Pfeeder Report
Applicazioni e servizi web

Martina Cavallucci - 919588
martina.cavallucci@studio.unibo.it

Nicolas Farabegoli - 979664
nicolas.farabegoli@studio.unibo.it

20 gennaio 2021

Indice

1 Introduzione

PFeeder è un servizio web che consente di rimanere connessi e monitorare il programma alimentare e le porzioni di cibo dei propri animali domestici.
Il sistema mira a offrire un servizio di monitoraggio dello stato di salute degli animali domestici permettendo di tener traccia delle calorie assunte e gestire le porzioni che il dispenser dovrà erogare; inoltre permette di controllare i costi che vengono sostenuti per il singolo animale offrendo una vista delle informazioni.

In particolare l'utente potrà aggiungere e modificare informazioni riguardanti: animali, mangimi, razioni; il tutto accedendo ad una propria area riservata.

2 Requisiti

2.1 Requisiti utente

Si ritiene che l'utente finale del nostro sistema sia un padrone di animali domestici, e si aspetta dal nostro sistema di:

- Poter accedere ad una sezione con accesso privato e poter gestire i propri animali;
- Possibilità di aggiungere un nuovo fornitore di cibo;
- Poter aggiungere informazioni circa i suoi animali;
- Poter configurare razioni con determinata cadenza e quantità;
- Poter visualizzare statistiche suoi propri animali;
- Ricevere notifiche in tempo reale sulle azioni che il distributore di cibo effettua;
- Visualizzare stato di connessione del proprio distributore del cibo (simulato);

2.2 Requisiti funzionali

- Registrazione di un nuovo utente;
- Registrazione del dispositivo acquistato dall'utente;
- Registrazione di uno o più animali domestici;
- Registrazione di prodotti alimentari (mangimi);
- Registrazione di programmi alimentari;
- Generazione di statistica sul singolo animale in termini di calorie assunte e costi sostenuti;
- Generazione di notifiche e/o avvisi sullo stato dell'erogatore; ad esempio pasto completato, mancanza di mangime, errore di erogazione.

2.3 Requisiti non funzionali

- Rendere il sistema facile da utilizzare anche ad utenti non esperti;
- Rendere il sistema sicuro mediante meccanismi di autenticazione e autorizzazione;
- Rendere il sistema estendibile a nuove funzionalità;
- Rendere il sistema reattivo.

3 Design e architettura

3.1 Design

Il modello utilizzato è di tipo iterativo e basato su **User Centered Design** con utenti virtualizzati; in particolare sono stati individuati gli utenti target dell'applicazione e su di essi si sono sviluppate le *Personas* che hanno contribuito a sviluppare e comprendere le funzionalità del sistema e come possano esse rispondere ai requisiti utente.

Il team ha gestito tutto il progetto tramite *card* su **Trello** che ha permesso di organizzare lo sviluppo in vari sprint; in ognuno dei quali si individuavano le funzionalità da implementare e da testare. Il team ha seguito quindi un framework ispirato a quello AGILE; che ha permesso in modo incrementale di sviluppare le funzionalità in base alla priorità.

Nello svolgimento del progetto e del design delle interfacce si è sempre cercato di rispettare il principio di **KISS** e **DRY** per poter ottenere codice di qualità e risolvere le funzionalità nel modo più semplice possibile; sopratutto nello sviluppo delle interfacce si è sempre tenuto in considerazione l'usabilità di esse.

Si sono utilizzati i principi della **Responsive Design**, alla base per ottenere un buon grado di accessibilità.

3.2 Architettura sistema

L'architettura del sistema aderisce al modello REST (*Representional State Transfer*), perciò si sono identificate le entità che caratterizzano il dominio applicativo.

Di seguito vengono riportate le entità:

- User: modella l'utente finale del sistema nonché il padrone di animali domestici;
- Pet: modella l'animale con le sue caratteristiche proprie;
- Fodder: modella il mangime che può essere erogato;
- Ration: modella una singola razione che viene programmata dal padrone ad un certo orario per un determinato animale;
- Notification: modella qualsiasi notifica che il padrone riceve dal sistema;
- Feed: modella la singola erogazione di mangime effettuata dal dispenser.

Di seguito vengono mostrate in figura 1 le relazioni tra le varie entità. In particolare nel sistema esistono le seguenti relazioni:

- Un utente può registrare nel sistema uno o più animali;
- Un utente nel corso della sua esperienza con il sistema può ricevere una o più notifiche. Esse possono essere di tipo informativo (l'animale ha completato il pasto), di tipo errore (il dispenser ha esaurito il cibo da erogare), di tipo avvertimento (il dispenser sta per terminare il cibo da erogare);
- Un animale può essere di 3 tipi principali: cane, gatto, altro;
- Ad un animale può essere assegnata una o più razioni ad un determinato orario scelto dall'utente;
- Un animale può effettuare uno o più pasti;
- Ogni pasto è associato ad un mangime;
- Un animale può avere un mangime corrente che può cambiare durante il tempo; ciò determina un cambiamento di costi legati all'acquisto del mangime e delle calorie assunte ad ogni pasto.

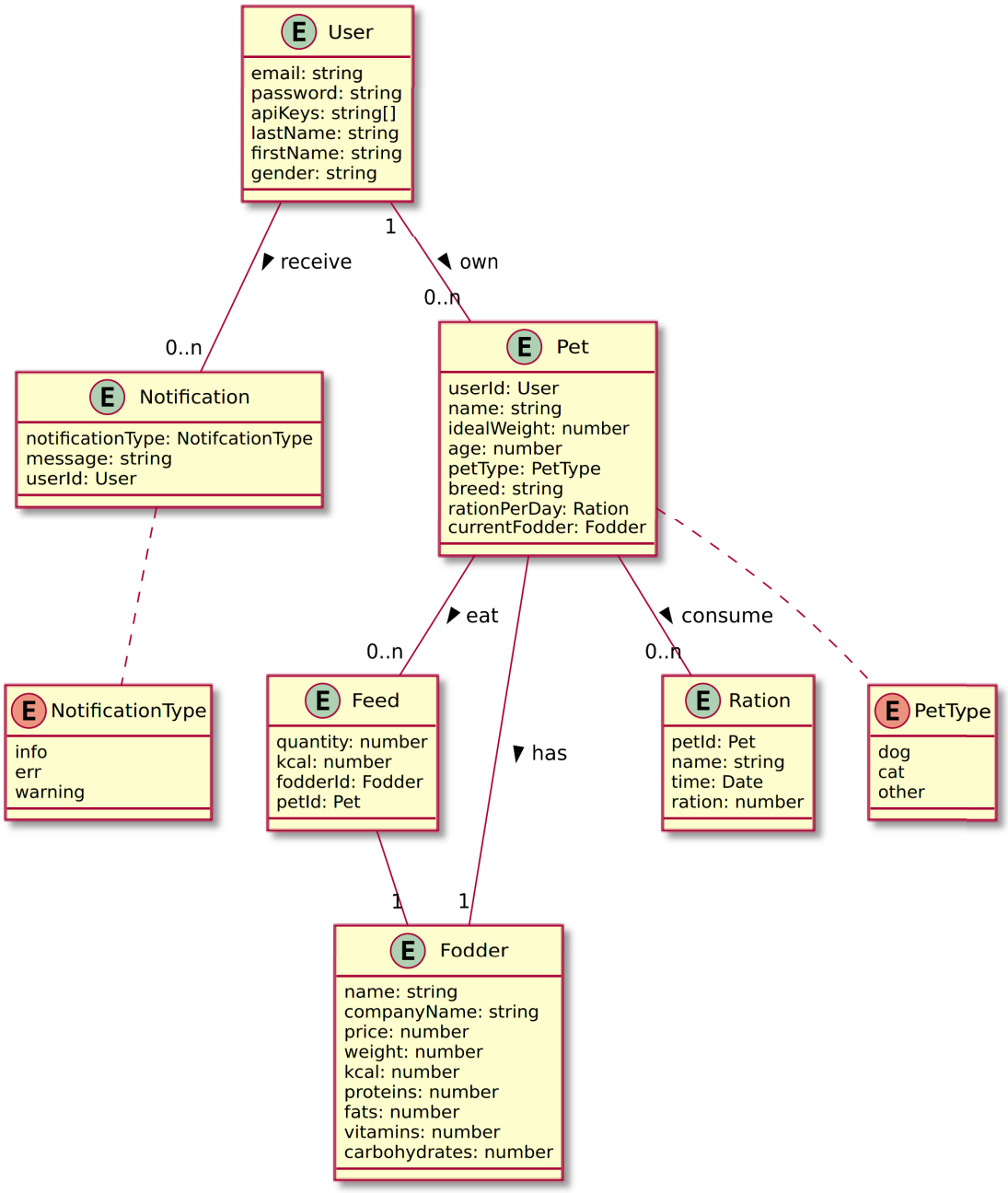

Figura 1: Diagramma delle relazioni tra le entità

3.3 Architettura interfacce utente - Mockup

Di seguito si riportano i principali mock up disegnati in fase di analisi dei requisiti del progetto; essi rappresentano una prima versione e semplice di ciò che ci aspettavamo di ottenere dal sistema.

Mock up pagina iniziale

Si prevede una pagina iniziale con alcune frasi di indicazione che mirano a dare una panoramica sui servizi che offre il sistema.

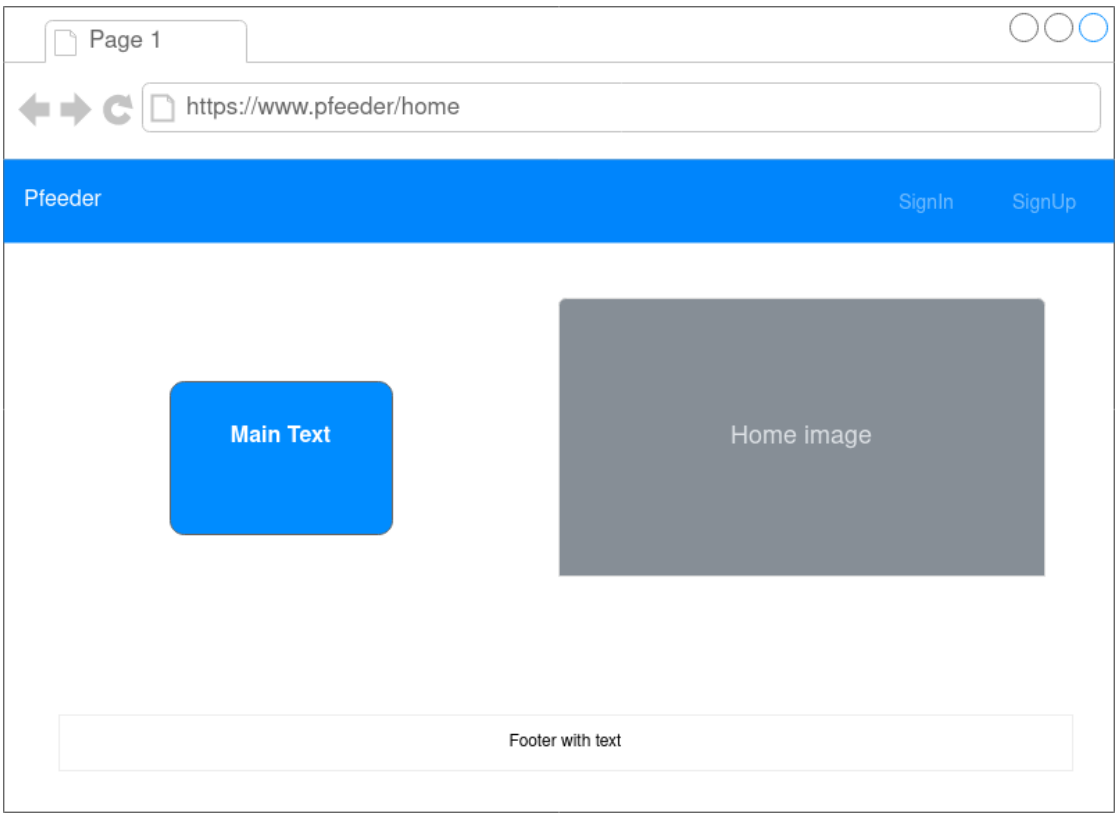

Figura 2: Mock up dell'home page

Mock up dashboard

Si prevede una pagina di dashboard dove l'utente finale potrà tenere sotto controllo le informazioni circa: razioni, animali, costi, calorie assunte, stato del dispositivo dispenser.

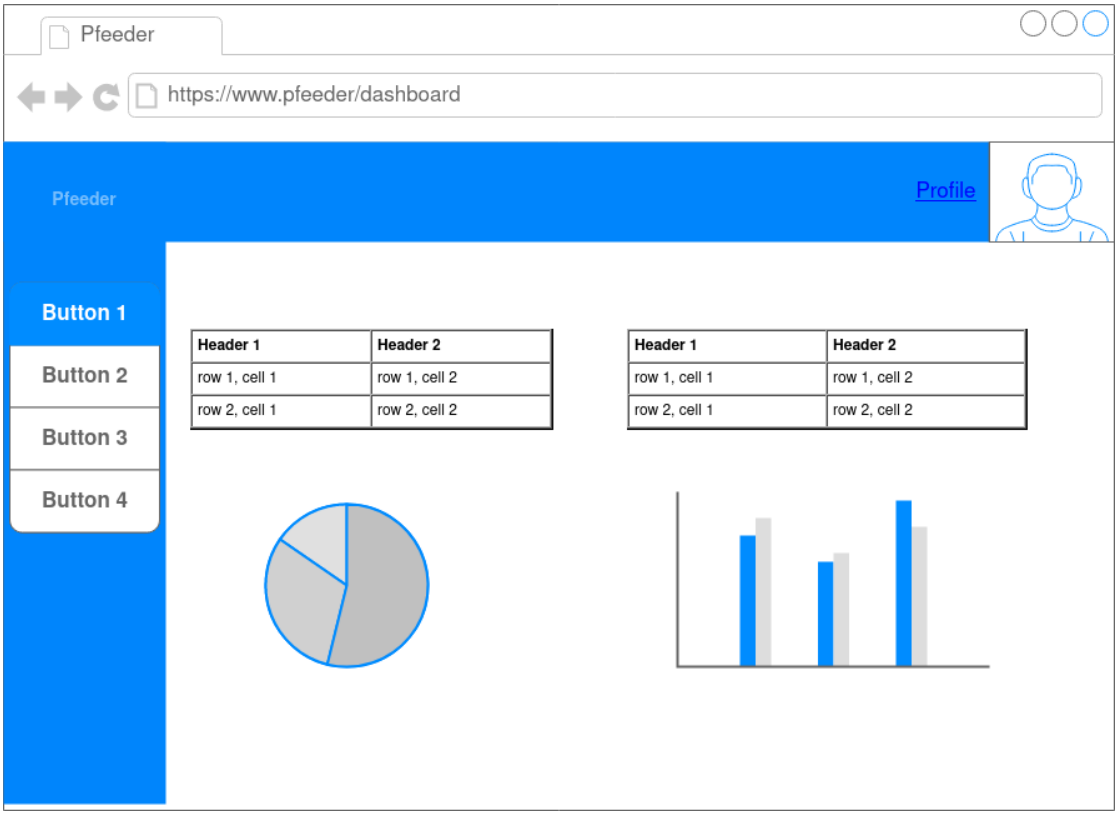

Figura 3: Mock up della dashoboard utente

Mock up login e registrazione

Si prevedono pagine di registazione e accesso al sistema.

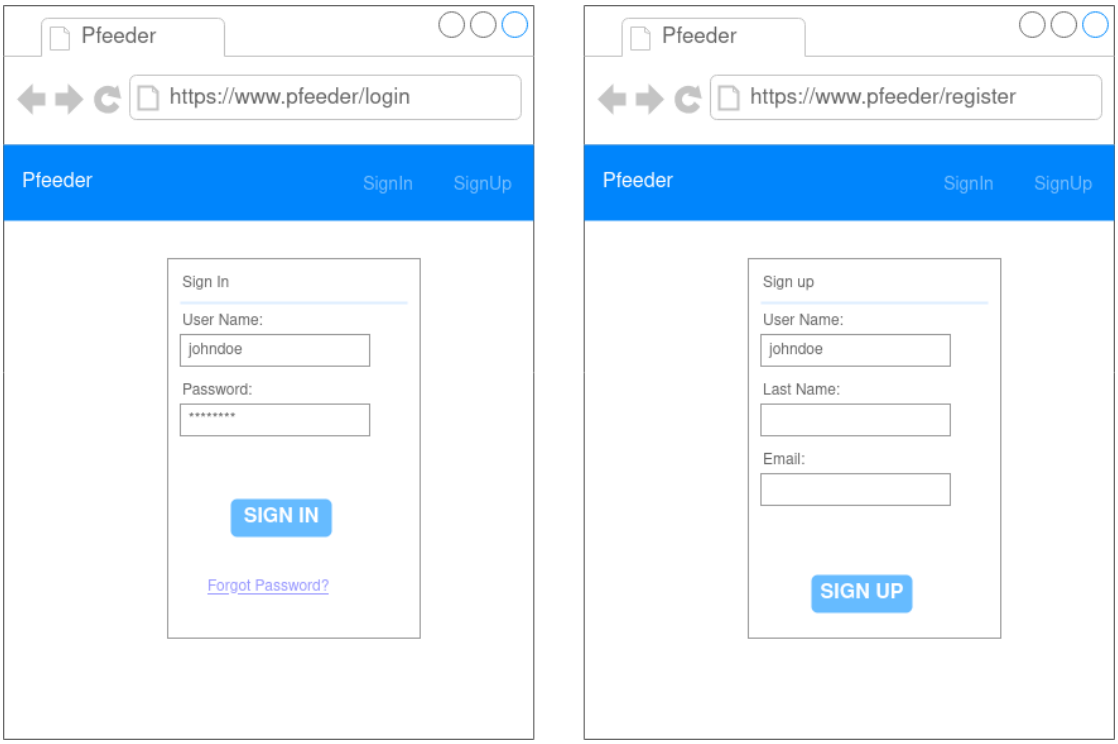

Figura 4: Mock up login e registrazione

Mock up aggiunta/modifica animale

Si prevedono pagine di aggiunta e modifica di animali, mangimi, razioni.

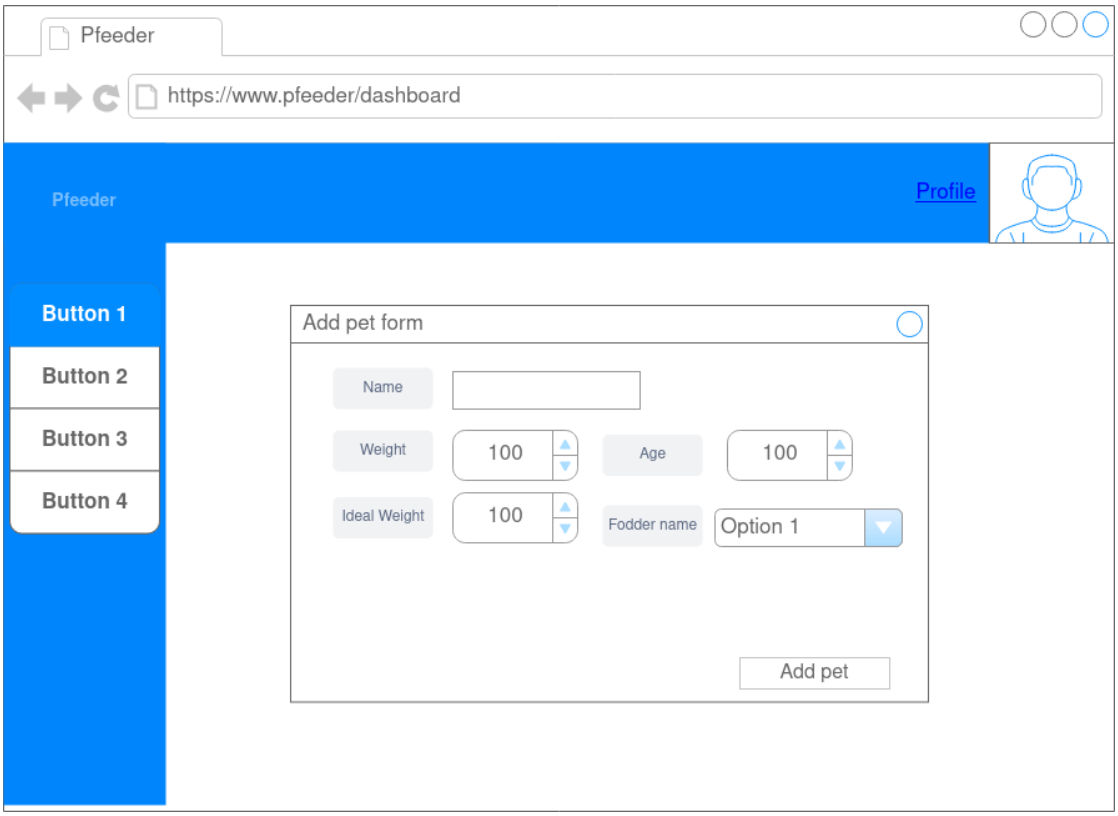

Figura 5: Mock up login e registrazione

3.4 Design di dettaglio

Backend

In figura 6 viene mostrato il *component diagram* che descrive come è stato strutturato il backend.

Nello dettaglio abbiamo i `controlles` nonché classi preposte a fornire, mediante la specifica **OpenAPI**, le API per interagire con il backend. Ogni entità descritta alla sezione precedente ha un proprio controller, il quale espone le operazioni *CRUD* (Create Read Update Delete) mediante web API.

I controllers a loro volta sfruttano i repository: questi non sono altro che classi preposte ad interagire con il DB offrendo diverse funzionalità quali creazione, modifica, cancellazione, ecc. Anche in questo caso si è definito un repository per ogni tipo di entità del sistema. Tali entità sono descritte dalle classi di `Models`, le quali definiscono le proprietà che tali entità hanno.

Infine il backend è provvisto di classi che esprimono servizi: nello specifico abbiamo un servizio che si occupa di gestire le `WebSocket` per le notifiche in real time e un servizio che si occupa di gestire il protocollo `MQTT` per l'iterazione con i dispenser.

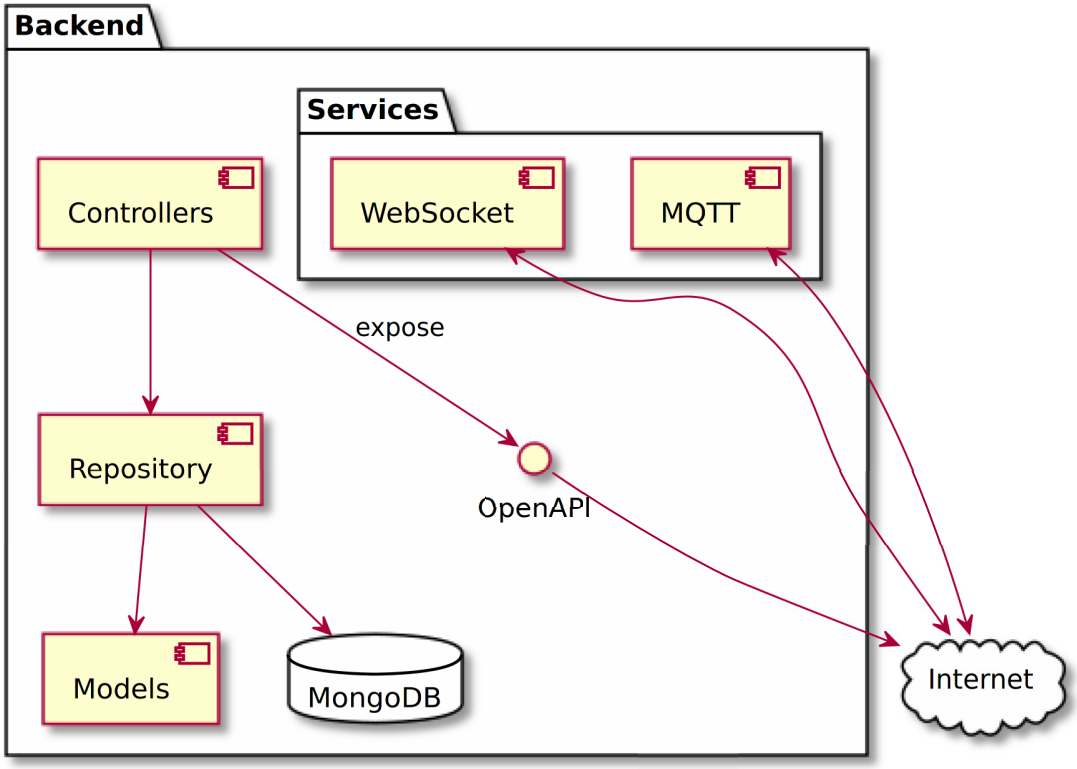

Figura 6: Diagramma package dei componenti del backend

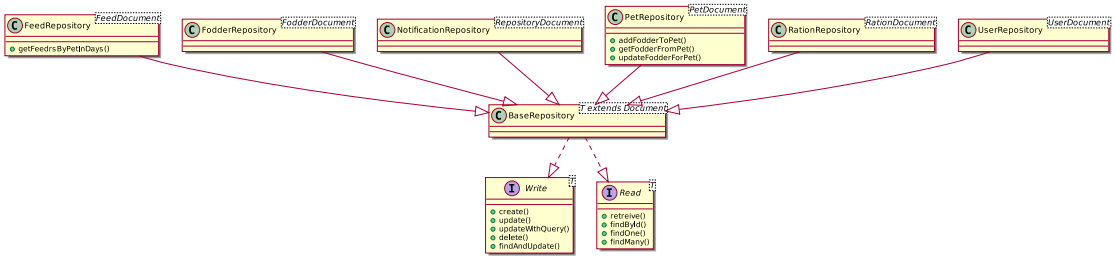

Figura 7: Diagramma UML rappresentante le classi utilizzate per implementare il repository pattern.

3.4.1 Design pattern utilizzati

Durante lo sviluppo del backend si sono identificati alcuni schemi ricorrenti che sono stati implementati mediante design pattern noti. Di seguito vengono illustrati i principali e più rilevanti pattern adottati.

Repository Pattern: in figura 7 è mostrato il diagramma UML del repository pattern. Nel dettaglio sono state definite due interfacce: `Read` e `Write`, queste modellano le operazioni di lettura e scrittura più comuni che vengono effettuate in un DB. Mediante la classe `BaseRepository` vengono implementate le due interfacce per avere un entità unica su cui fare le varie operazioni. Questa classe è generica sul tipo di documento di MongoDB, ciò significa che estendendo la classe e reificando il generico con la classe effettiva che rappresenta il documento di MongoDB possiamo effettuare le operazioni appena descritte direttamente sulla collezione senza dover riscrivere lo stesso identico codice ma per collezioni differenti. Utilizzando questo pattern otteniamo due vantaggi evidenti: il primo riguarda l'estensibilità del sistema, infatti, estendendo la classe base, è possibile definire nuovi repository. Il secondo è conseguenza del primo poiché qualora fosse necessario aggiungere un nuovo repository il codice da implementare è praticamente nullo: occorre solo estendere la classe base e istanziare il generico.

Dependency Injection: mediante la libreria `typedi` è stato possibile utilizzare il DI pattern. Mediante un annotazione `@Service` su una classe si definisce che quella classe dovrà essere "iniettata" in tutte quelle classi che ne fanno uso. Questo pattern semplifica notevolmente il tedioso lavoro di capire quando e come un oggetto deve essere creato all'interno di una classe che poi lo utilizzerà. Infine questo pattern riduce l'accoppiamento tra classi semplificando quindi lo sviluppo delle classi stesse.

3.4.2 Sicurezza

La sicurezza è un aspetto fondamentale per le applicazioni web oggigiorno. Dal momento che nell'applicazione vengono memorizzati dati sensibili quali: indirizzo email, password, dati anagrafici, ecc. occorre che tali dati siano protetti. Altro aspetto da considerare (oltre alla sicurezza) è l'autorizzazione nell'effettuare determinate operazioni. A tal proposito si è fatto uso di *Json Web Token* (JWT) per poter accedere alle API del sistema. Questo approccio semplifica notevolmente il processo di autenticazione di un

utente: se prendiamo per esempio meccanismi più tradizionali di autenticazione basati su sessione è facile individuare problematiche ad essi associati; per gestire le sessioni occorre registrare (sul backend) strutture che memorizzano cookies e strutture ad essi associati. La prima problematica è subito chiara poiché è necessario proteggere queste strutture in quanto potrebbero essere usate per rubare identità dal sistema. Un altra problematica è derivata dalla scalabilità di questo approccio (approccio basato su sessione): sistemi distribuiti sono alla base di architetture e sistemi scalabili, supponiamo di avere n repliche del nostro sistema sparse in vari cluster, il sistema deve gestire la sessione in modo trasparente al client, ovvero se un client si collega a un determinato nodo della rete e per motivi di load balancing, per esempio, il traffico viene ridirezionato su un altra istanza, la sessione deve permanere identica. Questo risulta essere un problema in quanto ci deve essere condivisione dello stato tra le varie istanze e questa procedura non sempre viene effettuata istantaneamente ma occorre del tempo affinché i nodi del cluster si allineino.

Utilizzando JWT come metodo di autenticazione si sopperisce a questo problema: questo meccanismo si basa sulla condivisione di un token da parte del server verso il client una sola volta al login. Dopo l'operazione di login, l'utente ha il compito di "allegare" il token che gli è stato assegnato ad ogni richiesta che fa al server, questo gli consente di identificarsi al backend e quest'ultimo può concedere (oppure negare) l'operazione richiesta. L'algoritmo che valida il token si basa su una codifica hash, quindi non occorre memorizzare una sessione per ogni client ma basta solo verificare che il token sia valido. Questo meccanismo sopperisce al problema della replicazione della sessione nei sistemi distribuiti poiché la decodifica si basa su un `SECRET` ovvero una stringa utilizzata come "seme" di generazione dei token. Se ogni istanza (in fase di creazione) riceve lo stesso token, essendo l'algoritmo di codifica/decodifica noto, ogni istanza è in grado di determinare se il token è valido oppure no senza condividere alcun dato. L'unica accortezza lato backend che deve essere presa è rendere non accessibile il `SECRET`. Anche lato frontend va presa qualche accortezza: il token non deve essere condiviso il alcun modo poiché se un attaccante si impossessa del token (anche senza sapere username e password) è in grado di fare operazioni sul sistema spacciandosi per un altro utente.

Per rendere il sistema scalabile si è adottato quest'ultimo approccio. Tutte le API del backend che richiedono un' interazione con il sistema da parte di un utente autorizzato sono protette mediante autenticazione basata su JWT.

Frontend

In figura 8 viene mostrata la struttura del front-end sviluppato in Angular; in particolare si è gestita l'applicazione in 3 sotto-package per separare al meglio le funzionalità di ognuno di essi:

- **Service**
  - **Auth**: contiene l'implementazione del servizio che grazie a chiamate POST, GET, PATCH interagisce con il server e si occupa della gestione delle chiamate per l'accesso da parte dell'utente;
  - **Data**: contiene l'implementazione del servizio che permette di reperire informazioni sulle entità del sistema, inoltre di aggiungerle e modificarle;
  - **Notification**: contiene l'implementazione dei servizi di gestione websocket e di gestione delle varie tipologie di notifiche che l'utente può visualizzare.
- **Model**: contiene la definizione di tutte le entità del sistema, tali devono essere conformi tra client e server.
- **Component**
  - **Fodder**: contiene tutti i componenti utili alla gestione dei mangimi: aggiunta/modifica, visualizzazione e quindi il rispettivo codice html, scss e typescript.
  - **Home**: contiene tutti i componenti utili che vengono visualizzati nella home come la navbar, il footer e il contenuto centrale e quindi il rispettivo html, scss e typescript.
  - **Main**: contiene tutti i componenti utili che vengono visualizzati nella dashoboard come la sidenav, i grafici, le tabelle e quindi il rispettivo codice html, scss e typescript.
  - **Pet**: contiene tutti i componenti utili alla gestione degli animali: aggiunta/-modifica, visualizzazione e quindi il rispettivo codice html, scss e typescript.
  - **User**:contiene tutti i componenti utili alla gestione degli utenti: registrazione e accesso; quindi il rispettivo codice html, scss e typescript.
  - **Utility**: contiene tutti le regole css che tutti i componenti condividono.

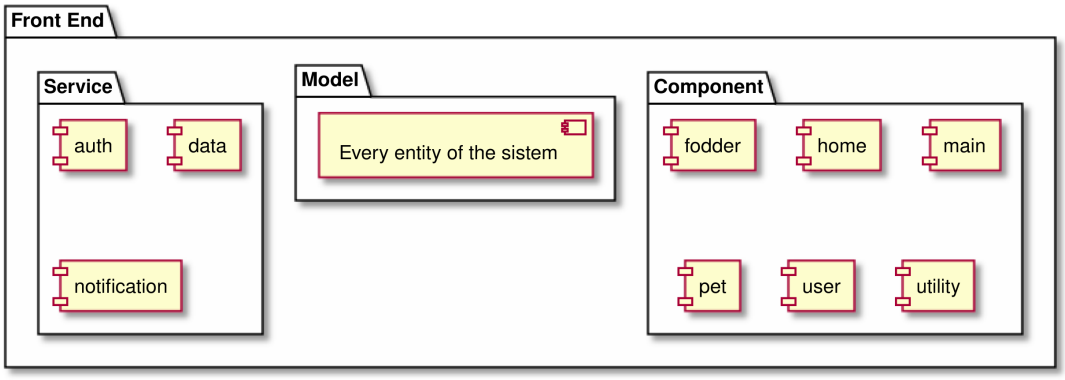

Figura 8: Diagramma package dei componenti del front-end

Nelle struttura del front-end si è cercato di suddividere il più possibile i **Component** in base al topic di rappresentazione, mentre per i **Service** si si sono suddivisi in base alle funzionalità che essi svolgono: dividendo così il servizio che si occupa di gestione degli utenti da quello che svolge il ruolo di gestione della comunicazione con il Server per ricevere i dati oppure gestire la comunicazione delle notifiche tramite WebSocket.

3.5 Target User Analysis

Il target principale del sistema è il padrone di animali; di seguito vengono individuate diverse *personas* differenziate da ciò che si aspettano dal sistema e il servizio che esso gli offre.

Personas: Giulia

Giulia ha appena ricevuto in regalo dai suoi genitori un cucciolo di Labrador.

Scenario d'uso: Giulia vuole controllare le calorie assunte dal cucciolo ogni giorno

- Giulia accedere alla piattaforma;
- Giulia si iscrive, registrando i suoi dati;
- Giulia aggiunge il suo cucciolo tra gli animali;
- Giulia aggiunge un determinato mangime per cani appena nati;
- Giulia associa due razioni al cane: una al mattino e una alla sera;
- Giulia tramite la dashboard può controllare le calorie assunte dal suo animale.

Personas: Leonardo

Leonardo è un rappresentante di una grande azienda ed è spesso fuori casa e vive con due siamesi di 14 anni.

Scenario d'uso: Leonardo vuole assicurarsi che essi mangino anche quando lui non c'è.

- Leonardo accede alla piattaforma in cui si è già registrato;
- Leonardo aggiunge i due gatti;
- Leonardo aggiunge razioni specifiche nei periodi in cui è lontano da casa;
- Leonardo riceverà notifiche sul completamento del pasto ed eventuali allarmi sul dispenser.

Personas: Luisa e Mario

Luisa e Mario sono due nonni a tempo pieno che vivono con i figli e i propri nipoti; hanno 3 pastori tedeschi sovrappeso.

Scenario d'uso: Luisa e Mario vogliono controllare che il cibo venga dato ai loro animali solo in determinate ore del giorno e nella quantità giusta

- Luisa accede alla piattaforma in cui si è già registrata;
- Luisa aggiunge i 3 cani con le loro caratteristiche;
- Luisa gestisce per ognuno di essi una razione ad orari diversi;
- Luisa potrà tenere sotto controllo le calorie assunte e i costi che la famiglia sta sostenendo per i propri animali;

4 Tecnologie

Si è utilizzato come solution stack *MEAN*.

Le tecnologie utilizzate per sviluppare il sistema sono:

- **Node ed Express**: sviluppo componente server;
- **Angular** : sviluppo componente client;
- **MongoDB**: gestione persistenza dati;
- **WebSocket**: gestione notifiche real time;
- **Redis**: db in-memory per memorizzare gli ID delle websocket;
- **Docker**: per il deploy dell'applicazione.

I linguaggi utilizzati all'interno del progetto sono:

- **Typescript**: sia lato client che server;
- **Html**: per la struttura delle pagine web;
- **Scss**: per i fogli di stile;

Docker

Docker rappresenta lo standard de-facto per la distribuzione di sistemi. A tal proposito il sistema è distribuito mediante `docker-compose`; il file presenta quattro servizi:

- **backend**
- **frontend**
- **mongodb**
- **loadbalancer**

Se con i primi tre servizi è piuttosto chiaro il ruolo che svolgono, per l'ultimo risulta importante chiarire lo scopo e la funzione che ricopre. Il servizio **loadbalancer** non è altro che `nginx` configurato per servire le richieste http in ingresso verso il servizio **frontend** e gestire le chiamate API (interne all'ambiente docker) verso il servizio **backend**. Inoltre `nginx` può fungere anche da load balancer qualora fosse richiesta scalabilità al sistema.

WebSocket

WebSocket è una tecnologia che consente la comunicazione bidirezionale tra client e server. Questa tecnologia è stata impiegata per implementare funzionalità di notifiche real-time. Questa tecnologia supera le limitazioni che poneva, ad esempio, l'utlizzo di `AJAX`: con quest'ultima tecnologia si agiva con una tecnica definita *polling*, ovvero è il client che periodicamente interroga il server per vedere se è disponibile una notifica. Questo metodo è piuttosto inefficiente e per tal motivo le **WebSocket** superano questa problematica instaurando una connessione TCP (mediante socket) tra client e server. Questo consente di avere una comunicazione asincrona e bidirezionale tra le parti potendo così implementare meccanismi di notifiche in tempo reale.

Swagger - OpenAPI

OpenAPI rappresenta uno standard per definire e documentare REST API di un sistema. Una delle principali problematiche dell'utilizzo di REST API è quello di capire come usare le API; a tal proposito in passato venivano descritte in linguaggio naturale o con forme più strutturate ma altamente eterogenee tra loro. Questo ha portato negli utilizzatori delle API confusione e discrepanze d'uso. **OpenAPI** ha lo scopo di definire uno standard rigoroso per la descrizione di REST API, per questo motivo si è scelto di aderire a questo standard per documentare le API che l'applicazione offre.

5 Codice

Backend

Nel listato di cui sotto, viene riportato un frammento di codice che descrive come è stato possibile definire le specifiche *OpenAPI* mediante annotazioni e definizione di classi.

```
1  /**
2   * Create a new user in the system.
3   * @param body .
4   */
5  @Post()
6  @ResponseSchema(UserResponse)
7  public async createNewUser(@Body() body: CreateUserBody): Promise<
       UserResponse> {
8      const newUser = new User()
9      newUser.email = body.email
10     newUser.password = body.password
11     newUser.profile = {
12         gender: body.gender,
13         picture: '',
14         firstName: body.firstName,
15         lastName: body.lastName
16     }
17     const saveState = await this.userRepository.create(newUser)
18     if (saveState) return new UserResponse(`New user create successfully:
        ${newUser.email}`)
19     else throw new HttpError(500, `Error on create user`)
20 }
```

Mediante l'annotazione `@Post()` si dichiara che questo metodo dovrà essere invocato quando si effettua una `POST` alla rotta indicata nell'annotazione della classe. Con `@ResponeSchema()` si definisce che struttura deve avere la risposta che il server produce a seguito della richiesta. I paramenti del metodo come `@Body() body: CreateUserBody` definisce la struttura che deve avere la richiesta da parte del client.

Mediante semplici annotazioni è possibile la documentazione automatica delle API.

```
export interface Read<T> {
    retrieve(): Promise<T[]>
    findById(id: string, populate?: string): Promise<T>
    findOne(query: any, projection?: string): Promise<T>
    findMany(query: any, projection?: string): Promise<T[]>
}

export interface Write<T> {
    create(item: T): Promise<T>
    update(id: mongoose.Types.ObjectId, item: T): Promise<T>
    updateWithQuery(query: any, item: any): Promise<T>
    delete(id: mongoose.Types.ObjectId): Promise<T>
    findAndUpdate(query: any, item: any): Promise<T>
}

export class BaseRepository<T extends Document>
    implements Write<T>, Read<T> {
    ..
}

@Service()
class UserRepository extends BaseRepository<UserDocument> {
    constructor() {
        super(User);
    }

    async findOne(query: any, projection?: string, populate?: string):
    Promise<UserDocument> {
        return this._model.findOne(query, projection).populate(populate)
    }
}
Object.seal(UserRepository)
export = UserRepository
```

Nel listato viene mostrato come è stato implementato il repository pattern. Mediante la definizione di due interfacce `Read` e `Write` sono state definite le operazioni che si possono effettuare su un modello. Viene definita quindi una classe base che implementa tali interfacce ed è generica sul tipo di collezione di mongoose. Il generico estende la classe `Document` che modella un documento di MongoDB, per motivi di semplicità è omessa l'implementazionde della classe `BaseRepository`. La reificazione di questa classe la si ha con le entità definite nel sistema, nell'esempio viene rappresentata la classe `UserRepository` che non fa altro che estendere la classe `BaseRepository` e passare al costruttore (mediate `super`) il modello di definizione e istanziare il generico mediante la classe `UserDocument`.

Questo approccio ha semplificato notevolemtne le operazioni sul DB senza ripetizioni di codice.

Frontend

Gestione errori lato client e server e visualizzazione su pagina

Quando l'utente clicca il pulsante di login viene richiamata la funzione `loginUser()` la quale controllerà se il form è valido e tramite il *Service* `AuthService` eseguirà la funzione di `signIn` inviando quindi una richiesta di POST al server. Tale codice si può vedere nel listato sotto.

```
  async loginUser()
    this.submitted = true;
    this.errorMessage = '';
    if (this.signInForm.invalid) {
      return;
    }
    this.authService.signIn(this.signInForm.value).subscribe(() => {
        this.router.navigate(['/dashboard']);
      },
      (error => {
        console.error('error caught in component');
        this.errorMessage = error;
        throw error;
      }));
  }

  ngOnInit(): void {
    this.signInForm = this.fb.group({
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required, Validators.minLength(5)]]
    });
  }
```

Se l'operazione va a buon fine si naviga nella pagina di dashboard, altrimenti viene generato un errore e lato html viene mostrato un alert; tale codice si può vedere nel listato sotto.

```
<div class="form-group">
 <input type="password" placeholder="Password"
  formControlName="password"
    [ngClass]="{ 'is-invalid': submitted &&
    f.password.errors }"/>
     <div *ngIf="submitted && f.password.errors" class="invalid-feedback"
    >
        <div *ngIf="f.password.errors.required">
        Password is required</div>
        <div *ngIf="f.password.errors.minlength">
        Password must be at least 6 characters</div>
    </div>
</div>
<div *ngIf="errorMessage" class="alert alert-danger">
    <strong>Error!</strong> {{'Username or password are not correct'}}
</div>
```

Gestione notifiche con WebSocket Service

Nel listato sotto viene riportato il servizio che si occupa di stabilire una connesione con il server tramite web socket e si occupa di inviare il token dell'utente attuale per poter gestire in canali autorizzati le informazioni riguardanti uno specifico dispositivo. Tramite il comando `socket.on()` ci si mette in ascolto sul topic *notifications* e quando arriva un messaggio tramite un oggetto Observable viene gestito il contenuto.

```
export class WebsocketService {
  private url = 'http://' + environment.apiBaseUrl;
  private token = localStorage.getItem('access_token');
  private socket: any;

  constructor() {
    this.socket = io(this.url, {
      transports: ['websocket']
    });
  }

  setupSocketConnection() {
    this.socket.on('connection', () => {
      this.socket.emit('authentication', {
        token: this.token
      });
    });
  }

  getMessage() {
    return new Observable(observer => {
      this.socket.on('notifications', data => {
        observer.next(data);
      });
    });
  }
}
```

Tramite la memorizzazione in fase di login del token attuale dell'utente, come mostrato nel listato sotto, è possibile instaurare una nuova connessione per l'utente loggato.

```
  // Sign-in
  signIn(user: User): Observable<any>  {
    return this.http.post(`${this.endpoint}/users/login`, user).
    pipe(
      map((result: any) => {
          localStorage.setItem('access_token', result.token);
          this.socketService.setupSocketConnection();
          return result || {};
      }),
        catchError(this.handleError)
    );
  }
```

Dentro al componente della *Dashboard* ci si sottoscrive all'oggetto osservabile che appena viene riempito si effettuerà una chiamata al *NotificationService* per mostrare la

notifica in base alla tipologia e contenuto di essa; tale comportamento è osservabile nel listato sotto.

```
1  this.websocket.getMessage().subscribe((data: Notification) => {
2       this.notifyService.showNotification(data.message, data.
     notificationType);
3     });
```

6 Deployment

Per la fase di deployment si è fatto uso di **Docker**. Questo semplifica notevolmente la distribuzione del sistema senza alcuna dipendenza specifica con una piattaforma. Nello specifico sono stati definiti due container docker: uno per il frontend e uno per il backend. Si è deciso di adottare questa scelta per un semplice fatto di modularità; ovvero si rende possibile la creazione di diverse interfacce grafiche senza legarsi all'implementazione proposta con angular. Come altra motivazione c'è quella dell'interazione con il backend: qualora si volesse realizzare una applicazione mobile (IOS o Android) questa dovrà interagire solamente con il backend, quindi per un migliore isolamento del sistema si è preferito separare frontend e backend. L'interazione di questi due container viene effettuata in automatico mediate `docker-compose`. Nel file di docker-compose è stato definito un terzo servizio che funge da loadbalacer per la parte di frontend e backend; oltre a tale servizio è definito il servizio di MongoDB che serve per lo storage dei dati.