SETTEMBRE 2022

SAVE THE MEAL

Alessia Laghezza - Anna Metelli - Alessio Novel

UNIVERSITÀ DEGLI STUDI
DI TRENTO

Titolo del documento:

Sviluppo Applicazione

Indice generale

Scopo del documento

Il seguente documento illustra le informazioni necessarie allo sviluppo delle funzioni principali della web app “Save The Meal”. Viene presentata la documentazione per realizzare i servizi di acquisto e gestione dei Meal, della registrazione e del login degli utenti.

È riportata una descrizione dell’uso dell’applicazione tramite user flow da parte di fornitori ed utenti, per poi presentare le API necessarie alle principali funzioni del sistema (utilizzando l’API Model e il Modello delle risorse). Ogni API presentata è accompagnata da una descrizione delle proprie funzionalità, dalla propria documentazione e dai test effettuati su di essa. Un ultima sezione del documento è dedicata alle specifiche del Git Repository e al deployment dell’applicazione web.

1 User Flow

Nella seguente sezione del documento è riportato lo user flow che descrive le funzioni di “Save The Meal”.

Un utente si ritroverà inizialmente nella Home Page del sistema. Potrà, se lo desidera, modificare la visualizzazione della mappa cercando una nuova posizione o visualizzare la lista dei fornitori. Potrà creare un nuovo account (come fornitore o come utente). Potrà effettuare l’accesso al sistema con le proprie credenziali e, in base alla tipologia di utente, svolgere le proprie attività. Se è un utente potrà visualizzare lo stato dei propri acquisti, lasciare un feedback (scrivendone uno nuovo o modificandone uno già esistente) o tornare alla visualizzazione della mappa, dove potrà eventualmente effettuare un nuovo acquisto. Se è un fornitore potrà visualizzare i propri feedback, gestire le richieste di Meal, autenticare gli utenti al ritiro e gestire i Meal disponibili (aggiungendone o eliminandone).

Sempre dalla Home Page si potranno visualizzare i fornitori e, esclusivamente se si è effettuato l’accesso con le proprie credenziali, si potrà procedere alla visualizzazione dei Meal disponibili e procedere all’acquisto.

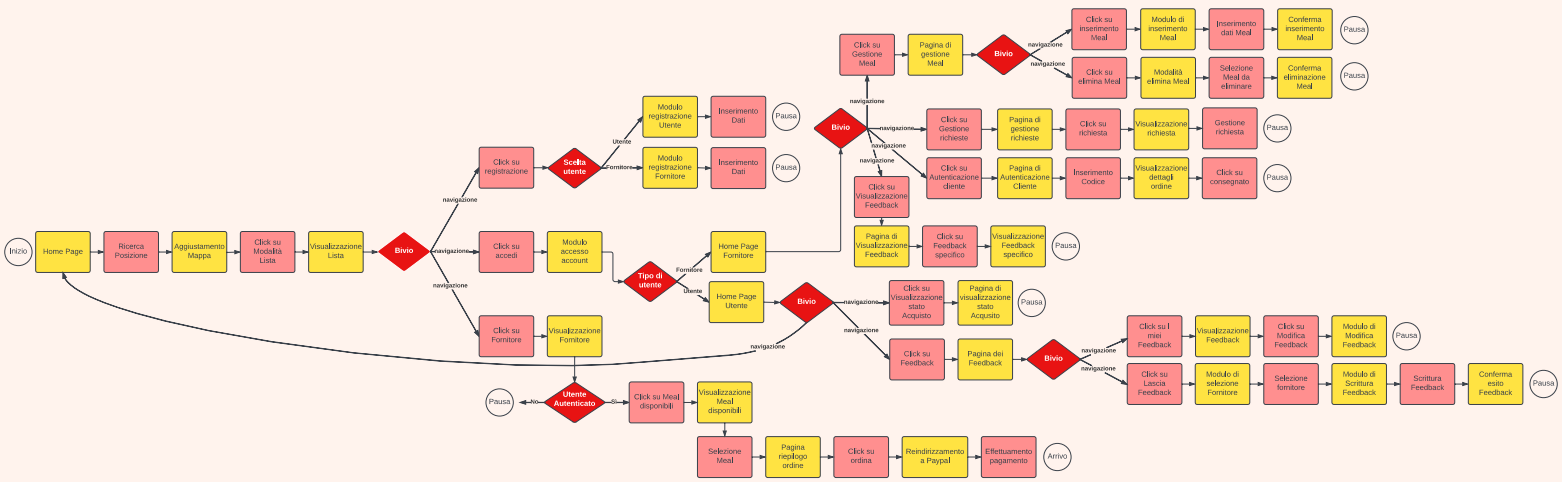

Fig 1. User flow diagram

2 Application Implementation and Documentation

Nelle sezioni precedenti sono state identificate le varie features che devono essere implementate per l'applicazione web con un'idea di come l'utente finale può utilizzarle. L'applicazione è stata sviluppata utilizzando HTML, JajaScript, CSS e NodeJS. Per la gestione dei dati è stato utilizzato MongoDB.

2.1 Project Structure

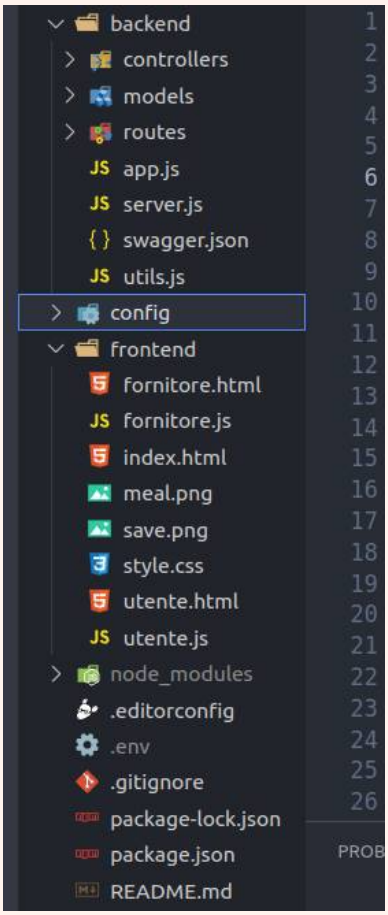

Figura 2.1.1: struttura del progetto

La struttura del progetto è presentata in Figura 1 ed è composta da due cartelle principali: backend e frontend.
Nella prima possiamo trovare tutte le API locali gestite e ordinate nelle cartelle controllers, routes, e models. Inoltre è presente un file swagger.json per la relativa documentazione delle API in swagger, un file utils.js dove sono presenti le funzioni comuni che vengono usate in più file, il file server.js che è il cuore dell'applicazione dove vengono gestite le chiamate ad altre cartelle ed infine il file app.js che viene usato per far partire l'applicazione con il relativo comando "nmp start".
Nella seconda cartella, ovvero quella del frontend, sono presenti tutte le funzionalità per la parte del front-end, con l'occupazione della parte grafica tramite css (nel file style.css), le tre schermate principali in html e con la memorizzazione di immagini a supporto. La dinamicità alle pagine HTML è stata aggiunta tramite JavaScript.

2.2 Project Dependecies

I seguenti moduli Node sono stati utilizzati e aggiunti al file Package.Json

```
"dotenv": "^16.0.3",
"express": "^4.18.2",
"jsonwebtoken": "^9.0.0",
"mongoose": "^6.8.2",
"swagger-ui-express": "^4.6.0",
"supertest": "^6.3.3",
"jest": "^29.3.1"
```

2.3 Project Data or DB

Per le gestione dei dati utilizzati all'interno dell'applicazione "Save The Meal" sono state delineate cinque strutture, come visibile nella figura presentata.

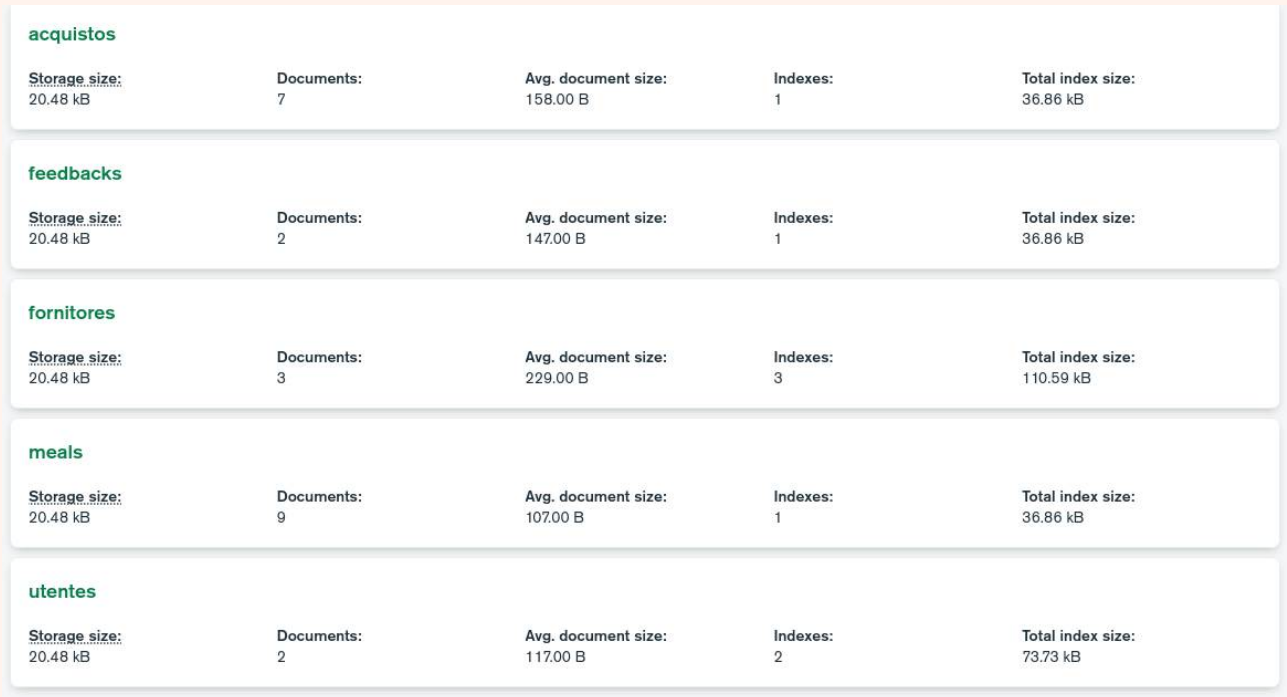

Figura 2.3.1: Collezione Dati usati nell'applicazione.

Per rappresentare gli utenti, i fornitori, gli acquisti, i feedbacks e i Meals sono stati definiti i tipi di dati riportati di seguito:

```
const utenteSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
  nome: {
    type: String,
  },
  cognome: {
    type: String,
  },
});
```

Figura 2.3.2: Tipo di dato - Utente

```
const fornitoreSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
  nomeAttivita: {
    type: String,
    unique: true,
    required: true,
  },
  indirizzoNegozio: {
    type: String,
    required: true,
  },
  tipologiaAlimenti: {
    type: String,
  },
  IBAN: {
    type: String,
    required: true,
  },
  immagine: {
    type: String,
  },
```

Figura 2.3.3: Tipo di dato - Fornitore

```
const acquistoSchema = new mongoose.Schema({
  meal: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "meals",
  },
  acquirente: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "utentes",
  },
  presenzaIntolleranze: Boolean,
  intolleranze: String,
  isPaid: Boolean,
  borsa: Boolean,
  stato: {
    type: String,
    default: "In attesa",
  },
});
```

Figura 2.3.4: Tipo di dato - Acquisto

```
const feedbackSchema = new mongoose.Schema({
  fornitore: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "fornitores",
    required: true,
  },
  utente: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "utentes",
    required: true,
  },
  valutazione: {
    type: Number,
    required: true,
  },
  puntiDiForza: {
    type: String,
  },
  commento: {
    type: String,
  },
});
```

Figura 2.3.5: Tipo di dato - Feedback

```
const mealSchema = new mongoose.Schema({
  fornitore: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "fornitores",
    required: true,
  },
  prezzo: {
    type: Number,
    required: true,
  },
  dimensione: {
    type: String,
    required: true,
  },
  disponibilita: {
    type: Boolean,
    required: true,
    default: true,
  },
});
```

Figura 2.3.6: Tipo di dato - Meal

2.4 Project APIs

2.4.1 Resources Extraction from the Class Diagram

Le principali risorse identificate a partire dal diagamma delle classi sono quelle di utente, fornitore, meal, acquisto e feedback.

La risorsa utente è caratterizzata da un attributo ID, generato da MongoDB in modo automatico, da un'email e una password decise dall'utente al momento della registrazione. Nella risorsa utente si potrà effettuare il login, registrare un nuovo utente, e ricavare tutti gli utenti registrati fino a quel momento.

La risorsa fornitore è caratterizzata da un ID, anche questi generato da MongoDB automaticamente, da un'email e una password, dai dati specifici dell'attività (nome, indirizzo, tipologia di alimenti, immagini) e dal codice IBAN forniti dall'utente in fase di registrazione. Nella risorsa fornitore sarà possibile registrare un nuovo fornitore, effettuare il login, recuperare uno o tutti i fornitori ed eliminare uno o tutti i fornitori registrati.

La risorsa meal è caratterizzata da un ID, generato automaticamente da mongoDB, dal fornitore che ne registra la disponibilità, da un prezzo, una dimensione (tra quelle predefinite dal sistema) e una disponibilità. Si potrà generare un nuovo Meal, recuperare tutti i Meal (di un fornitore o del sistema) ed eliminarli oppure recuperarne ed eliminarne uno in particolare.

La risorsa acquisto è definita da un ID, generato automaticamente da mongoDB, un Meal tra quelli esistenti nel sistema, un acquirente (tra gli utenti registrati), la presenza di intolleranze e una stringsa che le descrive, un attributo per controllare l'avvenuto pagamento, la presenza di una borsa e uno stato. Si potrà generare un nuovo acquisto e recuperarne uno solo o tutti, in più si potrà aggiornare lo stato di un acquisto.

La risorsa feedback è delineata da un ID, anch'esso come i precedenti generato automaticamente da MongoDB, un fornitore, un utente, una valutazione, dei punti di forza e un commento. Sarà possibile creare un nuovo feedback e ottenere tutti quelli esistenti (opzionalmente filtrati per fornitore e utente). Si potrà anche eliminare un feedback.

Le API di tipo POST nel sistema di "Save The Meal" sono quelle di login (di utente e di fornitore), dei vari new (newUtente, newFornitore, newMeal, newAcquisto, newFeedback) e hanno effetti sul back-end del sistema.

Le API di tipo GET sono, come si evince dal nome stesso, quelle che recupereranno uno o numerosi oggetti all'interno del sistema, quali getAllUtente, getAllFornitore, getOneFornitore, getAllMeal, getOneMeal, getAllAcquisto, getOneAcquisto, getAllFeedback. Avranno tutte effetto sul front-end del sistema.

Le API di tipo DELETE andranno a eliminare gli oggeti registrati nel sistema (deleteAllFornitore, deleteOneFornitore, deleteAllMeal, deleteOneMeal). Avranno effetto sul back-end del sistema.

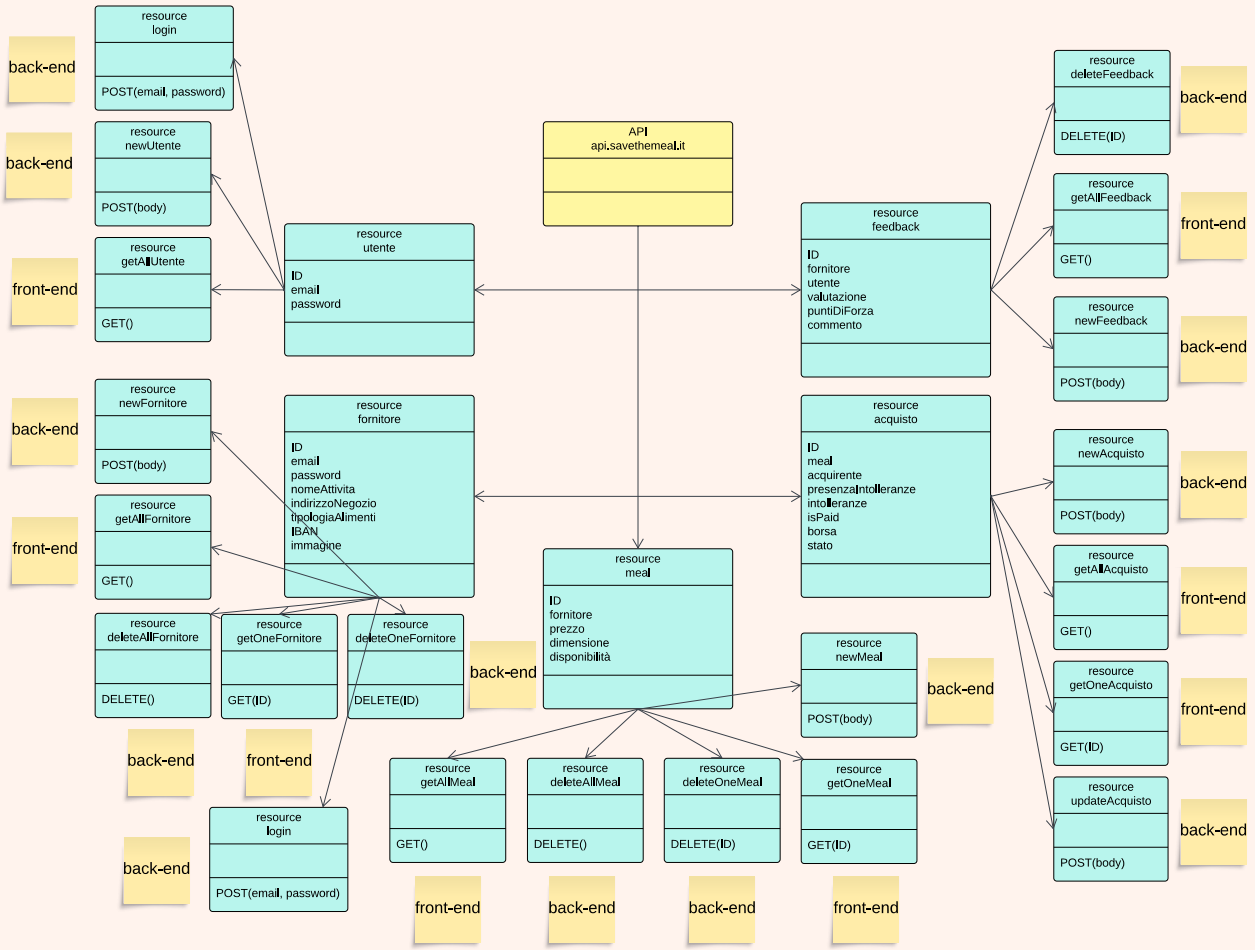

Fig 2.4.1.1 Resources extraction

2.4.2 Resources Models

Per ogni API sono stati identificati i body di richiesta e di risposta.

Per generare un nuovo Meal sono richieste le informazioni che lo andranno a creare, e in risposta si potrà avere la generazione di un Meal o un messaggio di errore. Per recuperare tutti i Meal esistenti è richiesto l'ID del fornitore e si otterrà un numero di Meal o un messaggio di errore. Eliminare tutti i Meal genererà un messaggio a indicare un esito positivo o negativo. Per recuperare uno specifico Meal si fornirà il suo ID e si otterrà il Meal specificato o un messaggio di errore. Per eliminare un singolo Meal si fornirà il suo ID e si otterrà un messaggio di errore o successo.

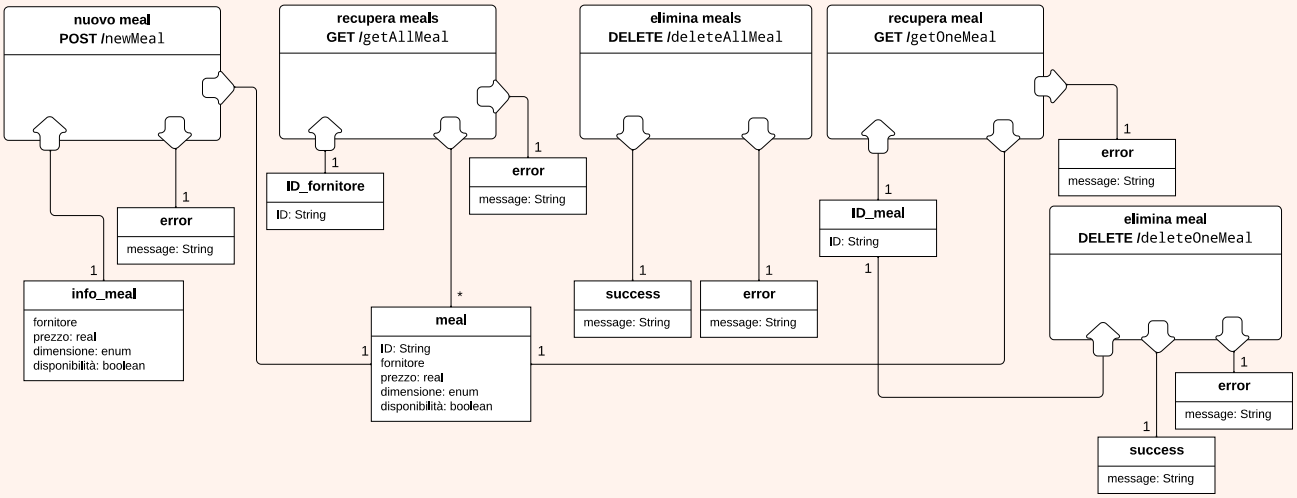

Fig 2.4.2.1 Resource Meal models

Il fornitore potrà effettuare il login al proprio account con le credenziali registrate, ottenendo un token se il processo è andato a buon fine o un messaggio di errore in caso contrario. Si registrerà un nuovo fornitore inserendo i suoi dati, ottenendo messaggi di errore o il fornitore stesso (in caso di successo). Si potranno ricavare tutti i fornitori, anche qui ottenendo i fornitori stessi in caso di successo o un messaggio di errore in caso contrario. Si potrà recuperare un singolo fornitore grazie al suo ID univoco (o ottenere un messaggio di errore in caso si insuccesso). Eliminare tutti i fornitori genererà un messaggio a indicare un esito positivo o negativo. Per eliminare un singolo fornitore si fornirà il suo ID e si otterrà un messaggio di errore o successo.

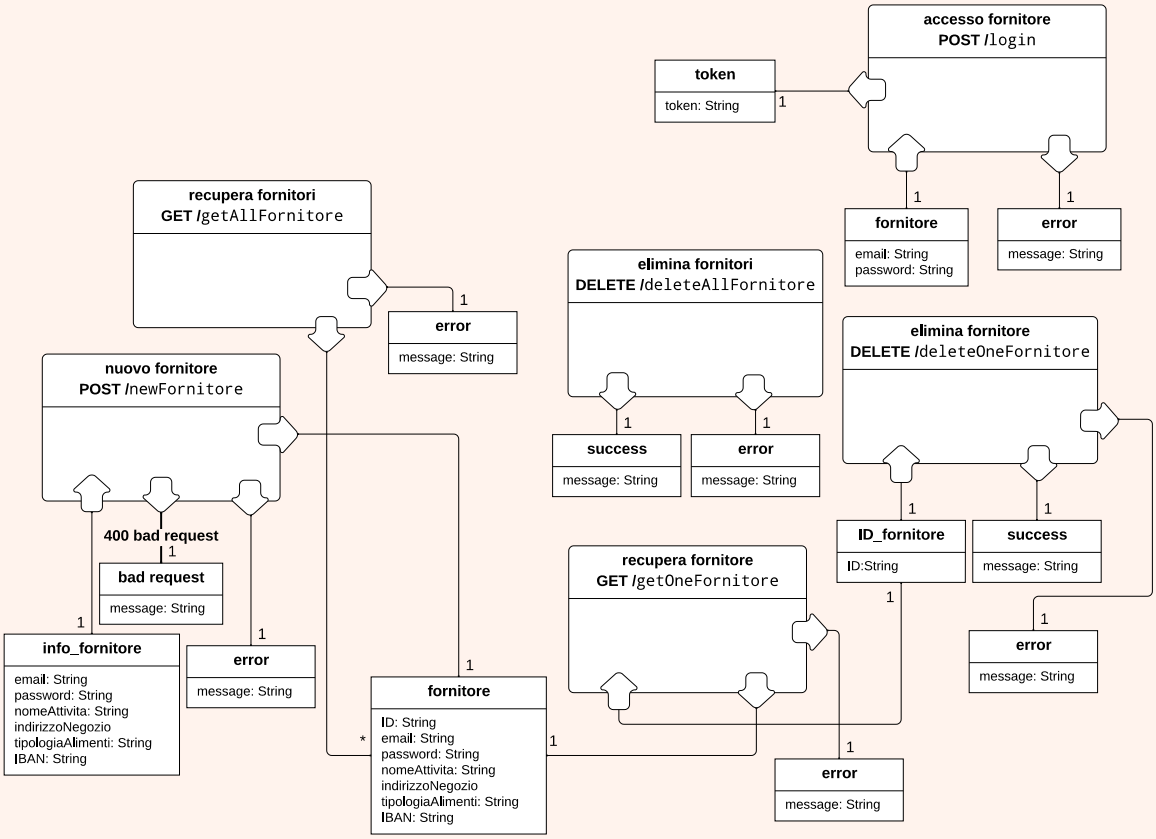

Fig 2.4.2.2 Resource fornitore models

Si potranno ottenere tutti i feedback di un fornitore grazie all'ID univoco di quest'ultimo, con un eventuale messaggio di errore. Si potrà generare un nuovo feedback inserendo i suoi dati (con eventuale messaggio di errore in caso di insuccesso). Fornendo l'ID del feedback è possibile eliminarlo, ottenendo un messaggio di errore o successo.

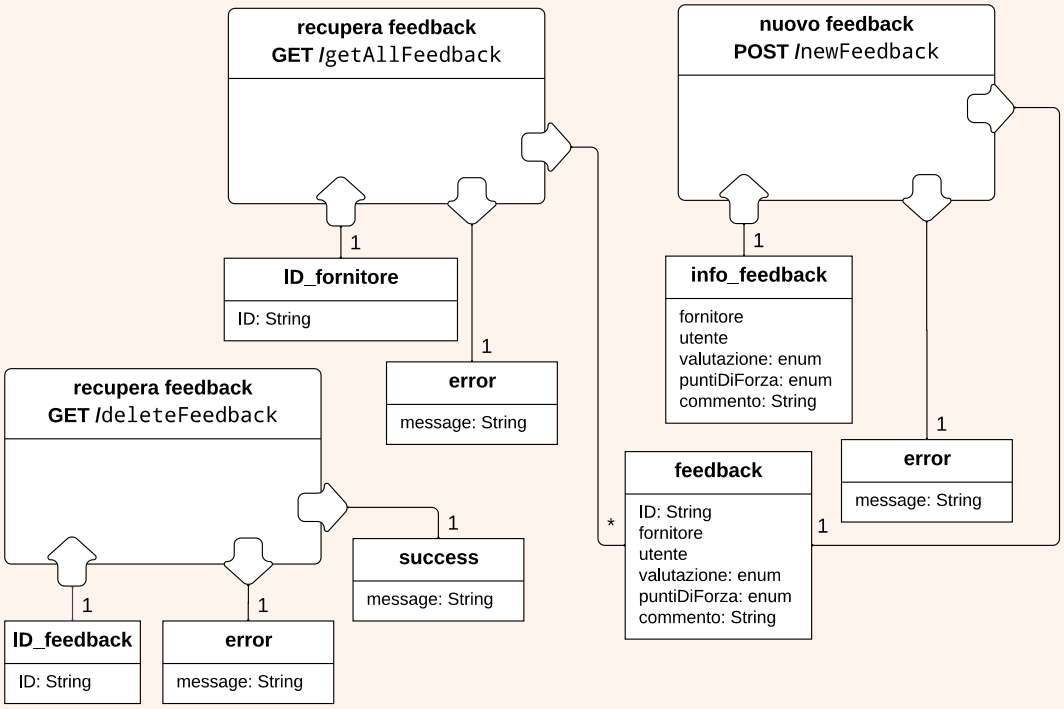

Fig 2.4.2.3 Resource feedback models

Si potrà generare un nuovo acquisto inserendo nel sistema le sue caratteristiche, con eventuali messaggi per comunicare gli errori. Fornendo l'ID di un utente è possibile recuperare tutti gli acquisti da lui effettuati o ottenere un messaggio di errore. Oppure è possibile recuperare un singolo acquisto grazie all'ID di un Meal, con eventuale messaggio di errore.

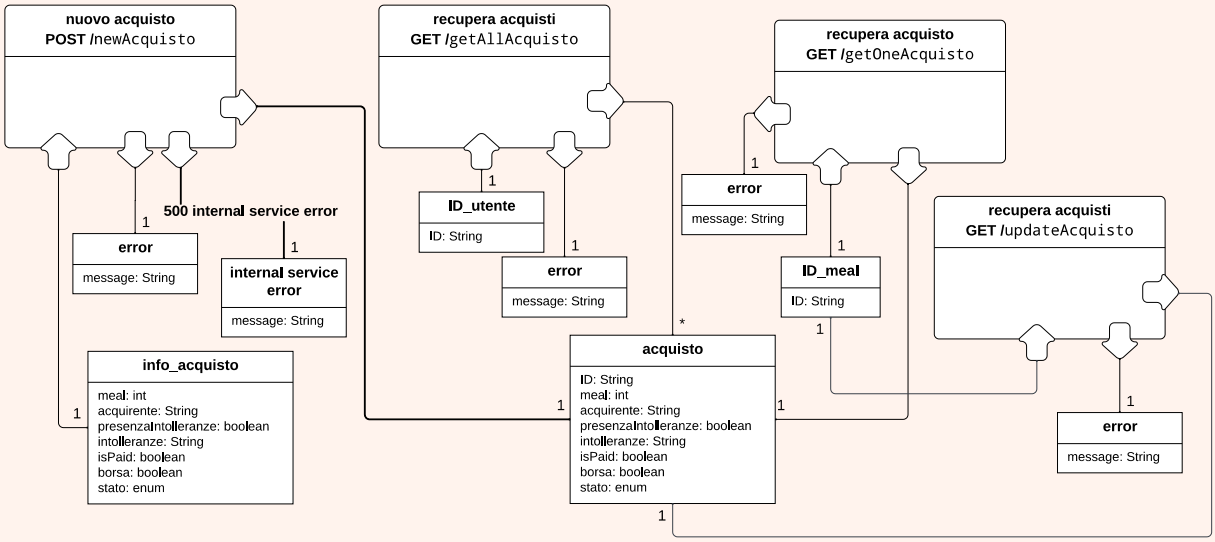

Fig 2.4.2.4 Resource acquisto models

Un utente potrà effettuare il login fornendo le sue credenziali, ottenendo un token in caso di esito positivo o un messaggio di errore in caso di esito negativo. Per generare un nuovo utente sarà

necessario fornire le nuove credenziali, ottenendo l'utente generato o eventuali messaggi di errore. Si potranno recuperare tutti gli utenti registrati nel sistema.

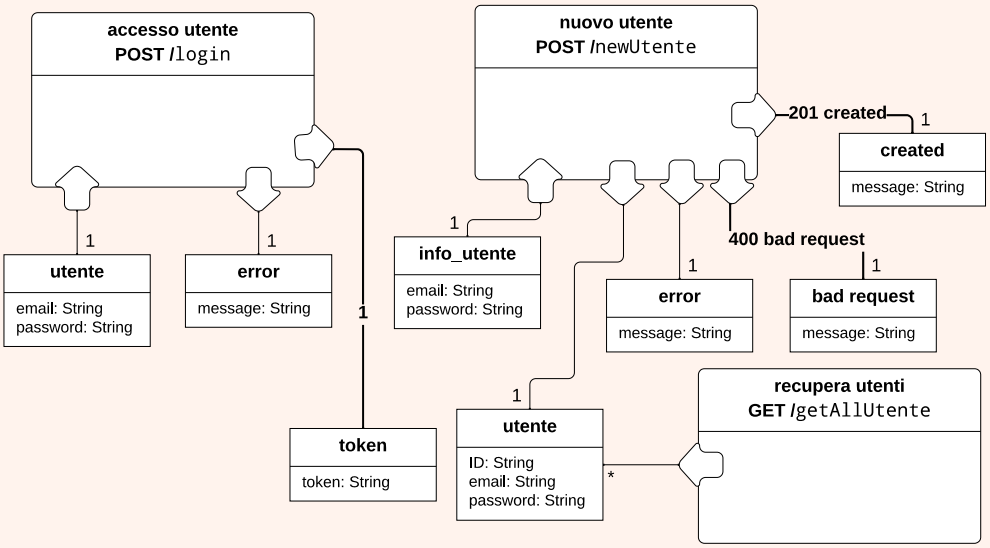

Fig 2.4.2.5 Resource utente models

2.5 Sviluppo API

2.5.1 POST /utente/login

Tramite questa API, che richiede come parametri la email a la password dell'utente, viene innanzitutto controllata l'esistenza di un utente con l'email inviata, successivamente viene verificata la correttezza della password.

L'API ritorna in risposta un oggetto di tipo utente con anche il token, che verrà riutilizzato più avanti per confermare l'autenticazione avvenuta.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const login = async function (req, res, next) {
  utente = req.body;
  if (utente.hasOwnProperty("email") && utente.hasOwnProperty("password")) {
    // find the user
    let user = await Utente.findOne({
      email: utente.email,
    }).exec();

    // user not found
    if (!user) {
      res.json({
        success: false,
        message: "Authentication failed. User not found.",
      });
    }

    // check if password matches
    else if (user.password != utente.password) {
      res.json({
        success: false,
        message: "Authentication failed. Wrong password.",
      });
    } else {
      // if user is found and password is right create a token
      var payload = {
        id: user._id,
        email: user.email,
        nome: user.nome,
        cognome: user.cognome,
        // other data encrypted in the token
      };
      var options = {
        expiresIn: 86400, // expires in 24 hours
      };
      var token = jwt.sign(payload, process.env.SUPER_SECRET, options);

      res.json({
        id: user._id,
        token: token,
        email: user.email,
        nome: user.nome,
        cognome: user.cognome,
        //self: "api/v1/" + user._id
      });
    }
  } else {
    return res.json({ message: "Utente object required" });
  }
};
```

2.5.2 POST /utente

Tramite questa API, che richiede come parametri un oggetto di tipo utente, viene innanzitutto controllata la correttezza della mail inserita, successivamente viene eseguito il controllo che non esista già un account utente creato con lo stesso indirizzo email. E infine l’account dell’utente viene creato e viene salvato sul database.

L’API ritorna in risposta un oggetto di tipo utente, quello appena creato.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const newUtente = (req, res, next) => {
  utente = req.body;
  if (
    utente.hasOwnProperty("email") &&
    utente.hasOwnProperty("password") &&
    utente.hasOwnProperty("nome") &&
    utente.hasOwnProperty("cognome")
  ) {
    if (
      typeof utente.email != "string" ||
      !Utils.checkIfEmailInString(utente.email)
    ) {
      res.json({
        message:
          'The field "email" must be a non-empty string, in email format',
      });
      return;
    }
    //check if the Utente name already exists in db
    Utente.findOne({ email: utente.email }, (err, data) => {
      //if Utente not in db, add it

      if (!data) {
        //create a new Utente object using the Utente model and req.body
        const newUtente = new Utente({
          email: utente.email,
          password: utente.password,
          nome: utente.nome,
          cognome: utente.cognome,
        });
        // save this object to database
        newUtente.save((err, data) => {
          if (err) return res.json({ Error: err });
          return res.json(data);
        });
        //if there's an error or the Utente is in db, return a message
      } else {
        if (err)
          return res.json(`Something went wrong, please try again. ${err}`);
        return res.json({ message: "User already exists" });
      }
    });
  } else {
    return res.json({ message: "Utente credentials required" });
  }
};
```

2.5.3 GET /utente

Tramite questa API, che non richiede parametri, vengono richiesti tutti gli utenti presenti nel database.

L’API ritorna in risposta un array di oggetti di tipo utente.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const getAllUtente = (req, res, next) => {
  Utente.find({}, (err, data) => {
    if (err) {
      return res.json({ Error: err });
    }
    return res.json(data);
  });
};
```

2.5.4 POST /fornitore/login

Tramite questa API, che richiede come parametri la email a la password del fornitore, viene innanzitutto controllata l’esistenza di un fornitore con l’email inviata, successivamente viene verificata la correttezza della password.

L’API ritorna in risposta un oggetto di tipo fornitore con anche il token, che verrà riutilizzato più avanti per confermare l’autenticazione avvenuta.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const login = async function (req, res, next) {
  utente = req.body;
  if (utente.hasOwnProperty("email") && utente.hasOwnProperty("password")) {
    // find the user
    let user = await Fornitore.findOne({
      email: utente.email,
    }).exec();

    // user not found
    if (!user) {
      res.json({
        success: false,
        message: "Authentication failed. User not found.",
      });
    }
    // check if password matches
    else if (user.password != utente.password) {
      res.json({
        success: false,
        message: "Authentication failed. Wrong password.",
      });
    } else {
      // if user is found and password is right create a token
      var payload = {
        email: user.email,
        nomeAttivita: user.nomeAttivita,
        indirizzoNegozio: user.indirizzoNegozio,
        // other data encrypted in the token
      };
      var options = {
        expiresIn: 86400, // expires in 24 hours
      };
      var token = jwt.sign(payload, process.env.SUPER_SECRET, options);

      res.json({
        token: token,
        id: user._id,
        email: user.email,
        nomeAttivita: user.nomeAttivita,
        indirizzoNegozio: user.indirizzoNegozio,
        //self: "api/v1/" + user._id
      });
    }
  } else {
    return res.json({ message: "Fornitore credentials required" });
  }
};
```

2.5.5 POST /fornitore

Tramite questa API, che richiede come parametri un oggetto di tipo fornitore, viene innanzitutto controllata la correttezza della mail inserita, successivamente viene eseguito il controllo che non esista già un account fornitore creato con lo stesso indirizzo email. E infine l’account del fornitore viene creato e viene salvato sul database.

L’API ritorna in risposta un oggetto di tipo fornitore, quello appena creato.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const newFornitore = (req, res, next) => {
  fornitore = req.body;
  if (
    fornitore.hasOwnProperty("email") &&
    fornitore.hasOwnProperty("password") &&
    fornitore.hasOwnProperty("nomeAttivita") &&
    fornitore.hasOwnProperty("indirizzoNegozio") &&
    fornitore.hasOwnProperty("tipologiaAlimenti") &&
    fornitore.hasOwnProperty("IBAN") &&
    fornitore.hasOwnProperty("immagine")
  ) {
    if (
      typeof fornitore.email != "string" ||
      !Utils.checkIfEmailInString(fornitore.email)
    ) {
      res.json({
        success: false,
        message:
          'The field "email" must be a non-empty string, in email format',
      });
      return;
    } else {
      //check if the Fornitore name already exists in db
      Fornitore.findOne(
        {
          $or: [
            { nomeAttivita: fornitore.nomeAttivita },
            { email: fornitore.email },
          ],
        },
        (err, data) => {
          //if fornitore not in db, add it

          if (!data) {
            //create a new fornitore object using the fornitore model and req.body
            const nuovoFornitore = new Fornitore({
              email: fornitore.email,
              password: fornitore.password,
              nomeAttivita: fornitore.nomeAttivita,
              indirizzoNegozio: fornitore.indirizzoNegozio,
              tipologiaAlimenti: fornitore.tipologiaAlimenti,
              IBAN: fornitore.IBAN,
              immagine: fornitore.immagine,
            });
            // save this object to database
            nuovoFornitore.save((err, data) => {
              if (err) return res.json({ Error: err });
              return res.json(data);
            });
            //if there's an error or the fornitore is in db, return a message
          } else {
            if (err)
              return res.json(`Something went wrong, please try again. ${err}`);
            return res.json({ message: "Fornitore already exists" });
          }
        }
      );
    }
  } else {
    return res.json({ message: "Fornitore object required" });
  }
};
```

2.5.6 GET /fornitore

Tramite questa API, che non richiede parametri, vengono richiesti tutti i fornitori presenti nel database.

L’API ritorna in risposta un array di oggetti di tipo fornitore.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const getAllFornitore = (req, res, next) => {
  Fornitore.find({}, (err, data) => {
    if (err) {
      return res.json({ Error: err });
    }
    return res.json(data);
  });
};
```

2.5.7 DELETE /fornitore

Tramite questa API, che non richiede parametri, vengono eliminati tutti i fornitori presenti nel database.

L’API ritorna in risposta un messaggio di conferma.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const deleteAllFornitore = (req, res, next) => {
  Fornitore.deleteMany({}, (err) => {
    if (err) {
      return res.json({ message: "Complete delete failed" });
    }
    return res.json({ message: "Complete delete successful" });
  });
};
```

2.5.8 GET /fornitore/:id

Tramite questa API, che richiede come parametro l'id di un fornitore, viene richiesto al database il fornitore con quel specifico id.

L'API ritorna in risposta un oggetto di tipo fornitore.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const getOneFornitore = (req, res, next) => {
  id = req.params["id"]; //get the fornitore email
  //find the specific fornitore with that email
  Fornitore.findOne({ _id: id }, (err, data) => {
    if (err || !data) {
      return res.json({ message: "Fornitore doesn't exist." });
    } else return res.json(data); //return the fornitore object if found
  });
};
```

2.5.9 DELETE /fornitore/:id

Tramite questa API, che richiede come parametro l'id di un fornitore, viene prima controllato che esista un fornitore con quel id, e poi il fornitore con quel id viene eliminato dal database.

L'API ritorna in risposta un messaggio di conferma.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const deleteOneFornitore = (req, res, next) => {
  id = req.params["id"];
  Fornitore.findOne({ _id: id }, (err, data) => {
    if (err || !data) {
      return res.json({ message: "Fornitore doesn't exist." });
    } else
      Fornitore.deleteOne({ _id: id }, (err) => {
        if (err) {
          return res.json({ message: "Complete delete failed" });
        }
        return res.json({ message: "Complete delete successful" });
      });
  });
};
```

2.5.10 POST /meal

Tramite questa API, che richiede come parametri un oggetto di tipo meal, viene innanzitutto controllato che esista il fornitore che sta inserendo il meal (tramite il parametro dell'oggetto di tipo meal). Successivamente il meal viene salvato sul database.

L'API ritorna in risposta un oggetto di tipo meal, quello appena creato.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const newMeal = (req, res, next) => {
  meal = req.body;
  if (
    meal.hasOwnProperty("fornitore") &&
    meal.hasOwnProperty("prezzo") &&
    meal.hasOwnProperty("dimensione")
  ) {
    if (!meal.hasOwnProperty("disponibilita")) {
      meal.disponibilita = true;
    }
    Fornitore.findOne({ _id: meal.fornitore }, (err, data) => {
      if (data) {
        //create a new fornitore object using the fornitore model and req.body
        //create a new meal object using the meal model and req.body
        const nuovoMeal = new Meal({
          id: meal._id,
          fornitore: meal.fornitore,
          prezzo: meal.prezzo,
          dimensione: meal.dimensione,
          disponibilita: meal.disponibilita,
        });
        // save this object to database
        nuovoMeal.save((err, data) => {
          if (err) return res.json({ Error: err });
          return res.json(data);
        });
      } else {
        if (err)
          return res.json(`Something went wrong, please try again. ${err}`);
        return res.json({ message: "The fornitore doesn't exists" });
      }
    });
  } else {
    return res.json({ message: "Meal object required" });
  }
};
```

2.5.11 GET /meal

Tramite questa API, che non richiede parametri, vengono richiesti tutti i meal presenti nel database.

Si può però specificare il parametro facoltativo "fornitore" che fa in modo di richiedere solo i meal appartenenti a quel specifico fornitore.

L'API ritorna in risposta un array di oggetti di tipo meal.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const getAllMeal = (req, res, next) => {
  if (req.query.hasOwnProperty("fornitore")) {
    fornitore = req.query.fornitore;
    Meal.find({ fornitore: fornitore }, (err, data) => {
      if (err || !data || data.length == 0) {
        return res.json({ message: "Meals don't exist from this supplier." });
      } else return res.json(data); //return the meal object if found
    });
  } else {
    Meal.find({}, (err, data) => {
      if (err) {
        return res.json({ Error: err });
      }
      return res.json(data);
    });
  }
};
```

2.5.12 DELETE /meal

Tramite questa API, che non richiede parametri, vengono eliminati tutti i meal presenti nel database.

L'API ritorna in risposta un messaggio di conferma.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const deleteAllMeal = (req, res, next) => {
  Meal.deleteMany({}, (err) => {
    if (err) {
      return res.json({ message: "Complete delete failed" });
    }
    return res.json({ message: "Complete delete successful" });
  });
};
```

2.5.13 GET /meal/:id

Tramite questa API, che richiede come parametro l'id di un meal, viene richiesto al database il meal con quel specifico id.

L'API ritorna in risposta un oggetto di tipo meal.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const getOneMeal = (req, res, next) => {
  id = req.params["id"]; //get the meal ID
  //find the specific meal with that ID
  Meal.findOne({ _id: id }, (err, data) => {
    if (err || !data) {
      return res.json({ message: "Meal doesn't exist." });
    } else return res.json(data); //return the meal object if found
  });
};
```

2.5.14 DELETE /meal/:id

Tramite questa API, che richiede come parametro l'id di un meal, viene prima controllato che esista un meal con quel id, e poi il meal con quel id viene eliminato dal database.

L'API ritorna in risposta un messaggio di conferma.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const deleteOneMeal = (req, res, next) => {
  id = req.params["id"];
  Meal.findOne({ _id: id }, (err, data) => {
    if (err || !data) {
      return res.json({ message: "Meal doesn't exist." });
    } else
      Meal.deleteOne({ _id: id }, (err) => {
        if (err) {
          return res.json({ message: "Complete delete failed" });
        }
        return res.json({ message: "Complete delete successful" });
      });
  });
};
```

2.5.15 POST /acquisto

Tramite questa API, che richiede come parametri un oggetto di tipo acquisto, viene innanzitutto eseguita una JOIN tra i meal che hanno id uguale al meal della richiesta e gli acquisti presenti sul db per controllare che il meal abbia disponibilità uguale a “true” e che gli acquisti associati a quel meal siano stati rifiutati (altrimenti non si può acquistare quel meal), controllando anche che esista un meal con quel id. Successivamente viene anche controllato che esista un utente con id uguale a quello della richiesta. Alla fine l’acquisto viene salvato sul database e viene aggiornata la disponibilità del Meal in “false”.

L’API ritorna in risposta un oggetto di tipo acquisto, quello appena creato.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const newAcquisto = (req, res, next) => {
  acquisto = req.body;
  if (
    acquisto.hasOwnProperty("meal") &&
    acquisto.hasOwnProperty("acquirente") &&
    acquisto.hasOwnProperty("presenzaIntolleranze") &&
    acquisto.hasOwnProperty("intolleranze") &&
    acquisto.hasOwnProperty("isPaid") &&
    acquisto.hasOwnProperty("borsa") &&
    acquisto.hasOwnProperty("stato")
  ) {
    Meal.aggregate([
      {
        $lookup: {
          from: "acquistos",
          localField: "_id",
          foreignField: "meal",
          as: "acquisto",
        },
      },
      {
        $unwind: {
          path: "$acquisto",
          preserveNullAndEmptyArrays: true,
        },
      },
      { $match: { _id: mongoose.Types.ObjectId(acquisto.meal) } },
    ]).exec(function (err, data) {
      disponibile = true;
      for (let i = 0; i < data.length; i++) {
        let meal = data[i];
        if (
          meal.disponibilita == false ||
          (meal.hasOwnProperty("acquisto") &&
            meal.acquisto.stato != "rifiutato")
        ) {
          disponibile = false;
        }
      }
      if (disponibile && data.length > 0) {
        Utente.findOne({ _id: acquisto.acquirente }, (err, data) => {
          if (data) {
            //create a new acquisto object using the acquisto model and req.body
            const newAcquisto = new Acquisto({
              meal: acquisto.meal,
              acquirente: acquisto.acquirente,
              presenzaIntolleranze: acquisto.presenzaIntolleranze,
              intolleranze: acquisto.intolleranze,
              isPaid: acquisto.isPaid,
              borsa: acquisto.borsa,
              stato: acquisto.stato,
            });
            // save this object to database
            newAcquisto.save((err, data) => {
              if (err) {
                return res.json({ Error: err });
              } else {
                Meal.findOneAndUpdate(
                  { _id: acquisto.meal },
                  { disponibilita: false },
                  { upsert: true },
                  function (err, doc) {
                    if (err) return res.send(500, { error: err });

                    return res.json(data);
                  }
                );
              }
            });
          } else {
            if (err)
              return res.json(`Something went wrong, please try again. ${err}`);
            return res.json({ message: "The user doesn't exists" });
          }
        });
      } else {
        return res.json({ message: "The meal isn't available" });
      }
    });
  } else {
    return res.json({ message: "Acquisto object required" });
  }
};
```

2.5.16 GET /acquisto

Tramite questa API, che non richiede parametri, vengono richiesti tutti gli acquisti presenti nel database.

Si può però specificare il parametro facoltativo “utente” che fa in modo di richiedere solo gli acquisti fatti da quel specifico utente.

L’API ritorna in risposta un array di oggetti di tipo acquisto.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const getAllAcquisto = (req, res, next) => {
  if (req.query.hasOwnProperty("utente")) {
    utente = req.query.utente;
    Acquisto.find({ acquirente: utente }, (err, data) => {
      if (err || !data || data.length == 0) {
        return res.json({
          message: "Purchases don't exist from this user.",
        });
      } else return res.json(data); //return the meal object if found
    });
  } else {
    Acquisto.find({}, (err, data) => {
      if (err) {
        return res.json({ Error: err });
      }
      return res.json(data);
    });
  }
};
```

2.5.17 GET /acquisto/:id

Tramite questa API, che richiede come parametro l'id di un acquisto, viene richiesto al database l' acquisto con quel specifico id.

L'API ritorna in risposta un oggetto di tipo acquisto.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const getOneAcquisto = (req, res, next) => {
  codiceID = req.params["id"]; //get the meal ID
  //find the specific meal with that ID
  Acquisto.findOne({ meal: codiceID }, (err, data) => {
    if (err || !data) {
      return res.json({ message: "Acquisto doesn't exist." });
    } else return res.json(data); //return the acquisto object if found
  });
};
```

2.5.18 POST /acquisto/:id

Tramite questa API, che richiede come parametro l'id di un acquisto e lo stato, viene aggiornato lo stato dell'acquisto con quel specifico in id con il nuovo stato inviato. Nel caso in cui il nuovo stato sia "rifiutato" aggiorniamo anche la disponibilità del meal, rimettendolo disponibile.

L'API ritorna in risposta un oggetto di tipo acquisto, quello appena aggiornato.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const updateAcquisto = (req, res, next) => {
  id = req.params["id"]; //get the meal ID
  acquisto = req.body;
  if (acquisto.hasOwnProperty("stato")) {
    stato = acquisto.stato;
    Acquisto.findOneAndUpdate({ _id: id }, { stato: stato }, (err, data) => {
      if (err || !data) {
        return res.json({ message: "Acquisto doesn't exist." });
      } else {
        if (stato == "rifiutato") {
          Meal.findOneAndUpdate(
            { _id: data.meal },
            { disponibilita: "true" },
            (err, data) => {
              if (err || !data) {
                return res.json({ message: "Meal doesn't exist." });
              } else {
              }
              return res.json(data); //return the acquisto object if found
            }
          );
        }
        return res.json(data); //return the acquisto object if found
      }
    });
  } else {
    return res.json({ message: "Stato required" });
  }
};
```

2.5.19 POST /feedback

Tramite questa API, che richiede come parametri un oggetto di tipo feedback, viene innanzitutto eseguito un JOIN tra Acquisto e Meal, filtrando gli acquirenti tramite l'attributo utente dell'oggetto feedback e filtrando i fornitori tramite l'attributo fornitore dell'oggetto feedback, così da controllare se l'utente ha effettuato un acquisto presso quel fornitore (in caso contrario non può scrivere un feedback). Successivamente controlliamo che l'utente non abbia già scritto un feedback a quel fornitore. Infine salviamo il feedback sul database

L'API ritorna in risposta un oggetto di tipo feedback, quello appena creato.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const newFeedback = (req, res, next) => {
  feedback = req.body;
  if (
    feedback.hasOwnProperty("fornitore") &&
    feedback.hasOwnProperty("utente") &&
    feedback.hasOwnProperty("valutazione") &&
    feedback.hasOwnProperty("puntiDiForza") &&
    feedback.hasOwnProperty("commento")
  ) {
    Acquisto.aggregate([
      {
        $lookup: {
          from: "meals",
          localField: "meal",
          foreignField: "_id",
          as: "mealData",
        },
      },
      {
        $unwind: {
          path: "$mealData",
          preserveNullAndEmptyArrays: true,
        },
      },
      {
        $match: {
          $and: [
            { acquirente: mongoose.Types.ObjectId(feedback.utente) },
            {
              "mealData.fornitore": mongoose.Types.ObjectId(feedback.fornitore),
            },
          ],
        },
      },
    ]).exec(function (err, data) {
      if (data.length > 0) {
        Feedback.findOne(
          { utente: feedback.utente, fornitore: feedback.fornitore },
          (err, data) => {
            if (!data) {
              const newFeedback = new Feedback({
                fornitore: feedback.fornitore,
                utente: feedback.utente,
                valutazione: feedback.valutazione,
                puntiDiForza: feedback.puntiDiForza,
                commento: feedback.commento,
              });
              newFeedback.save((err, data) => {
                if (err) return res.json({ Error: err });
                return res.json(data);
              });
            } else {
              if (err)
                return res.json(
                  `Something went wrong, please try again. ${err}`
                );
              return res.json({
                message: "You've already make a feedback for this store",
              });
            }
          }
        );
      } else {
        return res.json({
          message: "You can't make a feedback before buying from the store",
        });
      }
    });
  } else {
    return res.json({ message: "Feedback object required" });
  }
};
```

2.5.20 GET /feedback

Tramite questa API, che non richiede parametri, vengono richiesti tutti i feedback presenti nel database.

Si può però specificare il parametro facoltativo “fornitore” che fa in modo di richiedere solo i feedback scritti a quel specifico fornitore.

Si può però specificare anche il parametro facoltativo “utente” che fa in modo di richiedere esattamente il feedback scritto da quell’utente a quel fornitore.

L’API ritorna in risposta un array di oggetti di tipo feedback.

Nel caso in cui ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const getAllFeedback = (req, res, next) => {
  if (
    req.query.hasOwnProperty("fornitore") &&
    req.query.hasOwnProperty("utente")
  ) {
    fornitore = req.query.fornitore;
    utente = req.query.utente;
    Feedback.find({ fornitore: fornitore, utente: utente }, (err, data) => {
      if (err || !data) {
        return res.json({
          message: "Error.",
        });
      } else return res.json(data); //return the meal object if found
    });
  } else if (req.query.hasOwnProperty("fornitore")) {
    fornitore = req.query.fornitore;
    Feedback.find({ fornitore: fornitore }, (err, data) => {
      if (err || !data || data.length == 0) {
        return res.json({
          message: "Feedbacks don't exist from this supplier.",
        });
      } else return res.json(data); //return the meal object if found
    });
  } else {
    Feedback.find({}, (err, data) => {
      if (err) {
        return res.json({ Error: err });
      }
      return res.json(data);
    });
  }
};
```

2.5.21 DELETE /feedback/:id

Tramite questa API, che richiede come parametro l'id di un feedback, viene prima controllato che esista un feedback con quel id, e poi il feedback con quel id viene eliminato dal database.

L'API ritorna in risposta un messaggio di conferma.

Nel caso in cui i controlli non vadano a buon fine, o ci siano problemi con le chiamate al database, verrà ritornato un messaggio che specifica quale è stato il problema.

```
const deleteFeedback = (req, res, next) => {
  id = req.params["id"];
  Feedback.findOne({ _id: id }, (err, data) => {
    if (err || !data) {
      return res.json({ message: "Feedback doesn't exist." });
    } else
      Feedback.deleteOne({ _id: id }, (err) => {
        if (err) {
          return res.json({ message: "Complete delete failed" });
        }
        return res.json({ message: "Complete delete successful" });
      });
  });
};
```

3 API documentation

Le API Locali fornite dal sistema di "Save The Meal" e descritte nella sezione precedente sono state documentate con l'uso del modulo NodeJS nominato "Swagger UI Express", di modo che la documentazione relativa alle API fosse direttamente disponibile a chi può visualizzare il codice sorgente. In particolare, di seguito è mostrata la pagina web relativa alla documentazione che presenta tutte le API (GET, POST and DELETE) per la gestione dei dati dell'applicazione.

Per generare l'end point dedicato alla presentazione delle API è stato utilizzato Swagger Swagger UI, che permette di generare una pagina web dalle definizioni delle specifiche OpenAPI.

Di seguito è riportata la pagina web relativa alla documentazione che presenta le 21 API per la gestione dei dati all'interno di "Save The Meal".

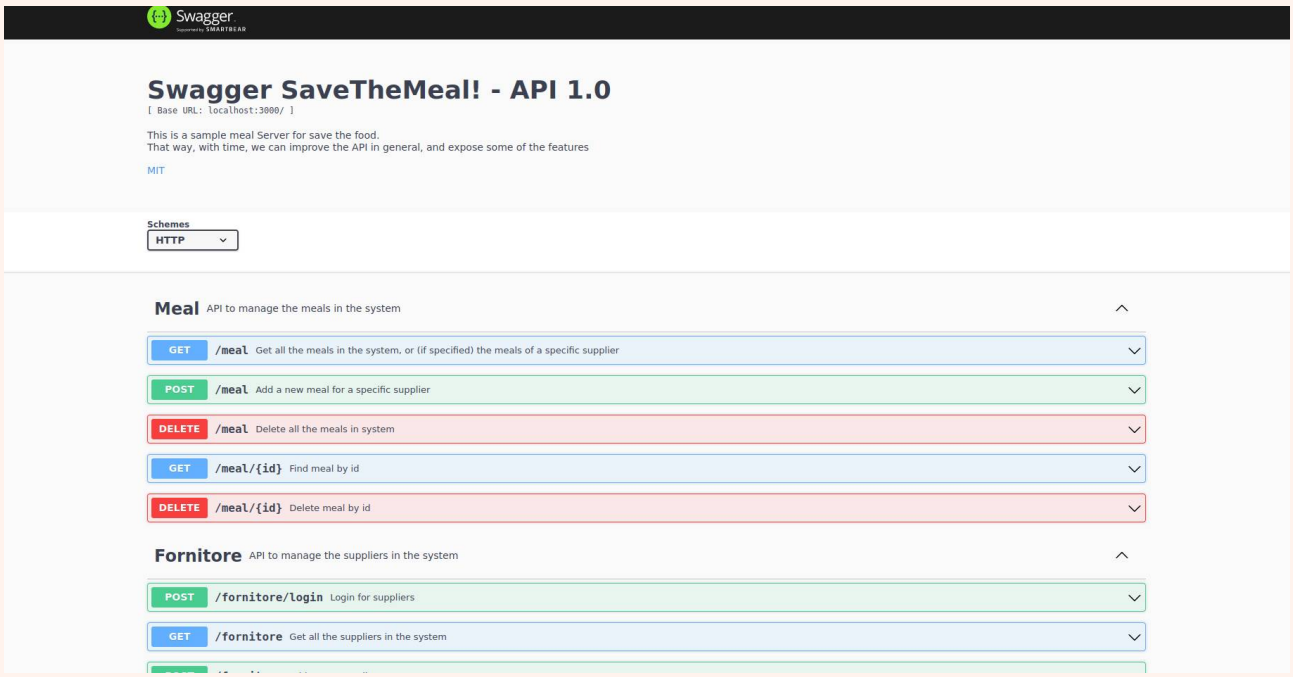

Figura 3.1: Documentazione delle API di SaveTheMeal

Per poter visuallizare la pagina di documentazione di swagger bisogna dare i seguenti comandi da terminale:

1. git clone https://github.com/SaveTheMeal/Web-API.git saveTheMeal
2. cd saveTheMeal
3. npm start

Successivamente si può visuallizzare la pagina da un browser al seguente link:

http://localhost:3000/api-docs/

4 FrontEnd Implementation

Il FrontEnd fornisce le funzionalità di visualizzazione, inserimento e cancellazione dei dati del sistema "Save The Meal". In particolare, l'applicazione è composta da una Home Page (dove è possibile scegliere se loggarsi come utente o come fornitore), da una pagina per la gestione degli utenti e una pagina per la gestione dei fornitori.

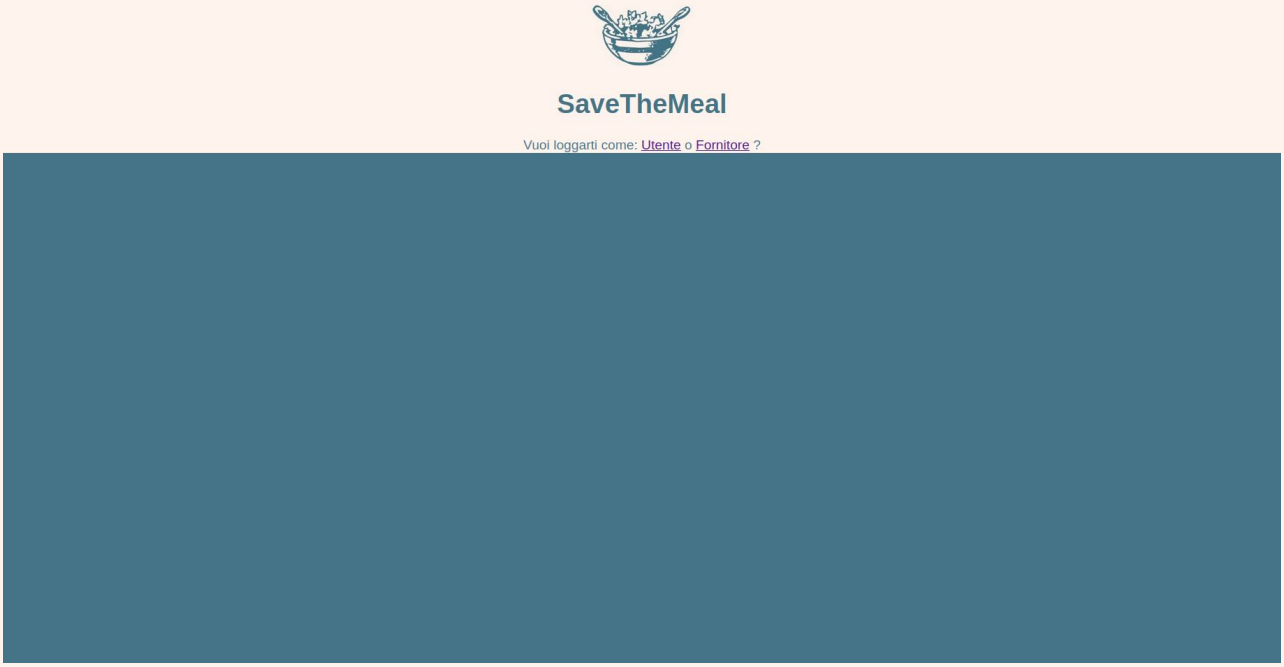

Figura 4.1: Home page dell'applicazione web SaveTheMeal

Nella Home page, sotto al logo del sistema, sono a disposizione due bottoni che guidano la navigazione all'interno dell'applicazione e vengono utilizzati per raggiungere la pagina dei fornitori e degli utenti.

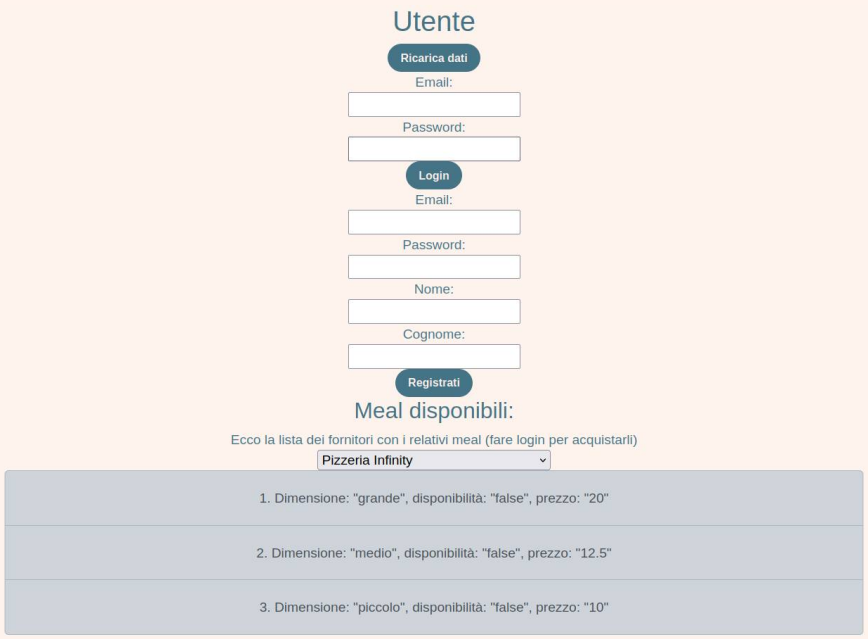

Figure 4.2: Pagina "Utente" dell'applicazione SaveTheMeal

La pagina dell'utente viene utilizzata per la gestione degli utenti in modo tale che possano loggarsi in caso abbiano già effettuato la registrazione oppure, in caso contrario, possano registrarsi inserendo i propri dati personali. Viene inoltre utilizzata per la visualizzazione dei Meal disponibili dei vari fornitori, scegliendo dal menù a tendina. Per procedere all'acquisto dei vari Meal l'utente deve eseguire il login. Dopo aver fatto ciò comparirà una schermata come quella visualizzata in figura.

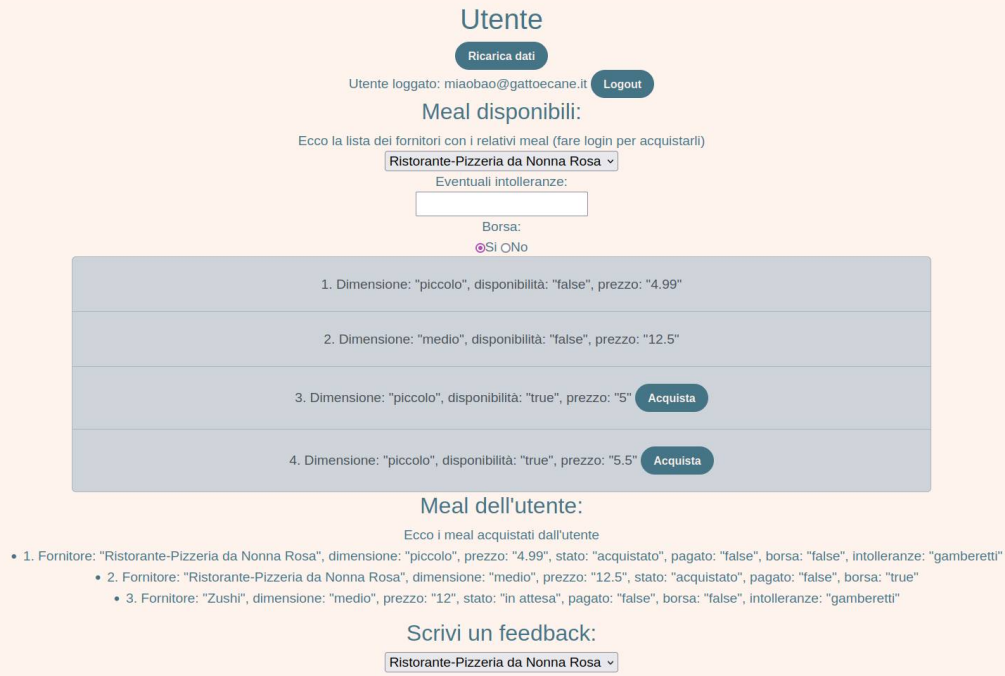

Figura 4.3: Pagina "Utente loggato" dell'applicazione web SaveTheMeal

In questa pagina l'utente autenticato può acquistare un Meal di un fornitore tra quelli disponibili alla vendita cliccando sul tasto "Acquista". In basso è presente una cronologia dei Meal già acquistati in passato dall'utente. È infine possibile scrivere una recensione ai negozi dei fornitori nei quali si ha acquistato almeno un Meal. La valutazione consiste in una scala da 0 a 5 punti, seguita da un commento personale.

Nella figura successiva è possibile visualizzare la pagina relativa al fornitore, dove anch'esso, se possiede già un account, può eseguire il login. In caso contrario può registrarsi inserendo i dati relativi al proprio punto vendita. È possibile visualizzare tutti i Meal di tutti i fornitori.

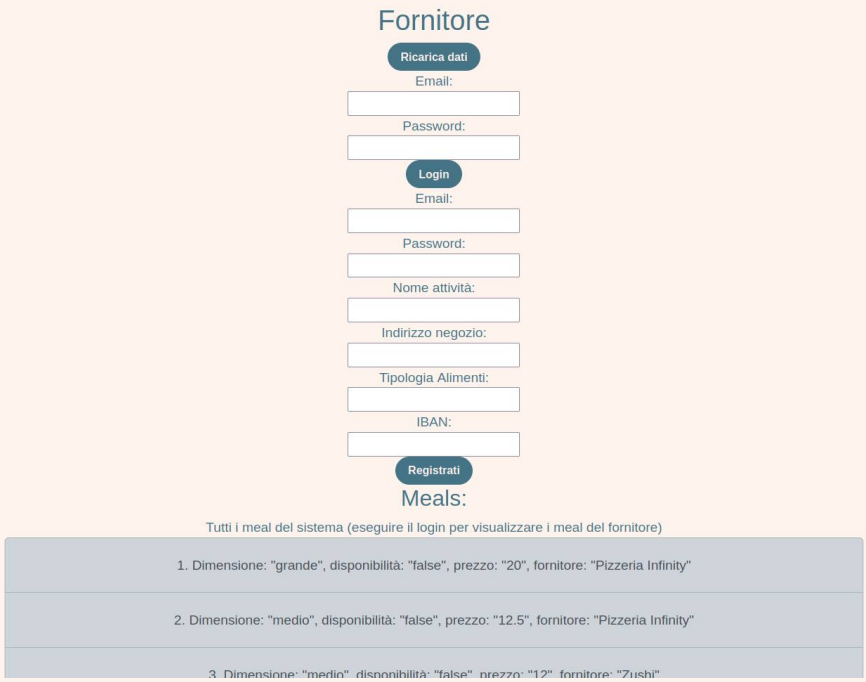

Figure 4.4: Pagina "Fornitore" dell'applicazione web SaveTheMeal

Una volta eseguito il login si visualizzerà una schermata come quella mostrata nella figura successiva. Qui il fornitore ha la possibilità di visualizzare i propri Meal e decidere se eliminarne uno che ancora non è stato venduto. Inoltre può inserire nel sistema nuovi Meal disponibili alla vendita, specificandone dimensione e prezzo. È possibile gestire le richieste d'acquisto, e visualizzare i feedback relativi al proprio punto vendita effettuati dagli acquirenti.

Figure 4.5: Pagina "Fornitore loggato" dell'applicazione web SaveTheMeal

5 GitHub Repository and Deployment Info

Il progetto “SaveTheMeal” su gitHub è suddiviso in quattro cartelle:

- Documents, la quale contiene tutti i documenti del progetto: Documento di progetto, Documento di architettura, Documento di specifica dei requisiti e Documento di sviluppo dell’applicazione;
- Web-API, la quale contiene tutto il codice relativo al progetto, diviso in due sottocartelle Front-end e Back-end che contengono lo sviluppo delle API e della struttura del sito;
- Info, la quale contiene una tabella in cui abbiamo inserito, volta per volta, tutte le ore che abbiamo passato lavorando sul progetto;
- Deriverables, la cartella che contiene tutte le consegne per il progetto di ingegneria del software.

Abbiamo eseguito il deployment su Heroku, il risultato è raggiungibile tramite il seguente link:

savethemeal

6 Testing

Per la parte di testing abbiamo lavorato tramite la libreria npm “jest”.

Abbiamo implementato una serie di test per la funzione che controlla che una stringa sia nel formato corretto per essere una mail valida, e li abbiamo inseriti nel file “utils.test.js”.

Questi test consistono nel verificare varie stringhe, di cui una valida e altre non valide per vari motivi, e verificare che la funzioni ritorni correttamente “true” oppure “false” in base alla correttezza.

```
const checkEmail = require("./utils").checkIfEmailInString;

describe("checkEmail test suite", () => {
  test("the string is an email", () => {
    const result = checkEmail("email@trento.it");
    expect(result).toBe(true);
  });

  test("the string isnt an email because misses the point", () => {
    const result = checkEmail("email@trento");
    expect(result).toBe(false);
  });
  test("the string isnt an email because misses the at", () => {
    const result = checkEmail("trento.it");
    expect(result).toBe(false);
  });
});
```

Abbiamo implementato una serie di test per verificare la correttezza della configurazione del server, e li abbiamo inseriti nel file “server.test.js”.

Questi test consistono nel verificare se il server è stato definito correttamente e viene poi effettuata una chiamata per verificare che lo stato del server sia quello atteso.

```
/**
 * https://www.npmjs.com/package/supertest
 */
const request = require("supertest");
const app = require("./app");

describe("should test server configuration", () => {
  test("app module should be defined", async () => {
    expect(app).toBeDefined();
  });

  test("GET / should return 200", async () => {
    return request(app).get("/").expect(200);
  });
});

```

Poi abbiamo implementato una serie di test per verificare la correttezza del router riguardante i meal, e li abbiamo inseriti nel file “meal.test.js”.

Questi test consistono nel verificare che per ogni tipo di chiamata venga chiamata la funzione giusta da parte del router Meal.

```
const mealController = require("../controllers/meal");

const getSpy = jest.fn();
const postSpy = jest.fn();
const deleteSpy = jest.fn();

jest.doMock("express", () => {
  return {
    Router() {
      return {
        get: getSpy,
        post: postSpy,
        delete: deleteSpy,
      };
    },
  };
});

describe("should test meal router", () => {
  test("should test post meals", () => {
    require("./meal");
    expect(postSpy).toHaveBeenCalledWith("/meal", mealController.newMeal);
  });
  test("should test get meals", () => {
    require("./meal");
    expect(getSpy).toHaveBeenCalledWith("/meal", mealController.getAllMeal);
  });
  test("should test delete meals", () => {
    require("./meal");
    expect(deleteSpy).toHaveBeenCalledWith("/meal", mealController.deleteAllMeal);
  });
});
```

Ora vediamo l’output dopo aver inviato il comando “npm test”:

```
alessionovel@AlessiosMacBook Web-API % npm test

> savethemeal@1.0.0 test
> jest --verbose --silent

 PASS  backend/routes/meal.test.js
  should test meal router
    ✓ should test post meals (280 ms)
    ✓ should test get meals (1 ms)
    ✓ should test delete meals

 PASS  backend/utils.test.js
  checkEmail test suite
    ✓ the string is an email (3 ms)
    ✓ the string isnt an email because misses the point (1 ms)
    ✓ the string isnt an email because misses the at

 PASS  backend/server.test.js
  should test server configuration
    ✓ app module should be defined (2 ms)
    ✓ GET / should return 200 (58 ms)

 PASS  backend/controllers/meal.test.js
  GET /meal
    ✓ GET / should return 200 (46 ms)

Test Suites: 4 passed, 4 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        4.801 s, estimated 9 s
```

Come possiamo vedere i test vengono divisi in base all’argomento che testano, e possiamo notare che abbiamo implementato tutto correttamente, in quanto tutti i test vanno a buon fine.