



Transilvania
University
of Brasov

FACULTY OF MATHEMATICS
AND COMPUTER SCIENCE

Lucrare de licență

FoodSpy

Aplicație web pentru gestiunea aportului zilnic de calorii

Autor: Manghiuc Teodor-Adrian

Coordonator: Monescu Vlad

Brașov, România
2020-2021

Abstract

Mențiuni

Contents

1	Introducere	5
2	Aplicații web - noțiuni teoretice	6
3	Limbaje de programare	7
3.1	JavaScript / TypeScript	7
3.2	C#	9
3.3	HTML și CSS / SCSS	12
4	Framework-uri web	16
4.1	Angular	16
4.1.1	Components	17
4.1.2	Templates	17
4.1.3	Dependency Injection	18
4.2	ASP.NET Core	19
5	Tehnologii	21
5.1	MikroORM și ORM	21
5.2	MongoDB și MongoDB Atlas	21
5.3	Node.js	22
5.4	Express	23
5.5	NPM și publicarea pachetului "FoodSpy-shared"	24
5.6	Heroku	25
5.7	Utilitare (Postman, Git și GitHub)	25
6	Descrierea aplicației	28
6.1	Clientul	29
6.2	API pentru administrarea utilizatorilor	33
6.3	API pentru gestiunea jurnalelor utilizatorilor	37
7	Utilizarea aplicației	44
8	Concluzii	45
9	Perspective de dezvoltare	46

Capitolul 1

Introducere

Capitolul 2

Aplicații web - noțiuni teoretice

Capitolul 3

Limbaje de programare

3.1 JavaScript / TypeScript

JavaScript este un limbaj de programare care permite implementarea unor funcționalități complexe în paginile web. Câteva dintre aceste funcționalități sunt actualizarea conținutului paginilor web în mod dinamic, animarea de grafice 2D sau 3D și afișarea hărților interactive. Alături de HTML și CSS, JavaScript este unul dintre tehnologiile web standard.

JavaScript este un limbaj multi-platformă, poate fi folosit pe partea de client, dar și pe partea de server, este ușor de învățat și este dinamic și flexibil. Această dinamicitate este însă și unul din dezavantajele limbajului. De exemplu, JavaScript pune la dispoziție variabile primitive de tip "string" sau "number", dar pentru că nu este necesară compilarea codului, nu se verifică folosirea corectă a acestor variabile, programatorul putând folosi o variabilă "title" pentru a reține un "string" (Fig. 3.1), dar care mai târziu poate va fi suprascrisă și se poate reține (în variabilă) un "number", fără niciun fel de avertizare din partea mediului de programare, putând conduce la bug-uri și la comportamente neprevăzute.

```
1 let title = "FoodSpy";
```

Fig. 3.1: Definirea variabilei "title"

Aici intervine TypeScript, un superset al limbajului JavaScript ce adaugă "siguranța de tip" necesară pentru evitarea situațiilor neplăcute precum este cea descrisă mai sus. TypeScript încurajează scrierea codului în stil declarativ, asemănător limbajelor C# sau Java prin facilități ca interfețe, clase și inferența de tip a variabilelor.

Este necesară compilarea codului scris în limbaj TypeScript, dar acest lucru poate fi văzut ca un avantaj pentru că mediul de programare poate astfel scoate în evidență comportamentul neașteptat al codului și prin urmare poate duce la detecția mult mai rapidă a bug-urilor. Referindu-ne la situația neplăcută descrisă mai sus, dacă variabila "title" ar fi adnotată cu tipul "string", atunci nu va putea fi suprascrisă de o valoare pretabilă unei variabile de tip "number".

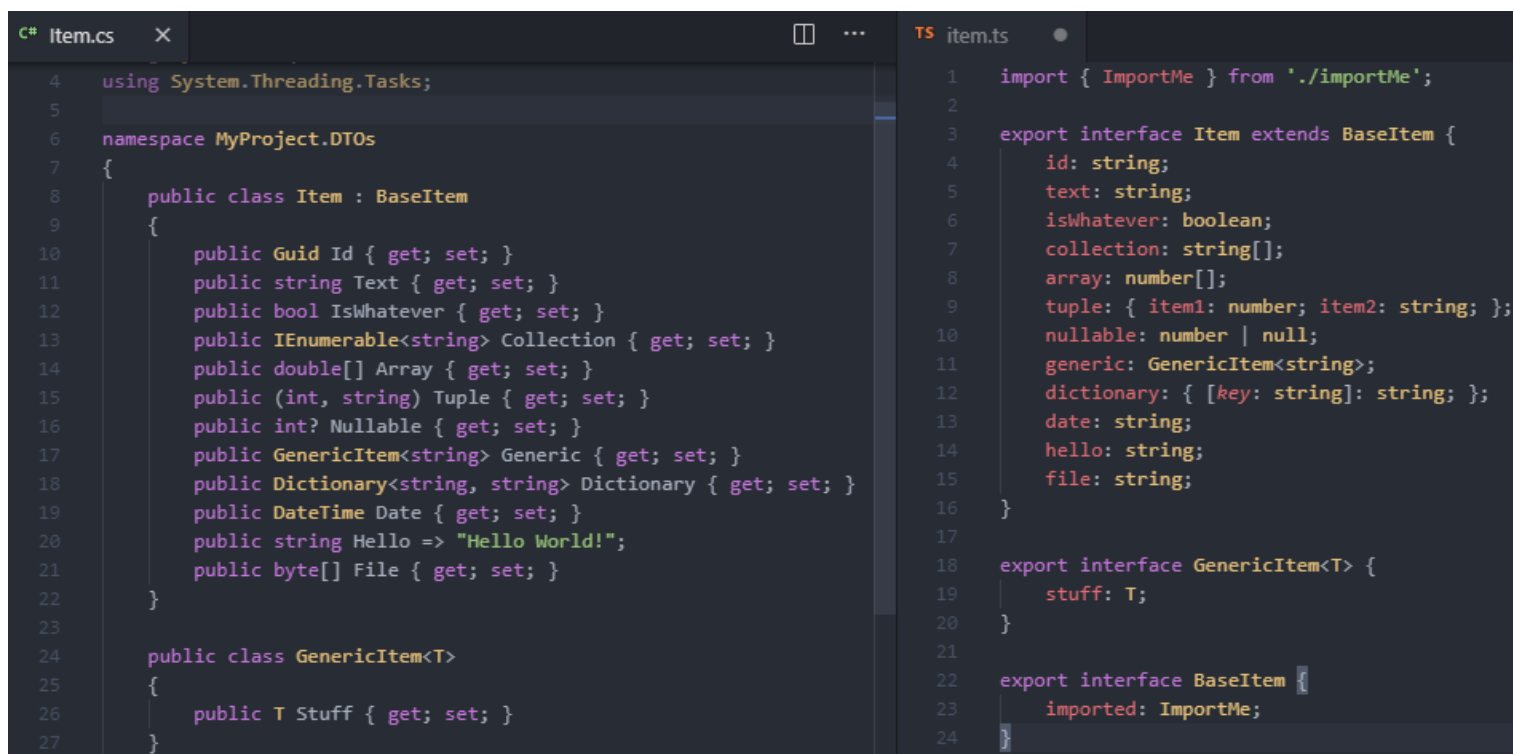
TypeScript vine în ajutorul programatorilor care sunt deja obișnuiți cu limbaje de programare cu stil declarativ și propune astfel o alternativă față de JavaScript care nu oferă

```
1 let title: string = "FoodSpy";
2 let title: string
3
4 Type 'number' is not assignable to type 'string'. (2322)
5 View Problem (Alt+F8) No quick fixes available
6 title = 4;
```

Fig. 3.2: Adnotarea previne suprascrierea variabilei "title"

niciun suport pentru "siguranța de tip".

Deși o mai bună înțelegere a limbajului JavaScript este recomandată, TypeScript se poate învăța și folosi în mod independent, fără a cunoaște toate dedesubturile JavaScript. Mai mult, un bonus pentru programatorii cu experiență în C# este faptul că autorul limbajului TypeScript este arhitectul limbajului C#, deci, din punct de vedere sintactic, TypeScript și C# împărtășesc multe similitudini.



The image shows a side-by-side comparison of C# and TypeScript code in Visual Studio Code. On the left, the C# file 'Item.cs' defines a namespace 'MyProject.DTOS' containing a 'BaseItem' class and a 'GenericItem<T>' class. The 'BaseItem' class has properties like 'Id', 'Text', 'IsWhatever', 'Collection', 'Array', 'Tuple', 'Nullable', 'Generic', 'Dictionary', 'Date', 'Hello', and 'File'. On the right, the TypeScript file 'item.ts' defines an 'importMe' module, an 'Item' interface extending 'BaseItem', a 'GenericItem<T>' interface, and a 'BaseItem' interface. The TypeScript code mirrors the structure and property names of the C# code, demonstrating the syntactic similarities between the two languages.

Fig. 3.3: Asemănări între C# și TypeScript

Față de JavaScript, TypeScript întinde o plasă de siguranță care poate prinde defectele de programare introduse prin modificări majore ale codului. TypeScript nu va înlătura necesitatea depanării programelor, dar "siguranța de tip" va ajuta fără echivoc la detectarea din timp ale defectelor. Mai mult, TypeScript beneficiază de sugestii și avertizări din partea mediului de programare încă de la compile-time, în timp ce erorile în JavaScript pot fi descoperite abia la run-time.

Printre alte avantaje ale TypeScript includem și posibilitatea re-factorizării într-un mod

mai sigur deoarece se cunoaște codul la nivel semantic și capabilitățile TypeScript care permit construirea aplicațiilor de mărime foarte mare, în completă antiteză cu JavaScript care a fost gândit ca un limbaj folosit strict pentru modificarea dinamică a paginilor web.

3.2 C#

C# este un limbaj de programare orientat pe obiect, simplu, dar puternic și este gândit pentru dezvoltarea de aplicații pe platforma .NET Framework. Moștenește părțile bune ale limbajului C++ și Visual Basic, dar fără anumite inconsistențe¹ și anacronisme ceea ce duc la un limbaj mai logic și mai puțin complicat. La fel cum pentru o variabilă, simbolul "++" înseamnă incrementarea cu 1 după ce variabila a fost evaluată, pentru C#, simbolul "#" semnifică o incrementare a limbajului C++.

Prima versiune a apărut în 2001. Odată cu versiunea 2.0 au apărut lucrul cu generice, iteratori și metode anonime. În versiunea 3.0 au debutat metodele de extensie, expresiile lambda și cel mai important, LINQ sau Language-INtegrated Query. Versiunea 5.0 a venit cu suport nativ pentru operațiile asincron prin introducerea operatorului "await" și modifierului de funcție "async".

Lucrul cu clase și metode generice reprezintă un concept care face posibilă proiectarea de clase și metode ale căror tipuri se cunoaște doar la momentul declarării sau instanțierii. Acest concept combină reutilizarea codului, "siguranța de tip" și eficiența într-un mod în care nu este posibil prin proiectarea specifică.

"GenericList" din (Fig. 3.4) este un exemplu de clasă generică care modelează o listă simplă înlănțuită capabilă să memoreze obiecte de tip "Node" de orice tip, inclusiv "int", "double" sau "string".

Parametrul de tip "T" ia locul declarării tipului explicit și este folosit ca parametru în metoda "AddHead(T t)", ca parametru de retur al proprietății "Data" și ca tip de membru privat, cum este cazul membrului "T data". La momentul instanțierii clasei "GenericList<T>", fiecare apariție a lui "T" va fi înlocuită de tipul specificat între "<>". Se pot crea interfețe, clase, metode, evenimente și delegați generici.

Metodele de extensie sunt o funcționalitate a limbajului C# care permit adăugarea de metode tipurilor deja existente. Metodele de extensie sunt metode statice care sunt apelate ca și cum ar fi membre ale clasei, cum este de exemplu metoda "ToString()", membră a clasei "Object".

O metodă de extensie are cel puțin un parametru, "this" care reprezintă obiectul curent peste care operează metoda. Din cauza aceasta, când se apelează o metodă de extensie de către un obiect, obiectul nu trebuie trimis ca parametru. "WordCount" din (Fig.

¹O astfel de inconsistență în C++ este accesul la un element dintr-un "std::vector". În C++ se poate folosi operatorul "[]" pentru a ajunge la elemente care nu sunt de fapt elementele ale vectorului, ci sunt bucăți de informație conținute la adresele de memorie adiacente memoriei alocate vectorului. Deoarece nu se face niciun fel de verificare a indicelui elementului pe care încercăm să-l accesăm, operatorul "[]" va întoarce ceea ce se află la locația de memorie vecină memoriei alocate vectorului.

```
1 public class GenericList<T>
2 {
3     5 references
4     private class Node
5     {
6         1 reference
7         public Node(T t)
8         {
9             next = null;
10            data = t;
11        }
12
13        private Node next;
14        1 reference
15        public Node Next
16        {
17            get { return next; }
18            set { next = value; }
19        }
20
21        // T ca membru privat:
22        private T data;
23
24        // T ca parametru de retur al unei proprietăți:
25        0 references
26        public T Data
27        {
28            get { return data; }
29            set { data = value; }
30        }
31
32        private Node head;
33
34        // T ca parametrul unei metode:
35        0 references
36        public void AddHead(T t)
37        {
38            Node n = new Node(t);
39            n.Next = head;
40            head = n;
41        }
42    }
```

Fig. 3.4: Exemplu de clasă generică

3.5) este o metodă de extensie care numără caracterele dintr-un obiect de tip "string". Se observă că "WordCount" este definită pentru obiecte din clasa "System.String" deoarece parametrul "str" este prefixat de "this string".

Apelul metodei de extensie se poate observa în (Fig. 3.6).

```

3  public static class StringExtensions
4  {
5      public static int WordCount(this string str)
6      {
7          return str.Split(new char[] { ' ', '.', '?' },
8                          StringSplitOptions.RemoveEmptyEntries).Length;
9      }
10 }
```

Fig. 3.5: Exemplu de metodă de extensie

```

3  public static class Extension
4  {
5      public static void Main(string[] args)
6      {
7          string s = "FoodSpy";
8          int count = s.WordCount();
9      }
10 }
```

Fig. 3.6: Apelul unei metode de extensie

O definiție simplistă pentru LINQ este următoarea: modalitatea prin care se poate interoga o bază de date folosind un limbaj de interogare apropiat de SQL, dar care este compilat în interiorul unei aplicații care rulează pe platforma .NET. Prin "bază de date" se înțelege un spectru larg de surse de date, printre care baze de date SQL, baze de date "in-memory" sau reprezentări XML.

LINQ permite scrierea unei interogări sub forma unei structuri SQL, după cum se poate observa în (Fig. 3.7). Această structură are avantajul că variabila "name" este "scoped", adică nu trebuie definită din nou pentru fiecare clauză din interogare.

Un alt avantaj este claritatea originii variabilei "name" - chiar din primul rând (linia 11) se distinge faptul că "name" provine din sursa de date "names", spre deosebire de interogarea SQL clasică, unde clauza "from" ar fi fost declarată ultima.

LINQ nu ar fi fost un concept realizabil dacă nu s-ar fi implementat în prealabil funcționalități precum metodele de extensie, tipurile anonime², tipurile implicite³ și expresiile lambda.

²List<string> names = new List<string>() { "Teodor", "Adrian", "Manghiuc" };

Din construirea unei variabile, compilatorul are la dispoziție suficiente informații pentru a pune la dispoziție un tip de date care să modeleze variabila.

³var anonymousType = names.Select(n => new { Name = n, Capitalized = n.ToUpper() });

Din inițializarea unei variabile, compilatorul are la dispoziție suficiente informații pentru a infera tipul acesteia.

var inference = names.Select(n => n);

devine

IEnumerable<string> inference = names.Select(n => n);

```
public class LINQExample
{
    0 references
    public IEnumerable<string> GetNames()
    {
        List<string> names = new List<string>() { "Teodor", "Adrian", "Manghiuc" };
        IEnumerable<string> result = from name in names
                                    where name.StartsWith("A")
                                    select name;

        return result;
    }
}
```

Fig. 3.7: Exemplu de interogare LINQ

3.3 HTML și CSS / SCSS

Acronimul HTML vine de la Hypertext Markup Language și reprezintă limbajul care permite structurarea conținutului unei pagini web prin construirea de secțiuni, definirea de rubrici, paragrafe și link-uri. HTML nu este un limbaj de programare, deci nu permite construirea de conținut cu funcționalitate dinamică, dar face posibilă organizarea și formatarea paginilor web, asemenea documentelor scrise în Microsoft Word.

HTML este format dintr-o colecție de elemente folosite pentru a "înfășura" bucățile de conținut astfel încât să arate sau să se comporte într-un anumit fel. De exemplu, folosind elementul "<a>" putem crea un link către o altă pagină web, cu "<i>" putem scrie înclinat, iar cu "<h1>" putem defini o rubrică. În (Fig. 3.8), este ilustrat elementul "<p>".

```
1. <p>This is how you add a paragraph in HTML.</p>
2. <p>You can have more than one!</p>
```

Fig. 3.8: Exemplu de element HTML

"p" vine de la paragraf. "<p>" se mai numește "opening tag" și definește punctul de intrare al elementului de tip paragraf și se poate observa că este închis între simbolurile "<" și ">". "</p>" reprezintă "closing tag" și marchează punctul de ieșire al elementului, care, spre deosebire de punctul de intrare, mai are în componență simbolul "/". Textul sau conținutul paragrafului este definit între punctul de intrare și cel de ieșire. Un element HTML este format din toate cele 3 părți descrise mai sus.

Acronimul CSS vine de la Cascading Style Sheets și reprezintă limbajul care permite determinarea aspectului unei pagini web, din punct de vedere vizual, schematic și estetic. Dacă HTML este planul unei case, CSS precizează stilul acesteia, Victorian sau Art Deco, Post-modern sau Gotic.

CSS schimbă aspectul paginii web interacționând cu elementele HTML. După cum am

observat în (Fig. 3.8), un paragraf este un asemenea element HTML. Dacă vrem ca textul paragrafului să fie de culoare roz și cu scris îngroșat, vom scrie codul CSS ilustrat în (Fig. 3.9).

```
p { color:pink; font-weight:bold; }
```

Fig. 3.9: Exemplu de stilizare CSS

"p" se numește "selector" și indică elementul HTML căruia îi vom schimba aspectul folosind reguli definite în CSS. "color" și "font-weight" sunt câteva dintre proprietățile de stilizare care se pot aplica selectorului de tip "p". Alte proprietăți ale acestuia sunt dimensiunea textului, adică "font-size" sau "text-transform" care poate transforma textul paragrafului în litere de tipar dacă se folosește împreună cu valoarea "uppercase".

Codul CSS poate fi adăugat unei pagini web în 3 moduri: intern, extern sau inline. În modul intern, codul CSS este scris în elementul HTML "<style>", ilustrat în (Fig. 3.10).

```
<head>
<style>
Body { background-color:thistle; }
P { font-size:20px; color:mediumblue; }
</style>
</head>
```

Fig. 3.10: Modul intern de legare a codului CSS

Extern înseamnă scrierea codului într-un fișier separat salvat cu extensia ".css". Legătura dintre pagina web și regulile de stilizare din fișier se creează folosind elementul "<link>", după cum se poate observa în (Fig. 3.11).

```
<head>
<link rel="stylesheet" type="text/css" href="mysitestyle.css">
</head>
```

Fig. 3.11: Modul extern de legare a codului CSS

Codul "inline" este codul CSS scris înăuntrul elementelor HTML, folosind proprietatea "style", reprezentat în (Fig. 3.12).

```
<h1 style="font-size:40px;color:violet;">Check out this headline!</h1>
```

Fig. 3.12: Modul inline de legare a codului CSS

Se preferă scrierea regulilor de stilizare într-un fișier separat deoarece astfel rezultă o separare între structura paginii web, elementele HTML și aspectul acesteia, regulile

CSS. Codul este mai ușor de citit, mai ușor de întreținut și actualizat și permite o detectare mai ușoară a erorilor.

SCSS vine de la Sassy CSS și este un limbaj de preprocesare al cărui rezultat de compilare este cod CSS. În CSS este dificilă scrierea de reguli organizate și ușor de menținut deoarece lipsește suportul pentru cod imbricate (din engl. "nested"), funcții și moștenire. Folosind SCSS se poate evita scrierea codului duplicat, prin construirea de bucăți de cod reutilizabile și se pot scrie secvențe de cod imbricate, care sunt structurate și ușor de citit (Fig. 3.13).

SASS	CSS
<pre>.class { color: #ffdd00; font: 16/1.4 Arial; margin: 0; a { display: block; width: 100px; height: 50px; } }</pre>	<pre>.class { color: #ffdd00; font: 16/1.4 Arial; margin: 0; } .class a { display: block; width: 100px; height: 50px; }</pre>

Fig. 3.13: Secvență de cod "nested"

Simbolul "." din ".class" reprezintă o clasă de reguli de stilizare care se poate aplica unuia sau mai multor elemente HTML, spre deosebire de simbolul "#" care definește o regulă aplicabilă în mod unic unui singur element. După cum se poate observa în (Fig. 3.13), codul "nested" evită dublarea selectorului ".class" atunci când se face referire la elementul "<a>", de tip ancoră.

SCSS oferă suport pentru așa numitele "directive de control", care pot defini o regulă de stilizare dacă o anumită condiție este îndeplinită. Una dintre aceste directive este "@if" care funcționează asemenea structurilor alternative din limbajele de programare precum C#. Folosind "@if", se aplică unui element "<div>" o margine albă de grosime de 1 pixel, dacă rezultatul condiției este egal cu 4, după cum este ilustrat în (Fig. 3.14). După evaluarea condiției din "@if" și compilarea codului, rezultă regula de stilizare reprezentată în (Fig. 3.15).

```
div {  
  @if 2+2 == 4 { border : 1px solid white ;}  
}  
@if 1 > 2 { border : 3px solid white ;}  
@if null { border : 3px solid white ;}  
}
```

Fig. 3.14: Directiva de control "@if"

```
div{  
  Border : 1px solid white;  
}
```

Fig. 3.15: Directiva de control "@if"

Capitolul 4

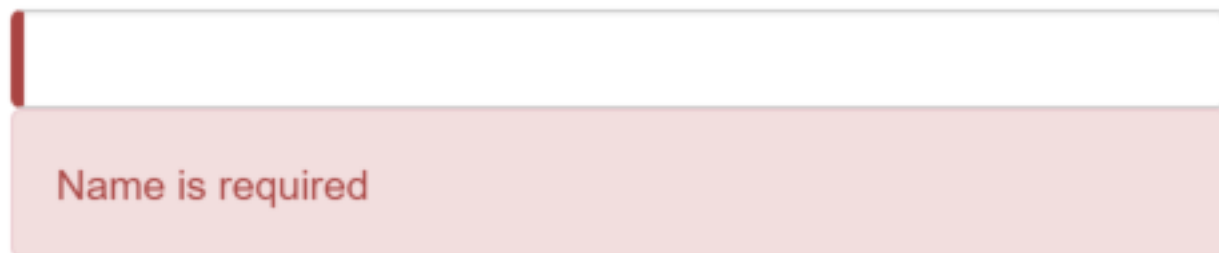
Framework-uri web

4.1 Angular

Numele "Angular" vine de la caracterele "<" și ">" (în engleză, "angled brackets") care delimitează numele tuturor tag-urilor din HTML, de exemplu "<a>" sau "<div>". Angular este o platformă de dezvoltare, bazată pe limbajul TypeScript, care este un superset al limbajului JavaScript. Angular este un framework care face posibilă implementarea într-un mod trivial a cerințelor complexe ale aplicațiilor web, cum ar fi animații sau "data binding".

Platforma Angular oferă un mod de lucru bazat pe componente individuale care duc cu ușurință la construirea de aplicații web scalabile, dar și la posibilitatea reutilizării acestor componente în alte aplicații. Angular vine integrat cu o multitudine de biblioteci care acoperă o gamă largă de cazuri de utilizare, printre care comunicarea cu un server și validarea formularelor HTML (Fig. 4.1).

Name



The image shows a web form with a label 'Name' above an input field. The input field is empty. Below the input field, there is a red rectangular box containing the text 'Name is required' in a dark red font, indicating a validation error.

Fig. 4.1: Validarea câmpului "Name" din formular

Mai mult, Angular este gândit să faciliteze actualizarea cât mai ușoară a aplicației, de aceea pune la dispoziție o suită de unelte care ajută la dezvoltarea, testarea și actualizarea codului printre care și "Live Reload", o funcționalitate prin care aplicația web este recompilată și deservită la orice modificare a codului-sursă.

4.1.1 Components

Asemenea blocurilor de LEGO, componentele Angular sunt puse laolaltă pentru a construi o aplicație web. O componentă (Fig. 4.2) este formată dintr-o clasă TypeScript, adnotată cu decoratorul "@Component", un șablon HTML și opțiuni de stilizare.

În clasa TypeScript este scris codul care determină funcționalitatea componentei, șablonul HTML spune cum este structurată componenta, iar opțiunile de stilizare definesc aspectul acesteia. Această separare ajută la testarea mai ușoară a codului și lizibilitatea acestuia.

```
@Component({
  selector: 'hello-world',
  template: `
    <h2>Hello World</h2>
    <p>This is my first component!</p>
  `,
})
export class HelloWorldComponent {
  // The code
}
```

Fig. 4.2: O componentă Angular

Componenta se folosește printr-un apel după numele descris sub proprietatea "selector", în cazul acesta "<hello-world></hello-world>".

4.1.2 Templates

"Template" sau "șablonul" descrie felul în care este structurată componenta folosind limbaj HTML. Limbajul HTML poate fi scris "inline", cum se vede în (Fig. 4.2) sub proprietatea "template" sau poate fi scris într-un fișier separat cu extensia ".html". Angular extinde capabilitățile limbajului HTML oferind așa numitele "directives", care sunt clase ce pot modifica în mod dinamic structura aplicației.

De exemplu, "*ngIf" poate fi folosit pe un "<div>" (dar și alte tag-uri HTML) pentru a-l afișa sau nu în funcție de valoarea unei variabile de tip "boolean", astfel:

```
<div *ngIf="showDiv">
  <p> Dacă "showDiv" este "true" atunci "<div>" este vizibil în pagină </p>
</div>
```

```
<div *ngIf="!showDiv">
  <p> Dacă "showDiv" este "false" atunci "<div>" nu este vizibil în pagină </p>
</div>
```

De reținut este faptul că "showDiv" este o variabilă ce a fost definită în clasa TypeScript a componentei.

4.1.3 Dependency Injection

Prin "dependency injection" a se înțelege un design pattern prin care este posibilă scrierea de cod-sursă mai flexibil și mai ușor de testat. În Angular, prin "dependency injection", într-o clasă se pot declara dependențe către alte clase, fără să fie necesară instanțierea explicită a lor - instanțierea se efectuează în mod automat. O clasă către care este definită o dependență din altă clasă trebuie să fie adnotată cu decoratorul "@Injectable", astfel:

```
@Injectable({providedIn: 'root'})  
export class Logger {  
  writeCount(count: number) {  
    console.warn(count);  
  }  
}
```

Fig. 4.3: "Logger" este o clasă ce poate fi "injectată"

Funcția "writeCount()" din clasa "Logger" este folosită mai departe în interiorul clasei "HelloWorldDependencyInjectionComponent", după cum se vede în (Fig. 4.4).

```
@Component({  
  selector: 'hello-world-di',  
  templateUrl: './hello-world-di.component.html'  
})  
export class HelloWorldDependencyInjectionComponent {  
  count = 0;  
  
  constructor(private logger: Logger) {  
  }  
  
  onLogMe() {  
    this.logger.writeCount(this.count);  
    this.count++;  
  }  
}
```

Fig. 4.4: "HelloWorldDependencyInjectionComponent" cu membrul "private logger"

A se observa constructorul clasei "HelloWorldDependencyInjectionComponent", unde s-a definit o dependență către clasa "Logger" prin membrul "private logger".

4.2 ASP.NET Core

ASP.NET Core este un framework web, care rulează pe platforma .NET și .NET Core. Bazat pe limbajul de programare C#, ASP.NET Core suportă construirea serviciilor REST (așa numite API-uri web). Un "request" sau o "cerere" venită către un asemenea serviciu REST este tratată de către un controller. În ASP.NET Core, un controller este o clasă ce derivă din clasa de bază "ControllerBase".

```
[ApiController]
public class PeopleController : ControllerBase
{
    [HttpGet("people/all")]
    public ActionResult<IEnumerable<Person>> GetAll()
    {
        return new []
        {
            new Person { Name = "Ana" },
            new Person { Name = "Felipe" },
            new Person { Name = "Emillia" }
        };
    }
}
```

Fig. 4.5: "PeopleController" este un exemplu de controller

Deoarece "PeopleController" este adnotat cu atributul "[HttpGet("people/all")]", acesta este capabil să răspundă cererilor metodei "GET" efectuate la URL-ul "/people/all".

"people/all" se mai numește "endpoint" și intrinsec este configurat să răspundă într-un mod serializabil, sub forma unui JSON, după cum se poate observa în (Fig. 4.6).

```
[{"name": "Ana"}, {"name": "Felipe"}, {"name": "Emillia"}]
```

Fig. 4.6: Exemplu de răspuns dat de un controller

După cum se poate observa și în exemplele de mai sus, ASP.NET Core permite definirea "inline" a endpoint-urilor și metodelor REST prin adnotări, folosind attribute. Un exemplu de controller capabil să răspundă la metodele "GET" și "POST" este ilustrat în (Fig. 4.7).

În practică deci, un web API construit folosind ASP.NET Core nu este altceva decât o colecție de unul sau mai multe controller-e care sunt capabile să răspundă cererilor venite din partea interfeței-utilizator (și nu numai).

```
[ApiController]
public class PeopleApiController : ControllerBase
{
    // Some code omitted for clarity

    [HttpGet("people/{id}")]
    public ActionResult<Person> Get(int id)
    {
        var person = db.People.Find(id);

        if (person == null)
        {
            return NotFound();
        }

        return person;
    }

    [HttpPost("people/create")]
    public IActionResult Create(Person person)
    {
        db.Add(person);
        db.SaveChanges();
        return Accepted();
    }
}
```

Fig. 4.7: Exemplu controller

Capitolul 5

Tehnologii

5.1 MikroORM și ORM

ORM sau Object-Relational Mapping este o tehnică de a interoga și manipula datele dintr-o bază de date folosind o paradigmă orientată pe obiect. MikroORM este o bibliotecă TypeScript care implementează această tehnică. MikroORM încapsulează codul necesar manevrării datelor și astfel nu mai este nevoie de scrierea explicită a interogărilor.

Mai mult, biblioteca permite interacțiunea directă cu obiectele din baza de date folosind același limbaj în care este scris codul-sursă al aplicației - în cazul nostru, TypeScript.

MikroORM prezintă o serie de avantaje printre care faptul că utilizarea procedurilor stocate sau a tranzacțiilor se face la fel de simplu precum un apel de funcție, dar și faptul că modelul de date este definit o singură dată, ceea ce îl face ușor de actualizat și menținut. Un exemplu de model de date este "User". "User" modelează utilizatorul aplicației, prin câmpuri precum "email", și este reprezentat în (Fig. 5.1).

5.2 MongoDB și MongoDB Atlas

MongoDB este un sistem de gestiune a bazelor de date orientat pe document. Deoarece un document este asemenea unui obiect JSON, câmpurile conținute în acesta pot să difere de la document la document și de asemenea, structura acestuia se poate modifica de-a lungul timpului. MongoDB permite o asociere directă între documentul din baza de date și obiectul din codul-sursă al aplicației, oferind programatorului posibilitatea de a manipula datele cu ușurință.

La rădăcini, MongoDB este un sistem de gestiune scalabil și pretabil pentru a rula pe sisteme distribuite - aici intervine MongoDB Atlas care este serviciul MongoDB găzduit în cloud. Atlas pune la dispoziție toate funcționalitățile serviciului MongoDB, dar în același timp automatizează partea de administrare a bazei de date, de la configurarea acesteia până la efectuarea operației de "backup".

În oferta de găzduire în cloud există și un nivel gratis, așa numitul "MO Sandbox", care poate fi folosit pentru acomodare cu serviciul MongoDB, pentru prototipare și pentru primele implementări ale bazei de date.

```
12 @Entity()
13 export class User implements IUser {
14
15     @PrimaryKey()
16     _id!: ObjectId;
17
18     @SerializedPrimaryKey()
19     id!: string;
20
21     @Property()
22     email!: string;
23     @Property()
24     password!: string;
25
26     public constructor(partial?: Partial<User>) {
27         Object.assign(this, partial);
28     }
29 }
```

Fig. 5.1: Model de date MikroORM



```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

Fig. 5.2: Document MongoDB

5.3 Node.js

Node.js este un mediu de rulare JavaScript construit pe engine-ul V8 din Chrome. Este folosit pentru a rula aplicații web în afara browser-ului clientului. Deoarece folosește un mediu de rulare asincron, bazat pe evenimente, Node.js este gândit pentru scalabilitatea aplicațiilor de rețea, fiind capabil să administreze un număr mare de conexiuni în mod concurent.

Un modul Node.js este asemenea unei librării JavaScript care poate fi inclus înăuntrul unei aplicații pentru a facilita anumite funcționalități. De exemplu, în (Fig. 5.3) este ilustrată utilizarea modului HTTP care face posibil crearea unui server web, care ascultă pe portul 2000.

```
// CREATING A WEB SERVER

// Include modules
var http = require('http');
var server =
http.createServer(function(req, res){
  //write your code here
});
server.listen(2000);
```

Fig. 5.3: Modulul HTTP din Node.js

Față de mediul de rulare convențional, bazat pe thread-uri, Node.js are avantajul că funcțiile nu execută operații de input / output, deci procesul de rulare al aplicației nu este blocat niciodată. Mai mult, librăria Express din Node.js poate fi utilizată în diferite scenarii, de la aplicații de chat în timp real până la API-uri REST.

5.4 Express

Node.js nu oferă suport pentru diferitele verbe HTTP, precum GET, POST sau DELETE. De asemenea, nu poate servi fișiere statice și nici nu poate trata cererile care pot veni la diferite URL-uri, așa numitele "rute" sau "endpoint"-uri. Pentru aceste cazuri, este necesară implementarea manuală sau folosirea unui framework web, în speță, Express.

Express adresează neajunsurile Node.js enumerate mai sus prin pachete "middleware" și mai mult decât atât, aceste pachete pot fi folosite pentru lucrul cu sesiuni sau cookie-uri, extragerea parametrilor din URL sau a datelor transmise prin corpul cererilor POST. Inițial, se importă modulul Express prin "require('express')". Aplicația Express se instanțiază prin "const app = express()". Cu ajutorul "app" definim "endpoint"-urile tratate de API. În cazul prezentat în (Fig. 5.4), "app.get()" definește o funcție care va fi apelată atunci când se face un request GET la URL-ul rădăcină, adică "/". Răspunsul se trimite înapoi folosind "send()" și va întoarce string-ul "Hello World". Cu "app.listen()" se

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!')
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}!`)
});
```

Fig. 5.4: "Hello World" în Express

pornește server-ul care ascultă pe portul 3000 și care poate fi accesat la adresa "localhost:3000".

5.5 NPM și publicarea pachetului "FoodSpy-shared"

NPM este manager-ul de pachete pentru Node.js. A fost creat în 2009 ca un proiect care să ajute dezvoltatorii JavaScript să partajeze cu ușurință secvențe de cod sub formă de module [REF](referință către Node.js) ambalate ca pachete. Registrul NPM este o colecție publică de astfel de pachete care sunt folosite de către comunitatea JavaScript pentru dezvoltarea aplicațiilor web sau aplicațiilor mobile. NPM pune la dispoziție un client care poate fi folosit în linie de comandă și oferă dezvoltatorilor capacitatea de a instala, folosi și de a publica propriile pachete.

"FoodSpy-shared" este un pachet propriu, disponibil în registrul public NPM. Acesta conține o colecție de constante, interfețe și enum-uri folosite în aplicația Angular, dar și în API-ul care administrează datele despre utilizatori. Una dintre aceste interfețe este "IUser", ilustrată în (Fig. 5.5). "IUser" definește contractul care trebuie îndeplinit atât pe partea de front-end cât și pe partea de back-end, când vine vorba de manipularea datelor despre un utilizator. Adăugarea pachetului "FoodSpy-shared" și implementarea interfeței "IUser" în API este ilustrată în (Fig. 5.6).

```
1  export interface IUser {
2    id: string;
3    email: string;
4    targetCalories: number;
5  }
```

Fig. 5.5: Interfața "IUser"


```
1  import { IUser } from 'foodspy-shared';
2
3  export class User implements IUser {
4      id!: string;
5      email!: string;
6      password!: string;
7      targetCalories!: number;
8
9      public constructor(partial?: Partial<User>) {
10         Object.assign(this, partial);
11     }
12 }
```

Fig. 5.6: Implementarea interfeței "IUser" în API

5.6 Heroku

Heroku este o platformă care pune la dispoziție posibilitatea de a construi, implementa și rula aplicații găzduite în cloud. Spre deosebire de Amazon Web Services sau Windows Azure, care oferă doar infrastructura în cloud necesară pentru a găzdui aplicația, Heroku oferă un serviciu de management asupra infrastructurii, care abstractizează toate detaliile legate de configurare și implementare. Astfel, Heroku permite dezvoltatorilor să concentreze toate resursele spre implementarea aplicației. Heroku suportă o colecție de limbaje de programare și framework-uri, printre care JavaScript și Node.js.

Platforma pune la dispoziție toate uneltele necesare implementării unei aplicații, inclusiv administrarea mediului de rulare a aplicației și administrarea laturii DevOps. Din acest motiv, Heroku este un loc potrivit pentru a învăța despre microservicii și metodologii de implementare, precum livrarea continuă ("continuous delivery"). Mai mult decât atât, oferă un abonament destinat exclusiv studenților care permite utilizarea platformei pentru o perioadă de doi ani, suficient timp pentru a exersa orice limbaj de programare și orice șablon arhitectural.

5.7 Utilitare (Postman, Git și GitHub)

Postman este un utilitar interactiv prin care se pot examina endpoint-urile unui API. Postman poate fi folosit pe tot parcursul dezvoltării unei aplicații deoarece dispune de o interfață grafică care permite construirea și trimiterea de request-uri HTTP. De asemenea, se poate folosi pentru citirea și validarea răspunsurilor oferite de către API. Un exemplu de request trimis direct din Postman către "FoodSpyAPI", care interoghează baza de date pentru alimente care au în componența numelui termenul "magiun" este reprezentat în (Fig. 5.7).

Git este un sistem de versionare distribuit care poate urmări schimbări în fișiere. Git este folosit de programatori pentru a colabora la scrierea codului-sursă în procesul de dezvoltare software. A fost creat în anul 2005 și a fost gândit pentru a fi eficient, pentru a păstra integritatea datelor și pentru a oferi suport pentru numeroase medii de

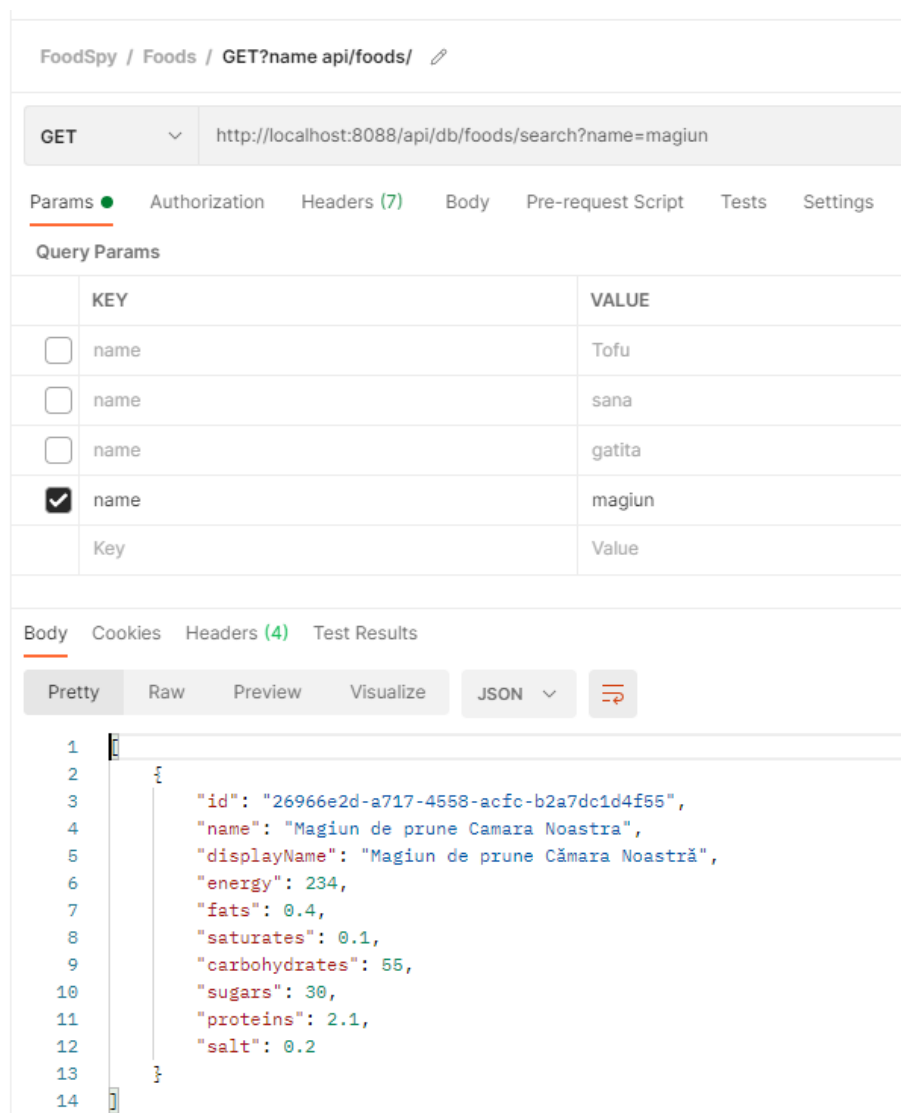


Fig. 5.7: Request "GET" trimis din Postman

dezvoltare.

Git urmărește schimbările în fișier pe toată durata dezvoltării software, ceea ce înseamnă că în orice moment, codul-sursă poate fi revizuit sau poate fi adus la o versiune precedentă.

Git se instalează și rulează pe sistemul local și nu are nevoie de conexiune la internet pentru a funcționa. Față de alte sisteme de versionare, Git este ușor de folosit și gratis și a fost gândit să lucreze în principal cu fișiere text, adică exact tipul de fișiere folosit pentru a salva cod-sursă. Întregul cod-sursă este menținut sub evidența sistemului de versionare într-un director care poartă numele de "repository". Avantajul major al Git este așa numitul "branching model". Prin procesul de "branching" se pot crea "ramuri", care sunt copii independente ale repository-ului. Asta se traduce prin ușurința de a testa și implementa noi funcționalități în aplicație, fără a afecta munca celorlalți participanți la dezvoltarea aplicației.

GitHub este un serviciu de găzduire a unui repository. Este bazat exclusiv pe infrastructură cloud și este o bază de date care reține o colecție de repository-uri. GitHub este practic o versiune online a unui repository local și astfel oferă posibilitatea de a partaja un repository sau de a adăuga alți programatori care pot contribui în mod independent la dezvoltarea aplicației. Cu alte cuvinte, GitHub extinde funcționalitățile oferite de Git printr-o interfață intuitivă și unelte de management asupra unui repository. Un instantaneu al repository-ului care conține codul-sursă al aplicației "FoodSpy" este reprezentat în (Fig. 5.8). Se poate observa că branch-ul "calculate-calories" a suferit 9 modificări de la momentul în care a fost clonat din branch-ul principal, adică "master".

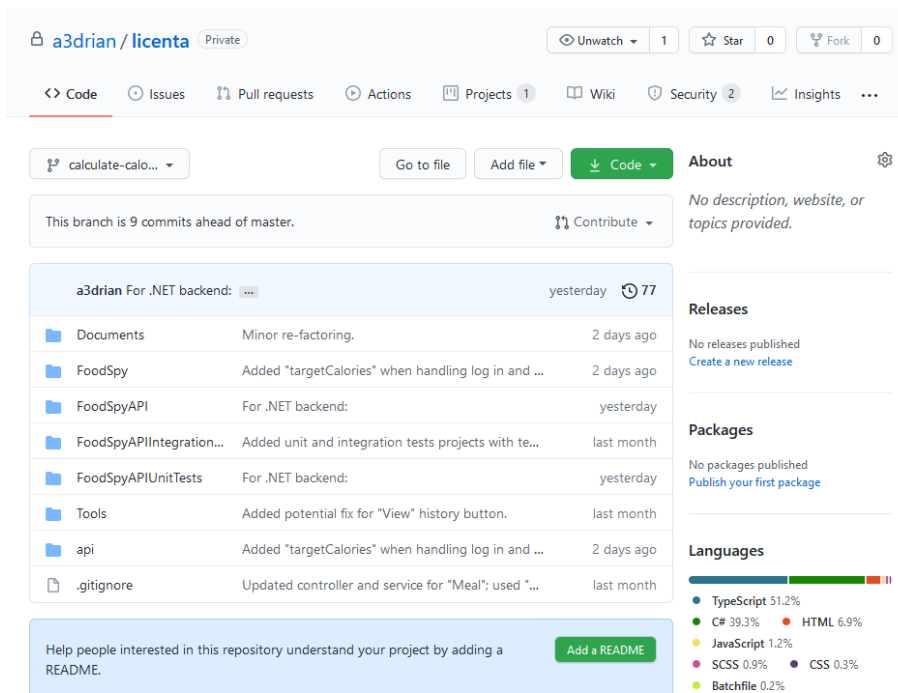


Fig. 5.8: Repository-ul "FoodSpy"

Capitolul 6

Descrierea aplicației

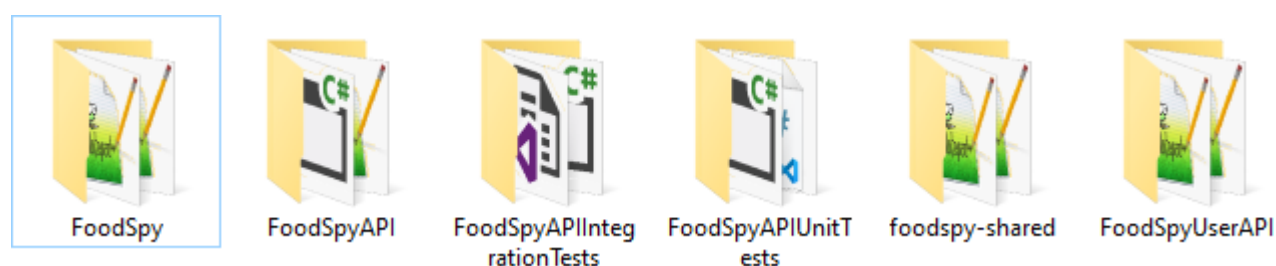


Fig. 6.1: Arhitectura aplicației FoodSpy

Arhitectura aplicației FoodSpy presupune o implementare realizată pe mai multe nivele de infrastructură, nivele ce sunt detaliate în rândurile următoare.

Proiectul "FoodSpy" reprezintă nivelul clientului sau nivelul de prezentare, adică interfața grafică prin care utilizatorul interacționează cu aplicația.

"FoodSpyUserAPI" este interfața de programare care efectuează comunicarea cu serverul care se ocupă de administrarea utilizatorilor și a datelor acestora.

Proiectul care se ocupă de persistența datelor reținute în jurnalul fiecărui utilizator este "FoodSpyAPI". Acesta este responsabil pentru stocarea informațiilor legate de mâncărurile pe care utilizatorii le pot folosi pentru a-și compune mesele și pentru a înregistra aportul caloric în jurnal.

Nivelul de testare este acoperit de proiectele "FoodSpyAPIIntegrationTests" și "FoodSpyAPIUnitTests".

"foodspy-shared" este proiectul care reprezintă nivelul auxiliar, un nivel care face posibilă partajarea dependențelor aplicației. Acesta conține o colecție de enumerări și interfețe care sunt folosite atât la nivelul clientului cât și la nivelul interfețelor de programare (API-urilor).

6.1 Clientul

Nivelul clientului este împărțit pe mai multe componente Angular.

"Auth" constituie componenta care administrează pagina de autentificare în aplicație. La baza acestei componente este un formular HTML. Pentru un utilizator care urmează să își creeze un cont, formularul are 3 câmpuri: e-mail-ul, parola și aportul zilnic de calorii pe care acesta dorește să-l atingă în fiecare zi. Pentru e-mail se folosește o validare bazată pe o expresie regulată care impune în componența sa existența unui domeniu (".com", ".ro", etc.). Parola trebuie să aibă un număr minim de caractere, număr definit de o constantă, iar aportul este exprimat ca un număr întreg cuprins într-un anumit interval.

Pentru cazul în care utilizatorul are deja cont și efectuează operația de autentificare în aplicație, formularul nu mai afișează câmpul numărului de calorii. Se observă astfel faptul că același formular este folosit pentru procesul de înregistrare, dar și pentru procesul de autentificare. Această implementare este posibilă folosind directiva "*ngIf".

```
<form [formGroup]="authForm" (ngSubmit)="onSubmit()" *ngIf="!isLoading">

  <div id="formFields">
    <div [ngClass]="emailValid() ? 'validField': 'invalidField'">
      <input [ngClass]="emailValid() ? 'valid': 'invalid'"
        formControlName="email" type="email" name="email"
        placeholder=" " required />
      <label class="label" for="email">E-Mail</label>
    </div>

    <div class="field">
      <input class="input" formControlName="password"
        [type]="showPassword ? 'text' : 'password'" name="password"
        placeholder=" " required minlength="{{defaultPasswordLength}}" />/.input
      <label class="label" for="password">Password</label>
      <button type="button" class="toggle-password" (click)="onShowPassword()"
        [disabled]="!hasTypedPassword()">
        <mat-icon{{ showPassword ? 'visibility_off' : 'visibility' }}</mat-icon>
      </button>/.toggle-password
    </div>/.field

    <div class="field" *ngIf="!isInLoginMode">
      <input class="input" formControlName="targetCalories" type="number" id="targetCalories"
        name="targetCalories" placeholder=" "
        min="{{defaultMinCalories}}" max="{{defaultMaxCalories}}" required />
      <label class="label" for="targetCalories">Target calories</label>
    </div>/.field
  </div>/#formFields

  <div id="formButtons">
    <button class="formButton" type="submit" [disabled]="!isFormValid()"
      {{ isInLoginMode ? loginButtonText : registerButtonText }}
    </button>/.formButton
    <button class="formButton" type="button" (click)="onSwitchBetweenLoggedInAndOut()"
      {{ isInLoginMode ? noAccountButtonText : existingAccountButtonText }}
    </button>/.formButton
  </div>/#formButtons
</form>
```

Fig. 6.2: Formularul de înregistrare și autentificare în aplicație

"Auth" utilizează serviciul de autentificare "AuthService". Acesta validează datele introduse în formular și trimite un request "POST" către interfața de programare care se ocupă de administrarea utilizatorilor. Request-ul "POST" este împachetat într-o interfață "IAuthResponseData" compusă din proprietățile "email", "password" și "targetCalories". Interfața este partajată cu "FoodSpyUserAPI" pentru a facilita despachetarea cu ușurință a informațiilor conținute în request-ul "POST".

```
1 export interface IAuthResponseData {  
2     id: string;  
3     email: string;  
4     password: string;  
5     token: string;  
6     expiresIn: number;  
7 }
```

Fig. 6.3: Interfața "IAuthResponseData"

După înregistrare sau autentificare, utilizatorul este redirecționat către componenta "Intakes". "Intakes" reprezintă tabloul de bord al aplicației și afișează numărul de calorii pe care acesta și le-a setat ca obiectiv de atins la momentul înregistrării. De asemenea, oferă posibilitatea de a urmări istoricul utilizării aplicației, pe zile, ordonate de la cea mai îndepărtată zi, la cea mai apropiată. Acest istoric reprezintă de fapt jurnalul aportului de calorii al utilizatorului.

```
28 <div class="intakes-section" *ngIf="!errorMessage && canShowIntakesCards()">  
29     <app-intakes-list [intakes]="intakes"></app-intakes-list>  
30     <app-add-intake-card></app-add-intake-card>  
31 </div>/.intakes-section
```

Fig. 6.4: Componenta "Intakes"

Obiectul "intakes" de la linia 29 din (Fig. 6.4) este populat cu rezultatul unui request "POST" către "FoodSpyAPI" care întoarce o listă de intrări din jurnalul unui anume utilizator, identificat prin adresa acestuia de e-mail. Nu se face un request "GET" deoarece acest endpoint din (Fig. 6.5) permite sortarea intrărilor din jurnal, din punct de vedere cronologic, dacă se populează o proprietate numită "sortOrder" cu valoarea 0 pentru ascendent sau cu valoarea 1 pentru descendent.

În tabloul de bord, fiecare dintre aceste zile care face parte din istoric sunt conținute într-o componentă numită "IntakeCard". În "IntakeCard", istoricul nu este detaliat, afișează doar procentul de completare al aportului de calorii, și cantitatea totală de grăsimi, carbohidrați și zaharuri pe care utilizatorul le-a consumat în ziua respectivă.

Detaliile precum mâncărurile care fac parte din componența unei mese și cantitatea fiecăruia dintre mâncăruri consumate de utilizator sunt afișate în componenta "IntakeHistory", la care se ajunge prin intermediul unui hyperlink. Hyperlink-ul este atașat unei element "h2" prin evenimentul "(click)", după cum se poate observa în (Fig. 6.6). Elementul "h2" exprimă numărul de mese consumate de utilizator într-o anumită zi. "Intakes" face legătură către "AddMeal" prin componenta "AddIntakeCard". Cu ajutorul "AddMeal", un utilizator poate căuta mâncărurile pe care să le adauge la o masă, iar masa o poate adăuga la jurnalul din ziua respectivă.

"AddMeal" este cea mai complicată dintre componente, deoarece ea implementează funcționalitatea pentru căutarea mâncărurilor în baza de date și construcția unui obiect de tip "Intake", care reprezintă o zi din jurnal.

Când utilizatorul accesează componenta "AddMeal", se inițializează obiectul de tip "In-

```
85 // Get intakes by e-mail
86 getIntakesByEmail(
87     email: string
88 ): Observable<IIntake> {
89     const request = this.http
90         .post<IIntake>(
91             this.SEARCH_URL,
92             new SearchByEmail(
93                 {
94                     email: email
95                 }
96             )
97         )
98         .pipe(
99             tap(
100                 () => { }
101             )
102         );
103
104     return request;
105 }
```

Fig. 6.5: Request "POST" către "/api/db/intakes/search/"

```
37 | <h2 (click)="viewIntakeDetails(intake)">Meals: {{ intake.meals.length }}</h2>
```

Fig. 6.6: Componenta "Intakes"

take" cu valori default. De exemplu, o proprietate a acestui obiect este "createdAt", care va reține momentul la care s-a adăugat în jurnal prima masă. Se mai inițializează proprietatea "email" cu e-mail-ul utilizatorului și "targetCalories" cu obiectivul setat de utilizator.

Căutarea mâncărilor din baza de date se face folosind un "<input>" de tip "text" atașat la o funcție care preia termenul de căutare introdus de către utilizator și face un request "GET" către "FoodsService". Termenul de căutare populează parametrul din URL numit "name" și asta va determina endpoint-ul definit în "FoodSpyAPI" să returneze o listă de mâncăruri filtrate după nume. Un exemplu de request "GET" după criteriul numelui este "/api/db/foods/search?name=ciocolata".

```
219     private initializeIntake() {  
220         const initialMealIDs: string[] = [];  
221         const initialMeals: IMeal[] = [];  
222         this.intake = new Intake({  
223             email: this.userEmail,  
224             mealIDs: initialMealIDs,  
225             meals: initialMeals,  
226             createdAt: this.today,  
227             targetCalories: this.userTargetCalories,  
228             calories: 0  
229         });  
230     }
```

Fig. 6.7: Inițializarea componentei "Intakes"

```
<div id="formFields" *ngIf="!errorMessage">  
  <div class="field" *ngIf="canSearch()">  
    <input class="input" #searchBox type="text" id="search-box" name="searchBox"  
      (input)="search(searchBox.value)" placeholder=" " pattern="[A-Za-z0-9 ]+" />  
    <label class="label" for="searchBox">Look for food...</label>  
  </div>/.field  
</div>/#formFields
```

Fig. 6.8: Bara de căutare a mâncărilor după nume

Odată găsită mâncarea în baza de date, utilizatorul poate deschide o căsuță de dialog, care reprezintă componenta "EditFoodDialogue". Aici, utilizatorul poate vedea mai multe detalii despre mâncarea selectată în urma căutării, printre care cantitatea de săruri sau proteine și de asemenea poate introduce cantitatea de mâncare, exprimată în grame.

Pentru cantitate este construită o validare care impune valoarea acesteia să nu poată depăși 1000 de grame. Validarea este ilustrată în (Fig. 6.9). Aceasta poartă numele de "foodQuantityValidator" și este implementată folosind clasa abstractă "AbstractControl". "AbstractControl" permite accesul la valoarea introdusă de utilizator în formular și astfel se pot impune restricții asupra acesteia.

După adăugarea unei mâncăruri, utilizatorul poate selecta tipul de masă pe care acesta a consumat-o. Tipurile de masă sunt "Breakfast", "Lunch", "Dinner" sau "Snack". La momentul alegerii unuia dintre tipuri, formularul este valid și utilizatorului i se permite să înregistreze masa în jurnal.

Când se înregistrează o masă, există două scenarii.

Primul scenariu este acela în care utilizatorul nu are nicio intrare în jurnalul zilei respective, iar al doilea scenariu reprezintă modificarea unui jurnal deja existent. În primul scenariu, implementarea este simplă deoarece este suficient să adăugăm masa în colecția "Meals" din baza de date și apoi să introducem ID-ul acestei mese în proprietatea "mealIDs" a obiectului de tip "Intake". Ultimul pas este să trimitem un request "POST" către "FoodSpyAPI" pentru a înregistra jurnalul în colecția "Intakes" din baza


```
4 export function foodQuantityValidator(): ValidatorFn {
5   return (control: AbstractControl): ValidationErrors | null => {
6
7     const value = control.value;
8
9     if (!value) {
10      return null;
11    }
12
13    let validType: boolean = false;
14    let validValue: boolean = false;
15    if (typeof value === 'number') {
16      validType = true;
17      if (
18        value >= Constants.MIN_MEAL_QUANTITY &&
19        value <= Constants.MAX_MEAL_QUANTITY
20      ) {
21        validValue = true;
22      }
23    }
24
25    const valid = validType && validValue;
26
27    return !valid ? { positiveInteger: true } : null;
28  }
29 }
```

Fig. 6.9: Validarea pentru cantitatea de mâncare

de date.

Al doilea scenariu este mai complicat deoarece în primul rând trebuie să determinăm dacă această masă nou adăugată face parte dintr-un tip de masă care există deja în în obiectul "Intake". Dacă nu face parte, se persistă o masă nouă și se aduce referința în proprietatea "mealIDs" prin ID, iar în final se execută un request "PUT" pentru a modifica un "Intake" deja existent. Dacă face parte, atunci trebuie identificată colecția de mese care are același tip cu tipul mesei pe care utilizatorul dorește să o adauge. După identificarea acesteia, se concatenează colecția existentă cu masa ce urmează să fie adăugată, se aduce referința (ID-ul) în obiectul "Intake" și din nou se modifică jurnalul (obiectul de tip "Intake") prin același request "PUT".

După executarea oricăruia dintre cele 2 scenarii descrise mai sus, se reîncarcă pagina pentru a reflecta modificarea din obiectul "Intake".

Prezent deasupra tuturor celorlalte componente, "Header" reprezintă antetul aplicației. Acesta afișează e-mail-ul utilizatorului autentificat, logo-ul aplicației și un buton pe care utilizatorul îl poate folosi pentru deconectare și redirectionare la componenta "Auth".

6.2 API pentru administrarea utilizatorilor

Interfața de programare pentru administrarea utilizatorilor pune la dispoziție două endpoint-uri, un endpoint pentru operația de înregistrare a unui nou utilizator și unul pentru procesul de autentificare în aplicație.

```

async onSubmit(): Promise<void> {
  this.addTypeAndCreatedAt();
  let editMeal: boolean = false;
  if (this.existingMealInIntake) {
    // need to check against meal types => edit Meal
    if (this.existingMealTypes.includes(this.selectedMealType)) {
      // same meal type => edit Meal => embed in Intake => edit Intake
      // need to loop through all existing meals to find the one which matches our (soon to be added) new meal
      await this.editMealFromIntake();
      editMeal = true;
    } else {
      // different meal type => new Meal => embed in Intake => edit Intake
      await this.addNewMealToIntake();
    }
  } else {
    // need to add a new meal type => new Meal => embed in Intake => new Intake
    await this.addNewMealToIntake();
  }
  if (editMeal) {
    // no need to modify the existing Intake because it already contains a reference to the edited Meal => reload page
    this.reloadPage(this.onSubmit.name);
  } else {
    if (this.existingIntake) {
      this.intakesService
        .editIntake(this.intake)
        .subscribe();
    } else {
      this.intakesService
        .addIntake(this.intake)
        .subscribe();
    }
    this.reloadPage(this.onSubmit.name);
  }
}
}

```

Fig. 6.10: Adăugarea unei intrări în jurnalul zilnic

```

3   <div id="wrapper" class="mainDiv" *ngIf="isAuthenticated">
4
5     <div>
6       <p id="first" class="elem">Welcome, {{ userEmail }}!</p>
7     </div>
8
9     <div>
10      <p id="second" class="elem">FoodSpy</p>
11    </div>
12
13    <div>
14      <button id="third" class="elem" (click)="onLogout()">
15        Logout
16      </button> /#third.elem
17    </div>
18
19  </div> /#wrapper.mainDiv
20
21  <div id="wrapper" class="mainDiv" *ngIf="!isAuthenticated">
22    <h3 class="elem">Welcome!</h3>
23    <h3 class="elem">FoodSpy</h3>
24    <button class="elem" (click)="login()">
25      Log in
26    </button> /.elem
27  </div> /#wrapper.mainDiv

```

Fig. 6.11: Componenta "Header"

La baza API-ului stă MikroORM, biblioteca TypeScript care implementează tehnica de

a interoga și manipula datele dintr-o bază de date folosind o paradigmă orientată pe obiect. Cu ajutorul MikroORM, se definește o entitate care modelează profilul unui utilizator, dar și un manager peste această entitate care permite înregistrarea sau autentificarea acestuia în aplicație.

Pentru înregistrarea unui utilizator este nevoie să definim un endpoint pentru verbul "POST" cu ajutorul unei rute. Din (Fig. 6.12), se poate observa că ruta folosită pentru procesul de înregistrare este ruta rădăcină, adică `"/api/auth/register"`.

```
16 function setRegisterRoute(router: Router): Router {  
17     router.post('/', registerUser);  
18     return router;  
19 }
```

Fig. 6.12: Ruta pentru înregistrare

"registerUser" este funcția care face un apel către "getUserByEmail" din "UserService". Funcția respectivă întoarce imediat o eroare dacă din corpul request-ului "POST" lipsește e-mail-ul utilizatorului. Altfel, folosind manager-ul peste entități oferit de MikroORM, se caută utilizatorul identificat de e-mail-ul introdus în formularul din componenta "Auth" din client. Dacă se găsește utilizatorul, detaliile despre acesta se returnează într-un obiect "User", iar dacă nu există, funcția întoarce "null".

În (Fig. 6.13) este ilustrată funcția "getUserByEmail".

La momentul înregistrării, este important ca "getUserByEmail" să returneze "null". Dacă returnează "null", înseamnă că utilizatorul nu există în baza de date și se poate înregistra cu succes folosind e-mail-ul, parola și obiectul caloric propus. Parola este criptată folosind biblioteca "bcrypt" pentru a nu fi persistată în baza de date în format "plain text".

Dacă "getUserByEmail" nu returnează "null", înseamnă că utilizatorului trebuie să îi fie afișat un mesaj de eroare care să-l atenționeze că există deja un cont creat cu același e-mail.

Pentru autentificarea unui utilizator este nevoie să definim un endpoint pentru verbul "POST" cu ajutorul unei rute. Din (Fig. 6.15), se poate observa că ruta folosită pentru procesul de autentificare este ruta rădăcină, adică `"/api/auth/login"`.

Asemenea funcției "registerUser", funcția "loginUser" face un apel către "getUserByEmail" din "UserService".

La momentul autentificării, această funcție trebuie să returneze un obiect "User", care conține datele despre utilizator. În caz contrar, există scenariul în care utilizatorul nu există și trebuie mai întâi să se înregistreze sau există scenariul în care utilizatorul se află în baza de date. În acest caz, parola introdusă în formular este comparată cu parola persistată, iar dacă nu există o potrivire, un mesaj de eroare este afișat în client.

La momentul pornirii server-ului care administrează datele despre utilizator, se caută un fișier numit "env" care conține string-ul de conectare la baza de date MongoDB precum și portul la care ascultă pentru cererile venite din partea clientului. "env" este interpretat de către API, care identifică string-ul de conectare și deschide portul de comunicare pentru client.

```
57     if (!email || typeof email !== 'string') {
58         return Error('invalid params');
59     }
60
61     console.log('');
62     log(CLASS_NAME, getUserByEmail.name, '');
63     log(CLASS_NAME, getUserByEmail.name, 'email:', email);
64
65     try {
66
67         const user = await em
68             .findOne(
69                 User,
70                 { email: email }
71             );
72
73         log(CLASS_NAME, getUserByEmail.name, 'user:', user);
74         log(CLASS_NAME, `${getUserByEmail.name}^`, '');
75         console.log('');
76
77         return user;
78     } catch (ex) {
79         return ex;
80     }
81 }
```

Fig. 6.13: Funcția "getUserByEmail"

```
7  function encryptPassword(password: string): string {
8      const hash = bcrypt.hashSync(password, SALT_ROUNDS);
9      return hash;
10 }
```

Fig. 6.14: Criptarea parolei folosind "bcrypt"

```
21 function setLoginRoute(router: Router): Router {
22     router.post('/', loginUser);
23     return router;
24 }
```

Fig. 6.15: Ruta pentru autentificare

Fișierul ".env" nu este înregistrat în repository-ul aplicației deoarece conține informații sensibile.

```
let response: Error | User | null;
try {

  const body = req.body;

  response = await userService
    .getUserByEmail(
      req.em,
      body.email
    );

  if (!response) {
    const userDoesNotExistError: Error = new Error;
    userDoesNotExistError.name = ERROR_MESSAGES.NO_ACCOUNT;
    userDoesNotExistError.message = 'No account found!';
    return res.status(STATUS_CODES.NOT_FOUND).json(userDoesNotExistError);
  }

  if (response instanceof Error) {
    return next(response);
  }

  if (response instanceof User) {
    const formPassword = body.password;
    const storedPassword = response.password;
    const match = decryptPassword(formPassword, storedPassword);
    if (match) {
      let authResponseData: IAuthResponseData = new AuthResponseData;
      authResponseData.email = response.email;
      authResponseData.id = response.id;
      authResponseData.targetCalories = response.targetCalories;
      return res.status(200).json(authResponseData);
    } else {
      const wrongPasswordError: Error = new Error;
      wrongPasswordError.name = ERROR_MESSAGES.WRONG_PASSWORD;
      wrongPasswordError.message = 'Wrong e-mail or password!';
      return res.status(STATUS_CODES.UNAUTHORIZED).json(wrongPasswordError);
    }
  }
} catch (ex) {
  return next(ex);
}
```

Fig. 6.16: Corpul funcției "loginUser"

6.3 API pentru gestiunea jurnalelor utilizatorilor

Structura proiectului "FoodSpyAPI" este reprezentată în (Fig. 6.16).

Directorul "Common" conține clase și enumerări, precum "Alphabet" sau "MealType". În "Alphabet" există totalitatea caracterelor permise în aplicație, o reuniune a caracterelor alfabetului englez împreună cu literele cu diacritice din alfabetul limbii române. "Alphabet" conține și o funcție "ConvertDiacritic" care este utilizată pentru a îndepărta diacriticele dintr-un caracter.

"MealType" este o clasă statică care are în componența sa enumerarea "MealType" folosită pentru definirea tipurilor de masă permise în aplicație. De asemenea, "Meal-

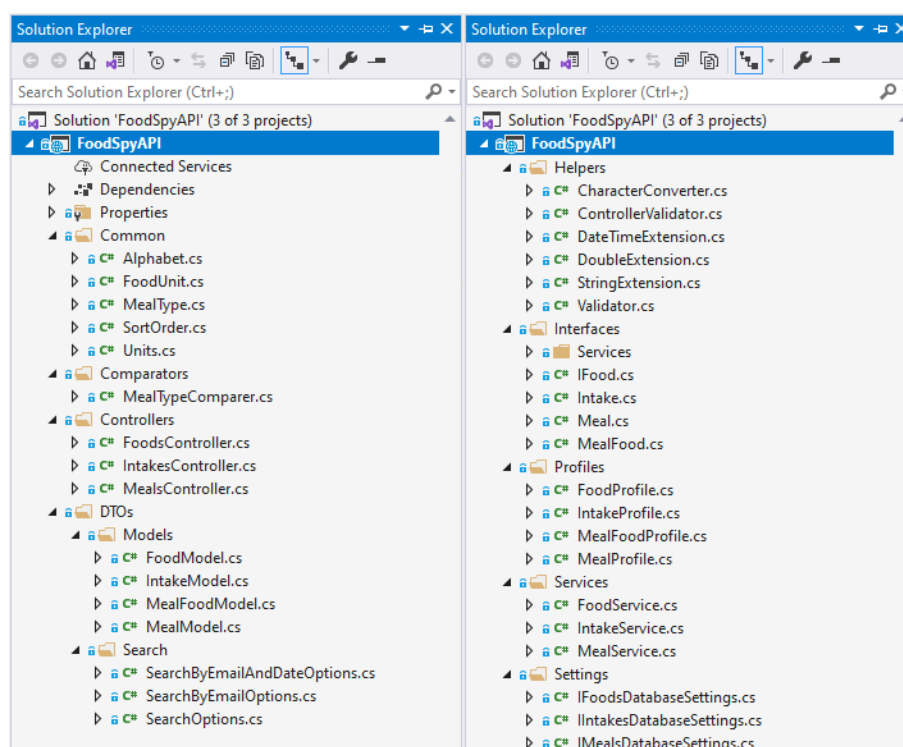


Fig. 6.17: Arhitectura proiectului "FoodSpyAPI"

Type" definește funcția "GetMealTypesOrder" care are ca parametru de retur un "Dictionary<string, uint>". Dicționarul are ca utilitate sortarea meselor în funcție de tipul acestora, "Breakfast" primul, "Lunch" al doilea, "Dinner" al treilea și "Snack" ultimul.

```

16 public static Dictionary<string, uint> GetMealTypesOrder()
17 {
18     Dictionary<string, uint> order = new Dictionary<string, uint>()
19     {
20         { MealType.Breakfast.ToString(), (int)MealType.Breakfast },
21         { MealType.Lunch.ToString(), (int)MealType.Lunch },
22         { MealType.Dinner.ToString(), (int)MealType.Dinner },
23         { MealType.Snack.ToString(), (int)MealType.Snack }
24     };
25
26     return order;
27 }

```

Fig. 6.18: Corpul funcției "GetMealTypesOrder"

Dicționarul este folosit în funcția "Compare" a clasei "MealTypeComparer" din directorul "Comparators". În funcție de ponderea tipului de masă, clasa poate sorta mesele dintr-un jurnal în ordine crescătoare sau descrescătoare.

Interfața de programare pentru gestiunea jurnalelor utilizatorilor este compus din trei controller-e, "IntakesController", "MealsController" și "FoodsController", din directorul "Controllers". Fiecare dintre acestea trei are definite endpoint-uri pentru verbele "GET", "GET" cu parametrul din URL "id", "POST", "PUT" și "DELETE".

"GET" returnează lista tuturor obiectelor de același tip cu tipul reprezentat de controller ("Intake", "Meal" sau "Food"). "GET" returnează un singur obiect identificat de parametrul "id" din URL. "POST" este folosit pentru adăugarea unui nou obiect, "PUT"

```
8      0 references
      internal class MealTypeComparer : IComparer<Meal>
9      {
10         private readonly Dictionary<string, uint> order = MealTypes.GetMealTypesOrder();
11
12         0 references
13         public int Compare(Meal x, Meal y)
14         {
15             string typeX = x.Type;
16             string typeY = y.Type;
17
18             uint a = order[typeX];
19             uint b = order[typeY];
20
21             return a.CompareTo(b);
22         }
23     }
```

Fig. 6.19: Implementarea comparării tipurilor de mese

pentru editarea unui obiect deja existent, iar "DELETE" pentru ștergerea unui obiect din baza de date.

Pentru implementarea anumitor endpoint-uri este nevoie mai mult de unul din aceste verbe. Spre exemplu, pentru "PUT" trebuie mai întâi să inițiem un request "GET" după parametrul "id" din URL. Rezultatul acestui request va fi un obiect care ulterior poate fi modificat cu ajutorul request-ului "PUT".

Utilizatorul poate filtra mâncărurile din baza de date după criteriul numelui. De această filtrare se ocupă "FoodsController" și funcția "SearchFoodsByName", ilustrată în (Fig. 6.20). Această funcție primește ca parametru un obiect de tip "string" care reprezintă termenul de căutare primit de la client. Cu ajutorul acestei funcții, a verbului "GET" și a rutei "/api/db/foods/search" se definește endpoint-ul care filtrează mâncărurile.

```
176      [HttpGet("search")]
177      0 references
178      public async Task<ActionResult<List<FoodModel>>> SearchFoodsByName(string name)
179      {
180          try {
181              if (!Validator.IsValidFoodName(name)) {
182                  return BadRequest("Food name is invalid!");
183              }
184
185              string convertedName = CharacterConverter.ConvertDiacritics(name);
186
187              List<Food> searchResults = await _foodService.SearchFoodsByName(convertedName);
188              List<FoodModel> mappedFoods = _mapper.Map<List<FoodModel>>>(searchResults);
189              return mappedFoods;
190          } catch (Exception e) {
191              return LogDatabaseException(e);
192          }
193      }
194  }
```

Fig. 6.20: Corpul funcției "SearchFoodsByName"

Se poate observa că prima oară se validează numele introdus de către utilizator și se returnează ca răspuns un "status code" cu numărul 400, așa numitul "BadRequest" și mesajul de eroare "Food name is invalid!", dacă numele nu este valid.

În următorul pas, se îndepărtează diacriticele și se apelează "SearchFoodsByName" din "FoodService". Aici, se construiește un filtru care nu ține cont de stilul de scriere a numelui ("case insensitive") și se interoghează baza de date. Rezultatul sau rezultatele

căutării sunt returnate pentru a putea fi afișate utilizatorului și pentru a-i permite să adauge mâncarea la o masă din jurnal.

Dacă nu sunt rezultate în urma căutării, se returnează o listă goală pe care clientul o interpretează cu un mesaj de informare și anume "No foods found".

```
116 public async Task<List<Food>> SearchFoodsByName(string name)
117 {
118     string SEARCH_BY_NAME = nameof(Food.Name);
119
120     FilterDefinition<Food> nameFilter = Builders<Food>
121         .Filter
122         .Regex(SEARCH_BY_NAME, new BsonRegularExpression(name, "i"));
123
124     IAsyncCursor<Food> foods = await _foods
125         .FindAsync<Food>(
126             filter: nameFilter
127         );
128     List<Food> foodsList = foods.ToList();
129
130     return foodsList;
131 }
```

Fig. 6.21: Funcția "SearchFoodsByName" din "FoodService"

În directorul "DTOs" sunt așa numitele "data transfer objects". Aceste obiecte sunt folosite strict pentru comunicarea între API și client. Acestea sunt o versiune simplificată a modelelor propriu-zise de date pentru a nu îngreuna transferul datelor cu proprietăți (informații) care nu sunt absolut necesare.

De exemplu, clasa care modelează o masă se numește "Meal" și are definite proprietățile din (Fig. 6.22). Din obiectul de transfer "MealModel", lipsește proprietatea "List<Food> Foods", după cum se poate observa din (Fig. 6.23).

Tot în directorul "DTOs" există clase care facilitează construirea unor request-uri "POST" pentru filtrarea obiectelor de tip "Intake", spre exemplu după e-mail și după o anumită dată, cum este cazul clasei "SearchByEmailAndDateOptions".

Directorul "Helpers" conține clase ajutătoare, precum "CharacterConverter", care poate lua un obiect de tip "string" și îndepărta diacriticele din caracterele care compun respectivul string.

Cea mai des folosită clasă din acest director este "Validator", o clasă statică plină de funcții statice care sunt folosite pentru validare, printre care funcția care verifică validitatea unui "id", funcția care întoarce "false" dacă e-mail-ul introdus de utilizator nu este sub o anumită formă, funcție care trece un string prin alfabetul descris în "Alphabet" și verifică existența unor caractere nepermise și funcția care este folosită pentru a verifica faptul că un anumit tip de masă este prezent în enumerarea "MealType". Tot în directorul "Helpers" sunt metodele de extensie "DateTimeExtension" și "StringEx-


```
public class Meal : IMeal
{
    [BsonId]
    [BsonElement(nameof(Id))]
    15 references
    public Guid Id { get; set; }

    [Required]
    [BsonElement(nameof(Type))]
    13 references
    public string Type { get; set; }

    [Required]
    [BsonElement(nameof(CreatedAt))]
    10 references
    public DateTime CreatedAt { get; set; }

    [Required]
    [BsonElement(nameof(MealFoods))]
    8 references
    public List<MealFood> MealFoods { get; set; }

    [BsonElement(nameof(Foods))]
    4 references
    public List<Food> Foods { get; set; }
}
```

Fig. 6.22: Clasa "Meal"

```
public class MealModel : IMeal
{
    [BsonId]
    [BsonElement(nameof(Id))]
    2 references
    public Guid Id { get; set; }

    [BsonElement(nameof(Type))]
    13 references
    public string Type { get; set; }

    [BsonElement(nameof(CreatedAt))]
    10 references
    public DateTime CreatedAt { get; set; }

    [BsonElement(nameof(MealFoods))]
    2 references
    public List<MealFoodModel> MealFoods { get; set; }
}
```

Fig. 6.23: Clasa "MealModel"

```

180     internal static bool IsValidMealType(string type)
181     {
182         if (!IsValidSearchTerm(type)) {
183             return false;
184         }
185
186         string firstLetterUppercased = type.FirstLetterUppercased();
187
188         bool valid = Enum.IsDefined(typeof(MealType), firstLetterUppercased);
189         if (!valid) { return false; }
190
191         return true;
192     }

```

Fig. 6.24: Funcția de validare a unui tip de masă

tension”.

În “StringExtension” este definită o metodă “FirstLetterUppercased” care se aplică unui string și întoarce același obiect string, dar cu prima literă scrisă de tipar și restul cu literă mică. Această metodă este folosită spre exemplu la validarea unui tip de masă, după cum se poate observa în (Fig. 6.24), la linia 186.

Corpul metodei “FirstLetterUppercased” este ilustrat în (Fig. 6.25).

```

8     public static string FirstLetterUppercased(this string value)
9     {
10         return value.First().ToString().ToUpper() + value[1..].ToLower();
11     }

```

Fig. 6.25: Metoda de extensie “FirstLetterUppercased”

Interfețele care definesc setul minim de proprietăți pe care ar trebui să le aibă în componență modelele “Food”, “Meal” și “Intake” se află în directorul “Interfaces”. De exemplu, interfața “IMeal” este reprezentată în (Fig. 6.26).

```

6     public interface IMeal
7     {
8         10 references
        DateTime CreatedAt { get; set; }
9         13 references
        string Type { get; set; }
10    }

```

Fig. 6.26: Interfața “IMeal”

Clasele care se ocupă de asocierea proprietăților dintre modele și obiectele de transfer

ale datelor se află în directorul "Profiles". "MealProfile" este clasa care conține harta de asociere între "Meal" și "MealModel". Această clasă este ilustrată în (Fig. 6.27).

```
8      public class MealProfile : Profile
9      {
10         0 references
11         public MealProfile()
12         {
13             this.CreateMap<Meal, MealModel>();
14             this.CreateMap<MealModel, Meal>();
15         }
16     }
```

Fig. 6.27: Clasa de asociere "MealProfile"

Fiecare dintre controller-e are asociat un serviciu. Toate aceste servicii sunt reținute în directorul "Services". Serviciile sunt angajate de controller-e pentru a manipula datele din baza de date, în funcție de cererile care vin din partea clientului.

Serviciile mai sunt folosite și pentru a executa anumite operații auxiliare asupra datelor. Spre exemplu, în serviciul "MealService" există funcția "CalculateCalories" care preia ca parametru un obiect de tip "Meal", ia în calcul informațiile nutriționale ale mâncărilor care se află în componența obiectului și calculează numărul de calorii consumate.

"CalculateCalories" este supraîncărcată de o versiune a funcției care primește ca parametru o listă de obiecte de tip "Meal". În (Fig. 6.28) este reprezentată versiunea care primește un singur obiect "Meal" ca parametru.

```
257      public double CalculateCalories(Meal meal)
258      {
259          double calories = 0;
260          List<MealFood> mealFoods = meal.MealFoods;
261          foreach (MealFood mealFood in mealFoods) {
262              Food food = mealFood.Food;
263              double energy = food.Energy;
264              calories += energy;
265          }
266          return calories;
267      }
```

Fig. 6.28: Corpul funcției "CalculateCalories"

Directorul "Settings" conține interfețe și clase care facilitează accesul la colecțiile "Foods", "Meals" și "Intakes" din baza de date. Clasele conțin informații despre numele colecției, string-ul de conexiune și numele bazei de date, în acest caz "FoodSpyDb".

Capitolul 7

Utilizarea aplicației

Capitolul 8

Concluzii

Capitolul 9

Perspective de dezvoltare