SWEnergy

project.swenergy@gmail.com

Specifiche tecniche

Descrizione: Descrizione delle scelte architetturali e di design del progetto

Stato	Non approvato
Data	30/03/2024
Redattori	Carlo Rosso
Verificatori	/
Approvatore	/
Destinatari	prof. Tullio Vardanega
	prof. Riccardo Cardin
Versione	0.2.1

Registro delle modifiche

Versione	Data	Redattore	Verificatore	Approvatore	Modifiche
0.2.1	2024-05-16	Carlo Rosso	/	/	Conclusione della descrizione dei pattern usati 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 (ultimo accesso 25/03/2024). 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 4 sezioni principali:

- **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 di deployment**: illustra l'architettura di *deployment* e ne motiva la scelta;
- **Architettura implementativa**: espone le scelte implementative adottate per soddisfare i requisiti richiesti. Inoltre, sono spiegati i *desing pattern* utilizzati e le motivazioni che hanno portato alla loro adozione;
- **Requisiti**: sono riportati i requisiti concordati con il proponente e descritti in dettaglio nel documento "Analisi dei Requisiti", nella versione 2.0.0. 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 sarà 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";
- Documento del capitolato d'appalto C3 - *Easy Meal* (ultimo accesso 20/03/2024);
- Regolamento del progetto (ultimo accesso 25/03/2024);

1.5.2 Informativi

- Glossario v2.0.0;
- Analisi dei Requisiti v2.0.0;
- Linguaggio UML - Diagramma delle classi (ultimo accesso 4/03/2024);

1.5.3 Tecnologici

- Typescript (ultimo accesso 03/04/2024);
- Angular (ultimo accesso 03/04/2024);
- Axios (ultimo accesso 03/04/2024);
- NestJS (ultimo accesso 03/04/2024);
- Drizzle (ultimo accesso 03/04/2024);
- Docker (ultimo accesso 03/04/2024);
- Insomnia (ultimo accesso 03/04/2024);
- Tailwind CSS (ultimo accesso 03/04/2024);

2 Tecnologie adottate

2.1 Tecnologie per la codifica

2.1.1 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 committente 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 necessario 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.

2.1.2 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 è scarsamente supportata da ChatGPT e dispone di risorse online meno dettagliate rispetto alle versioni precedenti.
Consideriamo questo aspetto come un punto positivo: ci consente di sfruttare le intelligenze artificiali per comprendere i metodi di soluzione più appropriati, mentre richiede che il codice venga sempre adattato alle nostre specifiche esigenze, promuovendo così una migliore comprensione del *framework*. Inoltre, la documentazione fornita da Angular è chiara e completa, agevolando il nostro processo di apprendimento del *framework*.

2.1.3 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 proponente, che ha sottolineato la facilità e la velocità con cui è possibile effettuare *test* di carico sulle API utilizzando questo strumento.

2.1.4 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*.

2.1.5 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 che abbiamo scelto TypeScript come linguaggio principale. Node.js rappresenta la scelta più diffusa e supportata per questo tipo di scenario.

2.1.6 Nest.js

Per la gestione delle API, 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 proponente, 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.

2.1.7 Drizzle

Per la gestione del *database*, SWEnergy ha scelto Drizzle, un ORM (*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.

2.1.8 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.

2.1.9 Docker

L'adozione di Docker per containerizzare l'intero progetto 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 standardizzazione 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 per containerizzare il nostro progetto offre un modo efficiente e affidabile per gestire l'intero ciclo di vita dell'applicazione, garantendo flessibilità, portabilità e scalabilità.

2.2 Tecnologie per l'analisi statica del codice

2.2.1 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.

2.3 Tecnologie per l'analisi dinamica del codice

2.3.1 Insomnia

Per verificare il funzionamento delle API e del *backend*, SWEnergy ha optato per l'utilizzo di Insomnia, un'applicazione *open-source* progettata per testare le API REST. In effetti, il *team* ha sviluppato un insieme di test per assicurarsi che le API rispondano correttamente alle richieste e restituiscano i dati previsti, garantendo così il corretto funzionamento del sistema.

2.3.2 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.

2.3.3 Nest.js

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

3 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.

3.1 Architettura di Deployment

SWEnergy ha deciso di adottare un'architettura a tre livelli (*N-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.
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 ciascuno 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 delle semplici operazioni CRUD, che non richiedono un'interazione complessa tra i diversi livelli dell'applicazione.

3.2 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.

3.2.1 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 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*.

3.2.2 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.

3.3 *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.

3.3.1 GenericService

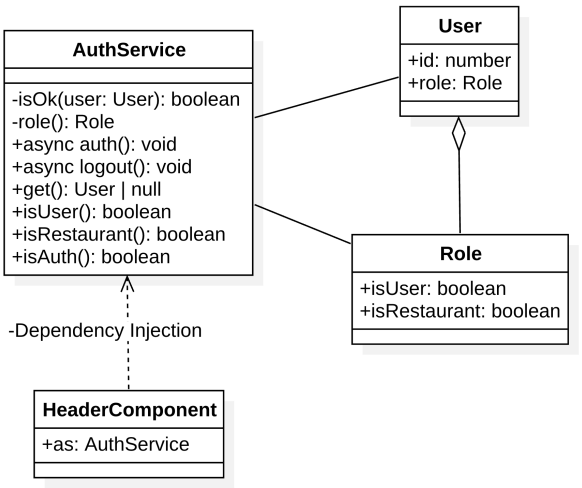

Figura 1: Diagramma delle classi per AuthService

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.

3.3.2 MessageService

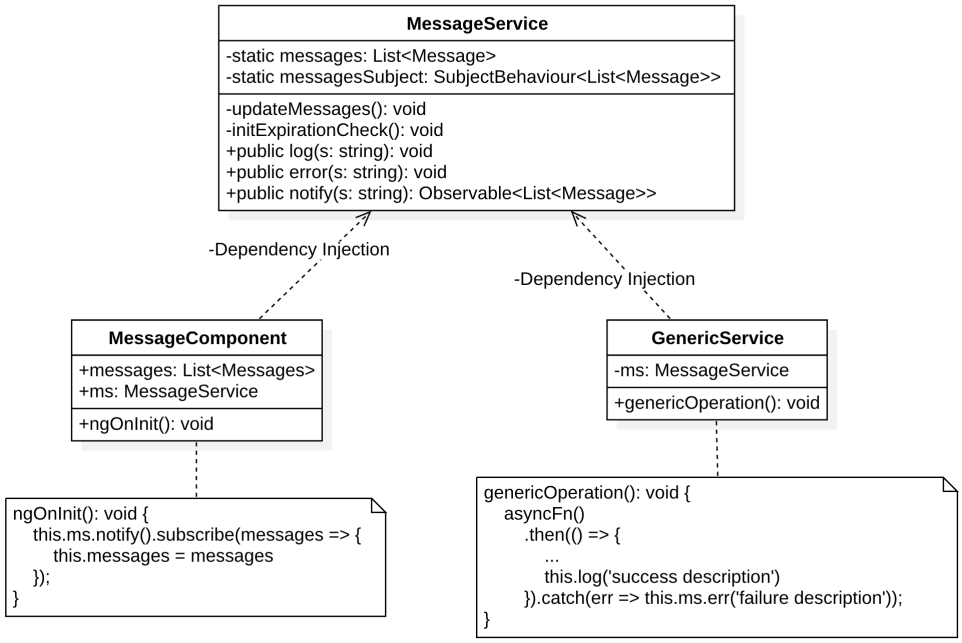

Figura 2: Diagramma delle classi per il MessageService

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 Requisiti

4.1 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

ID	Descrizione	Stato
RFD4	L'Utente generico e L'Utente base devono poter condividere un *link* di un ristorante.	Non 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.	Non soddisfatto
RFO14	L'Utente base deve poter visualizzare la notifica dello stato della sua prenotazione.	Non soddisfatto
RFD15	L'Utente base deve poter elimare il proprio *account*.	Non soddisfatto
RFO16	L'Utente base deve poter prenotare un tavolo.	Non soddisfatto
RFO17	L'Utente base deve poter condividere la prenotazione.	Non soddisfatto
RFO18	L'Utente base deve poter annullare la prenotazione.	Non soddisfatto
RFO19	L'Utente base deve poter accedere ad una prenotazione a lui condivisa.	Non 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

ID	Descrizione	Stato
RFO22	L'Utente base deve poter dividere il conto in maniera equa oppure proporzionale.	Non 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.	Non soddisfatto
RFF25	L'Utente base deve poter visualizzare l'errore relativo al pagamento fallito.	Non soddisfatto
RFD26	L'Utente base deve poter inserire *feedback*.	Soddisfatto
RFD27	L'Utente base deve poter visualizzare la notifica di richiesta di inserimento *feedback*.	Non soddisfatto
RFD28	L'Utente base deve poter visualizzare la notifica relativa alla modifica della sua ordinazione.	Non 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
RFO34	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
RFO36	L'Utente ristoratore deve poter visualizzare la notifica relativa ad una nuova prenotazione.	Non soddisfatto
RFO37	L'Utente ristoratore deve poter visualizzare la notifica relativa ad un nuovo ordine.	Non soddisfatto

ID	Descrizione	Stato
RFO38	L'Utente ristoratore deve poter visualizzare la notifica relativa all'avvenuto pagamento.	Non soddisfatto
RFD39	L'Utente ristoratore deve poter visualizzare la notifica relativa all'inserimento di un *feedback*.	Non soddisfatto
RFO40	L'Utente ristoratore deve poter visualizzare la lista delle prenotazioni	Non soddisfatto
RFO41	L'Utente ristoratore deve poter accettare una prenotazione.	Non soddisfatto
RFO42	L'Utente ristoratore deve poter rifiutare una prenotazione.	Non soddisfatto
RFO43	L'Utente ristoratore deve poter terminare una prenotazione.	Non soddisfatto
RFO44	L'Utente ristoratore deve poter visualizzare la lista delle ordinazioni.	Non 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.	Non soddisfatto
RFD47	L'Utente ristoratore deve poter visualizzare la lista dei *feedback*.	Non 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.	Non soddisfatto
RFO54	L'Utente ristoratore deve poter visualizzare la notifica relativa all'annullamento di un ordinazione.	Non soddisfatto

ID	Descrizione	Stato
RFO55	L'Utente ristoratore deve poter visualizzare la notifica relativa all'annullamento di una prenotazione.	Non 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.	Non 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