



SAPIENZA
UNIVERSITÀ DI ROMA

FoodFeed: progettazione e realizzazione di un social network per ricette

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Corso di laurea in Ingegneria Informatica

Giovanni Pecorelli
Matricola 1799865

Relatore
Prof. Leonardo Querzoni

A.A. 2019/2020

A Daniele Papa

Indice

1. Introduzione.....	6
1.1 Scopo della tesi	6
1.1 Organizzazione della tesi	6
 2. Raccolta ed analisi dei requisiti	8
2.1 Realtà d'interesse	8
2.2 Specifica dei requisiti: le <i>user stories</i>	8
2.2.1 Registrazione, login, logout	8
2.2.2 Gestione account	10
2.2.3 Utente	10
2.2.4 Moderatore	13
2.2.5 Admin	14
2.3 L'interfaccia: i <i>mockup</i>	15
 3. Raccolta ed analisi dei requisiti	22
3.1 Ruby on Rails e i metodi <i>agile</i>	22
3.2 Conduzione del progetto e organizzazione del lavoro	23
3.3 Pacchetti software aggiuntivi	23
3.3.1 Devise	24
3.3.2 OmniAuth-Facebook	24
3.3.3 Rolify	25
3.3.4 Cucumber	25
3.3.5 RSpec	25
3.3.6 Bootstrap	26
3.4 Spoonacular API	26

4. Analisi concettuale del sistema: la base di dati	27
4.1 Analisi dei requisiti	27
4.2 Progettazione concettuale: lo schema E-R	28
4.2.1 Tabella delle entità	30
4.2.2 Tabella delle relazioni	31
4.2.3 Tabella degli attributi	32
4.2.4 Tabella delle cardinalità	34
4.2.5 Vincoli esterni	36
5. Progettazione del sistema	37
5.1 Il pattern MVC	37
5.2 Data Layer	39
5.3 Application Layer	41
5.4 Presentation Layer	42
6. Validazione e dispiegamento	43
6.1 Testing	43
6.1.1 Behavior-driven development	43
6.1.2 Test-driven development	45
6.2 Distribuzione	48
6.2.1 GitHub	48
6.2.2 Heroku	49
7. Analisi di un caso d'uso	51
Bibliografia	56

1. Introduzione

1.1 Scopo della tesi

L'obiettivo di questa tesi è raccontare l'ideazione e lo sviluppo di un sito web che permetta ai propri utenti di scrivere e condividere ricette culinarie, e allo stesso tempo prendere ispirazione dalle ricette condivise dagli altri utenti. Il sito in questione, chiamato *FoodFeed*, segue le convenzioni dei principali social network nelle funzionalità offerte, in particolare Instagram, e permette quindi di mettere like e commentare ricette, salvarle tra i preferiti, seguire o essere seguiti da altri utenti, e ricevere notifiche riguardanti questi eventi.

L'applicazione è stata realizzata per il corso di Laboratorio di Applicazioni Software e Sicurezza Informatica tenuto dal professor Leonardo Querzoni, nel corso del quale sono stati trattati argomenti relativi alla metodologia di sviluppo Agile con Ruby on Rails.

L'obbiettivo del corso era quello di fornire agli studenti le conoscenze necessarie per dare vita ad un'applicazione web. I primi di agosto 2020 il team di cui facevo parte, composto da un'altra persona oltre a me, ha iniziato a sviluppare l'applicazione: FoodFeed. Partendo dall'ideazione del progetto, a settembre abbiamo portato a termine il lavoro con il rilascio al pubblico della versione finale dell'applicazione.

Questa tesi si pone come obiettivo la spiegazione esaustiva delle scelte di progetto che hanno permesso lo sviluppo del sistema informatico preso in esame, e la descrizione di come si siano utilizzate le tecnologie scelte per la sua implementazione.

1.2 Organizzazione della relazione

Prima di entrare nel vivo della descrizione della realizzazione del progetto, è bene avere una panoramica sugli argomenti che verranno affrontati e di cosa si andrà ad approfondire in ogni capitolo di questa relazione.

Nel Capitolo 2 vengono per prima cosa descritte le realtà d'interesse del progetto e le metodologie utilizzate nella conduzione del progetto e nell'organizzazione del lavoro. Vengono poi riportate le specifiche del sito attraverso le *user stories* e una prima bozza (*mockup*) dell'interfaccia grafica che dovrà soddisfare i requisiti indicati.

A seguire, nel Capitolo 3 verranno descritte le tecnologie, i framework e le piattaforme utilizzate nella realizzazione del progetto.

Nel Capitolo 4 analizzeremo gli elementi più importanti del sito, le loro caratteristiche di nostro interesse, e come si relazionano tra loro. La produzione di uno schema concettuale del progetto e la relativa documentazione porrà le basi su cui andremo effettivamente ad implementare il database dell'applicazione.

Il Capitolo 5 introduce l'architettura a tre *tier* e il pattern MVC, e approfondisce con esempi tratti dal codice l'implementazione dell'oggetto Ricetta (*Recipe*) sui tre strati: Data Layer, Application Layer, Presentation Layer.

Il Capitolo 6 descrive le tecniche di testing utilizzate durante la fase di validazione del sito, e contiene istruzioni su come installare o usufruire dell'app per averne una dimostrazione pratica, sia in locale che pubblicamente in rete.

Nel Capitolo 7 si dà una visione ad alto livello di astrazione di due casi d'uso dell'applicazione, ovvero il login e la pubblicazione di una ricetta, utilizzando diagrammi di attività e screenshot tratti dal sito.

2. Raccolta ed analisi dei requisiti

2.1 Realtà d'interesse

L'idea di FoodFeed nasce come ricerca di un punto di unione tra un forum e un social network.



Dai forum FoodFeed riprende l'organizzazione gerarchica degli utenti, divisi in ruoli quali admin, moderatori e utenti semplici, la possibilità di vedere i post in ordine cronologico, di filtrarli per categoria e di vedere quelli con più voti; dai social riprende la possibilità di mettere like e di seguire/essere seguiti dagli altri utenti, e in generale l'aspetto. Questo ibrido permette di creare uno spazio monotematico adatto a condividere le proprie ricette e ad esplorarne di nuove, e che allo stesso tempo abbia funzionalità più social, come seguire i propri amici per essere aggiornati sulle loro attività culinarie. Tuttavia, rispetto ai social più famosi già presenti, FoodFeed mette a disposizione delle apposite funzionalità che lo rendono adatto come luogo di condivisione per le ricette, come ad esempio i campi per gli ingredienti e una stima del tempo di preparazione durante la creazione di un nuovo post.

2.2 Specifica dei requisiti: le *user stories*

Terminata questa descrizione a grandi linee del progetto che si vuole andare a sviluppare, entriamo più nel dettaglio e definiamo in modo molto riassuntivo ma completo tutte le funzionalità che il sito dovrà offrire una volta ultimato.

Per fare ciò utilizziamo le cosiddette *user stories*, ovvero descrizioni scritte in linguaggio comune e informale delle funzionalità del sistema. Il formato scelto per la loro stesura è il seguente:

COME <ruolo> VOGLIO <feature> PER <scopo>

2.2.1 Registrazione, login, logout

Per prima cosa, ogni social network che si rispetti deve offrire ad un visitatore la possibilità di registrarsi se non è mai acceduto prima al sito, e di accedere in caso contrario. FoodFeed si spinge un passo oltre e rende mandatoria l'iscrizione e l'accesso al sito per vedere i suoi contenuti, vale a dire che un utente non registrato non può

visitare il sito e verrà sempre reindirizzato alla pagina di login. Per rendere più veloce l'accesso si è pensato di inserire la possibilità di iscriversi in modo più semplice e veloce con un account esterno, ovvero quello di Facebook. L'alternativa è la convenzionale registrazione con email, password, e scelta di uno username.

Di seguito le *user stories* correlate:

1 Registrazione

COME utente non registrato VOGLIO creare un nuovo account con email e password PER diventare un utente registrato

2 Registrazione OAuth

COME utente non registrato VOGLIO creare un nuovo account con Facebook PER diventare un utente registrato

3 Login

COME utente registrato VOGLIO effettuare il login con email e password PER utilizzare le funzionalità del sito

4 Login OAuth

COME utente registrato VOGLIO effettuare il login con Facebook PER utilizzare le funzionalità del sito

5 Logout

COME utente che ha effettuato il login VOGLIO effettuare il logout PER uscire dal sito

2.2.2 Gestione account

Durante la fase di registrazione un utente fornisce al sito alcune informazioni che potrebbero cambiare in futuro, ad esempio la foto profilo: viene offerta quindi la possibilità di cambiare questi dati. Voglio anche potermi cancellare dal sito, quindi:

6 Modifica account

COME utente registrato VOGLIO modificare il mio account PER cambiare le informazioni personali

7 Elimina account

COME utente registrato VOGLIO eliminare il mio account PER rimuovere i miei dati dal sito

2.2.3 Utente

L'utente è un visitatore del sito una volta che ha effettuato l'accesso. Ha i permessi minimi rispetto agli altri ruoli del sito, ma ha accesso a quasi tutte le funzionalità e pagine.

Innanzitutto deve essere in grado di postare e gestire le proprie ricette, la risorsa alla base di tutto il sistema:

8 Pubblicare una ricetta

COME utente VOGLIO pubblicare una ricetta PER condividerla con gli altri utenti

9 Modifica ricetta

COME utente VOGLIO modificare una ricetta che ho condiviso PER aggiungere informazioni o correggerle

10 Rimuovi ricetta

COME utente VOGLIO rimuovere una ricetta che ho postato PER eliminarla dal sito

11 Visualizza ricetta

COME utente VOGLIO visualizzare ricette di altri utenti PER leggere informazioni aggiuntive, preparazione e commenti

Poi, può interagire con i post mettendo like, commentandoli o salvandoli tra i propri preferiti:

12 Mettere/togliere like

COME utente VOGLIO mettere/togliere like alle ricette PER visualizzare le ricette più apprezzate dagli utenti

13 Commentare

COME utente VOGLIO commentare le ricette PER esprimere la mia opinione

14 Aggiungi/rimuovi preferiti

COME utente VOGLIO gestire la mia lista di ricette preferite PER tenere traccia delle mie ricette preferite

15 Visualizza preferiti

COME utente VOGLIO visualizzare la mia lista di ricette preferite PER trovare una ricetta che ho salvato

Può cercare altri utenti per username (unico all'interno del sito) per vederne il profilo e in caso seguirli; inoltre deve ricevere notifiche riguardo like, commenti e follow ricevuti:

16 Cercare utenti

COME utente VOGLIO cercare un altro utente con il suo username PER visualizzare il suo profilo

17 Follow

COME utente VOGLIO seguire altri utenti PER vedere le loro ricette

18 Follower/following

COME utente VOGLIO avere a disposizione la lista dei miei follower/following PER vedere il profilo di altri utenti

19 Notifiche

COME utente VOGLIO ricevere notifiche PER essere aggiornato sulle interazioni con altri utenti

Infine, l'utente deve poter visualizzare le ricette presenti all'interno del sito.

Nella 'Homepage' di FoodFeed vengono mostrate solo le ricette pubblicate dagli utenti che segue; nella sezione 'Discover' vengono mostrate le più recenti, ma con la possibilità di filtrarle per categoria; nella sezione 'Top' vengono mostrate le ricette che hanno ricevuto più like nell'ultima settimana. Inoltre ogni giorno il sito stesso consiglia una ricetta diversa nella pagina 'Ricetta del giorno'. L'utente può infine visualizzare la pagina dei 'Contatti'.

20 Homepage

COME utente VOGLIO visualizzare la homepage PER vedere le ricette delle persone che seguo

21 Filtrare per categorie

COME utente VOGLIO impostare filtri mentre visualizzo le ricette PER limitare la ricerca a determinate categorie

22 Top Ricette

COME utente VOGLIO visualizzare la sezione Top PER scoprire le migliori ricette della settimana

23 Ricetta del giorno

COME utente VOGLIO visualizzare la Ricetta del giorno PER prendere ispirazione

24 Contatti

COME utente VOGLIO visualizzare la pagina Contatti PER visualizzare informazioni sugli sviluppatori

25 Mail agli sviluppatori

COME utente VOGLIO inviare una mail agli sviluppatori PER consigliare nuove features o segnalare bug

2.2.4 Moderatore

Un moderatore è sostanzialmente un utente scelto da un admin per gestire post e commenti all'interno del sito. Li può rimuovere in caso non siano adeguati, e ha anche la capacità di sospendere il profilo degli utenti semplici, impedendone l'accesso al sito ma mantenendone il profilo. Per il resto è in grado di utilizzare il sito normalmente.

User stories correlate:

26 Rimozione ricette

COME moderatore VOGLIO rimuovere le ricette degli utenti PER moderare il sito

27 Rimozione commenti

COME moderatore VOGLIO rimuovere i commenti degli utenti PER moderare il sito

28 Ban

COME moderatore VOGLIO sospendere gli utenti PER moderare il sito

29 Permessi moderatore

COME moderatore VOGLIO avere le stesse funzionalità degli utenti PER utilizzare il sito

2.2.5 Admin

L'admin è il gestore del sito, e l'unico in grado di promuovere e retrocedere gli utenti semplici a moderatori e viceversa. È anche l'unico a poter sospendere i profili dei moderatori. Le user stories riguardanti questo ruolo sono quindi:

30 Promozione

COME admin VOGLIO promuovere utenti a moderatori PER gestire i ruoli del sito

31 Retrocessione

COME admin VOGLIO retrocedere moderatori a utenti PER gestire i ruoli del sito

32 Permessi moderatore

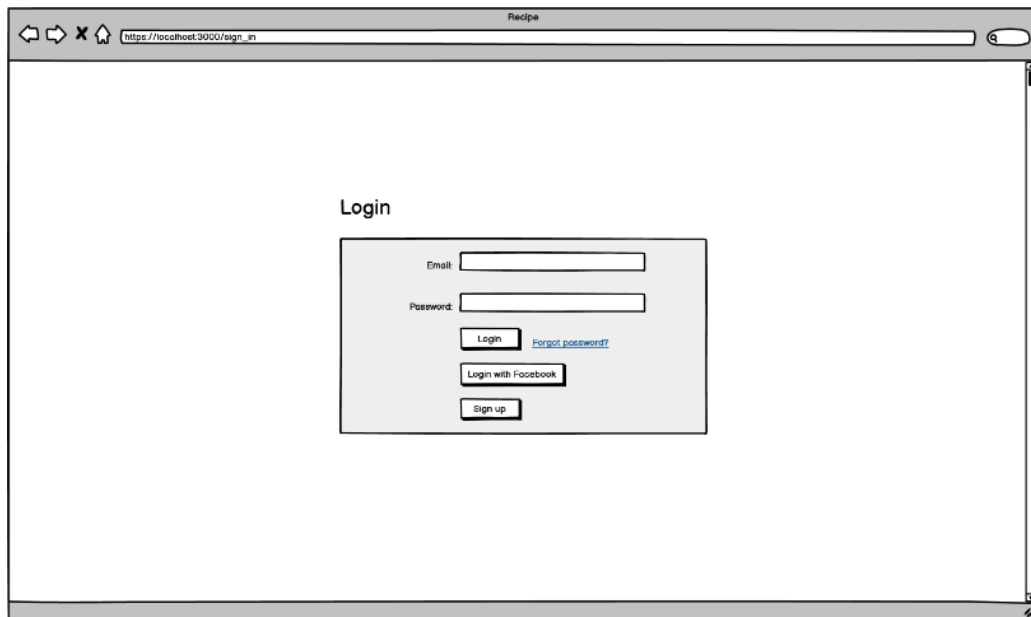
COME moderatore VOGLIO avere le stesse funzionalità dei moderatori PER utilizzare il sito

2.3 L'interfaccia: i *mockup*

L'utilizzo più comune dei *mockup* nello sviluppo del software è quello di creare *UI* (*user interfaces*) per dare un'idea al cliente, ma anche agli stessi sviluppatori, di che aspetto avrà a grandi linee il software finale, senza il bisogno di scrivere una sola linea di codice.

Nel nostro caso i mockup sono stati generati con un programma chiamato *Balsamiq*. Sono molto minimali, con uno stile che ricorda un disegno eseguito a mano e ci fanno capire quante e quali saranno le pagine all'interno del sito, oltre a dare un'idea su quali funzionalità abbiano e quindi a quali user stories siano collegate.

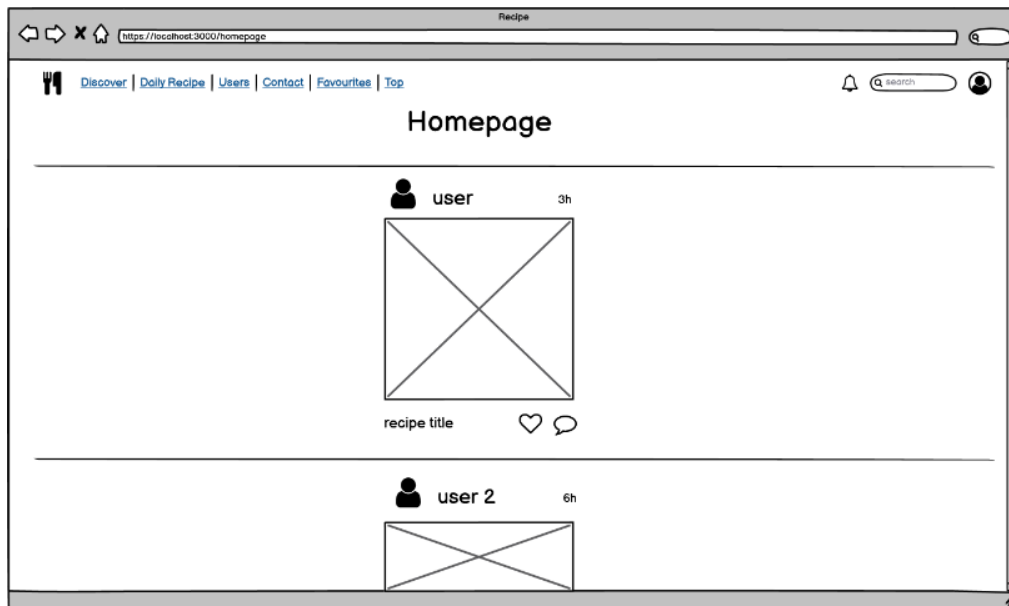
Nelle prossime pagine sono riportati questi mockup con annesso l'elenco delle user stories che vengono soddisfatte in quella schermata.



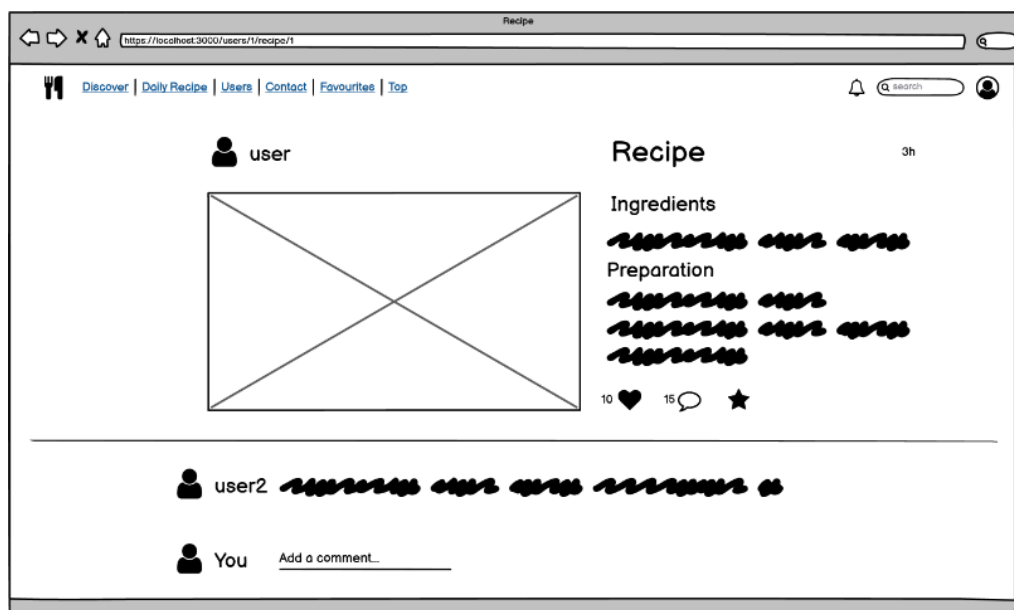
La prima pagina è la schermata di login, che permette di entrare inserendo email e password. Qui viene effettuato anche l'accesso con Facebook, e si recupera la password in caso venga dimenticata. Le user stories corrispondenti sono la 3 – *Login* e la 4 – *Login OAuth*.



Nella schermata di registrazione invece vengono mappate la 1 – *Registrazione* e 2 – *Registrazione OAuth*.

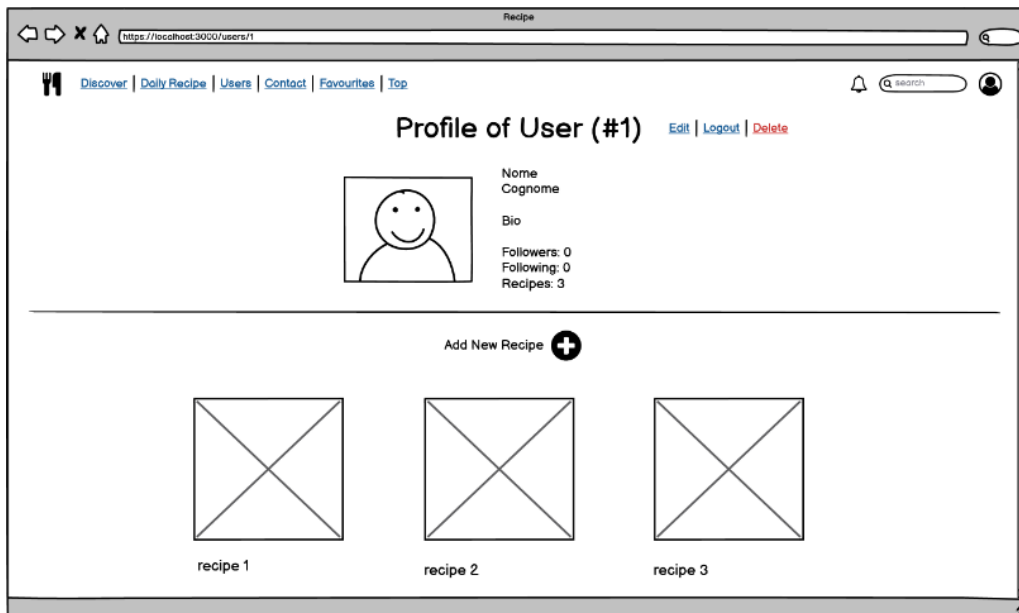


La Homepage è la prima pagina che si visita dopo il login. La barra di navigazione in alto riporta alle altre pagine del sito. La user story corrispondente è la 20 – *Homepage*.

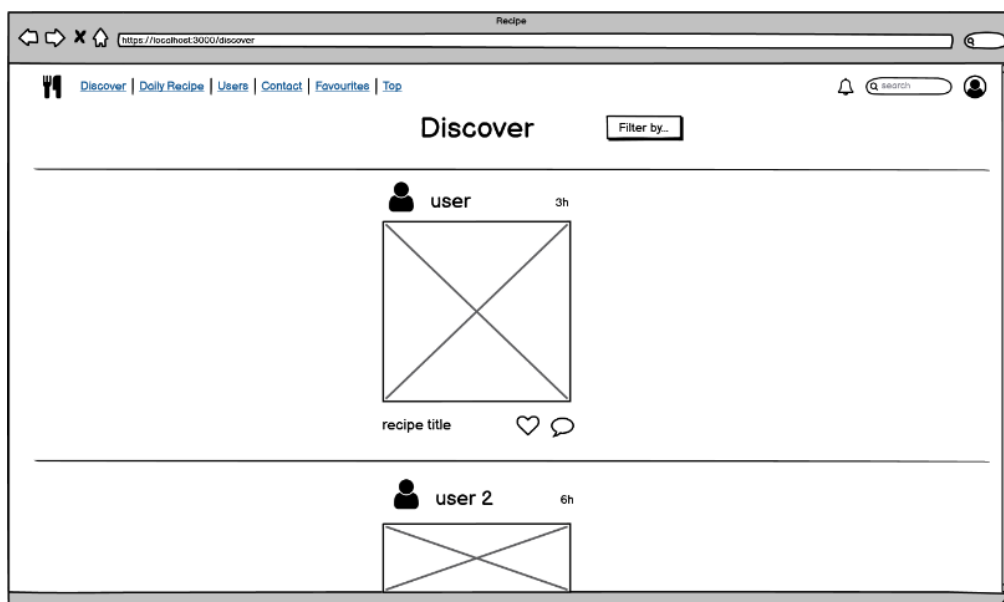


Nella pagina di una singola ricetta posso leggere le sue informazioni più in dettaglio e interagire con essa. Nel caso l'utente che visita la pagina sia lo stesso che ha postato la ricetta, da qui può anche modificarla o rimuoverla.

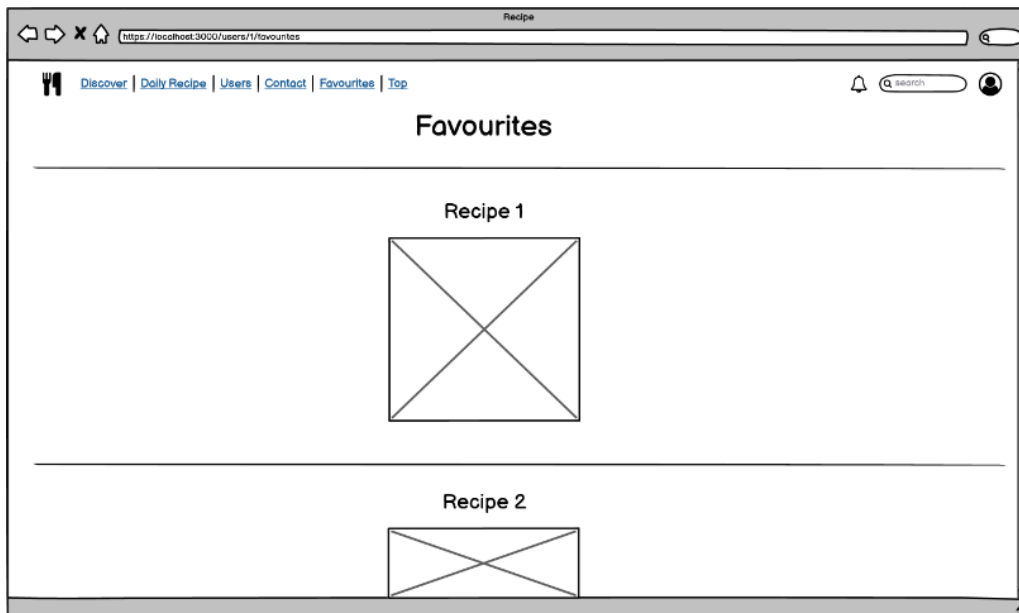
Su questa pagina sono mappate molte user stories: 9 – *Modifica ricetta*, 10 – *Rimuovi ricetta*, 11 – *Visualizza ricetta*, 12 – *Mettere/togliere like*, 13 – *Commentare*, 14 – *Aggiungi/rimuovi preferiti*.



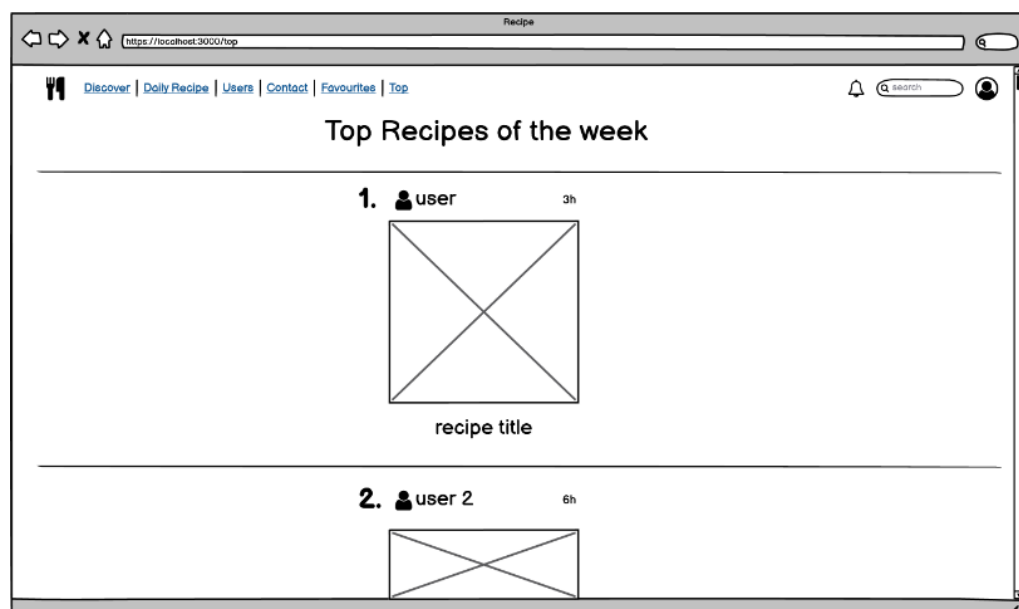
Nella pagina profilo di ogni utente è possibile leggere le informazioni a riguardo ed avere elencate le sue ricette. Se è il proprio profilo, si può pubblicare una nuova ricetta, modificare ed eliminare il profilo o eseguire il logout. Se invece si sta visitando il profilo di qualcun altro, qui è dove lo si può seguire (o smettere di seguirlo). Le user stories mappate qui sono: 5 – Logout, 6 – Modifica account, 7 – Elimina account, 8 – Pubblicare una ricetta, 15 – Visualizza preferiti, 17 – Follow, 18 – Follower/following.



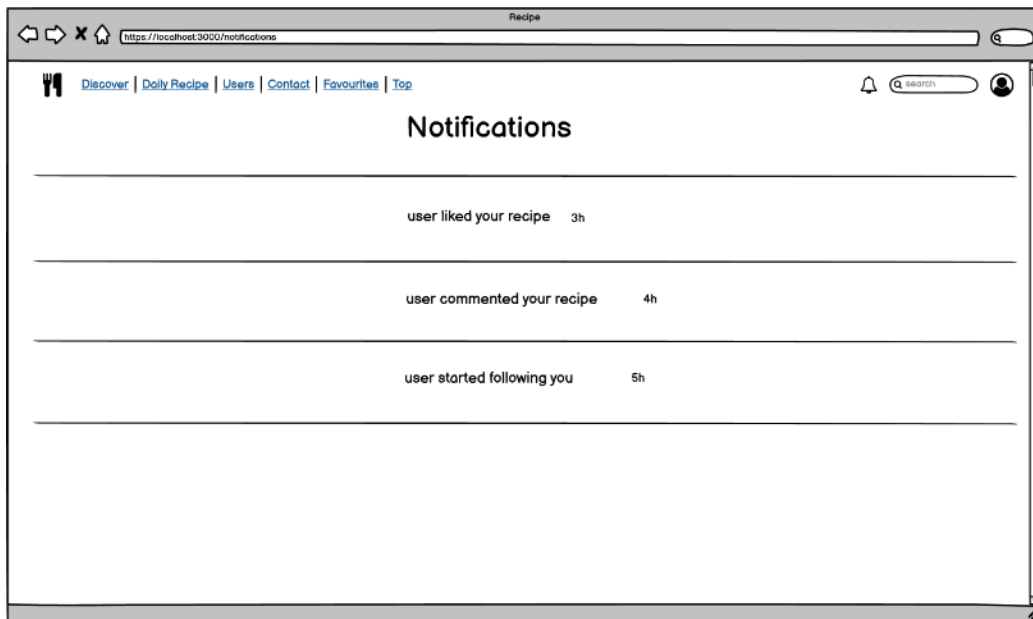
Nella pagina 'Discover' si esplorano le ricette più recenti pubblicate sul sito, e qui si possono filtrare le ricette per categoria (prezzo, difficoltà, tempo di preparazione, intolleranze alimentari...) come richiesto dalla user story 21 – Filtrare per categorie.



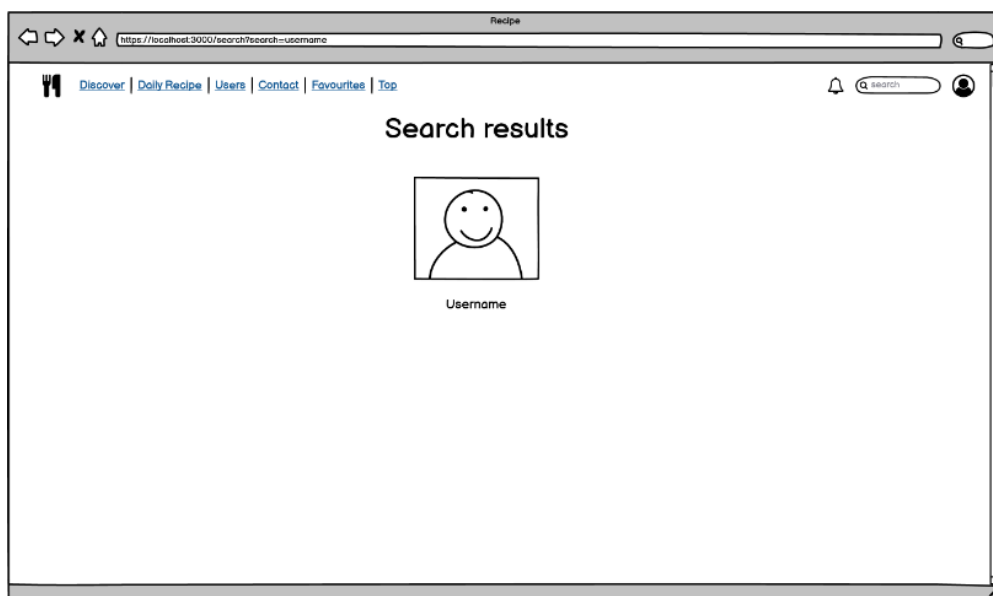
Nella pagina 'Favourites' vengono mostrate le ricette preferite di un utente. User story 15 – *Visualizza preferiti*.



Nella pagina 'Top Recipes of the Week' vengono mostrate le ricette migliori della settimana. La user story in questo caso è la 22 – *Top Ricette*.



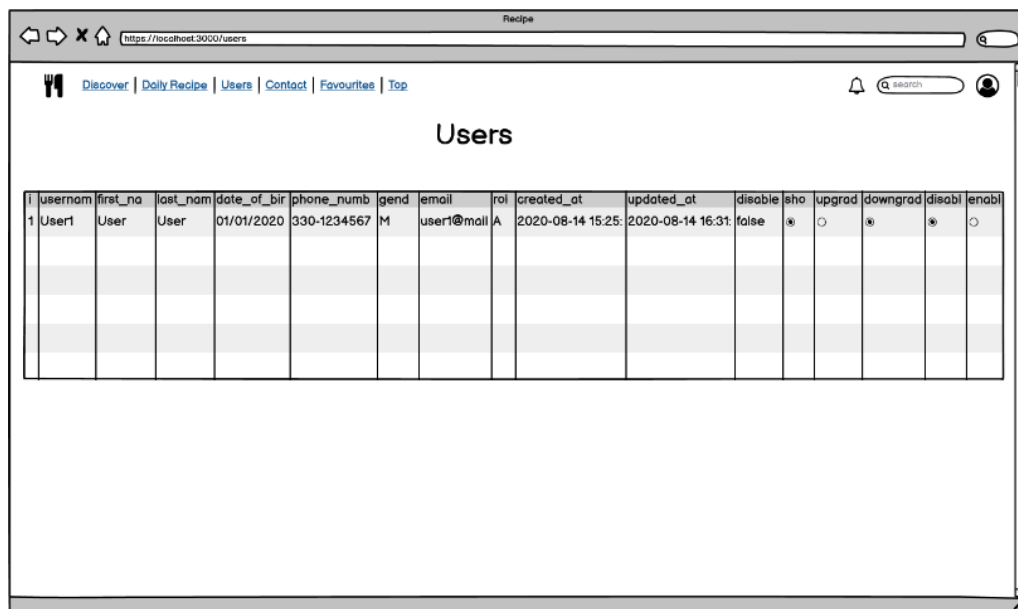
Nella pagina 'Notifications', privata per ogni utente, vengono elencate tutte le notifiche ricevute, come richiesto dalla user story 19 – *Notifiche*.



In questa pagina invece viene mostrato il risultato in seguito alla ricerca di un utente. User story 16 – *Cercare utenti*.



Nella pagina 'Contacts' vengono visualizzate informazioni sugli sviluppatori del sito, e qui vengono mappate le user stories 24 – *Contatti*, 25 – *Mail agli sviluppatori*.



Infine è disponibile solo a moderatori e admin una pagina 'Users' contenente una lista di tutti gli utenti del sito, che permette di eseguire tutte azioni richieste dalle user stories di questi due ruoli, dalla 26 alla 32.

3. Tecnologie e Metodologie Utilizzate

3.1 Ruby on Rails e i metodi *agile*

Per la realizzazione di questo progetto *full-stack* si è utilizzato *Ruby on Rails*, un framework per sviluppare applicazioni web.

La potenza di Rails è il racchiudere i meccanismi di programmazione all'interno di un modello di sviluppo che permette di ridurre drasticamente i tempi di sviluppo, mantenendo un codice semplice e più snello rispetto ad altri framework.

Il linguaggio Ruby, a sua volta, fu progettato per essere semplice e immediato e per questo motivo segue il principio di minima sorpresa (POLS), ovvero il linguaggio si comporta così come il programmatore si aspetta, in maniera coerente e senza imprevisti.

Non è dunque un caso che molti progetti software moderni, anche di grandi dimensioni, siano stati programmati con *RoR*; tra gli altri ricordiamo GitHub, AirBnB, Tumblr, Soundcloud, Twitter.¹

L'utilizzo di *Ruby on Rails* richiede l'utilizzo dei metodi di programmazione cosiddetti *agile*, meno strutturati ma più focalizzati sull'obiettivo di consegnare il software al cliente in tempi brevi e con frequenza.

In particolare ci si è concentrati sull'utilizzo di due tecniche di sviluppo del software.

La prima è lo sviluppo basato sui test, o *TDD (test-driven development)*, un modello che si basa sul passare i test automatici precedentemente predisposti, e che quindi prevede che la stesura dei test automatici avvenga prima di quella del software che deve essere sottoposto a test.

La seconda è lo sviluppo basato sul comportamento, o *BDD (behavior-driven development)*. Questa pratica, che incoraggia la collaborazione tra cliente, manager e sviluppatore, prevede la definizione a priori del funzionamento del sito, attraverso le cosiddette *user stories*, ovvero la scrittura in linguaggio naturale di scenari di utilizzo del sito e risultati attesi.

Entrambi i metodi di sviluppo saranno approfonditi nel Capitolo 6.1.

¹ Twitter è stato solo inizialmente sviluppato in Ruby on Rails, per poi spostarsi su Java e Scala

3.2 Conduzione del progetto e organizzazione del lavoro

Lo sviluppo del progetto è stato gestito da due sviluppatori: Giovanni Pecorelli e Jacopo Rossi. La modalità di sviluppo è stata a distanza, a causa dell'impossibilità di incontrarsi in sede universitaria dovuta all'emergenza sanitaria che ha colpito il paese quest'anno. Io ed il mio collega ci siamo quindi organizzati per lavorare da casa.

Per prima cosa è stato aperto un repository su *GitHub* per la condivisione del codice. In fase di scrittura del codice invece, abbiamo lavorato quasi sempre in contemporanea, grazie all'estensione *Live Share* di *Visual Studio Code*, l'*IDE* scelto per questo progetto. *Live Share* permette ad un utente di condividere il proprio spazio di lavoro in modo che più persone possano lavorare in contemporanea, visualizzando in tempo reale i cambiamenti apportati al codice dai collaboratori. Non solo viene condiviso la directory contenente il codice, ma anche il terminale e la porta del *localhost* su cui gira il sito.

L'intero ciclo di realizzazione del progetto, dalla fase di ideazione a quella di testing e pubblicazione, passando per la pianificazione delle user stories e la loro effettiva implementazione *full-stack*, ha richiesto poco meno di un mese, che ci è sembrato un periodo adeguato per lo sviluppo di un progetto a partire da zero in un linguaggio e con l'utilizzo di framework per entrambi completamente nuovi.

3.3 Pacchetti software aggiuntivi

Come riportato nella sezione precedente, il progetto è stato sviluppato partendo da zero. Tuttavia ciò non significa che non ci si sia avvalsi di strumenti esterni. In particolare, la programmazione con Ruby incoraggia all'utilizzo delle cosiddette "gemme" esterne, ovvero pacchetti che contengono programmi o librerie. Queste gemme permettono di rendere il codice modulare e intuitivo, e semplificano il lavoro, rimuovendo la necessità di scrivere da capo porzioni di codice già esistenti.

Tra le oltre 30 gemme utilizzate, vengono descritte di seguito le più interessanti.

3.3.1 Devise

Una delle gemme più importanti nel progetto è stata *devise*, che ci ha permesso di gestire la fase di registrazione, login e logout.



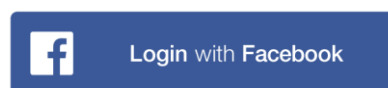
Grazie alle molte funzionalità offerte da devise, abbiamo potuto rendere le credenziali di accesso a FoodFeed *rememberable* e *recoverable*. *Rememberable* significa che un utente può decidere dopo il primo login di far ricordare al sito le proprie credenziali di accesso e saltare la fase di login; *recoverable* invece rende possibile l'invio di una mail che permette di impostare una nuova password nel caso venga dimenticata quella corrente.

Una volta impostato il funzionamento di devise sul modello *User*, si hanno a disposizione tra gli altri i metodi *current_user*, *user_signed_in?* E *before_action :authenticate_user!* che ci hanno permesso di proteggere l'accesso alle risorse del sito, per esempio garantendo che solo l'utente stesso possa modificare il proprio profilo o rimuovere una ricetta pubblicata. Inoltre viene verificato che l'accesso al sito stesso sia permesso solo agli utenti registrati che abbiano eseguito l'accesso.

Infine, devise mette a disposizione di default delle routes preimpostate che vanno a mapparsi su delle views che hanno la pagina html già scritta. Anche se inizialmente sono molto semplici, sono state il punto di partenza nello sviluppo del front-end del sito.

3.3.2 OmniAuth-Facebook

Per permettere ad un utente di registrarsi e accedere a FoodFeed con il suo account Facebook, si è optato per l'accesso con protocollo OAuth tramite API Facebook.



OAuth è un protocollo aperto che permette ad un service provider (in questo caso Facebook) di condividere le informazioni personali di un utente con un consumer (in questo caso FoodFeed) senza cedere le credenziali personali, il tutto ovviamente con il consenso dell'utente.

Per realizzare questa feature dobbiamo innanzitutto registrare l'app FoodFeed nella pagina *Facebook for Developers*. Ci verranno restituite due chiavi, un *ID* e un *secret*, gestite grazie ad un'altra gemma, *dotenv-rails*. Il processo è gestito da una chiamata asincrona verso il server Facebook contenente un token di identificazione che verrà confermato lato server. Questo in fase di registrazione ci restituirà informazioni sull'utente che sono state richieste nella chiamata, ad esempio nome, cognome, sesso,

data di nascita e foto profilo, e sarà nostro compito salvare le informazioni nel database creando un nuovo utente.

La gemma *omniauth-facebook* riduce il codice necessario a scrivere queste istruzioni, e si integra perfettamente con *devise*, la gemma utilizzata per gestire il normale processo di registrazione e login con email e password.

3.3.3 Rolify

Abbiamo cominciato a parlare della divisione tra ruoli nel Capitolo 2.2 e ne discuteremo ancora nel Capitolo 4.

La creazione di questi ruoli e l'assegnazione dei permessi ai vari utenti sono state gestite grazie alla gemma *rolify*. L'utilizzo è molto semplice: basta dichiarare che il modello User deve essere *rolifiable*, e sia avranno a disposizione molti metodi tra i quali i più importanti ed utilizzati sono stati *add_role*, *remove_role*, *has_role?*, *assign_default_role*, il cui utilizzo è comprensibile anche senza ulteriori spiegazioni, applicabili alle istanze di user.

3.3.4 Cucumber

La gemma *cucumber* si occupa di aiutarci nella fase di testing del progetto. Per approfondire quest'argomento e si veda il Capitolo 6.1.1 in cui verranno illustrati i test eseguiti sul nostro progetto con cucumber e con rspec (paragrafo successivo).



Riassumendo qui in breve, cucumber si occupa di verificare la correttezza (o il fallimento) dei test, scritti sottoforma di scenari. Questi scenari non sono altro che le *user stories* discusse nel Capitolo 2 scritte in linguaggio Gherkin, un insieme di regole grammaticali che permettono a cucumber di interpretare questo linguaggio quasi naturale come una serie di istruzioni da eseguire all'interno del progetto.

3.3.5 RSpec

RSpec è un'altra gemma utilizzata in fase di testing. La principale differenza con cucumber è la minore leggibilità dei test per il cliente e i non programmatori. Per questo motivo è più comune che RSpec venga utilizzato per eseguire test minori in

dimensioni e complessità, ad esempio testando la validità dei singoli modelli, controllers, views, routes. Anche i test con RSpec verranno illustrati nel Capitolo 6.1.2.

3.3.6 Bootstrap

Originariamente creato da uno sviluppatore di Twitter, Bootstrap oggi è probabilmente il più conosciuto framework per lo sviluppo del lato front-end di applicazioni per il web. Bootstrap contiene modelli di progettazione basati su HTML e CSS, sia per la tipografia, che per le varie componenti dell'interfaccia, come moduli, pulsanti e navigazione.



All'interno di FoodFeed si può ritrovare l'utilizzo di Bootstrap soprattutto nei bottoni, nei form, negli input e nella barra di navigazione in cima alla pagina.

3.4 Spoonacular API

Per realizzare la pagina della Ricetta del Giorno (/dailyrecipe) su FoodFeed, che propone ogni giorno una ricetta diversa, abbiamo pensato di fare affidamento ad un sito esterno. Spoonacular.com è un sito che si occupa della pianificazione pasti, e offre informazioni su ingredienti, calorie consumate e simili. Oltre a tutto ciò offre ciò che cerchiamo, ovvero un buon servizio di API che permette di recuperare informazioni sulle varie ricette che sono presenti nel loro database. Al momento della pubblicazione queste ricette sono oltre 1.100.000, contando anche quelle inviate dagli utenti di Spoonacular.



Avere accesso ai dati di queste ricette è semplice: bisogna fare una chiamata con metodo GET all'url `https://api.spoonacular.com/recipes/ID/information`.

Per avere l'autorizzazione a fare questa chiamata, bisogna registrarsi al sito di Spoonacular come sviluppatori. Questo ci fornirà una *key* da appendere all'url in questo modo: `https://.../information?apiKey=SPOONACULAR_KEY`, dove `SPOONACULAR_KEY` è la chiave presa dal file `.env`.

Per avere ogni giorno una ricetta casuale, l'ID della ricetta è generato casualmente con `srand(seed)`, dove il `seed` è generato a partire dalla data odierna (`Date.today`).

Spoonacular risponde alla nostra richiesta con un file JSON, da cui recuperiamo i campi di nostro interesse, che sono *title*, *image*, *summary* e *instructions*.

4. Analisi concettuale del sistema: la base di dati

Ora che abbiamo un'idea su come sia organizzata la struttura di FoodFeed, possiamo cominciare ad occuparci dell'analisi dei requisiti che ci siamo imposti e organizzare uno schema che contenga gli elementi principali del sito e le loro relazioni. Al termine di questo capitolo quindi avremo prodotto il cosiddetto schema E-R (entità-relazione) del progetto e la relativa documentazione. Questo sarà di enorme aiuto nella creazione del database di FoodFeed.

4.1 Analisi dei requisiti

Il primo passo nella creazione di una base di dati è individuare gli elementi del progetto che possono essere considerati come entità, che diventeranno i modelli della nostra architettura MVC. Li riportiamo a seguire, con una breve descrizione per ognuno.

USER. Lo User è l'utente del sito, a prescindere dal ruolo che assume. Dello User ci interessano lo username (unico all'interno del sito), il nome, il cognome, l'email, la password, il ruolo. Altri campi opzionali che possono interessarci sono la data di nascita, il numero di telefono, il sesso, la biografia e l'immagine del profilo.

RECIPE. La Recipe è la ricetta postata da un utente. Di ogni Recipe ci interessa il titolo, gli ingredienti, la preparazione e l'immagine. Oltre a queste informazioni obbligatorie l'utente può anche decidere di aggiungere di che portata si tratta, se il piatto segue una particolare dieta, o se è adatto per chi ha determinate intolleranze. Si può inserire il tempo di preparazione della ricetta, e valutarne prezzo e difficoltà su una scala da 1 a 5. Vogliamo anche sapere il numero di like e commenti relativi alla ricetta.

COMMENT. Dei commenti ci interessa la ricetta a cui si riferiscono e l'utente che l'ha scritto, oltre ovviamente al testo contenuto nel commento.

LIKE. Dei like ci interessano solo la ricetta a cui si riferiscono e l'utente che l'ha messo.

ROLE. Il ruolo rappresenta i permessi che un utente ha nei confronti di una risorsa, che può essere una ricetta, un commento o un altro utente, o nei confronti di un'intera categoria di risorse. Per esempio un utente semplice ha i permessi di Mod rispetto alle

proprie ricette, mentre un Mod ha gli stessi permessi su tutte le ricette del sito. Di ogni ruolo vogliamo sapere di che tipo di ruolo si tratta ('U', 'M' o 'A'), il tipo di risorsa a cui si riferisce e in alcuni casi l'id di tale risorsa.

4.2 Progettazione concettuale: lo schema E-R

Partendo dagli elementi che abbiamo appena identificato siamo ora in grado di costruire uno schema che li metta in relazione tra loro. Questo schema è lo schema E-R. Gli elementi del paragrafo precedente saranno rappresentati nello schema come entità, le loro caratteristiche di nostro interesse saranno i loro attributi e i legami tra due entità diventeranno relazioni.

Lo schema E-R da solo però non è in grado di rappresentare esaustivamente tutti gli aspetti di un'applicazione. È indispensabile corredare ogni schema E-R con una documentazione per facilitare l'interpretazione dello schema e a descrivere proprietà dai dati rappresentati che non possono essere espressi direttamente dai costrutti del modello. A seguito dello schema quindi seguono delle tabelle.

La prima descrive le entità dello schema con il nome, una breve definizione in linguaggio naturale, l'elenco di tutti gli attributi e l'identificatore.

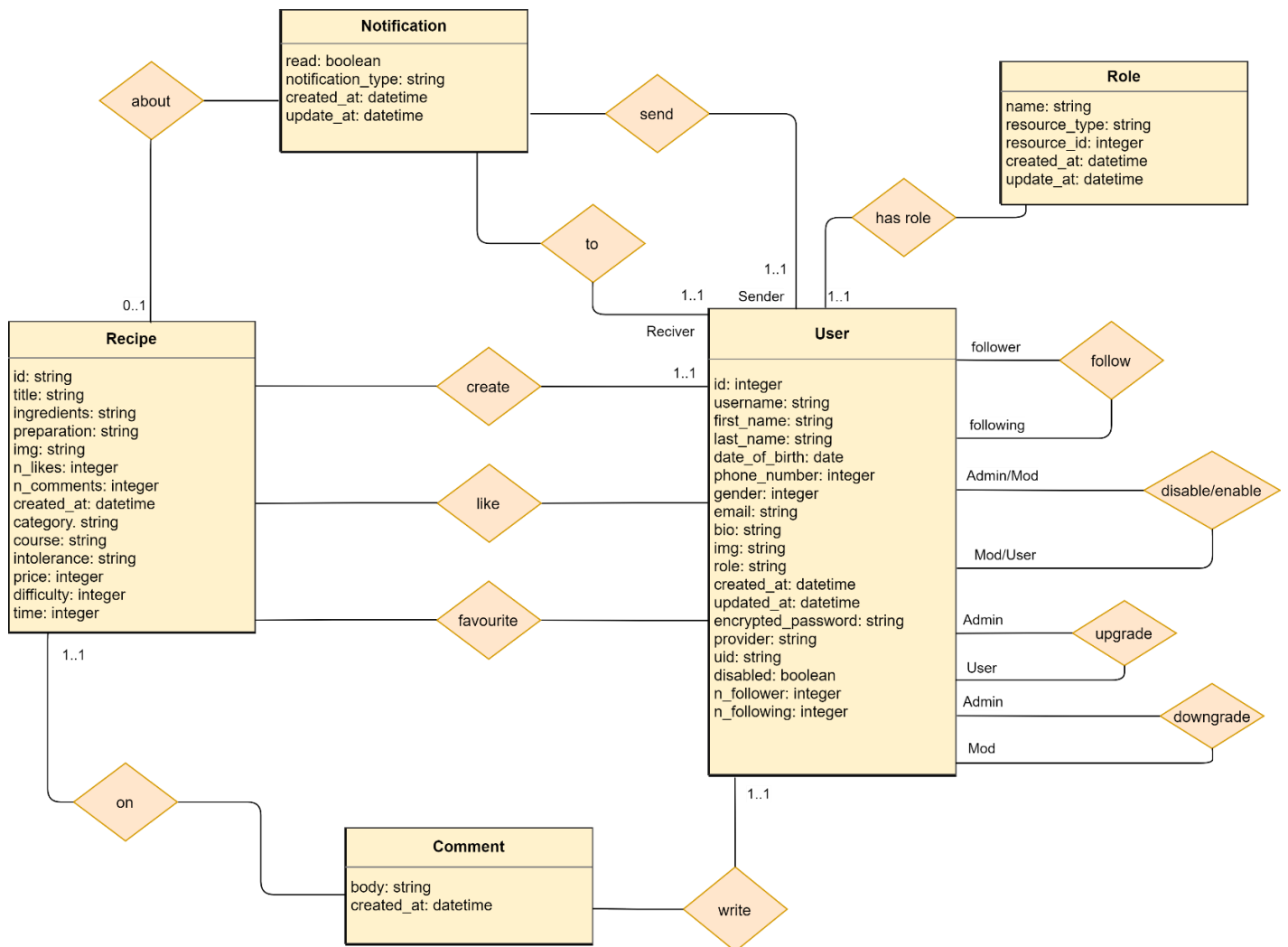
La seconda descrive le relazioni con il nome, una loro descrizione informale e l'elenco delle entità coinvolte, oltre all'identificatore.

La terza descrive gli attributi, specificando l'entità o la relazione a cui si riferiscono, il loro tipo e una breve descrizione.

La quarta tabella chiarisce le ragioni per cui i vincoli di cardinalità sulle relazioni sono stati definiti così come nello schema seguente.

Infine segue una lista di *business rules*, o vincoli esterni, ovvero le specifiche della realtà modellata che non possono essere espressi con il modello E-R. Oltre alle regole che descrivono vincoli non espressi dallo schema risulta a volte utile specificare più approfonditamente anche regole che documentano vincoli già espressi nello schema.

Schema E-R



4.2.1 Tabella delle entità

Entità	Descrizione	Attributi	Identificatore ²
User	Utente registrato a FoodFeed	id, username, nome, cognome, data di nascita, numero di telefono, sesso, email, bio, immagine del profilo, ruolo, password, disabilitato, numero di followers, numero di following	{username}
Recipe	Ricetta postata da un utente	id, titolo, ingredienti, preparazione, immagine, portata, dieta, intolleranza, numero di like, numero di commenti, prezzo, difficoltà, tempo di preparazione, creata alle	{id}
Comment	Commento di un utente ad una ricetta	id, contenuto, creato alle	{id}
Notification	Notifica di un evento generata da un utente su una risorsa e segnalata ad un altro utente	id, letta, tipo di risorsa, creata alle	{id}
Role	Ruolo di un utente su una risorsa	id, tipo, risorsa	{id}

² Per quanto riguarda l'identificatore, nell'implementazione del progetto abbiamo seguito la convenzione di Rails di identificare ogni entità con un id, indipendentemente da quanto dichiarato nello schema E-R.

4.2.2 Tabella delle relazioni

Relazione	Componenti	Descrizione	Identificatore
has role	User, Role	Un utente ha un ruolo su determinate risorse	{User, Role}
follow	User, User	Un utente può seguire ed essere seguito da altri utenti	{User, User}
disable	Admin/Mod, Mod/User	Un utente con ruolo più elevato di un altro può disabilitargli l'accesso	{Admin/Mod, Mod/User}
enable	Admin/Mod, Mod/User	Un utente con ruolo più elevato di un altro può riabilitargli l'accesso	{Admin/Mod, Mod/User}
upgrade	Admin, User	L'admin può promuovere un utente semplice a moderatore	{Admin, User}
downgrade	Admin, Mod	L'admin può retrocedere ad utente semplice un moderatore	{Admin, Mod}
write	User, Comment	Un utente può commentare una ricetta	{Comment}
on	Comment, Recipe	Un commento si riferisce a una specifica ricetta	{Comment}
create	User, Recipe	Un utente può postare una ricetta	{Recipe}
like	User, Recipe	Un utente può mettere like ad una ricetta	{User, Recipe}
unlike	User, Recipe	Un utente può togliere like ad una ricetta	{User, Recipe}
favourite	User, Recipe	Un utente può salvare una ricetta tra i preferiti	{User, Recipe}
send	User, Notification	Un utente genera notifiche con le sue azioni nel sito	{Notification}
to	Notification, User	Una notifica viene generata per un utente	{Notification}
about	Notification, Recipe	Una notifica può riferirsi ad una ricetta	{Notification}

4.2.3 Tabella degli attributi

Attributi	Entità/Relazione	Descrizione	Tipo
username	User	Username unico all'interno del sito	String
nome	User		String
cognome	User		String
data di nascita	User		String
numero di telefono	User		Integer
sex	User	0 per uomo, 1 per donna	Integer
email	User		String
bio	User		String
immagine del profilo	User		BLOB
ruolo	User	'U', 'M' o 'A'	String
password	User	Criptata	String
disabilitato	User	Utente bannato o meno	Boolean
numero di follower	User		Integer
numero di following	User		Integer
titolo	Recipe		String
ingredienti	Recipe		String
preparazione	Recipe		String
immagine	Recipe		BLOB

portata	Recipe	Antipasto, primo, secondo...	String
dieta	Recipe	Vegana, vegetariana, fruttariana...	String
intolleranza	Recipe	Glutine, latticini...	String
numero di like	Recipe		Integer
numero di commenti	Recipe		Integer
prezzo	Recipe	Su una scala da 1 a 5	Integer
difficoltà	Recipe	Su una scala da 1 a 5	Integer
tempo di preparazione	Recipe		Integer
creata alle	Recipe	Data creazione ricetta	Datetime
contenuto	Comment	Il testo del commento	String
creato alle	Comment	Data creazione commento	Datetime
letta	Notification		Boolean
tipo di risorsa	Notification	Ricetta o utente	String
creato alle	Notification	Data creazione notifica	Datetime
tipo	Role	Admin, Mod o Utente	String
risorsa	Role	Ricetta, commento o utente	Integer

4.2.4 Tabella delle cardinalità

Relazione	Ruolo	Cardinalità	Spiegazione
has role	User	1..1	Un ruolo riguarda un solo utente
	Role	0..*	Un utente può avere infiniti ruoli, tanti quanti i suoi commenti o le sue ricette
follow	User	0..*	Un utente può seguire quanti utenti vuole
	User	0..*	Un utente può essere seguito da quanti utenti vuole
disable	Mod/Admin	0..*	
	User	0..*	
enable	Mod/Admin	0..*	
	User	0..*	
upgrade	Admin	0..*	
	User	0..*	
downgrade	Admin	0..*	
	Mod	0..*	
write	User	1..1	Un commento è scritto da un solo utente
	Comment	0..*	
on	Comment	0..*	
	Recipe	1..1	Un commento è scritto riguardo una ricetta
create	User	1..1	Una ricetta è scritta da un utente
	Recipe	0..*	
like	User	0..*	
	Recipe	0..*	

unlike	User	0..*	
	Recipe	0..*	
favourite	User	0..*	
	Recipe	0..*	
send	User	1..1	Una notifica è generata da un utente
	Notification	0..*	
to	Notification	0..*	
	User	1..1	Una notifica viene ricevuta da un utente
about	Notification	0..*	
	Recipe	0..1	Una notifica può riguardare una ricetta oppure no

4.2.5 Vincoli esterni

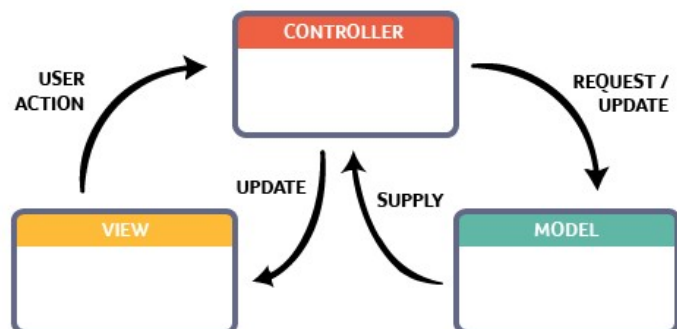
- L'attributo *ruolo* della classe **User** può valere 'A', 'M' o 'U';
- L'attributo *tipo* della classe **Role** può valere 'Comment', 'Recipe' o 'User';
- L'attributo *tipo di risorsa* della classe **Notification** può valere 'Commento', 'Like' o 'Utente';
- Se e solo se l'attributo *tipo di risorsa* della classe **Notification** è uguale a 'Utente', **Notification** partecipa con molteplicità 0 alla relazione *about*;
- Se uno **User** ha ruolo: 'A', partecipa alla relazione *has role* con tre **Role** che hanno tutti e tre i *tipo di risorsa* (Comment, Recipe e User) e *tipo*=admin;
- Se uno **User** ha ruolo: 'M', partecipa alla relazione *has role* con tre **Role** che hanno tutti e tre i *tipo di risorsa* (Comment, Recipe e User) e *tipo*=mod;
- Se uno **User** ha ruolo: 'U', partecipa alla relazione *has role* con **Role** che hanno *tipo di risorsa*='Comment' e 'Recipe', *tipo*=mod, e la *risorsa* corrispondente ai **Comment** e alle **Recipe** che lui stesso ha creato;
- Un admin può partecipare alle relazioni *disable* e *enable* sia con un moderatore che con un utente, mentre un moderatore può partecipare alle relazioni *disable* e *enable* solo con un utente;
- Una **Notification** viene creata solo a seguito della creazione di un **Comment** o di una relazione *like* o *follow*.

5. Progettazione del sistema

In questo capitolo verrà discussa l'organizzazione e l'architettura del sistema, basandoci su quanto ottenuto dal capitolo precedente. Vedremo come a ciascuna entità e ad alcune relazioni definite nel diagramma E-R corrisponde una tabella nel modello relazionale. Seguendo il pattern MVC, il modello ha il compito di effettuare le operazioni CRUD richieste dai controller che rispondono alle interazioni degli utenti con le viste. Approfondiamo nella prossima sezione questo pattern di design del software.

5.1 Il pattern MVC

Il noto approccio *Model-View-Controller* (o MVC) è un metodo di organizzazione del codice nei software interattivi, diventato ormai comune anche per la realizzazione di applicazioni web. Le applicazioni Rails prevedono l'utilizzo di questo approccio. Sostanzialmente viene divisa un'applicazione in tre parti: modello, viste e controller. Questo permette di raggiungere un'ottima separazione tra il codice che si occupa di gestire i dati e quello che li presenta all'utente.



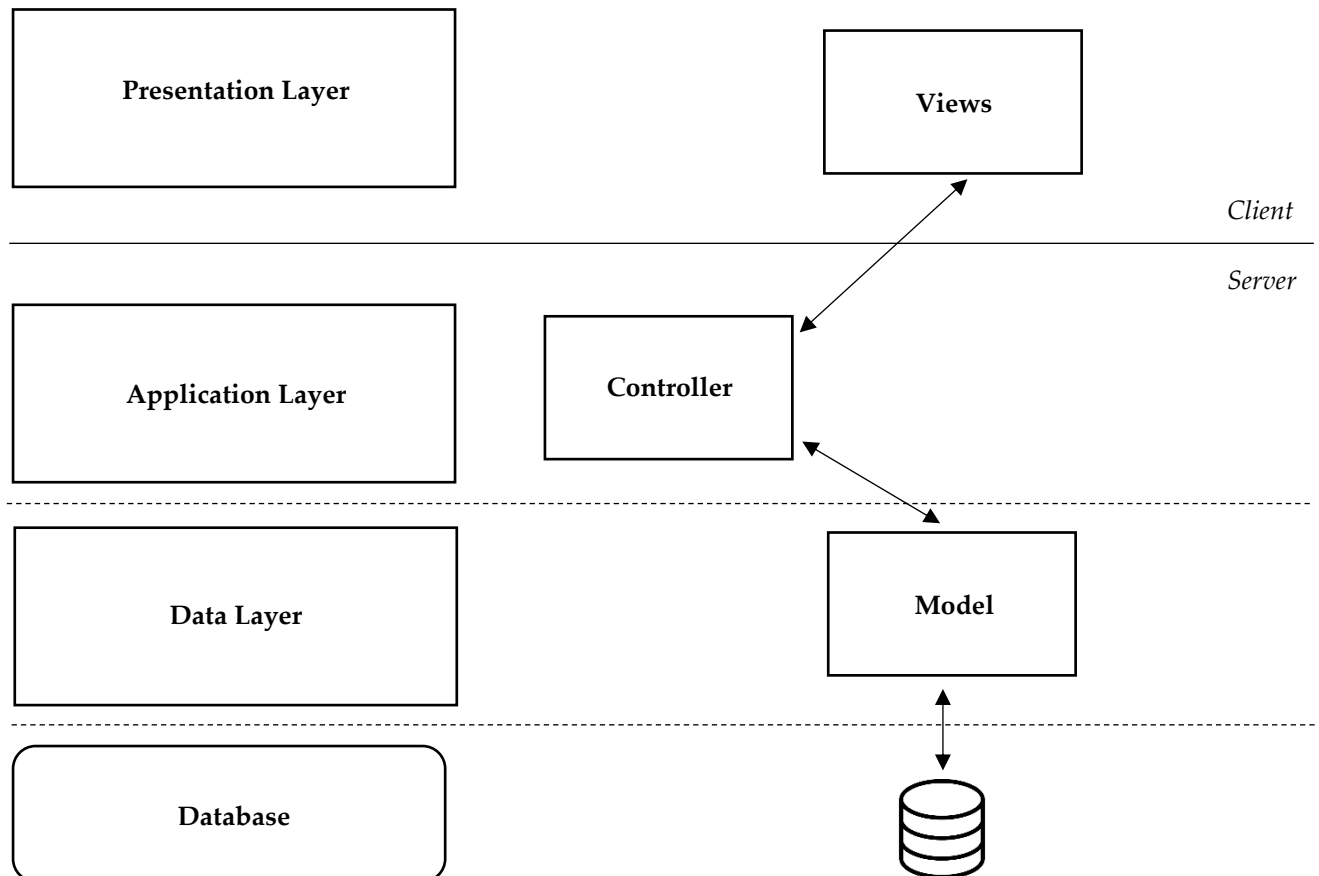
Il *model* gestisce direttamente la logica e le regole dell'applicazione. È il responsabile del mantenimento dello stato dell'applicazione, che viene salvato fuori dall'applicazione (spesso in un database). Il modello non rappresenta quindi i dati stessi, ma tutte raccoglie i metodi che possono essere applicate su questi dati; è completamente scollegato dal lato front-end e nessun altro nell'applicazione può rendere invalide le regole da lui dichiarate.

Il *controller* è il punto di incontro tra il modello e l'utente: esso riceve eventi dal mondo esterno (normalmente input dall'utente stesso), interagisce con il modello e visualizza la vista appropriata.

La *view* può essere una qualsiasi rappresentazione in output di informazioni, normalmente i dati del modello. Nel nostro caso queste informazioni sono raccolte in pagine web. Sebbene gli stessi dati si possono presentare all'utente in molti modi diversi a seconda delle richieste, una vista non gestisce mai i dati, ovvero il suo lavoro è soltanto quello di visualizzarli.

Ruby on Rails impone una struttura all'applicazione nella quale si sviluppano modelli, viste e controller come pezzi separati ma funzionali. Questi pezzi sono poi uniti al momento dell'esecuzione in un sistema organizzato come nella figura precedente. La divisione tra questi tre elementi principali può essere osservata anche visivamente, in quanto in seguito alla creazione di un nuovo progetto con Rails verranno automaticamente generate tre directory separate tra loro: `/app/models`, `/app/controllers`, `/app/views`. Inoltre Rails si occupa anche di gestire i dati nel database.

Nelle prossime sezioni analizzeremo i tre *tier* in cui l'applicazione si sviluppa, con esempi tratti da sezioni di codice che trattano la risorsa *Recipe* (ricetta).



5.2 Data Layer

Nel data layer sono contenute tutte le informazioni che gli altri livelli potranno gestire e presentare. Vogliamo mantenere le informazioni della nostra applicazione web all'interno di un database relazionale poiché offre un supporto completo e coerente. I database sono costruiti sulla base della teoria degli insiemi ed è per questo difficile collegarli ai concetti di programmazione ad oggetti: gli oggetti trattano dati e operazioni, mentre i database trattano insiemi di valori. Per collegare questi due aspetti Rails mette a disposizione le *migration*, che generano le tabelle nel cosiddetto *schema*. In questo modo possiamo gestire il database del progetto senza scrivere codice in SQL, ma solo in Ruby. Vediamo due esempi di migration, contenuti in `/db/migrate`.

`/db/migrate/20200807113401_create_recipes.rb`

```
class CreateRecipes < ActiveRecord::Migration[6.0]
  def change
    create_table :recipes do |t|
      t.references 'user'
      t.string :title, limit: 100, null: false
      t.string :preparation, limit: 5000, null: false
      t.string :img, default: "/default_recipe_img.png"
    end
  end
end
```

`/db/migrate/20200819154009_add_attributes_to_recipes.rb`

```
class AddAttributesToRecipes < ActiveRecord::Migration[6.0]
  def change
    add_column :recipes, :ingredients, :string, limit: 1000
    add_column :recipes, :course, :string, default: ''
    add_column :recipes, :intolerances, :string, default: ''
    add_column :recipes, :price, :integer, default: 1
    add_column :recipes, :difficulty, :integer, default: 1
    add_column :recipes, :time, :integer, default: 0
  end
end
```


Queste due migration riguardano il modello Recipe. La prima delle due richiede la creazione di una nuova tabella all'interno dello *schema*, chiamata Recipe e contenente gli attributi elencati, compreso un riferimento alla *foreign key* 'user', che deve essere un utente presente nella tabella User, riconosciuto tramite il suo identificatore *id*. Notiamo che non tutti gli attributi richiesti dallo schema E-R sono presenti nella prima migration: la seconda è stata generata proprio per aggiungere in un secondo momento quelli mancanti.

Ciascuna tabella del database ha un modello associato attraverso cui si possono effettuare le operazioni su dati. All'interno di ogni modello vengono specificati i vincoli sulla validità dei dati, attivati ogni volta che si accede in scrittura.

Vediamo cosa contiene la definizione del modello Recipe:

/app/models/recipe.rb

```
class Recipe < ActiveRecord::Base

  validates :image, presence: true
  validates :title, presence: true
  validates :preparazione, presence: true
  validates :ingredients, presence: true

  belongs_to :user
  has_many :likes
  has_many :comments
  has_many :favourites
  has_many :notifications

  has_one_attached :image

  validate :acceptable_image
  def acceptable_image
    //method body...
  end
end
```

In questo file vengono definiti i vincoli richiesti dalle nostre specifiche sugli oggetti di tipo *Recipe*. Tra questi controlli abbiamo la validazione della presenza degli attributi, l'appartenenza della ricetta a un determinato utente (allo stesso modo, in `/models/user.rb` è dichiarato che `has_many :recipes`) e la possibilità di partecipazione a relazioni come `like`, `comment`, `favourite`, `notification`. Il metodo che nel codice non è stato riportato verifica invece la correttezza del formato dell'immagine (PNG o JPEG) e stabilisce una dimensione massima del file in upload.

Ogni entità rilevante del progetto ha una definizione del suo modello simile a questo (`comment`, `favourite`, `follow`, `like`, `notification`, `recipe`, `role`, `user`), ed è stato definito nello *schema* con una o più migration come quella riportata in precedenza.

5.3 Application Layer

Nel pattern MVC l'application layer è gestito dai controller, che sono il centro logico dell'applicazione ed hanno il compito di coordinare l'interazione fra utente, viste e modello. Tuttavia Rails gestisce molte di queste interazioni dietro le quinte, lasciando concentrare il programmatore sul solo livello applicativo.

Un controller è una sottoclasse di `ApplicationController` e ha metodi come qualsiasi altra classe. Quando l'applicazione riceve una richiesta, l'Action Dispatcher di Rails determina quale controller e quale azione eseguire. Per fare ciò verifica come il programmatore abbia mappato tra loro azioni del controller e *URI* delle risorse. Alcune *routes* sono generate di default da Rails alla creazione di un modello, le altre sono definite in `/config/routes`. Tutte le routes dell'applicazioni sono visibili nella console eseguendo il comando `rails routes`.

Tra queste, alcune di quelle utilizzate riguardo le ricette sono:

<code>user_recipes</code>	<code>POST</code>	<code>/users/:user_id/recipes³</code>	<code>recipes#create</code>
<code>new_user_recipe</code>	<code>GET</code>	<code>/users/:user_id/recipes/new</code>	<code>recipes#new</code>
<code>edit_user_recipe</code>	<code>GET</code>	<code>/users/:user_id/recipes/:id/edit</code>	<code>recipes#edit</code>
<code>user_recipe</code>	<code>GET</code>	<code>/users/:user_id/recipes/:id</code>	<code>recipes#show</code>
	<code>PATCH</code>	<code>/users/:user_id/recipes/:id</code>	<code>recipes#update</code>
	<code>PUT</code>	<code>/users/:user_id/recipes/:id</code>	<code>recipes#update</code>
	<code>DELETE</code>	<code>/users/:user_id/recipes/:id</code>	<code>recipes#destroy</code>

³ Normalmente, l'URI creato di default sarebbe semplicemente `/recipes`, e similmente di seguito. Però, poiché nel modello abbiamo definito che la ricetta *belongs_to :user*, ogni ricetta appartiene ad uno specifico utente, che va specificato nell'URI.

Un utente, visitando il sito, effettua una richiesta con un metodo HTTP su uno degli URI presenti nelle routes (Es.: GET /users/1/recipe/2). A questo punto Rails controlla se la route richiesta dall'utente è presente e valida, e in caso positivo chiama l'azione del controller corrispondente (nel caso dell'esempio precedente, l'azione `show` del controller di `Recipe`).

Il controller di `Recipe` si trova in `/app/controllers/recipe_controller.rb` e contiene le *actions*, che sono i metodi in cui il programmatore definisce cosa debba essere eseguito lato server prima di restituire una risposta all'utente. All'interno delle azioni c'è quindi il cuore dell'intera applicazione. È qui dentro che innanzitutto vengono eseguiti i controlli sull'utente che fa la richiesta, e si verificano il suo ruolo e i suoi permessi, ritornando risultati diversi a seconda dell'esito di questa verifica. In questa parte di codice viene molto spesso interrogato il database, con *query* scritte in Ruby che vanno a restituire elementi dal database o ad aggiornarlo. Il controller è direttamente collegato alla view che poi andrà a restituire all'utente, da cui è sia in grado di recuperare dati (per esempio da un form compilato), sia di inviarli per essere visualizzati.

5.4 Presentation Layer

Nel layer precedente abbiamo visto come parte del lavoro di un controller è rispondere all'utente e questo può essere fatto in diversi modi. Quello più comune è di visualizzare un template. In termini dell'architettura MVC esso è una *view*, o vista, che prende le informazioni fornite dal controller e le usa per generare una risposta all'utente.

Se non specificato altrimenti, al termine di ogni azione in un controller, Rails cercherà di renderizzare al client un file con lo stesso nome dell'azione stessa: nel caso di `show#recipe` verrà cercato il file `show.html.erb` in `/app/views/recipe`.

Il formato `.erb` significa *embedded Ruby* e permette al programmatore di inserire parti di codice Ruby all'interno dell'html. Questo perché le pagine spesso non sono statiche ma devono mostrare informazioni diverse a seconda di chi la visualizza, o di quali informazioni ha richiesto. Per esempio, la pagina di informazioni `contacts` è statica e sempre uguale indipendentemente da chi la visualizza, mentre `discover` mostra informazioni diverse a seconda dei filtri che l'utente applica nella ricerca.

6. Validazione e Dispiegamento

6.1 Testing

La fase di testing in un progetto come questo è una fase molto importante, soprattutto considerando gli approcci *agile* utilizzati nello sviluppo del progetto.

Nel *Behavior-driven Development (BDD)* i test sono scritti in un linguaggio non tecnico che tutti possono comprendere, quindi forma un approccio per costruire una comprensione condivisa da sviluppatori e cliente su quale tipo di software costruire, discutendo degli esempi.

Nel *Test-driven Development (TDD)* i test vengono scritti prima del codice e il superamento dei test è la guida nello sviluppo.

Il processo di testing viene utilizzato per individuare le carenze di correttezza, completezza e affidabilità di un prodotto software in corso di sviluppo. Con tale attività si vuole quindi assicurare la qualità del prodotto tramite la ricerca di difetti, ovvero una sequenza di istruzioni e procedure che, quando eseguiti con particolari dati di input, generano dei malfunzionamenti.

6.1.1 Behavior-driven development

Il BDD consiste di una serie di passaggi da seguire ciclicamente:

- Identificare le funzionalità del software.
- Identificare gli scenari della funzionalità selezionata.
- Definire i passaggi per ogni scenario.
- Eseguire la funzionalità e fallire.
- Scrivere il codice per far passare i test.
- Eseguire funzionalità e passare i test.
- Generare i resoconti dei test.

Questi test sono normalmente scritti in un linguaggio di dominio (DSL) come ad esempio Gherkin, un linguaggio diffuso che viene utilizzato per scrivere storie utente narrative facendo uso di alcune parole chiave per definirne la struttura. Ogni parola chiave è tradotta in molte lingue parlate; nel nostro caso useremo l'inglese così come nel resto del progetto.

In ogni test vengono definiti i criteri di accettazione della funzionalità sotto forma di scenari. Uno scenario è formato da vari step a cui corrisponde il codice nel linguaggio in cui si sta sviluppando il sistema.

Senza andare a vedere l'intero insieme di test riportiamo un esempio pratico.

Decidiamo di voler testare la user story 8: *Pubblicare una ricetta*

COME utente VOGLIO pubblicare una ricetta PER condividerla con gli altri utenti

Nella cartella `/features` troviamo dei file con estensione `.feature`. Quello di nostro interesse in questo caso è `AddRecipe.feature`, il cui contenuto è:

```
Feature: User can add a recipe

Scenario: Add a recipe
    Given I am a registered user
    When I log in
    And I follow "Profile"
    And I press "Add new recipe"
    Then I should be on the Create New Recipe Page
    When I fill in "recipe[title]" with "Maccheroni"
    And I fill in "recipe[ingredients]" with "Ingredienti..."
    And I fill in "recipe[preparazione]" with "Preparazione..."
    And I attach the file "features/support/maccheroni.jpg" to "recipe[image]"
    And I press "Add Recipe"
    Then I should be on My Profile page
    And I should see "Maccheroni"
```

La *feature* ci dice quale funzionalità del sito stiamo testando, e possiamo definire diversi scenari al suo interno. Quello preso in esame elenca i vari *step* che portano alla pubblicazione di una nuova ricetta a partire dal login.

A seguito del comando `rails cucumber` la gemma Cucumber si occupa di fare il parsing delle stringhe presenti nel file di definizione di una feature e cerca, per ciascuno step, un'implementazione corrispondente dello stesso in codice in Ruby per eseguirlo. Nel caso non venga trovato lancia un errore richiedendo la sua implementazione, altrimenti esegue il codice.

In alcuni casi Cucumber riesce automaticamente ad interpretare gli step, per esempio le parole `follow` e `press` sono ricondotte alla pressione di un link o di un pulsante, come dichiarato nel file `web_steps.rb` in `/features/step_definitions`.

In altri casi bisogna definire manualmente il significato di alcune parole o espressioni. Per esempio abbiamo dovuto definire manualmente cosa sia un `registered user` o a quale pagina corrisponda `My Profile page`.

Sempre da `/features/step_definitions/web_steps.rb`:

```
Given /^I am a registered user$/ do
  @user= User.create(username: "user", email: "user@mail.it", password: "useruser",
    role: "U", first_name: "User", last_name: "User")
end

And /^I am on my profile page$/ do
  visit user_path(@user.id)
end
```

6.1.2 Test-driven development

Il TDD, lo sviluppo basato sui test, è un approccio allo sviluppo che prevede l'implementazione di un test prima della scrittura del codice sufficiente per eseguire tale test. Lo scopo di questa pratica è pensare alle esigenze della progettazione prima di scrivere il codice funzionale. Una volta scritto il codice dei test si verifica che questi falliscano: solo a questo punto si può procedere all'implementazione del codice funzionale. Si eseguono quindi nuovamente i test: se falliscono bisogna rivedere il codice; in caso contrario si procede alla scrittura di test per altre funzionalità.

Per gestire i test utilizziamo RSpec, un framework Rails che permette di scrivere diversi tipi di test tra i quali i test di unità sui singoli moduli che compongono il pattern MVC: ci saranno quindi test per le *views*, per i *controllers* e per i *models*. Ogni

singolo test in RSpec corrisponde ad una istruzione che verifica se il valore di una variabile o il risultato di una funzione corrisponde o meno al valore atteso.

Fatta una panoramica sullo sviluppo del software guidato dai test, concentriamoci ora su un esempio pratico tratto dal codice del progetto.

Uno dei modelli più importanti del progetto è il model User. Ciò che siamo andati a testare è la validità nella creazione di istanze di questo modello. Per fare ciò ci siamo aiutati con una gemma, *FactoryBot*, il cui scopo è proprio la creazione di un oggetto di default, in questo caso uno User, da inserire in un database di testing separato da quello principale. Questo utente ha tutti i campi corretti, e ci aspettiamo che venga inserito correttamente nel database. Tuttavia, in fase di progettazione sono stati posti alcuni requisiti che ogni utente deve rispettare per potersi iscrivere a FoodFeed: tra gli altri, la lunghezza minima della password, la lunghezza massima della bio, l'unicità dello username. Dobbiamo quindi testare la creazione di nuovi utenti che non rispettino queste regole, e assicurarci che l'app non accetti il loro inserimento nel database di testing.

Riportiamo di seguito il file che contiene l'utente 'Test', che ha tutti i requisiti minimi per essere valido, e il file che contiene i test RSpec:

`/spec/support/factories.rb`

```
FactoryBot.define do
  factory :user, class: User do
    username      {"Test"}
    email         {"test@mail.it"}
    password      {"testtest"}
    role          {"U"}
    first_name    {"Test"}
    last_name     {"Test"}
  end
end
```

`/spec/models/user_spec.rb`

```

require "rails_helper"

RSpec.describe User, :type => :model do

  before(:all) do
    @user1 = FactoryBot.create(:user)
  end

  after(:all) do
    @user1.destroy
  end

  describe "Creating a User" do

    it "is valid with valid attributes" do
      expect(@user1).to be_valid
    end

    it "has a unique username" do
      user2 = build(:user, email: "bob@gmail.com")
      expect(user2).to_not be_valid
    end

    it "is not valid if the username is too long" do
      user2 = build(:user, username: "u"*21)
      expect(user2).to_not be_valid
    end

    it "is not valid if the bio is too long" do
      user2 = build(:user, bio: "b"*600)
      expect(user2).to_not be_valid
    end

    it "has a unique email" do
      user2 = build(:user, username: "Bob")
      expect(user2).to_not be_valid
    end

    it "is not valid without a password" do
      user2 = build(:user, password: nil)
      expect(user2).to_not be_valid
    end

    it "is not valid if the password is too short" do
      user2 = build(:user, bio: "p"*7)
      expect(user2).to_not be_valid
    end

    it "is not valid without a username" do
      user2 = build(:user, username: nil)
      expect(user2).to_not be_valid
    end

    it "is not valid without an email" do
      user2 = build(:user, email: nil)
      expect(user2).to_not be_valid
    end

    it "is not valid if the profile image has a wrong extension" do
      user2 = build(:user, img: Rack::Test::UploadedFile.new('spec/support/factory_bot.rb'))
      expect(user2).to_not be_valid
    end

  end

end

```


Come atteso, al primo avvio di `rails spec` nel terminale, tutti questi test falliranno. L'obiettivo del programmatore ora è quello di trasformare l'esito di questi test da **10 examples, 10 failures FFFFFFFFFF** a **10 examples, 0 failures**

utilizzando tutte le armi a propria disposizione, ovvero modificando il model `User`, eseguendo delle *migration* nel database, cambiando parametri di `Devise`, etc.

6.2 Distribuzione

Una volta terminato un progetto è il momento di distribuirlo al pubblico.

FoodFeed è disponibile sia in un repository su GitHub, dove è presente il codice sorgente del progetto, sia su Heroku.

6.2.1 GitHub

Poiché il codice è stato costantemente aggiornato su GitHub in fase di lavorazione, è bastato pubblicare il repository contenente il codice definitivo per mettere a disposizione di tutti il codice sorgente dell'applicazione.



Basta seguire queste istruzioni per far girare FoodFeed su localhost:

Aprire il terminale in una directory a scelta ed eseguire

```
git clone https://github.com/GioPec/foodproject
```

In questo modo scarichiamo una cartella con tutti i file necessari. Spostandosi nella nuova directory dal terminale eseguiamo

```
bundle install
```

per far installare tutte le gemme contenute nel `Gemfile`.⁴

A questo punto lanciamo

```
rails db:migrate
```

```
rails db:seed
```

⁴ È importante segnalare che le versioni utilizzate sono state la 2.6.6 per Ruby e la 6.0.3.2 per Rails. Versioni diverse potrebbero comportare problemi di compatibilità.

Il primo comando dirà a Rails di eseguire le *migration*, il secondo popolerà il database con utenti di default, le cui credenziali sono:

- Utente: mail=user@mail.it, password=useruser
- Moderatore: mail=moderator@mail.it, password=moderator
- Admin: mail=admin@mail.it, password=adminadmin

Per far partire il server il comando è semplicemente

```
rails server
```

A questo punto è possibile aprire il proprio browser su `localhost:3000` e utilizzare il sito.

6.2.2 Heroku

Heroku è un servizio che permette l'hosting di una web app. È facilmente scalabile, ben integrato con GitHub, e offre soluzioni interessanti anche con il piano gratuito.



Prima di pubblicare l'app con questo servizio però, bisogna fare alcuni cambiamenti al progetto.

Innanzitutto Heroku utilizza Postgres per il proprio database, quindi bisogna installare una nuova gemma (`gem pg`) e utilizzarla al posto di `sqlite3`.

Poi vanno fatti alcuni cambiamenti riguardo lo storage dei file. Le foto pubblicate dagli utenti, che siano foto profilo o foto delle ricette, finora erano caricati in locale sul pc che fa girare l'app. Invece in questa situazione bisogna affidarsi a dei servizi esterni in quanto Heroku non mette a disposizione abbastanza spazio permanente per memorizzarle.

Sono stati quindi scelti i server di Amazon, offerti dal servizio AWS S3, che mette a disposizione dei *bucket* in cui conservare le immagini. Per collegare questi server al nostro progetto, cambiamo alcune linee nei file di configurazione:

In `config/environments/production.rb` settiamo
`config.active_storage.service = :amazon` anziché `:local`

In `config/storage.yml` settiamo

```
amazon:  
service: S3  
  region: eu-west-3  
  bucket: foodfeedproject-images
```

Infine inseriamo nel progetto la *key* e il *secret* forniti da Amazon per collegare i due servizi. A questo punto ogni volta che l'utente inserisce o visualizza una foto su FoodFeed viene contattato il database esterno per salvare o recuperare l'immagine.

Il sito è ora pronto per essere visualizzato all'indirizzo <https://foodfeedproject.herokuapp.com>.

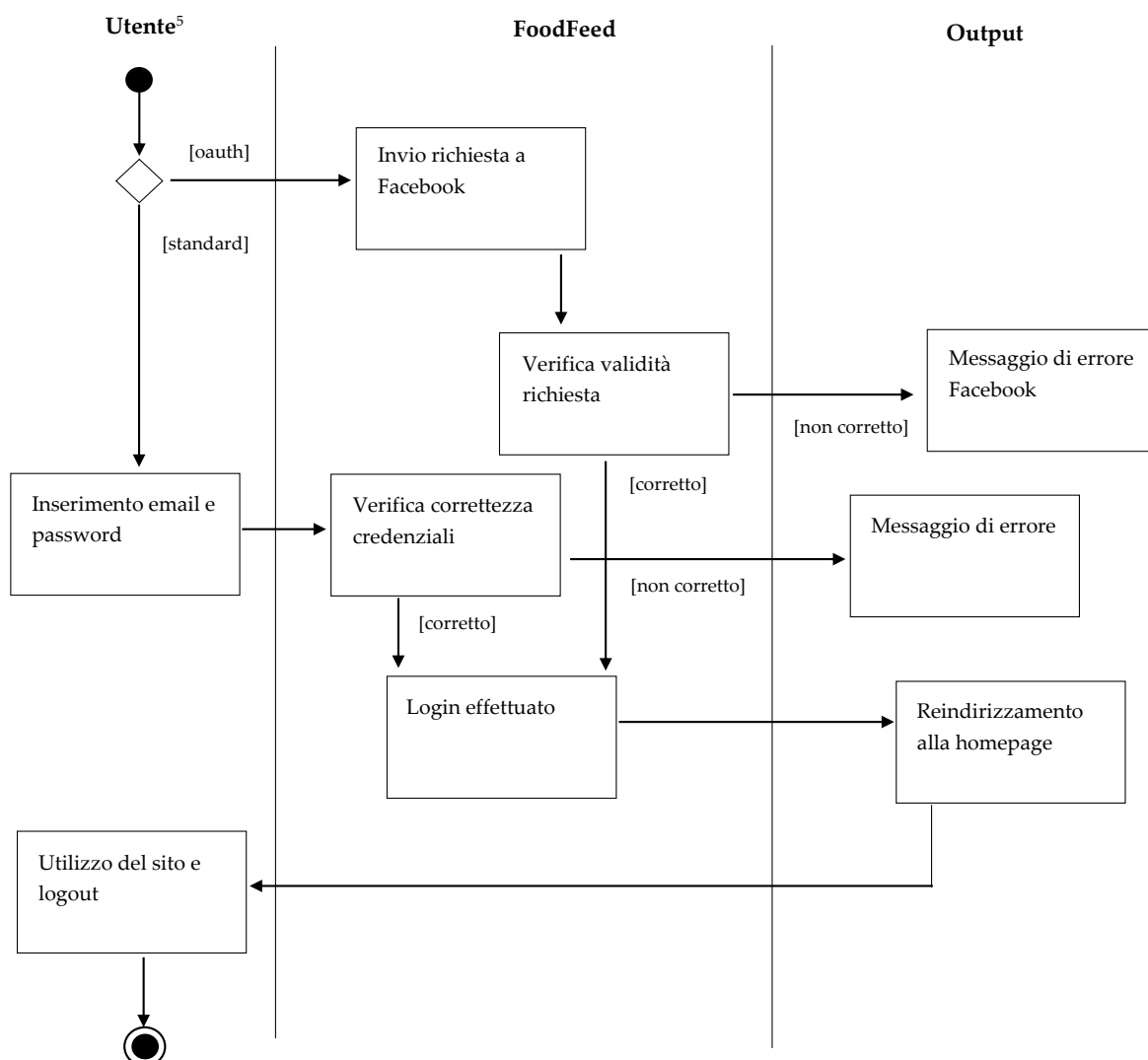
7. Analisi di un caso d'uso

Studiamo ora un esempio di caso d'uso di FoodFeed da parte di un utente che voglia postare una ricetta. Per fare ciò utilizziamo i diagrammi di attività (*activity diagram*), che permettono di descrivere un processo attraverso dei grafi in cui i nodi rappresentano le attività e gli archi l'ordine con cui vengono eseguite.

Prima di vedere le azioni che portano alla pubblicazione di una ricetta andiamo a rappresentare il funzionamento del login al sito, necessario prima di qualunque altra attività all'interno di FoodFeed.

Notiamo che il sistema o l'attore responsabile di una determinata attività è rappresentato in una colonna (*swimlane*) sotto cui si sviluppa una porzione dello schema.

Sul sito di FoodFeed le schermate che l'utente vede in questo processo sono quelle del login e la homepage, di cui riportiamo uno screenshot.





Log in

Email

Password

☐ Remember me

Log in

[Sign up](#)

[Forgot your password?](#)

[Sign in with Facebook](#)



[Discover](#) [Daily Recipe](#) [Top Recipes of the week](#) [Contact](#) [Show all Users](#)

Search users



Homepage



GiovanniPecorelli

1w



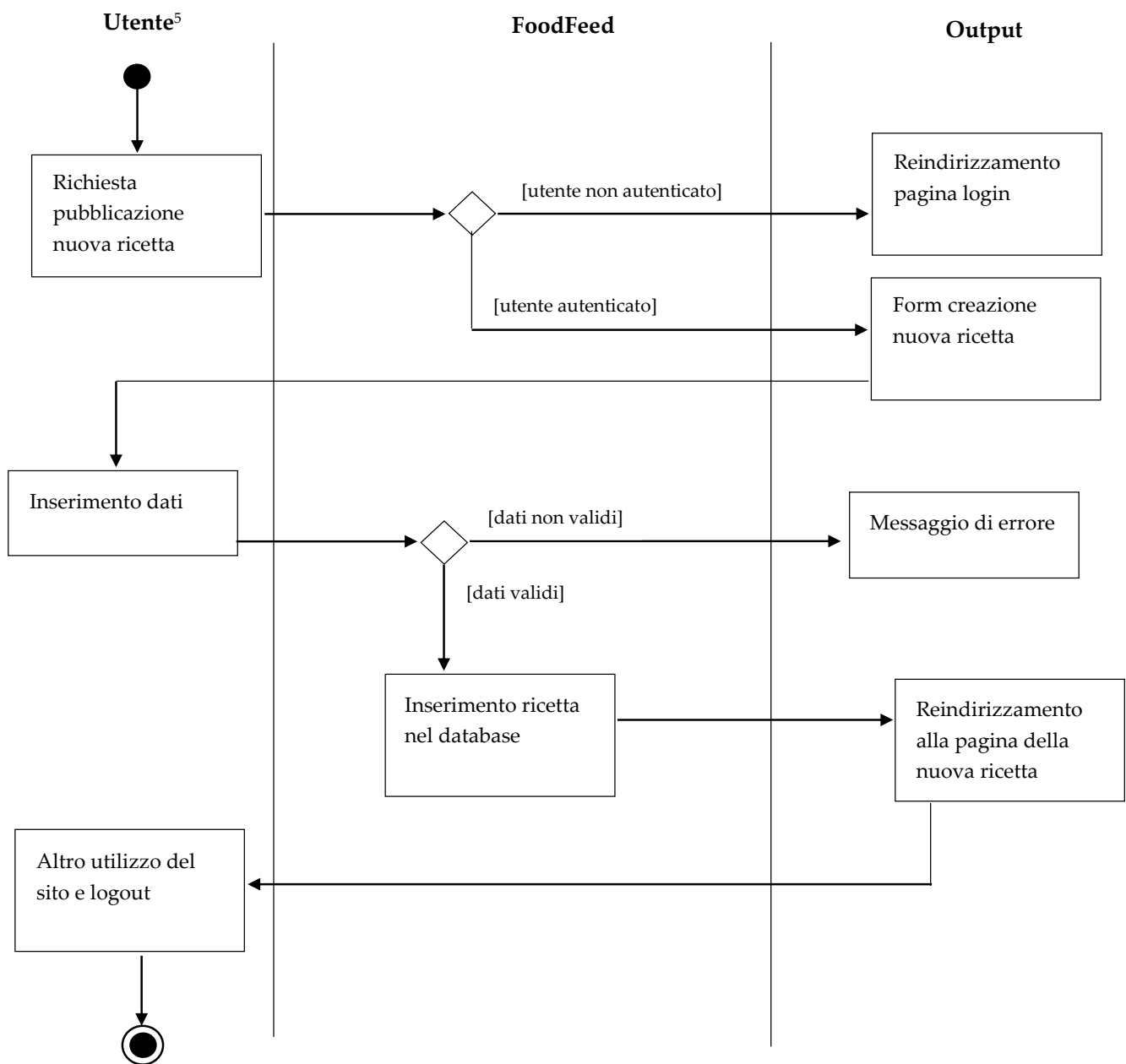
Tortino al cioccolato

2 ❤️ 1 💬


1 2 3




FoodFeed | Giovanni Pecorelli, Jacopo Rossi


Disegniamo ora il diagramma di attività per la pubblicazione di una ricetta.



⁵ In questo schema e nel precedente con Utente si intendono anche Mod e Admin.


[Discover](#)
[Daily Recipe](#)
[Top Recipes of the week](#)
[Contact](#)
[Show all Users](#)



GiovanniPecorelli (#9)

Followers: 4 Following: 4

Giovanni Pecorelli

Hey there! I'm using FoodFeed


[Add new recipe](#)

[Favourites](#)


[Edit your profile](#)

[Logout](#)


3 recipes:



Biscotti bustina di tè







Penne al tonno e pomodorini



Tortino al cioccolato

FoodFeed | Giovanni Pecorelli, Jacopo Rossi


[Discover](#)
[Daily Recipe](#)
[Top Recipes of the week](#)
[Contact](#)
[Show all Users](#)




Add a Recipe





INGREDIENTI PER 4 PERSONE --- 400 g di penne rigate - pomodorino 300 g di pomodorini pachino - tonno in scatola 280 g di tonno all'olio d'oliva - olive tagliasche 80 g di olive tagliasche - alici 3 filetti d' alici - scalogno 1 scalogno - olio q.b. olio extravergine d'oliva - peperoncino q.b. peperoncino in polvere - sale q.b. sale

Portiamo ad ebollizione
quando bolle saliamo e
tempo indicato sulla confezione
abbastanza grande, versare
e facciamo soffriggere u
scalogno e dorato unitam
Assaggiare i pomodori
quattro, insaporiamo co

Breakfast
Appetizer
Finger Food
First course
Soup
Pizza
Second course
Bread
Side course
Salad
Sauce
Snack
Drink
Dessert
Fruit

acqua per la pasta:
e le penne per il
mpo, in una padella
extravergine d'oliva
). Quando lo
acciamoli sciogliere.
ite lavati e tagliati in
eroncino in polvere e

Price    

Difficulty    

Time

FoodFeed | Giovanni Pecorelli, Jacopo Rossi



Penne al tonno e pomodorini

Edit post

Remove post



Course: First course | Price: \$ \$ | Difficulty: 🍳 🍳 | Time: 30 minutes

Ingredients

INGREDIENTI PER 4 PERSONE --- 400 g di penne rigate - pomodorino 300 g di pomodorini pachino - tonno in scatola 280 g di tonno all'olio d'oliva - olive taggiasche 80 g di olive taggiasche - alici 3 filetti d' alici - scalogno 1 scalogno - olio q.b. olio extravergine d'oliva - peperoncino q.b. peperoncino in polvere - sale q.b. sale

Preparation

Portiamo ad ebollizione, in una pentola, l'acqua per la pasta: quando bolle saliamo e mettiamo a cuocere le penne per il tempo indicato sulla confezione. Nel frattempo, in una padella abbastanza grande, versiamo un filo d'olio extravergine d'oliva e facciamo soffriggere uno scalogno tritato. Quando lo scalogno è dorato uniamo i filetti di alici e facciamo sciogliere. Aggiungiamo i pomodorini precedentemente lavati e tagliati in quattro, insaporiamo con un pizzico di peperoncino in polvere e facciamo cuocere 4-5 minuti. Uniamo ora il tonno ben scolato e sminuzzato e aggiungiamo un pizzico di sale, non troppo perché gli ingredienti sono già piuttosto saporiti. Infine arricchiamo il nostro sugo con le olive taggiasche, mescoliamo e facciamo cuocere ancora qualche minuto. Scoliamo la pasta al dente e trasferiamola nella padella: mescoliamo bene, aggiungiamo se necessario uno o due mestoli di acqua di cottura per amalgamare meglio. Serviamo subito aggiungendo a piacere una spolverata di prezzemolo tritato.

1w

♥ 1 likes

☆ Add to favourites

0 comments

What do you think of this recipe?

Post

Bibliografia

- Engineering Software as a Service: An Agile Approach Using Cloud Computing, *A. Fox, D. Patterson*
- Dispense del corso Basi di Dati, *M. Lenzerini*
- Dispense del corso Progettazione del Software, *G. De Giacomo*
- Dispense del corso Linguaggi e Tecnologie per il Web, *R. Rosati*
- rubygems.org
- guides.rubyonrails.org