



Progetto:

Baccalà

Titolo del documento:

Sviluppo applicazione

Scopo del documento

Il presente documento espone il processo di sviluppo del software Baccalà, in particolare approfondisce le seguenti sezioni:

- lo user flow degli utenti gestore e consumatore;
- lo sviluppo del back-end;
- la documentazione delle API;
- lo sviluppo del front-end;
- il deployment del software;
- il testing delle API.

Indice

[User Flow](#)

[Sviluppo del back-end](#)

[Struttura del DataBase](#)

[Estrazione delle risorse dal diagramma delle classi](#)

[Modelli delle risorse](#)

[Sviluppo API](#)

[Documentazione API](#)

[Sviluppo del Front-end](#)

[Deployment in locale](#)

[Testing](#)

User Flow

In questa sezione del documento vengono riportati i diagrammi user flow relativi agli utenti gestore (fig.1.a - 1.b) e consumatore (fig.2).

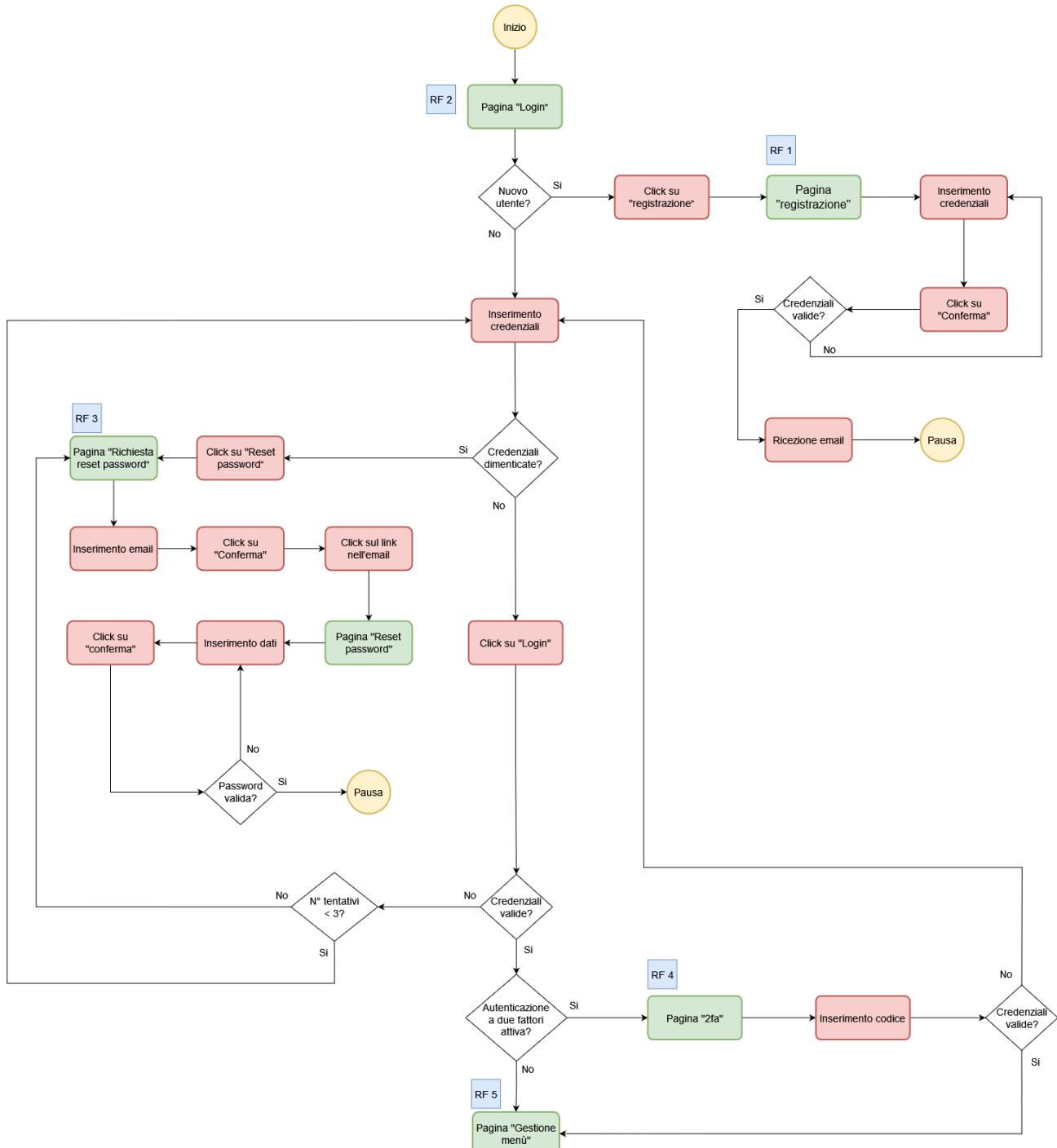


Figura 1.a: user flow dell'utente gestore (parte 1)

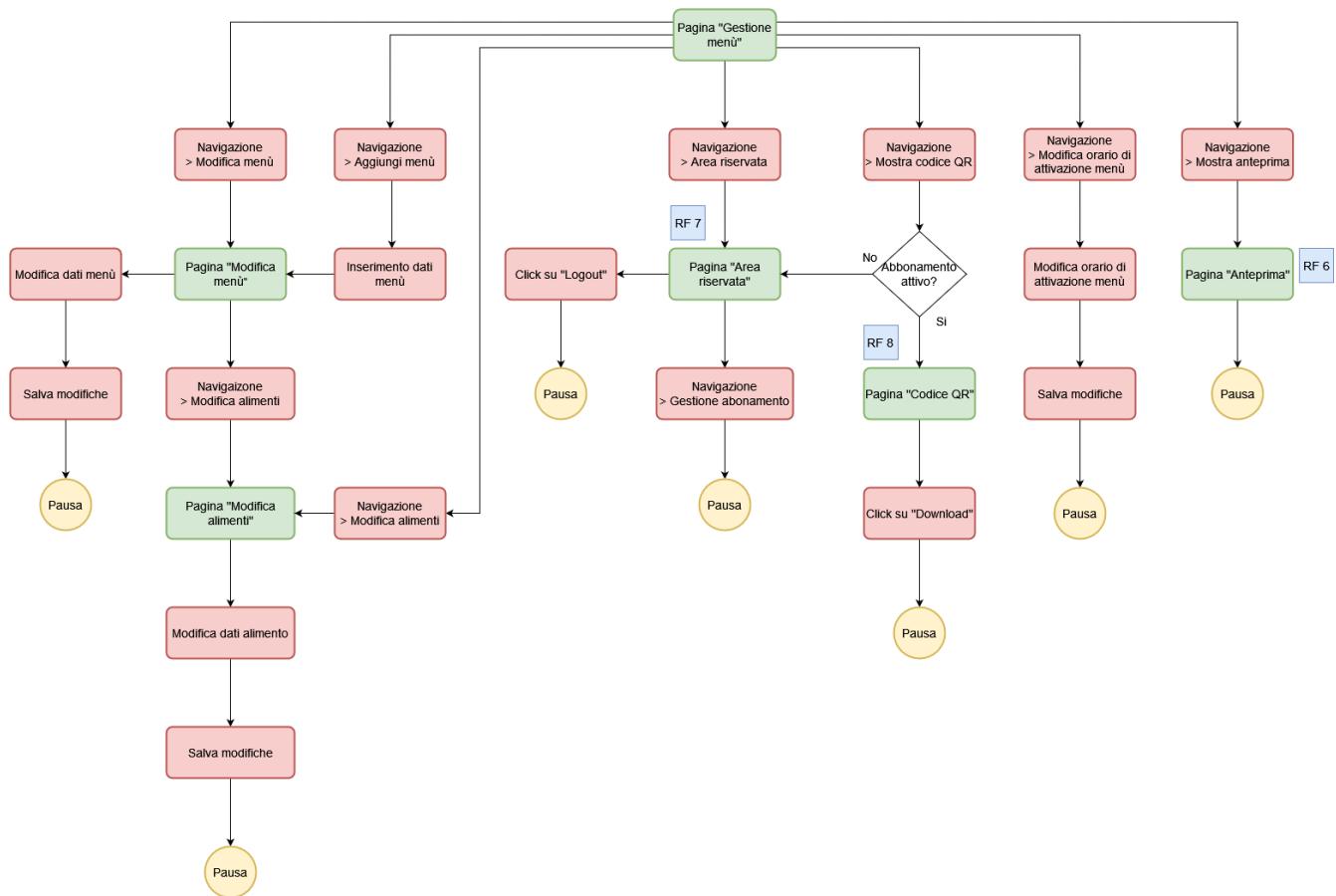


Figura 1.b: user flow dell'utente gestore (parte 2)

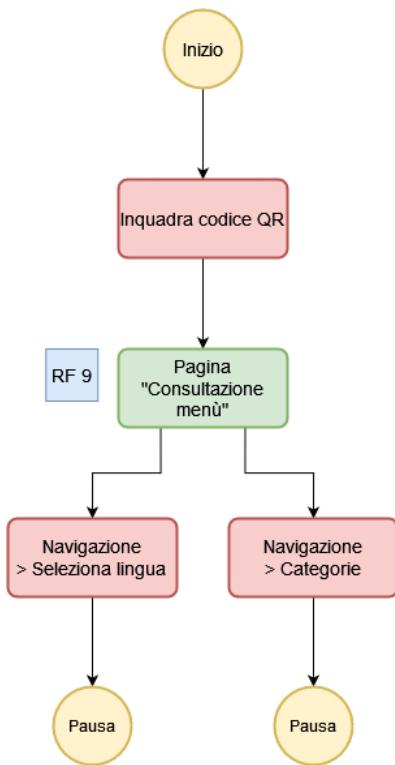


Figura 2: user flow dell'utente consumatore

Sviluppo del back-end

La struttura del back-end è riportata nella figura 3. All'interno della cartella `src/` è presente `app.js`, l'entry point del programma, e 5 sottocartelle:

- `models/` contiene le definizioni delle collection nel database (v. [Struttura del DataBase](#)) e permette di interfacciarsi con esse;
- `controllers/` contiene le funzioni che saranno chiamate quando saranno ricevute richieste alle API;
- `routes/` contiene le corrispondenze tra le funzioni in `controllers/` e i percorsi delle API (per esempio `createUser` è collegata a `POST /api/user`);
- `tests/` contiene le test suites (v. [Testing](#));
- `functions/` contiene alcune funzioni che sono state riutilizzate più volte nel progetto;

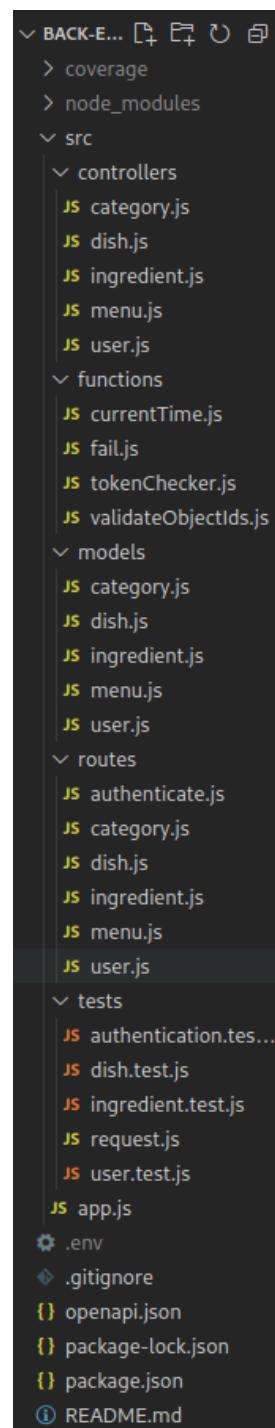


Figura 3: struttura del progetto del back-end

Dipendenze

Il back-end dipende dai seguenti moduli nodejs:

- argon2: algoritmo per l'hashing delle password;
- bent: libreria per le richieste HTTP, utilizzata per il testing delle API;
- cors: il cross-origin resource sharing permette di accettare le richieste da altri domini, in particolare il front-end;
- dotenv: permette di impostare variabili di ambiente attraverso un file ".env"
- express: web server;
- jest: framework per il testing;
- jsonwebtoken: implementazione di JWT per nodejs;
- mongoose: interfaccia con base di dati mongodb;
- swagger-ui-express: genera una pagina web per la documentazione delle API descritte attraverso la specifica OpenAPI;

Struttura del DataBase

Il progetto utilizza il database non relazionale MongoDB.

Sono presenti 5 collection: *users*, *menus*, *dishes*, *ingredients* e *categories*.

```
_id: ObjectId('63e1272db590ba7110e19bbc')
email: "mario.bianchi@unitn.it"
business_name: "Da Mario"
password_hash: "$argon2id$v=19$m=65536,t=3,p=4$s5DxCyUdp4V1j5z5ph1J6A$ohFwYLYkN1/v6PsP..."
enable_2fa: false
```

Figura 4: contenuto di un documento in *users*

```
_id: ObjectId('63e22b3bca998f0bdd847da4')
owner_id: ObjectId('63e1272db590ba7110e19bbc')
name: "Cena"
dishes: Array
  0: ObjectId('63e132e40a26f0e68476d12e')
  1: ObjectId('63e22397fff13cd5b75456f2')
end_time: 1380
start_time: 1050
```

Figura 5: contenuto di un documento in *menus*

```
_id: ObjectId('63e22397fff13cd5b75456f2')
owner_id: ObjectId('63e1272db590ba7110e19bbc')
name: "Pizza Patatosa"
description: "La preferita dello chef"
image: BinData(0, 'ZGF0YTppbWFnZS9qcGVnO2Jhc2U2NCwv
▼ ingredients: Array
  0: ObjectId('63e12d837d44de0589a26946')
  1: ObjectId('63e12d8a7d44de0589a26949')
  2: ObjectId('63e12d8f7d44de0589a2694c')
  3: ObjectId('63ea51544bcddeebd81247cf')
▼ categories: Array
  0: ObjectId('63e1500a4bd8138b8d8f84c3')
```

Figura 6: contenuto di un documento in *dishes*

```
_id: ObjectId('63e12d837d44de0589a26946')
owner_id: ObjectId('63e1272db590ba7110e19bbc')
name: "mozzarella"
```

Figura 7: contenuto di un documento in *ingredients*

```
_id: ObjectId('63e1500a4bd8138b8d8f84c3')
owner_id: ObjectId('63e1272db590ba7110e19bbc')
name: "vegetariano"
```

Figura 8: contenuto di un documento in *categories*

Estrazione delle risorse dal diagramma delle classi

Il seguente schema descrive le API messe a disposizione dal back-end.

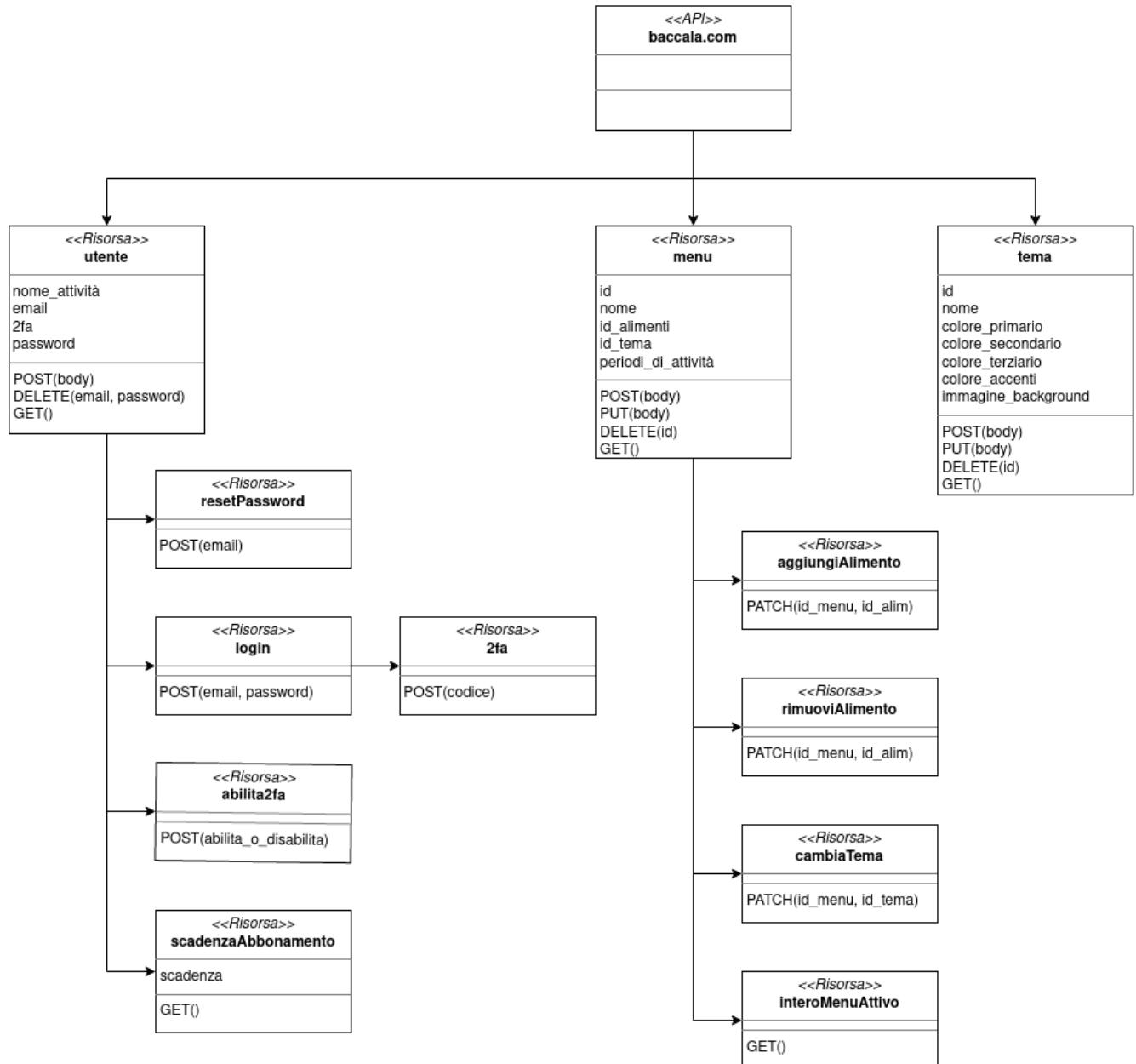


Figura 9: schema di estrazione delle risorse (parte 1)

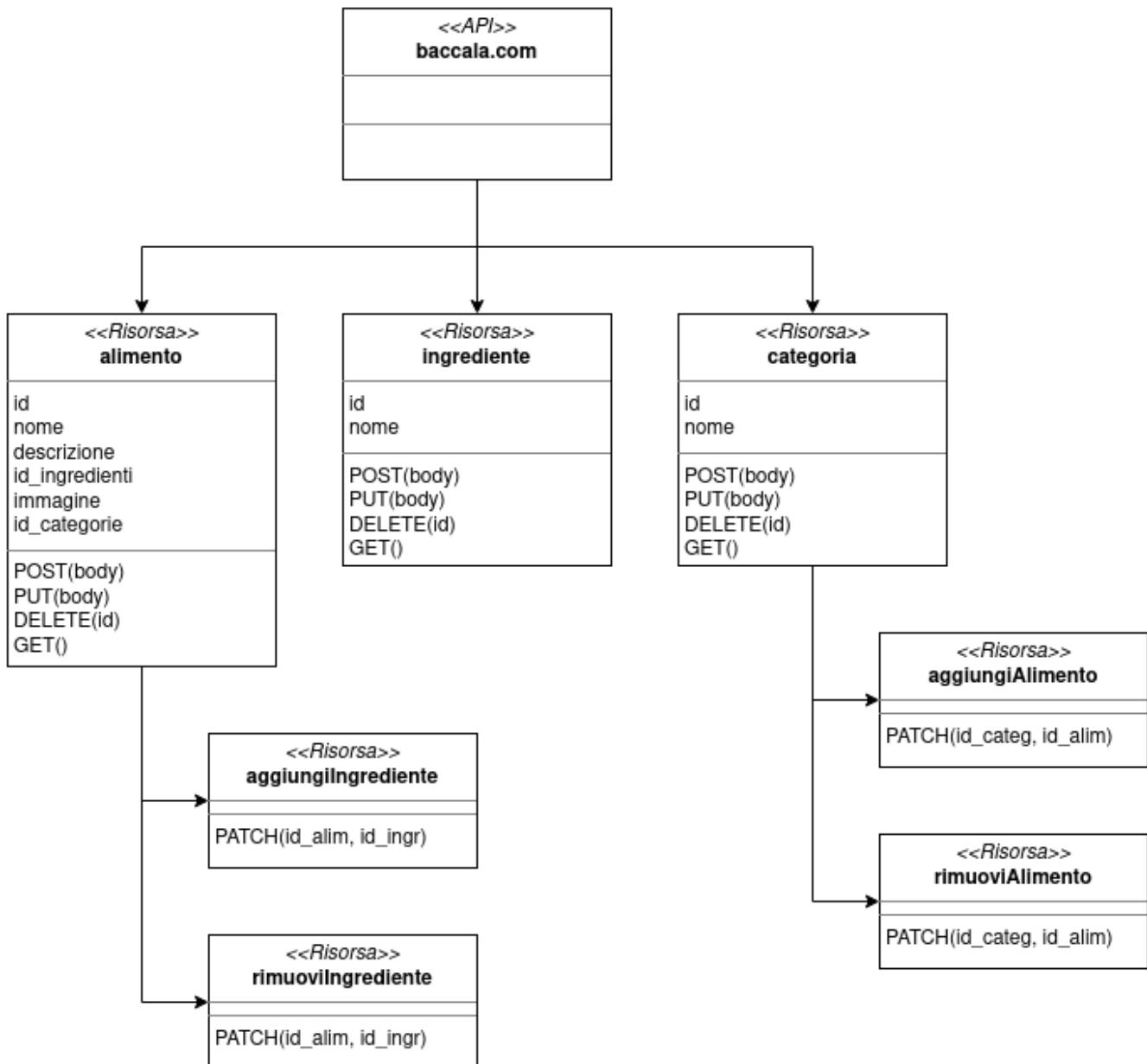


Figura 10: schema di estrazione delle risorse (parte 2)

Modelli delle risorse

A seguire i diagrammi resource model per le API messe a disposizione dal back-end.

Dove non specificato diversamente queste sono protette, ovvero necessitano della presenza di un JWT (JSON Web Token) firmato dal server nell'Authentication header; in caso questo sia assente o invalido rispondono con un errore 401 (Unauthorized).

L'ottenimento di questo JWT può avvenire in 2 modi:

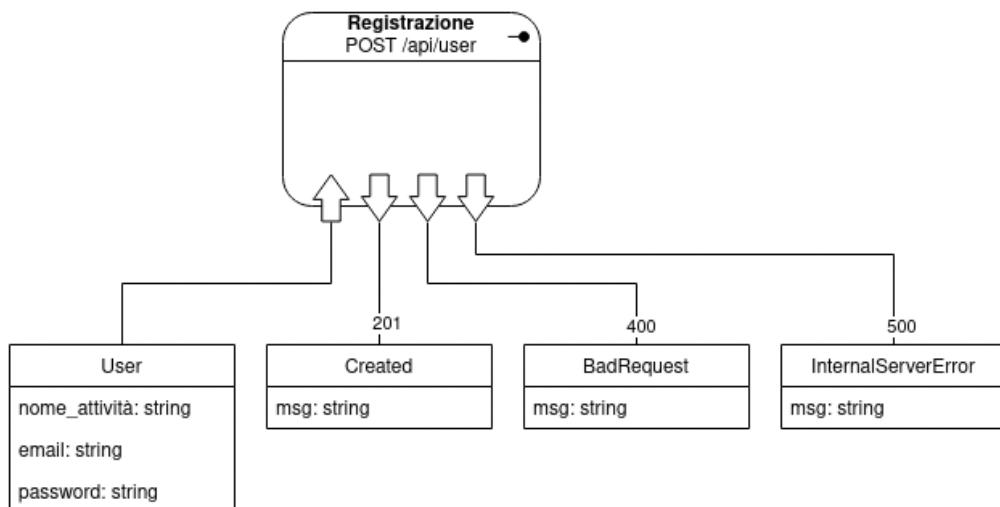
- se l'utente non ha abilitato la 2FA (autenticazione a 2 fattori) il JWT per accedere alle API protette si ottiene attraverso la API **Login** (POST /api/user/login).
- altrimenti il JWT ritornato da **Login** non permetterà l'accesso a nessuna API protetta a parte **2fa** (POST /api/user/login/2fa). Per ottenere un JWT che permetta l'accesso alle API protette sarà necessaria una richiesta a **2fa** contenente:
 - il JWT precedentemente ottenuto da **Login** nell'Authorization header;
 - il codice ricevuto per email nel body.

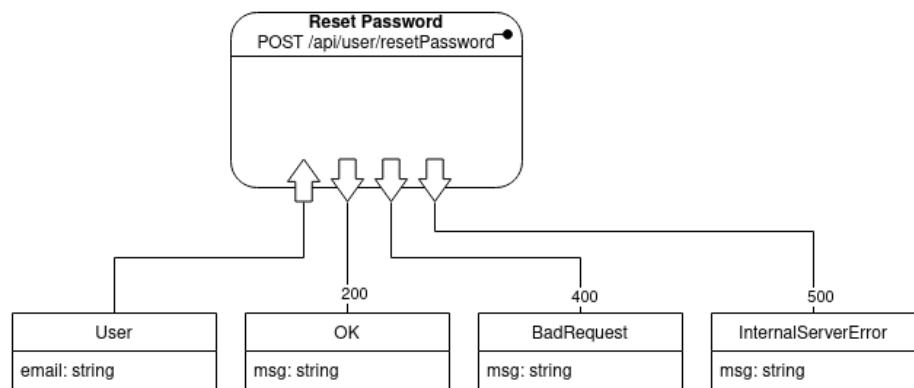
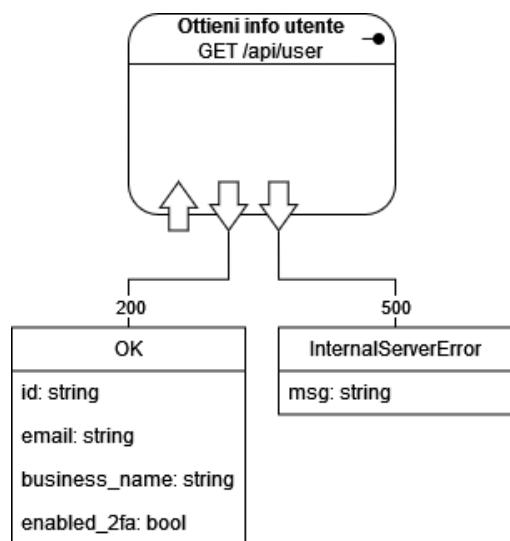
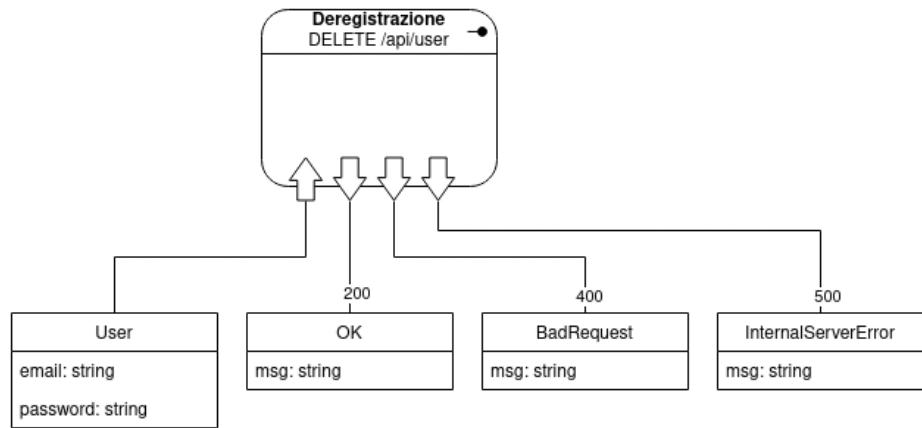
Un JWT ottenuto con uno dei metodi sopra elencati non permetterà l'accesso a **2fa**.

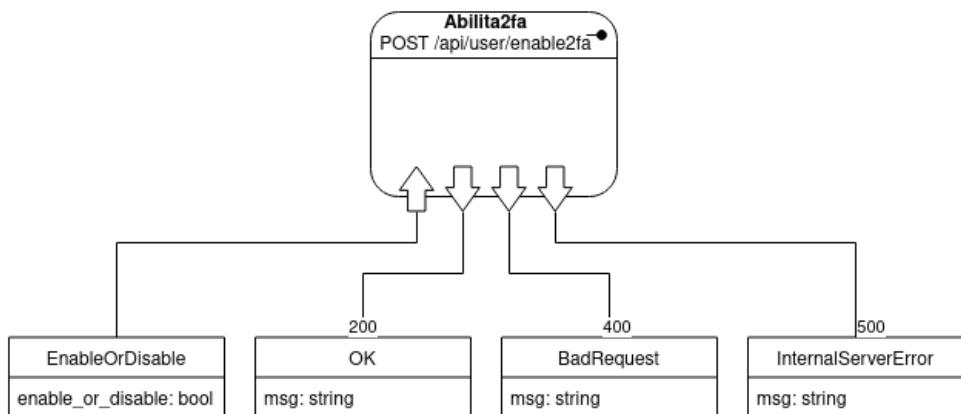
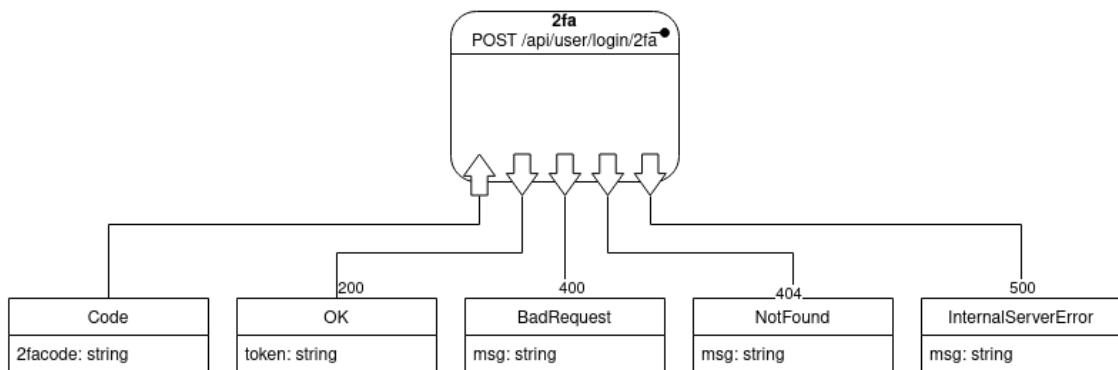
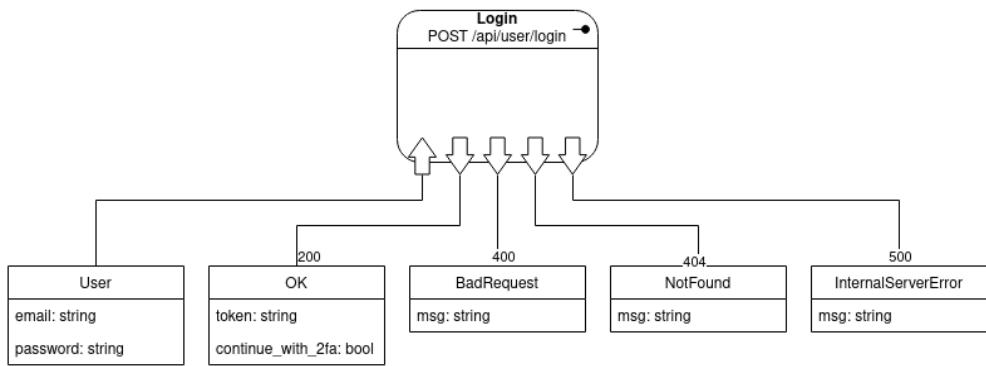
Utente

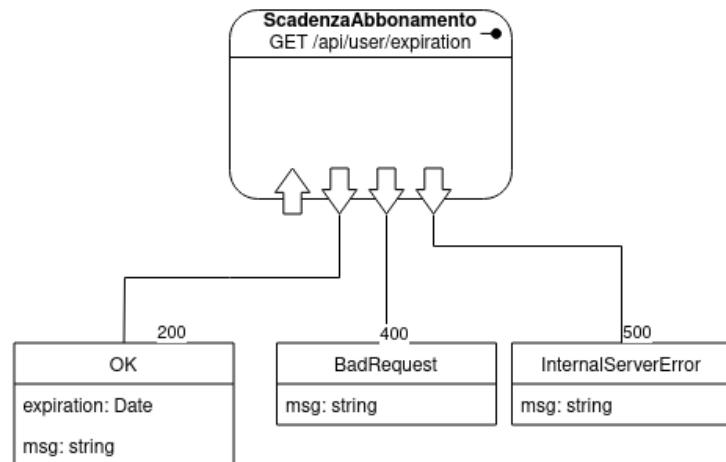
Le seguenti API riguardano l'utente gestore e forniscono le seguenti funzionalità: creazione ed eliminazione di un account, ottenimento delle informazioni relative ad un account, reset della password, login, login con 2fa abilitato, abilitazione del 2fa, ottenimento della data di scadenza dell'abbonamento.

Le API **Registrazione**, **Reset Password** e **Login** sono accessibili senza un JWT.



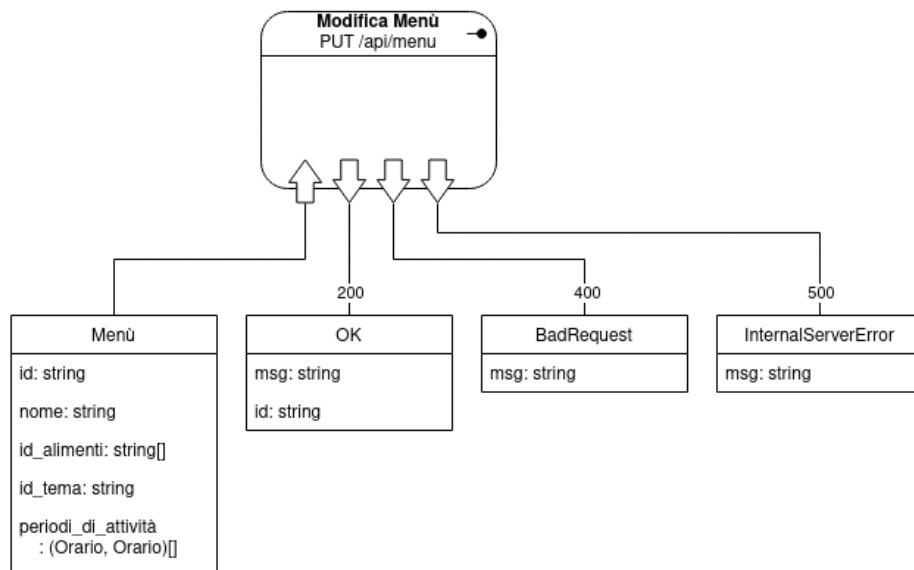
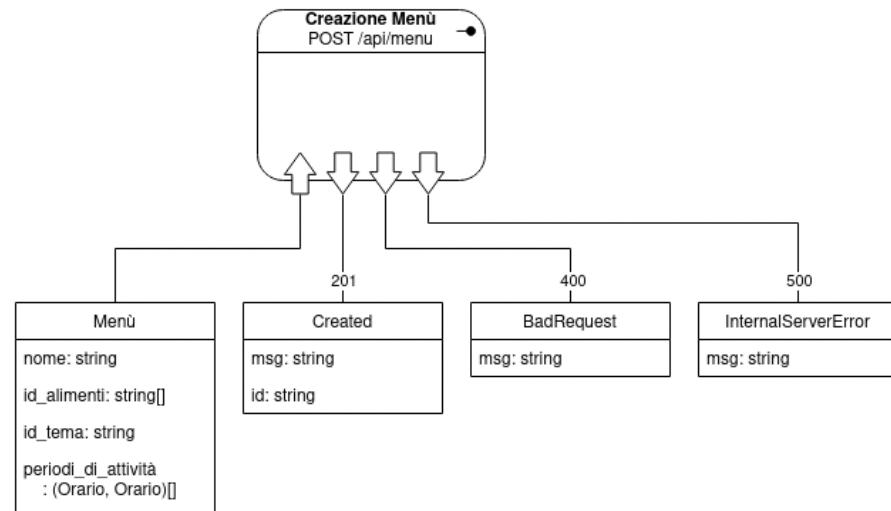


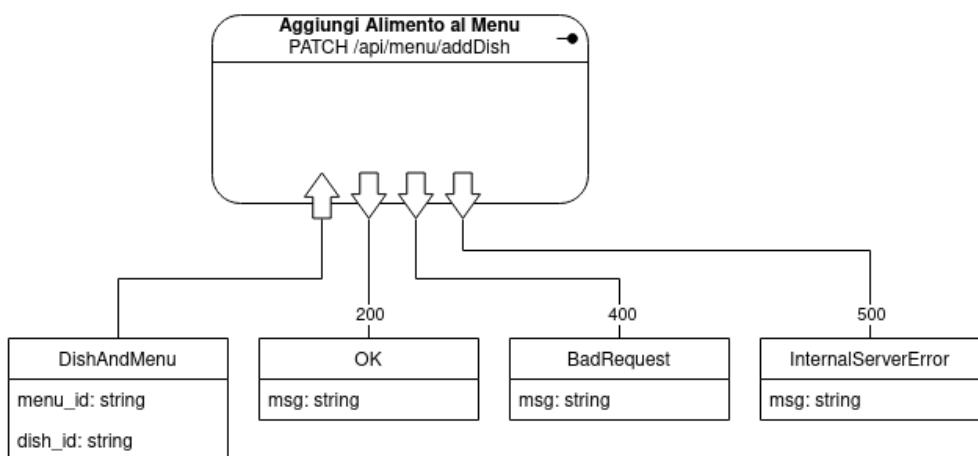
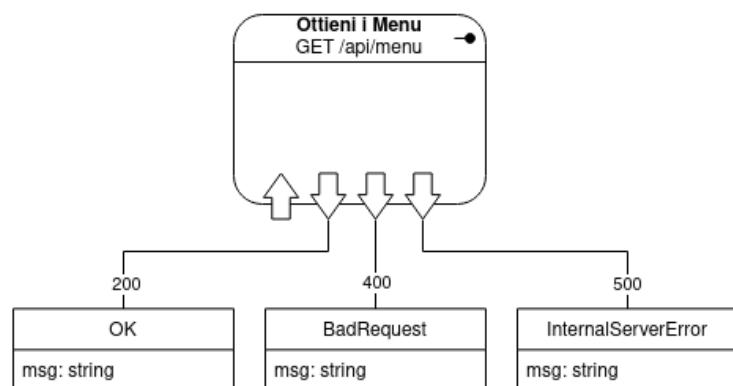
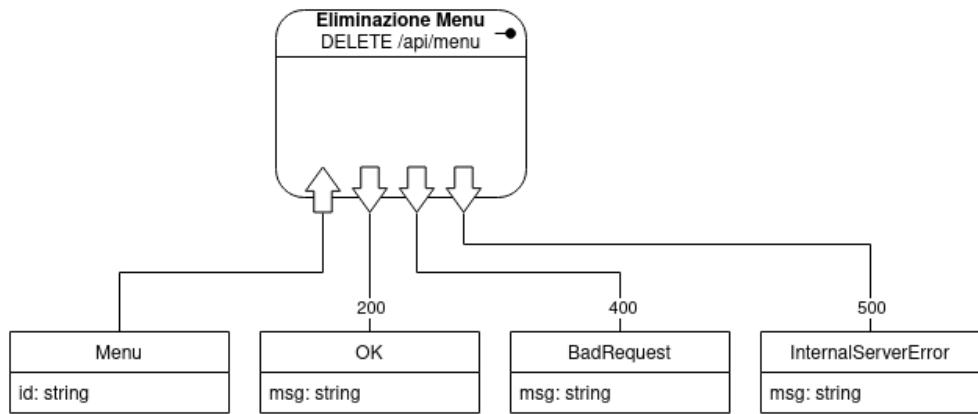


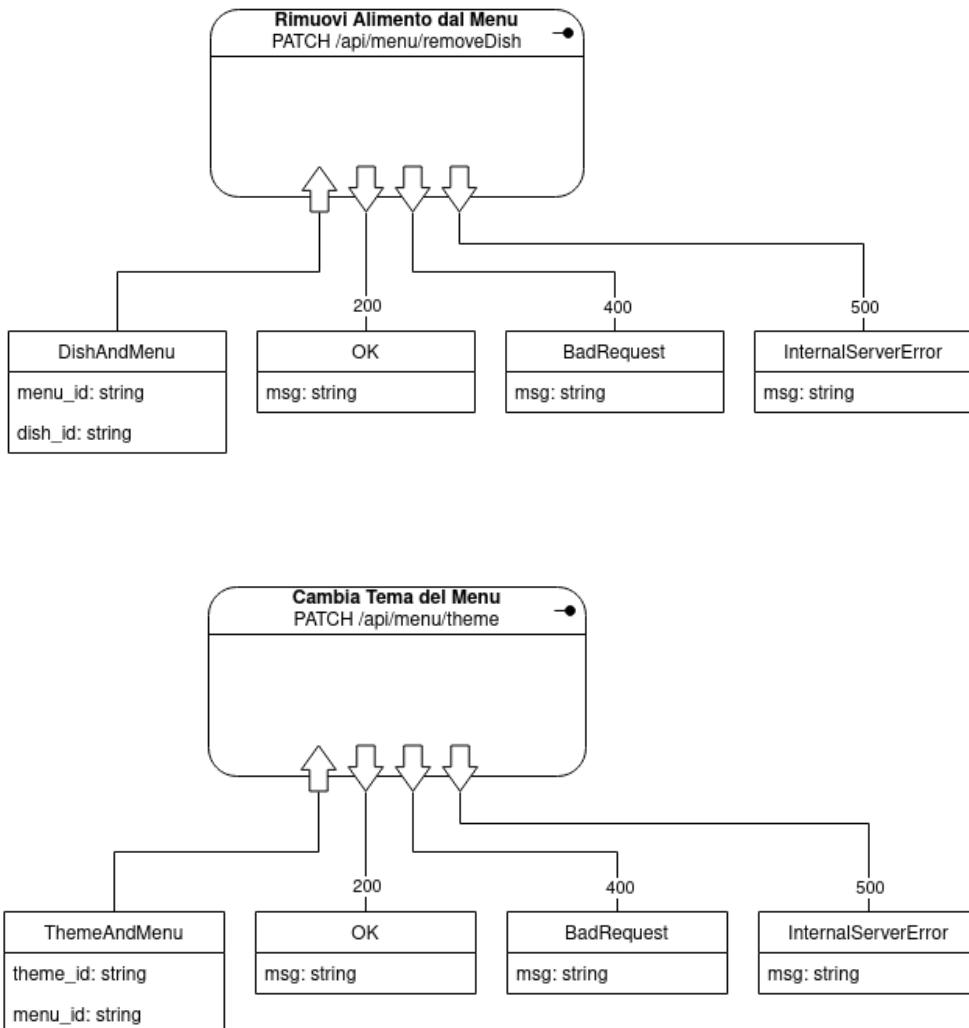


Menu

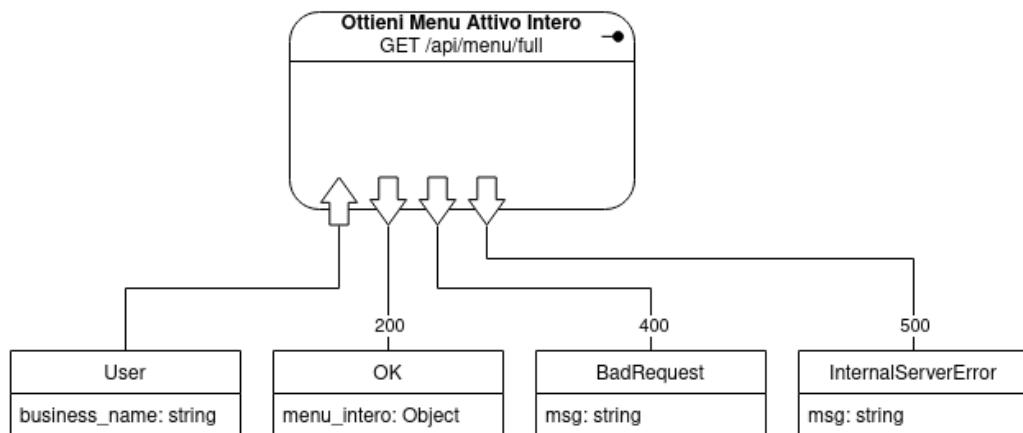
Le API di seguito forniscono le seguenti funzionalità per la gestione dei menù: creazione, modifica e eliminazione di un menù, ottenimento delle informazioni relative ai menù, aggiunta e rimozione di un alimento al menù, cambio del tema del menù.





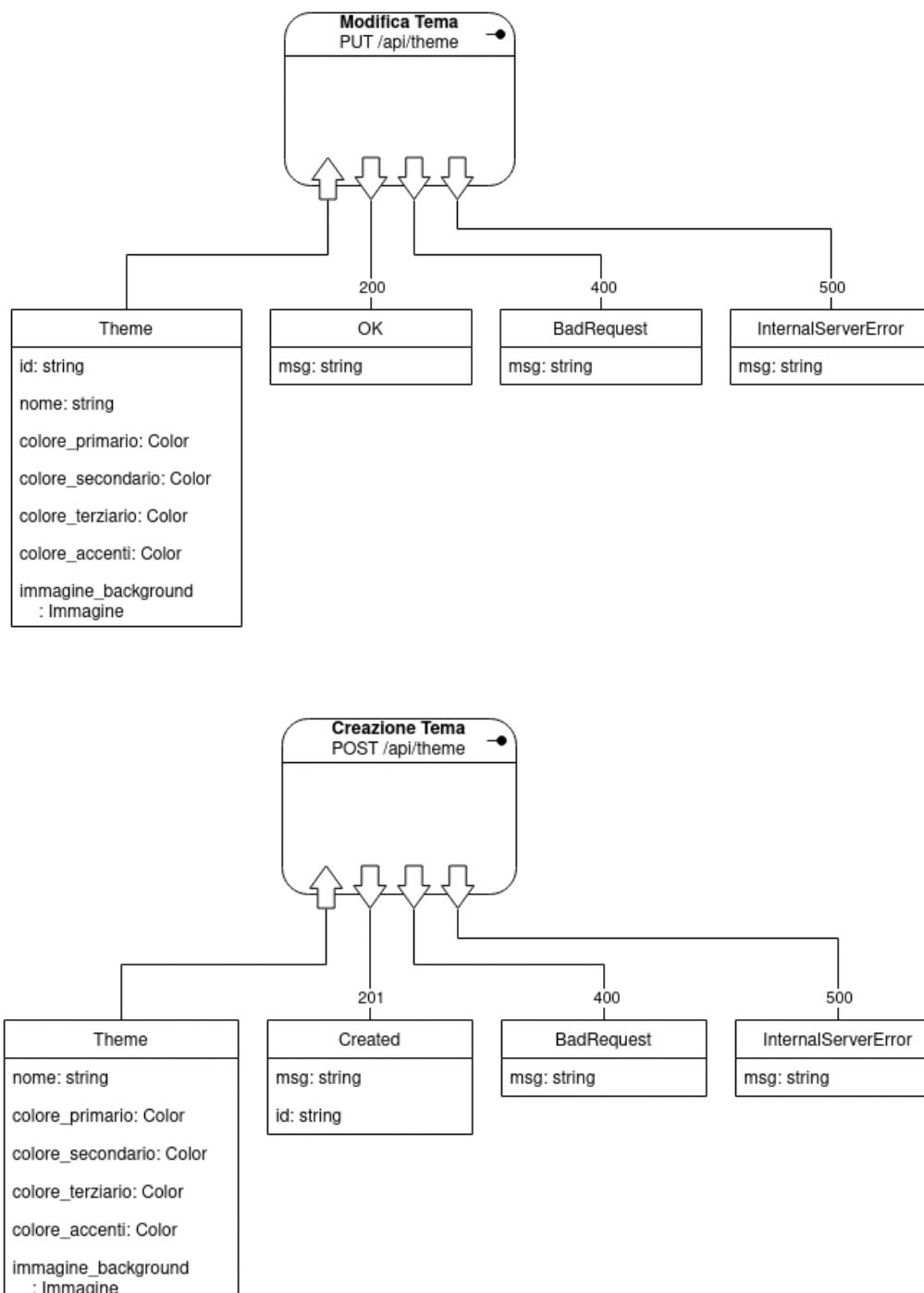


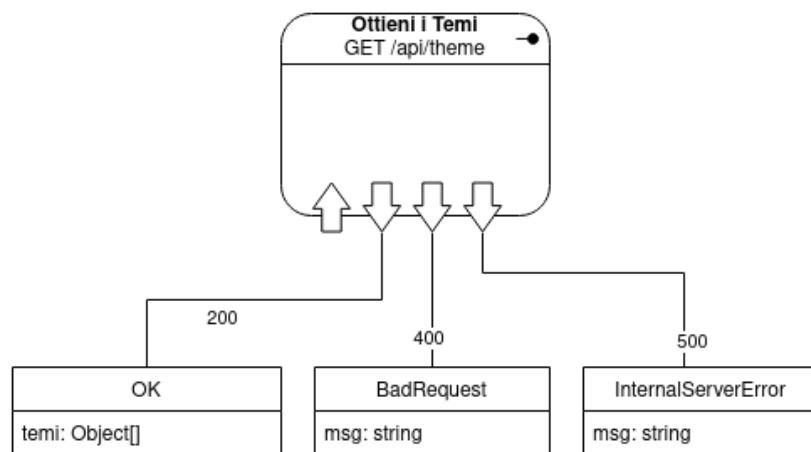
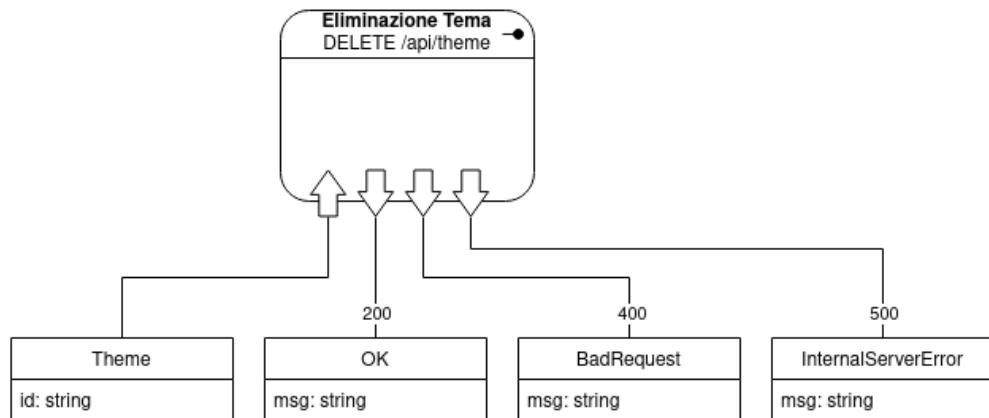
La successiva API **Ottieni Menù Attivo Intero** è accessibile senza un JWT. Questa API è utilizzata dal front-end per mostrare all'utente consumatore la pagina del menù attualmente attivo. La sua risposta in caso di successo è un oggetto contenente tutto quel che riguarda il menù attualmente attivo, ovvero gli attributi del menù, i piatti, le categorie che compaiono nel menù, gli ingredienti che compaiono nel menù. Inviare tutte queste informazioni in risposta a una singola richiesta HTTP evita la necessità di eseguire svariate richieste e aspettare che tutte siano andate a buon fine prima di mostrare il menù, e permette di alleggerire il carico del database ottenendo tutte queste informazioni con una sola query.



Tema

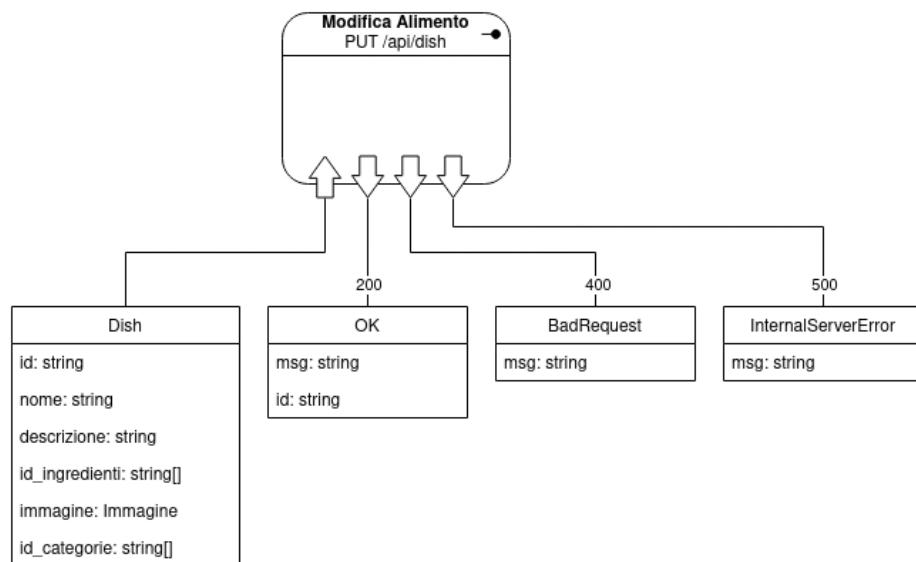
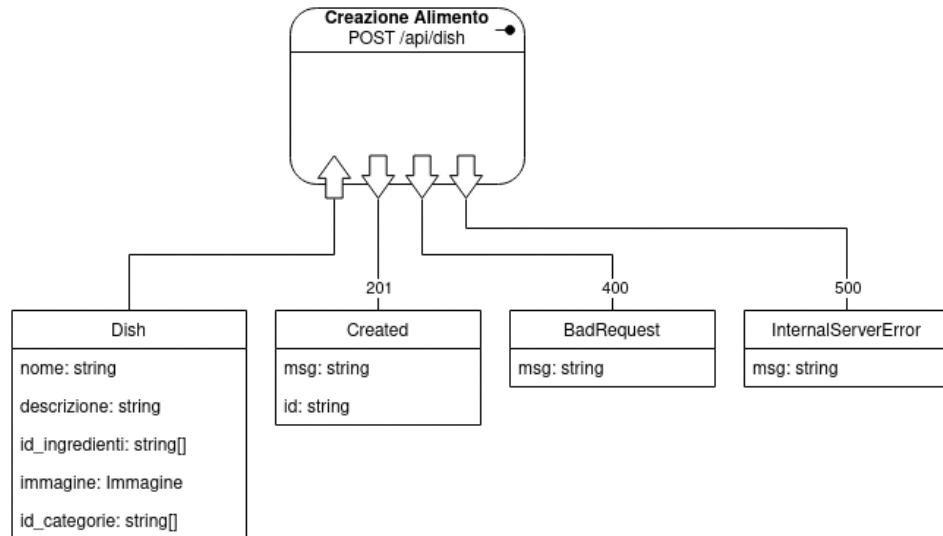
Le API di seguito forniscono le seguenti funzionalità per la gestione del tema: creazione, modifica ed eliminazione del tema, ottenimento dei temi disponibili.

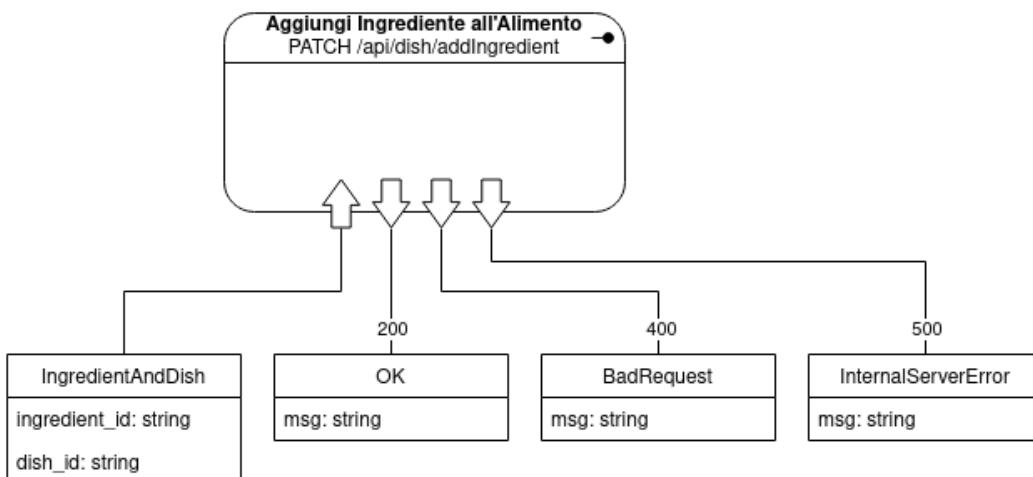
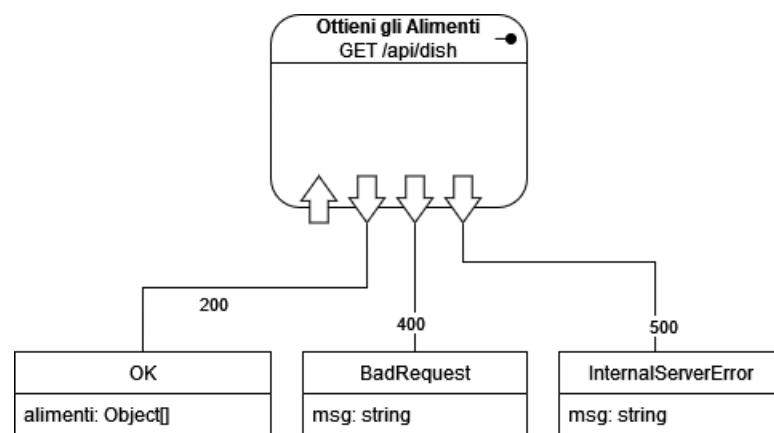
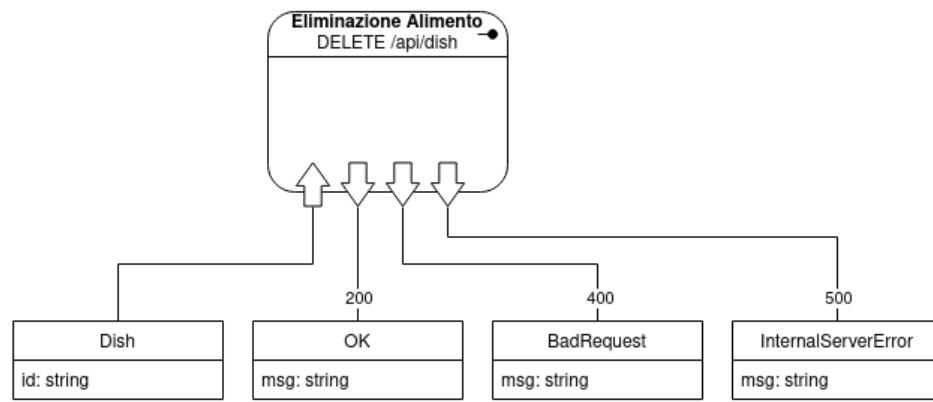


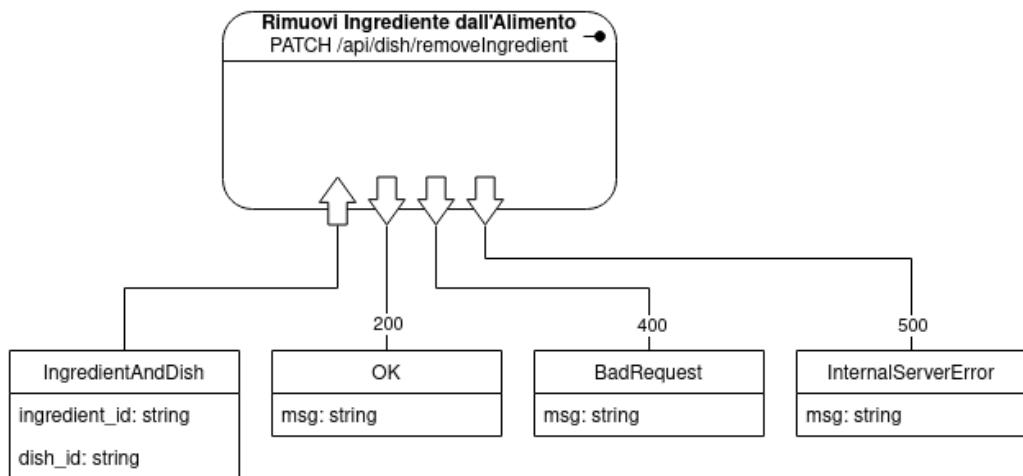


Alimento

Le API di seguito forniscono le seguenti funzionalità per la gestione degli alimenti: creazione, modifica ed eliminazione di un alimento, ottenimento delle informazioni relative agli alimenti, aggiunta e rimozione di un ingrediente ad un alimento.

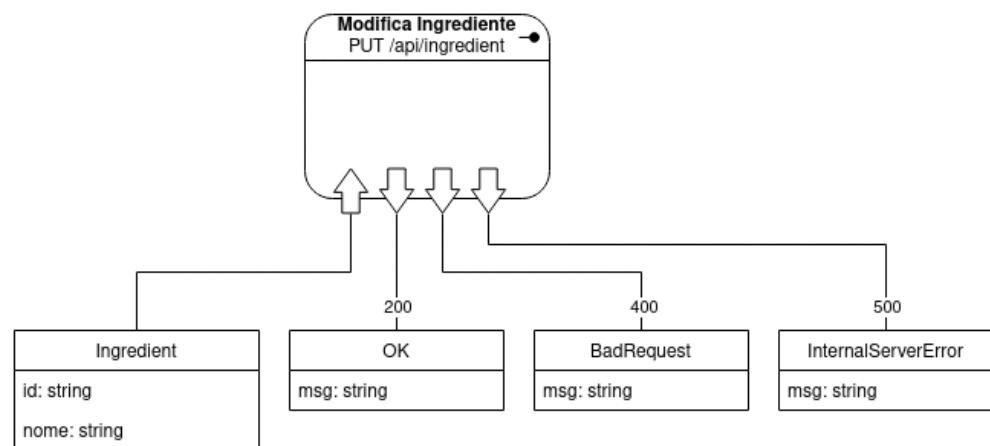
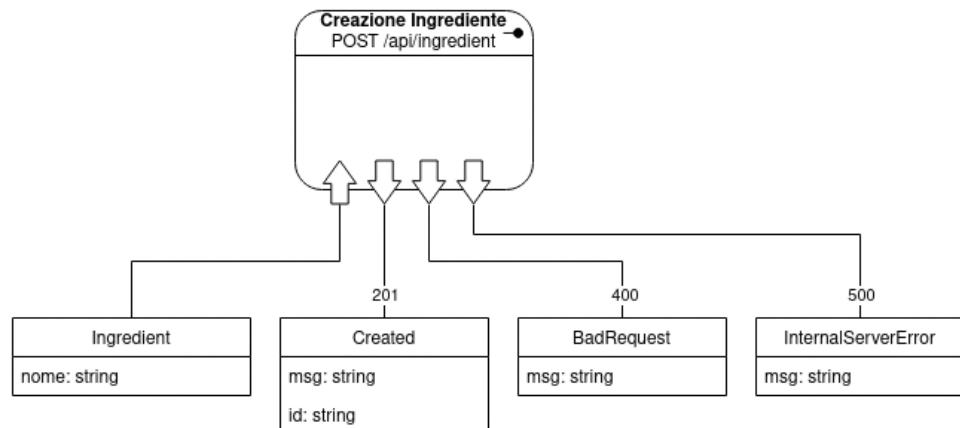


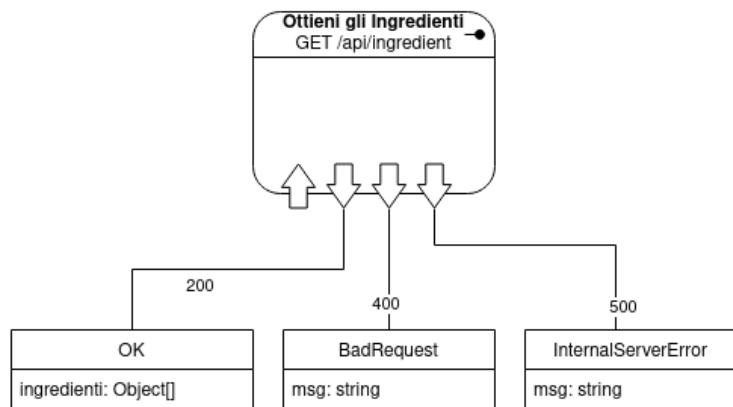
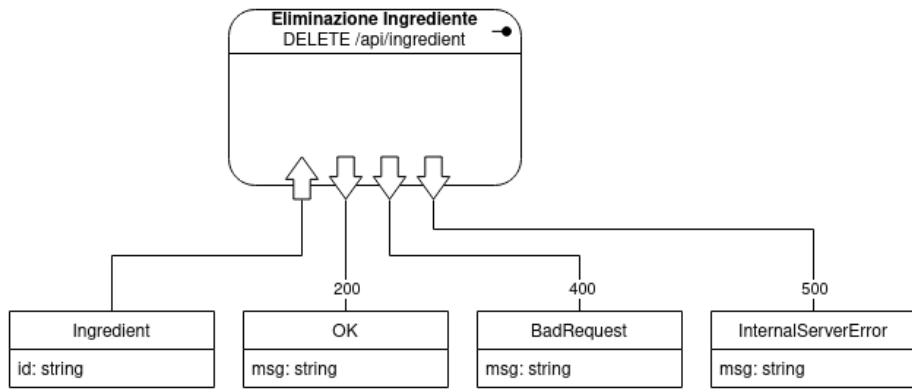




Ingrediente

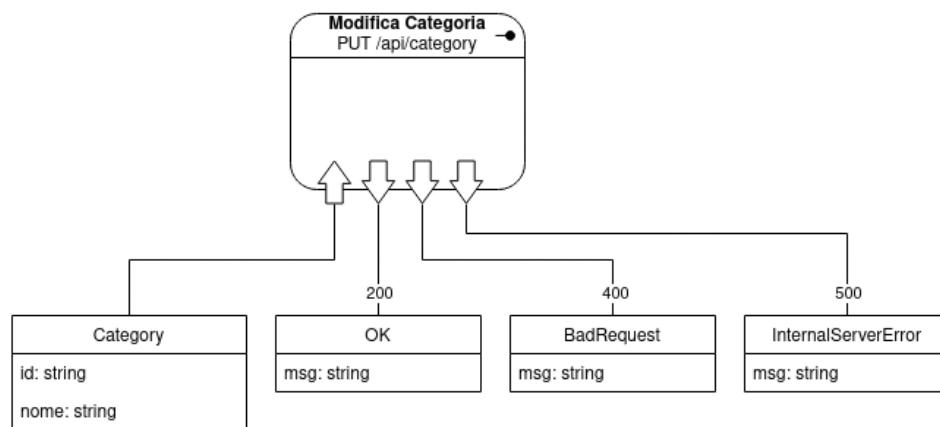
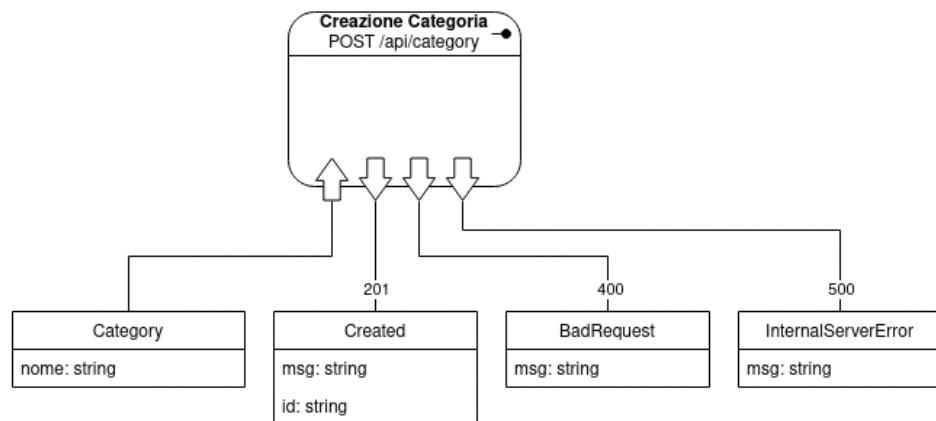
Le API di seguito forniscono le seguenti funzionalità per la gestione degli ingredienti: creazione, modifica ed eliminazione degli ingredienti, ottenimento delle informazioni relative agli ingredienti.

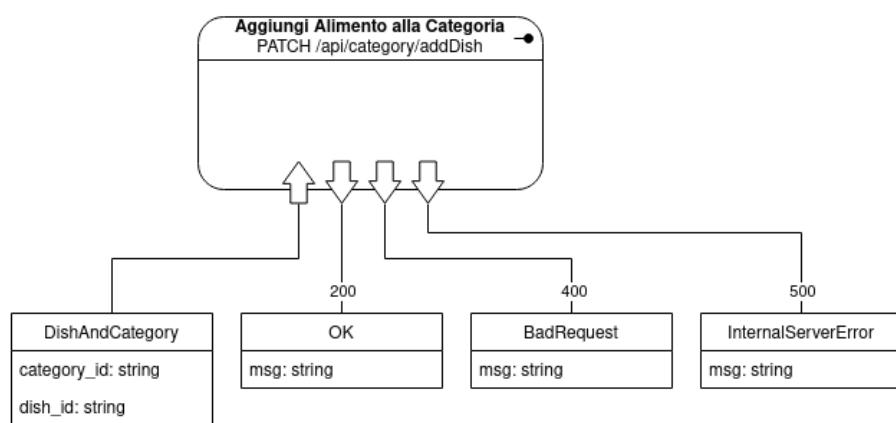
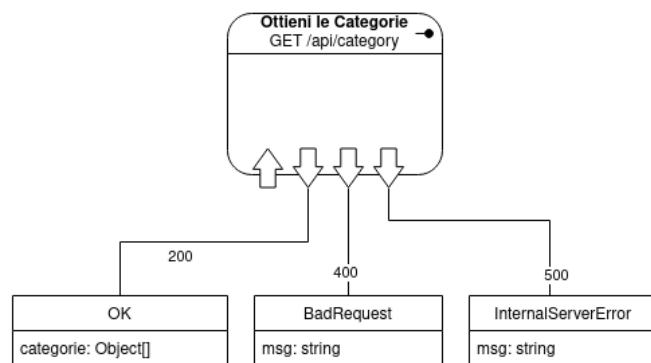
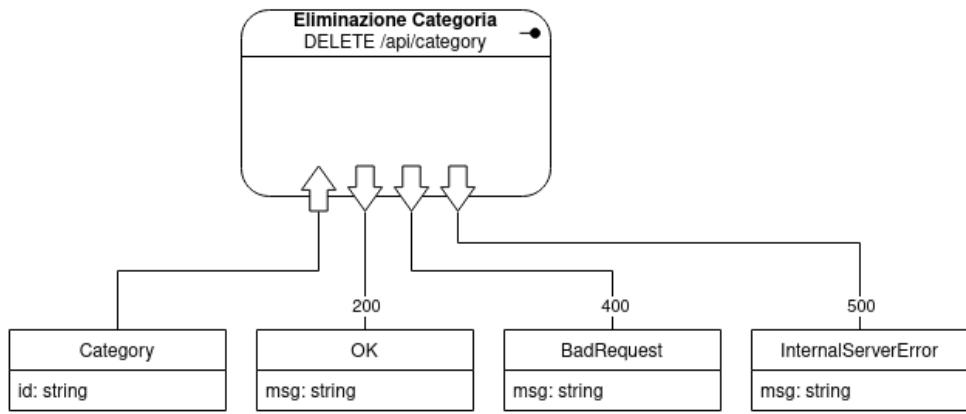


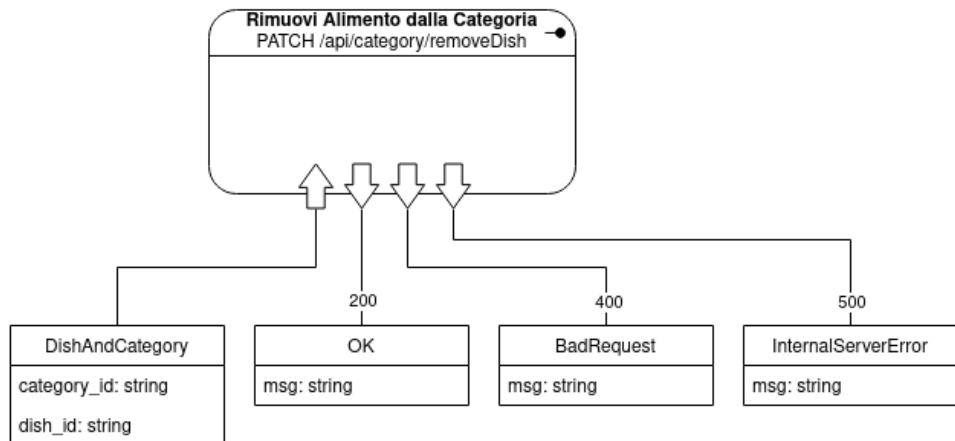


Categoria

Le API di seguito forniscono le seguenti funzionalità per la gestione delle categorie: creazione, modifica ed eliminazione delle categorie, ottenimento delle informazioni relative alle categorie, aggiunta e rimozione di un alimento alla categoria.







Sviluppo API

Sono state sviluppate 17 delle API a partire dai resource models visti in precedenza.

Registrazione

Questa API permette all'utente di registrarsi al servizio, cioè aggiungere le sue future credenziali al database. I controlli svolti dalla funzione sono i seguenti:

- email, password e business_name fornite con successo;
- email e password valide;
- email e business_name non già utilizzate da un altro utente nella collection *users*;

La funzione procede poi al calcolo di hash e salt della password e al salvataggio delle credenziali nel database.

```
const createUser = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to create user:");

    let email = req.body.email;
    const password = req.body.password;
    const business_name = req.body.business_name;

    if(req.body.email == null)
        return failResponse(400, "req.body.email == null");
    if(req.body.password == null)
        return failResponse(400, "req.body.password == null");
    if(req.body.business_name == null)
        return failResponse(400, "req.body.business_name == null");

    /*
     * pw must have 1 (lower or uppercase) letter and must have
     * between 12 and 64 characters
     * the maximum is important to protect from long password DoS attacks
     */
    if (!password.match(/^(?=.*[a-zA-Z])(.{12,64})$/))
        return failResponse(400, "insecure password");

    //case insensitiveness
    email = email.toLowerCase();

    //email address must be valid
    if (!email.match(/^([_a-zA-Z]+\.[_a-zA-Z]*)(\+[a-zA-Z]+)?@(([a-zA-Z]+\.)*[a-zA-Z]{2,4})$/))
        return failResponse(400, "invalid email address");

    const email_is_taken = await User.findOne({ email }).exec() !== null;
    if (email_is_taken)
        return failResponse(400, "email address is taken");

    const business_name_is_taken = await User.findOne({ business_name }).exec() !== null;
    if(business_name_is_taken)
        return failResponse(400, "business name is taken")

    const password_hash = await argon2.hash(password);
    const document = new User({ email, password_hash, business_name });
    const user_was_saved = (await document.save()) !== null;

    if(!user_was_saved)
        return failResponse(500, "internal server error");
    res.status(201).send({ msg: "user saved successfully" });
};
```

Deregistrazione

Questa API permette all'utente di eliminare il proprio account dal servizio e quindi rimuovere le proprie credenziali dal database. I controlli svolti dalla funzione sono i seguenti:

- `email` e `password` fornite con successo;
- `email` e `password` inserite corrispondono a quelle dell'utente che sta richiedendo l'eliminazione.

La funzione procede quindi ad eliminare tutte le informazioni relative all'utente e ai suoi menu dal database.

```
const deleteUser = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to delete user: ");

    let email = req.body.email;
    const password = req.body.password;
    const user_id = req.body.jwt_payload.user_id;

    if(email == null)
        return failResponse(400, "req.body.email == null");
    if(password == null)
        return failResponse(400, "req.body.password == null");

    email = email.toLowerCase();
    const user = await User.findOne({ _id: user_id, email }).exec();

    if(user === null)
        return failResponse(400, "wrong credentials");

    const password_hash = user.password_hash;
    const password_is_correct = await argon2.verify(password_hash, password);

    if(!password_is_correct)
        return failResponse(400, "wrong credentials");

    let user_was_deleted =
        (await User.deleteOne({ email, password_hash })).deletedCount != 0;

    if(!user_was_deleted)
        return failResponse(500, "internal server error");

    // delete documents owned by the user
    await Ingredient.deleteMany({ owner_id: user._id });
    await Dish.deleteMany({ owner_id: user._id });
    await Menu.deleteMany({ owner_id: user._id });
    await Category.deleteMany({ owner_id: user._id });

    res.status(200).send({ msg: "user deleted successfully" });
};
```

Ottenimento informazioni relative all'utente

Questa API consente di ottenere tutte le informazioni relative all'utente.

```
const getUser = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to delete user: ");

    const _id = req.body.jwt_payload.user_id;

    const user = await User.findOne({ _id }).exec();

    if(user === null)
        return failResponse(500, "internal server error");

    //send everything except password_hash
    res.status(200).send({
        _id: user._id, business_name: user.business_name,
        email: user.email, enabled_2fa: user.enable_2fa
    });
};
```

Login

Questa API permette all'utente di fare il login e quindi accedere alle funzionalità del servizio. I controlli svolti dalla funzione sono i seguenti:

- **email** e **password** fornite con successo;
- **email** e **password** inserite corrispondono a un utente nella collection *users*.

La funzione procede poi generando un JWT valido per 24 ore per permettere all'utente di accedere alle API protette.

```
const authenticateUser = async (req, res) => {
    const failResponse = failureResponseHandler(res, "authentication failed: ");

    const user_email = req.body.email, user_password = req.body.password;

    if(user_password == null)
        return failResponse(400, "req.body.password == null");
    if(user_email == null)
        return failResponse(400, "req.body.email == null");

    const found_user = await User.findOne({ email: user_email }).exec();
    const found_a_user_with_correct_pw =
        found_user != null && await argon2.verify(found_user.password_hash, user_password);

    if (!found_a_user_with_correct_pw)
        return failResponse(404, "wrong credentials");

    const payload = { user_id: found_user._id };
    const options = { expiresIn: 86400 };
    jwt.sign(payload, process.env.JWT_SECRET, options, (err, jwtToken) => {
        if(err)
            failResponse(500, "internal server error");
        else
            res.status(200).send({ token: jwtToken });
    });
};
```

Creazione menù

Questa API permette la creazione di un menù da parte dell'utente. I controlli svolti dalla funzione sono i seguenti:

- name, start_time e end_time forniti con successo;
- start_time < end_time;
- dishes array di ObjectId validi e presenti nella collection *dishes*;
- name non già in uso da un altro menù nella collection *menus* creato dall'utente.

La funzione procede poi salvando il menù nel database.

```
const createMenu = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to create menu: ");

    const owner_id = req.body.jwt_payload.user_id, name = req.body.name;
    const start_time = req.body.start_time, end_time = req.body.end_time;
    const unvalidated_dish_ids = req.body.dishes;

    if (name == null || String(name).length < 1)
        return failResponse(400, "invalid name");
    if (!Number.isInteger(start_time))
        return failResponse(400, "invalid 'start_time' parameter");
    if (!Number.isInteger(end_time))
        return failResponse(400, "invalid 'end_time' parameter");
    if (start_time >= end_time)
        return failResponse(400, "start_time >= end_time");

    const dishes = await validateObjectIds(unvalidated_dish_ids, Dish);

    if (dishes == null)
        return failResponse(400, "invalid 'dishes' parameter");

    const found = (await Menu.findOne({ owner_id, name }).exec()) !== null;

    if (found)
        return failResponse(400, "name taken");

    const document = new Menu({ owner_id, name, dishes, start_time, end_time });
    const was_saved = (await document.save()) !== null;

    if (!was_saved)
        return failResponse(500, "internal server error");

    res.status(201).send({ msg: "menu saved successfully", id: document._id });
};
```

Eliminazione menù

Questa API permette l'eliminazione di un menù da parte dell'utente. I controlli svolti dalla funzione sono i seguenti:

- menu_id fornito con successo;
- menu_id corrispondente a un menù nella collection *menus* creato dall'utente.

La funzione procede quindi eliminando il menù dal database.

```
const deleteMenu = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to delete menu:");
    const _id = req.body.menu_id, owner_id = req.body.jwt_payload.user_id;

    if (_id == null)
        return failResponse(400, "req.body.menu_id == null");

    const found_and_deleted =
        (await Menu.deleteOne({ owner_id, _id })).deletedCount != 0;

    if (!found_and_deleted)
        return failResponse(400, "no menu with such name");

    res.status(200).send({ msg: "menu deleted successfully" });
};
```

Ottenimento informazioni relative ai menù

Questa API permette all'utente di ottenere le informazioni relative ai propri menù.

```
const getMenus = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to get menus:");

    const owner_id = req.body.jwt_payload.user_id;
    const menus = await Menu.find({ owner_id });

    if (menus == null)
        return failResponse(500, "internal server error");

    res.status(200).send(menus);
};
```

Ottenimento menù attivo intero

Questa API permette di ottenere tutte le informazioni necessarie a mostrare all'utente cliente la pagina del menù attivo. I passi svolti dalla funzione sono i seguenti:

- controllo che il `business_name` sia stato passato tra i parametri GET;
- query al database, sfruttando la funzionalità di pipeline di MongoDB per ottenere tutte le informazioni necessarie con una sola query (la pipeline utilizzata è alla pagina successiva);
- controllo che la query sia andata a buon fine e che fosse presente un menù attivo;
- invio dei dati.

```
const getFullMenu = async (req, res, next) => {
    const failResponse =
        failureResponseHandler(res, "failed to get full active menu: ");

    const business_name = req.query.business_name;

    if (business_name == null)
        return failResponse(400, "no business name specified");

    const full_active_menu_pipeline =
        getFullActiveMenuPipeline(business_name, currentTime());
    const full_menu_array =
        await User.aggregate(full_active_menu_pipeline).exec();

    if (full_menu_array == null)
        return failResponse(500, "internal server error");
    if (full_menu_array[0] == null)
        return failResponse(400, "no menu currently active");

    res.status(200).send(full_menu_array[0]);
};
```

```

const getFullActiveMenuPipeline = (business_name, current_time) => [
  { // filter out other businesses
    $match: {
      business_name,
    },
  }, { // remove sensitive data (only _id and business_name remain)
    $project: {
      email: 0,
      password_hash: 0,
      enable_2fa: 0,
    },
  }, { // create and fill categories array with category subdocuments
    $lookup: {
      from: "categories",
      localField: "_id",
      foreignField: "owner_id",
      as: "categories",
      pipeline: [
        { // remove redundant owner_id
          $project: { owner_id: 0 },
        },
      ],
    },
  }, { // fill menu array with a menu subdocument for the active menu
    $lookup: {
      from: "menus",
      localField: "_id",
      foreignField: "owner_id",
      as: "menu",
      pipeline: [
        { // only match active menu(s)
          $match: {
            start_time: { $lte: current_time },
            end_time: { $gt: current_time },
          }
        }, { // only return 1 menu
          $limit: 1
        }, { // fill dishes array with subdocs instead of '_id's
          $lookup: {
            from: "dishes",
            localField: "dishes",
            foreignField: "_id",
            as: "dishes",
            pipeline: [
              { // fill ingredients array with subdocs instead of '_id's
                $lookup: {
                  from: "ingredients",
                  localField: "ingredients",
                  foreignField: "_id",
                  as: "ingredients",
                  pipeline: [
                    { // remove redundant owner_id
                      $project: { owner_id: 0 }
                    },
                  ],
                },
              },
            ], { // fill categories array with subdocs instead of '_id's
              $lookup: {
                from: "categories",
                localField: "categories",
                foreignField: "_id",
                as: "categories",
                pipeline: [
                  { // remove redundant owner_id
                    $project: { owner_id: 0 }
                  },
                ],
              },
            }, { // remove redundant owner_id
              $project: { owner_id: 0 }
            },
          ],
        },
      ],
    },
  }, { // turn the menu array containing only the active menu into a single menu object
    $unwind: "$menu"
  },
];

```

Creazione alimento

Questa API permette la creazione di un alimento da parte dell'utente. I controlli svolti dalla funzione sono i seguenti:

- name fornito con successo;
- ingredients array di ObjectId validi e presenti nella collection *ingredients*;
- categories array di ObjectId validi e presenti nella collection *categories*;
- name non in uso per un altro alimento nella collection *dishes* creato dallo stesso utente.

La funzione procede poi salvando l'alimento nel database.

```
const createDish = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to create dish: ");

    const owner_id = req.body.jwt_payload.user_id, name = req.body.name;
    const description = req.body.description, image = req.body.image;
    const unvalidated_ingredient_ids = req.body.ingredients;
    const unvalidated_category_ids = req.body.categories;

    if(name == null || String(name).length < 1)
        return failResponse(400, "invalid name");

    const ingredients =
        await validateObjectIds(unvalidated_ingredient_ids, Ingredient);
    const categories =
        await validateObjectIds(unvalidated_category_ids, Category);

    if(ingredients == null)
        return failResponse(400, "invalid 'ingredients' parameter" );
    if(categories == null)
        return failResponse(400, "invalid 'categories' parameter" );

    const found = (await Dish.findOne({ owner_id, name }).exec()) !== null;

    if(found)
        return failResponse(400, "name taken");

    const document =
        new Dish({ owner_id, name, description, image, ingredients, categories });
    const was_saved = (await document.save()) !== null;

    if(!was_saved)
        return failResponse(500, "internal server error");

    res.status(201).send({ msg: "dish saved successfully", id: document._id });
};
```

Eliminazione alimento

Questa API permette l'eliminazione di un alimento da parte dell'utente. I controlli svolti dalla funzione sono i seguenti:

- dish_id fornito con successo;
- dish_id corrispondente ad un alimento nella collection *dishes* creato dallo stesso utente.

La funzione procede poi eliminando l'alimento dal database.

```
const deleteDish = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to delete dish: ");

    const _id = req.body.dish_id, owner_id = req.body.jwt_payload.user_id;

    if(_id == null)
        return failResponse(400, "req.body.name == null");

    let found_and_deleted =
        (await Dish.deleteOne({ owner_id, _id })).deletedCount != 0;

    if(!found_and_deleted)
        return failResponse(400, "no dish with such name");

    res.status(200).send({ msg: "dish deleted successfully" });
};
```

Ottenimento informazioni relative agli alimenti

Questa API permette all'utente di ottenere le informazioni relative ai propri alimenti.

```
const getDishes = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to get dishes: ");

    const owner_id = req.body.jwt_payload.user_id

    let dishes = await Dish.find({ owner_id });

    for (let i = 0; i < dishes.length; i++)
        if (dishes[i].image != null)
            dishes[i].image = dishes[i].image.toString('base64');

    if (dishes == null)
        return failResponse(500, "internal server error");

    res.status(200).send(dishes);
};
```

Creazione ingrediente

Questa API permette la creazione di un ingrediente da parte dell'utente. I controlli svolti dalla funzione sono i seguenti:

- name fornito con successo;
- name non già in uso per un ingrediente nella collection *ingredients* creato dallo stesso utente.

La funzione procede poi salvando l'ingrediente nel database.

```
const createIngredient = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to create ingredient: ");

    const name = req.body.name, owner_id = req.body.jwt_payload.user_id;

    if(name == null || String(name).length < 1)
        return failResponse(400, "invalid name");

    const found = (await Ingredient.findOne({ name, owner_id }).exec()) !== null;

    if(found)
        return failResponse(400, "name taken");

    let document = new Ingredient ({ name, owner_id });
    const was_saved = (await document.save()) !== null;

    if(!was_saved)
        return failResponse(500, "internal server error")

    res.status(201).send({
        msg: "ingredient saved successfully" , id: document._id
    });
};
```

Eliminazione ingrediente

Questa API permette l'eliminazione di un ingrediente da parte dell'utente. I controlli svolti dalla funzione sono i seguenti:

- `ingredient_id` fornito con successo;
- `ingredient_id` corrispondente ad un ingrediente nella collection `ingredients` creato dallo stesso utente.

La funzione procede poi eliminando l'ingrediente dal database.

```
const deleteIngredient = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to delete ingredient: ");

    const _id = req.body.ingredient_id, owner_id = req.body.jwt_payload.user_id;

    if(_id == null)
        return failResponse(400, "req.body.ingredient_id == null");

    let found_and_deleted =
        (await Ingredient.deleteOne({ _id, owner_id })).deletedCount != 0;

    if(!found_and_deleted)
        return failResponse(400, "no ingredient with such id");

    res.status(200).send({ msg: "ingredient deleted successfully" });
};
```

Ottenimento informazioni relative agli ingredienti

Questa API permette all'utente di ottenere le informazioni relative ai propri ingredienti.

```
const getIngredients = async (req, res, next) => {
  const failResponse = failureResponseHandler(res, "failed to get ingredients: ");

  const owner_id = req.body.jwt_payload.user_id;

  const ingredients = await Ingredient.find({ owner_id });

  if (ingredients == null)
    return failResponse(500, "internal server error");

  res.status(200).send(ingredients);
};
```

Creazione categoria

Questa API permette la creazione di una categoria da parte dell'utente. I controlli svolti dalla funzione sono i seguenti:

- name fornito con successo;
- name non già in uso per un'altra categoria nella collection `categories` creata dallo stesso utente.

La funzione procede poi salvando la categoria nel database.

```
const createCategory = async (req, res, next) => {
  const failResponse = failureResponseHandler(res, "failed to create category:");
  const name = req.body.name, owner_id = req.body.jwt_payload.user_id;

  if(name == null || String(name).length < 1)
    return failResponse(400, "invalid_name");

  const found = (await Category.findOne({ name, owner_id }).exec()) !== null;

  if(found)
    return failResponse(400, "name taken");

  let document = new Category ({ name, owner_id });
  const was_saved = (await document.save()) !== null;

  if(!was_saved)
    return failResponse(500, "internal server error");

  res.status(201).send({ msg: "category saved successfully" });
};
```

Eliminazione categoria

Questa API permette l'eliminazione di una categoria da parte dell'utente. I controlli svolti dalla funzione sono i seguenti:

- category_id fornito con successo;
- category_id corrispondente ad una categoria nella collection categories creata dallo stesso utente.

La funzione procede poi eliminando la categoria dal database.

```
const deleteCategory = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to delete category: ");

    const _id = req.body.category_id, owner_id = req.body.jwt_payload.user_id;

    if(_id == null)
        return failResponse(400, "req.body.category_id == null");

    let found_and_deleted =
        (await Category.deleteOne({ _id, owner_id })).deletedCount != 0;

    if(!found_and_deleted)
        return failResponse(400, "no category with such name");

    res.status(200).send({ msg: "category deleted successfully" });
};
```

Ottenimento informazioni relative alle categorie

Questa API permette all'utente di ottenere le informazioni relative alle proprie categorie.

```
const getCategories = async (req, res, next) => {
    const failResponse = failureResponseHandler(res, "failed to get category: ");

    const owner_id = req.body.jwt_payload.user_id;
    const categories = await Category.find({ owner_id: user._id });

    if (categories == null)
        return failResponse(500, "internal server error");

    res.status(200).send(categories);
};
```

Documentazione API

Le API implementate sono state documentate utilizzando il modulo NodeJS `swagger-ui-express`, seguendo la specifica OpenAPI 3.0.

Per visualizzare la pagina “Swagger UI” contenente la documentazione è sufficiente connettersi all'endpoint `/api-docs`.

Swagger Baccalà - OpenAPI 1.0.0 OAS3

API documentation of the baccalà service

Some useful links:

- [The Baccalà repository](#)

Servers Authorize

user Operations about users ^

POST	/api/user	Create a new user		
DELETE	/api/user	Delete a user		
GET	/api/user	Retrieve the user's information		
POST	/api/user/login	Obtain a JSON Web Token		

ingredient Operations about ingredients ^

POST	/api/ingredient	Create a new ingredient		
DELETE	/api/ingredient	Delete an ingredient		
GET	/api/ingredient	Retrieve the user's ingredients		

category Operations about categories ^

POST	/api/category	Create a new category		
DELETE	/api/category	Delete a category		
GET	/api/category	Retrieve the user's categories		

dish Operations about dishes ^

POST	/api/dish	Create a new dish		
DELETE	/api/dish	Delete a dish		
GET	/api/dish	Retrieve the user's dishes		

menu Operations about menus ^

POST	/api/menu	Create a new menu		
DELETE	/api/menu	Delete a menu		
GET	/api/menu	Retrieve the user's menus		
GET	/api/menu/full	Retrieve a user's categories and menus where ids are substituted with actual values		

La pagina “Swagger UI”

Di seguito vengono mostrati alcuni esempi di documentazione di API del progetto.

The screenshot shows a detailed API documentation for the `POST /api/user` endpoint. The top navigation bar indicates the method (`POST`) and the endpoint (`/api/user`). Below this, there's a section for **Parameters** (which is currently empty), a **Request body** section (set to `application/json`), and an **Example Value** (containing a JSON object with fields like `email`, `business_name`, `password`, and `enable_2fa`). The main content area is titled **Responses**. It lists three possible status codes: `201` (user saved successfully), `400` (failed to create user), and `500` (internal server error). Each response entry includes a **Media type** dropdown (set to `application/json`), an **Example Value** (containing a JSON object with a `msg` field), and a **Links** column (which is empty for all entries).

Code	Description	Links
201	user saved successfully	No links
400	failed to create user	No links
500	internal server error	No links

La documentazione di POST /api/user

GET /api/dish Retrieve the user's dishes

Parameters

No parameters

Responses

Code	Description	Links
200	dishes retrieved successfully	No links
	<p>Media type</p> <p>application/json</p> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>{ { "id": "some_alphanumeric_id_24", "owner_id": "some_alphanumeric_id_24", "name": "Spaghetti alla carbonara", "description": "Buonissima carbonara del nostro chef", "image": "binary_data", "ingredients": [{ "id": "some_alphanumeric_id_24" }], "categories": [{ "id": "some_alphanumeric_id_24" }] } }</pre>	
400	failed to retrieve dishes	No links
	<p>Media type</p> <p>application/json</p> <p>Example Value Schema</p> <pre>{ "msg": "informational message" }</pre>	
401	token not provided	No links
	<p>Media type</p> <p>application/json</p> <p>Example Value Schema</p> <pre>{ "msg": "token not provided" }</pre>	
403	token not valid	No links
	<p>Media type</p> <p>application/json</p> <p>Example Value Schema</p> <pre>{ "msg": "token not valid" }</pre>	
500	internal server error	No links
	<p>Media type</p> <p>application/json</p> <p>Example Value Schema</p> <pre>{ "msg": "internal server error" }</pre>	

La documentazione di GET /api/dish

GET /api/menu/full Retrieve a user's categories and menus where ids are substituted with actual values

Parameters

Name	Description
business_name <small>required</small>	The name of the business which owns the menus that will be returned
string (query)	<input type="text" value="business_name"/>

Responses

Code	Description	Links
200	menus retrieved successfully	No links
	<p>Media type</p> <p><input checked="" type="button" value="application/json"/> <input type="button" value="text/html"/></p> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>{ "menu": [{ "id": "some_alphanumeric_id_24", "name": "Cena", "dishes": [{ "id": "some_alphanumeric_id_24", "name": "Spaghetti alla carbonara", "description": "Buonissima carbonara del nostro chef", "image": "binary_data", "ingredients": [{ "id": "some_alphanumeric_id_24", "name": "cipolle" }], "categories": [{ "id": "some_alphanumeric_id_24", "name": "piccante" }] }, { "start_time": "minutes_since_midnight", "end_time": "minutes_since_midnight" }] }] }</pre>	
400	failed to retrieve menus	No links
	<p>Media type</p> <p><input checked="" type="button" value="application/json"/> <input type="button" value="text/html"/></p> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>{ "msg": "informational message" }</pre>	
500	internal server error	No links
	<p>Media type</p> <p><input checked="" type="button" value="application/json"/> <input type="button" value="text/html"/></p> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>{ "msg": "internal server error" }</pre>	

La documentazione di GET /api/menu/full

Sviluppo del Front-end

Il front-end è stato sviluppato utilizzando il framework NextJS e la libreria ReactJS:

- ReactJS permette lo sviluppo di interfacce complesse e dinamiche;
- NextJS migliora ed espande le funzionalità di ReactJS; ha permesso di migliorare le performance del front-end attraverso ottimizzazioni e di fare routing delle pagine, altrimenti non possibile con ReactJS che è mirato a single-page applications.

Abbiamo anche utilizzato Material UI, una libreria di componenti basati sul Material Design, lo stile grafico sviluppato da Google.

Per la validazione degli input è stata utilizzata la libreria React Hook Form.

Per la gestione della sessione sono state applicate le seguenti misure:

- quando l'utente si autentica il JWT viene salvato in un cookie;
- il JWT è sempre disponibile per essere utilizzato dalle pagine che devono fare uso delle API, questo perché il cookie viene trasmesso dal browser in tutte le richieste SameSite. L'utente non deve autenticarsi ogni volta che accede ad una pagina protetta;
- se il JWT è scaduto (validità 24 ore) o non è più valido, l'utente viene reindirizzato alla pagina di Login dove potrà rinnovare il token e aggiornare il cookie.

La schermata principale presenta due pulsanti in alto a destra che portano alle schermate registrazione e login. Cliccando invece su “scopri di più” si viene reindirizzati alla pagina “About”.

L'utente viene avvisato in basso a sinistra tramite una componente “snackbar” in caso di errori.

Una volta registrato un account ed effettuato il login l'utente accede alla pagina di gestione del menù



Baccalà

Il servizio numero uno per la gestione del tuo menù digitale!

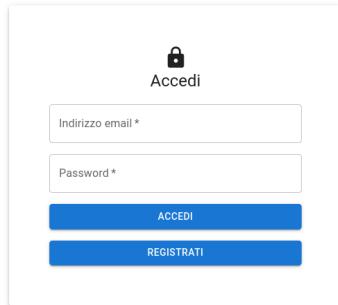
[SCOPRI DI PIÙ](#)

About

Prezzi

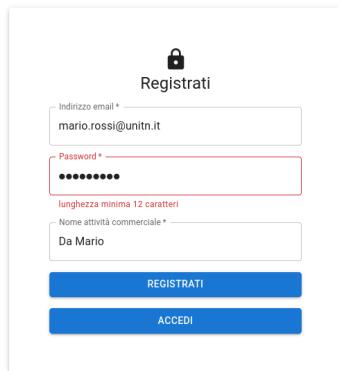
Demo Sperimenta con il nostro servizio 0€/mo • 1 menu	Pro Inizia a lavorare con Baccalà 10€/mo • 100 menu • Codice QR
--	---

Qui puoi trovare la documentazione del servizio: [Docs](#)



A login form titled "Accedi" (Log In) featuring a lock icon. It includes fields for "Indirizzo email *" (Email address) and "Password *". Below the fields are two blue buttons: "ACCEDI" (Log In) and "REGISTRATI" (Register).

Indirizzo email *
Password *
ACCEDI
REGISTRATI



A registration form titled "Registrati" (Register) featuring a lock icon. It includes fields for "Indirizzo email *" (Email address) containing "mario.rossi@unitn.it", "Password *", and "Nome attività commerciale *". The "Password *" field has a red border and a validation message "lunghezza minima 12 caratteri" (minimum length 12 characters). The "Nome attività commerciale *" field contains "Da Mario". Below the fields are two blue buttons: "REGISTRATI" (Register) and "ACCEDI" (Log In).

Indirizzo email *
mario.rossi@unitn.it
Password *

lunghezza minima 12 caratteri
Nome attività commerciale *
Da Mario
REGISTRATI
ACCEDI

! Impossibile accedere
authentication failed: wrong credentials

Nella pagina di gestione dei menù l'utente può usare le seguenti funzionalità:

- creare nuovi menù premendo il pulsante "+" in alto a destra;
- selezionare molteplici menù per eliminarli;
- cliccare il menù a tendina per navigare tra le pagine menu, alimenti, ingredienti e categorie;
- cliccare il menu hamburger per accedere alle pagine profilo, menù, codice QR e menù attivo.

The screenshot shows a user interface for managing menus. At the top, there's a blue header bar with the word "Admin". Below it is a navigation bar with tabs: "Tabella" (selected) and "Menu". The main area is a table titled "Elementi" with the following columns: "Nome" (Name), "Alimenti" (Food), "Orario di attività - inizio" (Activity start time), and "Orario di attività - fine" (Activity end time). There are two rows of data:

Nome	Alimenti	Orario di attività - inizio	Orario di attività - fine
Menu Cucina	⚠️ WIP - Lista degli alimenti	19:00	23:00
Menu Bar	⚠️ WIP - Lista degli alimenti	7:00	19:00

Admin

Tabella
Menu

Elementi				
	Nome	Piatti	Orario di attività - inizio	Orario di attività - fine
<input type="checkbox"/>	Menu Cucina	⚠ WIP - Lista dei piatti	19:00	23:00
<input type="checkbox"/>	Menu Bar	⚠ WIP - Lista dei piatti	7:00	19:00

Aggiungi menu

Nome *
12:00 PM

Orario di fine *
03:00 PM

INVIA
CANCEL

Admin

Tabella
Menu

2 selezionati				
	Nome	Piatti	Orario di attività - inizio	Orario di attività - fine
<input checked="" type="checkbox"/>	Menu Cucina	⚠ WIP - Lista dei piatti	19:00	23:00
<input checked="" type="checkbox"/>	Menu Bar	⚠ WIP - Lista dei piatti	7:00	19:00

☰ Admin

Tabella
Alimenti

Elementi				
	Nome	Descrizione	Ingredienti	Categorie
<input type="checkbox"/>	Pizza Margherita	La pizza Margherita è la tipica pizza napoletana	WIP - Lista degli Ingredienti	WIP - Lista delle Categorie
<input type="checkbox"/>	Pizza Diavola	Fuoco	WIP - Lista degli Ingredienti	WIP - Lista delle Categorie
<input type="checkbox"/>	Spritz Aperol	Il classico	WIP - Lista degli Ingredienti	WIP - Lista delle Categorie
<input type="checkbox"/>	Moscow Mule	Cocktail fresco	WIP - Lista degli Ingredienti	WIP - Lista delle Categorie

☰ Profilo

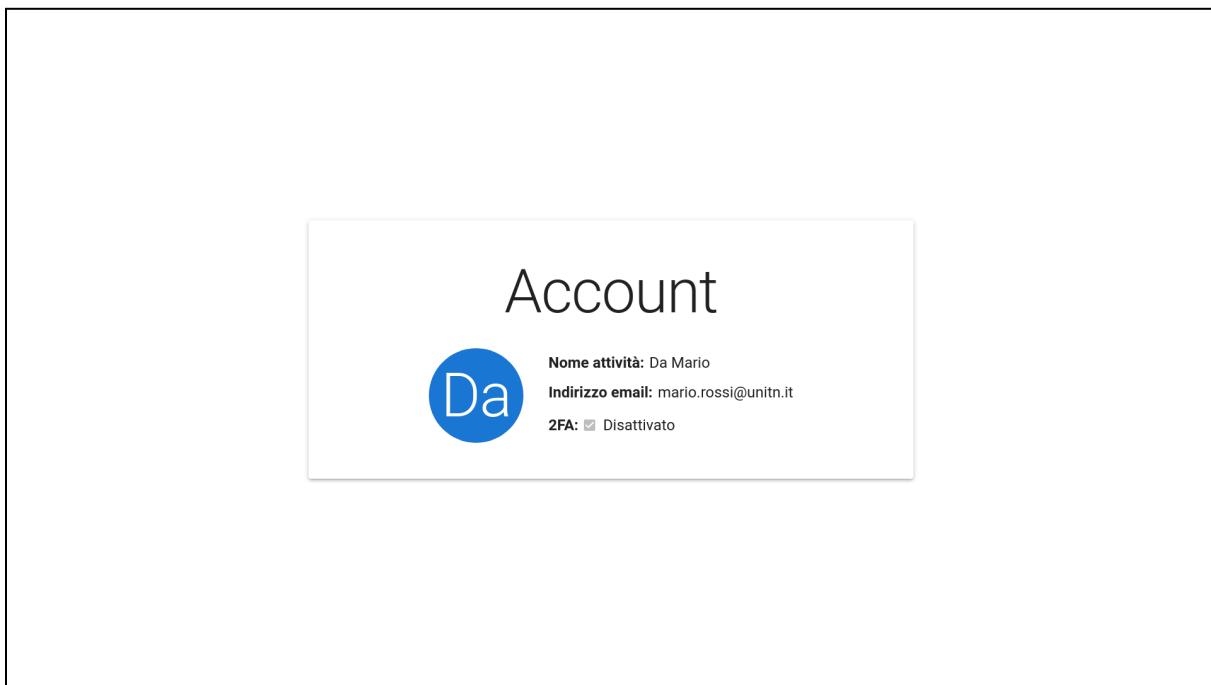
✖ Menus

QR Codice QR

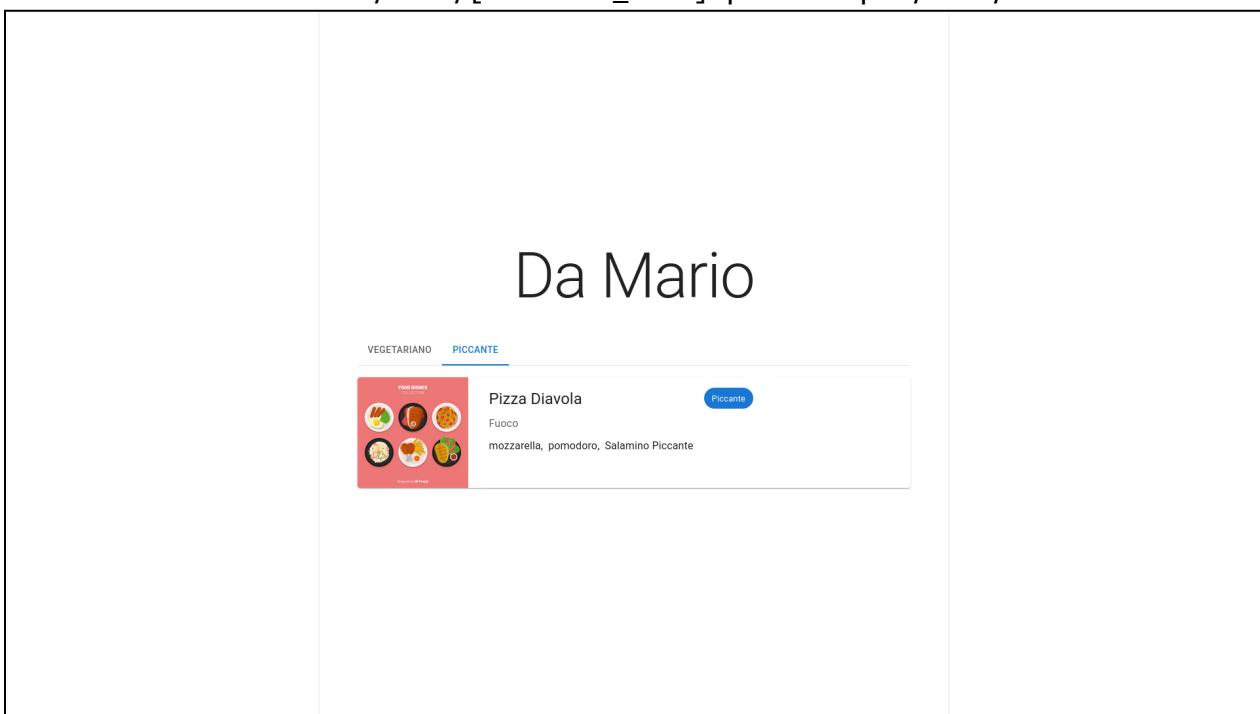
🕒 Menu attivo

Elementi				
	Nome	Descrizione	Ingredienti	Categorie
<input type="checkbox"/>	Pizza Margherita	La pizza Margherita è la tipica pizza napoletana	WIP - Lista degli Ingredienti	WIP - Lista delle Categorie
<input type="checkbox"/>	Pizza Diavola	Fuoco	WIP - Lista degli Ingredienti	WIP - Lista delle Categorie
<input type="checkbox"/>	Spritz Aperol	Il classico	WIP - Lista degli Ingredienti	WIP - Lista delle Categorie
<input type="checkbox"/>	Moscow Mule	Cocktail fresco	WIP - Lista degli Ingredienti	WIP - Lista delle Categorie

Nella pagina profilo l'utente può vedere le proprie informazioni e abilitare la funzione 2fa per il login.



Nella pagina menù attivo l'utente gestore può navigare tra le categorie del proprio menù attivo e vedere gli alimenti presenti. L'utente consumatore raggiunge questa pagina tramite lo stesso URL che utilizza l'utente gestore. La pagina generata dinamicamente è accessibile attraverso la route dinamica /menu/[business_name]. per esempio /menu/Da Mario.



Deployment in locale

Per avviare il progetto in locale, è necessario eseguire front-end e back-end.

Per lanciare back-end:

- Clonare la repository con il seguente comando:

```
git clone https://github.com/unitn-baccala/Back-end.git
```

- Seguire il seguente link:

```
https://send.bitwarden.com/#SEBx2xJmu0mctq-rAC260A/pVYRt3xitd5jxuJ4NPTnrw
```

Quindi inserire la password “baccala” per scaricare il file, rinominarlo a “.env” e spostarlo nella root della repository.

Questo file non è stato incluso nella repository perché contiene il JWT secret e le credenziali per accedere al database che sono dati sensibili che comprometterebbero la sicurezza del sistema se resi pubblici.

- Assicurandosi di avere la root della repository come working directory, eseguire il seguente comando per installare i moduli NodeJS necessari:

```
npm install
```

- Per lanciare il back-end sarà ora sufficiente eseguire il seguente comando:

```
npm start
```

- Per eseguire i test eseguire il seguente comando:

```
npm test
```

Per lanciare il front-end:

- assicurarsi di avere installato node >= 18.14.0

- Clonare la repository con il seguente comando:

```
git clone https://github.com/unitn-baccala/Front-end.git
```

- Assicurandosi di avere la root della repository come working directory, eseguire il seguente comando per installare i moduli NodeJS necessari:

```
npm install
```

ed il seguente per generare una build ottimizzata:

```
npm run build
```

- Per lanciare il front-end sarà ora sufficiente eseguire il seguente comando:

```
npm start
```

Una volta lanciati front-end e back-end il sito sarà raggiungibile a `localhost:8080`, e si potrà comunicare con le api a `localhost:3000`. Per velocizzare la visualizzazione dell'applicativo abbiamo preparato un profilo già pronto, le credenziali sono:

- email: mario.rossi@unitn.it
- password: PasswordSicura23

Testing

Gli strumenti utilizzati per il testing sono stati:

- Jest: un framework per il testing che si occupa di eseguire i test divisi in test suite e generare il report dei successi e insuccessi dei test e della copertura;
- Bent: un pacchetto NodeJS per eseguire richieste HTTP.

I test sono nella cartella `src/test/`, insieme a `request.js`, un file in cui sono state collezionate alcune funzioni per facilitare l'utilizzo di Bent e semplificare l'autenticazione.

File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Line #s
All files	100	100	100	100	
src	100	100	100	100	
app.js	100	100	100	100	
src/controllers	100	100	100	100	
dish.js	100	100	100	100	
ingredient.js	100	100	100	100	
user.js	100	100	100	100	
src/functions	100	100	100	100	
failureResponseHandler.js	100	100	100	100	
tokenChecker.js	100	100	100	100	
validateObjectIds.js	100	100	100	100	
src/models	100	100	100	100	
category.js	100	100	100	100	
dish.js	100	100	100	100	
ingredient.js	100	100	100	100	
menu.js	100	100	100	100	
user.js	100	100	100	100	
src/routes	100	100	100	100	
category.js	100	100	100	100	
dish.js	100	100	100	100	
ingredient.js	100	100	100	100	
menu.js	100	100	100	100	
user.js	100	100	100	100	
src/tests	100	100	100	100	
request.js	100	100	100	100	


```
Test Suites: 5 passed, 5 total
Tests:       66 passed, 66 total
Snapshots:   0 total
Time:        11.39 s
Ran all test suites.
```

Coverage dei casi di test. Non sono stati fatti test sulle API relative a categorie e menù, che sono per questo state escluse dal coverage, e sono state escluse dal coverage le linee di codice relative agli errori del database che non possono essere riprodotti.

```

const mongoose = require('mongoose');
const request = require('./request'); //for http requests
const app = require('../app');

const api_path = '/api/user/login';

const post = request.post(api_path);

const invalid_credentials = [
  [ 404, { email: "not an email", password: "badpw" } ],
  [ 404, { email: "test.user@for.tests.com", password: "incorrectpw" } ],
  [ 400, { email: "test.user@for.tests.com", password: null } ],
  [ 400, { password: "VeryGoodPassword!" } ],
];
describe(api_path, () => {
  test("successful login", async () => {
    await request.init_test_auth();
  });

  test.each(invalid_credentials)("POST with jwt %d %o",
    async (code, data) => await post(code,data));

  afterAll(async () => {
    app.server.close();
    await mongoose.disconnect();
  });
});

```

Esempio di test suite: login