



Dipartimento di Ingegneria e Scienza
dell'Informazione

Progetto:

U-Sushi

Titolo del documento:

Documento di sviluppo dell'applicazione web

INDICE

Scopo del documento	2
1. User Flow	2
Admin	3
Cliente	4
Cucina	5
Logout	5
2. Implementazione e Documentazione dell'App	5
Struttura del progetto	6
Dipendenze del progetto	8
Relazioni e Tipi nel database	9
3. APIs del progetto	12
Estrazione delle risorse dal class diagram	12
Diagramma delle risorse	12
Resource Diagram 1	13
Resource Diagram 2	13
Resource Diagram 3	14
Resource Diagram 4	14
Resource Diagram 5	15
Resource Diagram 6	15
Resource Diagram 7	16
Resource Diagram Completo	17
4. Sviluppo delle APIs	18
APIs relative a USER	18
APIs relative a PLATE	18
APIs relative a CATEGORY	19
APIs relative a INGREDIENT	20
APIs relative a COMMAND	20
APIs relative a IMAGE	21
APIs relative a SESSION	21
5. API Documentation	22
6. Testing	25
7. GitHub e Deployment	28
Esecuzione in locale	28

Scopo del documento

Il seguente documento riporta tutte le informazioni necessarie per descrivere lo sviluppo di una parte, molto completa, dell'applicazione web U-Sushi.

Nel primo capitolo viene riportato lo user flow, ovvero una descrizione tramite diagramma di tutte le azioni che si possono eseguire sulla parte implementata di U-Sushi.

Successivamente rappresentiamo una struttura del codice realizzato, descrivendo le dipendenze installate, i modelli realizzati e le APIs implementate.

Una descrizione delle APIs implementate viene fatta con il diagramma delle risorse e il diagramma di estrazione delle risorse, in cui si individuano le risorse estratte.

Si spiega poi ciò che si è fatto con Swagger per la documentazione delle APIs.

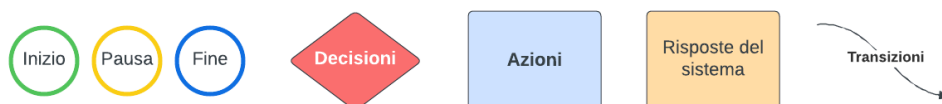
Successivamente viene fornita una breve descrizione per le pagine implementate e una descrizione del repository di GitHub con le istruzioni per effettuare il deployment.

Per finire mostriamo i vari casi di test realizzati per verificare il corretto funzionamento delle APIs.

1. User Flow

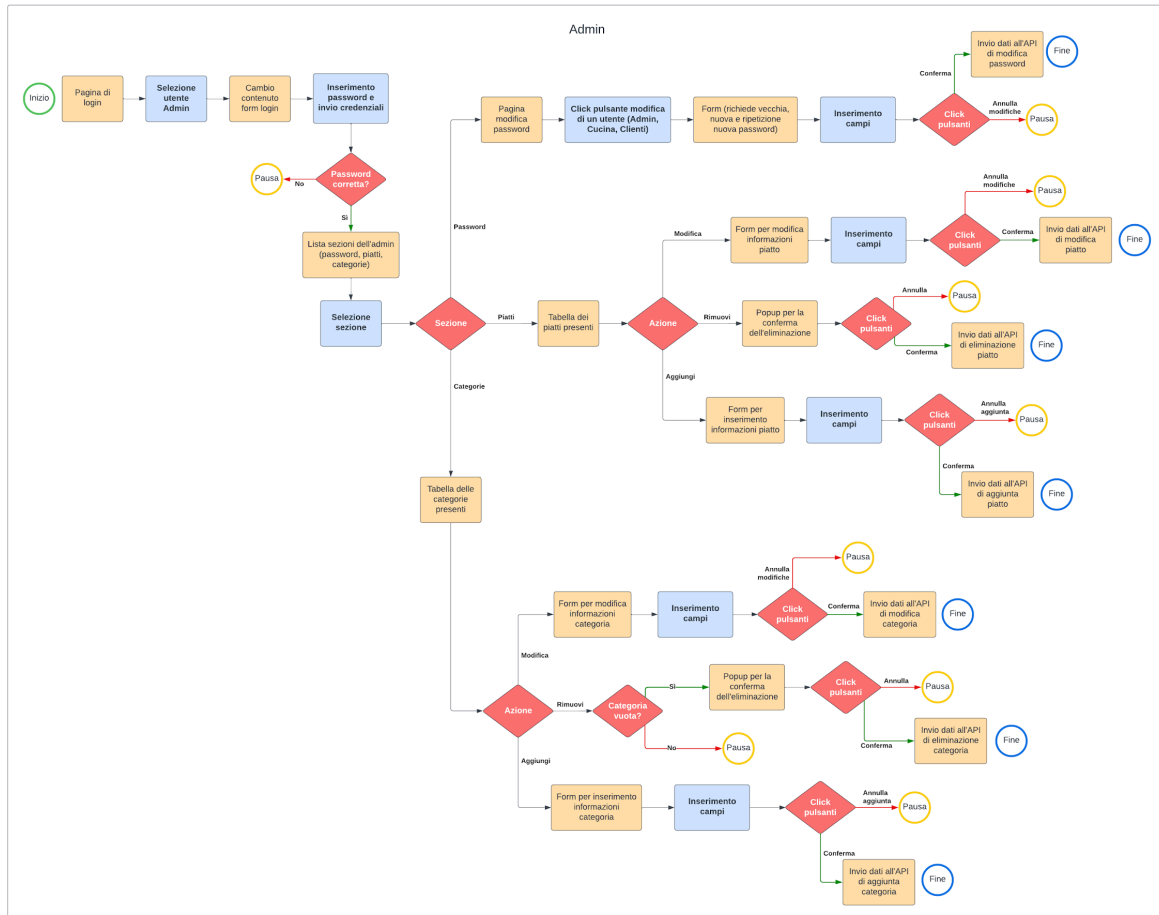
In questa sezione viene riportato lo user-flow dell'applicazione, il quale descrive ciò che è possibile fare nell'implementazione descritta nel dettaglio in questo documento.

Qui sotto è riportata una didascalia dei vari componenti utilizzati nello user-flow.

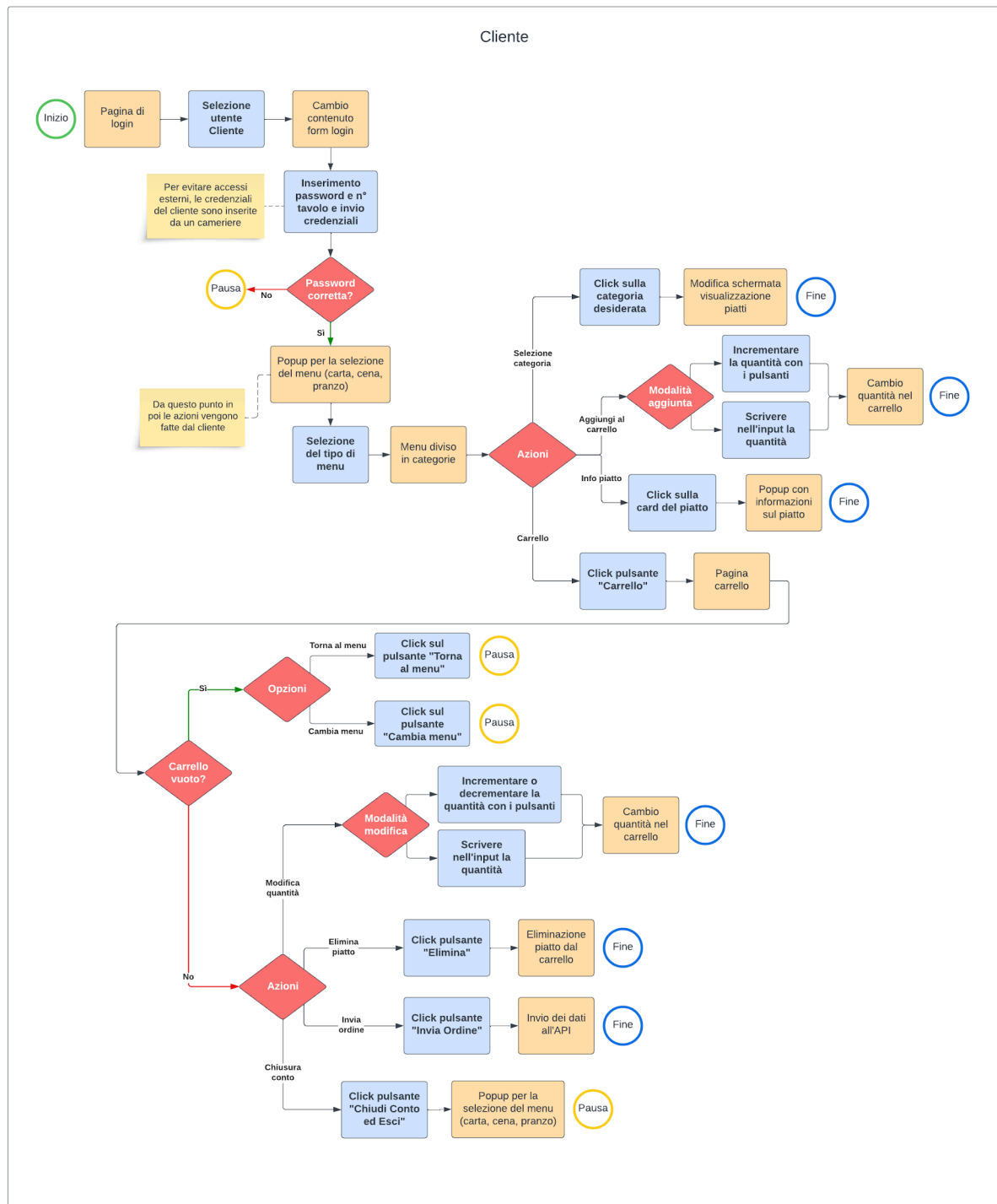


Di seguito sono riportati gli user flow degli utenti e del logout fatti separatamente in modo da rendere più leggibili i diagrammi:

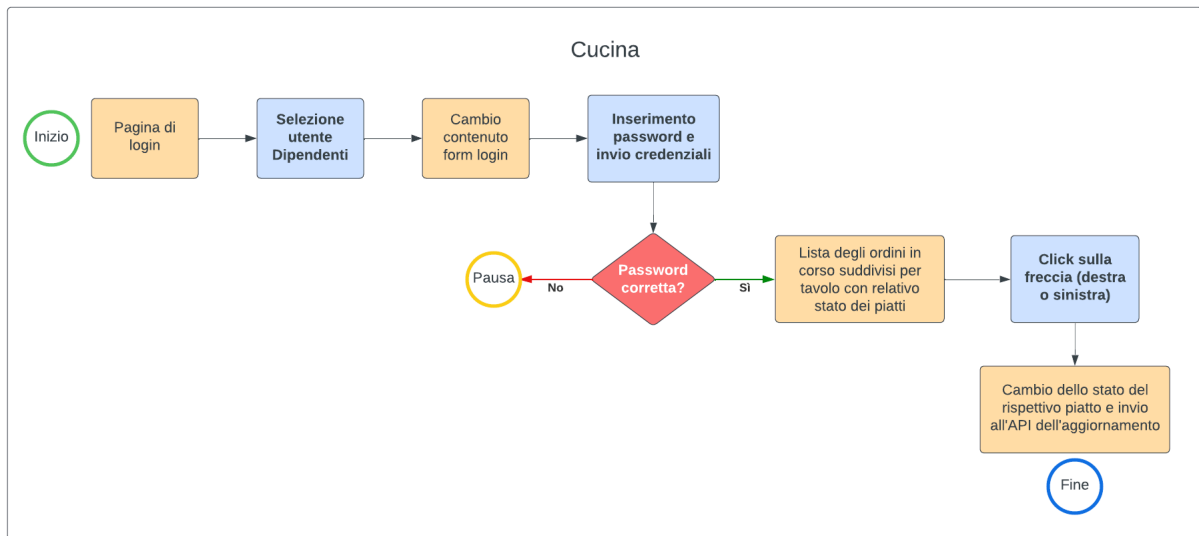
Admin



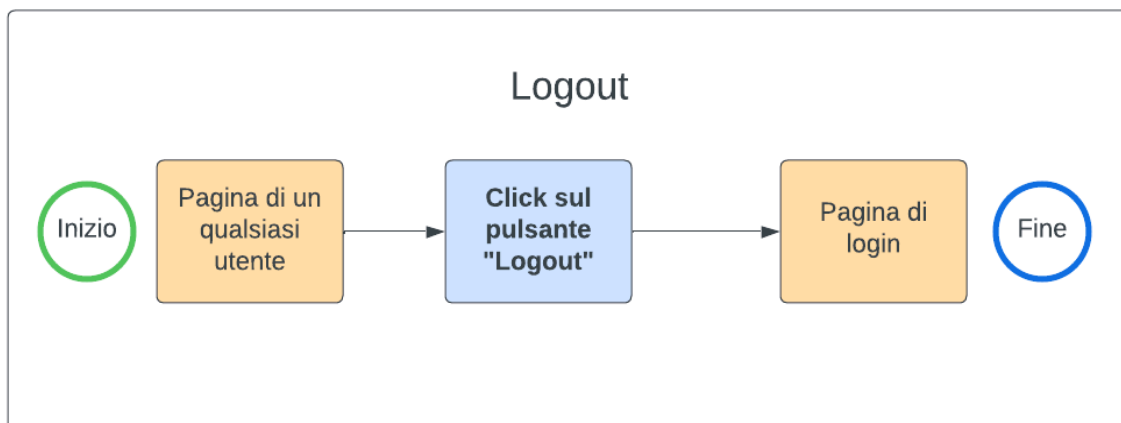
Cliente



Cucina



Logout



2. Implementazione e Documentazione dell'App

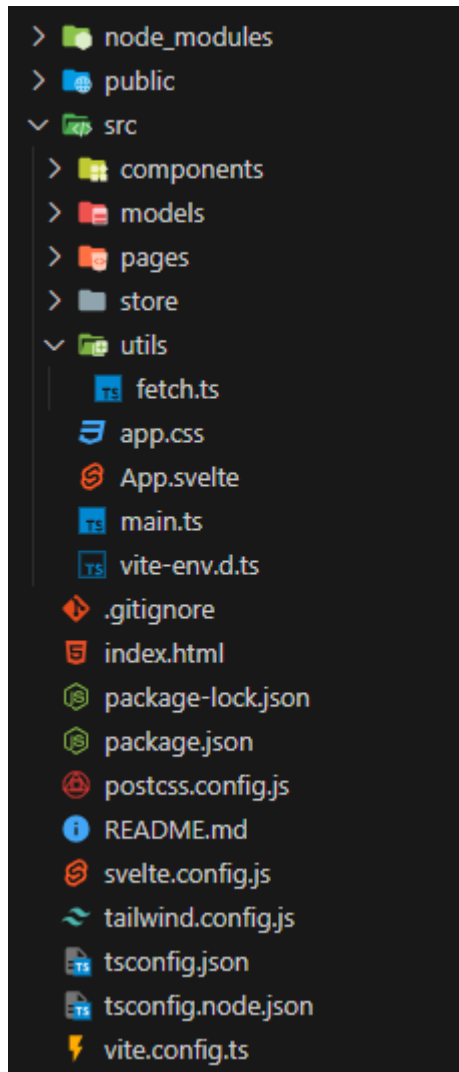
L'applicazione U-Sushi è stata sviluppata utilizzando Svelte per la parte di front-end. A livello di back-end è stato usato GoLang per lo sviluppo di API e PostgreSQL per la memorizzazione dei vari dati.

Come si può vedere dallo user flow, abbiamo sviluppato tutte le parti riguardanti il login, il cambio della password (admin), l'aggiunta, modifica ed eliminazione di un piatto e/o di una categoria, l'invio di un ordine, l'aggiornamento dello stato di un ordine e il logout (solo fittizio con cancellazione del token di sessione).

Struttura del progetto

Di seguito è riportata la struttura del progetto separando front-end e back-end.

Front-end:

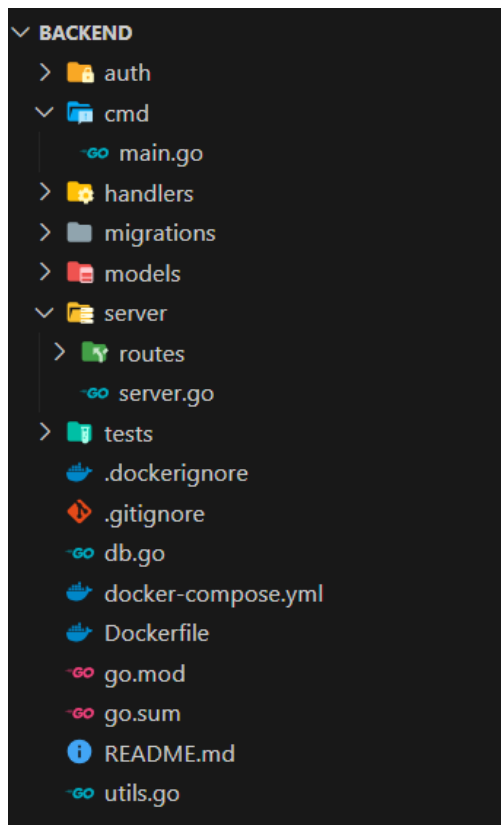


La directory principale del front-end di U-Sushi è costituita dai seguenti elementi:

- cartella `/node_modules` dove vengono scaricate le dipendenze necessarie
- cartella `/public` che contiene il logo e la favicon dell'app
- cartella `/src` che contiene la parte grafica e di logica dell'app e si divide in:
 - cartella `/components` contiene tutti i componenti dell'app in formato `*.svelte`
 - cartella `/models` contiene tipi, enum e variabili usati nei vari componenti
 - cartella `/pages` contiene i componenti principali delle varie pagine dell'app
 - cartella `/store` che permette di mantenere lo stato dell'app (es. carrello) e di dividerlo tra i componenti

- file `/utils/fetch.ts` che espone delle funzioni di supporto per le richieste alle APIs
- file `app.css` che applica lo stile alle pagine
- file `App.svelte`, componente principale dell'app
- file `main.ts` che inizializza l'app svelte
- file `vite-env.d.ts` specifica delle dipendenze di typescript
- file `.gitignore` usato da git per ignorare file o cartelle
- file `index.html`, ovvero il DOM principale dell'app
- file `package.json` e `package-lock.json` che contengono informazioni sulle dipendenze
- file `README.md`, file di markdown di descrizione del progetto
- altri file di configurazione

Back-end:



La directory principale del front-end di U-Sushi è costituita dai seguenti elementi:

- cartella `/auth` contenente i file e le funzioni per gestire l'autenticazione
- file `/cmd/main.go` che contiene la funzione `main()` del backend
- cartella `/handlers` che contiene i vari handler delle richieste HTTP fatte alle APIs, ovvero tutte le funzioni che gestiscono le chiamate e le elaborano
- cartella `/migrations` che contiene i file di migrazione del database

- cartella `/models` contenente i mappings tra relazioni del database e classi di Go
- cartella `/server/routes` che contiene i mappings tra path http e handlers
- file `/server/server.go` che fa partire il server Go del backend
- cartella `/tests` contenente i file di testing delle funzionalità del codice
- file `.dockerignore` e `.gitignore` che dichiarano i file e le cartelle da ignorare rispettivamente da docker e dal VCS di git
- file `db.go` che contiene funzionalità per la connessione al DB
- file `docker-compose.yml` e `Dockerfile` che sono file di configurazione per docker
- file `go.mod` e `go.sum` che contengono le dipendenze e informazioni su di esse
- file `README.md`, file di markdown di descrizione del progetto
- file `utils.go` contiene funzioni "utili" al backend

Dipendenze del progetto

Front-end:

I moduli Node aggiunti al file `package.json` nella sezione "dependencies" sono:

- **flowbite-svelte**: una libreria di componenti grafici che semplifica lo sviluppo di un'interfaccia utente personalizzata in modo rapido.
- **flowbite-svelte-icons**: una libreria di icone vettoriali progettata per migliorare ulteriormente l'aspetto dell'interfaccia utente.
- **svelte-navigator**: implementa il routing per le Single Page Applications (SPA), facilitando la gestione delle route.

NOTA : l'ultima versione di svelte-navigator dà incompatibilità con Vite. Usare `--force` nell'installazione delle dipendenze

Altre dipendenze nel front-end, elencate nella sezione "devDependencies" di `package.json`, includono:

- **tailwindcss**: consente l'applicazione di stili CSS a un elemento tramite classi predefinite, facilitando la gestione dello stile dell'applicazione.
- **typescript**: utilizzato per gli script nei componenti, aggiungendo un supporto avanzato per il tipaggio statico al codice.
- **vite**: un framework JavaScript leggero, utilizzato per lo sviluppo di applicazioni web moderne.

Back-end:

Il file `go.mod` contiene le seguenti dipendenze:

- **gorilla/mux**: Utilizzato come router e dispatcher HTTP, il pacchetto "gorilla/mux" semplifica la gestione delle richieste e delle route nelle applicazioni web Go.
- **jmoiron/sqlx**: Essenziale per semplificare l'interazione con database SQL in Go, "jmoiron/sqlx" estende il pacchetto standard "database/sql" offrendo funzionalità aggiuntive.

- **lib/pq**: Questo modulo fornisce un driver PostgreSQL per Go, consentendo al backend di connettersi e interagire con database PostgreSQL.
- **guregu/null**: La libreria "guregu/null" è inclusa per gestire in modo agevole i valori nulli, particolarmente utili quando si lavora con database che supportano questo concetto.
- **golang-jwt/jwt/v5**: Necessario per la gestione dei JSON Web Tokens (JWT) in Go, questo modulo semplifica la creazione e la verifica di token di autenticazione.
- **golang.org/x/crypto**: Parte del repository "x/crypto" di Go, questo modulo fornisce funzionalità crittografiche essenziali per garantire la sicurezza delle comunicazioni nel backend.
- **stretchr/testify**: Un modulo di testing che semplifica la scrittura di test in Go, aiutando a garantire l'affidabilità del backend attraverso un'adeguata copertura di test.

Relazioni e Tipi nel database

Per gestire i dati presenti nell'app, sono state definite diverse relazioni all'interno di un database relazionale SQL. In alcuni casi si è rivelata necessaria o comunque più funzionale la definizione di tipi di dato personalizzati da applicare ad alcuni campi delle relazioni.

Tipi custom:

1. Durante la fase di login, dal momento che esistono 3 utenti diversi e unici, si è rivelato superfluo l'uso di username personalizzati. Pertanto, al posto dei classici username è stato definito un tipo enumerativo con 3 valori differenti.

```
CREATE TYPE sushi_user_type as ENUM ('Employee', 'Admin', 'Client');
```

2. Un altro tipo enumerativo, è stato definito per il tipo di menù che un cliente va a scegliere, dal momento che anche in questo caso le scelte sono 3 e sono diverse e uniche.

```
CREATE TYPE menu as ENUM ('Carte', 'Dinner', 'Lunch');
```

3. Il terzo tipo rivelatosi utile è quello che indica lo stato di un ordine, anche in questo caso creato per la diversità e unicità dei 4 stati che un ordine può assumere.

```
CREATE TYPE command_status as ENUM ('Ordered', 'Preparing', 'Prepared', 'Delivered');
```

Relazioni:

Passiamo ora all'analisi delle tabelle (o relazioni) individuate per l'app U-Sushi.

1. Partendo sempre dal login, è stata individuata una relazione utile e necessaria alla fase di autenticazione dell'app. La relazione contiene semplicemente le credenziali:
 - l'utente sotto forma di enum specificato nella sezione precedente
 - l'hash della password del relativo profilo

```
CREATE TABLE sushi_user (  
    user_type sushi_user_type PRIMARY KEY,  
    password TEXT NOT NULL  
);
```

2. Si è resa necessaria poi la creazione della tabella che memorizza i piatti prodotti dal ristorante, gestibili dall'admin e visualizzabili dai clienti nella fase di ordinazione.

```
CREATE TABLE plate (  
    id SERIAL PRIMARY KEY,  
    name TEXT NOT NULL,  
    price DECIMAL NOT NULL,  
    category INT NOT NULL REFERENCES category(id),  
    menu menu NOT NULL,  
    description text,  
    image_id INT REFERENCES image(id),  
    pieces INT NOT NULL  
);
```

3. Dalla relazione `plate` si può notare che ci sono 2 campi che referenziano ad altre tabelle:

- **category**: relazione individuata per salvare separatamente le categorie in modo da mantenere la consistenza dei dati. Le categorie inoltre, possono essere aggiunte, rimosse o modificate dall'admin e di conseguenza, una tabella separata offre una gestione più semplice.

```
CREATE TABLE category (  
    id SERIAL PRIMARY KEY,  
    name TEXT UNIQUE  
);
```

- **image**: relazione individuata per avere modo di salvare delle immagini in modo semplice, togliendo una dipendenza da un servizio esterno di storage di immagini in cloud. Le immagini vengono identificate da un id unico e salvate in codifica base64 sotto forma di testo, dal momento che non abbiamo la possibilità di sfruttare una CDN.

```
CREATE TABLE image (  
    id SERIAL PRIMARY KEY,  
    -- image base64 encoding  
    image TEXT NOT NULL  
);
```

4. Un'altra relazione individuata nella definizione dello schema del database è quella riguardante gli ingredienti che compongono i vari piatti.

```
CREATE TABLE ingredient (  
    id SERIAL PRIMARY KEY,  
    name TEXT UNIQUE,  
    allergen INT REFERENCES allergen(id)  
);
```

5. Si può notare che nella tabella `plate` non è presente la voce ingredienti e il motivo sta nel fatto che l'associazione tra le 2 relazioni sia una $N:N$ e, di conseguenza, si è rivelato necessario creare una terza relazione formata dalle chiavi primarie di `plate` e `ingredient`.

```
CREATE TABLE plate_ingredient (  
    plate_id INT REFERENCES plate(id),  
    ingredient_id INT REFERENCES ingredient(id),  
    PRIMARY KEY (plate_id, ingredient_id)  
);
```

6. Come ultima relazione nel contesto dei piatti, come si può notare dalla referenziazione della tabella `ingredient`, è stata individuata la tabella `allergen` adibita appunto all'archiviazione degli allergeni.

```
CREATE TABLE allergen (  
    id SERIAL PRIMARY KEY,  
    name TEXT UNIQUE  
);
```

Nella parte legata alla cucina sono state individuate 2 tabelle:

7. La prima tabella memorizza la sessione di ordinazione di un tavolo.

```
CREATE TABLE session (  
    id SERIAL PRIMARY KEY,  
    start_at TIMESTAMP NOT NULL,  
    end_at TIMESTAMP,  
    table_number INT NOT NULL  
);
```

8. La seconda relazione riguarda invece l'ordine di un singolo piatto inviato dal carrello del cliente.

```
CREATE TABLE command (  
    session_id INT NOT NULL REFERENCES session(id),  
    plate_id INT NOT NULL REFERENCES plate(id),  
    at TIMESTAMP NOT NULL,  
    quantity INT NOT NULL,  
    status command_status NOT NULL,  
    PRIMARY KEY (session_id, plate_id, at));
```

3. APIs del progetto

In questa parte del documento vengono descritte le varie APIs implementate nel progetto. Useremo un diagramma per rappresentare l'estrazione delle risorse a partire dal class diagram e uno per rappresentare le risorse sviluppate.

Estrazione delle risorse dal class diagram

Questo diagramma mostra come sono state estratte le varie risorse sviluppate nel sistema a partire dal diagramma delle classi.

A partire dalle classi presenti abbiamo individuato i tipi di dati che dovevano essere memorizzati nel database preservandone gli attributi fondamentali. Infatti gli attributi specificati nel class diagram sono stati riportati anche nel diagramma delle risorse.

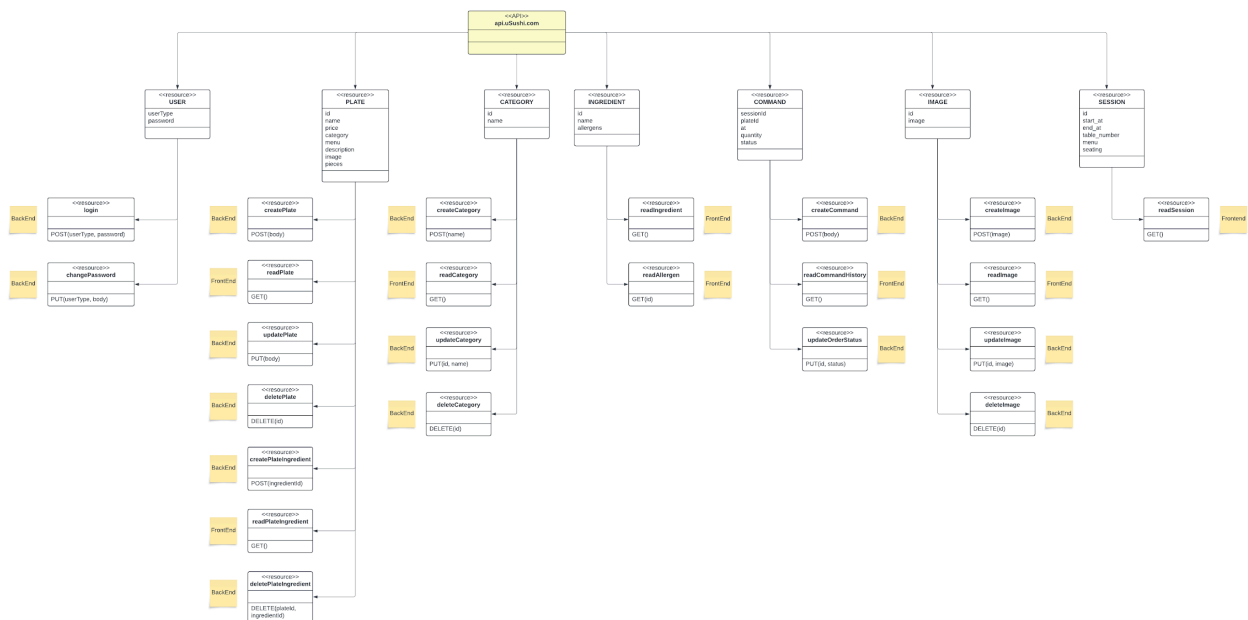
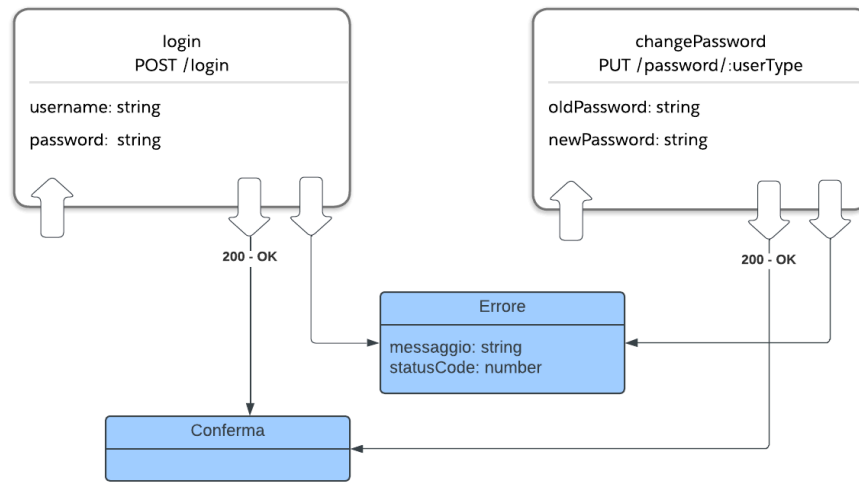


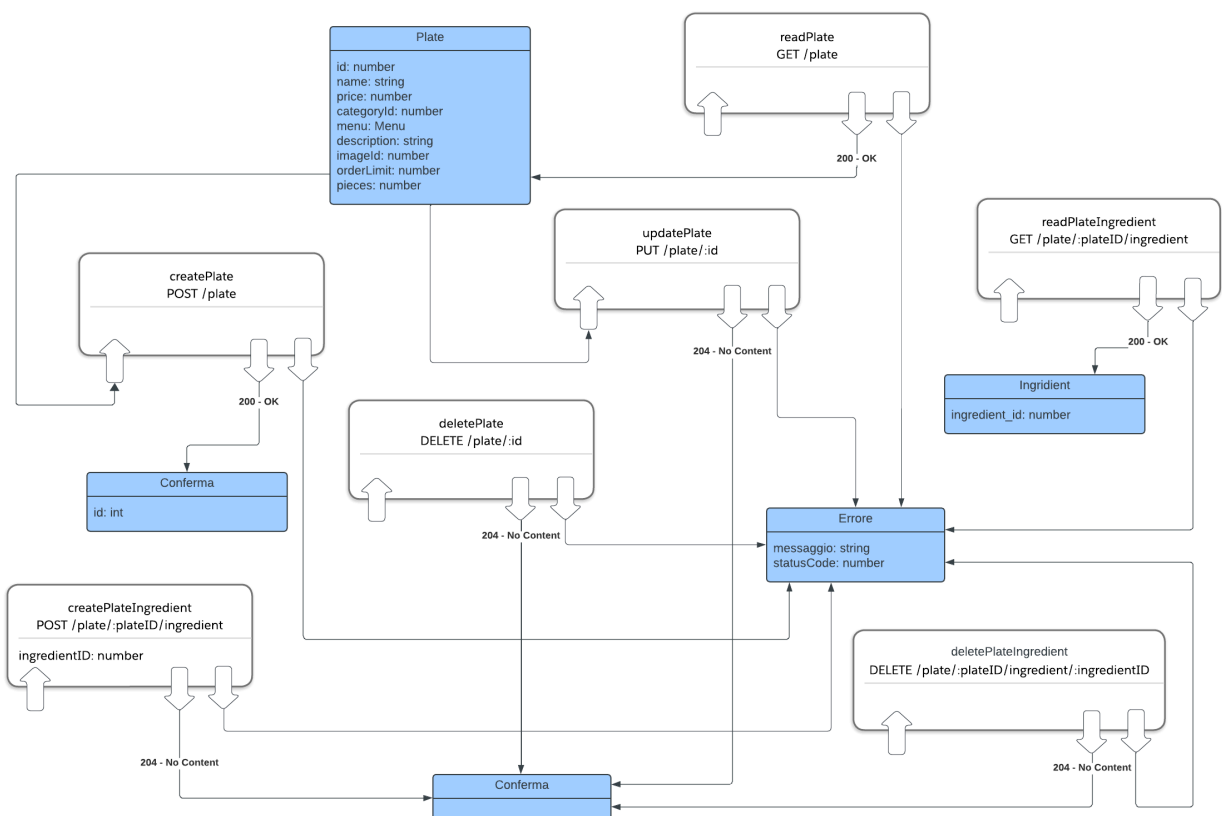
Diagramma delle risorse

Nel seguente diagramma delle risorse rappresentiamo le varie APIs sviluppate. Per permettere una visione più chiara, abbiamo diviso il diagramma delle risorse per contesto. In tutti i diagrammi sono stati specificati gli input e gli output delle varie APIs che possono anche ritornare errori di varia natura. Per quest'ultima affermazione si è deciso di non creare un output per ogni possibile errore ma piuttosto un output comune per tutti gli errori. Infatti in ogni API il messaggio di errore è standardizzato con i campi **statusCode**, per indicare il tipo di errore, e **message**, che spiega cosa è andato storto o non è possibile fare.

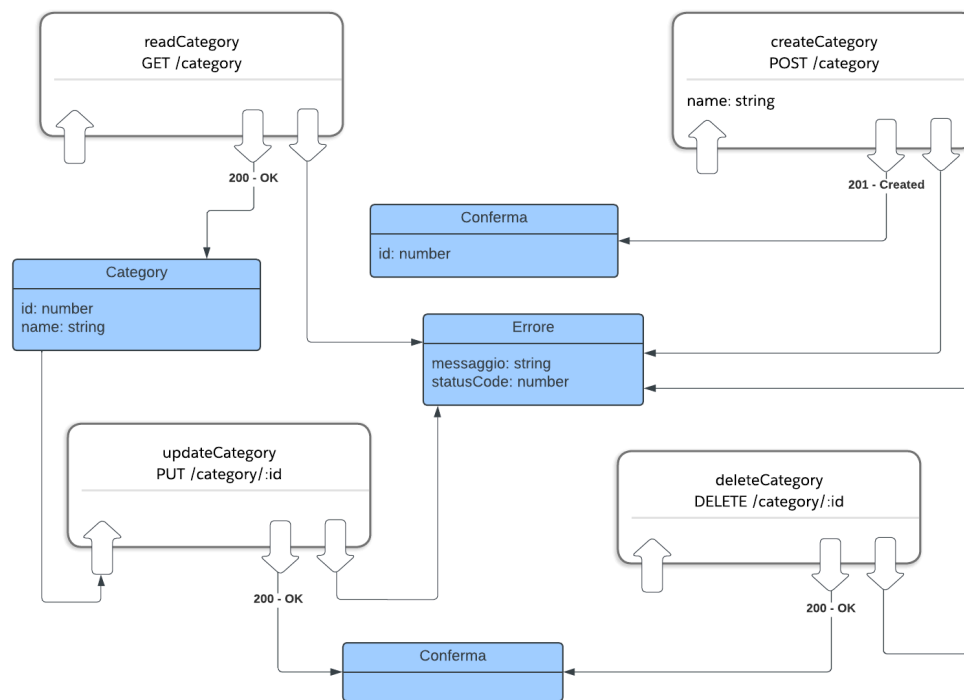
Resource Diagram 1



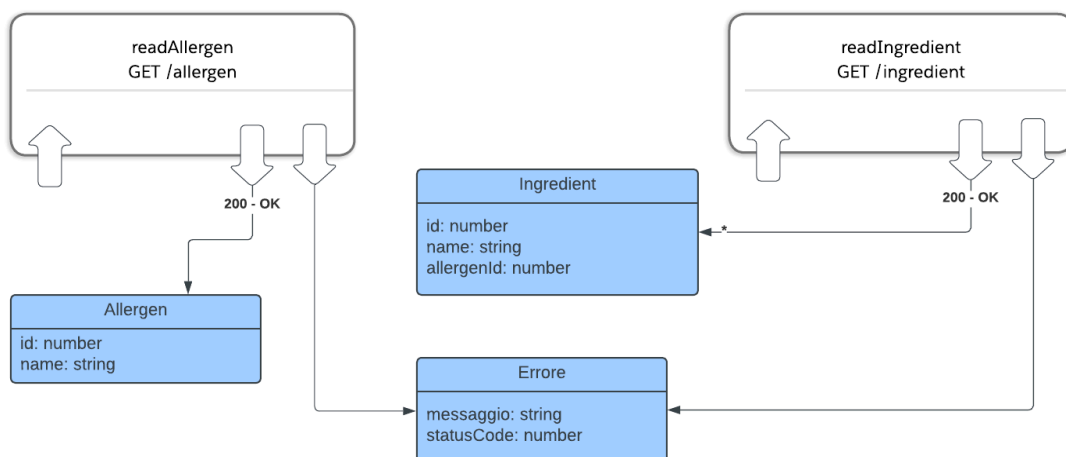
Resource Diagram 2



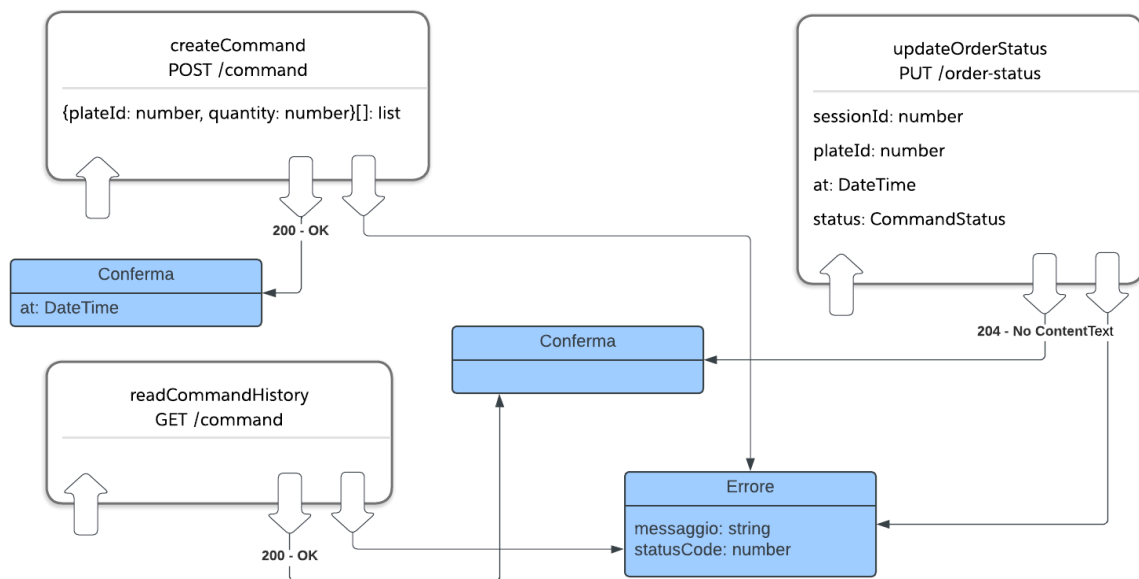
Resource Diagram 3



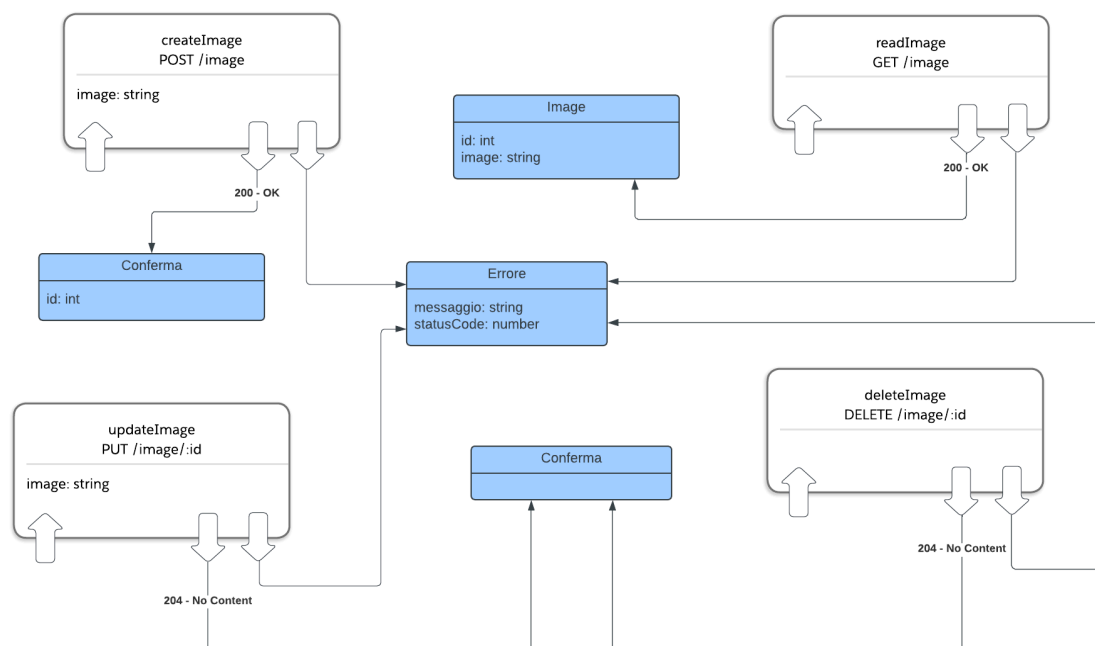
Resource Diagram 4



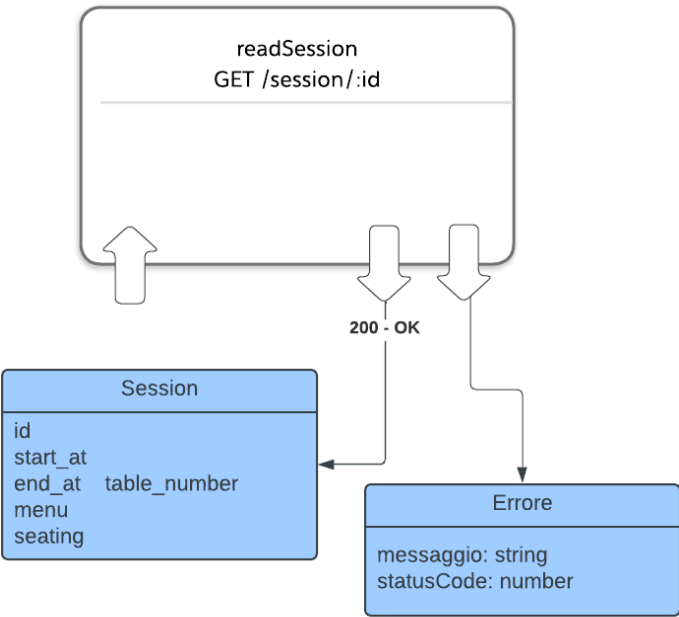
Resource Diagram 5



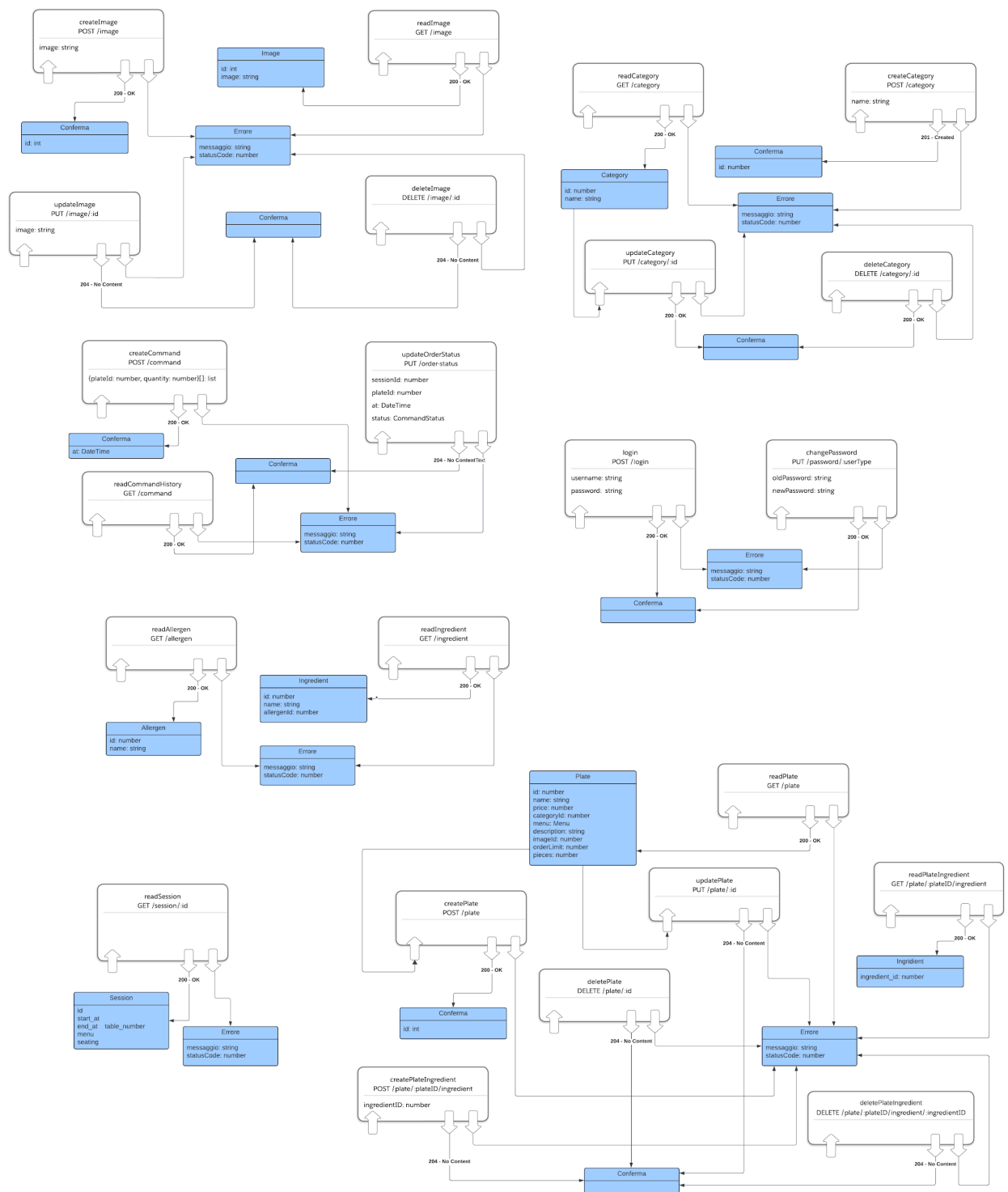
Resource Diagram 6



Resource Diagram 7



Resource Diagram Completo



4. Sviluppo delle APIs

Di seguito viene descritto il funzionamento delle APIs sviluppate nel progetto. Il codice effettivo viene omesso, dal momento che porterebbe a una crescita esagerata del numero di pagine del documento. Inoltre, il codice è interamente disponibile su GitHub e non abbiamo visto la necessità di ripeterlo in questo documento.

APIs relative a *USER*

- Login di un utente

L'API `login` ha lo scopo di autenticare e autorizzare uno dei tre tipi di utente (Admin, Cucina e Clienti). Riceve quindi in input il tipo di utente e la relativa password. Successivamente effettua il controllo sulla correttezza della password:

- in caso di errore, viene ritornato l'errore stesso
- altrimenti, viene inviato un token di accesso, con una validità di una settimana, per chiamare le altre APIs che lo richiedono

- Modifica di una password

L'API `changePassword` viene usata da parte dell'admin e richiede in input il tipo di utente (come parte dell'url) e un body composto da: vecchia password, nuova password. In caso di errore, restituisce l'errore stesso. Nel caso in cui la password venga correttamente modificata, l'API ritorna un codice di successo.

NOTA: l'API può essere chiamata solo con un token di accesso "livello Admin"

APIs relative a *PLATE*

- Aggiunta di un piatto al database

L'API `createPlate` sarà utilizzata dall'admin per aggiungere un piatto al menu.

L'API richiede in input un body formato da tutte le informazioni necessarie, ovvero nome, prezzo, immagine, descrizione, categoria, menu, numero di pezzi. Verrà poi assegnato un id univoco al piatto nella tabella del db che sarà dato come output in caso di successo.

- Modifica di un piatto nel database

L'API `updatePlate` sarà utilizzata dall'admin per modificare un piatto del menu.

L'API richiede in input un body formato da tutte le informazioni necessarie, ovvero nome, prezzo, immagine, descrizione, categoria, menu, pezzi. In caso di corretta modifica ritorna una conferma, altrimenti un errore.

- Recupero dei piatti

L'API **readPlate** viene usata da parte dell'admin nella pagina che mostra la tabella dei piatti da gestire e dai clienti in fase di caricamento del menu. L'API non richiede dati in input e restituisce tutti i piatti presenti nella relazione **Plate**.

- Eliminazione di un piatto dal menu

L'API **deletePlate** viene usata da parte dell'admin nella pagina che mostra la tabella dei piatti da gestire. L'API prende come parametro dell'url l'id del piatto da eliminare e può restituire un codice di successo o uno di errore.

Per quanto riguarda il campo ingredienti di un piatto, essendo implementato nel database come entries di una tabella di associazione molti a molti, si è rivelata utile e quasi necessaria la creazione di tre APIs per aggiungere, togliere e modificare queste associazioni, piuttosto che farlo nell'endpoint della relativa operazione sul piatto.

- Aggiunta di un'associazione piatto-ingrediente

L'API **createPlateIngredient** viene usata da parte dell'admin quando crea un nuovo piatto o ne modifica uno aggiungendo ingredienti. L'API richiede l'id dell'ingrediente nel body e l'id del piatto come parametro nell'url.

- Rimozione di un'associazione piatto-ingrediente

L'API **deletePlateIngredient** viene usata da parte dell'admin quando elimina un piatto o quando ne modifica uno rimuovendo ingredienti. L'API richiede l'id dell'ingrediente nel body e l'id del piatto come parametro nell'url.

- Recupero di un'associazione piatto-ingrediente

L'API **readPlateIngredient** viene usata da parte dell'admin quando vuole visualizzare le info di un piatto da modificare e dal cliente quando vuole visualizzare le info di un piatto nel menu. La risposta contiene l'id e il nome dell'ingrediente e l'id dell'eventuale allergene.

APIs relative a **CATEGORY**

- Aggiunta di una categoria al database

L'API **createCategory** sarà utilizzata dall'admin per aggiungere una categoria del menu nel database. L'API richiede in input il nome della categoria. Verrà poi assegnato un id univoco alla categoria nella tabella del db che sarà poi dato come output in caso di successo.

- Modifica di una categoria nel database

L'API **updateCategory** sarà utilizzata dall'admin per modificare un piatto del menu. L'API richiede in input il nuovo nome nel body della richiesta e l'id della categoria come parametro nell'url. In caso di corretta modifica ritorna una conferma, altrimenti un errore.

- Recupero delle categorie

L'API **readCategory** viene usata da parte dell'admin nella pagina di aggiunta/modifica piatti e nella pagina che mostra la tabella delle categorie da gestire. L'API non richiede dati in input e restituisce tutte le categorie presenti nella relazione **Category**.

- Eliminazione di una categoria

L'API **deleteCategory** viene usata da parte dell'admin nella pagina che mostra la tabella delle categorie da gestire. L'API prende come parametro dell'url l'id della categoria da eliminare e può restituire un codice di successo o uno di errore.

APIs relative a **INGREDIENT**

Di seguito descriviamo solo due APIs: recupero degli ingredienti e recupero degli allergeni. Per motivi di semplicità di implementazione assumiamo ingredienti e allergeni già presenti nel DB e non modificabili/eliminabili.

- Recupero degli ingredienti del ristorante

L'API **readIngredient** sarà utilizzata dall'admin in fase di aggiunta/modifica di un piatto per vedere gli ingredienti disponibili e dal cliente quando vuole visualizzare le info di un piatto. L'API restituisce tutti gli ingredienti presenti nel database.

- Recupero degli allergeni dovuti agli ingredienti del ristorante

L'API **readAllergen** sarà utilizzata dal cliente quando vuole visualizzare le info di un piatto. L'API restituisce tutti gli allergeni presenti nel database.

APIs relative a **COMMAND**

- Creazione di una comanda

L'API **createCommand** sarà utilizzata dal cliente all'invio di un ordine in cucina. L'API richiede in input un body formato da tutte le informazioni necessarie, ovvero id della sessione, id del piatto e quantità. Verrà ritornato un codice di successo o di errore.

- Creazione di una comanda

L'API **readCommandHistory** sarà utilizzata dalla cucina per ricevere gli ordini inviati dai clienti. Non richiede nulla in input e restituisce la lista degli ordini in corso.

- Aggiornamento stato di un ordine

L'API **updateOrderStatus** sarà utilizzata dalla cucina per aggiornare lo stato di un ordine di un cliente (Ricevuto, In corso, Pronto, Servito). In caso di errore, ritorna l'errore stesso.

APIs relative a **IMAGE**

- Aggiunta di un'immagine al database

L'API **createImage** sarà utilizzata dall'admin per aggiungere un'immagine nel database. L'API richiede in input il codice base64 dell'immagine. Verrà poi assegnato un id univoco all'immagine nella tabella del db.

- Modifica di un'immagine nel database

L'API **updateImage** sarà utilizzata dall'admin per modificare un piatto del menu. L'API richiede in input il nuovo codice base64 dell'immagine nel body della richiesta e l'id dell'immagine come parametro nell'url. In caso di corretta modifica ritorna una conferma, altrimenti un errore.

- Recupero delle immagini

L'API **readImage** viene usata da parte dell'admin nella pagina di aggiunta/modifica piatti e dal cliente nel menu. L'API non richiede dati in input e restituisce tutte le immagini presenti nella relazione **Image**.

- Eliminazione di un'immagine

L'API **deleteImage** viene usata da parte dell'admin in fase di eliminazione di un piatto. L'API prende come parametro dell'url l'id dell'immagine da eliminare e può restituire un codice di successo o uno di errore.

APIs relative a **SESSION**

Di seguito descriviamo solo un API: recupero di una sessione di un tavolo da parte della cucina.

- Lettura di una sessione

L'API `readSession` sarà utilizzata dall'employee per farsi restituire l'intera sessione. L'API richiede in input l'id della sessione e restituisce la sessione completa con tutte le informazioni inerenti o un errore.

5. API Documentation

Le APIs sviluppate per l'app U-Sushi sono state documentate con swagger. Il processo poteva essere automatizzato ma per tenere più "pulito" il codice, si è deciso di non aggiungere documentazione sotto forma di commenti e di scrivere la documentazione in file *.yml di swagger scritti a mano.

Per accedere alla documentazione serve clonare la repository GitHub

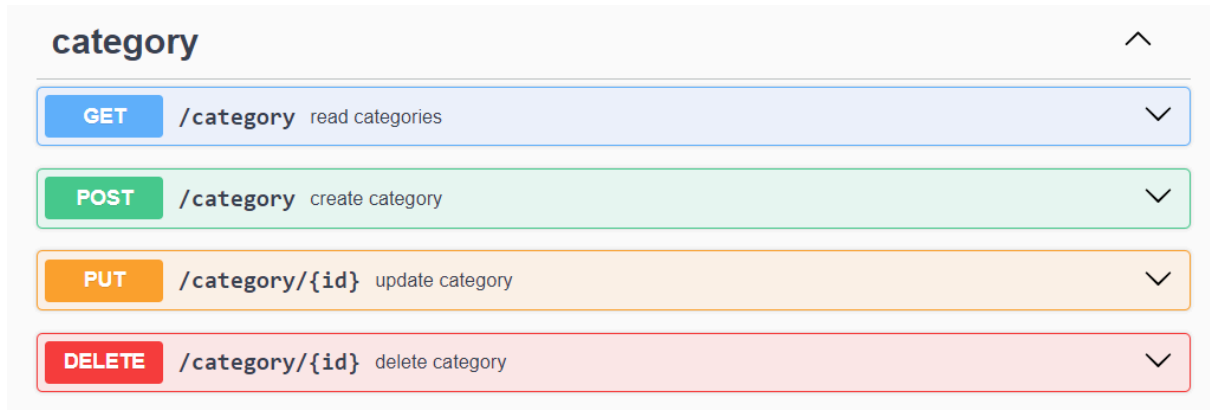
<https://github.com/USushi-G18/swagger-doc> ed eseguire il comando `docker compose up -d`. Successivamente si può visualizzare eseguendo il container docker e accedendo a <http://localhost:80>

La documentazione è divisa in base agli utenti (Admin, Cliente, Cucina), intercambiabili tramite il menù a tendina presente nella pagina in alto a destra.

The screenshot shows the Swagger UI for the 'admin-api' definition. At the top, there's a 'Select a definition' dropdown menu with 'admin-api' selected. Below this, the title 'U-SUSHI api docs admin' is displayed with version tags '0.1.0' and 'OAS 3.0'. A subtitle 'doc/admin.yml' is also present. A description states: 'This page contains the documentation of all the paths of U-SUSHI api for the user admin'. Below the description, there's a 'Servers' section with a dropdown menu showing '/admin'. The main content area lists several API endpoints: 'plate', 'plate_ingredient', 'category', 'image', 'ingredient', 'allergen', and 'auth', each with a dropdown arrow to its right.

Endpoint	Action
plate	▼
plate_ingredient	▼
category	▼
image	▼
ingredient	▼
allergen	▼
auth	▼

L'immagine soprastante rappresenta la prima schermata visualizzata usando uno dei due metodi. Espandendo una sezione tra quelle disponibili si ottengono le APIs legate alla rispettiva risorsa.



Ogni API ha una descrizione (minimale) che fa comprendere cosa comporta la sua chiamata.

Espandendo un endpoint (es. `PUT /category/{id}`) si potranno vedere le seguenti informazioni:

- parametri che vengono presi in input tramite url
- body della richiesta
- possibili risposte da parte dell'endpoint, con relativo body

PUT

/category/{id} update category

update category

Parameters

Try it out

Name	Description
id <small>★ required</small>	category id
integer (path)	<input type="text" value="id"/>

Request body required

application/json

Example Value | Schema

```
{
  "name": "Nigiri"
}
```

Responses

Code	Description	Links
200	OK	No links
400	BadRequest. Usually due to a parsing error	No links
	<div>Media type application/json</div> <div>Example Value Schema</div> <div><pre>{ "error": "json: unable to parse json" }</pre></div>	
500	InternalServerError. This should never happen	No links
	<div>Media type application/json</div> <div>Example Value Schema</div> <div><pre>{ "error": "db: connection refused" }</pre></div>	

6. Testing

La fase di testing è stata effettuata con la libreria venom (<https://github.com/ovh/venom>), un tool per eseguire test da linea di comando. I file che specificano la copertura del codice invece, sono stati ottenuti grazie a go test.

Venom prende in input delle testsuite, ovvero dei file *.yaml che servono a descrivere come un test deve essere eseguito. Di seguito viene presa come esempio la testsuite per il testing delle APIs legate al piatto (plate.yaml):

```
name: test plate

testcases:
  - name: create plate
    steps:
      - type: create
        path: plate
        bodyFile: "models/plate.json"
        assertions:
          - result.bodyjson ShouldContainKey id
        vars:
          id:
            from: result.bodyjson.id
  - name: read plate admin
    steps:
      - type: read
        base_url: "{{.admin_url}}"
        path: plate
  - name: read plate client
    steps:
      - type: read
        base_url: "{{.client_url}}"
        path: plate
  - name: update plate
    steps:
      - type: update
        path: "plate/{{.create-plate.id}}"
        bodyFile: "models/plate.json"
  - name: delete plate
    steps:
      - type: delete
        path: "plate/{{.create-plate.id}}"
```

La testsuite si serve di file ausiliari presenti nella cartella test del backend per specificare il tipo di richiesta, il body della richiesta, l'url di base, ecc.

I test vengono poi eseguiti tramite 2 comandi:

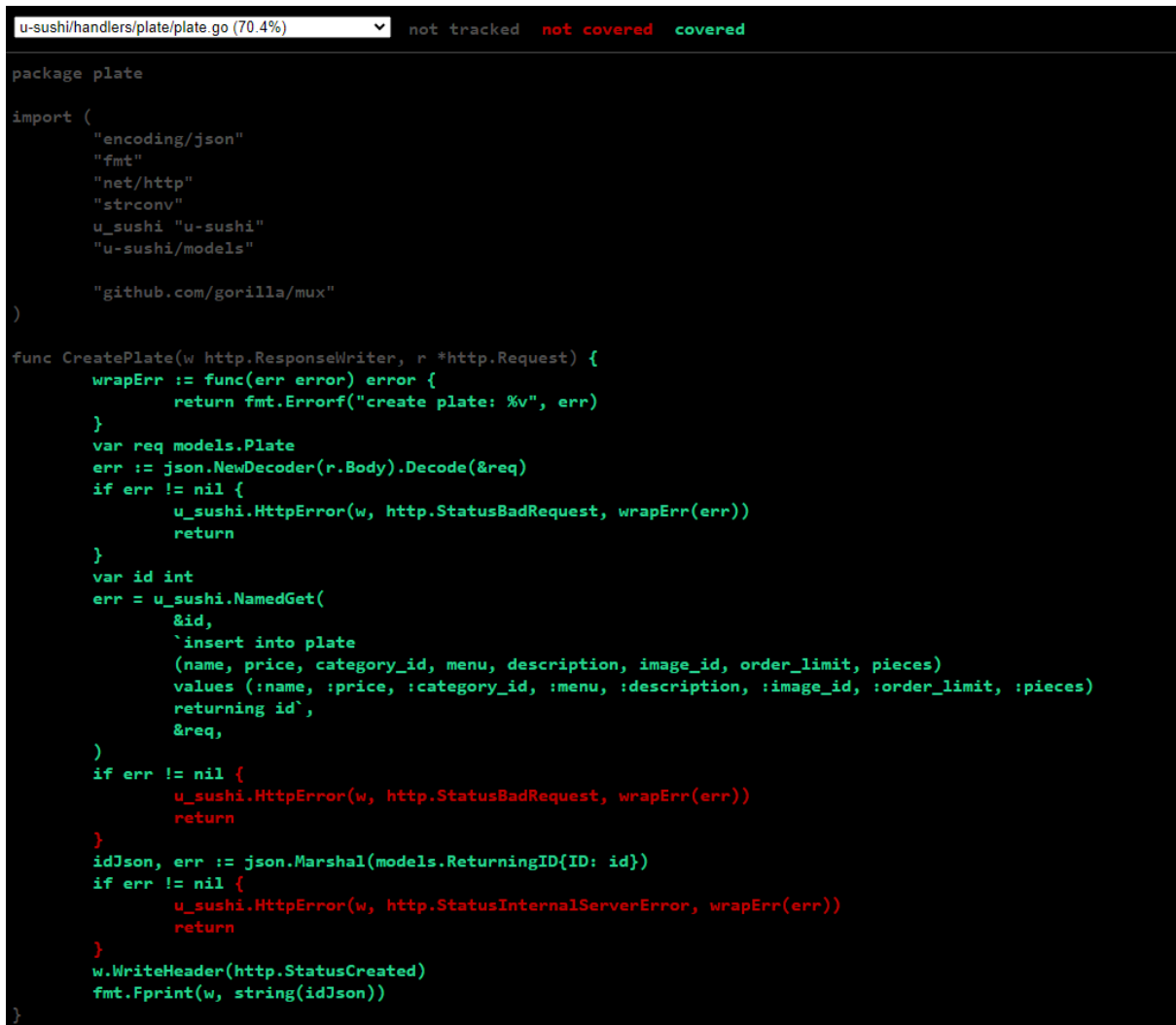
- go test -covermode=set -coverpkg=./... -coverprofile=coverage.out ./tests
- go tool cover -html=coverage.out -o coverage.html

Combinando questi due comandi, vengono prodotti due file:

- coverage.html per il file di copertura

- tests/venom/test_result.html per le testsuite eseguite e passate correttamente

Di seguito viene riportato un esempio dei file generati, focalizzati sulla parte di plate:
coverage.html



```
package plate

import (
    "encoding/json"
    "fmt"
    "net/http"
    "strconv"
    u_sushi "u-sushi"
    "u-sushi/models"
    "github.com/gorilla/mux"
)

func CreatePlate(w http.ResponseWriter, r *http.Request) {
    wrapErr := func(err error) error {
        return fmt.Errorf("create plate: %v", err)
    }
    var req models.Plate
    err := json.NewDecoder(r.Body).Decode(&req)
    if err != nil {
        u_sushi.HttpError(w, http.StatusBadRequest, wrapErr(err))
        return
    }
    var id int
    err = u_sushi.NamedGet(
        &id,
        `insert into plate
        (name, price, category_id, menu, description, image_id, order_limit, pieces)
        values (:name, :price, :category_id, :menu, :description, :image_id, :order_limit, :pieces)
        returning id`,
        &req,
    )
    if err != nil {
        u_sushi.HttpError(w, http.StatusBadRequest, wrapErr(err))
        return
    }
    idJson, err := json.Marshal(models.ReturningID{ID: id})
    if err != nil {
        u_sushi.HttpError(w, http.StatusInternalServerError, wrapErr(err))
        return
    }
    w.WriteHeader(http.StatusCreated)
    fmt.Fprint(w, string(idJson))
}
```

Si può notare che ci sono 3 componenti principali in una scheda della pagina:

1. Un menù a tendina dal quale si può scegliere il file da visualizzare, con relativa percentuale di copertura
2. Legenda che specifica il colore delle parti del codice non tracciate, non coperte e coperte
3. Il codice testato con relative colorazioni in base alla copertura

test_result.html

The screenshot displays the Venom test results interface. On the left, a sidebar titled 'Tests Suites' lists several test suites, each with a 'PASSED' status and a duration. The main area on the right shows the detailed results for the 'test plate' suite, which is also marked as 'PASSED'. This section includes a list of test cases (e.g., 'read-plate-admin', 'read-plate-client', 'create-plate', 'update-plate') and their corresponding steps, each with a 'PASSED' status and a duration. The interface is clean and modern, with a dark header and a light background for the content area.

Questa pagina specifica se i test sono stati passati, quanti ne sono passati, il tempo impiegato e in generale tutte le informazioni relative ad ogni test eseguito. Ci si può spostare di test in test interagendo con la sidebar e si possono visualizzare input, output ed errori di ogni singolo test cliccando sui relativi link.

Entrambi i file si possono trovare nella repository github all'indirizzo <https://github.com/USushi-G18/deliverables>

Attenzione: la fase di testing fa un wipe del database, ovvero cancella tutti i dati per crearne di fittizi e testabili (NON usare nel database in produzione).
Un'altra accortezza da fare è quella di avere nelle variabili d'ambiente le seguenti entry:
DB_CONNECTION_URL=<url_database>
KEY_FILE=<path chiave pem>

Per generare la chiave pem è necessario creare una cartella **/secrets** nella root del progetto ed eseguire il seguente comando nella cartella appena creata:
openssl genrsa -out key.pem 2048

7. GitHub e Deployment

Il progetto è disponibile al link <https://github.com/orgs/USushi-G18/repositories>.
L'organizzazione è divisa in 5 repository: frontend, backend, swagger-doc, deliverables, .github (per aggiungere una descrizione nella home dell'organizzazione).
Il deployment è stato effettuato su Digital Ocean (<https://www.digitalocean.com/>) sia lato frontend che lato backend.

La webapp è disponibile all'URL seguente:

<http://167.172.191.111/>

Esecuzione in locale

Se l'app non fosse più raggiungibile tramite il link fornito precedentemente, di seguito sono elencate le istruzioni per eseguire frontend e backend in locale.

- Frontend:

Per prima cosa è necessario clonare la repo GitHub del frontend tramite il link

<https://github.com/USushi-G18/frontend.git>.

A clonazione terminata eseguire (dentro alla directory) il comando `npm install --force` per installare le dipendenze necessarie.

NOTA: l'opzione **force** si rende necessaria a causa di un warning dato da svelte-navigator in relazione con Vite

Al termine dell'installazione delle dipendenze, eseguire il comando `npm run dev`

Il sito si può visitare all'url <http://localhost:5173> (può variare la porta, seguire le istruzioni sul terminale).

NOTA: per raggiungere le api in locale sarà necessario cambiare tutti gli url nei fetch usati nel codice

- Backend:

Per il backend è necessario avere Docker installato. Soddisfatto questo requisito è necessario clonare la repository tramite il link

<https://github.com/USushi-G18/backend.git>. È necessario poi generare una chiave pem e per farlo basta creare una cartella `/secrets` nella root del progetto ed

eseguire il seguente comando nella cartella appena creata:

`openssl genrsa -out key.pem 2048`.

Generata la chiave, basta eseguire (all'interno della cartella principale del progetto) il comando `docker compose up -d --build`.

Le APIs esposte dal backend si possono raggiungere sul localhost e sulle porte 8081 per l'admin e 8082 per i clienti e la cucina:

- Admin -> <http://localhost:8081/admin>
- Clienti -> <http://localhost:8082/client>
- Cucina -> <http://localhost:8082/employee>