

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E
TECNOLOGIE DELL'INFORMAZIONE



CORSO DI LAUREA IN INFORMATICA INSEGNAMENTO DI
INGEGNERIA DEL SOFTWARE ANNO ACCADEMICO 2022/2023

Specifiche, progettazione,
implementazione e validazione del
Sistema Informativo “Ratatouille23”

Autori

Mario De Luca N86/3911
Alessandro Bonomo N86/3852

Indice

I Documento dei Requisiti Software	4
1 Analisi dei requisiti	4
1.1 Modellazione dei casi d'uso richiesti	4
1.2 Individuazione del target degli utenti	5
1.3 Prototipazione visuale via Mock-up dell'interfaccia utente per tutti i casi d'uso	8
1.4 Tabelle di Cockburn	10
1.4.1 Crea nuovo ristorante	10
1.4.2 Crea nuova categoria	12
1.5 Mock-up dei casi d'uso descritti nelle tabelle di Cockburn	13
1.6 Valutazione dell'usabilità a priori	14
1.6.1 Testing del prototipo con Figma	14
1.6.2 Analisi dei risultati	15
1.6.3 Correzioni in base al feedback dei tester	16
1.7 Glossario	18
2 Specifica dei Requisiti	19
2.1 Classi, oggetti e relazioni di analisi	19
2.1.1 Entities	19
2.1.2 Boundaries	19
2.1.3 Cucina	19
2.1.4 Dashboard iniziale & Dashboard Ristorante	19
2.1.5 Gestisci categorie & Gestisci Conti	20
2.1.6 Gestisci elementi categoria	20
2.1.7 Inserimento elementi ordinazione	20
2.1.8 Login	21
2.1.9 Pagina Utenze	21
2.1.10 Prendi Ordinazione & Statistiche	21
2.1.11 Controller	22
2.2 Diagrammi di sequenza di analisi	23
2.2.1 Crea utenza	23
2.2.2 Visualizza e evadi ordini	23
2.3 Prototipazione funzionale via statechart	24
2.3.1 Statechart - Crea un Ristorante	24
2.3.2 Statechart - Crea una Categoria	26
II Documento di Design del Sistema	26
3 Analisi dell'architettura	26
3.1 Architettura 3 Tier	26
3.1.1 Client - Tier 1	26
3.1.2 Server - Tier 2	27
3.1.3 Database - Tier 3	27
3.1.4 Schema architettura	27
3.1.5 Docker containers in azione	27
3.2 Documentazione del backend	27
3.2.1 Tutte le routes del backend	28
3.2.2 Esempio di route	29
3.3 Documentazione del frontend	31
3.4 Cloud Hosting	32
3.5 Pannello di controllo in cloud	32
3.6 Servizi utilizzati in cloud	32
4 Motivazione delle scelte adottate	32

5 Diagramma delle classi di design	33
5.1 Le Entities	33
5.1.1 Utilizzo del design pattern Factory Method	33
5.2 I Boundaries	34
5.2.1 Cucina	34
5.2.2 Dashboard ristorante	34
5.2.3 Dashboard supervisore	34
5.2.4 Gestisci categorie	35
5.2.5 Gestisci conti	35
5.2.6 Gestisci elementi di una categoria	35
5.2.7 Gestisci utenza	35
5.2.8 Inserimento elementi di un'ordinazione	36
5.2.9 Login e registrazione	36
5.2.10 Ordina	36
5.2.11 Statistiche	37
5.2.12 Tutti i components	37
5.2.13 Interfacce e enums di utilità	37
5.3 Il Controller	38
5.4 DAOs	39
6 Diagrammi di sequenza di design	40
6.1 aggiungiRistorante	40
6.2 getContiUltime24h	41
III Testing e valutazione sul campo dell'usabilità	42
7 Codice xUnit Black Box per Unit Testing	42
7.1 Funzione addElemento	42
7.1.1 Strategia di testing Black Box	42
7.1.2 Codice xUnit Black Box	43
7.2 Funzione scambiaElementi	44
7.3 Strategia di testing Black Box	44
7.3.1 Codice xUnit Black Box	45
7.4 Funzione registraUtente	45
7.4.1 Strategia di testing Black Box	45
7.4.2 Codice xUnit Black Box	46
7.5 Funzione accediUtente	47
7.5.1 Strategia di testing Black Box	47
7.5.2 Codice xUnit Black Box	47
8 Valutazione dell'usabilità sul campo	48
8.1 Test di valutazione finale	48
8.2 Analisi mediante file di log	48
8.3 Integrazione con SDK di Firebase	49
8.4 Analisi dei dati ottenuti con Google Analytics	50
8.5 File di log	51

Abstract

Ratatouille23 è un sistema informativo sviluppato per supportare la gestione e l'operatività di attività di ristorazione. L'obiettivo del team di sviluppo è quello di creare un'applicazione performante e affidabile, dotata di funzionalità intuitive e rapide da utilizzare, al fine di garantire una piacevole esperienza d'uso agli utenti. Tra le principali funzionalità del sistema vi sono la possibilità per gli amministratori di creare utenze per i propri dipendenti, la personalizzazione del menù dell'attività di ristorazione, la registrazione delle ordinazioni e la visualizzazione in tempo reale degli ordini da parte degli addetti alla cucina, la generazione del conto per ogni tavolo

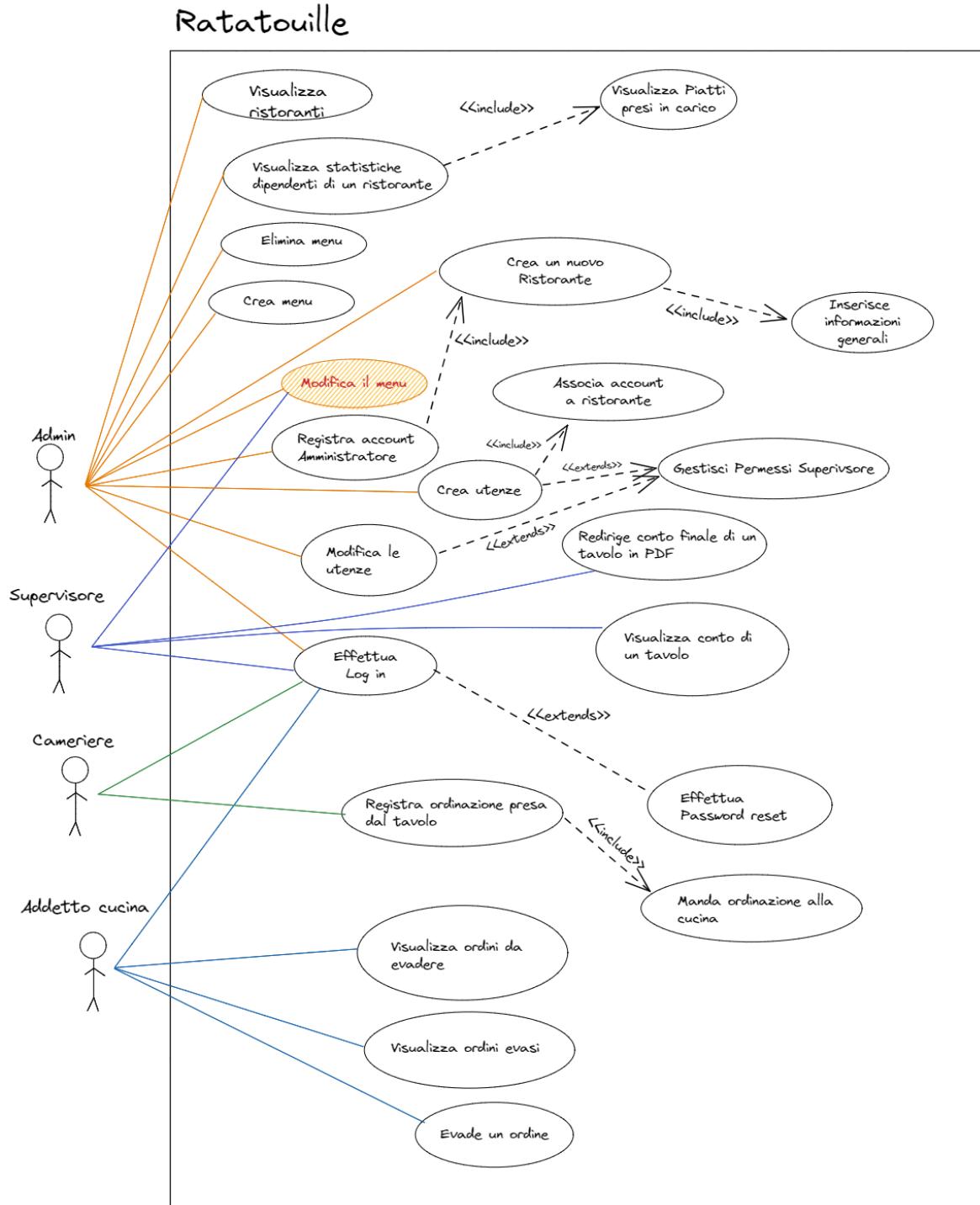
e la visualizzazione di statistiche dettagliate sulla produttività del personale addetto alla cucina. Ratatouille23 mira quindi a semplificare e ottimizzare le attività di gestione e operatività all'interno delle attività di ristorazione, fornendo agli utenti un'esperienza d'uso fluida e intuitiva.

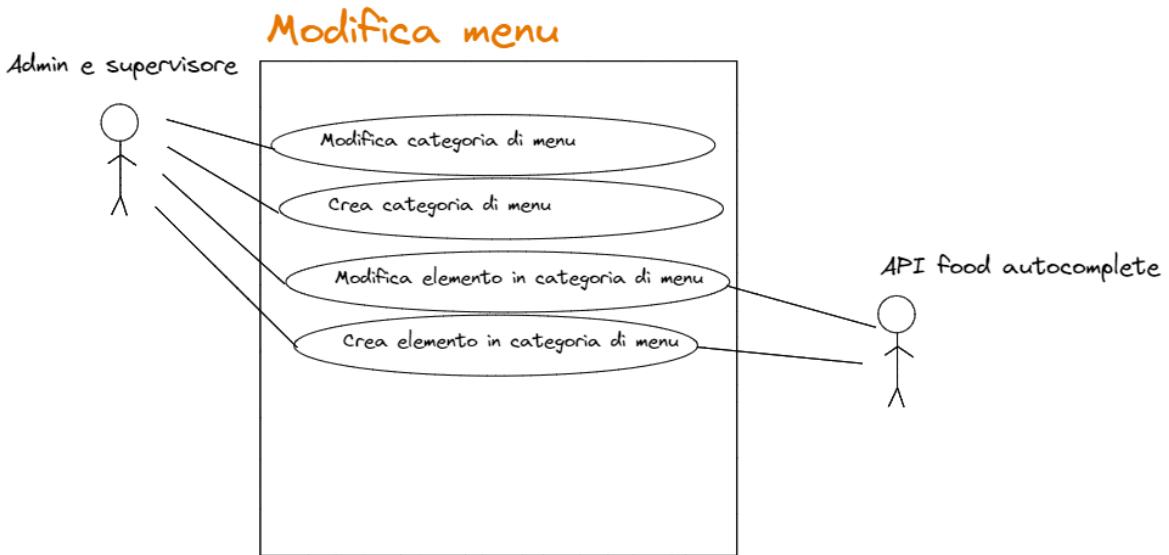
I Documento dei Requisiti Software

1 Analisi dei requisiti

1.1 Modellazione dei casi d'uso richiesti

Ecco il diagramma dei casi d'uso:

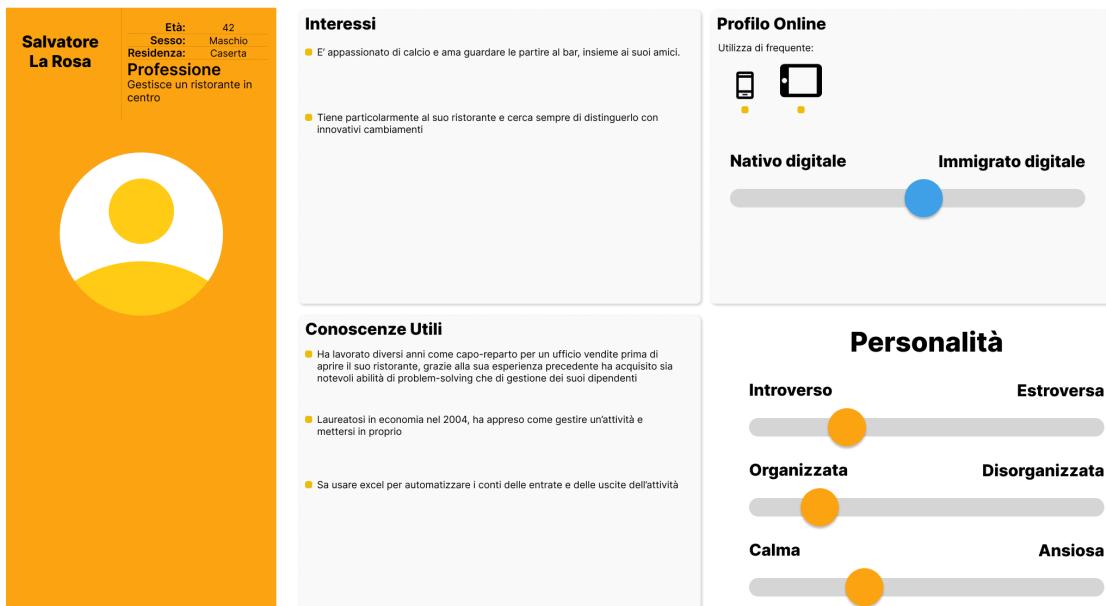




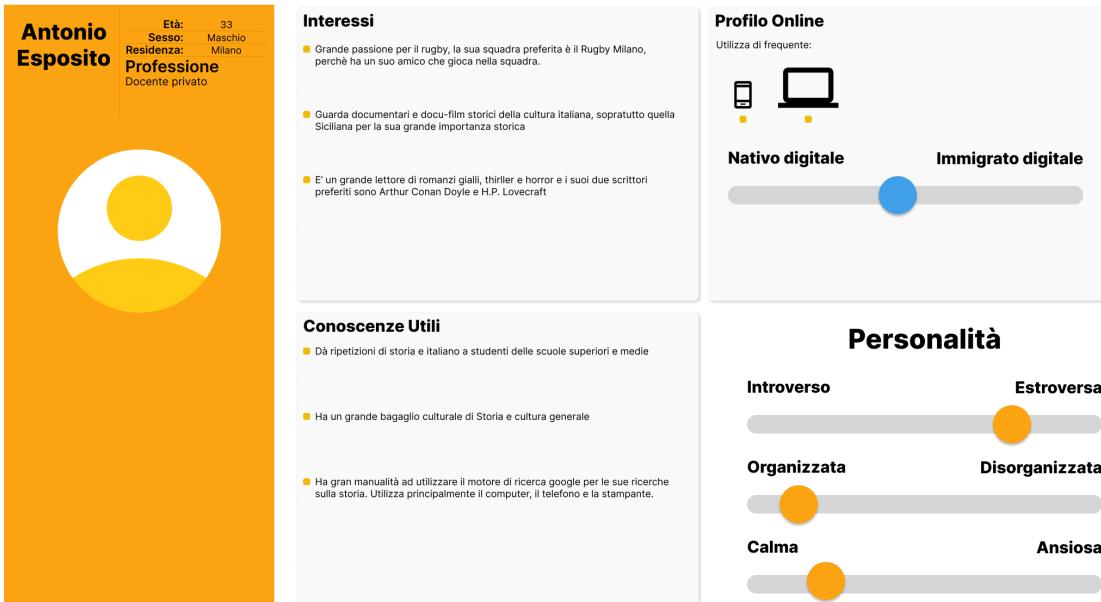
1.2 Individuazione del target degli utenti

La nostra applicazione è stata creata su misura per tutti gli utenti che lavorano nel ambiente della ristorazione. Facilita la gestione del ristorante all'amministratore, aiuta i camerieri a prendere le ordinazioni in maniera più agevole e ordinata. Infine, facilita le interazioni tra gli addetti alla cucina e il personale di sala. Abbiamo individuato 4 categorie di utenti:

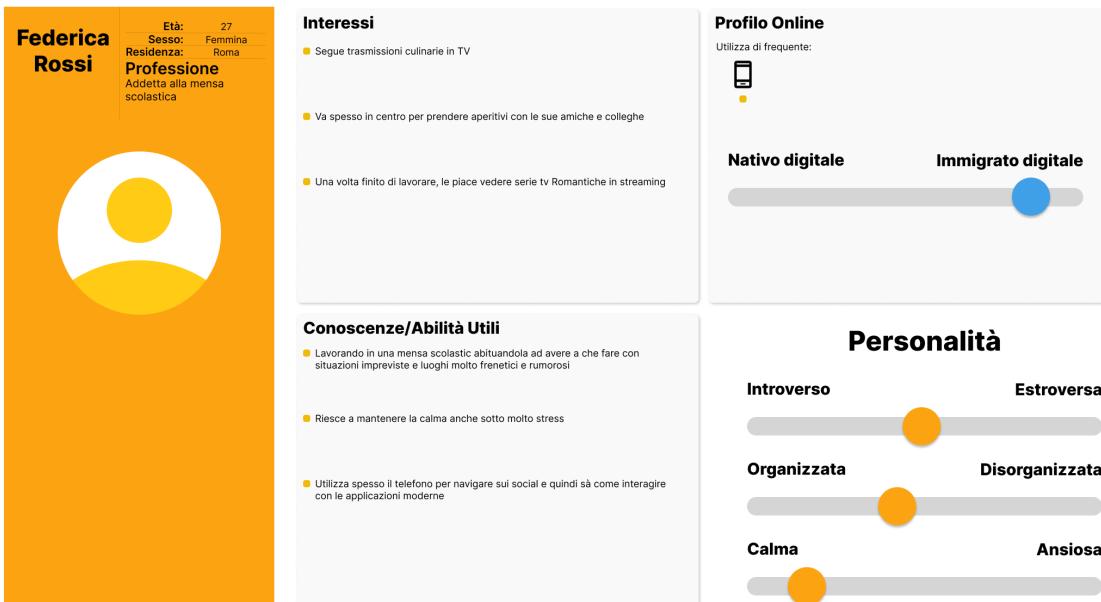
Amministratore Colui che gestisce i ristoranti e i suoi dipendenti, può gestire il menù di ogni singolo ristorante e avere un report sulle vendite di ogni locale.



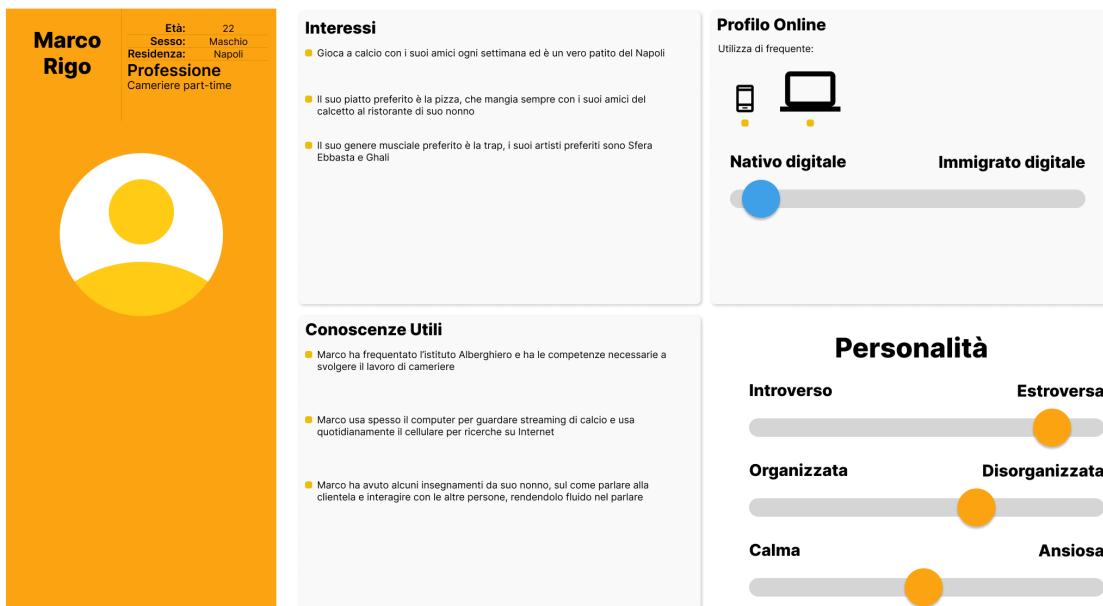
Supervisore E' un addetto alla cucina/sala, scelto dall'amministratore per supervisionare i suoi colleghi. Il supervisore può modificare il menu e gestire i conti dei tavoli.



Addetto alla cucina E' la persona che si occupa di leggere ed evadere gli ordini ricevuti dal cameriere.



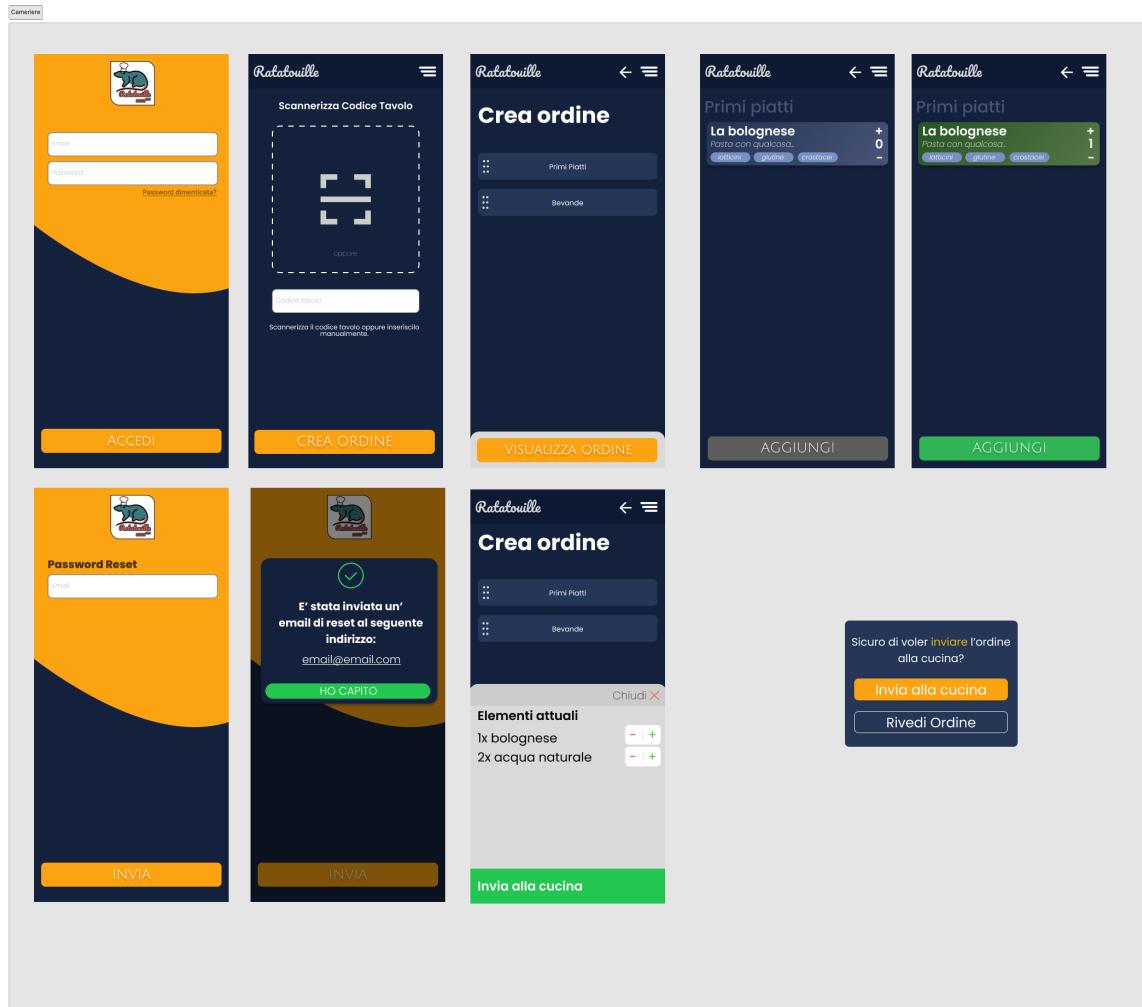
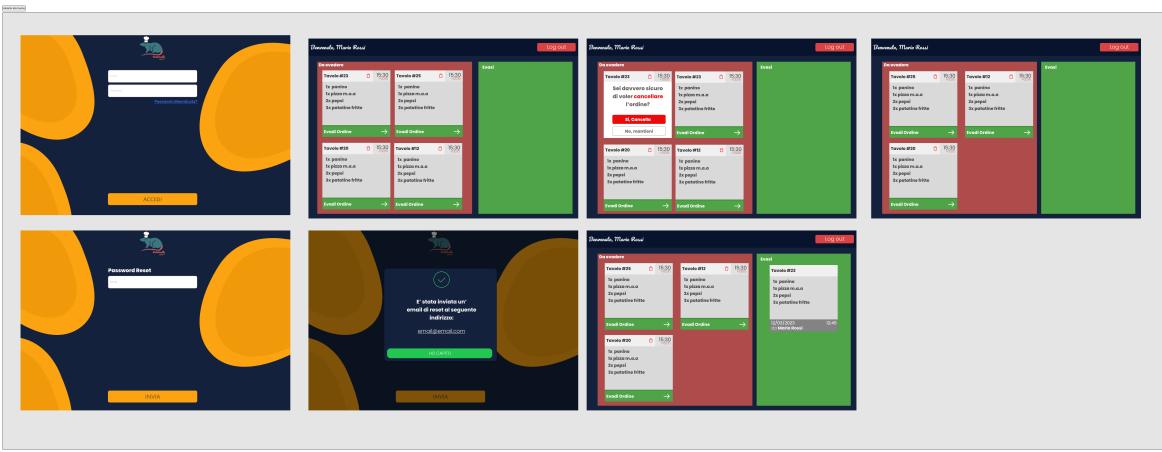
Addetto alla sala (Cameriere) E' colui che prende le ordinazioni e le invia alla cucina.



1.3 Prototipazione visuale via Mock-up dell'interfaccia utente per tutti i casi d'uso

Riportiamo di seguito tutti i mock-up dell'applicazione. I mock up sono stati realizzati con Figma. E' possibile visualizzare i mock up in dettaglio [cliccando qui](#) (Link al progetto Figma in cloud).





1.4 Tabelle di Cockburn

Di seguito riportiamo le tabelle di cockburn associate ai relativi casi d'uso:

- Crea nuovo ristorante
- Crea nuova categoria

1.4.1 Crea nuovo ristorante

Use Case #1	Crea nuovo ristorante		
Goal in Context	Creazione di un nuovo ristorante con le relative informazioni.		
Preconditions	L'utente deve essere autenticato come amministratore e trovarsi nella schermata Dashboard admin.		
Success End Conditions	Il ristorante viene aggiunto al sistema e viene mostrato nell'elenco dei ristoranti registrati.		
Failed End Conditions	L'utente clicca sull'icona di annullamento e ritorna alla schermata Dashboard admin.		
Primary Actor	Utente amministratore autenticato		
Trigger	L'utente clicca sul pulsante '+' nella schermata Dashboard admin		
Description	Step	User Action	System
	1	Clicca sul pulsante '+' nella schermata Dashboard admin	
	2		Mostra schermata Crea Ristorante
	3	Inserisce nome ristorante	
	4	Inserisce locazione ristorante	
	5	Inserisce numero di telefono ristorante	
	6	Clicca crea	
	7		Torna a Dashboard admin

Extension #1	Step	User Action	System
	1.1	Clicca sulla 'X'	
	1.2		Ritorna alla schermata Dashboard admin
Extension #2	Step	User Action	System
	6.1	Clicca su 'CREA' senza aver inserito il nome del ristorante	
	6.2	Mostra messaggio di errore	
Extension #3	Step	User Action	System
	6.1	Clicca su 'CREA' senza aver inserito la locazione del ristorante	
	6.2	Mostra messaggio di errore	
Extension #4	Step	User Action	System
	6.1	Clicca su 'CREA' senza aver inserito il numero di telefono del ristorante	
	6.2	Mostra messaggio di errore	
Subvariation #1	Step	User Action	System
	3.1	Inserisce l'url del sito web	
	3.2		Vai al punto 4
Subvariation #2	Step	User Action	System
	5.1	Seleziona immagine ristorante	
	5.2		Apre il browser del filesystem
	5.3	Seleziona l'immagine desiderata e conferma	
	5.4		Mostra preview immagine
	5.4		Vai al punto 6
Notes			

1.4.2 Crea nuova categoria

Use Case #2	Crea nuova categoria		
Goal in Context	Creazione di una nuova categoria del menu.		
Preconditions	L'utente deve essere autenticato come amministratore o supervisore e trovarsi nella schermata Gestione Menu.		
Success End Conditions	La categoria viene aggiunta al sistema e viene mostrata nel menu nella schermata Gestione Menu.		
Failed End Conditions	L'utente clicca sull'icona di annullamento e ritorna alla schermata Gestione Menu.		
Primary Actor	Utente amministratore o supervisore autenticato		
Trigger	L'utente clicca sul pulsante '+' nella schermata Gestione Menu		
Description	Step	User Action	System
	1	Clicca sul pulsante '+' nella schermata Gestione Menu	
	2		Mostra schermata Crea categoria
	3	Inserisce nome della categoria	
	4	Clicca crea	
	5		Torna a schermata Gestione Menu
Extension #1	Step	User Action	System
	1.1	Clicca sulla 'X'	
	1.2		Ritorna alla schermata Gestione Menu
Extension #2	Step	User Action	System
	4.1	Clicca su 'CREA' senza aver inserito il nome della categoria	
	4.2		Mostra messaggio di errore
Notes			

1.5 Mock-up dei casi d'uso descritti nelle tabelle di Cockburn

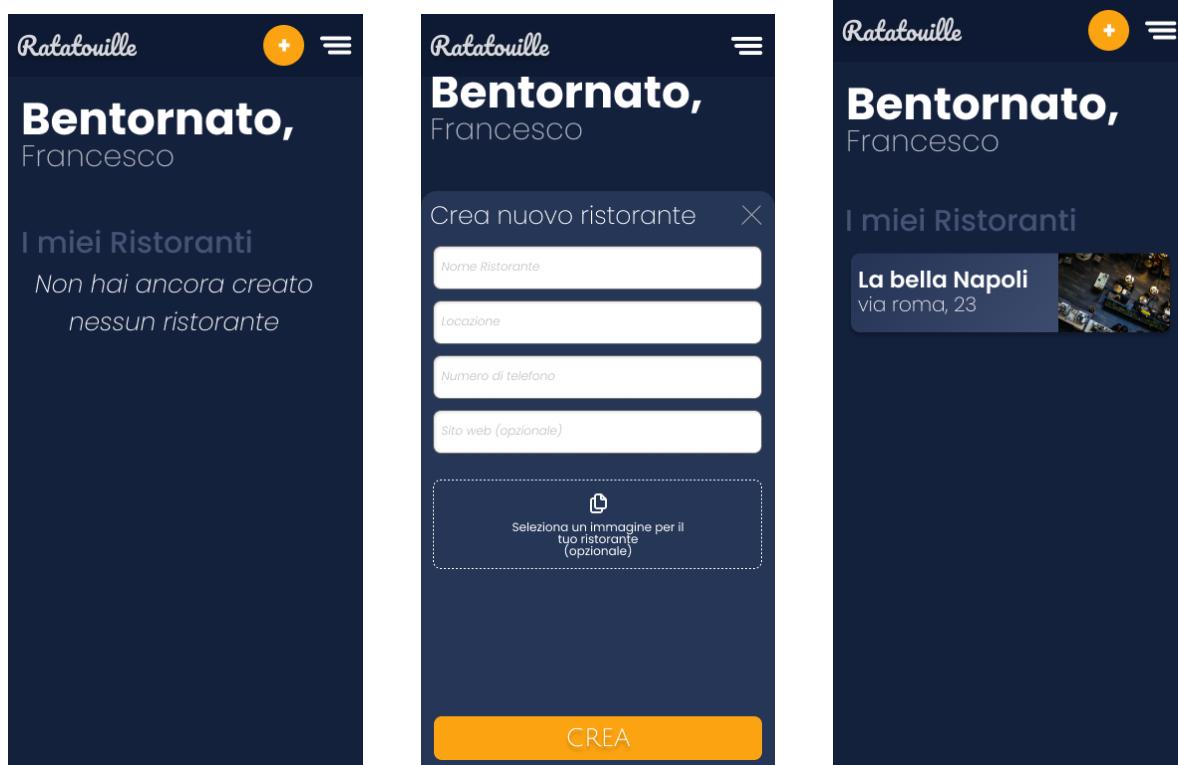




Figure 1: Mock-ups per Crea categoria

1.6 Valutazione dell’usabilità a priori

Prima di proseguire con lo sviluppo dell’applicazione, si è deciso di far effettuare ad un gruppo di utenti (tester) una valutazione del prototipo realizzato su Figma.

1.6.1 Testing del prototipo con Figma

Abbiamo dato ai tester una serie di Task da completare e abbiamo osservato le loro interazioni e difficoltà nello svolgere i compiti assegnati. Abbiamo poi compilato una tabella con gli esiti dei Task dati ai tester. Infine con una semplice formula siamo in grado di misurare la facilità di utilizzo della nostra applicazione.

Task analizzati:

1. Login
2. Registrazione
3. Crea ristorante
4. Crea ordine e invia alla cucina
5. Completa ed evadi ordine
6. Creazione di un menu con una portata
7. Stampa conto

1.6.2 Analisi dei risultati

	1	2	3	4	5	6	7
Tester 1	✓	✓	✓	✗	✓	トラック	✓
Tester 2	✓	✓	✓	トラック	✓	✓	✓
Tester 3	✓	✓	トラック	トラック	✓	✓	✓
Tester 4	✓	✓	✓	トラック	✓	✓	✓

Leggenda:

- ✓ Successo +1
- ✗ Fallimento -1
- トラック Successi Parziali +0.5

Ogni tester avrà il suo punteggio associato, che indica la sua facilità di esecuzione di quello specifico task.

$$tasks = 7$$

$$testers = 4$$

$$punteggioMassimo = (testers * tasks) = 28$$

$test_{i,n}$ = esito del test effettuato dal tester "i" nel task "n". Valori possibili: 0.5, 1, -1

$$punteggioSingoloTester_i = \sum_{n=1}^{tasks} test_{i,n}$$

$$punteggioTotale = \sum_{i=1}^{testers} punteggioSingoloTester_i$$

$$\text{Facilità d' uso } [1,-1] = \text{punteggioTotale}/\text{punteggioMassimo}$$

La facilità d'uso è un numero compreso tra 1 e -1. Se è negativa allora vuol dire che i fallimenti sono maggiori dei successi. Se è positiva, allora i successi sono maggiori dei fallimenti. Se è uguale a 0 allora i successi sono uguali ai fallimenti. Più il punteggio finale si avvicina ad 1, meglio è.

Il punteggio calcolato PRIMA delle correzioni è:

$$\text{punteggioTotale} = 4.5 + 6.5 + 6 + 6.5$$

$$\text{Facilità d' uso} = 23.5/28 = 0.83$$

Il punteggio calcolato DOPO le correzioni è:

$$\text{punteggioTotale} = 6.5 + 6.5 + 7 + 6.5$$

$$\text{Facilità d' uso} = 26.5/28 = 0.94$$

	1	2	3	4	5	6	7
Tester 1	✓	✓	トラック	✓	✓	✓	✓
Tester 2	✓	✓	✓	トラック	✓	✓	✓
Tester 3	✓	✓	✓	✓	✓	✓	✓
Tester 4	✓	✓	✓	✓	✓	トラック	✓

Figure 2: Testing dopo le correzioni

1.6.3 Correzioni in base al feedback dei tester

Grazie al feedback dei nostri tester siamo riusciti ad individuare 2 problematiche che impattavano sull'usabilità del prodotto.

Correzione logo Grazie al suggerimento di uno dei nostri tester abbiamo deciso di modificare leggermente il logo dell'applicazione per renderlo più affine al contesto culinario.



Figure 3: Correzione del logo. A sinistra la vecchia versione, a destra, la nuova

Correzione Task N°4 (Crea ordine) Una difficoltà comune a tutti i tester era riuscire a trovare l'interazione per riuscire a visualizzare e inviare l'ordine alla cucina. Abbiamo così deciso di modificare i Mock up per aumentare l'usabilità dell'applicazione, rendendo più visibile e intuitivo il pulsante per visualizzare l'ordine in corso.

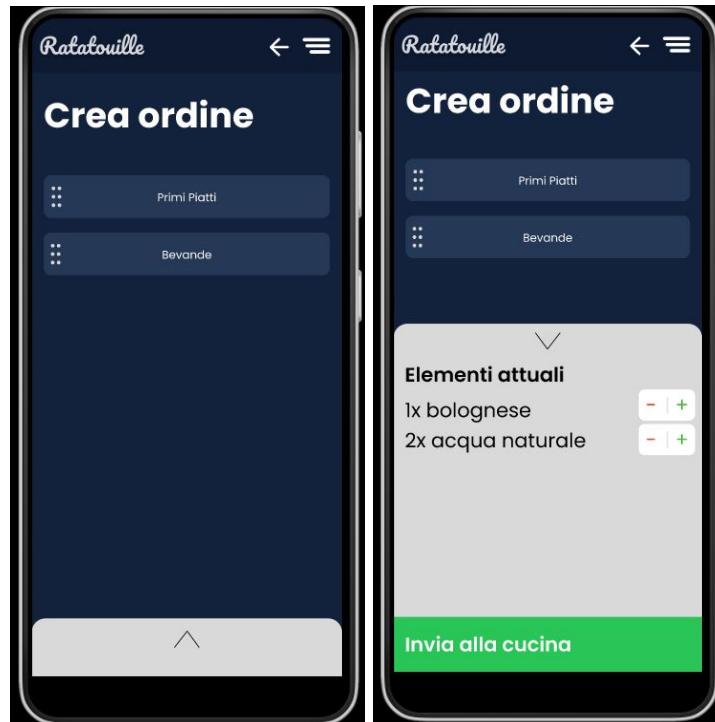


Figure 4: Prima della correzione

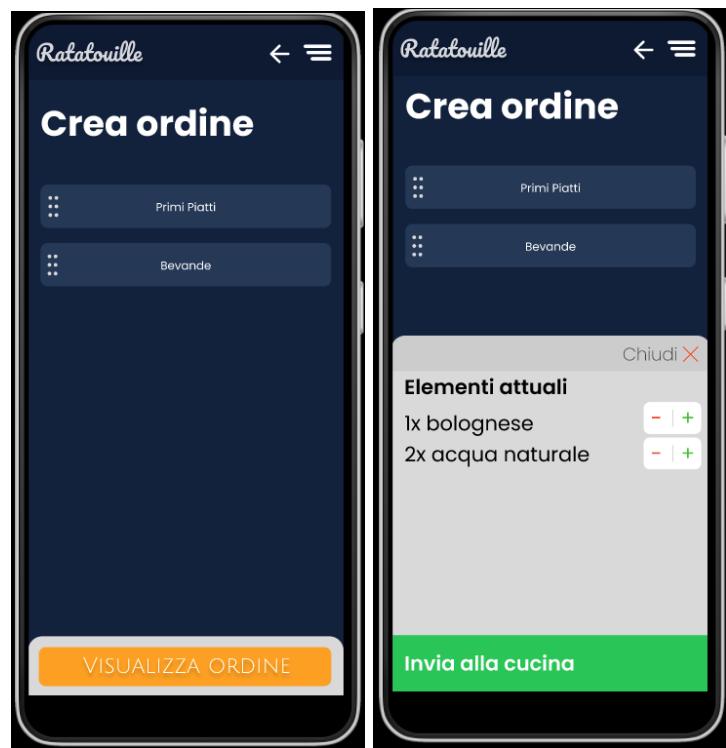


Figure 5: Dopo la correzione

1.7 Glossario

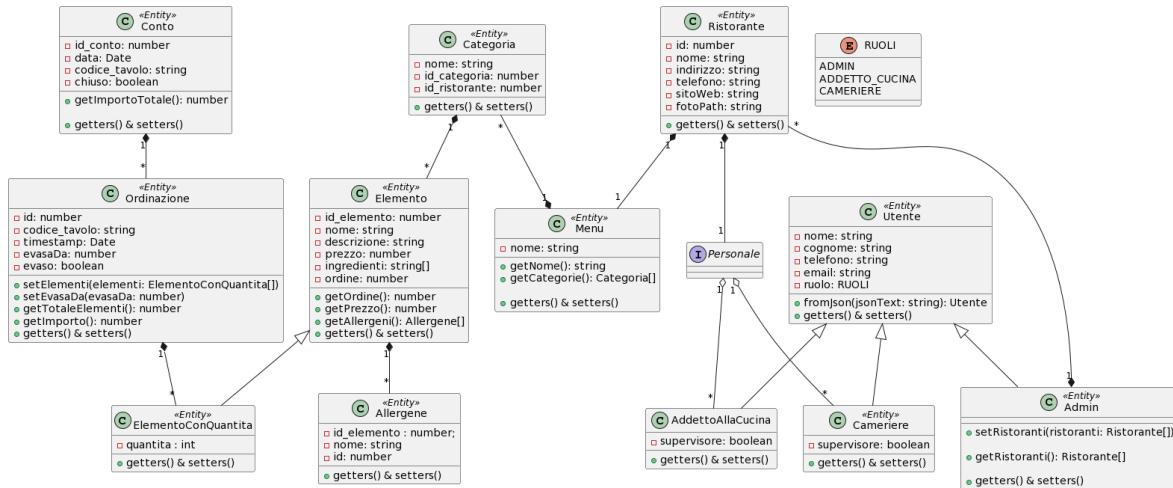
Raccolta dei termini utilizzati all'interno della documentazione:

Termine	Descrizione
Admin / Amministratore	Colui che crea le utenze, visiona le statistiche del personale, crea il menu e amministra i ristoranti
Supervisore	Utente con permessi superiori a quelli dell'utente base ma inferiori all'admin. Può modificare categorie e elementi del menù, stampare il conto e gestire i tavoli e le ordinazioni del locale
Figma	Figma è un software di progettazione grafica basato su cloud, utilizzato principalmente per la creazione di interfacce utente, web design e design di prodotto.
Immigrato digitale	Un immigrato digitale è una persona che ha difficoltà nell'utilizzo della tecnologia digitale e delle piattaforme online.

2 Specifica dei Requisiti

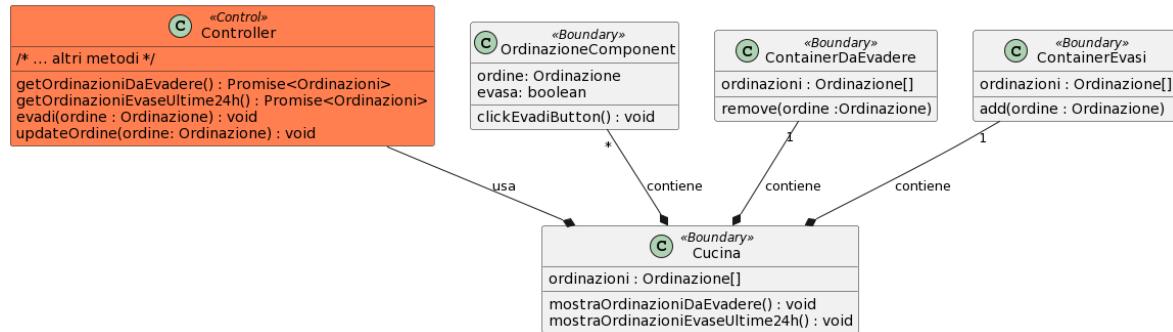
2.1 Classi, oggetti e relazioni di analisi

2.1.1 Entities

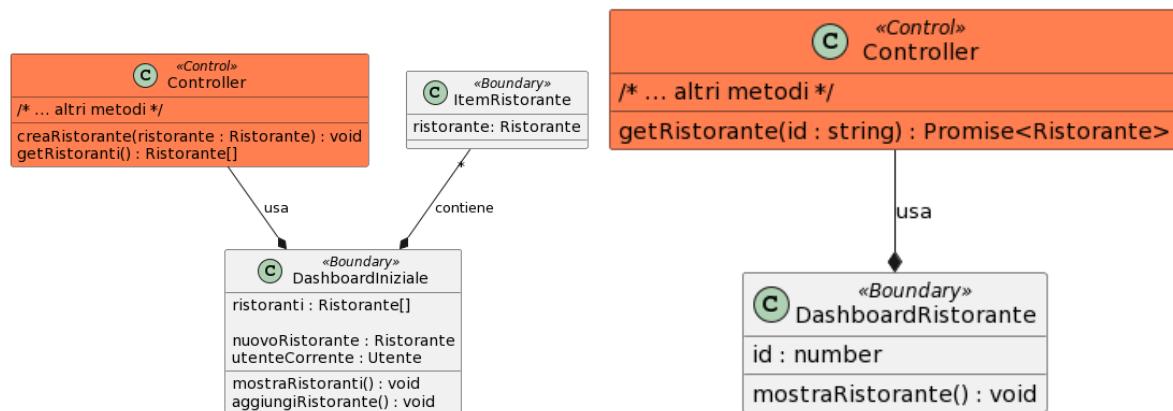


2.1.2 Boundaries

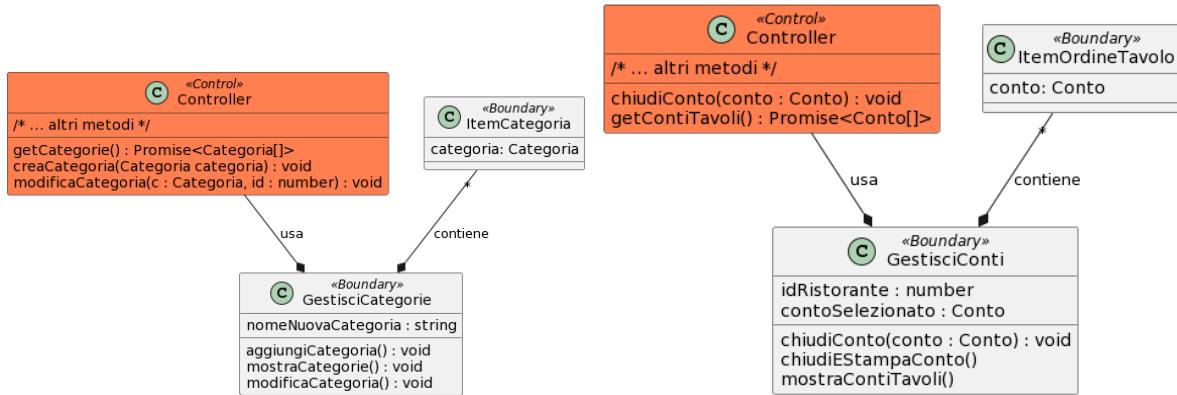
2.1.3 Cucina



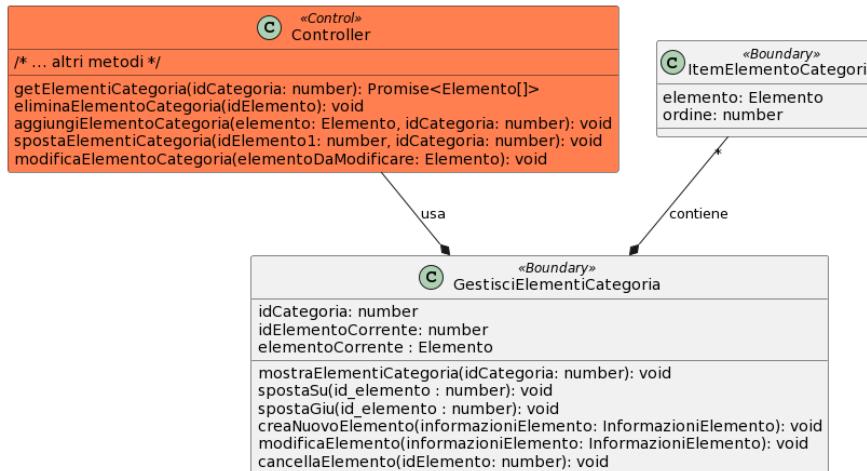
2.1.4 Dashboard iniziale & Dashboard Ristorante



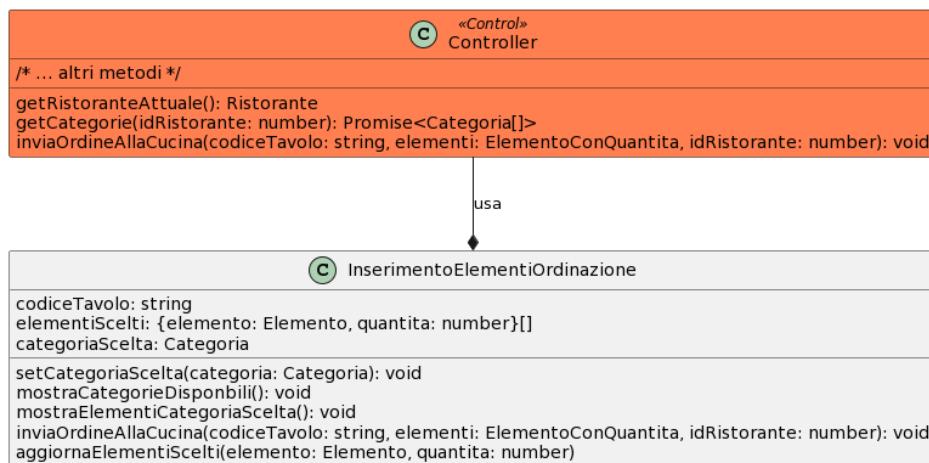
2.1.5 Gestisci categorie & Gestisci Conti



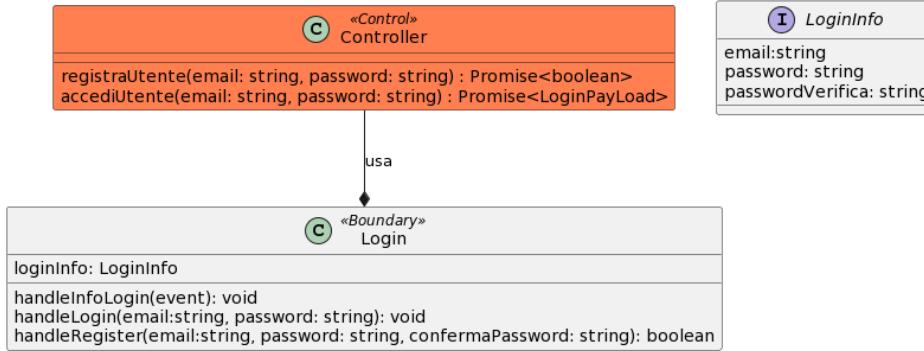
2.1.6 Gestisci elementi categoria



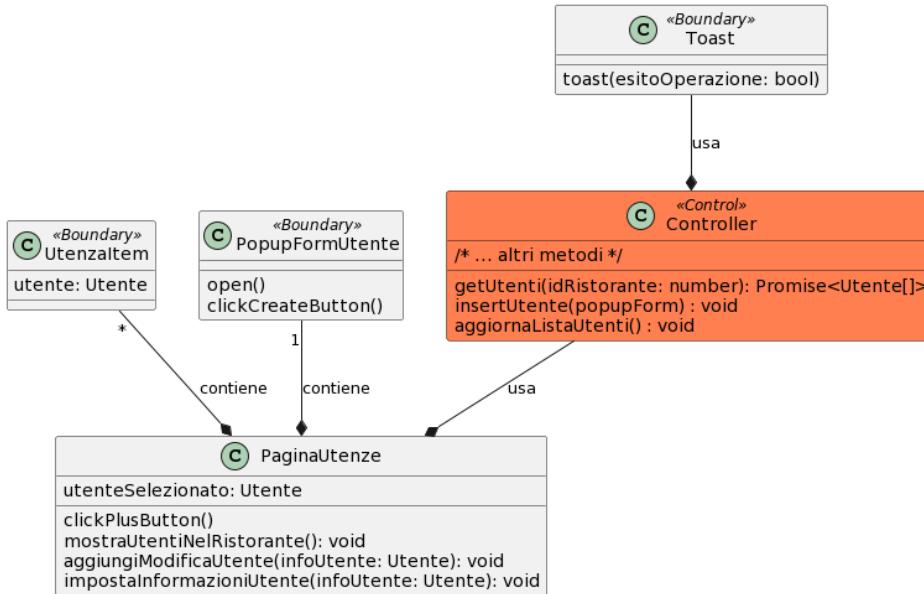
2.1.7 Inserimento elementi ordinazione



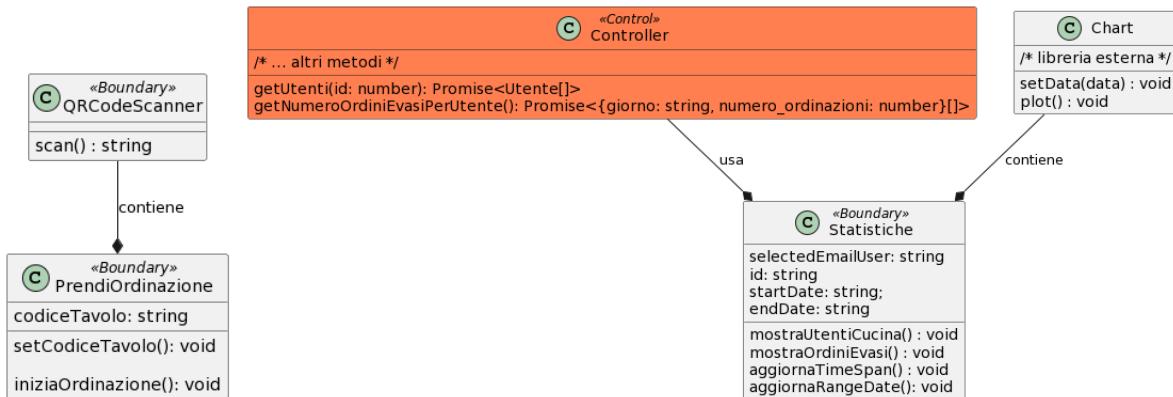
2.1.8 Login



2.1.9 Pagina Utenze



2.1.10 Prendi Ordinazione & Statistiche



2.1.11 Controller

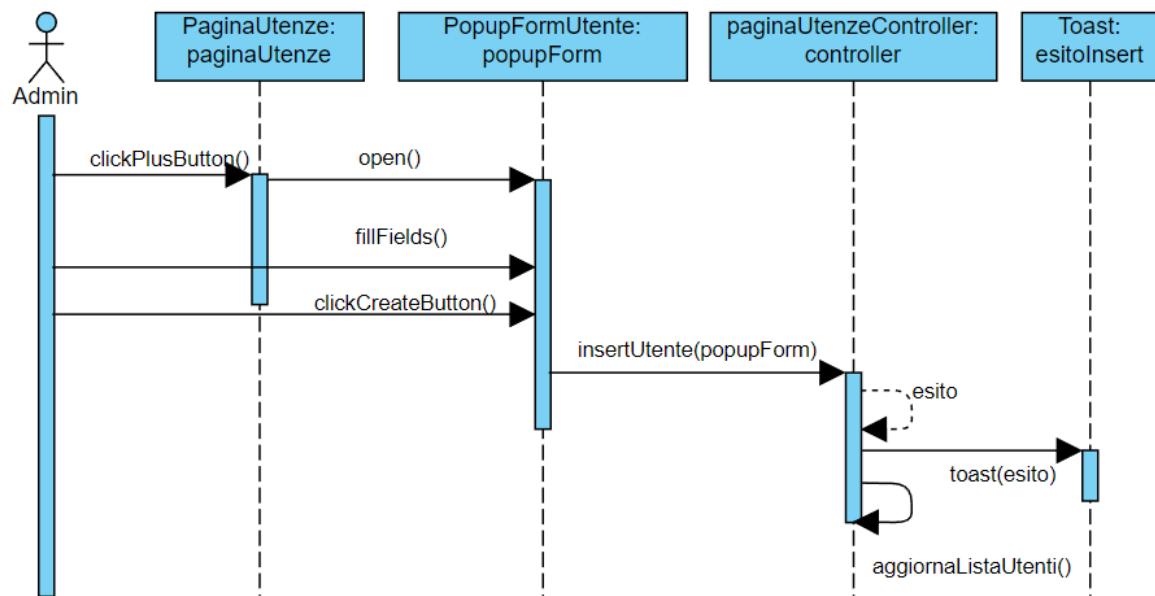
Il controller nell'EBC agisce come un intermediario tra Boundary e Entity, nel funzionamento tipico di un controller EBC (Entity Boundary Control), il Boundary invia una richiesta al controller, che può essere una richiesta di esecuzione di un'operazione, un aggiornamento dei dati o una richiesta di ottenere informazioni. Il controller riceve la richiesta e utilizza le informazioni fornite dal Boundary per prendere decisioni appropriate.

«Control» Controller	
<ul style="list-style-type: none"> □ static _instance: Controller; □ ristoranteDAO: RistoranteDAO; □ utenteDAO: UtenteDAO; □ contoDAO: ContoDAO; □ ordinazioneDAO: OrdinazioneDAO; □ elementoMenuDAO: ElementoDAO; □ categoriaDAO: CategoriaDAO; <ul style="list-style-type: none"> ● static getInstance() : Controller ● getOrdinazioniDaEvadere() : Promise<Ordinazioni> ● getOrdinazioniEvasiUltime24h() : Promise<Ordinazioni> ● evadi(ordine : Ordinazione) : void ● updateOrdine(ordine: Ordinazione) : void ● creaRistorante(ristorante : Ristorante) : void ● getRistoranti() : Ristorante[] ● getRistorante(id : string) : Promise<Ristorante> ● getCategorie() : Promise<Categoria[]> ● creaCategoria(Categoria categoria) : void ● modificaCategoria(c : Categoria, id : number) : void ● chiudiConto(conto : Conto) : void ● getContiTavoli() : Promise<Conto[]> ● getElementiCategoria(idCategoria: number): Promise<Elemento[]> ● eliminaElementoCategoria(idElemento): void ● aggiungiElementoCategoria(elemento: Elemento, idCategoria: number): void ● spostaElementiCategoria(idElemento1: number, idCategoria: number): void ● modificaElementoCategoria(elementoDaModificare: Elemento): void ● getUtenti(idRistorante: number): Promise<Utente[]> ● insertUtente(popupForm) : void ● aggiornaListaUtenti() : void ● getRistoranteAttuale(): Ristorante ● getCategorie(idRistorante: number): Promise<Categoria[]> ● inviaOrdineAllaCucina(codiceTavolo: string, elementi: ElementoConQuantita, idRistorante: number): void ● registraUtente(email: string, password: string) : Promise<boolean> ● accediUtente(email: string, password: string) : Promise<LoginPayLoad> ● getUtenti(id: number): Promise<Utente[]> ● getNumeroOrdiniEvasiPerUtente(): Promise<{giorno: string, numero_ordinazioni: number}[]> 	

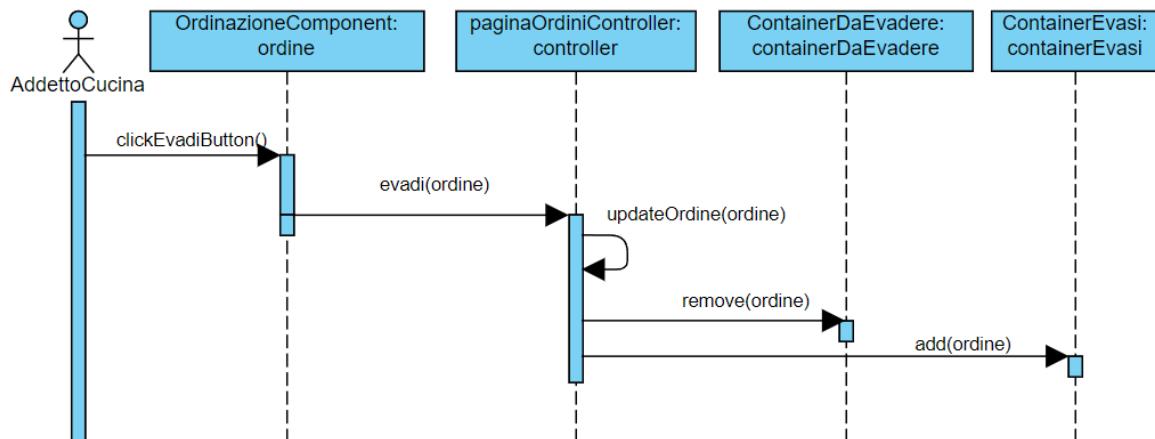
Utilizzo del design pattern Singleton Il design pattern Singleton è un pattern creazionale che garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza. La sua implementazione prevede la definizione di una classe con un costruttore privato e un metodo statico per ottenere l'istanza unica della classe. Il metodo statico controlla se l'istanza è già stata creata, se non lo è, la crea e la restituisce. In questo modo, ogni chiamata al metodo di accesso restituirà sempre la stessa istanza della classe. Il Singleton garantisce l'unicità della classe Controller permette di accedere ad esso in qualsiasi momento mediante l'invocazione del metodo statico *Controller.getInstance()*.

2.2 Diagrammi di sequenza di analisi

2.2.1 Crea utenza

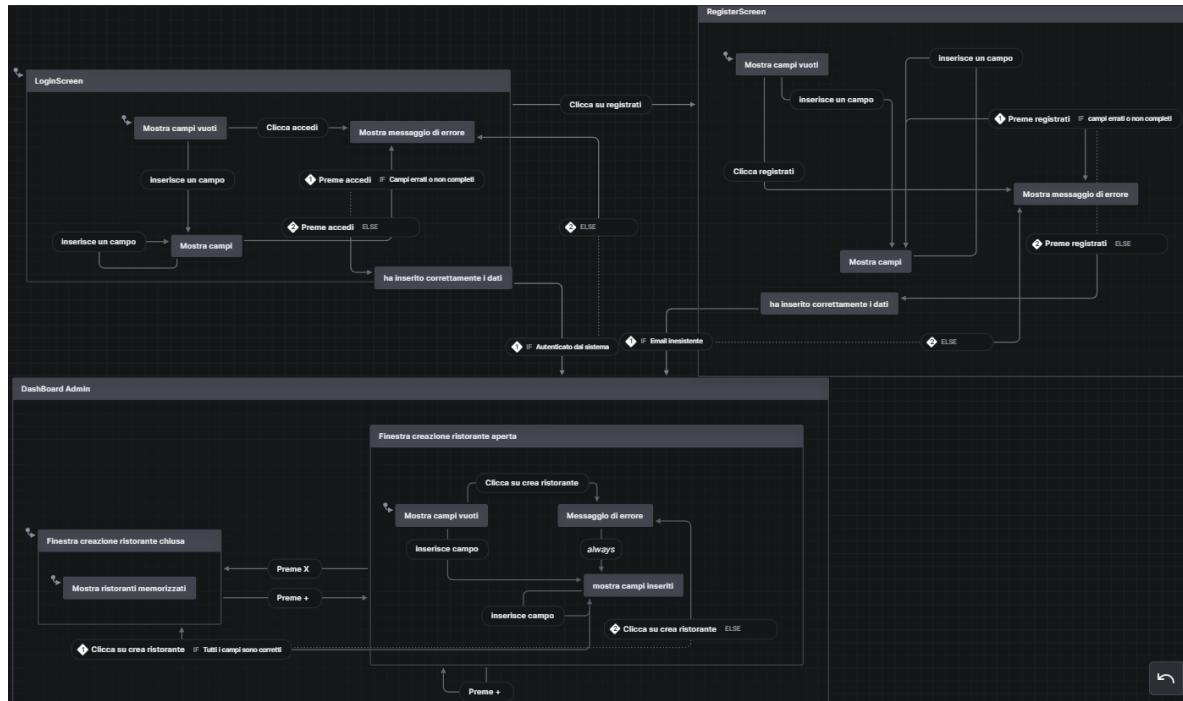


2.2.2 Visualizza e evadi ordini

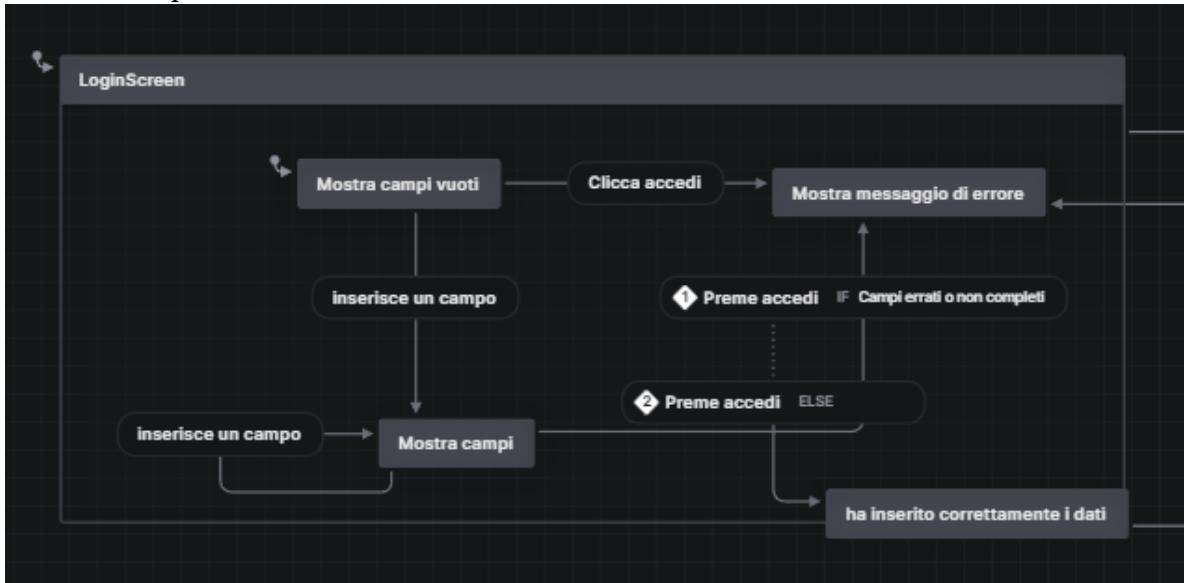


2.3 Prototipazione funzionale via statechart

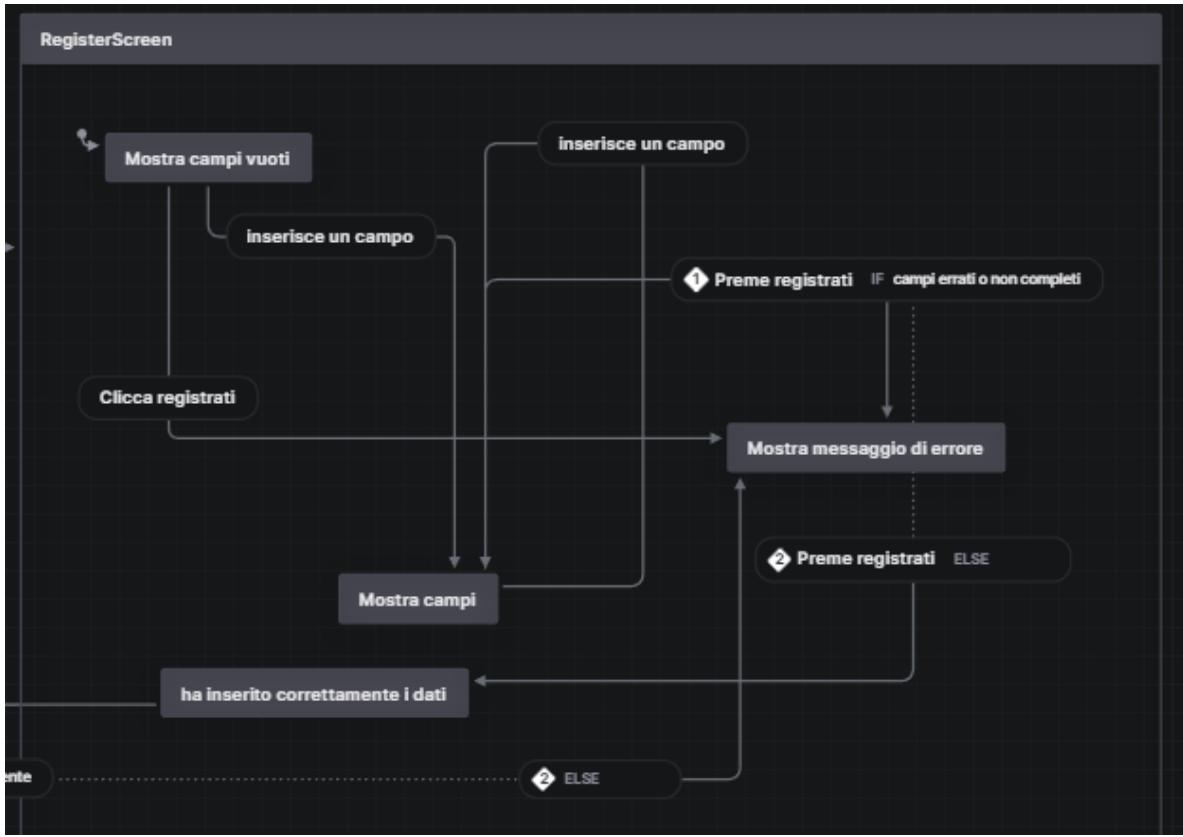
2.3.1 Statechart - Crea un Ristorante



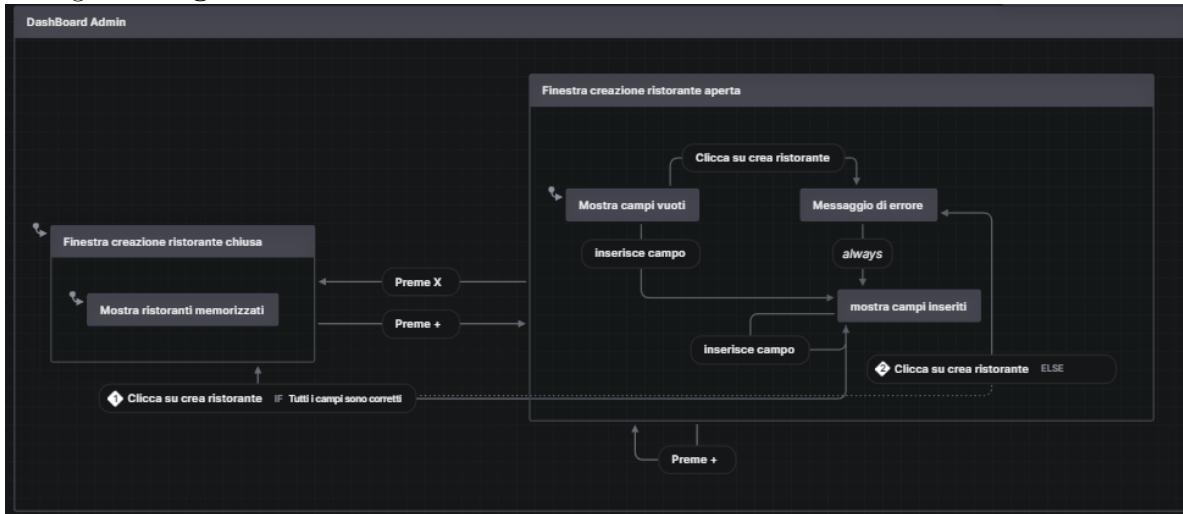
Schema complessivo



Dettaglio su Login

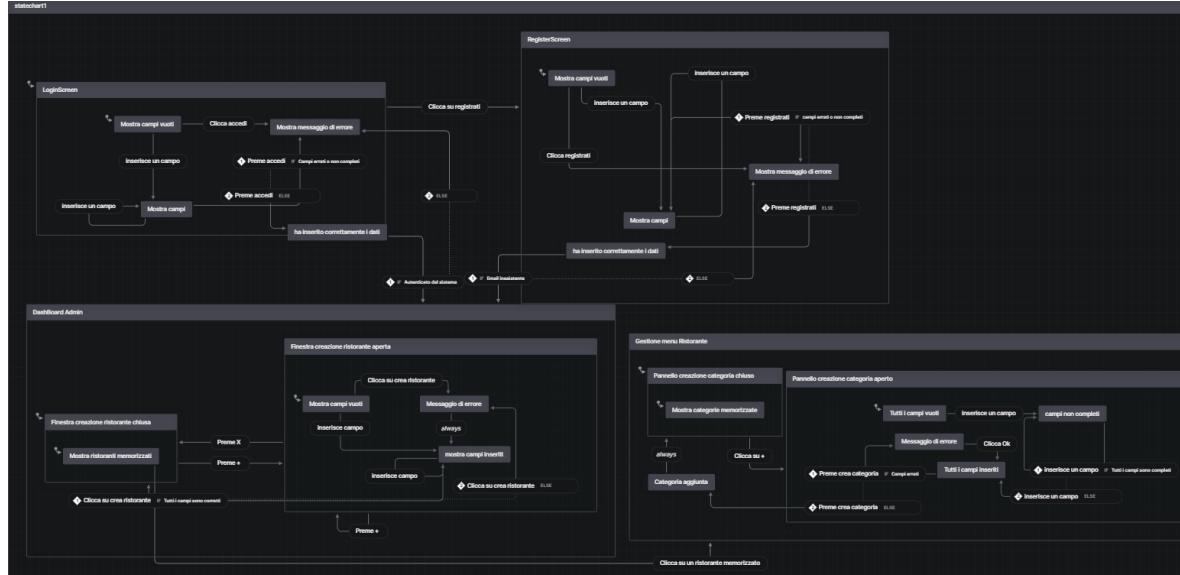


Dettaglio su Register

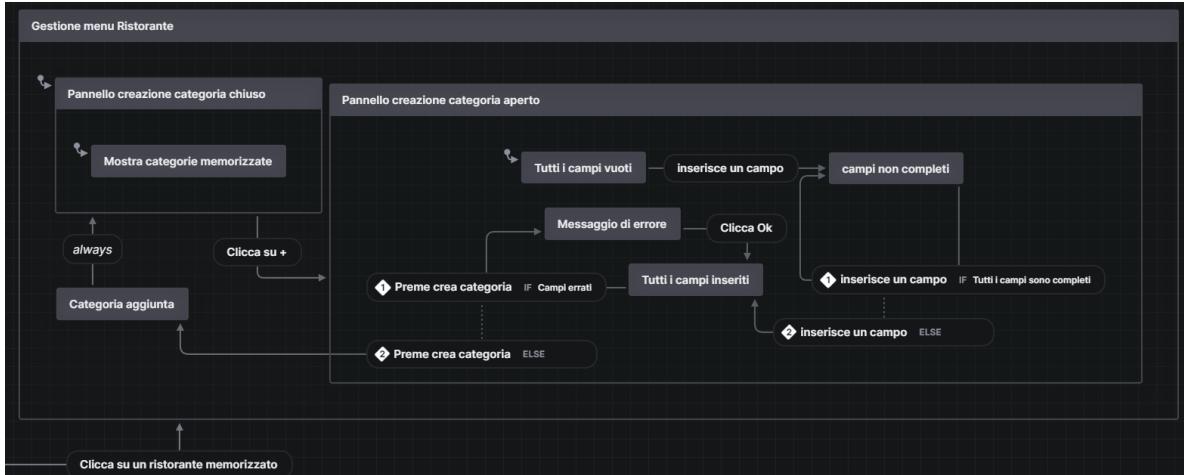


Dettaglio su Dashboard Admin

2.3.2 Statechart - Crea una Categoria



Schema complessivo



Dettaglio su Gestione menu Ristorante

II Documento di Design del Sistema

3 Analisi dell'architettura

3.1 Architettura 3 Tier

E' stato deciso di utilizzare un'architettura di tipo 3 tier. L'architettura 3 tier (o a 3 livelli) è un'architettura software che prevede la suddivisione dell'applicazione in tre livelli distinti: un livello di presentazione (client frontend), un livello di elaborazione (web server backend) e un livello di persistenza dei dati (information source).

3.1.1 Client - Tier 1

Il frontend, o livello di presentazione, ha lo scopo di creare l'interfaccia utente dell'applicazione, che l'utente finale vedrà e con cui interagirà. Nel caso specifico, il frontend è stato realizzato utilizzando React, una libreria JavaScript per la creazione di interfacce utente. Grazie alla sua leggerezza, React consente di creare interfacce utente reattive e performanti, migliorando l'esperienza dell'utente. Inoltre, il frontend dialoga con le API del backend, che forniscono i dati e le funzionalità necessarie per

l'applicazione, e con le API esterne per l'autocompletamento del testo. Questo significa che il frontend non si occupa di elaborare i dati o di effettuare operazioni complesse, ma si limita a presentare i dati e a interagire con gli altri livelli dell'applicazione. In definitiva, lo scopo del frontend è quello di fornire all'utente un'interfaccia intuitiva e funzionale per interagire con l'applicazione, senza dover preoccuparsi degli aspetti tecnici sottostanti.

3.1.2 Server - Tier 2

Il livello di backend, o livello di elaborazione, è responsabile dell'elaborazione dei dati e delle logiche di controllo dell'applicazione. Nel caso specifico, è stata realizzata una REST API in Node.js. Una REST API (API di tipo Representational State Transfer) è un'interfaccia che permette alle applicazioni di comunicare tra loro, scambiando dati in formato JSON o XML attraverso richieste HTTP. Essa è basata sul concetto di risorse, ovvero oggetti o dati che possono essere richiesti, creati, aggiornati o cancellati. Per garantire la sicurezza delle richieste e delle risposte, è stato utilizzato JSON Web Token (JWT). Un JWT è un token di sicurezza che viene utilizzato per autenticare e autorizzare gli utenti in modo sicuro, senza dover trasmettere le credenziali dell'utente ad ogni richiesta. Il server è un layer di controllo tra il client (ovvero il frontend) e il database. Esso riceve le richieste dal frontend, le elabora e le inoltra al database, poi riceve la risposta dal database, la elabora e la invia al frontend. In questo modo, il server offre un'interfaccia sicura e controllata per l'accesso ai dati e alle funzionalità dell'applicazione. In definitiva, il livello di backend è responsabile dell'elaborazione dei dati e delle logiche di controllo dell'applicazione, garantendo al contempo la sicurezza e il controllo dell'accesso ai dati e alle funzionalità dell'applicazione stessa.

3.1.3 Database - Tier 3

Il terzo tier dell'architettura a tre livelli è rappresentato dal database PostgreSQL. In questo caso, il database si trova sullo stesso server in cui gira il backend ed è stato creato utilizzando un file di creazione dello schema e delle istanze, che definisce la struttura del database e le tabelle in esso contenute. Inoltre, è stato deciso di utilizzare Docker per la gestione dell'intera applicazione, incluso il database. In particolare, il container di PostgreSQL viene avviato come primo, poiché il container del backend dipende da esso e richiede una connessione al database per funzionare correttamente.

3.1.4 Schema architettura

3.1.5 Docker containers in azione

	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	esame-ingsw-22-23	-	Running (2/2)			<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	node-db	postgres:latest	Running	42069:5432	10 hours ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	node-server	esame-ingsw-22-23-node-back	Running	3000:3000	22 seconds ago	<input type="button"/> <input type="button"/> <input type="button"/>

3.2 Documentazione del backend

Documentazione delle API Abbiamo documentato le API con il framework SwaggerUI che permette di generare una descrizione dettagliata di ogni route del backend e dà anche la possibilità di formattare e inviare richieste HTTP al server. [Link documentazione](#) (Link alla documentazione in cloud nella route /doc).

3.2.1 Tutte le routes del backend

Ratatouille - Documentazione API 0.0.1 OAS3

Software gestionale destinato all'uso nel settore della ristorazione.

Servers Authorize

Utente

- POST** /login
- POST** /register
- POST** /utenza/{id_ristorante}
- GET** /utenti/{id_ristorante}
- PUT** /utente/{email}
- DELETE** /utente/{email}
- GET** /utente/{id}
- GET** /pw-changed
- POST** /pw-change

Ristorante

- GET** /restaurants
- POST** /restaurant/{id}
- GET** /restaurant
- POST** /restaurant

Menu

- GET** /categorie/{id_ristorante}
- POST** /categoria
- DELETE** /categoria/{id_categoria}
- PUT** /categoria/{id_categoria}

Ordinazione

- POST** /ordina/{id_ristorante}
- POST** /ordinazioni/evase/
- GET** /ordinazioni/{is_evase}
- GET** /ordinazione/{id_ordinazione}

Elemento

- GET** /elementi/{id_categoria}
- GET** /elemento/{id_elemento}
- PUT** /scambia-elementi/{id_elemento1}/{id_elemento2}
- PUT** /elemento/:id_elemento
- DELETE** /elemento/:id_elemento
- POST** /elemento

The screenshot shows the SwaggerUI interface with two main sections:

- Conto** section:
 - GET /conto**: A blue button.
 - PUT /conto/{id_conto}**: An orange button.
- Allergene** section:
 - POST /allergene**: A green button.
 - GET /allergeni/{id_elemento}**: A blue button.
 - DELETE /allergene/{id_allergene}**: A red button.

3.2.2 Esempio di route

In questo esempio viene mostrato come SwaggerUI permette una documentazione dettagliata delle routes, l'invio di richieste HTTP formattate al server e la ricezione delle risposte.

This screenshot shows the configuration for a POST request to `/utenza/{id_ristorante}`:

- Description**: Crea l'account di un utente amministratore.
- Parameters** tab:

Name	Description
<code>id_ristorante</code> (path)	id del ristorante a cui appartiene l'utente
1	
- Request body** tab:


```
{
  "nome": "x",
  "cognome": "y",
  "ruolo": "CAMERIERE",
  "telefono": "3445566778",
  "supervisore": "false",
  "email": "xxxxxx.caio@gmail.com",
  "password": "123"
}
```
- Buttons**: Cancel (red), Reset (white).

Figure 6: Dettaglio su `/utenza/{id_restaurant}`.

This screenshot shows the 'Responses' section for the `/utenza/{id_ristorante}` route:

- Curl** tab:


```
curl -X POST -d '{"nome": "x", "cognome": "y", "ruolo": "CAMERIERE", "telefono": "3445566778", "supervisore": "false", "email": "xxxxxx.caio@gmail.com", "password": "123"}' -H "Content-Type: application/json" http://localhost:3000/api/utenza/1
```
- Request URL** tab:


```
http://localhost:3000/api/utenza/1
```
- Server response** tab:

Code	Details
200	Response body: <pre>{ "success": true, "data": "Registrazione avvenuta con successo" }</pre> Response headers: <pre>access-control-allow-origin: * connection: keep-alive content-length: 61 content-type: application/json; charset=utf-8 date: Mon, 01 May 2023 15:18:12 GMT etag: W/3d-1BnixXCG1g-bgIMHvtelMPc5dk8* keep-alive: timeout=5 x-powered-by: Express</pre>

Figure 7: Reale risposta al curl effettuato mediante SwaggerUI.

Responses		Links
Code	Description	
200	Utente registrato con successo	No links
	<p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header</p> <p>Example Value Schema</p> <pre>{ "success": true, "data": "Registrazione avvenuta con successo" }</pre>	
400	Errore durante la registrazione	No links
	<p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header</p> <p>Example Value Schema</p> <pre>{ "success": false, "data": "Errore durante la registrazione" }</pre>	
403	Accesso consentito solo a un admin	No links
	<p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header</p> <p>Example Value Schema</p> <pre>{ "success": false, "data": "Invalid token" }</pre>	

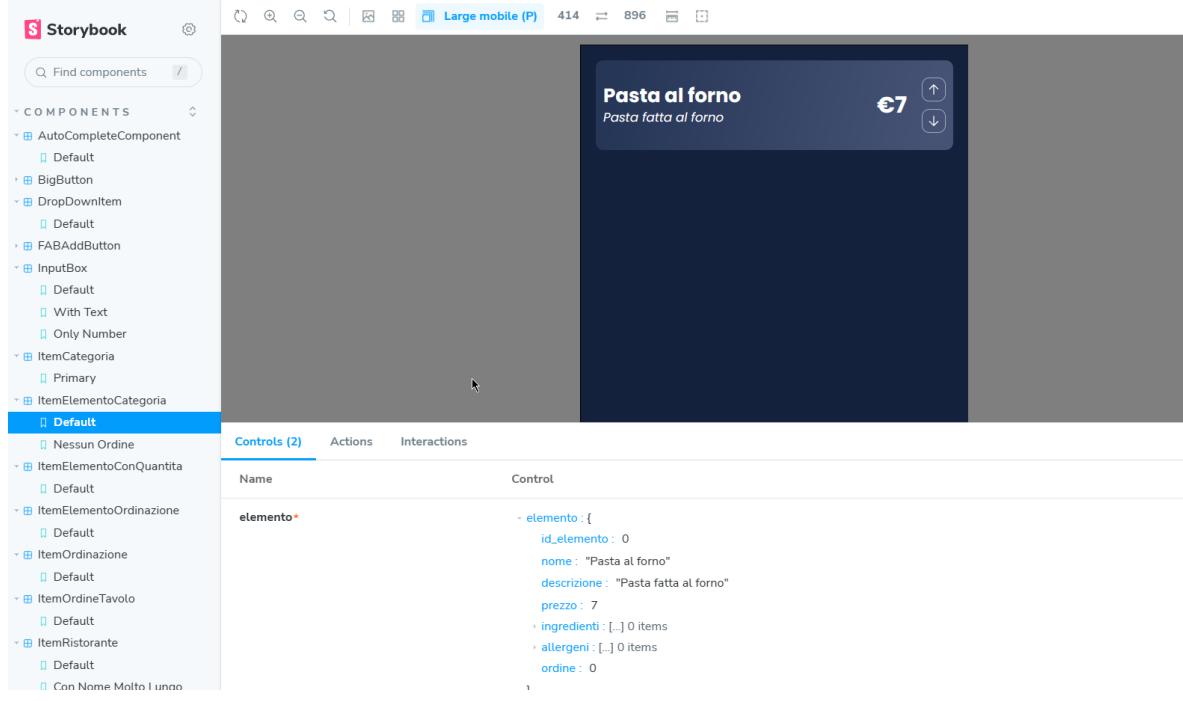
Figure 8: Esempi predefiniti di risposte.



Figure 9: SwaggerUI - Tool di documentazione del backend.

3.3 Documentazione del frontend

Il frontend è stato documentato con l'ausilio di Storybook. Storybook è un tool utile per la documentazione e il testing dei componenti di un'applicazione web sviluppata in React. Consente di creare un ambiente di sviluppo in cui è possibile visualizzare tutti i componenti React e generare automaticamente una documentazione completa. La documentazione può essere compilata e rilasciata su un server, rendendola disponibile anche per altri membri del team o per la comunità open source. Storybook riduce il tempo e lo sforzo necessario per lo sviluppo e la documentazione dei componenti React. Ecco un'anteprima di Storybook:



The screenshot shows the Storybook interface. On the left is a sidebar with a tree view of components, including AutoCompleteComponent, BigButton, DropDownItem, FABAddButton, InputBox, ItemCategoria, ItemElementoCategoria, and ItemOrdineTavolo. The 'Default' item under ItemElementoCategoria is selected. The main area displays a component preview for 'Pasta al forno' with a price of €7. Below the preview is a table titled 'Controls (2)' showing the state of the 'elemento' prop. The table has two columns: 'Name' and 'Control'. The 'elemento' row shows the following JSON state:

Name	Control
elemento*	- elemento : { id_elemento : 0 nome : "Pasta al forno" descrizione : "Pasta fatta al forno" prezzo : 7 ingredienti : [...] 0 items allergeni : [...] 0 items ordine : 0}

[Link documentazione](#) (Link alla documentazione in cloud).



Figure 10: StoryBook - Tool di Documentazione del frontend

3.4 Cloud Hosting

DigitalOcean è stata la nostra scelta per rendere accessibile il software su Internet, grazie alla sua capacità di fornire un'infrastruttura di hosting affidabile e facile da usare. DigitalOcean ci ha fornito un indirizzo IP statico, il che significa che il nostro sito web sarà sempre raggiungibile utilizzando lo stesso indirizzo IP. Questo è particolarmente importante per le applicazioni che richiedono un'accessibilità costante e affidabile. Per quanto riguarda la configurazione, Docker ha reso la distribuzione dell'applicazione estremamente semplice e portatile, consentendoci di caricare l'immagine del server in cloud. Infine, abbiamo caricato il frontend compilato e ha funzionato senza problemi.

3.5 Pannello di controllo in cloud

3.6 Servizi utilizzati in cloud

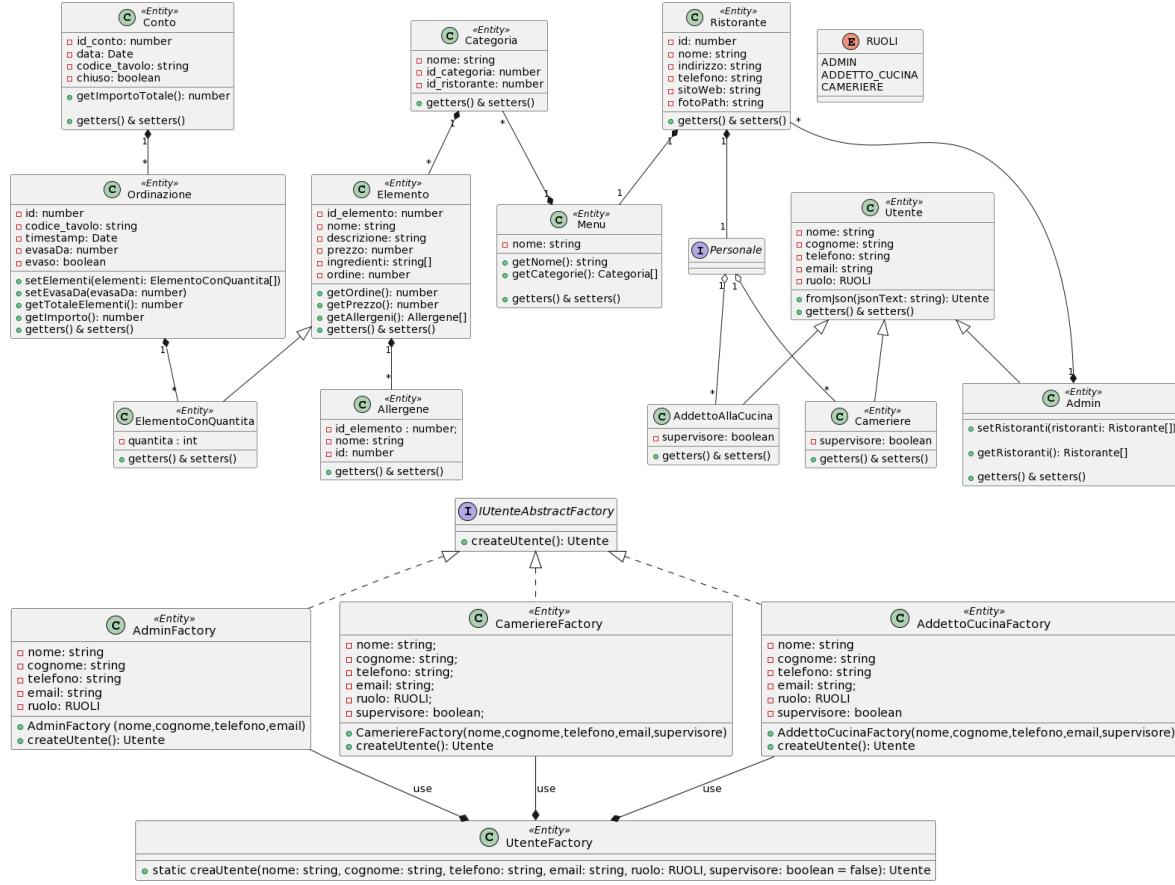
1. **Firebase con Google Analytics:** funzionalità di raccolta dati sugli utenti.
2. **DigitalOcean:** hosting dell'applicativo
3. **MealDB:** API food autocomplete

4 Motivazione delle scelte adottate

L'architettura a tre livelli è stata scelta per organizzare il progetto in modo più strutturato e versatile. In questo modo, ogni livello ha un compito specifico e ben definito, e le interazioni tra di essi sono chiare e gestibili. Inoltre, questo tipo di architettura consente di separare la logica di presentazione dall'elaborazione dei dati, semplificando lo sviluppo e la manutenzione dell'applicazione. Per quanto riguarda la scelta del cloud, è stata fatta considerando diversi fattori come il numero di accessi al mese, il costo, la banda e lo spazio a disposizione. Il cloud offre molte soluzioni di hosting scalabili e personalizzabili, che possono adattarsi alle esigenze dell'applicazione. L'utilizzo di Docker è stato scelto per ottenere un'applicazione robusta e facilmente replicabile in diversi ambienti. Docker permette di isolare i servizi in container, in modo da gestirli in maniera indipendente e controllata. Inoltre, Docker garantisce che i servizi vengano avviati nell'ordine corretto, semplificando la configurazione dell'applicazione. Infine, l'uso di Docker consente di avere il controllo completo delle porte esposte, che possono essere gestite in modo flessibile e personalizzato a seconda delle esigenze dell'applicazione.

5 Diagramma delle classi di design

5.1 Le Entities

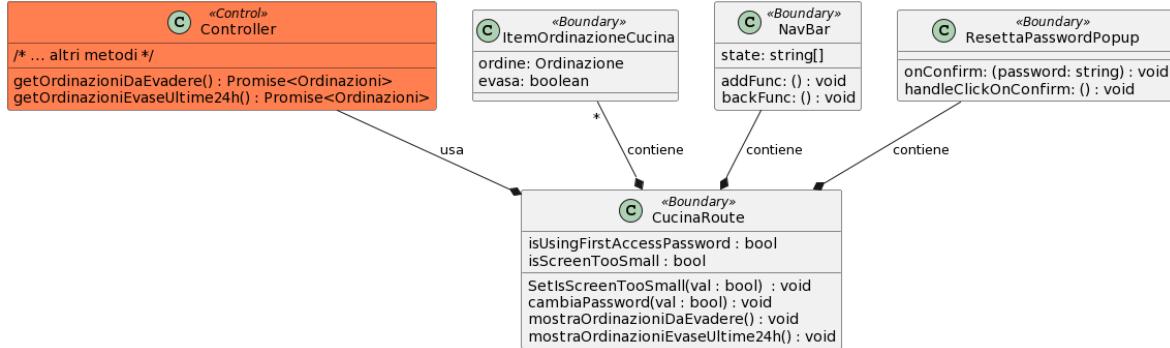


5.1.1 Utilizzo del design pattern Factory Method

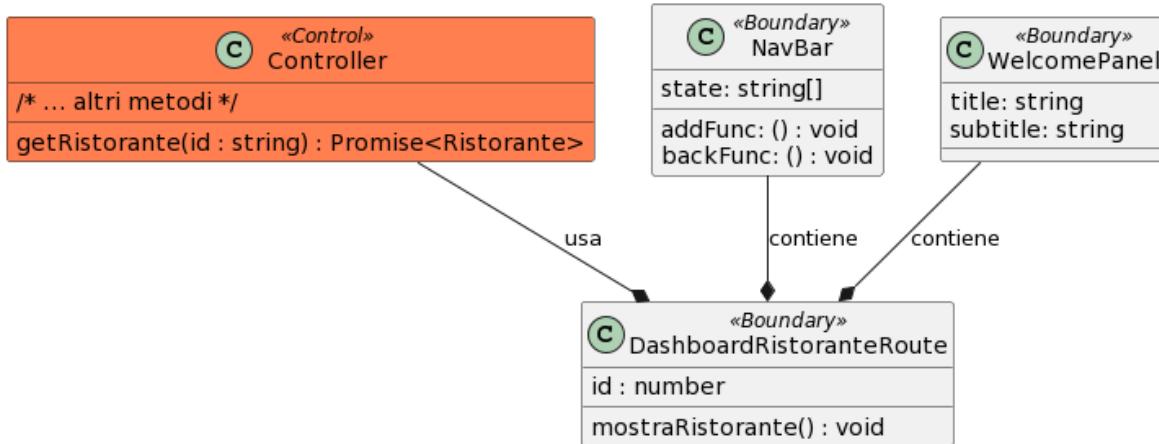
E' stato utilizzato il design pattern "Factory Method" per istanziare l'utente correttamente. Il design pattern Factory Method permette di creare oggetti senza bisogno di dover specificare la loro classe. Ciò significa che il codice interagisce esclusivamente con l'interfaccia risultante o la classe astratta, in modo che funzioni con qualsiasi classe che implementi l'interfaccia o che estenda la classe *IUtenteAbstractFactory*. Per istanziare un utente basterà chiamare il metodo statico *UtenteFactory.creaUtente(nome, cognome...)* con i relativi parametri. Il Factory Method provvederà a gestire la logica di creazione dell'utente e restituirà correttamente un *Utente* di sottoclassi *Admin*, *Cameriere* o *AddettoAllaCucina*. Strutturando il codice secondo questo pattern è possibile anticipare il cambiamento delle specifiche e aggiungere nuovi tipi di utenti senza dover rivoluzionare il class diagram.

5.2 I Boundaries

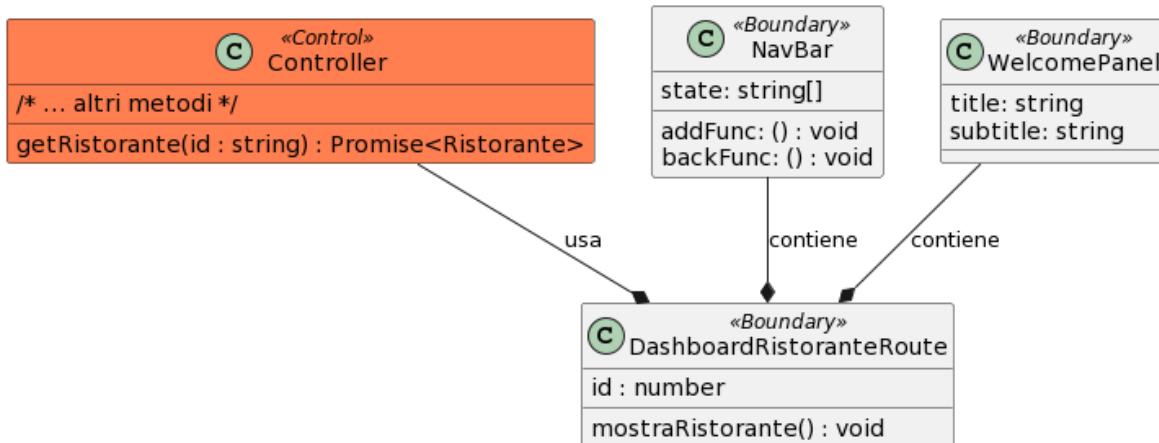
5.2.1 Cucina



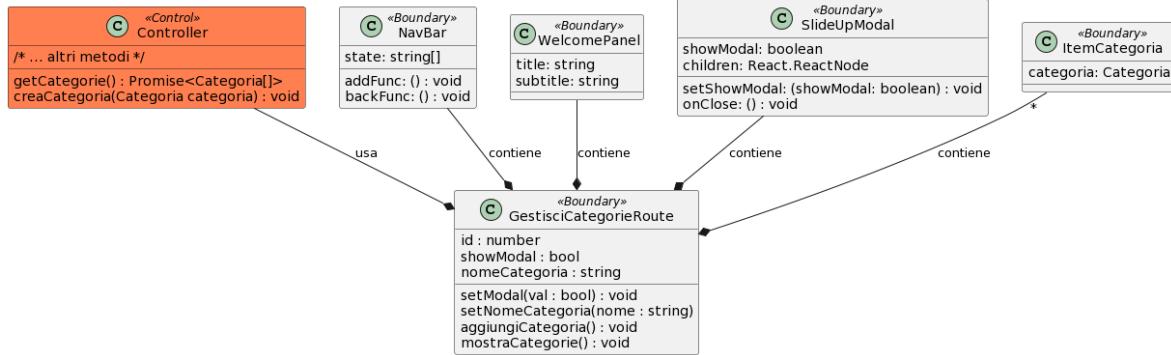
5.2.2 Dashboard ristorante



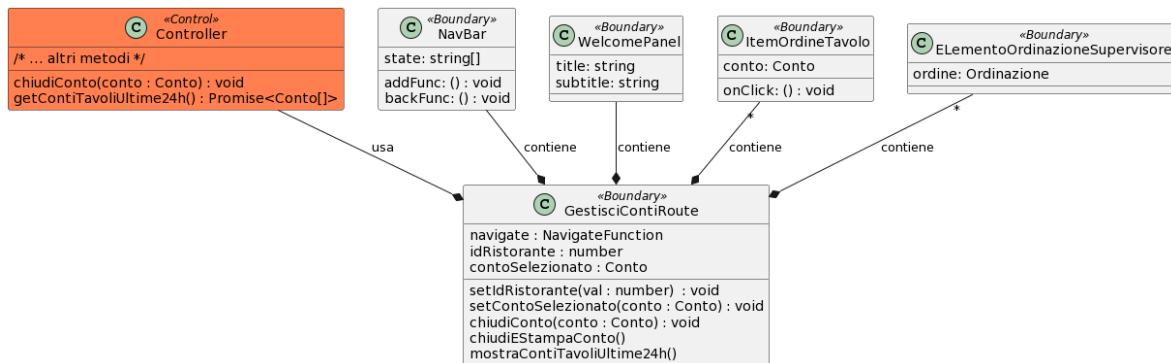
5.2.3 Dashboard supervisore



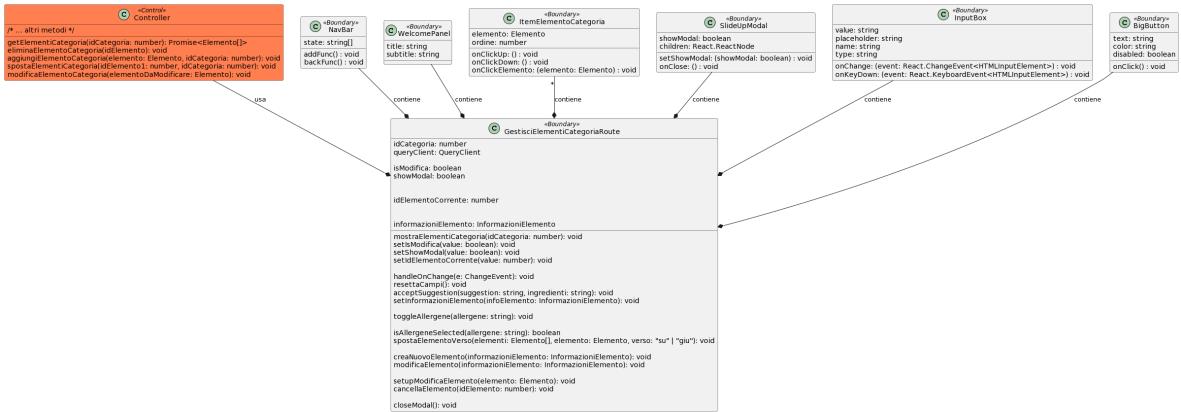
5.2.4 Gestisci categorie



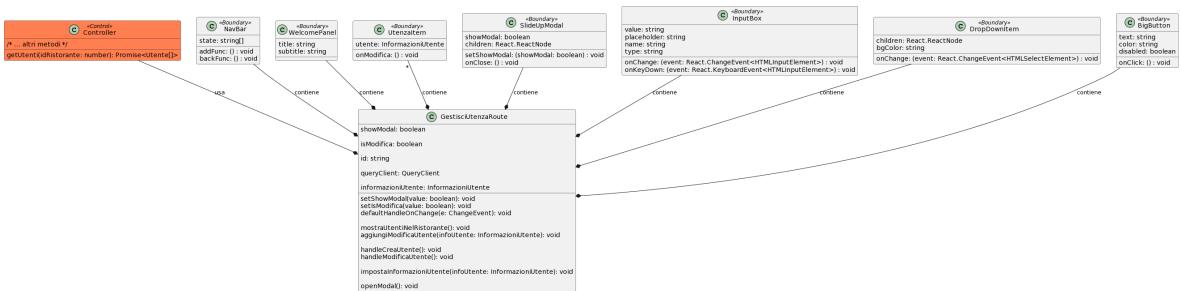
5.2.5 Gestisci conti



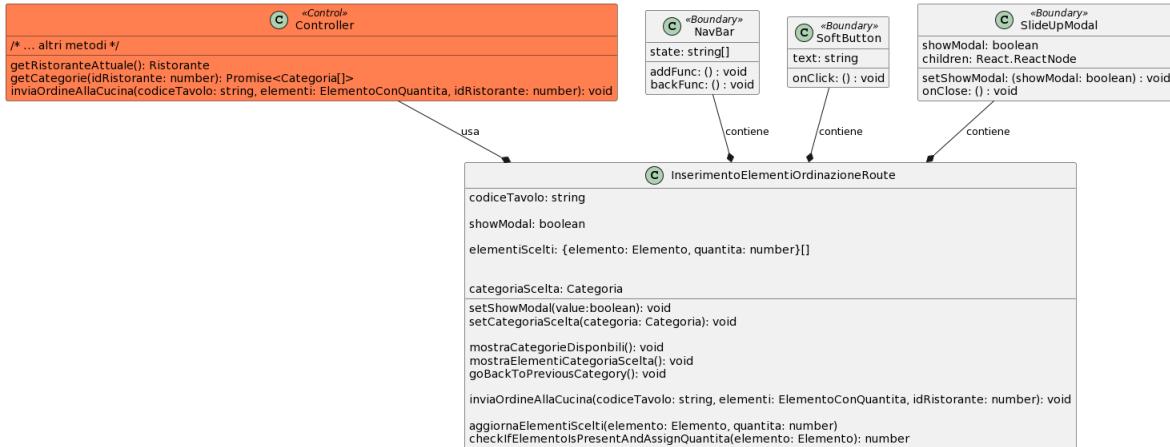
5.2.6 Gestisci elementi di una categoria



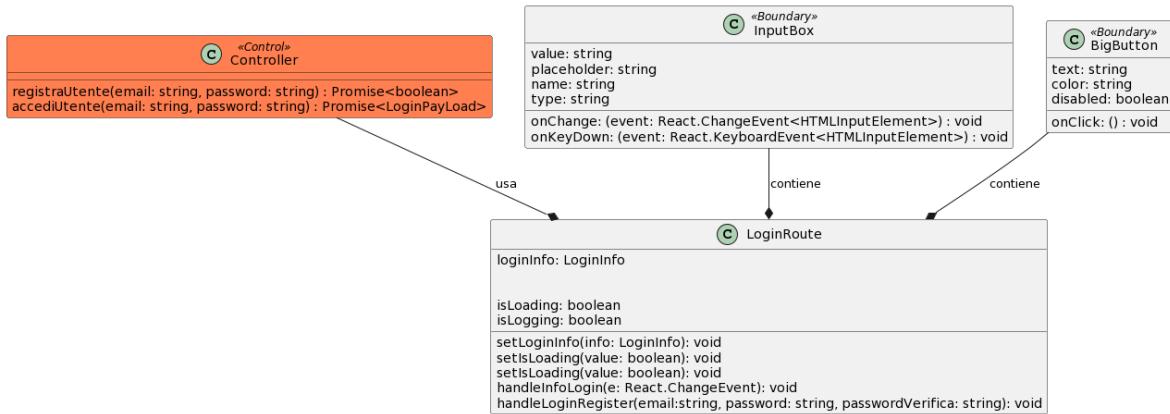
5.2.7 Gestisci utenza



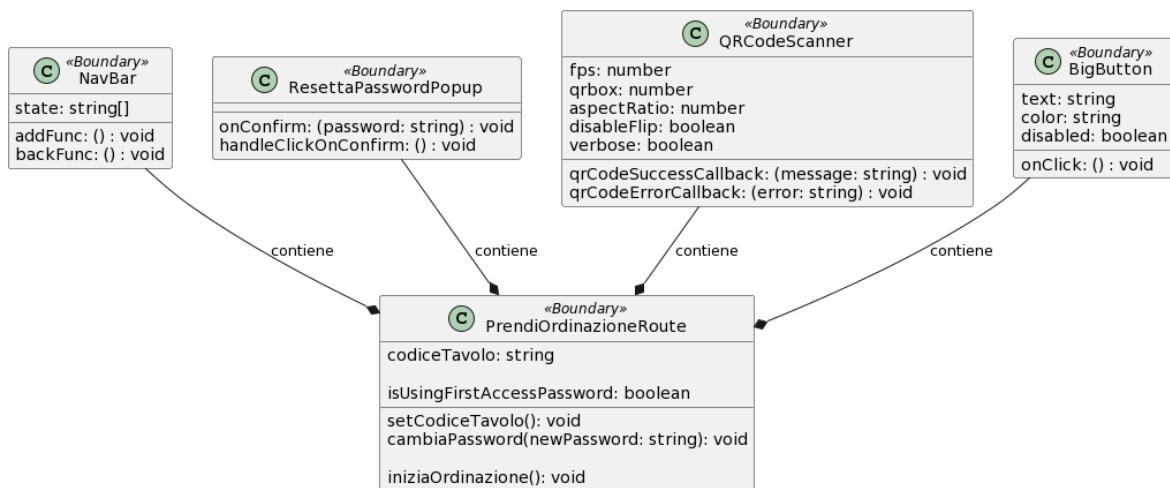
5.2.8 Inserimento elementi di un'ordinazione



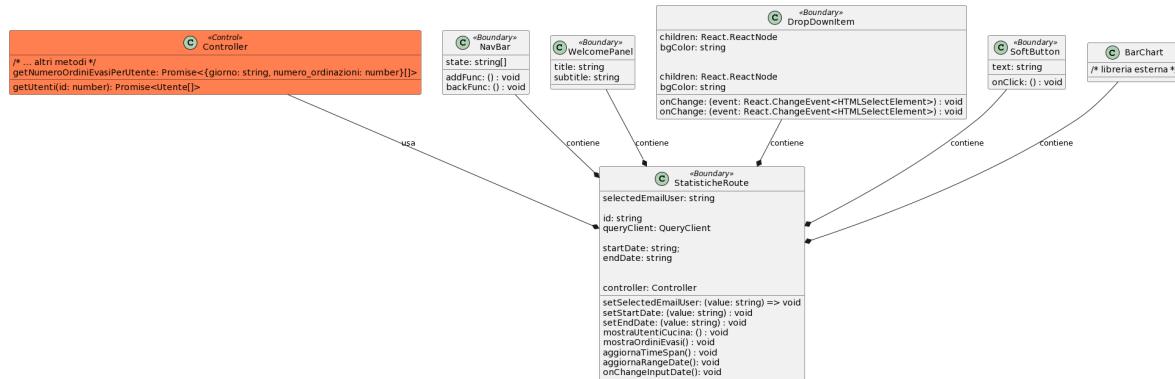
5.2.9 Login e registrazione



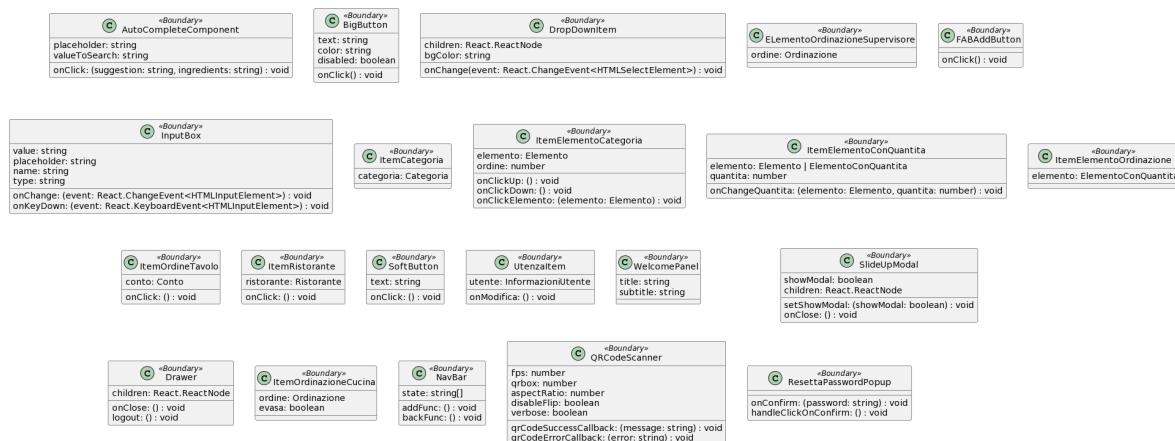
5.2.10 Ordina



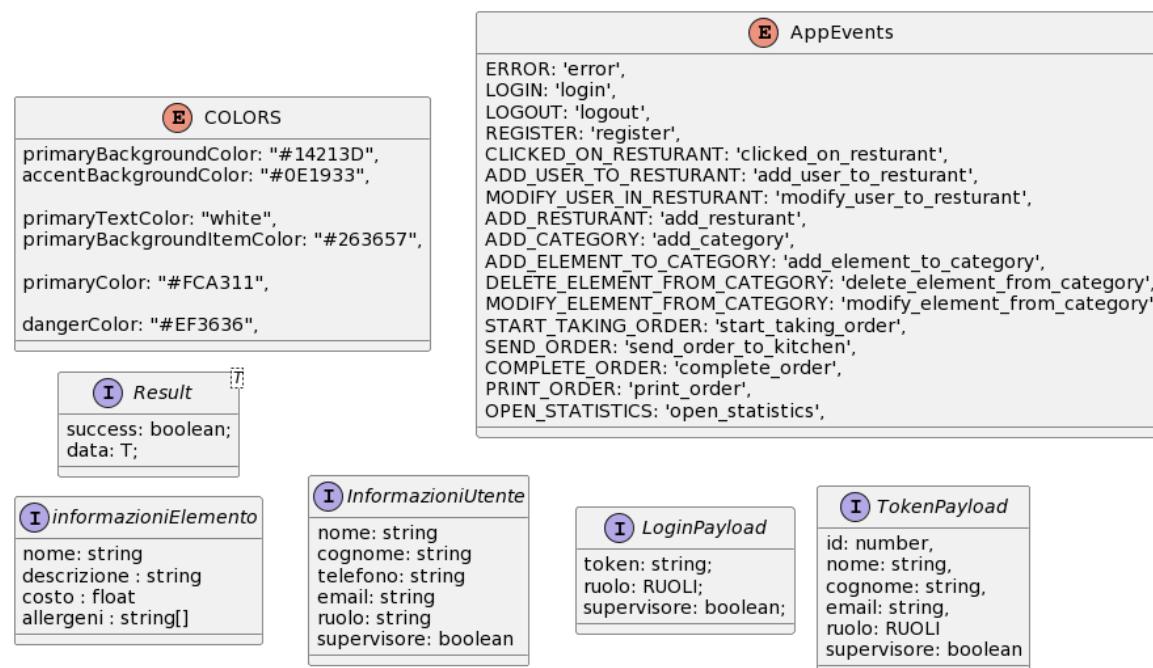
5.2.11 Statistiche



5.2.12 Tutti i components



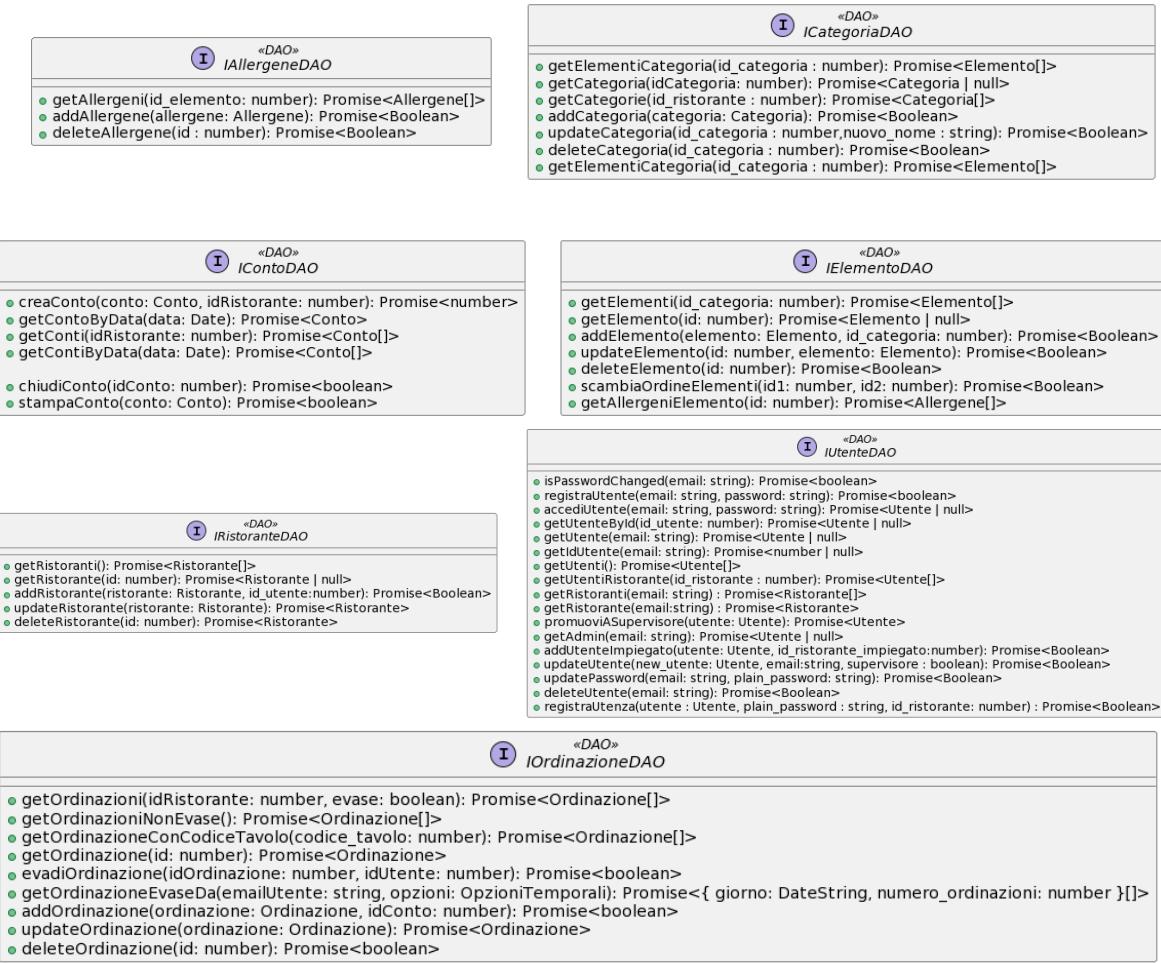
5.2.13 Interfacce e enums di utilità



5.3 Il Controller

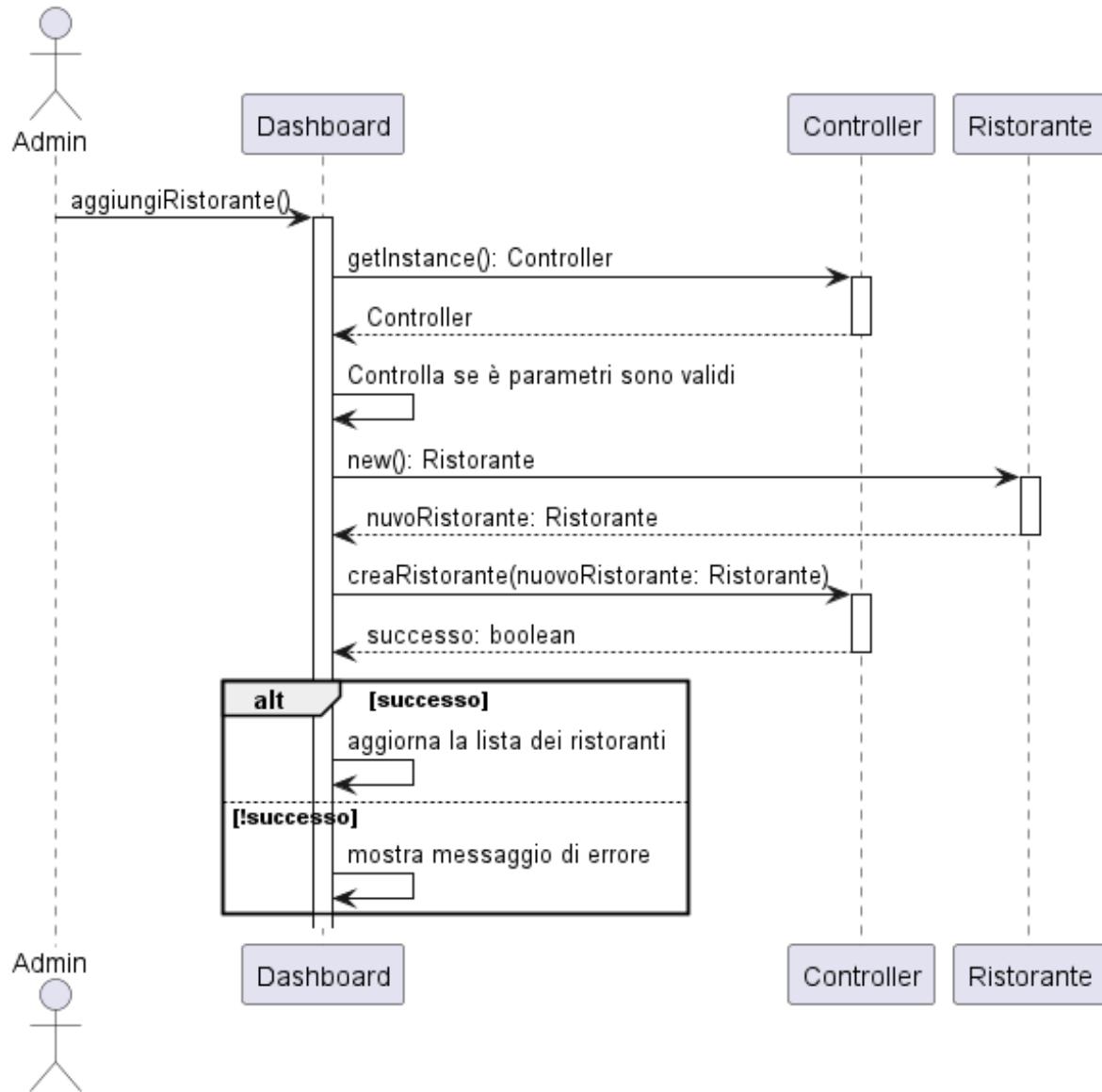


5.4 DAOs

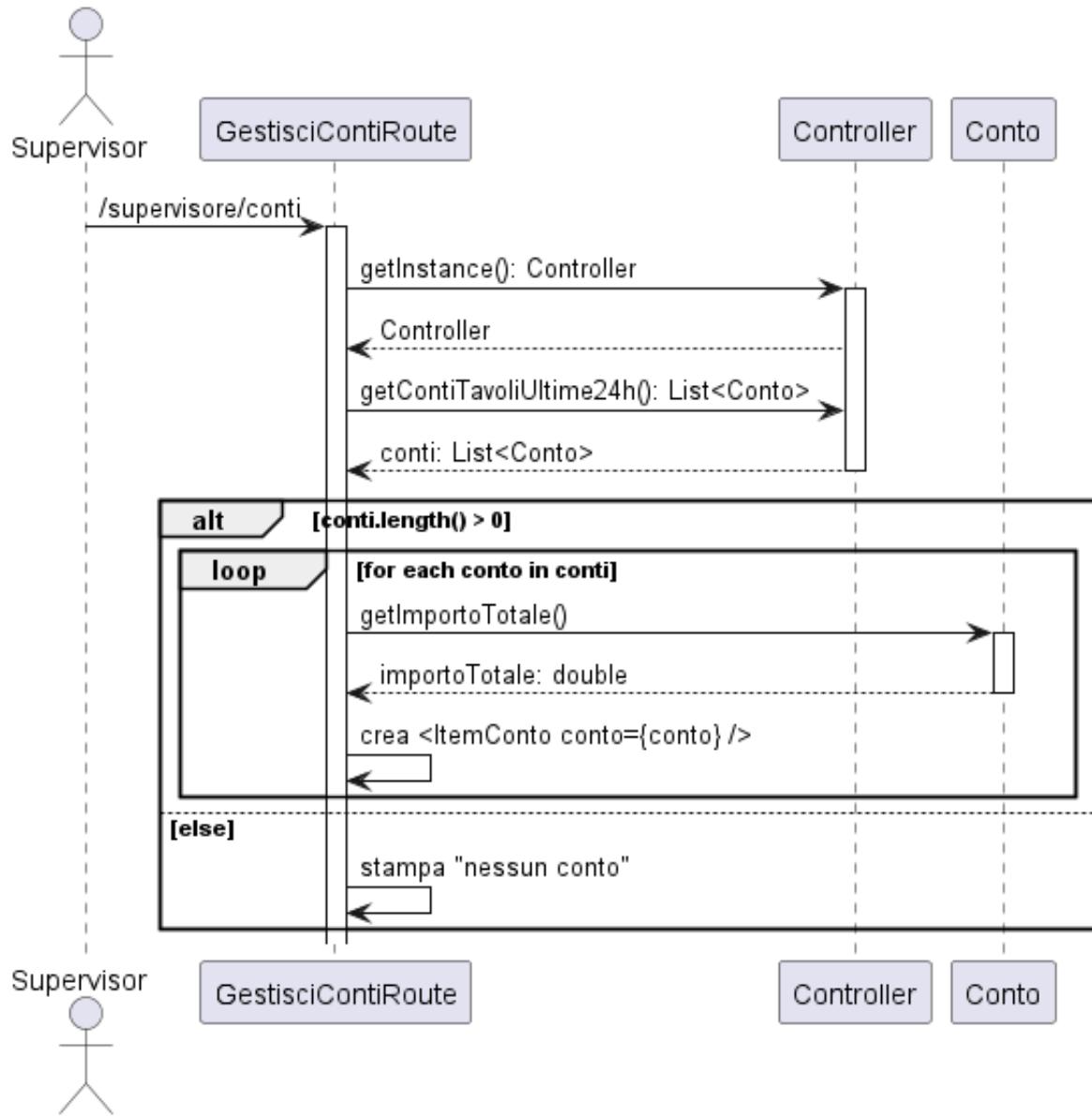


6 Diagrammi di sequenza di design

6.1 aggiungiRistorante



6.2 getContiUltime24h



III Testing e valutazione sul campo dell'usabilità

7 Codice xUnit Black Box per Unit Testing

Al fine di garantire una maggiore affidabilità del software è stato effettuato unit testing su 4 funzioni non banali con almeno 2 parametri.

7.1 Funzione addElemento

La funzione **addElemento** aggiunge un elemento a una categoria del menu. I primi due parametri sono destinati alla query all'API e il terzo parametro è il token JWT. Ecco la signature del metodo:

```
async addElemento(elemento: Elemento, idCategoria: number, token?: string): Promise<Result<string>>
```

7.1.1 Strategia di testing Black Box

Sono state individuate le classi di equivalenza come oggetto del test.

Nome classe equivalenza	Nome parametro	Insieme dei valori	Output
elementoCE1	elemento	{nome vuoto}	Errore
elementoCE2	elemento	{quantità <=0}	Errore
elementoCE3	elemento	{valido}	Successo
idCategoriaCE1	idCategoria	{inesistente nel db}	Errore
idCategoriaCE3	idCategoria	{esistente nel db}	Successo
tokenCE1	token	{undefined}	Errore
tokenCE2	token	{valido}	Autenticato
tokenCE3	token	{non valido}	Non autenticato

E' stata utilizzata la strategia SECT (Strong Equivalence Class Testing) per implementare i metodi riportati nel codice xUnit Black Box di seguito.

7.1.2 Codice xUnit Black Box

```
const email = 'mario.rossi@gmail.com';
const password = 'mario';
const token = (await utenteDAO.accediUtente(email, password)).data.token;

describe('addElement()', () => {
    it("dovrebbe ritornare true se l'elemento è stato registrato con successo", async () => {
        const elemento = new Elemento(`test${randomInt(8000)}`,
            "Descrizione",
            2,
            {
                ingredienti: [],
                allergeni: [],
                ordine: 0
            }
        );
        const result = await elementoDAO.addElement(elemento,1,token);
        expect(result.success).toBe(true);
        expect(result.data).toBeTypeOf('string');
    });

    it("dovrebbe ritornare false se il nome dell'elemento è ''", async () => {
        const elemento = new Elemento("", "Descrizione",
            2,
            {
                ingredienti: [],
                allergeni: [],
                ordine: 0
            }
        );
        const result = await elementoDAO.addElement(elemento,1,token);
        expect(result.success).toBe(false);
        expect(result.data).toBeTypeOf('string');
    });

    it("dovrebbe ritornare false se il prezzo è negativo", async () => {
        const elemento = new Elemento("prova", "Descrizione",
            -1,
            {
                ingredienti: [],
                allergeni: [],
                ordine: 0
            }
        );
        const result = await elementoDAO.addElement(elemento,1,token);
        expect(result.success).toBe(false);
        expect(result.data).toBeTypeOf('string');
    });
});
```

7.2 Funzione scambiaElementi

La funzione **scambiaElementi** ha due parametri destinati alla query dell'API e un terzo parametro che è il token JWT. Ecco la signature del metodo:

```
async scambiaElementi(idElemento1: number, idElemento2: number, token?: string): Promise<Result<string>>
```

7.3 Strategia di testing Black Box

Sono state individuate le classi di equivalenza come oggetto del test.

Nome classe equivalenza	Nome parametro	Insieme dei valori	Output
idElemento1CE1	idElemento1	{id > 0}	Successo
idElemento1CE2	idElemento1	{id ≤ 0}	Errore
idElemento2CE1	idElemento2	{id > 0}	Successo
idElemento2CE2	idElemento2	{id ≤ 0}	Errore
tokenCE1	token	{undefined}	Errore
tokenCE2	token	{valido}	Autenticato
tokenCE3	token	{non valido}	Non autenticato

E' stata utilizzata la strategia SECT (Strong Equivalence Class Testing) per implementare i metodi riportati nel codice xUnit Black Box di seguito.

7.3.1 Codice xUnit Black Box

```
describe('scambiaElementi()', () => [  
  it("Lo scambio di due elementi esistenti dovrebbe ritornare true", async () => {  
    const result = await elementoDAO.scambiaElementi(1,2,token);  
    expect(result.success).toBe(true);  
    expect(result.data).toBeTypeOf('string');  
  });  
  
  it("Lo scambio dello stesso elemento dovrebbe ritornare un messaggio di alert", async () => {  
    const result = await elementoDAO.scambiaElementi(1,1,token);  
    expect(result.success).toBe(true);  
    expect(result.data).toBeTypeOf('string');  
    expect(result.data).toBe('Non ha senso scambiare lo stesso elemento');  
  });  
  
  it("Lo scambio elementi che non esistono non solleva errore e ritorna false", async () => {  
    const result = await elementoDAO.scambiaElementi(-2,-1,token);  
    expect(result.success).toBe(false);  
    expect(result.data).toBeTypeOf('string');  
  });  
];
```

7.4 Funzione registraUtente

La funzione **registraUtente** ha due parametri e se non esiste un utente con la stessa email lo crea. Ecco la signature del metodo:

```
async registraUtente(email: string, password: string): Promise<boolean> {
```

7.4.1 Strategia di testing Black Box

Sono state individuate le classi di equivalenza come oggetto del test.

Nome classe equivalenza	Nome parametro	Insieme dei valori	Output
emailCE1	email	{stringa vuota}	Errore
emailCE2	email	{stringa valida}	Esito autenticazione
emailCE3	email	{email esistente}	Esito negativo autenticazione
passwordCE1	password	{stringa vuota}	Errore
passwordCE2	password	{stringa valida}	Esito autenticazione

E' stata utilizzata la strategia SECT (Strong Equivalence Class Testing) per implementare i metodi riportati nel codice xUnit Black Box di seguito.

7.4.2 Codice xUnit Black Box

```
describe('registraUtente()', () => {  
  
    it("dovrebbe ritornare false se l'email o la password sono vuoti ('')", async () => {  
        const email = '';  
        const password = '';  
  
        const response = await utenteDAO.registraUtente(email, password);  
  
        expect(response).toBe(false)  
    });  
  
    it("dovrebbe ritornare true se l'utente è stato registrato con successo", async () => {  
        const email = `test${randomInt(8000)}@test.com`;  
        const password = 'password';  
  
        const result = await utenteDAO.registraUtente(email, password);  
  
        expect(result).toBe(true)  
    });  
  
    it("dovrebbe ritornare false se l'utente non è stato registrato con successo, perchè già registrato", async () => {  
        const email = 'mario.rossi@gmail.com';  
        const password = 'mario';  
  
        const result = await utenteDAO.registraUtente(email, password);  
  
        expect(result).toBe(false)  
    });  
});
```

7.5 Funzione accediUtente

La funzione `accediUtente` ha due parametri, autentica l'utente e ritorna un payload. Ecco la signature del metodo:

```
async accediUtente(email: string, password: string): Promise<Result<LoginPayload>>
```

7.5.1 Strategia di testing Black Box

Sono state individuate le classi di equivalenza come oggetto del test.

Nome classe equivalenza	Nome parametro	Insieme dei valori	Output
emailCE1	email	{stringa vuota}	Errore
emailCE2	email	{stringa valida}	Esito autenticazione dell'utente
passwordCE1	password	{stringa vuota}	Errore
passwordCE2	password	{stringa valida}	Esito autenticazione dell'utente

E' stata utilizzata la strategia SECT (Strong Equivalence Class Testing) per implementare i metodi riportati nel codice xUnit Black Box di seguito.

7.5.2 Codice xUnit Black Box

```
describe('accediUtente()', () => {

    it("dovrebbe ritornare un oggetto di tipo Result<string> se l'utente è stato loggato con successo", async () {
        const email = "mario.rossi@gmail.com"
        const password = "mario"

        const result = await utenteDAO.accediUtente(email, password);

        expect(result.success).toBe(true);
        expect(result.data).toBeTypeOf('object');
    });

    it("dovrebbe ritornare un oggetto di tipo Result<string> se l'utente non è stato loggato con successo", async () {
        const email = "mario.rossi@gmail.com"
        const password = "password_errata"

        const result = await utenteDAO.accediUtente(email, password);

        expect(result.success).toBe(false);
        expect(result.data).toBeTypeOf('object');
    });

    it("dovrebbe ritornare un oggetto di tipo Result<string> se l'email o la password sono vuoti ('')", async () {
        const email = "";
        const password = "";

        const result = await utenteDAO.accediUtente(email, password);

        expect(result.success).toBe(false);
        expect(result.data).toBeTypeOf('object');
    });
});
```

8 Valutazione dell'usabilità sul campo

8.1 Test di valutazione finale

Durante lo sviluppo del software abbiamo avuto modo di correggere alcune problematiche che influenzavano in modo negativo l'esperienza utente. Grazie a un test di valutazione finale, effettuato su un gruppo di testers, siamo riusciti a ottenere un feedback positivo dell'applicazione. Ecco i risultati del test:

	1	2	3	4	5	6	7
Tester 1	✓	✓	✓	✓	✓	✓	✓
Tester 2	✓	✓	✓	✓	✓	✓	✓
Tester 3	✓	✓	✓	✓	✓	✓	✓
Tester 4	✓	✓	✓	✓	✓	🚚	✓

Leggenda:

- ✓ Successo +1
- ✗ Fallimento -1
- 🚚 Successi Parziali +0.5

Il punteggio ottenuto è migliorato rispetto al precedente test (Vedi Figura 2 nella sezione 1.6).

8.2 Analisi mediante file di log

Il sistema Ratatouille23 raccoglie dati sugli utenti a scopo di analisi. Per raccogliere dati sugli utenti è stato utilizzata la piattaforma Firebase. Firebase è una piattaforma di sviluppo mobile e web, acquisita da Google. Essa fornisce una vasta gamma di servizi, tra cui l'autenticazione degli utenti, il database in tempo reale, l'hosting dei siti web, il cloud storage, i servizi di messaggistica, le funzionalità di analisi come Google Analytics. Firebase e Google Analytics sono strettamente integrati e consentono di tracciare gli eventi degli utenti e le attività all'interno di un'applicazione mobile o web. In particolare, Firebase fornisce la possibilità di inviare eventi personalizzati ad Analytics, in modo da tenere traccia dell'utilizzo dell'applicazione e degli eventuali problemi riscontrati dagli utenti.



8.3 Integrazione con SDK di Firebase

Integrare Firebase nel progetto è stata estremamente semplice grazie alla ottima documentazione. Sono state necessarie poche modifiche per aggiungere il servizio nell'applicativo. Firebase è in grado di tener traccia delle azioni compiute dall'utente. Le azioni dell'utente sono conservate in un oggetto detto *AppEvents*. Quando si vuole registrare un evento viene invocata la funzione *logEvent* di cui è stato realizzato un wrapper che semplifica la scelta dei tipi di eventi da inviare con il relativo payload opzionale.

```
32 export const AppEvents = {
33   ERROR: 'error',
34   LOGIN: 'login',
35   LOGOUT: 'logout',
36   REGISTER: 'register',
37   CLICKED_ON_RESTAURANT: 'clicked_on_restaurant',
38   ADD_USER_TO_RESTAURANT: 'add_user_to_restaurant',
39   MODIFY_USER_IN_RESTAURANT: 'modify_user_to_restaurant',
40   ADD_RESTAURANT: 'add_restaurant',
41   ADD_CATEGORY: 'add_category',
42   ADD_ELEMENT_TO_CATEGORY: 'add_element_to_category',
43   DELETE_ELEMENT_FROM_CATEGORY: 'delete_element_from_category',
44   MODIFY_ELEMENT_FROM_CATEGORY: 'modify_element_from_category',
45   START_TAKING_ORDER: 'start_taking_order',
46   SEND_ORDER: 'send_order_to_kitchen',
47   COMPLETE_ORDER: 'complete_order',
48   PRINT_ORDER: 'print_order',
49   OPEN_STATISTICS: 'open_statistics',
50 } as const;
```

Figure 12: Oggetto con tutti i tipi di evento registrati.

```
1 import { AppEvents } from './utils/constants';
2 import { initializeApp } from "firebase/app";
3 import { getAnalytics, logEvent } from "firebase/analytics";
4
5 const firebaseConfig = {
6   apiKey: "████████████████",
7   authDomain: "ratatouille-app-2023.firebaseio.com",
8   projectId: "ratatouille-app-2023",
9   storageBucket: "ratatouille-app-2023.appspot.com",
10  messagingSenderId: "████████████",
11  appId: "████████████████",
12  measurementId: "████████"
13};
14
15 type AppEventsType = typeof AppEvents[keyof typeof AppEvents];
16
17 const logEventToFirebase = (eventType: AppEventsType, payload?: any) => {
18   logEvent(analytics, eventType.toString(), payload);
19 }
20
21 // Initialize Firebase
22 const app = initializeApp(firebaseConfig);
23 const analytics = getAnalytics(app);
24
25 export {
26   analytics,
27   logEventToFirebase
28 }
```

Figure 13: API keys di Firebase e wrapper di *logEvents*.

8.4 Analisi dei dati ottenuti con Google Analytics

Effettuando il login sulla piattaforma Firebase è possibile consultare le statistiche degli utenti del software Ratatouille²³. Ecco un'anteprima delle informazioni ottenute dagli utenti.

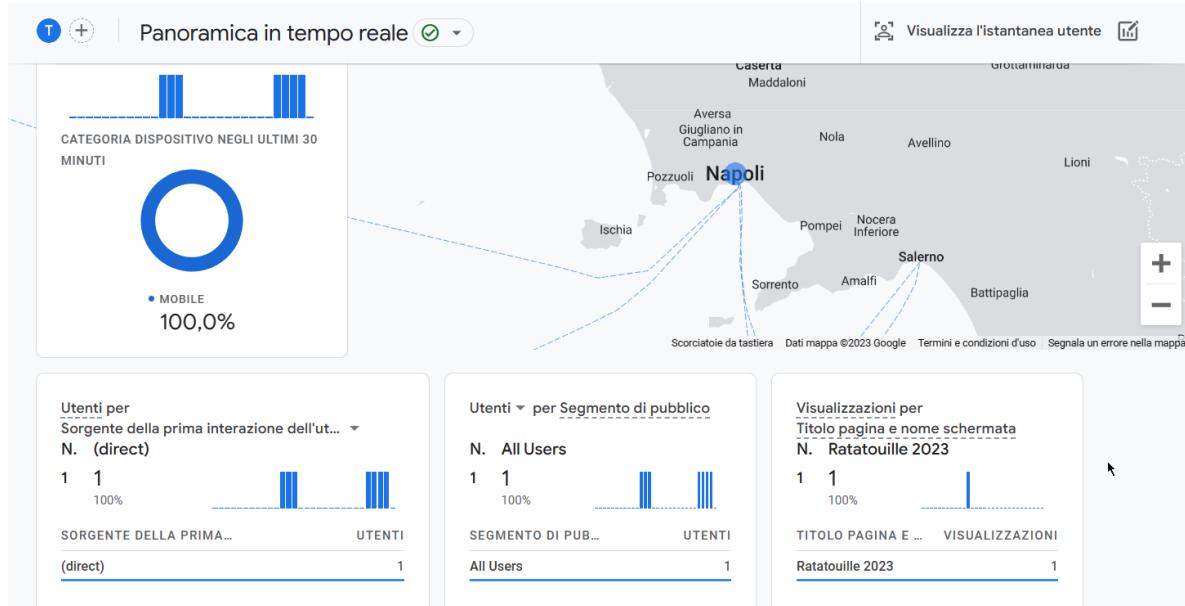
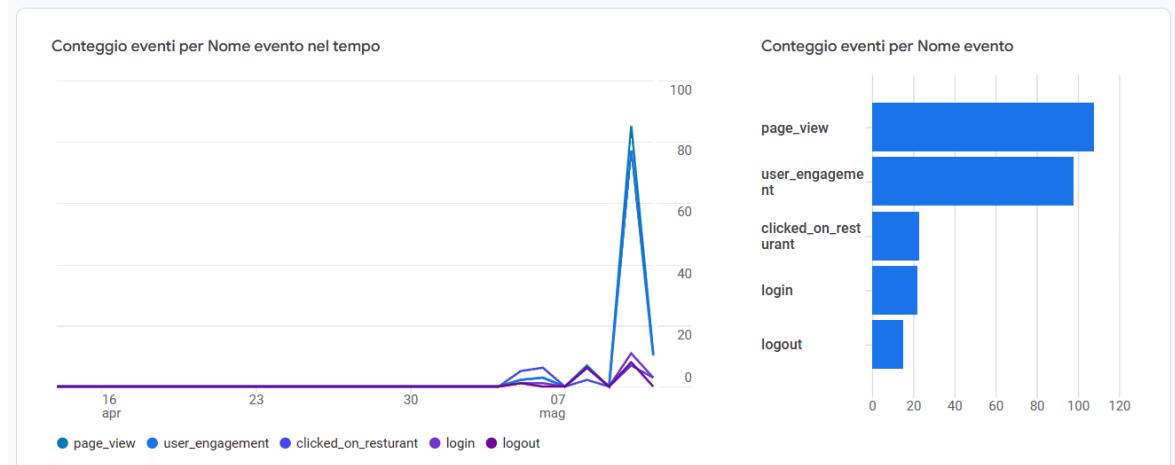


Figure 14: Panoramica generale con localizzazione degli utenti.



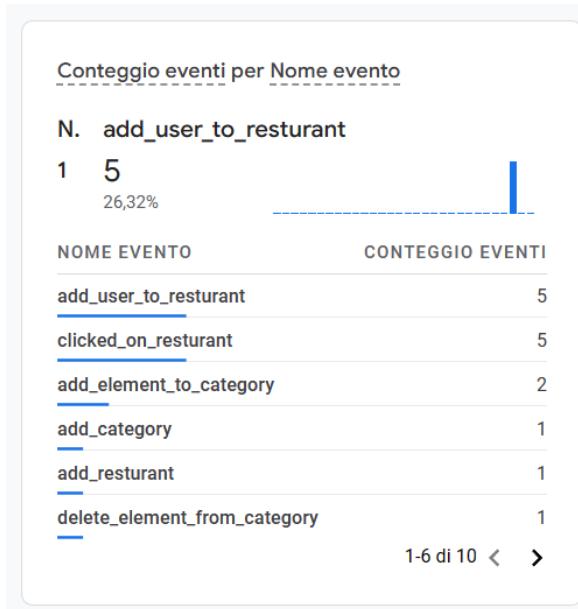


Figure 15: Conteggio degli eventi

8.5 File di log

Firebase permette di scaricare un resoconto (file di log) delle statistiche degli utenti. Ecco il file di log con tutte le informazioni ottenute dagli utenti [Link al file di log](#)

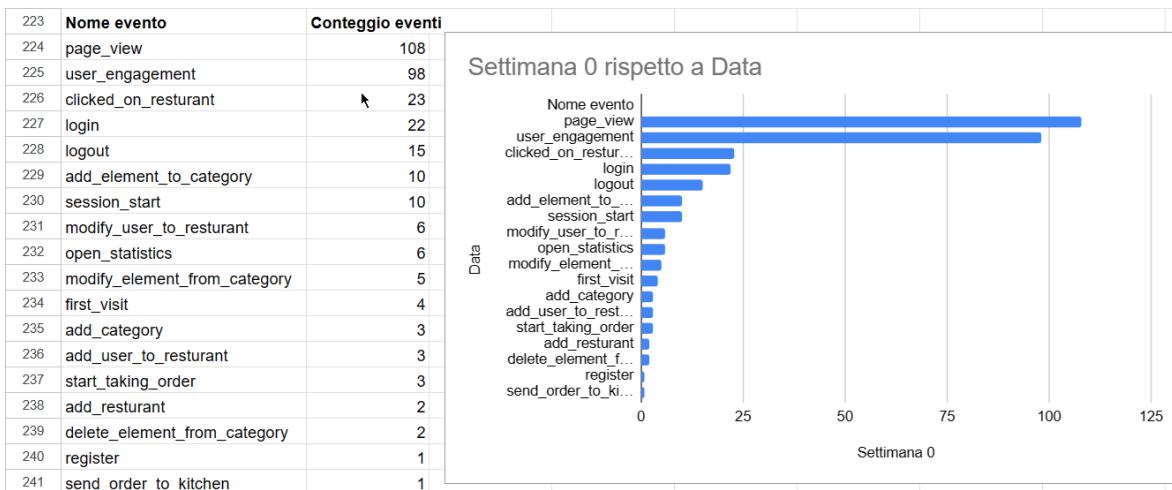


Figure 16: Visualizzazione del file di log CSV in Google Sheets