SWEnergy

project.swenergy@gmail.com

Specifiche tecniche

Descrizione: Descrizione delle scelte architetturali e di design del progetto

Stato	Approvato
Data	19/05/2024
Redattori	Carlo Rosso
	Alessandro Tigani Sava
Verificatori	Matteo Bando
Approvatore	Davide Maffei
Destinatari	prof. Tullio Vardanega
	prof. Riccardo Cardin
Versione	1.0.0

Registro delle modifiche

Versione	Data	Redattore	Verificatore	Approvatore	Modifiche
1.0.0	2024-05-19	/	/	Davide Maffei	Approvazione
0.2.3	2024-05-18	Carlo Rosso	Giacomo Gualato	/	Aggiornamento dell'introduzione e verifica delle fonti
0.2.2	2024-05-17	Alessandro Tigani Sava	Matteo Bando	/	Descrizione dei pattern usati, del setup di sviluppo e deploy inerenti il backend
0.2.1	2024-05-16	Carlo Rosso	Matteo Bando	/	Conclusione della descrizione dei pattern usati, descrizione del setup di sviluppo e deploy e riassunto dei requisiti completati nel frontend
0.2.0	2024-05-15	Carlo Rosso	/	/	Ridefinizione della struttura del documento. Descrizione dell'architettura di deployment e dei pattern architetturali. Inizio della descrizione dei pattern usati nel frontend
0.1.0	2024-04-03	Carlo Rosso	/	/	Prima stesura delle sezioni 2 e 3
0.1.0	2024-03-30	Carlo Rosso	/	/	Definizione della struttura generale del documento

Indice

1 Introduzione

1.1 Scopo del documento

Il presente documento, intitolato "Specifiche Tecniche", espone e approfondisce le scelte architetturali e di design adottate dal gruppo SWEnergy per lo sviluppo del progetto "*Easy Meal*", proposto dall'azienda Imola Informatica[19/05/2024]. In particolare, il documento mira a giustificare le scelte implementate al fine di consentire agli sviluppatori di comprendere e mantenere il codice in modo efficace ed efficiente. Inoltre, è inclusa una sezione dedicata ai requisiti implementati per garantire che il prodotto soddisfi le richieste del proponente.

1.2 Organizzazione del documento

Il documento è strutturato in quattro sezioni principali:

- **Setup**: descrive l'ambiente di sviluppo e di *deployment* utilizzato per lo sviluppo del progetto;
- **Tecnologie adottate**: si compone di una descrizione di ciascuna tecnolgia adottata per lo sviluppo del progetto. Le tecnologie sono divise in tre categorie: di codifica, di analisi statica e di analisi dinamica. Alla fine della sezione è presente una tabella riassuntiva;
- **Architettura**: descrive l'architettura del progetto, sono infatti descritti i *design pattern* e di *deployment* adottati;
- **Requisiti**: sono riportati i requisiti concordati con il proponente e descritti in dettaglio nel documento "Analisi dei Requisiti v3.0.0". Per ciascun requisito è indicato lo stato di implementazione. Infine, è presente un grafico riassuntivo dello stato di avanzamento dei requisiti.

1.3 Scopo del prodotto

"*Easy Meal*" è una *web app*^{G} progettata per gestire le prenotazioni presso i ristoranti, sia dal lato dei clienti che dei ristoratori. Il prodotto finale è composto da due parti:

- **Cliente**^{G} : consente ai clienti di prenotare un tavolo presso un ristorante, visualizzare il menù e effettuare un ordine^{G} ;
- **Ristoratore**: consente ai ristoratori di gestire le prenotazioni e gli ordini dei clienti, oltre a visualizzare la lista degli ingredienti necessari per preparare i piatti ordinati.

1.4 Glossario

Al fine di prevenire ambiguità linguistiche e garantire una coerenza nell'utilizzo delle terminologie attraverso i documenti, il *team* ha compilato un documento interno denominato "Glossario". Questo documento fornisce definizioni chiare e precise per i termini che potrebbero risultare ambigui o generare incomprensioni nel testo principale. I termini inclusi nel Glossario sono facilmente identificabili grazie a un apice 'G' (ad esempio, parola^{G}). Questa pratica agevola la consultazione del Glossario per una comprensione approfondita dei termini tecnici o specifici utilizzati nel contesto del progetto.

1.5 Riferimenti

1.5.1 Normativi

- "Norme di progetto v3.0.0";
- Documento del capitolato d'appalto C3 - *Easy Meal*[19/05/2024];
- Regolamento del progetto[19/05/2024];

1.5.2 Informativi

- Glossario v2.0.0;
- Analisi dei Requisiti v3.0.0;
- Linguaggio UML - Diagramma delle classi[19/05/2024];

1.5.3 Tecnologici

- Typescript[19/05/2024];
- Angular[19/05/2024];

- Axios[19/05/2024];
- NestJS[19/05/2024];
- Drizzle[19/05/2024];
- Docker[19/05/2024];
- Insomnia[19/05/2024];
- Tailwind CSS[19/05/2024];

2 Setup

In questa sezione viene approfondito il metodo per eseguire l'applicativo sviluppato. In particolare, si descrive come configurare l'ambiente di esecuzione e come eseguire il programma; si illustra come eseguire i test e come configurare l'ambiente di sviluppo.

2.1 Installazione

2.1.1 Pre-requisiti

Per eseguire l'applicativo è necessario avere installato Docker nel proprio sistema. Per installare Docker, si può fare riferimento alla documentazione ufficiale di Docker: Docker Desktop[19/05/2024]. Per effettuare le operazioni di *dump* e *load* del *database* sarà necessario disporre di una installazione PostgreSQL e dei comandi `pg_dump`, `pg_load`. Infine, è necessario avere installato git^G per clonare il *repository*.

2.1.2 Prima esecuzione

Di seguito sono riportati i passi da seguire per avviare Easy-Meal:

1. Clonare il *repository*^{G} :

   ```
   git clone https://github.com/Project-SWEnergy/Easy-Meal.git
   ```

2. Entrare nella cartella del progetto:

   ```
   cd Easy-Meal
   ```

3. Avviare il *daemon* di Docker;
4. Eseguire la *build* del progetto e avviare i *container*:
   `docker-compose up`
5. Posizionarsi all'interno della cartella `backend`:
   `cd backend`
6. Effettuare il caricamento dello schema relativo al *database*:
   `npm run postgres-load`
7. Inserire la *password* relativa al *database* (*password*: postgres).
8. Aprire il seguente indirizzo nel *browser*:
   `http://localhost:4200`

Dopo aver seguito i passi sopra descritti, l'applicativo Easy-Meal è pronto per essere utilizzato.

2.2 Esecuzione

Se l'applicativo è già stato avviato in precedenza, è possibile riavviarlo seguendo i seguenti passi:

1. Avviare il *daemon* di Docker;
2. Eseguire la *build* del progetto e avviare i *container*:
   `docker-compose up`
3. Aprire il seguente indirizzo nel *browser*:
   `http://localhost:4200`

2.3 Arresto

Per arrestare l'applicativo, è sufficiente eseguire il comando `docker-compose down` nella cartella del progetto.

2.4 Test

Sono configurati due sistemi di test indipendenti: i test del *backend* e i test del *frontend*.

2.4.1 *Backend*

Per eseguire i test del *backend* è necessario avere installato Node.js e quindi anche npm, oltre che Nest. Per installare Node.js, si può fare riferimento alla documentazione ufficiale di Node.js[19/05/2024]. Di seguito sono riportati i passi da seguire per avviare i test.

1. Entrare nella cartella del progetto:
   `cd Easy-Meal\backend`
2. Eseguire il comando:
   `npm install`
3. Eseguire uno degli *script* per l'esecuzione di test, come ad esempio:
   `npm run test:cov`
4. Aprire il seguente *file* con il *browser* per visualizzare i risultati:
   `..\Easy-Meal\backend\coverage\lcov-report\index.html`

Vengono messi a disposizione diversi *script* per l'esecuzione dei test tramite il *framework* Jest:

- `npm run test`: esecuzione completa dei test di unità.
- `npm run test:cov`: esecuzione completa dei test di unità con creazione di un *report* inerente la *coverage*.
- `npm run test:watch`: esegue automaticamente i test quando rileva modifiche nei *file* di origine o nei *file* di test.

2.4.2 *Frontend*

Per eseguire i test del *frontend* è necessario avere installato Node.js e quindi anche npm. Per installare Node.js, si può fare riferimento alla documentazione ufficiale di Node.js[19/05/2024]. Di seguito sono riportati i passi da seguire per avviare i test:

1. Entrare nella cartella del progetto:
   `cd Easy-Meal\frontend` (su Windows)
   `cd Easy-Meal/frontend` (su Unix)
2. Eseguire il comando:
   `npm install`
3. Eseguire il comando:
   `ng test`
4. Aprire il seguente indirizzo nel *browser*:
   `http://localhost:9876`
5. Cliccare sul pulsante `DEBUG` per eseguire i test.

Non solo, Angular mette a disposizione un servizio per visualizzare la copertura dei test, il codice e la qualità del codice. Per avviare il servizio di visualizzazione della copertura dei test, si seguano i seguenti passi:

1. Entrare nella cartella del progetto:
   `cd Easy-Meal\frontend` (su Windows)
   `cd Easy-Meal/frontend` (su Unix)
2. Eseguire il comando:
   `npm install`
3. Eseguire il comando:
   `ng test --code-coverage=true`
4. Aprire il seguente indirizzo nel *browser*:
   `http://localhost:9876`
5. Cliccare sul pulsante `DEBUG` per eseguire i test;
6. Aprire il seguente file con il *browser*:
   `coverage/po-c-frontend/index.html`

2.5 Configurazione dell'ambiente di sviluppo

2.5.1 Backend

Come per i test, anche in questo caso è necessario avere installato Node.js, npm e Nest. Di seguito sono riportati i passi da seguire per configurare l'ambiente di sviluppo del *backend*:

1. Entrare nella cartella del progetto relativa al *backend*:
   `cd Easy-Meal/backend`
2. Modificare il file `.env` sostituendo la stringa in DATABASE_URL:
   `"postgresql://postgres:postgres@db:5432/easymeal?schema=public"`
   con la stringa:
   `"postgresql://postgres:postgres@localhost:5432/easymeal?schema=public"`
3. Avviare il *container*:
   `docker-compose up`
4. Installare le dipendenze:
   `npm install`
5. Avviare il *server* di sviluppo:
   `npm run start:dev`

In questo modo, è possibile sviluppare il *backend* senza dover ricostruire il *container* del *backend* ad ogni modifica, poiché Nest lo ricompila automaticamente ad ogni cambiamento.

2.5.2 Frontend

Come per i test, anche in questo caso è necessario avere installato Node.js e npm. Di seguito sono riportati i passi da seguire per configurare l'ambiente di sviluppo del *frontend*:

1. Entrare nella cartella del progetto:
   `cd Easy-Meal`
2. Avviare i *container*:
   `docker-compose up`
3. Aprire la *dashboard* di Docker e bloccare il *container* `easymeal-frontend-<N>`.

4. Installare le dipendenze:

   ```
   npm install
   ```

5. Avviare il *server* di sviluppo:

   ```
   npm run start
   ```

6. Aprire il seguente indirizzo nel *browser*:

   ```
   http://localhost:4200
   ```

In questo modo è possibile sviluppare il *frontend* senza dover eseguire la *build* del *container* del *frontend* ad ogni modifica, invece Angular ricompila automaticamente il *frontend* ad ogni modifica. Infatti, ogni modifica di qualche *file* del *frontend* causa un aggiornamento della pagina *web*.

3 Tecnologie adottate

3.1 Tecnologie per la codifica

3.1.1 TypeScript

I membri del *team* SWEnergy, prima di iniziare il progetto, possedevano una solida conoscenza di C++. Alcuni di loro avevano esperienza anche con Java, Python e JavaScript. Il referente ha illustrato che Imola Informatica fa un ampio uso di TypeScript e Java per lo sviluppo *software*, sottolineando inoltre che per creare un'applicazione *web*, specialmente per quanto riguarda il *frontend*, è consigliato utilizzare JavaScript o TypeScript. Partendo da queste considerazioni, SWEnergy ha optato per l'uso di JavaScript come linguaggio principale per lo sviluppo del progetto, poiché consente di scrivere codice sia per il *frontend* che per il *backend*. Infine, è stato scelto TypeScript, un superset di JavaScript, per la sua capacità di aggiungere tipi statici al linguaggio, migliorando la robustezza, la manutenibilità e la leggibilità del codice. Attraverso l'analisi statica del codice, è possibile individuare errori di programmazione direttamente durante la fase di sviluppo, evitando che si manifestino durante l'esecuzione, il che migliora la qualità del *software* e riduce il tempo di sviluppo.

3.1.2 Angular

Per lo sviluppo del *frontend*, SWEnergy ha scelto di adottare Angular, un *framework open-source* per la creazione di applicazioni *web*. Questa decisione è stata guidata dalla familiarità di due membri del *team* con questo *framework*. SWEnergy ha inteso capitalizzare sulle competenze pregresse di questi due membri per ridurre il tempo di apprendimento, organizzando *workshop* dedicati per il resto del *team* al fine di uniformare le conoscenze.

Per il *frontend*, abbiamo optato per l'utilizzo della versione 17.3.3 di Angular, la più recente disponibile al momento. Abbiamo scelto questa versione ritenendola la più adatta per il nostro apprendimento, considerando anche possibili implementazioni future di Angular. È importante notare che questa versione dispone di risorse online meno dettagliate rispetto alle versioni precedenti. Tuttavia, la documentazione ufficiale fornita da Angular risulta essere sufficientemente chiara e completa, agevolando il nostro processo di apprendimento del *framework*.

3.1.3 Axios

Per gestire le chiamate HTTP dal *frontend* al *backend*, SWEnergy ha adottato Axios, un *client* HTTP basato su *promise*. La decisione di utilizzare Axios è stata influenzata dalla raccomandazione del referente aziendale, che ha sottolineato la facilità e la velocità con cui è possibile effettuare *test* di carico sulle API^G utilizzando questo strumento.

3.1.4 Tailwind CSS

Vista l'ampia necessità di utilizzo di CSS nello sviluppo del *frontend*, il *team* ha scelto di adottare Tailwind, un *framework* che introduce un nuovo approccio nel linguaggio CSS. L'obiettivo è rendere il codice CSS più riusabile e manutenibile, semplificando così il processo di sviluppo e migliorando l'efficienza del *team*.

3.1.5 Node.js

Per la realizzazione del *backend*, SWEnergy ha optato per Node.js, un *runtime* JavaScript *open-source* che si basa sul motore JavaScript V8 di Google Chrome. La scelta di Node.js è stata essenzialmente dettata dalla necessità di un *runtime* JavaScript per eseguire il codice

del *backend*, considerando la scelta di TypeScript come linguaggio principale. Node.js rappresenta la scelta più diffusa e supportata per questo tipo di scenario.

3.1.6 Nest.js

Per la gestione delle API^G , SWEnergy ha optato per l'utilizzo di Nest.js, un *framework* basato su Node.js progettato per la creazione di applicazioni *server-side* efficienti, scalabili e facili da mantenere. La decisione di adottare Nest.js è stata guidata dalla raccomandazione del referente aziendale, che ha evidenziato la sua similitudine con Angular nella dinamica di utilizzo della *dependency injection*. Questo approccio consente la creazione di un'applicazione modulare, dove i componenti possono essere facilmente sostituiti e testati.

3.1.7 Drizzle

Per la gestione del *database*, SWEnergy ha scelto Drizzle, un ORM^G (*Object Relational Mapping*) progettato per TypeScript. Drizzle consente di definire il *database*, le relazioni tra le tabelle e le *query* direttamente in TypeScript, fornendo quindi anche le classi necessarie per interagire con il *database*. La decisione di adottare Drizzle è stata motivata dalla sua somiglianza con SQL, in quanto tutti i membri del *team* sono molto familiari con questo linguaggio. Inoltre, la similitudine con SQL rende le *query* estremamente efficienti, poiché la traduzione è diretta e non richiede una complessa analisi.

3.1.8 PostgreSQL

SWEnergy ha optato per l'adozione di PostgreSQL come *database* per il progetto, poiché tutti i membri del *team* possiedono una discreta esperienza con questo DBMS. La decisione è stata motivata dalla consapevolezza che ci sono già numerose nuove tecnologie da apprendere nel contesto del progetto; pertanto, per quanto riguarda il *database*, è stata preferita una soluzione con cui tutti i membri del *team* si sentono già a proprio agio.

3.1.9 Docker

L'adozione di Docker per la creazione di *container* offre numerosi vantaggi fondamentali. In primo luogo, Docker consente di rendere il prodotto eseguibile su qualunque *hardware*, garantendo una maggiore portabilità e facilità di distribuzione. Grazie alla standardizzazio-

ne dell'ambiente di esecuzione tramite i *container* Docker, è possibile eliminare i problemi legati alla differenza di configurazioni *hardware* e *software* tra i vari ambienti di sviluppo, test e produzione.

Inoltre, l'utilizzo di Docker permette di semplificare il processo di gestione delle dipendenze e delle librerie necessarie per il progetto, fornendo un ambiente isolato e riproducibile in cui eseguire l'applicazione senza interferenze esterne. Ciò contribuisce a ridurre i conflitti e i problemi di compatibilità tra le diverse componenti del sistema, migliorando la coerenza e l'affidabilità del *software*.

Infine, Docker offre la possibilità di scalare orizzontalmente il prodotto utilizzando tecnologie come Kubernetes. Questo significa che è possibile gestire facilmente un carico di lavoro in crescita distribuendo automaticamente i *container* su più nodi, garantendo così una maggiore disponibilità e prestazioni dell'applicazione anche in presenza di un elevato numero di richieste. In sintesi, l'adozione di Docker offre un modo efficiente e affidabile per gestire l'intero ciclo di vita dell'applicazione, garantendo flessibilità, portabilità e scalabilità.

3.2 Tecnologie per l'analisi statica del codice

3.2.1 Language Server Protocol

L'adozione di un *Language Server Protocol* (LSP) nel nostro processo di sviluppo rappresenta un passo significativo verso un ambiente di sviluppo più efficiente e produttivo. Grazie all'utilizzo di un LSP, siamo in grado di integrare funzionalità avanzate di analisi statica e segnalazione degli errori direttamente nel nostro *editor* di codice, garantendo una rapida identificazione e risoluzione di potenziali problemi di programmazione.

3.3 Tecnologie per l'analisi dinamica del codice

3.3.1 Insomnia

Per verificare il funzionamento delle API^G e del *backend*, SWEnergy ha scelto di utilizzare Insomnia, un'applicazione *open-source* progettata per il *testing* delle API REST. Il *team* ha creato un ambiente condiviso all'interno di Insomnia, consentendo l'esecuzione di tutte le API disponibili e l'osservazione dei risultati. Questo ambiente memorizza le diverse tipologie di chiamate e vari *set* di dati di esempio. Insomnia facilita l'esecuzione di test

manuali, permettendo di modificare agevolmente i dati inviati al *backend* e di visualizzare i risultati restituiti. Questo consente una rapida verifica della corrispondenza tra i risultati ottenuti e quelli attesi, migliorando l'efficienza e l'accuratezza dei test.

3.3.2 Angular

Angular fornisce nativamente il comando `ng test`, che è uno strumento potente per eseguire test unitari e di integrazione.

I test unitari sono progettati per verificare l'accuratezza delle singole unità di codice, come componenti e servizi. Con Angular, `ng test` sfrutta il framework Jasmine per definire i test e Karma come test *runner* per eseguire i test in un ambiente di *browser*. Questo permette di simulare il comportamento dell'applicazione in un contesto realistico, garantendo che ogni unità funzioni come previsto.

Oltre ai test unitari, `ng test` supporta anche i test di integrazione, che verificano l'interazione tra più unità di codice. Questi test sono cruciali per assicurare che i vari componenti dell'applicazione funzionino correttamente insieme e che l'integrazione tra loro non introduca *bug* o comportamenti inattesi. Anche in questo caso, Jasmine offre la possibilità di definire i *mock* e le *stub* necessari per simulare le dipendenze esterne e garantire che i test siano eseguiti in modo isolato.

3.3.3 Nest.js

Nest.js fornisce una integrazione di *default* con Jest, un *framework* di *testing* JavaScript ampiamente utilizzato e noto per la sua velocità e semplicità. Si tratta della scelta ideale per eseguire test siccome offre un'ottima integrazione con NestJS, permettendo di sfruttare appieno le caratteristiche del *framework* come i moduli, i servizi e i *controller*. Inoltre, Jest supporta il *mocking* e la simulazione delle dipendenze, facilitando l'isolamento delle unità di codice da testare. Le sue capacità di test asincroni e il supporto integrato per TypeScript, che è comunemente utilizzato con NestJS, garantiscono una scrittura di test più fluida e meno soggetta a errori. Infine, Jest fornisce un'interfaccia utente intuitiva e *report* di copertura del codice dettagliati, rendendo più semplice identificare e correggere i *bug* e migliorare la qualità complessiva del *software*.

3.4 Riepilogo

La seguente tabella fornisce un riepilogo dettagliato delle tecnologie adottate dal gruppo per lo sviluppo dell'applicazione Easy Meal.

Nome	Descrizione	Versione
Typescript	Linguaggio di programmazione	5.4
Angular	*Framework* per lo sviluppo di applicazioni *web*	17.3.3
Axios	Libreria per effettuare richieste HTTP	1.6.8
Tailwind CSS	*Framework* per la gestione di *file* CSS	3.4.3
Node.js	*Runtime* JavaScript	20.12.0
NestJS	*Framework* per lo sviluppo di applicazioni *server-side*	10.3.6
Drizzle	*Object Relational Mapping*	0.30.6
PostgreSQL	Sistema di gestione di basi di dati	16.2
Docker	Piattaforma per lo sviluppo, il *deploy* e l'esecuzione di applicazioni	/
Insomnia	Crezione di un *batch* di *test* collaborativo	8.6.1

Tabella 2: Tabella delle tecnologie adottate

4 Architettura

Approfondiamo l'architettura scelta per Easy-Meal, spiegando le scelte architetturali e i *pattern* utilizzati per lo sviluppo dell'applicazione partendo dalla struttura generale del sistema e passando poi ai dettagli implementativi.

4.1 Architettura di Deployment

SWEnergy ha deciso di adottare un'architettura monolitica a tre livelli (*3-tier architecture*) per implementare Easy-Meal. Questa scelta è stata determinata dalla necessità di costruire una *web application* che permetta agli utenti di accedere al sistema da qualsiasi dispositivo connesso a Internet. L'architettura a tre livelli separa la logica di presentazione dalla logica

di *business* e dal *database*. I livelli comunicano tra loro attraverso interfacce ben definite, in particolare:

- **Livello di Presentazione (*Frontend*)**: rappresentato da Angular, riceve i dati dall'utente e visualizza i risultati restituiti dal *backend*.
- **Livello di Logica Applicativa (*Backend*)**: rappresentato da NestJS. Gestisce la logica di *business*, elabora le richieste provenienti dal *frontend* e interagisce con il *database* per recuperare o salvare i dati.
- **Livello di Dati (*Database*)**: rappresentato dal *database* PostgreSQL, si occupa della gestione dei dati.

Questo approccio ha permesso al team di sviluppo di lavorare in modo parallelo sul *frontend* e sul *backend*, riducendo i tempi di sviluppo e facilitando il *testing* del sistema. Inoltre, offre la possibilità di scalare ciascun livello separatamente, in modo da garantire prestazioni ottimali e una maggiore flessibilità. Infine, è possibile sostituire l'implementazione di uno dei livelli senza dover toccare gli altri, garantendo una maggiore manutenibilità del sistema. In particolare lo sviluppo di Easy-Meal è cominciato partendo dal *database*, l'elemento su cui abbiamo posto maggiore enfasi, in quanto è il cuore del sistema. Successivamente sono stati sviluppati il *backend* e il *frontend*, in modo da garantire una corretta integrazione tra i diversi livelli dell'applicazione. SWEnergy ha deciso di utilizzare questa metodologia, perché le funzionalità messe a disposizione da Easy-Meal al livello della logica di *business* sono principalmente delle semplici operazioni CRUD, che non richiedono un'interazione complessa tra i diversi livelli dell'applicazione.

4.2 Pattern Architetturali

Easy-Meal è stato sviluppato utilizzando due *pattern* architetturali: Model-View-Controller e Dependency Injection. Infatti, SWEnergy ha scelto di adottare Angular e NestJS, due *framework* che implementano questi *pattern* in modo nativo, garantendo una struttura ben definita e una maggiore manutenibilità del codice.

4.2.1 Model-View-Controller

Il *pattern* architetturale Model-View-Controller (MVC) è stato utilizzato per suddividere le responsabilità tra i diversi componenti dell'applicazione. In particolare, il *model* rappresenta i dati dell'applicazione e fornisce i metodi di accesso e di modifica di tali dati. NestJS è il *framework* che implementa il *backend* nell'applicativo e fornisce le API^G di accesso ai dati di modifica tramite i *controller*. Angular, invece, è il *framework* che implementa il *frontend* e fornisce le interfacce grafiche per la gestione delle interazione con l'utente, ovvero la *view*. I *component* e i *controller* sono responsabili di gestire le interazioni tra la *view* e il *model*, aggiornando i dati in base alle azioni dell'utente. L'aggiornamento della visualizzazione dei dati è gestito da Angular, che attraverso il *data binding* permette di mantenere sincronizzati i dati presenti nel *model* con la *view*.

4.2.2 Dependency Injection

Il *pattern* Dependency Injection (DI) è stato utilizzato per gestire le dipendenze tra i diversi componenti dell'applicazione. In particolare, Angular utilizza la DI per iniettare i servizi all'interno dei componenti, permettendo di scrivere codice più modulare e manutenibile. NestJS, invece, utilizza la DI per iniettare i servizi all'interno dei *controller*, permettendo di separare la logica di *business* dalla logica di accesso ai dati. Questo approccio consente di creare componenti indipendenti e riutilizzabili, facilitando la manutenibilità dell'applicazione. Infine la divisione in moduli rende il codice facilmente testabile, in quanto è possibile sostituire i servizi con dei *mock* per testare i componenti in modo isolato.

4.3 Frontend

Di seguito sono proposti alcuni diagrammi delle classi per il *frontend* di Easy-Meal. Poiché la struttura del *frontend* è piuttosto vasta, ma ridondante, nel senso che il *pattern dependency injection* è applicato allo stesso modo per tutti i componenti, si è deciso di selezionare alcuni esempi significativi.

4.3.1 GenericService

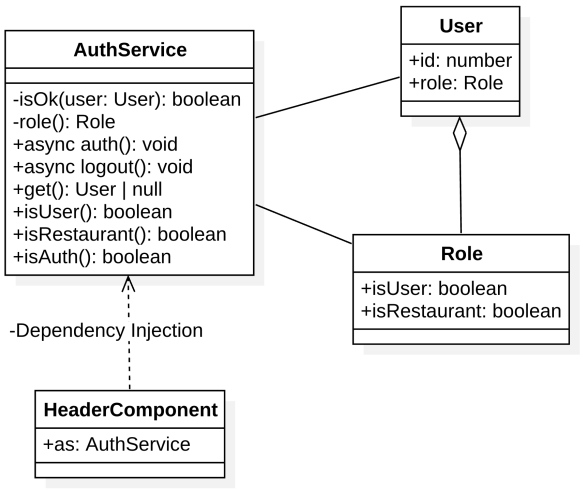

Figura 1: Diagramma delle classi per AuthService

In questo caso, viene mostrato come viene iniettato un servizio all'interno di un componente. Lo *specimen* scelto è il servizio AuthService, che è responsabile della gestione dell'autenticazione dell'utente. Iniettando AuthService all'interno del componente HeaderComponent, i metodi pubblici definiti in AuthService possono essere utilizzati per gestire l'autenticazione dell'utente. In particolare, HeaderComponent utilizza AuthService per gestire i *link* del menu di navigazione in base allo stato di autenticazione e al ruolo dell'utente. In questo modo viene favorita la separazione delle responsabilità e la modularità del codice, rendendo il componente HeaderComponent più manutenibile e permettendo di riutilizzare AuthService in altri componenti.

4.3.2 MessageService

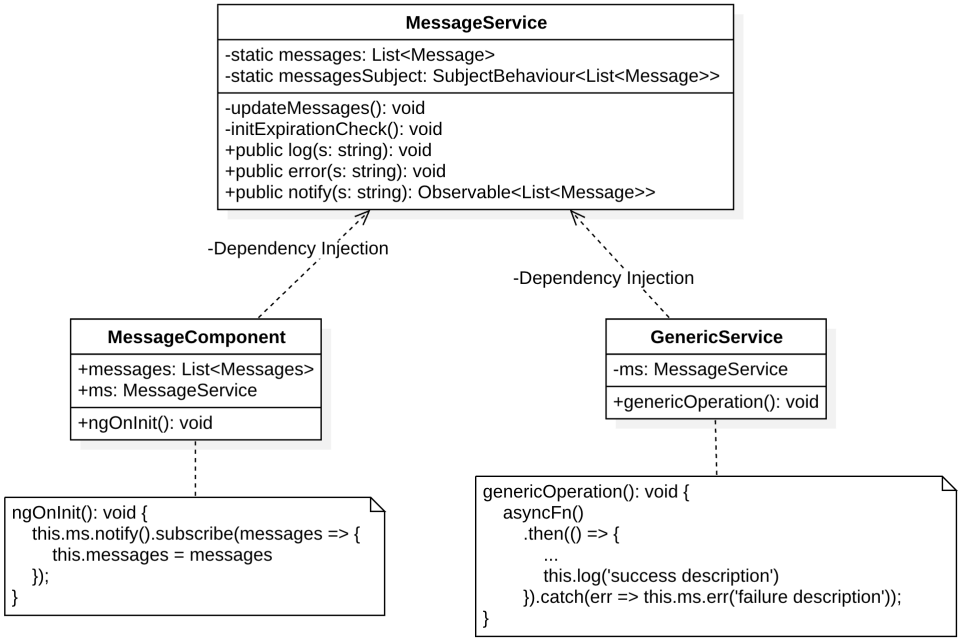

Figura 2: Diagramma delle classi per il MessageService

La *dependency injection* (DI) in Angular è un *design pattern* che consente di fornire dipendenze alle classi in modo automatizzato. In questo caso, la classe MessageService viene iniettata sia nella classe MessageComponent che nella classe GenericService. Questo è reso possibile dichiarando MessageService come dipendenza nei costruttori delle rispettive classi. L'annotazione `@Injectable` viene utilizzata sulla classe MessageService per indicare che questa può essere iniettata come dipendenza.

MessageService è implementato come *singleton* in Angular. Un *singleton* garantisce che una sola istanza della classe venga creata e condivisa in tutta l'applicazione. Questo è importante per la gestione centralizzata dello stato e delle operazioni comuni, come la gestione dei messaggi. In Angular, il *pattern singleton* è implicitamente implementato tramite il sistema DI, poiché il servizio è registrato nel *root injector*, garantendo una singola istanza. Angular offre un potente meccanismo di *data binding* che consente alla vista di aggiornarsi automaticamente quando i dati del modello cambiano. Nella classe MessageComponent, il metodo `ngOnInit` utilizza l'osservabile restituito da ms.notify() per iscriversi agli aggiornamenti dei messaggi. Quando i messaggi vengono aggiornati nel MessageService, la vista associata a MessageComponent si aggiorna automaticamente per riflettere i nuovi

dati. Questo è facilitato dall'utilizzo di Subject o BehaviorSubject in MessageService, che emettono nuovi valori quando i messaggi cambiano.
In sostanza, in questo modo si utilizza un servizio per condividere dati tra componenti che non hanno una relazione gerarchica.

4.4 Backend

Il *backend* dell'applicazione è stato realizzato utilizzando NestJS, il quale sfrutta diversi *pattern* creazionali, strutturali ed architetturali.

4.4.1 Dependency Injection

Gestisce la creazione delle dipendenze. NestJS integra nativamente un sistema di *Dependency Injection* che semplifica l'iniezione delle dipendenze tra i vari componenti dell'applicazione. Utilizza il concetto di *provider* per registrare e risolvere le dipendenze. Un esempio di utilizzo è la registrazione e l'iniezione di servizi nei moduli, generalmente eseguita come nel codice di esempio sottostante.

```
1  import { Injectable } from '@nestjs/common';
2
3  @Injectable()
4  export class UsersService { // Implementation }
5
6  import { Module } from '@nestjs/common';
7  import { UsersService } from './users.service';
8
9  @Module({
10   providers: [UsersService],
11   exports: [UsersService],
12 })
13 export class UsersModule {}
```

I vantaggi nell'utilizzo della Dependency Injection sono:

- **Separazione delle Preoccupazioni**: migliora la modularità e la manutenibilità del codice.

- **Riusabilità del Codice**: i componenti possono essere riutilizzati in diverse parti dell'applicazione senza dover gestire manualmente le loro dipendenze.
- **Testabilità**: semplifica il *testing* unitario poiché consente di sostituire facilmente le dipendenze con versioni *mock* durante i test.
- **Flessibilità e Scalabilità**: consente di sostituire o estendere facilmente le dipendenze senza modificare il codice esistente.

4.4.2 Module Pattern

Il codice dell'applicazione è suddiviso in moduli, ognuno dei quali incapsula una funzionalità specifica. I moduli possono importare altri moduli, rendendo l'applicazione altamente modulare e facilmente manutenibile. Ogni modulo può contenere *controller*, servizi, *provider* e altre risorse. I moduli in NestJS sono definiti usando il decoratore "Module", che specifica i componenti che fanno parte del modulo e le loro dipendenze.

```
1  import { Module } from '@nestjs/common';
2  import { UsersService } from './users.service';
3  import { UsersController } from './users.controller';
4
5  @Module({
6    providers: [UsersService],
7    controllers: [UsersController],
8  })
9  export class UsersModule {}
```

I vantaggi dell'utilizzo del *Module Pattern* includono:

- **Organizzazione del Codice**: aiuta a mantenere il codice ben organizzato, rendendo più facile trovare e gestire le parti dell'applicazione.
- **Manutenibilità**: ogni modulo può essere sviluppato e manutenuto indipendentemente dagli altri.
- **Riusabilità**: possono essere riutilizzati in altre applicazioni o parti della stessa applicazione.

- **Scalabilità**: facilitando la gestione delle dipendenze e la composizione dei moduli, il Module Pattern supporta la scalabilità dell'applicazione.

4.4.3 Controller-Service Pattern

NestJS separa la logica di presentazione dalla logica di *business*.
I *controller* sono responsabili di gestire le richieste HTTP, mappando le richieste ai metodi appropriati e restituendo le risposte agli utenti. Fungono da intermediari tra il *client* (*frontend*) e la logica di *business* dell'applicazione. Gestiscono la validazione dei parametri in ingresso e la formattazione delle risposte rilasciate.
I servizi contengono la logica di *business* dell'applicazione. Sono responsabili dell'interazione con il *database*, la gestione dei dati e l'implementazione delle regole di *business*. Forniscono metodi che possono essere chiamati dai *controller* o da altri servizi. Contengono operazioni CRUD, manipolazione dei dati e comunicazione con altre parti dell'applicazione.
Questo *pattern* promuove una chiara separazione delle responabilità.

```
1  import { Controller, Get } from '@nestjs/common';
2  import { UsersService } from './users.service';
3
4  @Controller('users')
5  export class UsersController {
6    constructor(private readonly usersService: UsersService) {}
7
8    @Get()
9    findAll() {
10   return this.usersService.findAll();
11   }
12 }
```

I vantaggi nell'utilizzo del *Controller-Service Pattern* includono:

- **Separazione delle Responsabilità**: separando la logica di presentazione (*controller*) dalla logica di *business (service)*, il codice diventa più modulare e facile da gestire.

- **Manutenibilità**: le modifiche alla logica di *business* non influenzano direttamente la logica di presentazione e viceversa, rendendo il codice più manutenibile.
- **Testabilità**: i servizi possono essere facilmente testati in isolamento senza la necessità di simulare richieste HTTP, migliorando la copertura dei test.
- **Riusabilità**: la logica di *business* incapsulata nei servizi può essere riutilizzata in diverse parti dell'applicazione o in altre applicazioni.

4.4.4 Decorator Pattern

Il Decorator Pattern è ampiamente utilizzato in NestJS per definire metadati e configurare i componenti dell'applicazione. I decoratori in NestJS sono funzioni che possono essere applicate a classi, metodi, proprietà o parametri per arricchirli con comportamenti aggiuntivi o configurazioni specifiche. Questo *pattern* permette di aggiungere funzionalità in modo dichiarativo e modulare. I decoratori più utilizzati nell'applicativo sono:

- ***Module***: definisce una classe come modulo NestJS, aggregando *controller, provider*, e altri moduli.
- ***Controller***: definisce una classe come *controller* che gestisce le richieste HTTP per un determinato percorso.
- ***Provider***: marca una classe come un *provider* (Injectable) che può essere iniettato tramite il sistema di Dependency Injection di NestJS.
- **Gestione delle Rotte HTTP**: associa un metodo del *controller* a una specifica rotta HTTP.
- **Intercettori**: applica uno o più *interceptor* a un metodo specifico.
- **Decoratori per Parametri della Rotta**: estrae parametri dalla rotta e li passa come argomenti al metodo del *controller*.
- **Decoratori per il Corpo della Richiesta**: estrae il corpo della richiesta e lo passa come argomento al metodo del *controller*.

4.4.5 Interceptor Pattern

Gli *interceptor* in NestJS permettono di gestire il comportamento delle richieste e delle risposte in modo centralizzato. Possono modificare, trasformare, o loggare i dati prima che vengano inviati al *client* o al *server*. In particolare ne è stato fatto uso per la gestione del caricamento delle immagini relative ai ristoranti ed ai piatti registrati dai ristoratori.

```
1  @UseInterceptors(FileFieldsInterceptor([
2    { name: 'logo', maxCount: 1 },
3    { name: 'banner_image', maxCount: 1 },
4    ]))
5    async create(
6    @UploadedFiles() files: { logo?: Express.Multer.File[],
     banner_image?: Express.Multer.File[] }
7    ) {
8    // Implementation
9    }
```

5 Requisiti

5.1 Funzionali

Di seguito viene riportata la specifica relativa ai requisiti funzionali, che delineano le funzionalità del Sistema, le azioni eseguibili da parte del Sistema e le informazioni che il Sistema può fornire. La presenza di ogni requisito viene giustificata riportando la fonte, che può essere un UC oppure presente nel testo del capitolato d'appalto. Mentre i codici univoci sottostanti indicano:

1. RFO: Requisito Funzionale Obbligatorio;
2. RFF: Requisito Funzionale Facoltativo;
3. RFD: Requisito Funzionale Desiderabile.

ID	Descrizione	Stato
RFO1	L'Utente generico e L'Utente base devono poter visualizzare l'elenco dei ristoranti disponibili.	Soddisfatto
RFO2	L'Utente generico e L'Utente base devono poter ricercare un ristorante attraverso il nome.	Soddisfatto
RFO3	L'Utente generico e L'Utente base devono poter visualizzare un ristorante.	Soddisfatto
RFD4	L'Utente generico e L'Utente base devono poter condividere un *link* di un ristorante.	Soddisfatto
RFD5	L'Utente generico e L'Utente base devono poter visualizzare la pagina delle *FAQ*^{G} .	Non soddisfatto
RFO6	L'Utente generico deve poter effettuare l'accesso al Sistema.	Soddisfatto
RFO7	L'Utente generico deve poter effettuare la registrazione al Sistema come Utente base o Utente ristoratore.	Soddisfatto
RFO8	L'Utente generico deve visualizzare un messaggio d'errore se l'accesso fallisce.	Soddisfatto
RFO9	L'Utente generico deve visualizzare un messaggio d'errore se la registrazione fallisce.	Soddisfatto
RFD10	L'Utente base deve poter visualizzare i suoi dati utente.	Non soddisfatto
RFD11	L'Utente base deve poter modificare i suoi dati utente.	Non soddisfatto
RFD12	L'Utente base deve poter visualizzare lo storico dei suoi ordini.	Non soddisfatto
RFO13	L'Utente base deve poter visualizzare la lista delle sue prenotazioni, ed in caso andare in dettaglio.	Soddisfatto
RFO14	L'Utente base deve poter visualizzare la notifica dello stato della sua prenotazione.	Soddisfatto
RFD15	L'Utente base deve poter elimare il proprio *account*.	Non soddisfatto
RFO16	L'Utente base deve poter prenotare un tavolo.	Soddisfatto
RFO17	L'Utente base deve poter condividere la prenotazione.	Soddisfatto
RFO18	L'Utente base deve poter annullare la prenotazione.	Soddisfatto

ID	Descrizione	Stato
RFO19	L'Utente base deve poter accedere ad una prenotazione a lui condivisa.	Soddisfatto
RFO20	L'Utente base deve poter annullare il proprio ordine.	Soddisfatto
RFO21	L'Utente base deve poter creare un ordinazione collaborativa dei pasti.	Soddisfatto
RFO22	L'Utente base deve poter dividere il conto in maniera equa oppure proporzionale.	Soddisfatto
RFD23	L'Utente base deve poter visualizzare il messaggio d'errore che la divisione del conto è stata già effettuata.	Non soddisfatto
RFF24	L'Utente base deve poter pagare il conto.	Soddisfatto
RFF25	L'Utente base deve poter visualizzare l'errore relativo al pagamento fallito.	Soddisfatto
RFD26	L'Utente base deve poter inserire *feedback*.	Soddisfatto
RFD27	L'Utente base deve poter visualizzare la notifica di richiesta di inserimento *feedback*.	Soddisfatto
RFD28	L'Utente base deve poter visualizzare la notifica relativa alla modifica della sua ordinazione.	Soddisfatto
RFF29	L'Utente base deve poter visualizzare la notifica relativa al suo *feedback* che ha ricevuto una risposta.	Non soddisfatto
RFD30	L'Utente base deve poter inserire e modificare le proprie allergie.	Non soddisfatto
RFD31	L'Utente base deve poter visualizzare un messaggio se seleziona un piatto di cui è allergico.	Non soddisfatto
RFO32	L'Utente base deve poter visualizzare il menù di un ristorante.	Soddisfatto
RFO33	L'Utente autenticato deve poter effettuare il *logout*.	Soddisfatto
RFD34	L'Utente autenticato deve poter comunicare attraverso la *chat*.	Non soddisfatto
RFD35	L'Utente autenticato deve poter visualizzare la notifica relativa all'arrivo di un nuovo messaggio in *chat*.	Non soddisfatto

ID	Descrizione	Stato
RFO36	L'Utente ristoratore deve poter visualizzare la notifica relativa ad una nuova prenotazione.	Soddisfatto
RFO37	L'Utente ristoratore deve poter visualizzare la notifica relativa ad un nuovo ordine.	Soddisfatto
RFO38	L'Utente ristoratore deve poter visualizzare la notifica relativa all'avvenuto pagamento.	Soddisfatto
RFD39	L'Utente ristoratore deve poter visualizzare la notifica relativa all'inserimento di un *feedback*.	Soddisfatto
RFO40	L'Utente ristoratore deve poter visualizzare la lista delle prenotazioni	Soddisfatto
RFO41	L'Utente ristoratore deve poter accettare una prenotazione.	Soddisfatto
RFO42	L'Utente ristoratore deve poter rifiutare una prenotazione.	Soddisfatto
RFO43	L'Utente ristoratore deve poter terminare una prenotazione.	Soddisfatto
RFO44	L'Utente ristoratore deve poter visualizzare la lista delle ordinazioni.	Soddisfatto
RFD45	L'Utente ristoratore deve poter modificare un ordinazione.	Non soddisfatto
RFO46	L'Utente ristoratore deve poter visualizzare lo stato di pagamento di una prenotazione.	Soddisfatto
RFD47	L'Utente ristoratore deve poter visualizzare la lista dei *feedback*.	Soddisfatto
RFD48	L'Utente ristoratore deve poter segnalare un *feedback*.	Non soddisfatto
RFD49	L'Utente ristoratore deve poter rispondere ad un *feedback*.	Non soddisfatto
RFD50	L'Utente ristoratore deve poter modificare le informazioni del suo ristorante.	Non soddisfatto
RFO51	L'Utente ristoratore deve poter gestire il menù, inserendo, eliminando e modificando dei piatti.	Soddisfatto
RFO52	L'Utente ristoratore deve poter gestire gli ingredienti, inserendo e eliminando degli ingredienti.	Soddisfatto
RFO53	L'Utente ristoratore deve poter assegnare gli ingredienti ad un piatto.	Soddisfatto

ID	Descrizione	Stato
RFO54	L'Utente ristoratore deve poter visualizzare la notifica relativa all'annullamento di un ordinazione.	Soddisfatto
RFO55	L'Utente ristoratore deve poter visualizzare la notifica relativa all'annullamento di una prenotazione.	Soddisfatto
RFF56	L'Utente generico deve poter effettuare l'accesso al Sistema attraverso un sistema di terze parti.	Non soddisfatto
RFD57	L'Utente generico e L'Utente base devono poter ricercare un ristorante attraverso luogo e filtri.	Non soddisfatto
RFO58	L'Utente ristoratore deve poter visualizzare la lista degli ingredienti necessari per soddisfare gli ordini di una giornata.	Soddisfatto
RFO59	L'Utente base deve poter modificare la propria ordinazione.	Soddisfatto
RFO60	L'Utente base deve poter togliere qualche ingrediente da un piatto ordinato.	Soddisfatto

Tipologia	Totale	Soddisfatti
Obbligatori	36	36
Desiderabili	20	4
Facoltativi	4	2

Tabella 4: Tabella riassuntiva dei requisiti funzionali soddisfatti.