ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

PROGETTAZIONE E SVILUPPO DI UN'APPLICAZIONE MOBILE IBRIDA MEDIANTE L'UTILIZZO DEL FRAMEWORK IONIC 2

Relazione finale in
MOBILE WEB DESIGN

Relatore
Dott. Mirko Ravaioli

Presentata da
Giacomo Ranzi

Sessione III
Anno Accademico 2015/2016

INDICE

Introduzione

Lo scopo di questo progetto di tesi è quello di sviluppare un'applicazione per smartphone che migliori l'esperienza delle persone che seguono un determinato regime alimentare quando vanno a mangiare in un locale. In particolare mi interessa lo studio e l'implementazione di un'applicazione di tipo ibrido, per testare le potenzialità di questo tipo di tecnologia. Lo sviluppo dell'applicazione avverrà in maniera completa, dall'idea iniziale all'implementazione finale, occupandosi delle fasi di analisi e progettazione. Il nome scelto per l'applicazione è FoodMatch, che credo riassuma bene lo scopo che si propone di raggiungere.

1 - Il mondo del food nel mercato delle app

L'idea dello sviluppo di questo tipo di app è nato dal fatto che molte delle persone che conosco seguono, per i motivi più svariati, un qualche tipo di dieta che prevede restrizioni di vario tipo e a vari livelli, dall'eliminazione totale di un qualche tipo di prodotto per motivi etici alla semplice diminuzione di un certo alimento per motivi di salute.
Ci sono poi molti dati a supporto della tesi che negli ultimi anni il numero di persone che seguono un qualche tipo di dieta sia aumentato in maniera importante.
Per quanto riguarda gli stili di vita che rinunciano al consumo di carne e derivati animali, si stima che in Italia i vegetariani e i vegani siano circa l'8% della popolazione[1]. Guardando al fenomeno da una prospettiva più ampia, dati recenti indicano come il numero di persone che seguono una dieta di tipo vegetariano siano 15 milioni negli Stati Uniti, fra 10 e 50 milioni in Europa, e 375 milioni in India, dove l'adesione a questo stile di vita ha le sue radici nella tradizione religiosa [2].
L'altro principale motivo per cui le persone adottano abitudini restrittive nei confronti di certi tipi di alimenti sono le allergie e le intolleranze alimentari. Si stima che in Europa più di 17 milioni di persone soffrano di un qualche tipo di allergia alimentare[3].

Ad oggi esistono sul mercato numerose applicazioni smartphone legate al mondo del cibo: le più diffuse a livello mondiale sono probabilmente quelle legate al cosiddetto "food delivery", cioè le piattaforme di ordinazione e consegna a domicilio. Un'altra importante categoria riguarda le applicazioni che forniscono ricette, consigli e suggerimenti a chi

segue una particolare dieta. Infine esistono, soprattutto per quanto riguarda le grandi catene, applicazioni specifiche di ristoranti e fast food che permettono di visualizzare il menù in tempo reale e di ricevere offerte e notizie.
Quello che tutt'ora manca è quindi un'applicazione che aiuti le persone a capire quali sono, tra i piatti proposti in un menù cartaceo, quelli che più si adattano al loro personale stile alimentare. Ad oggi le informazioni di questo tipo di cui si può disporre si limitano spesso all'indicazione di piatti vegani e vegetariani con l'apposito simbolo sul menù, e solo a volte all'indicazione degli allergeni presenti. Rimane quindi compito del cliente accertarsi quali siano i piatti migliori e più adatti a lui, magari con l'assistenza di un cameriere. Questo progetto di tesi si propone così di creare un'applicazione che semplifichi questo compito alle persone che per i motivi più svariati hanno necessità di eliminare alcuni alimenti dalla loro dieta, anche in maniera occasionale e non necessariamente continuativa e duratura.

2 – Analisi e progettazione

Lo sviluppo di un'applicazione mobile inizia nello stesso modo di qualunque altro software, ovvero dalla progettazione, e più precisamente dall'analisi dei requisiti e dall'individuazione delle possibili problematiche. In questa prima fase si chiariscono gli obiettivi da raggiungere, cioè cosa materialmente l'applicazione dovrà fare, senza entrare nel merito di come dovrà essere fatto.

2.1 – Analisi dei requisiti e delle problematiche

I requisiti individuati sono i seguenti:

L'obiettivo dell'applicazione è quello di semplificare l'esperienza di consultazione di un menù in un locale che serve cibo da parte di utenti che hanno specifiche abitudini alimentari (ovvero di chiunque non sia semplicemente "onnivoro"). L'applicazione visualizzerà sul dispositivo una versione personalizzata del menù proposto dal locale in base alle preferenze dell'utente.

L'utente deve poter scannerizzare un QR Code posto su un menù di un locale ed avere

immediatamente una panoramica dei piatti proposti, filtrati in base alle sue esigenze, evitando quindi di dover sfogliare il menù cartaceo e leggerne attentamente tutte le voci per verificare quali piatti siano adatti al suo stile di vita e quali no.

Il software deve quindi essere in grado di memorizzare tutte le necessità e le abitudini dell'utente per quanto riguarda l'alimentazione. Devono essere prese in considerazione non solo quelle dovute a patologie o motivi di salute, ma anche le possibili scelte dovute a motivi etici, religiosi e così via. La piattaforma deve essere in grado di gestire le preferenze in maniera completa ma intuitiva. Le preferenze devono poter essere cambiate dall'utente in maniera semplice e le modifiche devono essere immediatamente operative.
Dopo la scansione di un QR Code associato ad un menù deve essere possibile visualizzare comunque tutti i piatti proposti, ma devono essere suddivisi in base alla compatibilità (questo perché se nessun piatto è compatibile con le necessità dell'utente, questo deve comunque poter visualizzare quali sono le proposte, per poter eventualmente chiedere di personalizzare uno dei piatti presenti).
Di ogni piatto sul menù devono poi poter essere visualizzati i dettagli in maniera completa (includendo per esempio una descrizione e il prezzo).
Deve ovviamente essere possibile registrare un nuovo utente tramite l'applicazione, ed è necessario che questa riconosca un utente già loggato quando viene aperta, per evitare che questo debba ogni volta reimmettere le sue credenziali. Un utente deve poter utilizzare l'applicazione con il suo profilo e le sue preferenze indipendentemente dal dispositivo che usa (è quindi richiesto che un utente possa effettuare il login sul suo smartphone, utilizzare l'applicazione, effettuare il logout e poi loggarsi su un altro dispositivo mantenendo le stesse funzionalità).

Le problematiche da gestire sono le seguenti:

Uno dei problemi principali da affrontare in fase di progettazione del software riguarda le preferenze dell'utente, che devono poter essere facilmente associate a ciascun piatto di un menù per verificarne la compatibilità, e devono essere in numero tale da poter garantire al maggior numero possibile di utenti di poter inserire il proprio profilo in maniera sufficientemente dettagliata. Allo stesso tempo non devono essere troppo numerose o troppo specifiche, perché il gestore che deve associarle ai piatti che propone potrebbe non conoscerne alcune, e quindi potrebbe sbagliare nella categorizzazione dei suoi piatti (o,

più banalmente, potrebbe risultare troppo impegnativo associare un numero molto elevato di caratteristiche a tutti i piatti, scoraggiando il locale ad usare l'applicazione).
Il secondo problema è quello di fornire ad ogni piatto proposto un livello di compatibilità relativo allo specifico utente, e quindi alle sue specifiche preferenze.
E' necessario utilizzare un sistema che sia affidabile ed immediato per riconoscere se un certo piatto è adatto all'utente oppure no.

2.2 – Architettura generale e pattern MVC

L'applicazione ha necessità di comunicare con un web server, che deve ricevere ed elaborare le richieste per poi recuperare le informazioni dal database, eventualmente elaborarle, e restituirle all'applicazione perché vengano visualizzate.

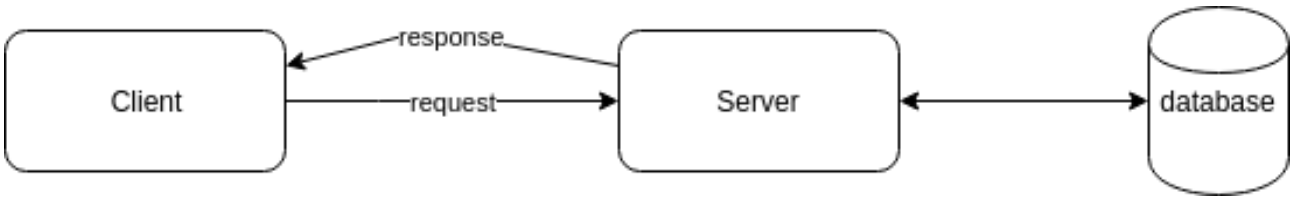

Figura 1: architettura generale client-server

Il server implementerà un framework che sfrutta il pattern MVC (Model-View-Controller) per la gestione delle richieste e la presentazione dei dati. Il server agisce quindi come livello intermedio fra il client ed il database, in quanto quest'ultimo non può ricevere ed elaborare direttamente le richieste.

Il controller si occupa di ricevere le richieste dal client, tramite la chiamata a specifiche funzioni (*action*) che avviene tramite protocollo HTTP.
Il controller poi esegue le operazioni richieste sul model, ed eventualmente elabora i dati ottenuti prima di restituire la risposta al client.
Il model è la parte che si occupa della modellazione e della gestione dei dati. Questo si interfaccia direttamente al database, e ne rispecchia la struttura, fornendo i metodi base di intervento sulle tabelle.
La view è la parte che si occupa di visualizzare i dati ottenuti dal model. In questo caso la parte relativa alla view è gestita dall'applicazione client.

Lo schema illustrato nella figura 1 può quindi essere ampliato per descrivere il pattern MVC del server come segue.

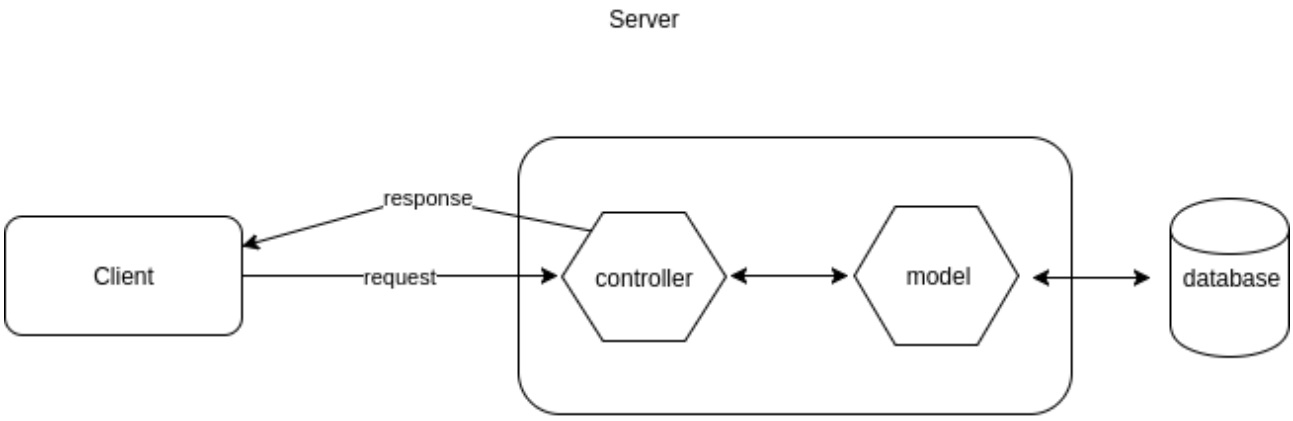

Figura 2: architettura client-server con pattern MVC

2.3 – Organizzazione dei dati

2.3.1 – Gestione dei tag

Il sistema di identificazione delle preferenze si basa su un sistema di tag: ci sono 18 tag disponibili, che vengono utilizzati sia in associazione ai piatti di un menù, sia in associazione agli utenti (in questo caso per identificare le loro preferenze alimentari).

I tag sono esposti di seguito, suddivisi in categorie per facilitarne la comprensione e il motivo della loro scelta.

Il primo gruppo di tag individua 8 delle più diffuse allergie alimentari esistenti, identificate dal *Food Allergen Labeling and Consumer Protection Act* (FALCPA) del 2004[4]. Nel mondo esistono più di 160 allergeni alimentari[5], ma gli 8 individuati sopra coprono più del 90% delle allergia alimentari riscontrate negli Stati Uniti. Sicuramente è possiblie ampliare questa lista (l'Unione Europea per esempio rende obbligatoria la segnalazione di 14 allergeni alimentari diversi). La scelta di questa lista piuttosto che di un'altra è dettata semplicemente dal bisogno di aderire ad uno standard riconosciuto ed affidabile.
I tag inseriti in questo gruppo sono:

- nuts-free
- peanuts-free
- soy-free
- sesame-free
- wheat-free
- milk-free
- fish-free
- shellfish-free
- egg-free

Come si può notare, tutti i tag indicano un'*assenza* di un particolare prodotto, cioè significa che un utente che si assegna uno di questi tag vuole visualizzare solo piatti che *non* contengono quel particolare alimento. Allo stesso modo, ciascun piatto associato ad uno di questi tag *non* contiene quel particolare alimento, ed è quindi sicuro per un utente che ha quel tag nelle sue preferenze.

Particolare attenzione va data ai tag *milk-free* e *wheat-free*, che riguardano l'assenza dello specifico prodotto, e non genericamente del lattosio (nel caso del tag milk-free) e del glutine (nel caso del tag wheat-free). Questo perché quella al lattosio ed al glutine sono intolleranze, mentre le allergie al latte ed al grano sono problematiche diverse, più specifiche e soprattutto separate. Una persona può essere intollerante al glutine senza essere allergica al grano e vice versa, e la stessa cosa vale per il latte ed il lattosio.
Il secondo gruppo di preferenze include appunto i due tag menzionati sopra, cioè quelli che individuano due delle intolleranze maggiormente diffuse.
I tag inseriti in questo gruppo sono quindi solo:

- gluten-free
- lactose-free

Anche in questo caso, i tag individuano l'*assenza* di un particolare prodotto all'interno del piatto, quindi il meccanismo è il medesimo visto sopra.

Il terzo gruppo di tag comprende i due tipi di dieta più diffusi al mondo dopo quella onnivora. I tag inseriti in questo gruppo sono:

- vegetarian
- vegan

Sicuramente ci sarebbero molte più diete specifiche inseribili (come quella pescetariana o quella ovo-vegetariana, o quella crudista), ma sono tutto sommato percentualmente poco rilevanti, soprattutto tenendo conto delle problematiche espresse in precedenza: inserire diete troppo specifiche e magari poco conosciute può essere controproducente, perché aumenta il rischio di errori nell'assegnazione dei tag da parte dei gestori dei locali.

Il quarto gruppo di tag include le due diete principali che si possono seguire per motivi religiosi. I tag in questo gruppo sono:

- kosher
- halal

Il tag kosher individua i piatti che seguono e rispettano i dettami alimentari imposti dall'Ebraismo, mentre il tag halal individua i piatti che seguono e rispettano i dettami alimentari imposti dall'Islam. A differenza di quanto avviene per i tag del gruppo precedente, qua il rischio di errori nell'assegnazione dei tag ai piatti dovuta alla poca conoscenza dei gestori è molto minore: le regolamentazioni della cucina kosher e halal sono infatti molto rigide e conosciute da chi le rispetta, quindi è improbabile (e tuttavia non impossibile) che un gestore inserisca erroneamente questi tag.

L'ultimo gruppo di tag individua più che altro preferenze alimentari generiche e spesso non restrittive come le precedenti, e sono:

- organic
- local
- seasonal

Rispetto alle precedenti infatti, queste preferenze sono spesso secondarie, ma sempre più diffuse, sia per moda che per una crescente consapevolezza dei consumatori rispetto alle tematiche ambientali, economiche e di salute.

2.3.2 – Gestione dell'utente

I dati utente da gestire sono, oltre ovviamente a email e password per la registrazione e il login, tutti i tag delle preferenze e l'importanza che l'utente assegna a ciascun tag. Sarà quindi necessario utilizzare una struttura dati in grado di gestire l'inserimento e la rimozione dei tag associati a ciascun utente.
Il sistema di autenticazione scelto prevede l'utilizzo di un *bearer token* di sicurezza legato allo specifico utente, che l'applicazione riceve e salva in locale quando viene effettuato con successo un login o una registrazione. Questo token poi è necessario per effettuare qualsiasi richiesta al server: una richiesta che non presenta un token di sicurezza valido associato ad un utente inserito nel database avrà come risposta dal server *401: unauthorized*.

Una volta che l'utente effettua il login, le sue informazioni vengono salvate in locale sul dispositivo, in modo che al successivo accesso non debba essere necessario effettuare nuovamente il login. Effettuando invece il logout tutte le informazioni salvate in locale vengono eliminate, ed è possibile effettuare un nuovo login, anche con un altro utente (o procedere ad una nuova registrazione).

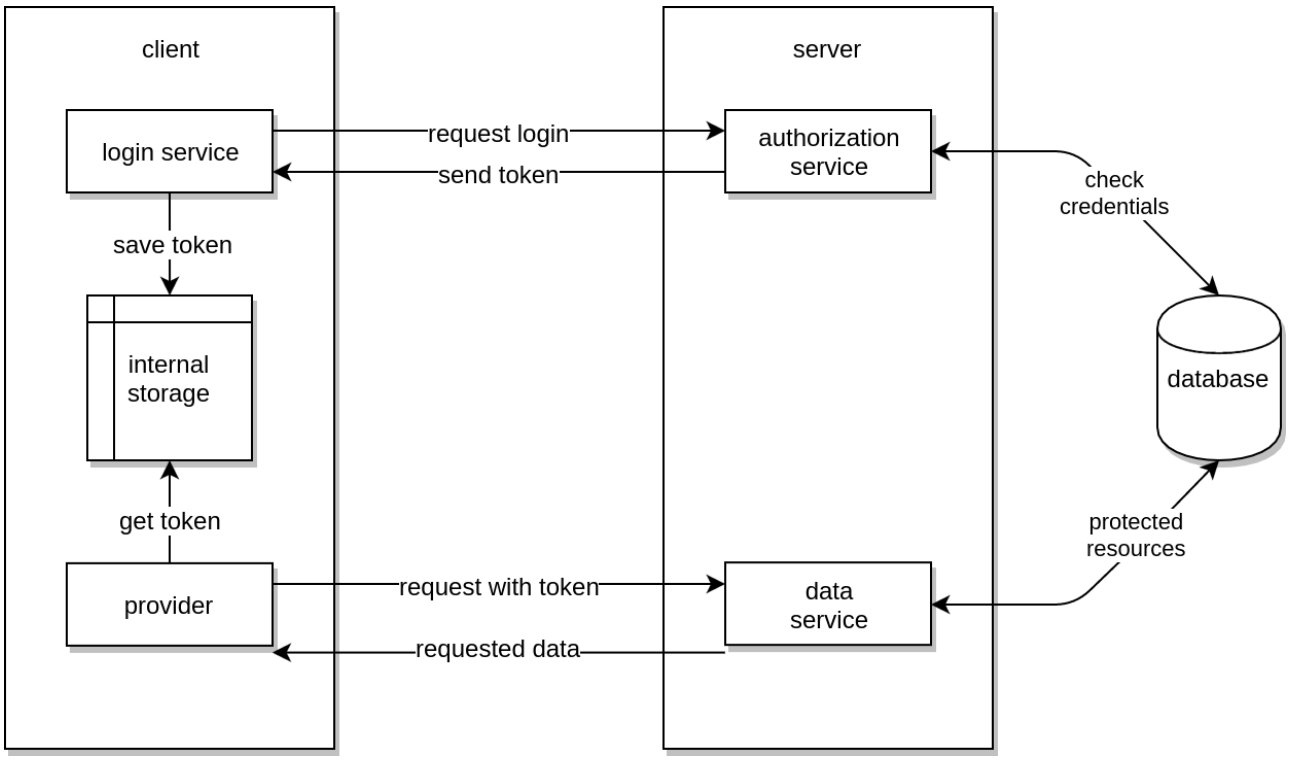

Figura 3: gestione dell'autorizzazione utente mediante l'uso di token

Di seguito sono riportati i mockup realizzati per le pagine di login e di registrazione:

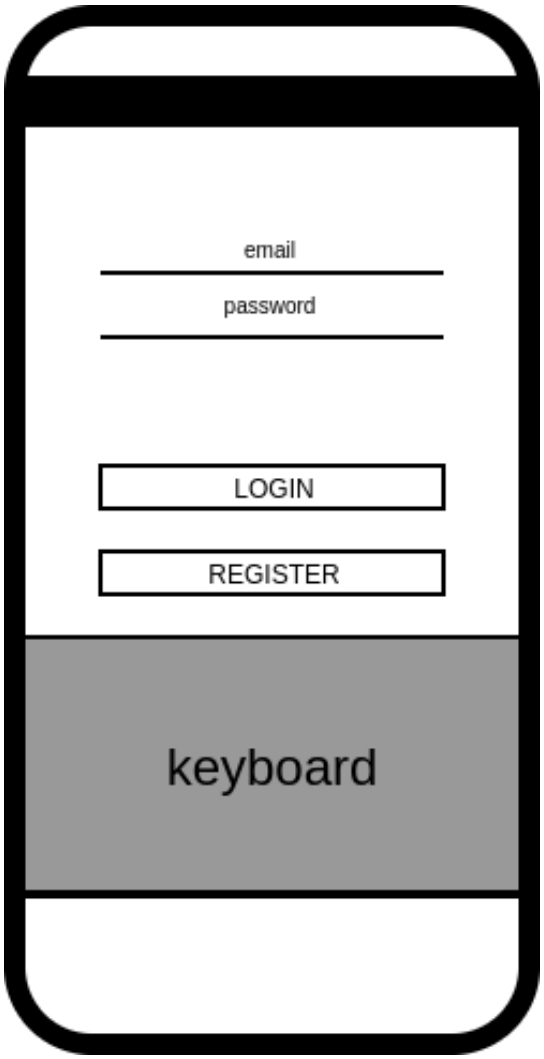

Figura 4: mockup della LoginPage

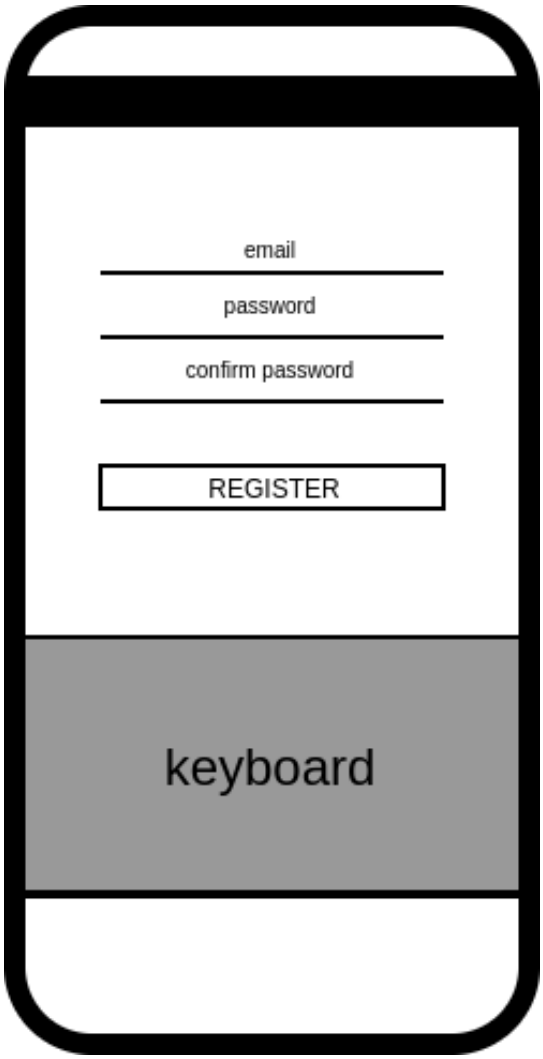

Figura 5: mockup della RegisterPage

2.3.3 – Gestione delle preferenze utente

Ogni piatto inserito in un menù può essere associato o meno a ciascuno dei tag elencati sopra (a seconda che abbia o meno le caratteristiche indicate dal tag). Le preferenze dell'utente (individuate dagli stessi tag) sono invece suddivise in tre livelli di importanza.

I tag inseriti tra le preferenze di primo livello sono quelli che l'utente ritiene obbligatori ed indispensabili. Per esempio, un utente che non può mangiare pesce a causa di un'allergia dovrà selezionare il relativo tag, con livello di preferenza 1. Un piatto a cui mancano uno o più tag di livello 1 verrà escluso dalla lista dei piatti ritenuti adatti all'utente, e viene

segnalato come “avoid”.
Il secondo livello di preferenze comprende i tag che l’utente ritiene molto importanti, ma non fondamentali (o a cui è disposto a rinunciare qualche volta). Per esempio, se un utente sa che per motivi di salute dovrebbe limitare il consumo di latticini, può inserire il tag relativo tra quelli di livello 2. Un piatto a cui mancano uno o più tag di livello 2 (ma ha tutti i tag di livello 1) viene segnalato come poco adatto all’utente (costituisce cioè un “bad match”).

Il terzo livello comprende i tag che costituiscono un “plus” , ma non sono fondamentali. Per esempio, se un utente preferirebbe mangiare prodotti locali e di stagione, ma è disposto a mangiare anche prodotti che non lo sono, può inserire i relativi tag nelle preferenze di livello 3. Un piatto che possiede tutti i tag di livello 1 e tutti i tag di livello 2 è considerato adatto all’utente (costituisce, cioè, un “match”); se, in aggiunta a questi, ha anche uno o più tag di livello 3, è considerato un “super match”, cioè un piatto particolarmente indicato per l’utente.

L’algoritmo dell’applicazione poi si occuperà di controllare quali sono i piatti del menù selezionato che sono più adatti all’utente, in base alle preferenze selezionate e ai livelli in cui queste sono state inserite.

Di seguito sono riportati i mockup realizzati per la pagina di gestione e modifica (visualizzazione, inserimento ed eliminazione) delle preferenze utente.

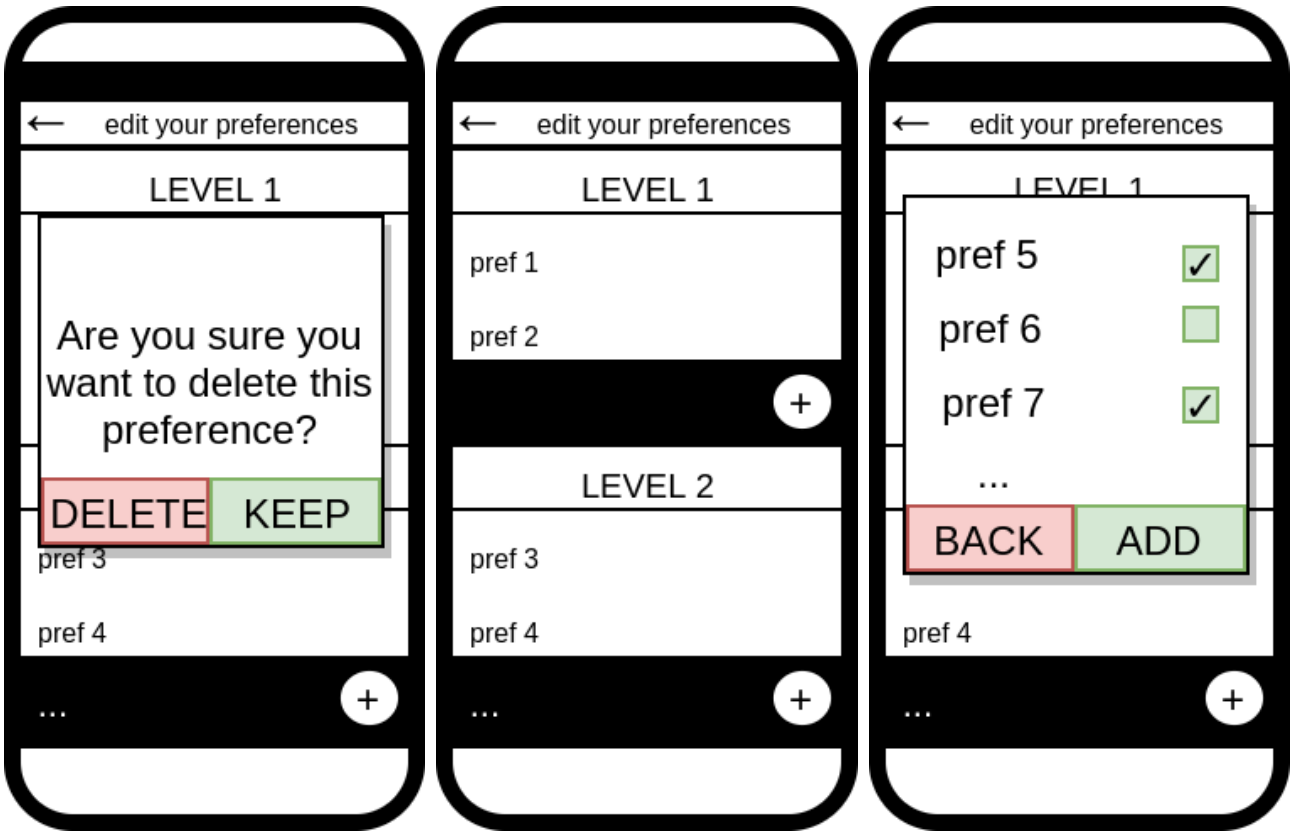

Figura 6: mockup della RegisterPage e dei menu per l'aggiunta e la rimozione delle preferenze

2.3.4 – Calcolo della compatibilità dei piatti e gestione del menù

Una delle parti fondamentali dell'applicazione è il calcolo della compatibilità di ciascun piatto di un menù con l'utente, ed è quindi molto importante progettare un algoritmo che la calcoli in maniera efficiente.
Il sistema usato si basa su un sistema di punteggio. Ogni piatto, come detto, è associato a diversi tag a seconda delle sue caratteristiche, ed ogni utente ha le sue preferenze suddivise su tre livelli. Il punteggio base di ogni piatto è pari a 0.

- Se ad un piatto manca un tag che l'utente ha inserito fra le sue preferenze di livello 1, questo otterrà un punteggio negativo "importante". Se un piatto ha tutti i tag che l'utente ha inserito fra le preferenze di livello 1 invece mantiene invariato il suo punteggio.
- Se ad un piatto manca un tag che l'utente ha inserito fra le sue preferenze di livello 2, questo otterrà un punteggio negativo "soft". Se un piatto ha tutti i tag che l'utente ha inserito fra le preferenze di livello 2 invece mantiene invariato il suo punteggio.
- Se un piatto possiede un tag che l'utente ha inserito fra le sue preferenze di livello

3, questo otterrà un punteggio positivo “minimo”. Se ad un piatto mancano dei tag che l’utente ha inserito fra le preferenze di livello 3 invece mantiene invariato il suo punteggio.

In questo modo, un piatto che soddisfa tutte le preferenze di livello 1 e di livello 2 dell’utente avrà un punteggio finale pari a 0, e sarà classificato come “match”, mentre se un piatto non soddisfa almeno una preferenza verrà classificato come “avoid”, se manca almeno una preferenza di livello 1, o come “bad match”, se manca almeno una preferenza di livello 2 ma soddisfa tutte quelle di livello 1. Se un piatto soddisfa invece tutte le preferenze di livello 1 e 2, ed almeno una di livello 3 viene classificato come “super match”.
In questo modo si ottengono 4 categorie, e l’utente ha modo di vedere subito quali sono i piatti più o meno adatti a lui.

La visualizzazione di un menù parte dalla scansione del codice QR presente sulla sua versione cartacea (o comunque all’interno del locale). Si accede al barcode scanner direttamente dalla home, ed una volta rilevato un codice presente nel database remoto, l’applicazione manda una richiesta HTTP al server, il quale recupera dal database tutti i piatti associati a quel menù.
Una funzione sul DMBS poi calcola gli score per ogni piatto e li invia al provider, il quale si occupa di raggruppare tutte le informazioni su ogni piatto (nome, score, descrizione, ecc..) in un array di oggetti JSon che viene inviato al client come risposta. Il client infine effettua il parsing degli oggetti JSon e presenta i dati nella view della MenuPage. Il click su uno degli elementi del menù porterà all’apertura di una pagina che riporta i dettagli relativi a quello specifico piatto.

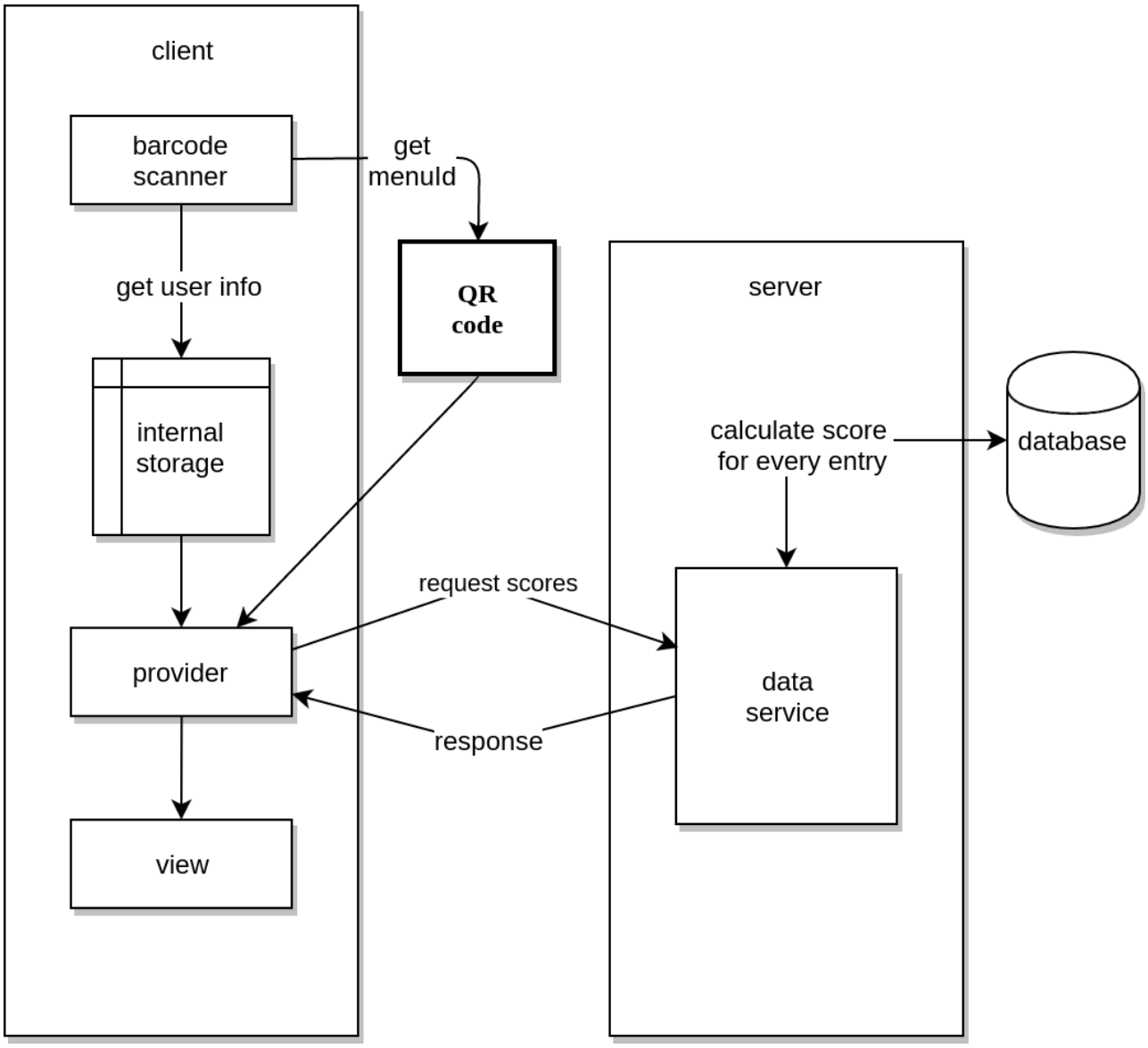

Figura 7: gestione della scansione e degli score

Di seguito si riportano i mockup realizzati per la pagina di visualizzazione del menù personalizzato in base alle preferenze dell'utente e per la pagina di visualizzazione dei dettagli relativi ad un elemento del menù.
Nella pagina del menù le voci sono divise nelle quattro categorie elencate sopra, e visualizzate su due tab per evidenziare meglio quali siano le scelte migliori e peggiori per l'utente.

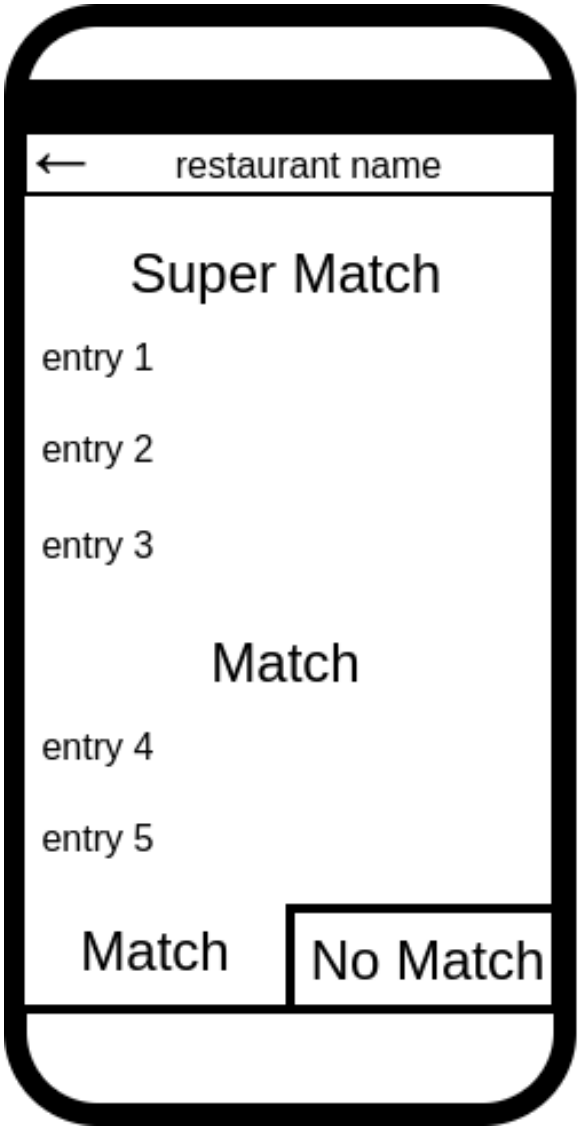

Figura 8: mockup della MenuPage

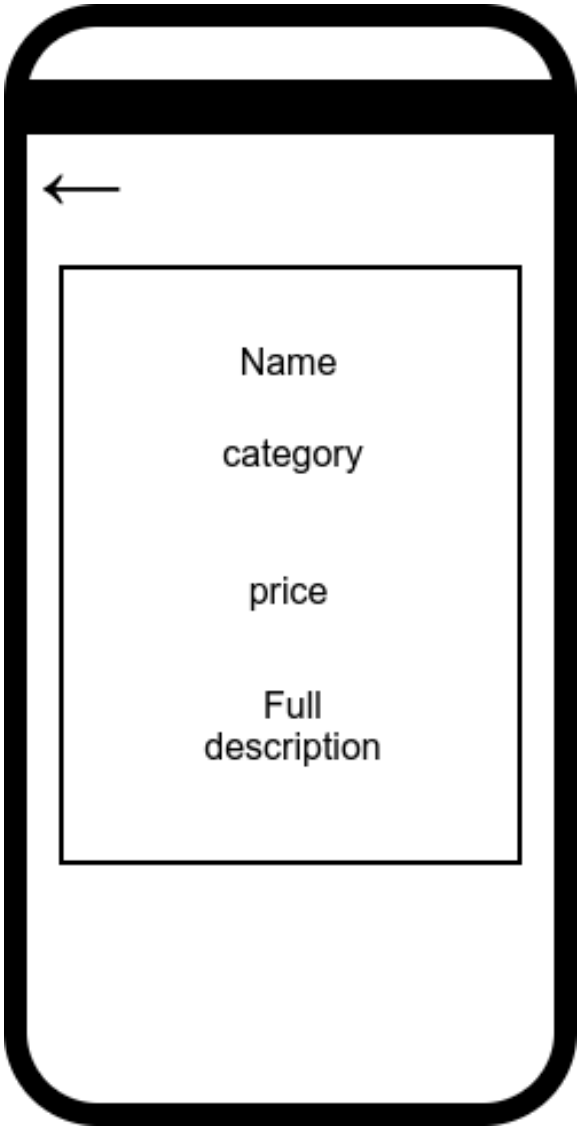

Figura 9: mockup della DetailsPage

2.3.5 – Schema E/R

A questo punto della progettazione è necessario creare uno schema E/R in grado di modellare le entità individuate nelle fasi precedenti e le relazioni tra queste. Da questo schema poi verrà ricavato il modello utilizzato per la creazione delle tabelle da utilizzare nel database del server.
Le entità coinvolte sono: USER, TAG, MENU, ENTRY:

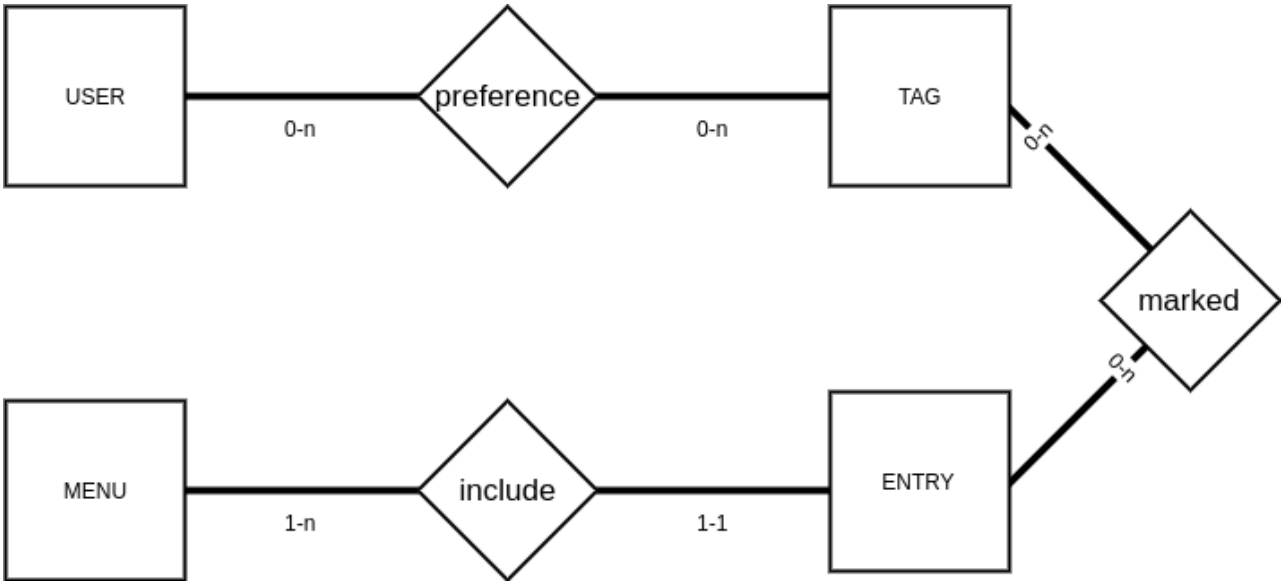

Figura 10: diagramma E/R per il database

L'entità USER può essere associata a 0-n tag (un utente può non selezionare alcuna preferenza, ma può anche selezionarle tutte).

L'entità TAG può essere associata a 0-n utenti (è possibile che qualche tag non venga inserito tra le preferenze di alcun utente, ma anche che sia scelto da tutti gli utenti) ed a 0-n entry (è possibile che un piatto non venga marcato con alcun tag o che venga marcato con tutti i tag).

L'entità MENU può essere associata ad 1-n entry (ogni menù deve avere almeno una voce).

L'entità ENTRY è associata ad un unico menù, e può essere associata ad un qualunque

numero di tag.

Il diagramma E/R non contiene gerarchie, e non ha necessità di essere ristrutturato; è comunque necessario effettuare la traduzione verso il modello logico relazionale, in modo da poter essere implementato in un DBMS. Verranno di seguito discusse tutte le associazioni presenti nel diagramma E/R, che saranno tradotte verso il modello logico relazionale, identificando per ciascuna la chiave primaria (identificata da sottolineatura) e le eventuali chiavi importate (identificata tramite associazioni di colori).

- USER – TAG:
  Associazione n-n tra l'entità USER e l'entità TAG. L'associazione PREFERENCE viene tradotta come relazione, ed ottiene una FK per ciascuna identità a cui è legata:

  USER(id_user, firstName, lastName, email, password_hash, auth_key);
  TAG(id_tag, name, description);
  PREFERENCE(id_preference, user, tag, level);

- TAG – ENTRY:
  Associazione n-n tra l'entità TAG e l'entità ENTRY. L'associazione TAG_ENTRY viene tradotta come relazione, ed ottiene una FK per ciascuna identità a cui è legata:

  TAG(id_tag, name, description);
  ENTRY(id_entry, name, category, price, description);
  ENTRY_TAG(id_entryTag, entry, tag);

- ENTRY – MENU:
  Associazione 1-n tra l'entità ENTRY e l'entità MENU. L'associazione viene tradotta inseme all'entità che partecipa con cardinalità massima 1 (ENTRY), che ottiene una foreign key:

MENU(id_menu, enty, name);
ENTRY(id_entry, name, category, price, description, menu);

Di seguito si riporta lo schema relazionale completo ottenuto dalla traduzione:

- USER(id:user, firstName, lastName, email, password_hash, auth_key);
- TAG(id_tag, name, description);
- PREFERENCE(id_preference, preference, user, tag, level);
  FK: tag REFERENCES TAG
- MENU(id_menu, name);
- ENTRY(id_entry, name, category, price, description, menu);
  FK: menu REFERENCES MENU
- ENTRY_TAG(id_entryTag, entry, tag);
  FK: entry REFERENCES ENTRY
  FK: tag REFERECES TAG

2.4 – Servizi web

Come visto in precedenza, il pattern MVC prevede che ogni richiesta inviata dal client al server sia gestita dal controller di competenza di quest'ultimo.
L'applicazione dunque ha bisogno di funzioni sul server (le cosiddette action dei vari controller) che raccolgano ciascuna una specifica richiesta e la elaborino come richiesto, andando quando necessario ad agire sulla base di dati tramite il model.
Verranno quindi ora elencate schematicamente le action ritenute indispensabili per gestire le richieste provenienti dall'applicazione client:

Controller	Action	HTTP Method	Parametri	Output	Descrizione
User	/signup	POST	BODY: email, password, firstName, lastName	token	Registra un nuovo utente. Se la registrazione va a buon fine, il server invia il token di autenticazione.
User	/login	POST	BODY: email, password	token	Effettua il login di un utente registrato. Se va a buon fine, il server invia il token di autenticazione.
Entry	/list	GET	URL: userId, menuId, HEADER: token	scoreArray	Ottiene un array dei piatti (completi degli score di compatibilità per l'utente) del menù passato in input
Preference	/create	POST	HEADER: token, BODY: userId, tagId, level	-	Inserisce una nuova preferenza per l'utente, con il livello inserito
Preference	/delete	DELETE	URL: tagId, HEADER: token	-	Elimina un tag dalle preferenze dell'utente
Preference	/find	GET	URL: userId, HEADER: token	prefArray	Ottiene un array di tutti i tag selezionati dall'utente fra le sue preferenze, con relativo livello
Preference	/available	GET	URL: userId, HEADER: token	availArray	Ottiene un array di tutti i tag non selezionati dall'utente fra le sue preferenze

Le action vengono richiamate indirizzando le richieste ad un url costruito secondo il pattern **server/provider/action/parameters**. Ogni action richiede un suo specifico metodo HTTP per essere invocata.

Per esempio, per effettuare il login, l'url a cui inviare la richiesta HTTP (con metodo POST) sarà: **server/user/login**, e sarà necessario allegare al body della richiesta i dati necessari al login (email e password).
Per ottenere invece la lista di tutte le preferenze dell'utente l'url a cui inviare la richiesta (questa volta con metodo GET) sarà: **server/preference/find?user='userId'.**
In questo caso i parametri sono inviati direttamente nell'url della richiesta (in quanto effettuata con metodo GET), ma è necessario inviare nell'header della richiesta il token di autenticazione, come spiegato nel paragrafo 2.3.2 .

2.5 – Diagramma delle attività

Un riassunto schematico di quanto studiato nella fase di progettazione viene riportato di seguito sotto forma di diagramma delle attività:

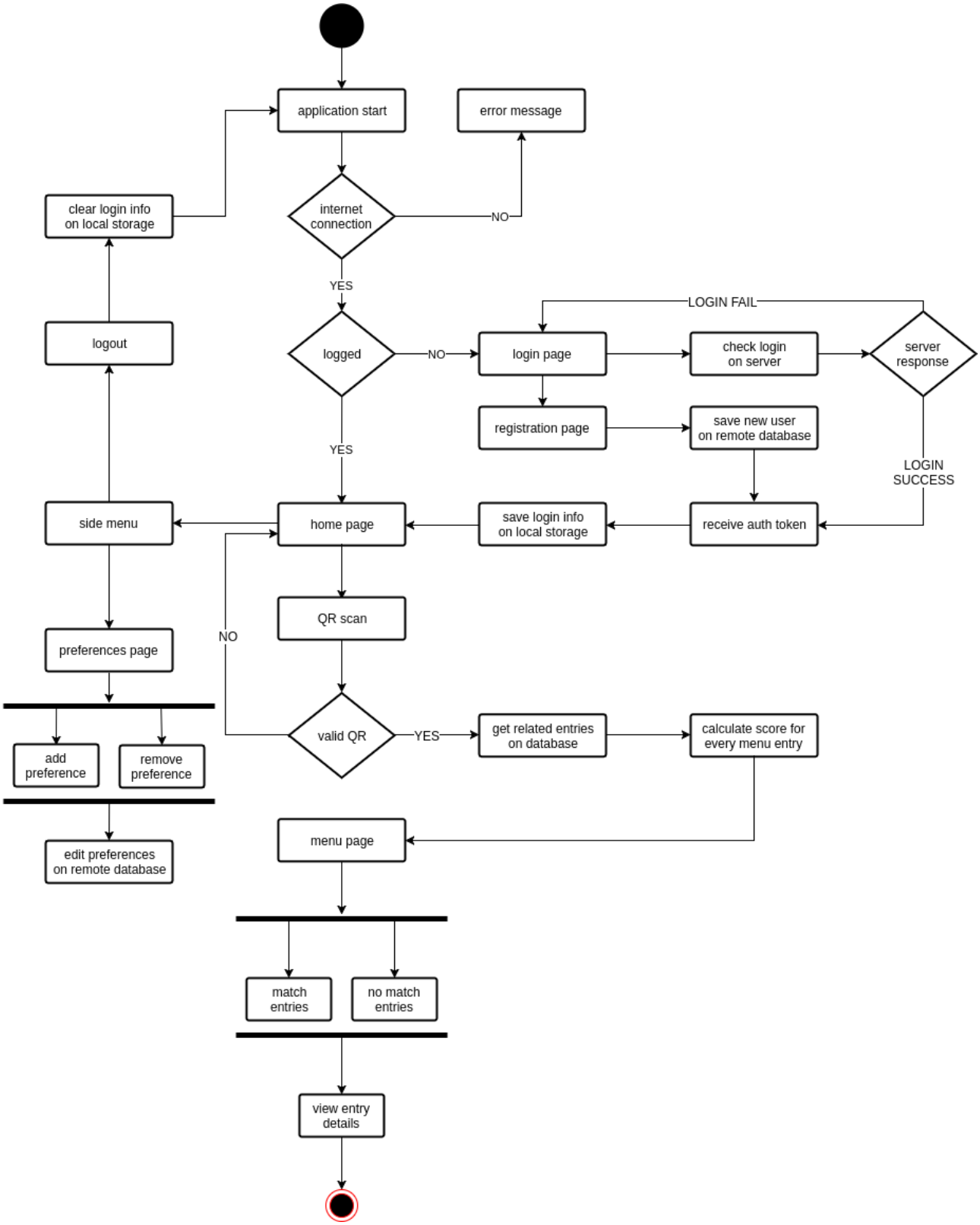

Figura 11: diagramma delle attività

2.6 – Diagramma di navigazione

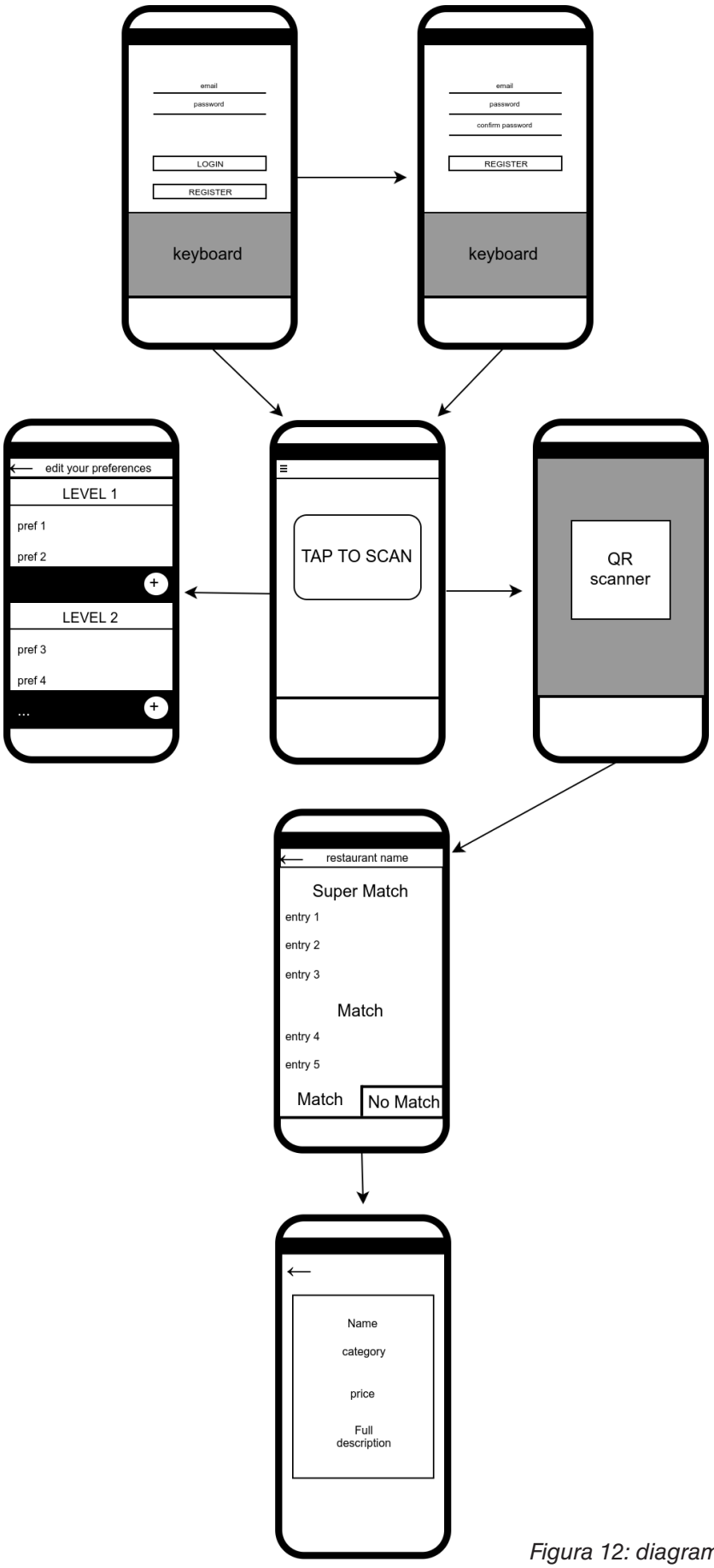

Figura 12: diagramma di navigazione

3 – Implementazione

Dopo aver concluso le fasi di analisi e progettazione del software si può iniziare ad effettuare l'implementazione, sia della parte server (database e framework MVC) sia della parte client (cioè l'applicazione vera e propria).

3.1 - Introduzione alle applicazioni ibride e ad Ionic 2

Il mercato delle applicazioni mobile è di fatto suddiviso tra due piattaforme principali: iOS (il sistema operativo degli iPhone e iPad di Apple) ed Android (sviluppato da Google). Gli altri sistemi operativi mobile hanno una diffusione minima, e sebbene alcuni utilizzino un ecosistema di applicazioni proprietario (il principale dei quali è Windows Phone), altri supportano in modo più o meno completo le app Android, aumentando ulteriormente la diffusione di questa piattaforma.
Di seguito si mostra la diffusione dei principali sistemi operativi per smartphone dal 2013 ad oggi.

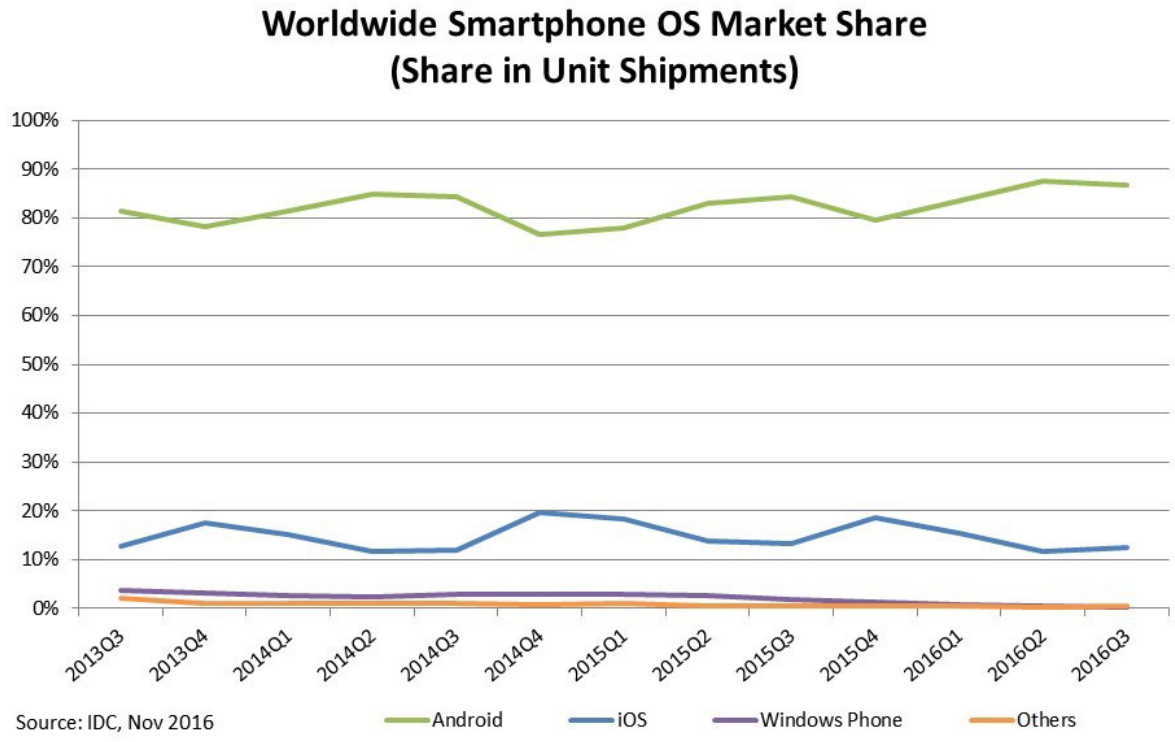

Figura 13: market share dei principali sistemi operativi per dispositivi mobile

Come si evince chiaramente dal grafico, Android è di gran lunga il sistema più diffuso al mondo in ambito mobile, con quasi il 90% di market share.
C'è però un secondo dato da considerare, cioè la redditività delle piattaforme: è un dato ormai consolidato infatti che gli utenti iOS siano mediamente più predisposti a spendere in applicazioni, e di conseguenza, pur essendo in numero molto inferiore, generano un guadagno molto maggiore per il market delle applicazioni iOS (cioè l'App Store di Apple) rispetto a quello del market concorrente (Google Play).

Di seguito si mostrano i ricavi netti dei due store negli ultimi due anni.

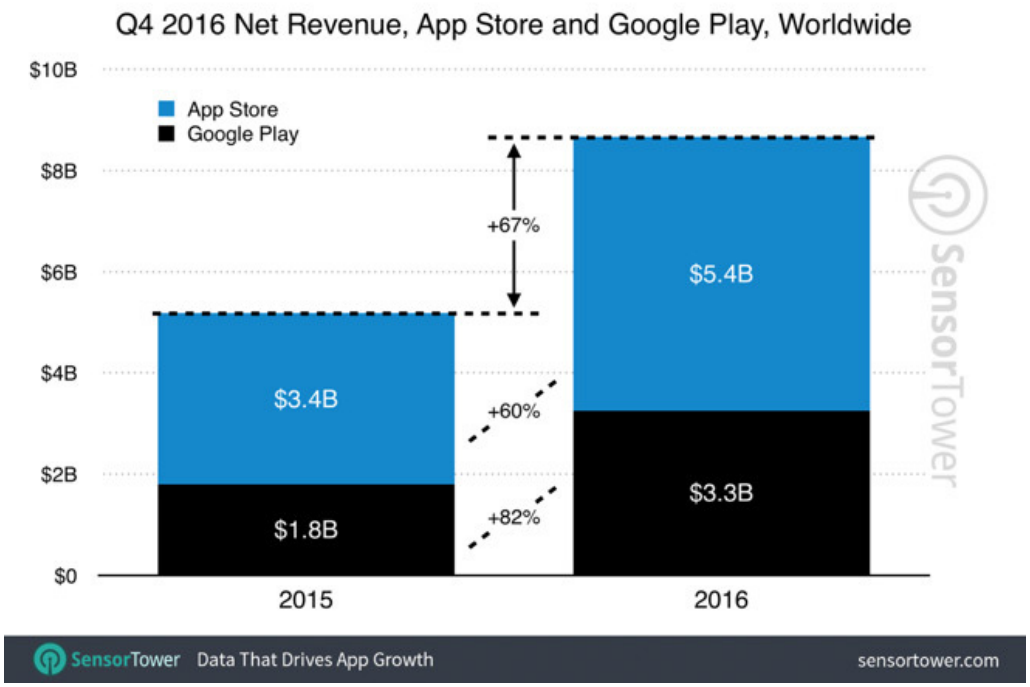

Figura 14: ricavi netti dei due principali market di applicazioni mobile

Se da un lato quindi sviluppare un'applicazione Android offre potenzialmente un pubblico più vasto (sempre che si riesca a raggungerlo questo pubblico, data l'enorme quantità di applicazioni presenti su Google Play), lo sviluppo per la piattaforma di Apple può risultare più remunerativo.
Un altro fattore da considerare è l'ambiente di sviluppo a disposizione: un'app Android può essere scritta utilizzando un pc Windows o GNU/Linux, mentre per lo sviluppo di un'app iOS è necessario utilizzare un Mac. Questo non è certo un limite per i professionisti, ma può essre un ostacolo per chi si avvicina a questo mondo per la prima volta.

3.1.1 - Applicazioni native e applicazioni ibride

Si capisce quindi quanto sia importante la scelta della piattaforma su cui sviluppare la propria applicazione. Per ovviare a questo problema sono nate e si sono diffuse le cosiddette *applicazioni ibride*, cioè sviluppate non nel linguaggio nativo della piattaforma di destinazione, ma con strumenti e linguaggi tipici del web. Questo approccio ha il vantaggio di rendere gran parte del codice scritto indipendente dalla piattaforma su cui verrà installato, in quanto viene eseguito all'interno di un ambiente web. Oltre a questo, lo sviluppo in ambiente web permette di velocizzare di molto il processo di scrittura del codice, riducendo in maniera significativa i tempi ed i costi di sviluppo[6]. Per questo le applicazioni ibride sono di fatto l'anello di congiunzione fra applicazioni native e web app, cercando di prendere il meglio da entrambi i modelli.
Se da una parte però questo approccio risulta spesso vantaggioso dal punto di vista della complessità, del tempo richiesto e delle risorse necessarie allo sviluppo, ci sono anche indubbi svantaggi. Il primo di questi è legato alle funzionalità disponibili. Le app ibride infatti non sempre possono disporre delle stesse funzionalità di quelle native, in quanto alcune funzioni che agiscono ad un livello più basso potrebbero non essere accessibili in un ambiente web. Questo era vero soprattutto per le prime applicazioni ibride, in quanto oggi esistono in alcuni casi sistemi di emulazione di tali funzioni. Il secondo problema che si può riscontrare nello sviluppo di questo tipo di applicazioni è quello dell'efficienza. Venendo eseguite in ambiente web (ma con necessità computazionali che possono essere importanti), ovviamente la richiesta di risorse è maggiore rispetto alla stessa applicazione sviluppata in modo nativo. Questa problematica non si presenta su app semplici, ma può diventare un vero problema al crescere della complessità e della quantità dei dati gestiti. Un esempio lampante in questo senso è quello di Facebook, che dopo aver puntato sul modello ibrido per la sua app, è stato costretto per motivi di performance a svilupparne una versione nativa[7].
In questo progetto ho quindi voluto sperimentare in prima persona pregi e difetti dello sviluppo di un'applicazione ibrida, dopo aver già sviluppato un'app Android in linguaggio nativo nel corso di Mobile Web Design.

3.1.2 - Cenni sul framework Ionic 2

Il framework scelto per lo sviluppo è Ionic, nella versione 2.1.8 . Ionic 2 si basa su AngularJS e Cordova, un framework open source per lo sviluppo di app ibride al cui sviluppo partecipano molte importanti aziende, come Adobe, Google, IBM, Microsoft e Mozilla, per citarne alcuni.
Ionic 2 sfrutta AngularJS 2 per fornire i *components*, cioè i moduli con cui costruire l'interfaccia ed i relativi metodi per farli interagire. Ogni component può poi essere modificato a seconda delle necessità, ed è possibile crearne da zero di nuovi.
Ogni pagina dell'applicazione è costituita principalmente da tre file: page.ts, page.html e page.scss.

- **page.ts** è il file TypeScript che contiene tutto il codice e la logica applicativa della pagina. TypeScript è un super-set di JavaScript basato su ECMAScript6, che una volta compilato produce codice JavaScript. Siccome i browser non supportano ancora in pieno ECMAScript6, Ionic si occupa di effettuare il *transpiling*, cioè la conversione in codice Javascript compatibile.

- **page.html** è il file che contiene la struttura dell'interfaccia della pagina, scritta in HTML utilizzando i components forniti da Ionic. L'utilizzo di questi rende molto più rapida la scrittura del codice, in quanto la maggior parte degli elementi più spesso utilizzati (come liste, buttons, cards, tabs eccetera) sono già resi disponibili dal framework, e la loro modifica è semplice e veloce.

- **page.scss** è il file che contiene il codice per definire lo stile della pagina. Ionic usa SASS (Syntactically Awesome Style Sheets), un preprocessore CSS che ne estende le funzionalità. Una volta che l'app viene compilata, il file .scss viene compilato in un file CSS standard ed ottimizzato.

Ionic 2 supporta le piattaforme Android ed iOS; è dunque possibile compilare il codice dell'applicazione ed avere due versioni della stessa app, ognuna compatibile con la piattaforma di riferimento. Da un punto di vista della compatibiltà, le app generate con Ionic sono equivalenti a quelle native, e all'utente finale non saranno visibili differenze sostanziali. Anche dal punto di vista della commercializzazione le app Ionic sono trattate

in maniera equivalente a quelle native, per cui valgono le stesse regole per poter essere pubblicate sui rispettivi market.
Nel caso specifico di questo progetto, l'applicazione è stata compilata per essere eseguita su Android. Questa scelta è stata dettata dalla familiarità con la piattaforma e dall'ambiente di sviluppo a disposizione (sistema operativo GNU/Linux e smartphone Android per i test).

3.2 – Implementazione lato server

Dopo aver introdotto brevemente il mondo delle applicazioni ibride, illustrerò di seguito il processo di implementazione tramite l'utilizzo degli strumenti descritti.

3.2.1 - Implementazione del database

Per prima cosa si procede con l'implementazione del database, e quindi delle tabelle e delle funzioni SQL.
Il DBMS scelto è MySQL, in quanto ormai è lo standard de facto per questo tipo di necessità, oltre ad essere gratuito, open source e perfettamente integrato con PHP.

Di seguito è inserito il diagramma E/R completo di attributi creato con MySQL Workbench:

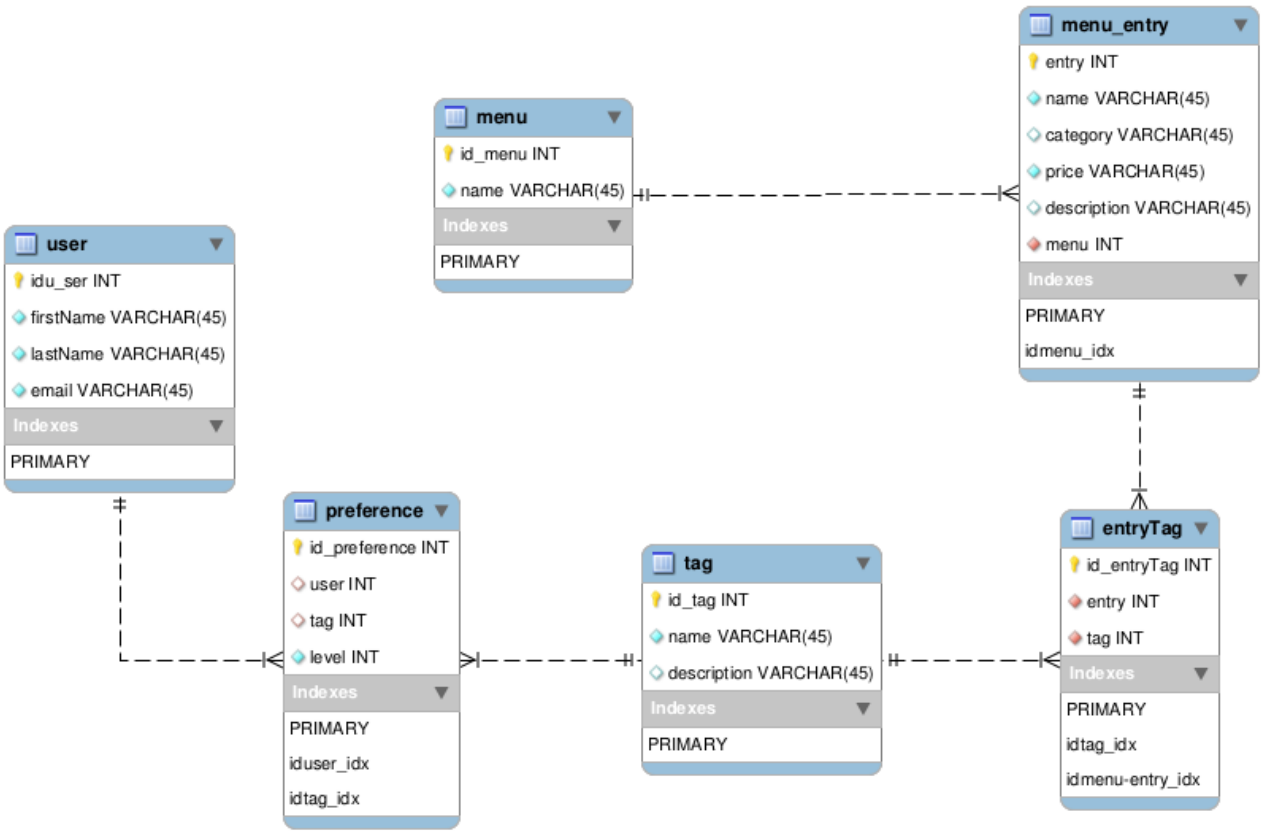

Figura 15: diagramma E/R da MySQL Workbench

Dal diagramma si ricavano poi direttamente le tabelle per la creazione del database, con relative chiavi primarie, chiavi importate e primary index.

Nel DBMS è implementata anche la funzione principale utilizzata per il calcolo degli score dei piatti.

Questa funziona prendendo in input gli id di un user e di una entry e calcolandone lo score utilizzando un algoritmo basato sul conteggio dei tag in comune tra il piatto e le preferenze dell'utente. Il funzionamento base dell'algoritmo è il seguente, illustrato in pseudo-codice:

```
function(id_user, id_entry)
BEGIN
	VAR array = [];

	// count tags in user pref for level 1
	LOOP preference WHERE user == id_user INTO prf {
		IF prf.level == 1 THEN {
			INSERT prf INTO array;
		}
	}

	// count tags both in user pref for level 1 and entry
	VAR match = 0;
		LOOP entryTag WHERE tag == id_entry INTO etg {
			FOREACH item IN array {
				IF item.tag = etg.tag THEN {
					match++;
				}
			}
		}
	}

	// calculate difference between match and count
	IF array.count() > match THEN {
		score = score – (count – match) * highInt;
	}

	// count tags in user pref for level 2
	array = [];
	LOOP preference WHERE user == id_user INTO prf {
		IF prf.level == 2 THEN {
			INSERT prf INTO array;
		}
	}

	// count tags both in user pref for level 2 and entry
	match = 0;
		LOOP entryTag WHERE tag ==  id_entry INTO etg {
			FOREACH item IN array {
				IF item.tag = etg.tag THEN {
					match++;
				}
			}
		}
	}
```

```
        // calculate difference between match and count
        IF array.count() > match THEN {
                score = score – (count – match) * lowInt;
        }

        // count tags in user pref for level 3
        array = [];
        LOOP preference WHERE user == id_user INTO prf {
                IF prf.level == 3 THEN {
                        INSERT prf INTO array;
                }
        }

        // count tags both in user pref for level 3 and entry
        match = 0;
                LOOP entryTag WHERE tag == id_entry INTO etg {
                        FOREACH item IN array {
                                IF item.tag = etg.tag THEN {
                                        match++;
                                }
                        }
                }
        }

        // calculate difference between match and count
        IF match > 0 THEN {
                score = score – (count – match) * highInt;
        }
        RETURN score;
END;
```

3.2.2 - Implementazione dei servizi web

Il passo successivo al settaggio del DBMS e all'implementazione delle tavole e delle funzioni del database è l'implementazione sul server dei servizi web necessari ad interagire sia con il database che con il client.
La scelta è ricaduta su Yii2, uno dei più avanzati framework PHP per lo sviluppo di applicazioni web. Yii2 implementa in maniera completa ed ottima il pattern MVC, ed è rilasciato in licenza open source.
Yii2 implementa il pattern ORM (Object-Relational Mapping) e si occupa di creare in autonomia un Active Record Model (che è un'estensione avanzata del Model tradizionale secondo il pattern ORM) del database ed i relativi controller per le operazioni base su di esso. Vengono quindi generate le REST API per le operazioni CRUD (Create, Read, Update, Delete) sul database attraverso gli Active Records; questi forniscono un'interfaccia object-oriented per l'accesso e la manipolazione dei dati presenti sul database. Ad ogni tabella del database viene associata una classe con i rispettivi metodi di lettura e scrittura, e ad ogni riga della tabella viene associata un'istanza di Active Record, che sarà sempre aggiornata per rispecchiare quella specifica riga. I controller quindi agiscono sugli Active Records per leggere, eliminare o modificare i dati presenti sul database.

Siccome le operazioni base messe a disposizione non sono sufficienti a gestire le richieste dell'applicazione viste nel paragrafo 2.4, è necessario implementare ulteriori action all'interno dei controller, per poter eseguire le operazioni richieste sul database. Sono quindi state scritte in PHP tutte le action descritte nel paragrafo 2.4, con l'eccezione di *preference*/*create* e *preference*/*delete*, che erano già messe a disposizione del framework.
A questo punto quindi il server ha tutte le API necessarie, che possono essere chiamate dall'applicazione client per l'esecuzione delle operazioni sul database attraverso gli Active Records, ed è possibile passare all'implementazione lato utente, cioè all'applicazione vera e propria.

3.3 – Implementazione lato utente

In questa sezione verrà illustrata l'implementazione del software lato client secondo la progettazione effettuata in precedenza.

Come già detto in precedenza, per lo sviluppo è stato usato il framework Ionic2, e andrò quindi in primo luogo ad esporre le componenti da cui è formata l'applicazione.

Innanzitutto è bene sottolineare come lo standard ECMAScript6 (e la sua implementazione Typescript utilizzata da Ionic) rappresenti un grosso passo avanti rispetto alla precedente versione (quella di cui Javascript è l'implementazione più diffusa e conosciuta), principalmente per l'introduzione ed il supporto completo alle classi. Questo permette di creare applicazioni anche piuttosto complesse (a differenza di come era stato inizialmente concepito Javascript) e, soprattutto, modulari.
Le applicazioni Ionic si basano sui *Component*, cioè "blocchi" di codice di alto livello con cui costruire l'interfaccia dell'appliczione. Ionic mette a disposizione una grande varietà di component per le principali esigenze di progettazione: esistono component specifici per la creazione di liste, tab, card, bottoni, icone, input form, loader, alert eccetera; le pagine stesse dell'applicazione sono dei component. Se i component predefiniti non fossero sufficienti a implementare qualche funzionalità, è possible modificare o estenderne uno esistente, o anche crearne uno nuovo. Ad ogni component è associato un layout HTML che ne gestice la struttura.
Un altro concetto fondamentale in Ionic è quello di esportazone: pagine, classi, provider, data models e qualunque altro elemento dell'applicazione può essere reso esportabile, in modo da poter poi essere importato ed utilizzato in qualunque altro punto dell'applicazione.
Ogni applicazione Ionic utilizza le *Pages* per la visualizzazione dell'interfaccia. Ad ogni page corrispondono, come anticipato, tre file: uno per la struttura, uno per la logica applicativa ed uno per lo stile.
Un template di una pagina Ionic 2 risulterà familiare a chi è abituato a vedere pagine web scritte in HTML, anche se presenta alcune componenti inedite.
Il component *ion-list* messo a disposizione da Ionic permette di creare con poche righe di codice una lista con l'utilizzo di *ngFor, che effettua un loop dell'array "items" e crea per ogni elemento un bottone cliccabile, collegato alla funzione "itemSelected(item)".
Un'altra importante componente delle app Ionic (così come delle app native) sono i cosiddetti "services", qui chiamati *providers*. I provider tecnicamente non sono dei component, in quanto non sono associati ad un layout, ma funzionano sostanzialmente in modo molto simile, agendo in background. Ogni provider quindi sarà costituito da un unico file Typescript, e sarà (come i component) esportabile ed utilizzabile da qualunque pagina all'interno dell'applicazione tramite *injection*. Il concetto di injection è anch'esso

molto importante in ambito Ionic: una classe con decoratore @Injectable funziona automaticamente come provider per tutte le classi che la importano tramite injection all'interno del loro costruttore.

3.3.1 – Plugin

Il supporto a numerosi plugin facilita l'implementazione di alcune funzioni che sarebbero altrimenti complesse da scrivere partendo da zero, soprattutto se si vuole mantenere la compatibilità con tutte le versioni di entrambe le piattaforme.
Alcuni plugin vengono installati in automatico alla creazione di un nuovo progetto Ionic, e sono quelli che servono praticamente in tutte le applicazioni, come per esempio quello per la visualizzazione e la gestione della status bar o della tastiera. Altri più specifici invece possono essere aggiunti durante lo sviluppo, e possono essere scelti principalmente fra quelli disponibili per Cordova e PhoneGap e tra i cosiddetti "Ionic Native", che forniscono delle versioni dei plugin sopracitati con l'aggiunta di classi wrappers che aggiungono funzionalità tipiche delle applicazioni native; gli sviluppatori di Ionic stanno lavorando per portare tutti i plugin Cordova e PhoneGap in versione Ionic Native, e renderli così più funzionali e compatibili.

Questo progetto nello specifico fa uso di tre plugin esterni:

1. **Whitelist**: un plugin Cordova che permette all'applicazione di comunicare con indirizzi web esterni, ed è quindi necessario per la comunicazione col server.
2. **BarcodeScanner**: un plugin Cordova che implementa il riconoscimento e la lettura di diversi tipi di codici a barre e codici QR. Questo plugin fa parte degli *Ionic Native* a cui si faceva riferimento poco sopra.
3. **Network**: un plugin Cordova facente parte degli *Ionic Native* che permette di gestire le informazioni riguardo la presenza o meno di una connessione da parte del dispositivo ed il suo tipo (wifi, 2g, 3g o 4g).

3.3.2 – Provider e funzioni asincrone

Come anticipato, il provider è la parte che si occupa di gestire le funzioni comuni a più

pagine. Il provider si occupa anche di gestire tutte le richieste HTTP al server e le relative risposte: se una pagina deve prendere un input dall'utente per poi utilizzarlo in una richiesta al server, la pagina si occuperà di raccogliere i dati e poi invocherà il provider per eseguire la chiamata. Queste avvengono sempre in maniera asincrona, tramite l'uso di *Promise* e *Observable*. Una funzione asincrona deve solitamente recuperare dei dati (per esempio da un server web) senza sapere quanto tempo questo richiederà, e quindi invece che aspettare di avere tutti i dati necessari prima di ritornarli, ritorna direttamente una promise, cioè una sorta di "garanzia" che metterà a disposizione i dati quando li avrà ricevuti. In questo modo l'applicazione può proseguire con altre attività, per poi tornare sui dati una volta disponibili ed utilizzarli.
Le funzioni asincrone quindi prevedono sempre una parte di codice da eseguire una volta ricevuti i dati necessari, che viene eseguita solo quando viene risolta la promise corrispondente.

Gli observables sono molto simili come concetto, ma invece che gestire una singola risposta dal server, gestiscono un flusso di risposte. Il funzionamento specifico degli observables non è qui descritto in quanto non sono stati utilizzati nel progetto.
Il concetto di servizio asincrono non è legato solo alle richieste HTTP, ma anche a servizi interni all'applicazione, come per esempio verrà descritto nel paragrafo riguardante il login e lo storage locale.
Le funzioni implementate nel provider per effettuare richieste al server sono le seguenti:

Funzione	**Parametri in input**	**Descrizione**
login	email: string, password: string	Prende in input i dati passati dalla funzione della LoginPage chiamata dal relativo bottone ed effettua la richiesta al servizio *user/login*. Se questa va a buon fine, inserisce nel local storage le informazioni sull'utente, altrimenti mostra un toast di errore.
register	firstName: string, lastName: string, email: string, password: string	Prende in input i dati passati dalla funzione della RegisterPage chiamata dal relativo bottone ed effettua la richiesta al servizio *user/signup*. Se questa va a buon fine, inserisce nel local storage le informazioni utente, altrimenti mostra un toast di errore con il problema riscontrato dal server.

getAvailPref	user: number, token: string	Effettua la richiesta al servizio *preference/available* per ottenere la lista dei tag disponibili per la scelta.
getUserPref	user: number, token: string	Effettua la richiesta al servizio *preference/list* per ottenere i tag inseriti dall'utente nelle sue preferenze.
addPref	user: number token: string array: any level: number	Per ogni elemento dell'array delle preferenze selezionate dall'utente effettua la richiesta al servizio *preference/create* per inserire il tag tra le preferenze dell'utente (con il livello selezionato).
removePref	user: number, token: string, level: number, index: number	Effettua una richesta al servizio *preference/delete* per eliminare un tag dalle preferenze dell'utente. Il tag è quello identificato dalla posizione index nell'array collegato al livello level passato in input.
getMenuEntries	menu: number, user: number, token: string	Riceve l'id del menù ottenuto dalla scansione e l'id dell'utente dal local storage ed effettua la richiesta al servizio *entry/list* per ottenere la lista dei piatti collegati al menù e relativi score. Crea poi un array per ogni categoria di punteggio ed altrettante liste per la visualizzazione sul dispositivo, assegnandovi i piatti.

Ci sono poi altre funzioni implementate nel provider che però non effettuano richieste al server. La maggior parte di queste effettuano operazioni sugli array ricevuti e sulle liste per la visualizzazione. Questo è necessario perché le direttive Ionic *ngFor che permettono di costruire dinamicamente liste nella view sono compatibili solo con array semplici, mentre i dati contenenti le informazioni sui piatti ricevuti dal server sono in forma di array di oggetti JSon.

Un'altra funzione del provider utilizzata da tutte le pagine dell'applicazione è *checkNetwork*, che controlla se il dispositivo è connesso alla rete internet facendo uso dell'apposito plugin. Questa funzione è sempre chiamata prima di effettuare una richiesta al server: se ritorna un valore true, si prosegue effettuando la chiamata, altrimenti si interrompe e si mostra un apposito toast di errore.

Ci sono poi tutta una serie di funzioni necessarie all'interazione con l'utente, o volte al miglioramento dell'esperienza d'uso, come per esempio la funzione per il logout (che

effettua una cancellazione delle informazioni dell'utente dallo storage locale), e tutte quelle legate alla gestione dei context menu per le conferme delle azioni, dei toast di errore e così via, oltre a quelle per mostrare ed interrompere le animazioni durante i momenti di attesa.

3.3.3 – Gestione del login e delle informazioni utente

In questa sezione verrà spiegato come è gestita la parte riguardante l'utente, cioè come avvengono il login e la registrazione, e dove sono salvati i dati in locale.
Come mostrato in precedenza nel diagramma delle attività, l'applicazione prevede un controllo sullo stato dell'utente all'apertura (loggato/non loggato). La funzione che si occupa di questo controllo è inserita nel constructor della classe *MyApp*, cioè la classe che viene istanziata all'apertura dell'applicazione. Qui infatti è possibile settare una qualunque pagina dell'applicazione come pagina iniziale, e per effettuare il controllo sullo stato dell'utente occorre quindi settare inizialmente rootPage: any; ed in seguito effettuare un controllo sulla variabile *logged* inserita nel local storage. Se la variabile esiste, significa che l'utente si era già loggato in precedenza, e si effettua quindi una chiamata alla funzione *get* dello storage per recuperare le informazioni da visualizzare (nome, cognome ed email); infine si rimanda l'utente alla home page dell'applicazione, con this.rootPage = HomePage; . Se invece la variabile *logged* non esiste nello storage locale, significa che non c'è nessun utente loggato, e si rimanda quindi alla pagina di login con this.rootPage = LoginPage; .
La classe *MyApp*, in quanto component a tutti gli effetti, è collegata ad un template, in questo caso *app.html*. Nelle applicazioni Ionic solitamente questo è il template usato per il side menu, e questa applicazione non fa eccezione. All'interno della classe *MyApp* sono quindi inseriti anche i metodi chiamati dai bottoni presenti nel side menu: *tryLogout()* ed *editPref()*, che servono rispettivamente ad invocare la funzione di logout del provider e ad aprire la pagina di modifica delle preferenze.
La navigazione tra le pagine in Ionic avviene utilizzando il *NavigationController* messo a disposizione dal framework, e le pagine sono gestite utilizzando uno stack. Effettuando quindi un push di una pagina, questa viene posta in cima allo stack ed è visualizzabile. Effettuando invece un pop, questa viene rimossa dallo stack e si visualizza la pagina immediatamente sotto. La terza opzione possibile è assegnare ad una nuova pagina il ruolo di rootPage, creando un nuovo stack con quella pagina come base (non è quindi

possibile effettuare il pop di una rootPage), e nessuna pagina sopra di essa. Questo è il caso visto in precedenza, quando a seconda della variabile *logged* si seleziona una pagina piuttosto che l'altra come rootPage.

3.3.3.1 – LoginPage

Il template della pagina di login rispecchia quanto progettato nel mockup: due campi di input per email e password, e due bottoni, uno per il login e l'altro per andare alla pagina di registrazione.

Il bottone di login chiama la funzione *tryLogin()*, che a sua volta chiama la funzione del provider che si occupa di effettuare la richiesta al server per autenticare l'utente.
Come si può notare, la funzione chiamata dal bottone di login è priva di parametri. Questo perché Ionic, utilizzando AngularJS, permette tramite la direttiva [(NgModel)] il binding tra un campo di input ed una variabile nello scope. In questo caso quindi sono state dichiarate due variabili *email* e *password* nella classe *LoginPage*, e nel layout sono state collegate

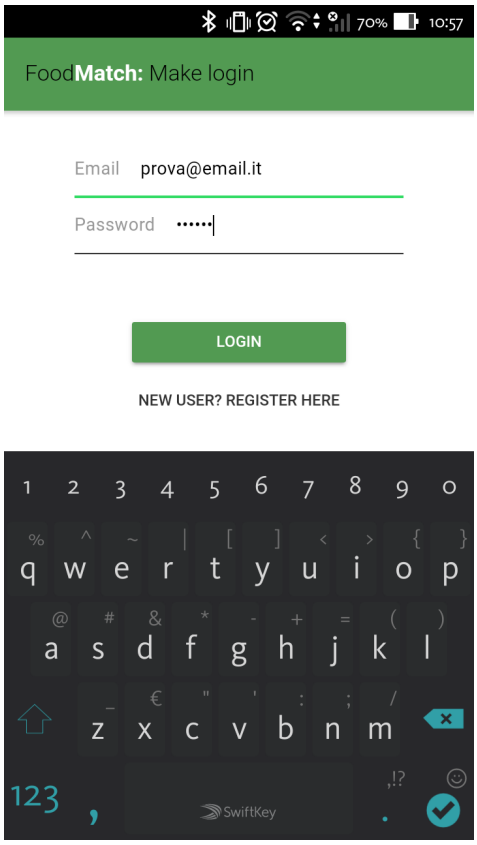

Figura 16: Screenshot della LoginPage

ai due campi di input nel modo descritto sopra.
La funzione *TryLogin* riceve la risposta dalla funzione di login del provider, e setta la rootPage a seconda del risultato, nello stesso modo visto in precedenza: se il login va a buon fine la HomePage diventa la nuova rootPage, altrimenti si rimane sulla LoginPage e si mostra un toast comunicando l'errore nel login. Mentre la funzione sul provider effettua la chiamata al server ed aspetta la risposta, viene visualizzato un loader, per far capire all'utente che l'applicazione sta aspettando dei dati dal server e non si è semplicemente bloccata.
Un'altra cosa da sottolineare riguardante il login è il recupero delle informazioni utente da visualizzare nel side menu. Nel caso l'utente sia già loggato all'apertura dell'applicazione, basta recuperare le informazioni dallo storage locale e assegnarle alle relative variabili da visualizzare nel menu; se però l'utente non era già loggato, il recupero avviene in maniera differente, attraverso l'uso degli *events*. Infatti nel constructor della classe *MyApp*, che si occupa del side menu e quindi della visualizzazione delle informazioni utente all'interno di esso, è stato inserito un *subscriber* ad un evento specifico (chiamato in questo caso 'user:signedIn'), che accetta due parametri *UserEventData* ed *UserEventPref*. La funzione di login del provider, una volta ricevuta risposta positiva dal server alla richiesta di autenticazione, procede al *publish* dell'evento 'user:signedIn', passando i parametri richiesti, cioè un array contenente nome, cognome ed email dell'utente (*UserEventData*) ed un array contenente le preferenze dell'utente (*UserEventPref*).
Questo sistema permette di visualizzare sempre nel side menu le informazioni di un utente che ha appena effettuato il login.

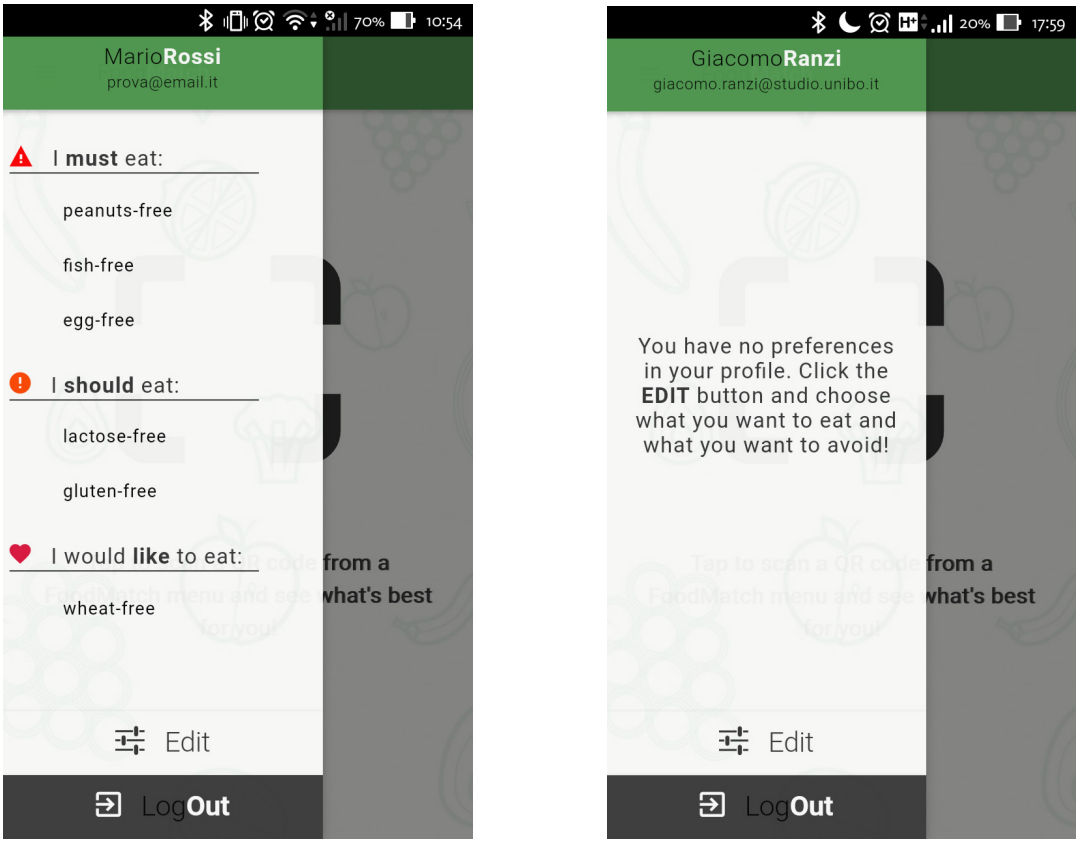

Figura 17: Screenshot del side menu della HomePage

3.3.3.2 – RegisterPage

La RegisterPage, accessibile dalla LoginPage, prende in input i dati di un nuovo utente e lo registra sul database. I campi richiesti per la registrazione sono: First name, Last name, email, password. Il bottone per inviare la registrazione invoca la funzione *tryRegister* che a sua volta chiama la funzione del provider per la chiamata al server. I campi in input sono inviati nel body della richiesta (di tipo POST), ed il server si occupa della validazione dei valori ricevuti: i controlli sono su First name e Last name (che non devono essere vuoti), email (che deve essere una stringa in formato email e non deve essere già presente nel database) e infine password (che deve essere una stringa di almeno 6 caratteri). Se un qualunque valore non viene convalidato dal server, questo manda un messaggio di errore nella response, che poi viene visualizzato in un toast. Se invece la registrazione va a buon fine, vengono salvati i dati utente appena immessi nello storage locale, come avviene quando si effettua un login, insieme con il token di autenticazione e il nuovo id utente, ricevuti nella risposta dal server. Come avviene per il login, anche in questo caso si procede al publish dell'event "user:signedIn", e si inviano i dati utente per la visualizzazione del profilo nel side menu.

Quando un nuovo utente si registra con successo, ed ha quindi accesso per la prima volta all'applicazione, viene reindirizzato direttamente alla pagina di scelta della preferenze, e viene mostrato un breve messaggio che spiega come iniziare ad utilizzare l'applicazione.

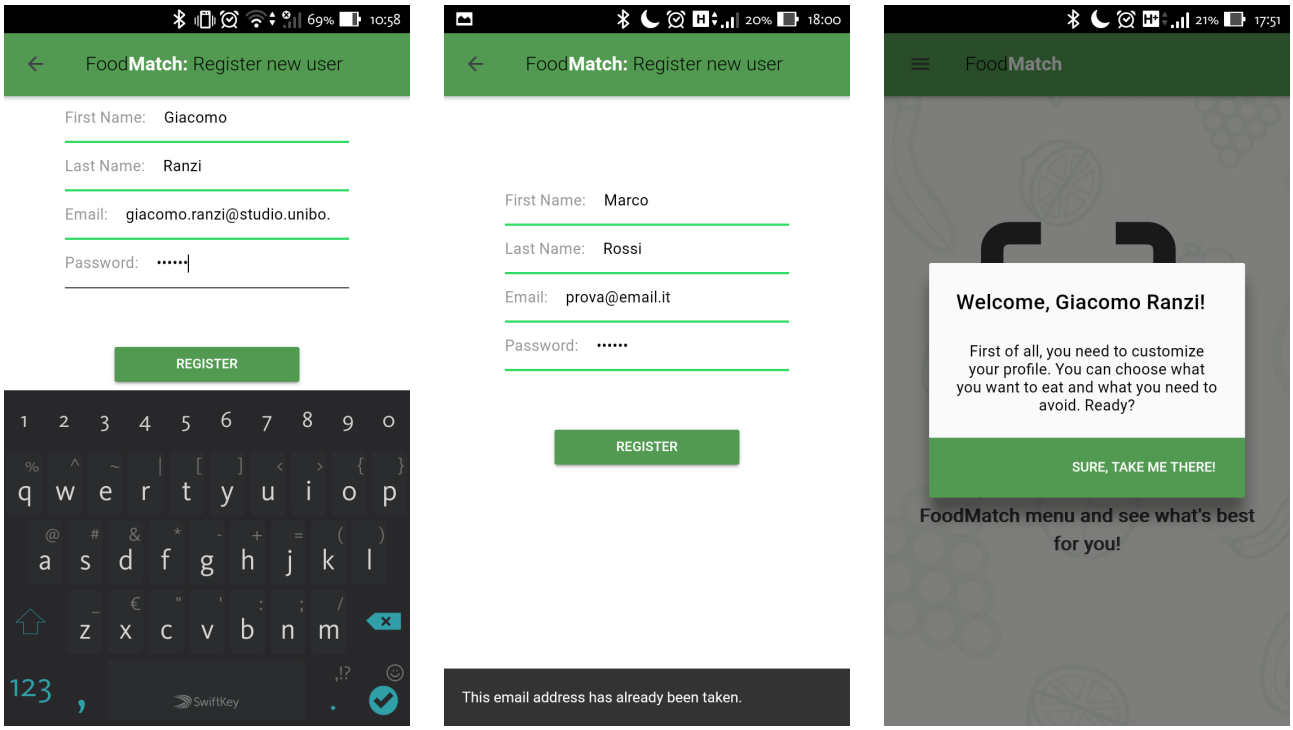

Figura 18: screenshot della RegisterPage

3.3.3.3 – StorageProvider

In questa sezione si descriverà l'implementazione e l'utilizzo dello storage locale per la persistenza dei dati di un utente che effettua il login.
Innanzitutto è bene precisare che finora si è usato il termine Local Storage (o Storage Locale) per essere più chiari nello specificare che si trattava di dati salvati in locale sul dispositivo, e non in remoto sul database.
Il tipo di storage utilizzato da Ionic tuttavia può essere di diversi tipi: SQlite, IndexedDB, WebSQL e LocalStorage, in quest'ordine di priorità. Se non si imposta uno dei tipi di storage come preferenziale, il provider che si occupa dello storage sceglierà in automatico il migliore. Il motivo per cui il LocalStorage, cioè il sistema di storage utilizzato dai browser per salvare dati in locale, è ultimo in ordine di priorità è dovuto sia al fatto che è abbastanza limitato nella capienza (è limitato a 5MB di dati), sia a problemi legati al

sistema operativo del telefono: per esempio iOS elimina i dati dal LocalStorage quando ha bisogno di memoria.
Il fatto che Ionic gestisca in autonomia il salvataggio in locale dei dati eliminando il problema dell'utilizzo esclusivo del LocalStorage permette l'utilizzo di questa funzionalità più ad alto livello e con maggiore sicurezza sulla persistenza dei dati.
Per utilizzare il provider dello storage di Ionic è sufficiente effettuare un import dello stesso tramite la direttiva apposita ed effettuare poi una injection all'interno del constructor della classe nel quale si vuole utilizzare.
Lo storage permette il salvataggio di informazioni come coppie di key/values e mette a disposizione varie funzioni. Quelle di cui l'applicazione fa uso sono:

- Storage.set('key', value); per inserire una coppia chiave/valore.
- Storage.get(name).then((value) => { }); per recuperare il valore associato ad una chiave. La funzione get ritorna una promise, ed è cioè una funzione che agisce in maniera asincrona.
- Storage.clear(); semplicemente elimina tutti i dati salvati (in questo caso viene utilizzata solo per il logout).

Quando un utente effettua con successo il login, le informazioni restituite dal server vengono salvate nello storage come segue:

- chiave: *logged* – valore: *true*
- chiave: *UserInfo* – valore: *object*
    - *id*
    - *email*
    - *firstName*
    - *lastName*
    - *token*

3.3.4 - Gestione delle preferenze utente

In questa sezione verrà spiegato come è stata implementata la gestione delle preferenze utente.
Le preferenze vengono recuperate dalla funzione *getUserPref* tramite il provider, chiamata al momento del login o, nel caso di utente già loggato, all'apertura dell'applicazione.

Le preferenze vengono ricevute sotto forma di array di oggetti, che viene poi ciclato, ed ogni preferenza (o meglio, il suo nome) viene assegnata tramite una seconda funzione chiamata *pushToList* ad una delle tre liste a seconda del suo livello, per la visualizzazione sul dispositivo. Come già anticipato, questo è necessario per poter visualizzare una lista in maniera dinamica, in quanto la direttiva *ngFor può costruire liste solo a partire da array semplici, e non di oggetti; per questo i nomi delle preferenze vengono salvati come stringhe in tre array.

3.3.4.1 – PreferencePage

La pagina PreferencePage contiene un riepilogo dei tag selezionati dall'utente, divisi per livello. All'apertura della pagina viene invocata la funzione *getAvailPref* del provider, che recupera un array di oggetti contenente tutti tag che non rientrano tra quelli selezionati dall'utente.
L'utente ha la possibilità di aggiungere e rimuovere tag ai singoli livelli di preferenza tramite due apposite funzioni. La prima (*removePref*), invocabile tramite il provider con un long press sull'elemento scelto, elimina la preferenza tramite la chiamata al servizio *preference*/*delete* del server. In seguito poi vengono aggiornate le preferenze utente e le relative liste, in modo da aggiornare immediatamente la view. La seconda invece (*addPref*) viene invocata sempre tramite il provider con un click sugli appositi bottoni, ed apre un *checkbox-alert* (uno dei component di Ionic) dove l'utente può selezionare tutti i tag non ancora scelti per aggiungerli alle proprie preferenze. Questa aggiunge tutti i tag selezionati tramite checkbox alle preferenze dell'utente, tramite chiamate al servizio *preference*/*create* (una per ciascuna preferenza aggiunta). Anche in questo caso vengono aggiornate le liste delle preferenze per l'aggiornamento della view, ed in aggiunta viene richiamata la funzione per ottenere la lista dei tag non scelti dall'utente.

Il template della PreferencePage prevede quindi, come da mockup, tre liste, generate in maniera dinamica utilizzando il component ion-list dagli array delle preferenze, sempre grazie alla direttiva *ngFor, ed un bottone per ognuna che apre il checkbox alert per l'inserimento di nuovi tag per quel livello di preferenza.
A lato dell'header di ogni lista è poi posizionato un bottone che mostra un messaggio con una breve spiegazione che aiuta l'utente a capire quali preferenze inserire in quella

determinata lista.

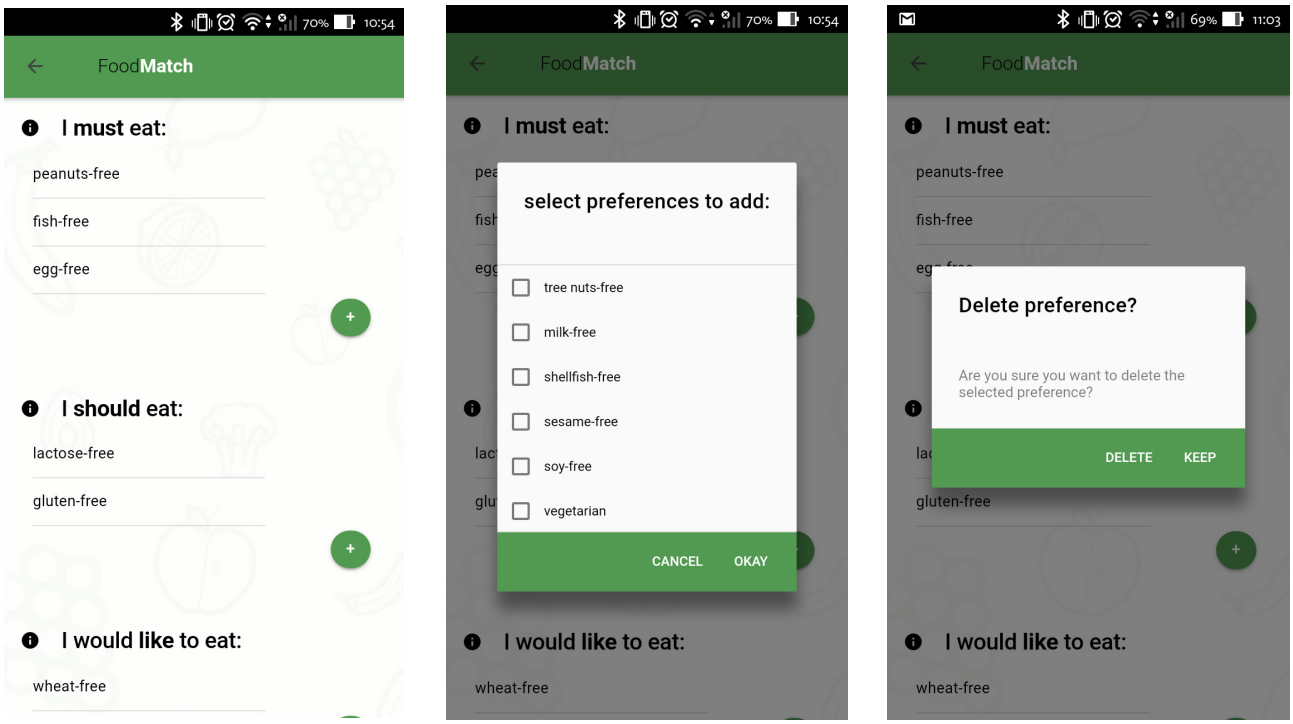

Figura 19: screeshot della PreferencesPage con menu per l'aggiunta e la rimozione delle preferenze

3.3.5 – Scansione del codice QR e visualizzazione del menù

In questa sezione verrà descritta l'implementazione della scansione del codice da parte dell'app e della visualizzazione del menù relativo in forma personalizzata.
Come già anticipato, la scansione tramite fotocamera ed il riconoscimento del testo associato al codice QR è demandato al plugin di Ionic Native chiamato BarcodeScanner.

3.3.5.1 – HomePage

La HomePage è la pagina principale a cui si accede dopo il login. Da qui è possibile aprire il side menu ed avviare lo scanner.

Anche in questo caso la funzione di scansione agisce in maniera asincrona, permettendo di utilizzare i dati contenuti nel QR solo quando la ricezione è avvenuta.

La funzione principale contenuta nel codice della HomePage è quella che si occupa di avviare la scansione e gestire poi i dati, e si chiama *startScanner*. Questa si occupa innanzitutto di avviare lo scanner, ed una volta che questo ha raccolto i dati contenuti nel codice QR, chiama la funzione *getMenuEntries* del provider. Questa si occupa di effettuare la chiamata al servizio *entry/list* del server, e come prima cosa controlla che il codice ottenuto dalla scansione sia presente fra i menuId contenuti nel database. Se questo non è presente (perché si sta facendo la scansione di un codice che non è quello di un menù collegato all'applicazione) si avvisa l'utente con un toast di errore e l'applicazione torna alla HomePage. Se invece il codice viene riconosciuto, il server si occupa di recuperare dal database la lista dei piatti associati a quel menù, e per ognuno di essi invoca la funzione SQL del database per ottenere le informazioni e gli score di ciascuno di questi piatti. Infine, il server ritorna un array di oggetti JSon contenente tutte le informazioni per ciascun piatto. A questo punto la funzione *getMenuEntries* si occupa di inserire i piatti nel giusto array, a seconda del suo punteggio, e da questi poi crea le liste per la visualizzazione nella view. Se l'intero processo va a buon fine, viene effettuato un push della MenuPage, nella quale l'utente può visualizzare il menù filtrato su misura per lui.

In questo caso il template della pagina è molto semplice e minimale, come da mockup: un'ampia area da cui avviare lo scanner e un breve testo di spiegazione.
In questo caso viene anche inserito nella toolbar il bottone per il toggle del side menu, a cui è associata la relativa icona (Ionic mette a disposizione un ampio set di icone, ed associa in automatico la giusta versione a seconda della piattaforma, per utilizzare lo stile grafico più consono).

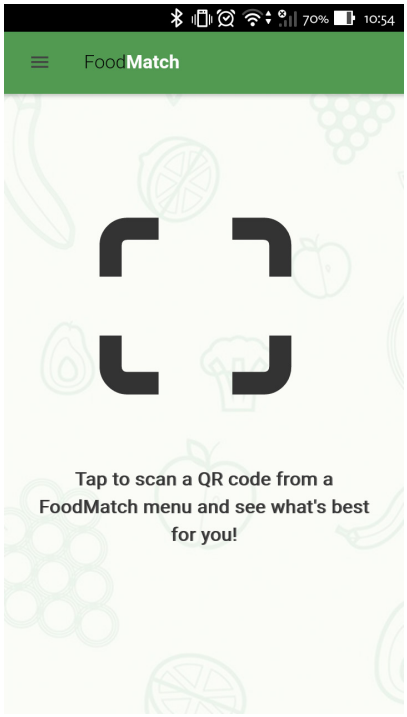

Figura 20: screenshot della HomePage

3.3.5.2 – MenuPage

La MenuPage è quella che mostra il risultato finale della scansione e del calcolo degli score di ogni piatto. Come deciso in fase di progettazione, la visualizzazione delle quattro sezioni è suddivisa su due tab.
Nella prima sezione (la prima della prima tab) vengono visualizzati i piatti classificati come “super match”, cioè quelli che hanno un punteggio positivo. Questi sono i piatti migliori e più affini alle preferenze dell’utente: soddisfano tutti i requisiti obbligatori (di livello 1), tutti i requisiti richiesti (di livello 2) ed almeno uno di quelli preferenziali (di livello 3).
Nella seconda sezione (la seconda della prima tab) vengono visualizzati i piatti classificati come “match”, cioè quelli che hanno un punteggio pari a 0. Questi sono i piatti che l’utente può mangiare in quanto rispettano tutti i requisiti obbligatori e tutti quelli richiesti. Non soddisfano però alcun requisito preferenziale.

Nella terza sezione (cioè la prima della seconda tab) vengono visualizzati i piatti classificati come “bad match”, cioè quelli che hanno totalizzato un punteggio negativo “basso”. Questi sono i piatti che soddisfano sempre tutti i requisiti obbligatori, ma non soddisfano uno o più tra i requisiti richiesti, e può quindi provocare problemi di vario tipo all’utente.

Nella quarta sezione infine (la seconda della seconda tab) sono visualizzati i piatti classificati come “avoid”, cioè quelli che hanno un punteggio negativo “alto”, in quanto non soddisfano uno o più requisiti fondamentali. Se i piatti classificati come “bad match” possono causare problemi all’utente, questi potrebbero risultare addirittura pericolosi, in quanto contenenti uno o più prodotti non compatibili. Ovviamente non è detto che un prodotto in questa categoria sia automaticamente pericoloso dal punto di vista della salute, in quanto questo dipende in gran parte da come l’utente gestisce le sue preferenze: un utente che inserisce tra i requisiti di primo livello “vegan” si vedrà un piatto contenente carne inserito in questa lista, e tuttavia mangiarlo non gli provocherà conseguenze gravi dal punto di vista della salute. Come specificato in fase di analisi e progettazione però questa applicazione vuole comprendere non solo i problemi di salute, ma anche tutte le altre esigenze che un utente può avere, ed è quindi corretto inserire in questa lista qualunque piatto non rispetti queste esigenze.

Il template della MenuPage contiene solo il codice necessario a visualizzare ulteriori due pagine, denominate *MenuMatchPage* e *MenuNoMatchPage* sotto forma di tabs.
Queste due pagine hanno la stessa struttura, come da mockup, costituita da due liste di bottoni cliccabili, una per categoria. Con un click su un qualunque elemento di una delle quattro liste si accede alla pagina dei dettagli di quell’elemento.
Dal punto di vista del codice le pagine relative al menù sono molto semplici, in quanto si limitano a prelevare i dati dalle quattro liste create dal provider e a riportarli sulla view. Le liste in questione sono delle istanze della classe MenuModel, creata appositamente per gestire tutti i dettagli relativi ai piatti del menù, come illustrato nel successivo paragrafo.
La funzione principale inserita in queste due pagine è *entryDetails*, che viene chiamata tramite click su un qualsiasi elemento delle due tab, prende in input l’index dell’elemento e la lista in cui è inserita, ed effettua il push della pagina dei dettagli, passando i due valori presi in input come parametri di navigazione.

3.3.5.2.1 – Il MenuModel

MenuModel costituisce un data model per la creazione di oggetti necessari a contenere le informazioni relative a tutti i piatti di una certa categoria di un menù ricevuto dal server. MenuModel è di fatto un component custom, ed è quindi necessario creare la relativa

classe come esportabile, per poi poterla istanziare ed utilizzare dove serve (in questo caso sempre nel provider).

Il constructor della classe prevede un unico array di elementi di tipo any chiamato items. La classe poi viene dotata di tutte le funzioni helper necessarie a gestire gli elementi di items:

Funzione	Parametri input	Return	Descrizione
addItem	name: string, score: number, id: number, description: string, price: any, category: string	-	Effettua il push di un elemento nell'array items, associando alle chiavi i valori inseriti in input.
refresh	-	-	Elimina tutti gli elementi in Items.
getEntryName	index: number	items[index].name	Restituisce il nome dell'elemento in posizione index.
getEntryDescription	index: number	items[index].description	Restituisce la descrizione dell'elemento in posizione index.
getEntryCategory	index: number	items[index].category	Restituisce la categoria dell'elemento in posizione index.
getEntryPrice	index: number	items[index].price	Restituisce il prezzo dell'elemento in posizione index.

3.3.5.5 – DetailsPage

La details page viene aperta con un click su uno degli elementi delle due tab della MenuPage, e mostra appunto i dettagli per il piatto selezionato. A seconda della lista di provenienza (indicata dal secondo parametro passato all'apertura della pagina), va a prendere l'elemento dal giusto *MenuModel* alla posizione indicata dal primo parametro (*index*) passato all'apertura della pagina.
I dettagli recuperati e visualizzati sono: nome del piatto, prezzo, categoria e descrizione

completa.

Tutte queste informazioni sono inserite in una card (altro component messo a disposizione da Ionic).

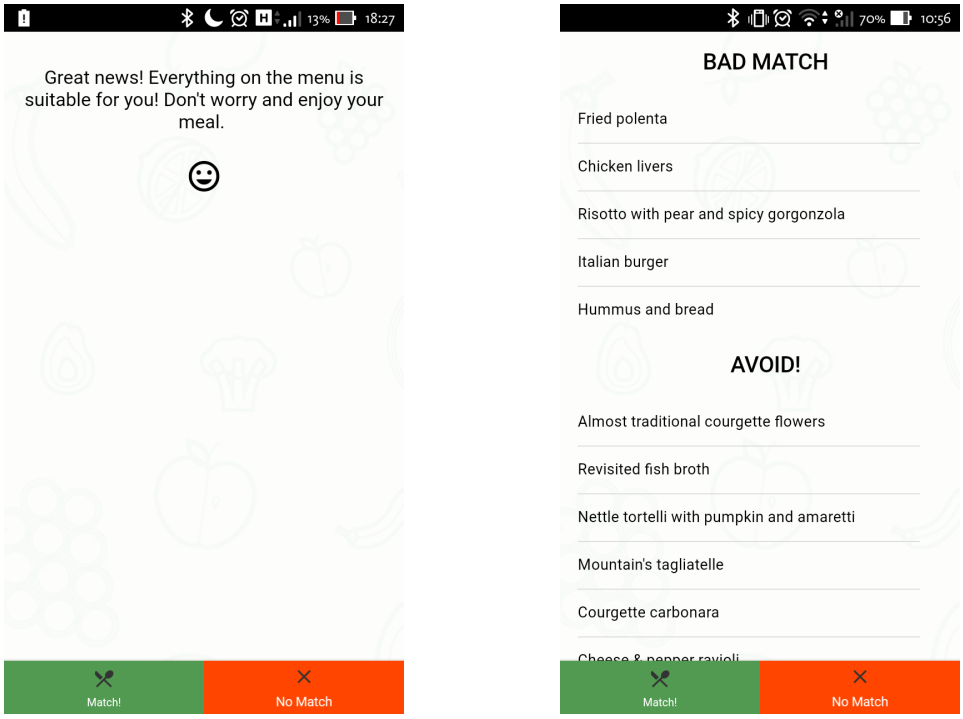

Figura 21: screenshot della tab "No Match" della MenuPage

Figura 22: screenshot della DetailsPage

3.4 – Stilizzazione dell'interfaccia in SASS

Come anticipato, le applicazioni Ionic vengono sviluppate ed eseguite in ambiente web, e quindi il linguaggio utilizzato per definire lo stile dell'interfaccia è CSS3.
Tuttavia, lo sviluppo dei fogli di stile per un'app Ionic non avviene direttamente in CSS, ma in SASS. Questo permette una maggiore flessibilità di utilizzo: SASS infatti è un linguaggio basato su CSS che però ne estende le funzionalità, introducendo le variabili e le funzioni, oltre a permettere una sintassi più compatta e meno rigida rispetto al CSS standard.
Nella pratica è sempre comunque possiblie scrivere un file scss come se fosse un normale foglio di stile CSS, in quanto la compatibilità è totale, tuttavia l'utilizzo di funzioni e variabili può velocizzare di molto la scrittura. Per esempio, nell'implementazione dello stile per FoodMatch, sono state utilizzate diverse variabili, tra cui le più importanti sono *$main-green* e *$text-gray*. Utilizzando queste variabili è possibile cambiare il colore di più elementi all'interno dell'applicazione modificando un solo codice colore (quello che definisce appunto la variabile), piuttosto che andare a cambiare tutti i codici colori associati a ciascun elemento da modificare.
Ionic prevede un file *variables.scss* in cui dichiarare tutte le variabili da usare nell'app, ed un file *app.scss* in cui inserire tutti gli stili da applicare globalmente all'applicazione. Ogni singola pagina poi ha il rispettivo file .scss contenente gli stili da applicare solamente ai componenti della pagina.
Inoltre, è possibile definire degli stili differenti per ciascuna piattaforma, nel caso si vogliano ottenere due versioni grafiche dell'interfaccia, ciascuna coerente con le linee guida della piattaforma di riferimento.
Quando l'applicazione viene compilata, SASS elabora tutti i file .scss in normali fogli di stile CSS, effettuando i binding delle variabili, la traduzione della sintassi eccetera. Questo processo porta ad avere file CSS mediamente molto più pesanti rispetto a quelli scritti manualmente, soprattutto nel caso in cui si siano utilizzate funzioni e variabili (quindi verosimilmente nella maggior parte dei casi).
Ancora una volta quindi si può vedere come l'ambiente di sviluppo ibrido offra molti vantaggi dal punto di vista della versatilità degli strumenti che mette a disposizione ma allo stesso tempo porti con sé problemi intrinseci, che se oggi possono sembrare di poco conto (per uno smartphone moderno non farà troppa differenza se un'applicazione pesa qualche Megabyte in più o in meno), restano comunque da considerare.

4 – Sviluppi Futuri

Sebbene il risultato ottenuto rispecchi in pieno l'idea iniziale e ne rappresenti un'implementazione molto vicina al risultato immaginato, si tratta certamente solo del primo passo; si può dire che lo sviluppo sia completo a livello base, ma sicuramente espandibile. Ritengo infatti che l'idea iniziale possa essere facilmente ampliata nel numero di funzioni e nei modi di utilizzo, e ci sono numerose possibilità di integrazione sia con i social network, sia con altre app molto diffuse (come per esempio quelle di *food delivery* a cui ormai siamo abituati). Se lo sviluppo dovesse continuare con la prospettiva di andare sul mercato, sicuramente includerà una localizzazione tramite Google Maps dei locali aderenti, con filtri a seconda dei tag proposti presenti nei piatti, possibilità di effettuare promozioni per i clienti dell'app, invio di proposte personalizzate agli utenti ed un sistema di recensioni.
Naturalmente sarà necessario implementare una view personalizzata per il gestore del locale, in cui inserire e classificare i piatti proposti.
Lo sviluppo multipiattaforma sarà poi un obiettivo fondamentale, e l'uso di Ionic 2 in questo senso ridurrà di molto il tempo necessario allo sviluppo di una versione iOS e, se ritenuta necessaria, Windows Phone.
Molte idee e molti possibili orizzonti di espansione sono venuti a galla durante lo sviluppo, ed in previsione di ciò la struttura base dell'applicazione è stata progettata in modo da essere poi espandibile in più modi senza dover essere modificata nelle fondamenta.

5 – Conclusioni

Posso dire di essere soddisfatto del lavoro svolto, anche se, come detto nel capitolo precedente, le possibilità di ampliamento sono numerose e potenzialmente molto interessanti.
Sicuramente ci sono state numerose difficoltà nello sviluppo, a partire dal fatto che per la prima volta mi sono confrontato con un progetto completo dal punto di vista delle richieste, e ho lavorato in completa autonomia, e non in team: ho infatti progettato ed implementato sia la parte server, incluso il database, che la parte client. I progetti presentati per gli esami del corso infatti spesso si concentravano maggiormente su uno di questi due aspetti, ed è stato quindi impegnativo e molto stimolante avere la possibilità di lavorare su entrambi allo stesso livello. Un altro aspetto importante è stato l'approccio a nuovi linguaggi di

programmazione e a nuovi framework: ho infatti imparato a conoscere e ad utilizzare PHP (che conoscevo ma che non avevo mai utilizzato per programmare lato server), AngularJS e Typescript (di cui non conoscevo nemmeno l'esistenza prima di iniziare questo progetto), oltre ovviamente ad Ionic 2 e Yii2.
L'altra grande difficoltà è stato l'approccio al mondo delle applicazioni ibride utilizzando Ionic 2. Sebbene il concetto di applicazione ibrida sia per me affascinante, la sua implementazione pratica è stata molto meno semplice di quello che può sembrare in apparenza, e questo è dovuto principalmente al fatto che il framework, nella sua versione 2, è sicuramente dotato di caratteristiche interessanti e funzionali, ma era anche (al momento dell'inizio di questo progetto) in fase beta. Questo ha creato parecchi problemi, prevalentemente di due tipi: di compatibilità fra le varie versioni dei componenti del framework (Cordova, Angula e NodeJS per esempio) e di continuo aggiornamento delle funzioni e delle caratteristiche del framework stesso. Un esempio lampante di questo tipo di problemi è che l'ultima versione di Cordova, quella utilizzata nello sviluppo di questo progetto, non è compatibile con il plugin per la gestione della splashscreen, e non è quindi stato possibile implementarla nell'applicazione; sicuramente questa mancanza verrà risolta a breve, ma sta di fatto che quando sorgono questo tipo di problemi si è sempre e totalmente in mano agli sviluppatori del framework, e finché il bug non viene risolto non è spesso possibile implementare alcune funzioni.

Avendo già acquisito conoscenze di HTML, Javascript (anche se non nello specifico di AngularJS) e CSS, il passaggio necessario a capire i concetti alla base dello sviluppo di applicazioni ibride non è stato troppo impegnativo.
Come già anticipato, il risultato è esternamente non distinguibile da un'applicazione nativa, se non per la reattività dell'interfaccia in alcuni momenti.
Ritengo però che questo tipo di sviluppo possa avere seguito, e che possa diventare un sistema efficace per lo sviluppo di tutte quelle applicazioni che non hanno come caratteristica imprescindibile e fondamentale l'ottimizzazione delle performance e la manipolazioni di grandi quantità di dati, ma che richiedono piuttosto uno sviluppo rapido ed un numero ristretto di sviluppatori.

BIBLIOGRAFIA

[1] ADN Kronos.
http://www.adnkronos.com/sostenibilita/tendenze/2016/01/28/crescono-vegetariani-vegani-italia-sono-della-popolazione_At8SsntHpLwpN7pCCemxYN.html

[2] Meat Atlas 2014: Facts and fi gures about the animals we eat.
https://www.foeeurope.org/sites/default/files/publications/foee_hbf_meatatlas_jan2014.pdf

[3] European Academy of Allergy and Clinical Immunology (EAACI): Food Allergy & Anaphylaxis Public Declaration, 2013.
http://www.anaphylaxisireland.ie/wp-content/uploads/2013/04/FoodAllergyAnaphylaxisPublicDeclarationCombined-1.pdf

[4] U.S. Food & Drug Administration, Guidance for Industry: Questions and Answers Regarding Food Allergens, including the Food Allergen Labeling and Consumer Protection Act of 2004 (Edition 4).
https://www.fda.gov/Food/GuidanceRegulation/GuidanceDocumentsRegulatoryInformation/Allergens/ucm059116.htm

[5] U.S. Food & Drug Administration, Food Allergies: What You Need To Know.
https://www.fda.gov/Food/ResourcesForYou/Consumers/ucm079311.htm

[6] Native vs Hybrid / PhoneGap App Development Comparison.
http://www.comentum.com/phonegap-vs-native-app-development.html

[7] Venture Beat, Mark Zuckerberg Interview.
http://venturebeat.com/2012/09/11/facebooks-zuckerberg-the-biggest-mistake-weve-made-as-a-company-is-betting-on-html5-over-native/