Controlled inputs pattern:

- **Stato Gestito**: Gli elementi di input (come ¡input¿, ¡textarea¿, ¡select¿) vengono collegati allo stato del componente. Il valore dell'input è determinato dallo stato e ogni modifica all'input aggiorna lo stato.
- **Gestione degli Eventi**: Gli eventi di input (come onChange) vengono utilizzati per aggiornare lo stato del componente. Ogni volta che l'utente digita qualcosa nell'input, l'evento onChange viene chiamato e aggiorna lo stato con il nuovo valore.
- **Renders Sincronizzati**: Poiché il valore dell'input è sempre sincronizzato con lo stato del componente, ogni aggiornamento dello stato provoca un re-render del componente con il nuovo valore dell'input.

I vantaggi del Controlled inputs pattern sono:

- **Prevedibilità**: Il valore degli input è sempre determinato dallo stato del componente, rendendo il comportamento degli input prevedibile e facile da gestire.
- **Validazione Semplificata**: Poiché tutti i valori degli input sono memorizzati nello stato, è facile eseguire la validazione in tempo reale e fornire *feedback*$_G$ immediato agli utenti.
- **Sincronizzazione**: Il pattern garantisce che il valore visualizzato nell'input sia sempre aggiornato con l'ultimo valore dello stato, evitando discrepanze tra l'input e lo stato interno.

```
'use client';
import { useFormState } from 'react-dom'
import { validateSignUp } from '@/src/actions/validateSignUp'

const initialState = {
  message: '',
}

export default function Signup() {
  const [state, formAction] = useFormState(validateSignUp, initialState);

  return (
    <>
      ...
            ...
              <form action={formAction}>
                ...

                ...
              </form>
              {state?.message && <p className="mt-4 text-center text-red-500">{
   state?.message}</p>}
            </div>
          </div>
        </div>
      </div>
    </>
  )
}
```

Listing 8: Esempio dell'uso del Controlled inputs pattern

4.2.3 Websocket

4.2.3.1 Publish-Subscribe pattern

Il *Publish-Subscribe pattern* è un pattern comportamentale utilizzato per la comunicazione asincrona tra componenti di un *sistema*$_G$. In questo modello, i *publisher* inviano messaggi senza conoscere i destinatari, e i *subscriber* ricevono messaggi senza conoscere gli mittenti. Questa separazione è resa possibile da un intermediario che gestisce la comunicazione, nel nostro caso, un *WebSocket*$_G$.

```
import { ConnectedSocket, MessageBody, SubscribeMessage, WebSocketGateway,
   WebSocketServer } from "@nestjs/websockets";
import { Server, Socket } from 'socket.io'
import { Inject, OnModuleInit } from "@nestjs/common";

@WebSocketGateway({ cors: { origin: '*' } })
export class MyGateway implements OnModuleInit {
    @WebSocketServer()
    server: Server;

    async handleConnection(socket: Socket) {...}

    onModuleInit() {
    this.server.on('connection', this.handleConnection);
    }

    @SubscribeMessage('onMessage')
    async onIncrement(@MessageBody() body: any) {...}

    @SubscribeMessage('onIngredient')
    async onIngredient(@MessageBody() body: any) {...}

    @SubscribeMessage('onConfirm')
    async onConfirm(@MessageBody() body, @ConnectedSocket() client: Socket)
   {...}
}
```

Listing 9: Esempio di come viene definita la classe MyGateway per implementare il socket

Dettagli dell'implementazione:

- **Connessione del client:** Quando un client si connette, il metodo `handleConnection` viene chiamato per verificare il client e iscriverlo a una stanza specifica.
- **Iscrizione agli eventi:** I client, una volta connessi, possono iscriversi a specifici eventi tramite il *WebSocket*$_G$. Gli eventi gestiti in questo esempio sono `onMessage`, `onIngredient` e `onConfirm`. Quando questi eventi vengono inviati dal server, i client che si sono iscritti a tali eventi ricevono i relativi messaggi.
- **Gestione dei messaggi:** Il *WebSocket*$_G$ si occupa di distribuire i messaggi ai client iscritti e di mantenere le connessioni attive tra il server e i client.

Vantaggi dell'utilizzo del pattern Publish-Subscribe:

- **Scalabilità:**
  - L'implementazione non richiede un numero preciso di *subscriber*, supportando facilmente un numero variabile di client connessi. Ciò consente al *sistema*$_G$ di adattarsi facilmente a picchi di traffico senza compromettere le performance.
  - In termini di scalabilità orizzontale, è possibile aggiungere ulteriori server *publisher* senza dover modificare l'implementazione del *sistema*$_G$. Questo permette di distribuire il carico tra più server, migliorando la resilienza e la capacità di gestione delle richieste.
- **Centralizzazione:**
  - La presenza di un *WebSocket*$_G$ come intermediario centralizzato semplifica la gestione della comunicazione tra componenti diversi. Tutti i messaggi passano attraverso il *WebSocket*$_G$, rendendo più facile il monitoraggio, il debug e la gestione delle connessioni.
  - La centralizzazione facilita anche l'implementazione di funzionalità aggiuntive come la logica di distribuzione dei messaggi, la gestione delle riconnessioni e la sicurezza.
- **Asincronicità**:
  - Il modello asincrono consente al *publisher* di inviare messaggi senza aspettare una conferma dai *subscriber*. Questo significa che il *publisher* può continuare a funzionare anche se non ci sono client connessi, migliorando l'efficienza del *sistema*$_G$.
  - L'asincronicità riduce la latenza percepita dai client, in quanto i messaggi vengono ricevuti non appena inviati dal *publisher*, senza dover attendere la disponibilità dei destinatari.

In sintesi, il *Publisher-Subscriber pattern* implementato tramite *WebSocket*$_G$ offre un modo flessibile, scalabile e efficiente per gestire la comunicazione tra componenti distribuiti, rendendolo ideale per applicazioni in tempo reale e sistemi con un elevato numero di client.

4.3 Architettura di deployment

L'intero stack tecnologico e i layer del modello architetturale sono stati implementati ed eseguiti all'interno di un ambiente *Docker*$_G$ che simula la suddivisione e la distribuzione dei servizi. Le informazioni dettagliate sulle immagini utilizzate e sulle configurazioni dell'ambiente sono disponibili nel file compose.yaml presente nella cartella root del *repository*$_G$ EasyMeal del progetto. Per l'ambiente di produzione, sono stati creati i seguenti container:

- **postgres**: container contenente il database PostgreSQL
  - **Immagine**: postgres;
  - **Porta**: 7070;
  - **Dipendenza**: nessuna;
  - **Rete**: webnet.
- **backend**: container contenente Nest.js
  - **Build**: Dockerfile;
  - **Porta**: 6969;
  - **Dipendenza**: postgres;
  - **Rete**: webnet.
- **socket**: container contenente il *websocket*$_G$ server
  - **Build**: Dockerfile;
  - **Porta**: 8000;
  - **Dipendenza**: postgres;
  - **Rete**: webnet.
- **frontend**: container contenente Next.js
  - **Build**: Dockerfile;
  - **Porta**: 3000;
  - **Dipendenza**: *backend*$_G$;
  - **Rete**: webnet.

Nello stesso file compose.yaml si specifica i container si trovano all'interno della stessa rete webnet che utilizza il driver di rete bridge.

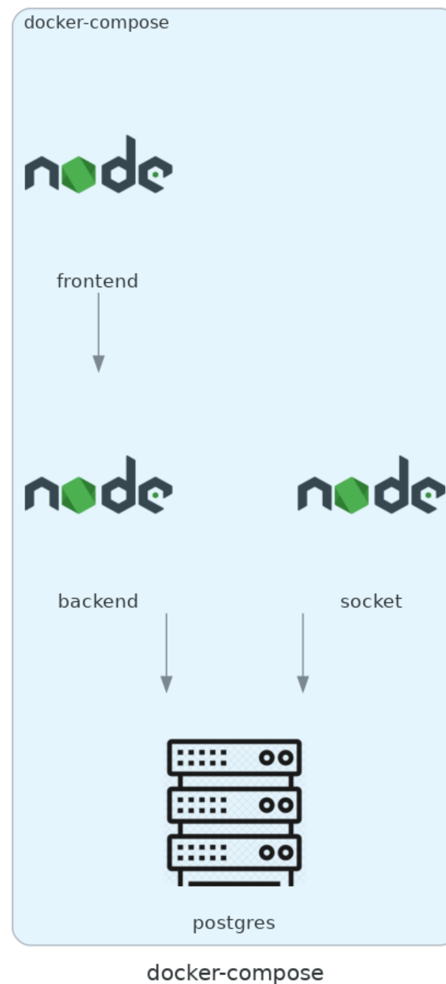

Figura 15: Archittettura di deployment del prodotto

4.3.1 Vantaggi

- **Isolamento e Consistenza**: Ogni componente è isolato in un container *Docker*$_G$, garantendo che le dipendenze e le configurazioni siano coerenti tra gli ambienti di sviluppo, *test*$_G$ e produzione.
- **Scalabilità**: I servizi possono essere scalati indipendentemente a seconda delle necessità, ad esempio aumentando il numero di istanze del *backend*$_G$ o del server *WebSocket*$_G$ per gestire carichi di lavoro maggiori.
- **Facilità di Manutenzione**: Con *Docker*$_G$ Compose è semplice gestire e aggiornare i servizi, grazie a una singola definizione centralizzata.
- **Portabilità**: I container *Docker*$_G$ possono essere eseguiti su qualsiasi piattaforma che supporti *Docker*$_G$, rendendo il *deployment*$_G$ e la migrazione tra diversi ambienti molto più agevoli.

Nonostante l'uso di container per ogni componente, l'*architettura*$_G$ complessiva è considerata monolitica perché tutte le parti del *sistema*$_G$ (frontend, *backend*$_G$ e *WebSocket*$_G$ server) sono gestite come un'unica unità in quanto tutte le componenti sono strettamente collegate e dipendono l'una dall'altra per funzionare correttamente.

5 Stato dei requisiti funzionali

Di seguito vengono riportati i requisiti funzionali individuati nel documento di Analisi dei Requisiti v 2.0.0 e il loro stato in relazione all'*architettura*$_G$ individuata.
Lo stato viene codificato con:

- S: soddisfatto;
- NS: non soddisfatto.

Codice	*Rilevanza*	*Descrizione*	*Stato*
ROF 1	Obbligatorio	L'utente deve poter selezionare un piatto dalla lista dei piatti disponibili.	S
RDF 2	Desiderabile	Il *sistema*$_G$ deve inviare una notifica se l'utente base sta aggiungendo un piatto che contiene elementi a cui è allergico/intollerante.	NS
ROF 3	Obbligatorio	L'utente deve visualizzare i pasti con i loro ingredienti e può modificarli, togliendo o modificandone la quantità.	S
ROF 4	Obbligatorio	L'utente base deve essere in grado di visualizzare il riepilogo di quanto ordinato e di confermare.	S
ROF 5	Obbligatorio	Il *sistema*$_G$ deve poter inviare una notifica di conferma dell'*ordinazione*$_G$ collaborativa all'amministratore del ristorante	S
ROF 6	Obbligatorio	L'utente deve poter inserire le informazioni (data, orario, persone) per effettuare una *prenotazione*$_G$.	S
ROF 7	Obbligatorio	Il *sistema*$_G$ deve poter inviare una notifica all'amministratore per comunicare la richiesta di *prenotazione*$_G$, che può accettare e rifiutare.	S
RDF 8	Desiderabile	L'utente deve poter essere in grado di cancellare la *prenotazione*$_G$.	NS
ROF 9	Obbligatorio	Il *sistema*$_G$ deve poter negare la *prenotazione*$_G$ se il ristorante non possiede abbastanza posti o tavoli.	S
ROF 10	Obbligatorio	L'utente base deve poter visualizzare una lista di ristoranti filtrata per nome oppure data, luogo, tipologia cucina.	S
ROF 11	Obbligatorio	L'utente base deve poter selezionare un ristorante.	S
ROF 12	Obbligatorio	L'amministratore deve essere in grado di poter visualizzare una lista di prenotazioni in attesa.	S
ROF 13	Obbligatorio	L'amministratore deve poter selezionare una specifica richiesta di *prenotazione*$_G$ dalla lista visualizzata.	S
ROF 14	Obbligatorio	L'amministratore deve poter accettare la richiesta di *prenotazione*$_G$ selezionata.	S
ROF 15	Obbligatorio	Il *sistema*$_G$, dopo l'accettazione, aggiunge nell'area "prenotazioni" dell'utente la *prenotazione*$_G$.	S
ROF 16	Obbligatorio	Il *sistema*$_G$ deve notificare gli utenti coinvolti (nel caso di *prenotazione*$_G$ collaborativa) l'accettazione della *prenotazione*$_G$.	S
RDF 17	Desiderabile	Il *sistema*$_G$ deve fornire un'interfaccia per consentire all'amministratore di verificare la disponibilità di posti in base alle specifiche della *prenotazione*$_G$ selezionata.	NS
RDF 18	Desiderabile	Il *sistema*$_G$ deve ridurre il numero di posti disponibili in base alle specifiche della *prenotazione*$_G$ dopo che questa è stata accettata.	NS
ROF 19	Obbligatorio	L'amministratore deve poter rifiutare la richiesta di *prenotazione*$_G$ selezionata.	S

Codice	*Rilevanza*	*Descrizione*	*Stato*
ROF 20	Obbligatorio	Il *sistema*$_G$ deve notificare gli utenti coinvolti (nel caso di *prenotazione*$_G$ collaborativa) del rifiuto della *prenotazione*$_G$.	S
RDF 21	Desiderabile	Il *sistema*$_G$ deve creare un canale di comunicazione tra l'utente e l'amministratore del ristorante quando l'utente lo richiede.	NS
RDF 22	Desiderabile	L'utente e l'amministratore devono poter scambiare messaggi in modo bidirezionale tramite l'interfaccia di comunicazione.	NS
RDF 23	Desiderabile	Durante la comunicazione, il *sistema*$_G$ deve inviare notifiche *push*$_G$ per informare l'utente e l'amministratore dei nuovi messaggi ricevuti.	NS
RDF 24	Desiderabile	La cancellazione della *prenotazione*$_G$ può essere effettuata con al massimo un giorno di anticipo rispetto alla data della *prenotazione*$_G$.	NS
ROF 25	Obbligatorio	Il *sistema*$_G$ deve permettere di pagare il conto in base alla modalità scelta (divisione equa, divisione proporzionale) da chi ha creato la *ordinazione*$_G$ collaborativa.	S
RDF 26	Desiderabile	L'utente base può pagare l'intero conto se nessun utente non ha ancora effettuato il pagamento.	NS
RDF 27	Desiderabile	L'amministratore deve poter essere in grado di modificare il menu' del proprio ristorante, aggiungendo, rimuovendo pietanze e modificando le informazioni del singolo piatto (nome, ingredienti e prezzo).	NS
RDF 28	Desiderabile	La modifica del menu' da parte dell'amministratore non deve causare problemi di sincronizzazione nella visualizzazione, ricerca del menu' e nell'*ordinazione*$_G$ da parte dell'utente base.	NS
RDF 29	Desiderabile	Si deve permettere all'utente base di poter inserire un coupon prima di pagare il conto, che, se applicato, deve far ricalcolare al *sistema*$_G$ il prezzo del conto.	NS
ROF 30	Obbligatorio	Il *sistema*$_G$ deve consentire all'amministratore di visualizzare la lista delle prenotazioni effettuate per il proprio ristorante.	S
ROF 31	Obbligatorio	L'amministratore deve poter visualizzare i dettagli di una specifica *prenotazione*$_G$.	S
RDF 32	Desiderabile	L'amministratore deve poter visualizzare lo stato delle ordinazioni associate alla *prenotazione*$_G$.	S
ROF 33	Obbligatorio	Il *sistema*$_G$ deve fornire all'amministratore la possibilità di visualizzare la lista totale degli ingredienti inclusi nella *prenotazione*$_G$.	S
ROF 34	Obbligatorio	Il *sistema*$_G$ deve consentire all'amministratore di visualizzare tutti gli ordini confermati per il ristorante selezionato.	S
RDF 35	Desiderabile	Il *sistema*$_G$ deve permettere di annullare l'*ordinazione*$_G$ collaborativa nel tempo utile per farlo	NS
RDF 36	Desiderabile	Il *sistema*$_G$ invia una notifica a tutti gli utenti associati alla *prenotazione*$_G$ la cui *ordinazione*$_G$ collaborativa è stata annullata	NS
ROF 37	Obbligatorio	Il *sistema*$_G$ deve fornire un'interfaccia per consentire agli utenti non autenticati di registrarsi come utenti base.	S
ROF 38	Obbligatorio	L'utente/L'amministratore deve poter inserire le sue informazioni personali durante la registrazione come nome, cognome, email, password.	S
ROF 39	Obbligatorio	L'utente/L'amministratore deve poter confermare di volersi registrare con le informazioni fornite prima di completare la registrazione.	S

Codice	*Rilevanza*	*Descrizione*	*Stato*
ROF 40	Obbligatorio	Il *sistema*$_G$ deve gestire correttamente gli errori nell'inserimento delle informazioni durante la registrazione.	S
ROF 41	Obbligatorio	L'utente base e l'amministratore devono poter visualizzare il menu' del ristorante selezionato	S
RDF 42	Desiderabile	L'amministratore deve poter modificare le informazioni del proprio ristorante (nome, indirizzo, orari, coperti e tipologia di cucina)	NS
RDF 43	Desiderabile	La modifica delle informazioni del ristorante da parte dell'amministratore non deve causare problemi di sincronizzazione nella visualizzazione, ricerca del ristorante e nell'*ordinazione*$_G$ da parte dell'utente base	NS
RDF 44	Desiderabile	L'utente autenticato deve poter visualizzare una lista con le prenotazioni passate	S
RDF 45	Desiderabile	L'utente autenticato deve poter rilasciare una recensione (con anche una votazione di gradimento tramite stelle) sui ristoranti nei quali ha effettuato almeno una *prenotazione*$_G$	NS
ROF 46	Obbligatorio	L'utente autenticato deve poter visualizzare gli ordini di un tavolo	S
RDF 47	Desiderabile	L'utente autenticato deve poter visualizzare le recensioni rilasciate ed eventualmente eliminarle	NS
RDF 48	Desiderabile	L'utente generico può visualizzare le recensioni di un ristorante e visualizzare per ognuna di essa le relative informazioni	NS
ROF 49	Obbligatorio	L'amministratore deve poter inserire le informazioni del ristorante di cui è amministratore	S
ROF 50	Obbligatorio	L'utente/L'amministratore deve poter inserire la propria email e la password durante la fase di login.	S
ROF 51	Obbligatorio	Il *sistema*$_G$ deve verificare che le informazioni inserite nel login corrispondano ad un account già esistente nel *sistema*$_G$.	S
RDF 52	Desiderabile	Il *sistema*$_G$ deve predisporre un'opzione per il recupero della password	NS
RDF 53	Desiderabile	L'utente non autenticato deve poter inserire la propria email durante il *processo*$_G$ di recupero password.	NS
RDF 54	Desiderabile	Se l'email inserita dall'utente corrisponde ad un account nel *sistema*$_G$, il *sistema*$_G$ deve inviare un'email contenente un link per il recupero della password.	NS
ROF 55	Obbligatorio	Se l'email inserita dall'utente non corrisponde a un account nel *sistema*$_G$, il *sistema*$_G$ lo mostra a schermo.	S
RDF 56	Desiderabile	L'utente deve poter accedere a una sezione dedicata per il recupero della password tramite il link fornito nell'email di recupero password.	NS
ROF 57	Obbligatorio	Il *sistema*$_G$ deve mostrare una lista di prenotazioni per l'utente base, ognuna con le seguenti informazioni: nome del ristorante, data, ora, stato della *prenotazione*$_G$ (se è ancora attiva o già completata) e numero di persone coinvolte.	S
RDF 58	Desiderabile	Il *sistema*$_G$ ordina la lista delle prenotazioni per data della *prenotazione*$_G$.	S
RDF 59	Desiderabile	L'utente base deve poter inserire le proprie allergie/intolleranze, se presenti, durante la modifica del profilo.	NS
RDF 60	Desiderabile	L'utente autenticato e l'amministratore devono poter inserire nome, cognome, email e password nell'area di modifica delle proprie informazioni.	NS

Codice	*Rilevanza*	*Descrizione*	*Stato*
ROF 61	Obbligatorio	Il *sistema*$_G$ deve verificare che l'email inserita dall'utente autenticato sia valida e non sia già presente nel *sistema*$_G$.	S
ROF 62	Obbligatorio	L'utente base autenticato deve poter accedere alla funzionalità di *ordinazione*$_G$ di un piatto.	S
ROF 63	Obbligatorio	Il *sistema*$_G$ deve consentire all'utente di visualizzare e togliere ingredienti del piatto selezionato.	S
RDF 64	Desiderabile	L'amministratore deve poter visualizzare le recensioni del proprio ristorante.	NS
RDF 65	Desiderabile	L'amministratore deve poter rispondere alle recensioni del proprio ristorante.	NS
RDF 66	Desiderabile	L'utente deve poter visualizzare le risposte alla propria recensione.	NS
ROF 67	Obbligatorio	Il *sistema*$_G$ deve poter consentire di modificare l'ordine ad un utente prima della scadenza del tempo previsto, inserendo piatti, modificando ingredienti e quantità delle pietanze ordinate.	S
RDF 68	Desiderabile	Il *sistema*$_G$ permette di far visualizzare all'amministratore il dettaglio degli ingredienti necessari per ogni giornata.	NS
ROF 69	Obbligatorio	Il *sistema*$_G$ deve consentire all'utente autenticato di accedere alla funzionalità di logout.	S
RDF 70	Desiderabile	Dopo che l'utente ha selezionato l'opzione di logout, il *sistema*$_G$ deve richiedere una conferma esplicita dall'utente prima di procedere con la disconnessione.	NS
ROF 71	Obbligatorio	Dopo aver terminato la sessione dell'utente, il *sistema*$_G$ deve reindirizzare l'utente alla pagina di accesso o a una pagina di destinazione predefinita.	S
RDF 72	Desiderabile	L'utente autenticato deve poter visualizzare le informazioni del suo profilo.	NS
RDF 73	Desiderabile	L'amministratore deve poter visualizzare tutte le chat aperte in precedenza da altri utenti.	NS
RDF 74	Desiderabile	L'amministratore deve poter rispondere ai messaggi inviati da gli utenti	NS
ROF 75	Obbligatorio	Il *sistema*$_G$ deve poter generare un link di invito per la *prenotazione*$_G$	S
RDF 76	Desiderabile	Il *sistema*$_G$ permette di far visualizzare all'amministratore in dettaglio la lista delle prenotazioni di ogni giornata.	S
ROF 77	Obbligatorio	Il *sistema*$_G$ invia una notifica all'amministratore in base agli aggiornamenti sul conto di una *prenotazione*$_G$ del suo ristorante.	S

Tabella 2: Tabella dei requisiti funzionali

Rilevanza	**Soddisfatti**	**Non soddisfatti**	**Totale**	**Percentuale soddisfatti**
Obbligatori	40	0	40	100%
Desiderabili	4	33	37	10.81%

Tabella 3: Tabella riassuntiva dei requisiti funzionali