RetetaTa.ro – Documentație

Proiect realizat de: *Dinca Andrei-Gabriel*

Github: github.com/Andreid27/RetetaTa.ro

Tehnologia folosita pentru crearea bazei de date: **MySql**.
„**MySql** este un sistem de gestiune a bazelor de date relaționale, produs de compania **MySQL AB**. Este cel mai popular sistem de gestionare de acest fel. Se găsește de obicei alături de limbajul **PHP**, însă cu **MySql** se pot construi aplicații in orice limbaj. ”
-https://ro.wikipedia.org/wiki/MySQL

Am realizat o baza de date in MySQL WORKBENCH cu motor InnoDB (InnoDB este un motor de stocare pentru baza de date MySQL. Acesta este folosit pentru a gestiona tabelele și indexurile în baza de date. InnoDB oferă suport pentru chei externe, care permit legarea între tabele prin intermediul unei chei comune. De asemenea, InnoDB oferă suport pentru tranzacții, ceea ce înseamnă că poți efectua mai multe operații asupra bazei de date într-un singur lot și poți anula întregul lot dacă se întâmplă ceva în timpul procesului.) , ea conținând 5 tabele : **Reteta** si **Ingredient**. Asocierea dintre ele este de tipul M:N , ceea ce presupune crearea unei tabele intermediare , pe care am numit-o **reteta _ingredient_cantitate**(care are o cheie primara compusa). Tabela de asociere se numește **reteta _ingredient_cantitate** deoarece este o asociere între tabela Reteta si tabela **ingredient_cantitate(**este o tabela unde se stochează ingredientul și asociat cu cantitatea sa(gandit așa pentru ca **ingredientul** creat de o persoana sa poate fi folosit de mai multe persoane în diverse cantități asociate și nu fie necesară crearea unei asocieri la fiecare rețetă) + ajută la structura folosită de JPA din Spring Boot). Cea de a 5 a tabelă este tabela **User** care conține informațiile despre utilizator.(necesar deoarece aplicația funcționează pe bază de rețete/ingrediente cu drepturi de CUD doar pentru utilizator care le-a creat, restul utilizatorilor având doar drepturi de Read)

Tabela **reteta** conține următoarele coloane:

- id , de tip BIGINT , cheie primara
- denumire , de tip VARCHAR
- descriere, de tip VARCHAR
- price_range , de tip TINYINT
- calorii, de tip SMALLINT
- autor, de tip BIGINT FK(id, din tabela user)

Tabela **ingredient** conține următoarele coloane :

- id , de tip BIGINT , cheie primara
- denumire , de tip VARCHAR
- descriere, de tip VARCHAR
- autor, de tip BIGINT FK(id, din tabela user)

Tabela **ingredient_cantitate** student conține următoarele coloane :

- id , de tip BIGINT , cheie primara
- cantitate , de tip INT
- ingredient_id , de tip BIGINT FK(id, din tabela ingredient)

Tabela **user** student conține următoarele coloane :

- id , de tip BIGINT , cheie primara
- nume , de tip VARCHAR
- prenume , de tip VARCHAR
- active, tip BIT
- mail, de tip VARCHAR
- password , de tip VARCHAR
- phone_number , de tip VARCHAR
- username , de tip VARCHAR

Tabela **ingredient_cantitate** student conține următoarele coloane :

- reteta_id , de tip BIGINT , cheie primara compusă
- ingredient_cantitate_id , de tip BIGINT , cheie primara compusă

Datorita asocierii M:N , **reteta** si **ingredient_cantitate** sunt chei străine(FK) pentru tabela **student .** Cheile primare(PK) corespunzătoare fiecarei tabele au fost setate cu următoarele proprietăți : not null (NN) si auto-increment (AI).

Cheie primara: Una sau mai multe coloane ale caror valori identifica in mod unic toate liniile unui table

-http://etutoriale.ro/articles/100/1/Introducere-in-SQL/

Diagrama asociata tabelelor este reprezentata in figura de mai jos :

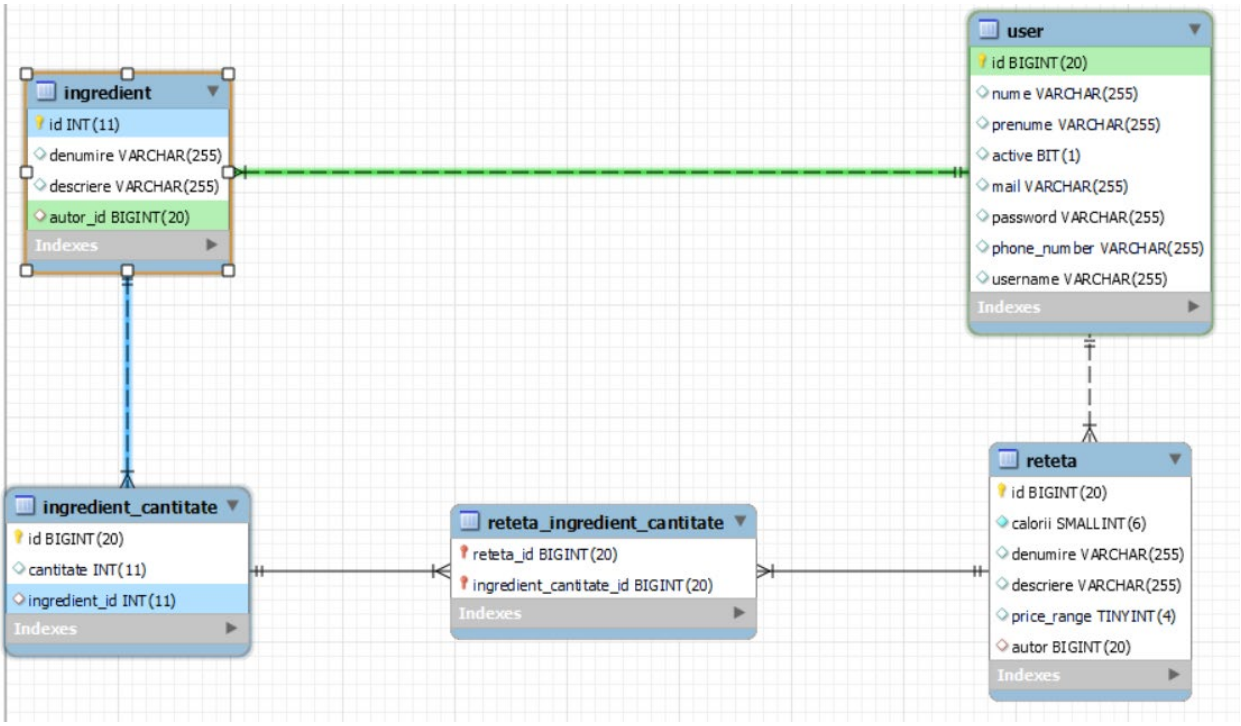


Figura 1 : diagrama asociata tabelelor

Tehnologii utilizate

Pentru interfața am folosi **Frontend** *(React, Redux-Toolkit(state manager tool), Package manager(npm, yarn), Material-UI, Fuse Framework)* și **Backend** *(Spring Framework, Spring Boot 3.0(care practic integrează și Hibernate) și Spring Security 3.0 cu JWT) și cum am descris mai sus MySQL pentru baza de date)*. In acest mod putem edita si vizualiza tabelele.

Link-ul proiectului pe Github: github.com/Andreid27/RetetaTa.ro

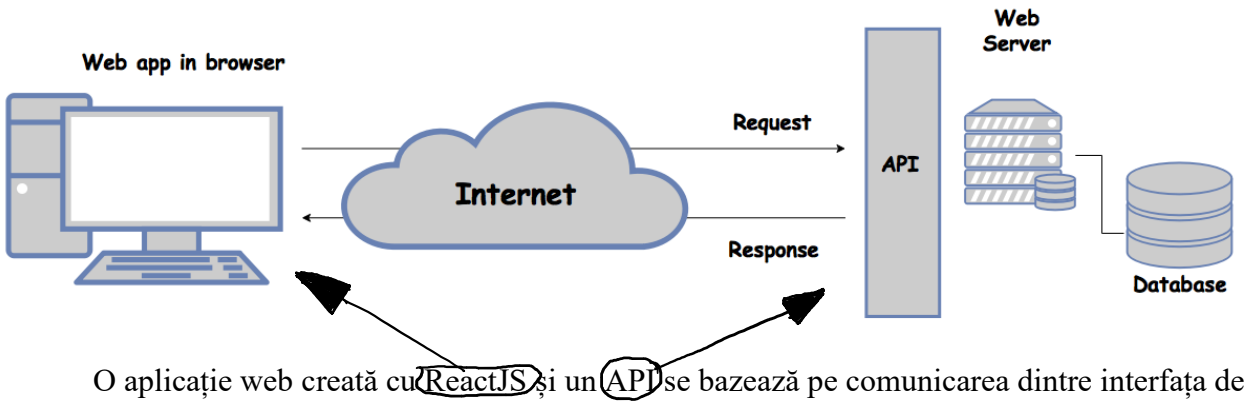
Spring Boot este o platforma Java open-source, bazată pe cadrul Spring Framework, care oferă un mediu de dezvoltare rapid și simplu pentru aplicațiile web și microservicii. Printre beneficiile acestei platforme se numără automatizarea configurării, facilitarea utilizării de către dezvoltatori a unor instrumente și biblioteci, precum și posibilitatea de a crea aplicații web scalabile și cu performanță ridicată. [1]

Spring Framework oferă mai multe avantaje precum: Spring Bean Factory și Dependency Injection care permit dezvoltatorilor să gestioneze obiectele în aplicație prin intermediul unui mecanism centralizat. Acest lucru îmbunătățește scalabilitatea și flexibilitatea aplicației. Spring Bean Factory permite dezvoltatorilor să gestioneze viața(lifespan-ul) obiectelor și să folosească Dependency Injection automat, ceea ce îmbunătățește testabilitatea și întreținerea

aplicației. Dependency Injection permite dezvoltatorilor să evite instanțierea repetată a obiectelor care îngreunează rularea aplicație, ci în schimb permitea reutilizarea eficientă a instanțelor deja existente, ceea ce îmbunătățește eficiența.

ReactJS este un cadru JavaScript open-source, bazat pe conceptul de componente, care permite dezvoltatorilor să creeze interfețe de utilizator interactive și dinamice pentru aplicațiile web. ReactJS utilizează un sistem de actualizare a datelor care se numește Virtual DOM (Document Object Model) care ține evidența schimbărilor în UI și permite actualizarea eficientă a acestora. Acest lucru permite dezvoltatorilor să creeze aplicații web cu performanță ridicată și flexibilitate în modificarea interfaței utilizatorului**.**

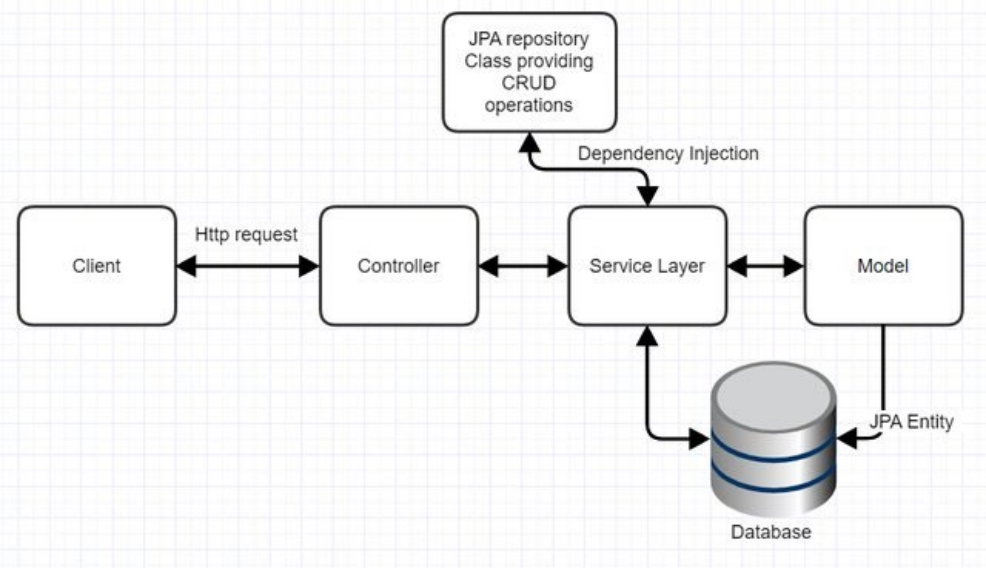
Modul de funcționare: API + Interfață web (aplicație ReactJS)



O aplicație web creată cu ReactJS și un API se bazează pe comunicarea dintre interfața de utilizator creată cu ReactJS și serviciile oferite de API. Interfața pentru utilizator ReactJS se ocupă cu afișarea datelor și cu gestionarea interacțiunilor utilizatorilor, iar API-ul este responsabil cu procesarea cererilor și returnarea răspunsurilor corespunzătoare. ReactJS permite dezvoltatorilor să creeze componente de interfață de utilizator individuale și să le combine pentru a construi interfața completă, iar API-ul permite accesul la bazele de date sau la alte servicii esențiale pentru funcționarea aplicației.

1) BACKEND

API-ul RetetaTa.ro este construit după regulile arhitecturii Spring MVC(Model-View-Controller). Spring MVC o așa zisă arhitectură pentru Spring Boot care permite separarea logicii aplicației, afișării datelor și controlului fluxului de date, într-un mod organizat și scalabil. Acesta oferă funcționalități precum gestionarea rutelor, cererilor și răspunsurilor, precum și suport pentru diferite formate de date. API-ul implementează logica din figura urmatoare:



Vom lua fiecare layer al MVC și îl vom analiza. Pe parcurs vom observa că fiecare din layere descrise în Spring MVC se vor regăsi în pachete separate.

Primul layer se numește Security Filter, el face parte din Spring Security 3.0. Spring Security este un modul din cadrul Spring Framework, care oferă mecanisme de securitate pentru aplicațiile web. Acesta funcționează prin intermediul unei lanțuri de filtre de securitate (Security Filter Chain), care sunt aplicate pe toate cererile către aplicație(practic interceptează toate cererile către API). Fiecare filtru din lanțul de securitate verifică dacă utilizatorul este autorizat să acceseze anumite resurse sau să execute anumite acțiuni. Autentificarea utilizatorilor este gestionată de un AuthenticationManager, care validează și autorizează utilizatorii bazat pe diferite metode de autentificare, cum ar fi autentificarea prin intermediul unui nume de utilizator și parolă sau prin intermediul unui token. Tot aici setăm și endpoint-urile deschise fără autentificare(ca de exemplu /login, etc). În figura de mai jos este explicat în mod simplist cum funcționează interceptarea request-urilor și verifică dacă utilizatorul este logat sau nu. În cazul în care nu este logat nu îi permite accesul la resursele cerute în request. În această aplicație s-a folosit JWT ca metoda de autentificare.

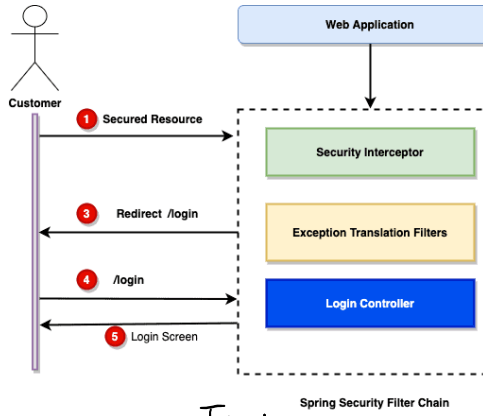


Fig.1.

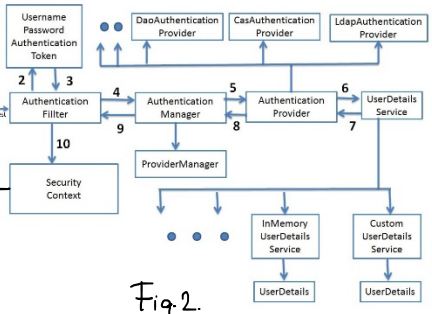
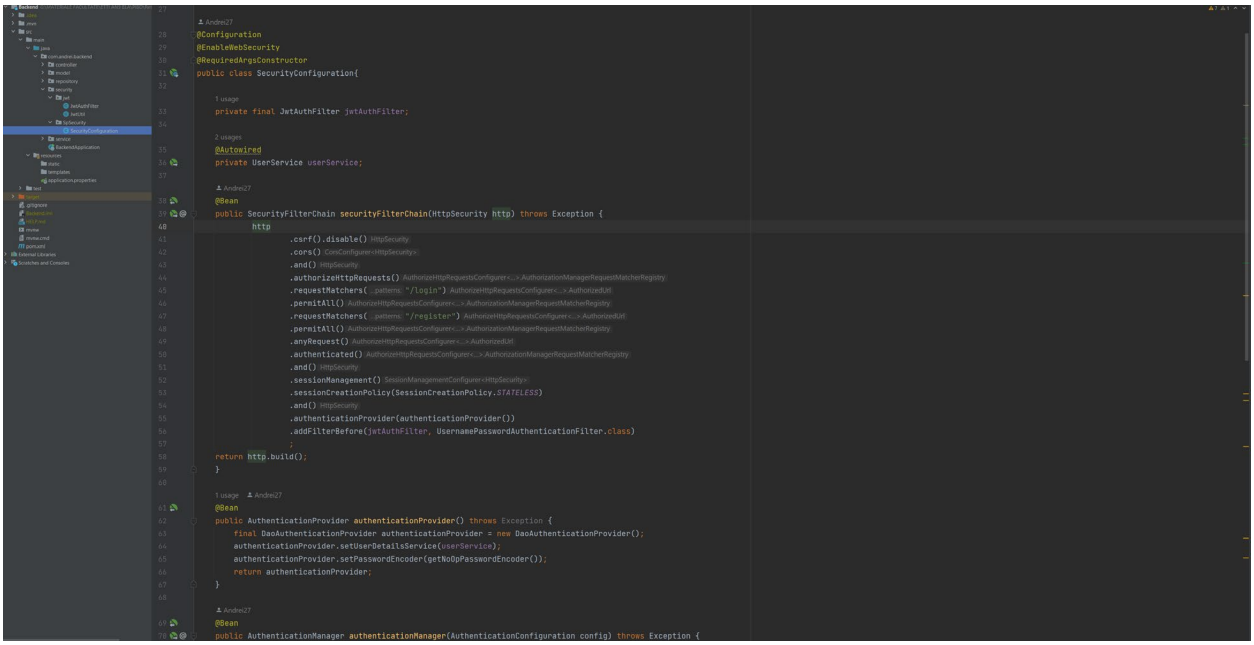


Fig.2.

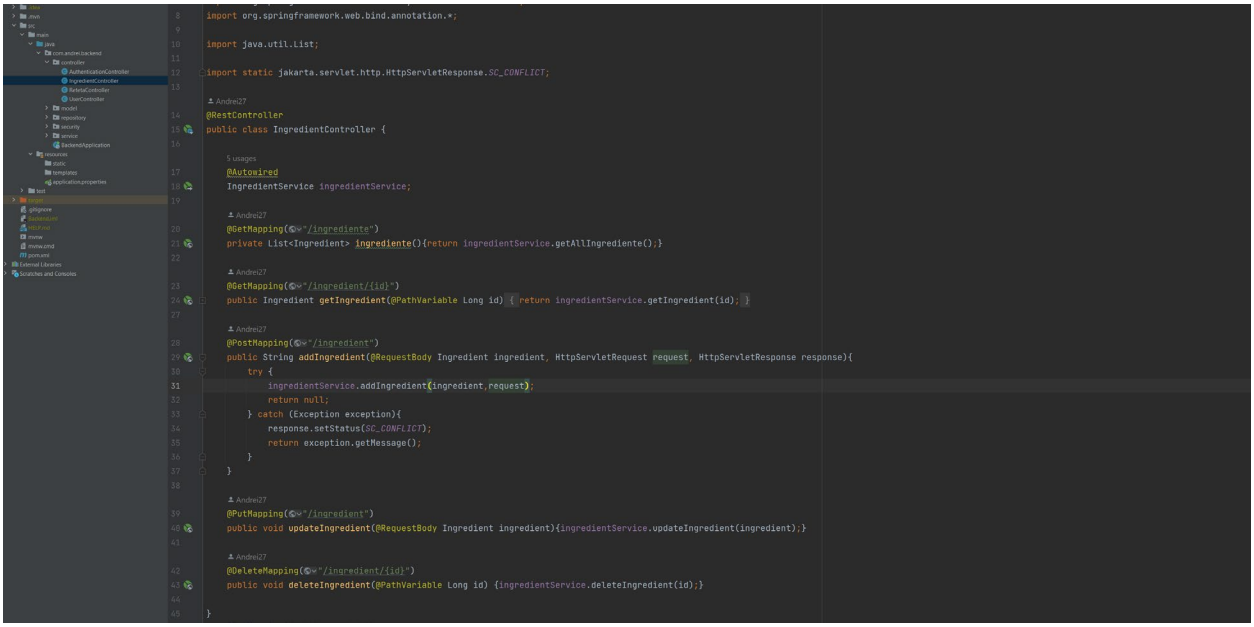
API-ul RetetaTa.ro folosește Spring Security 3.0 care era lansat de 2 săptămâni la dezvoltarea API-ului. Spring Security 2.0 se concentra pe suprascrierea claselor și metodelor pentru a personaliza comportamentul de securitate. Aceasta a făcut ca dezvoltatorii să fie nevoiți să înțeleagă în detaliu funcționarea internă a framework-ului pentru a-l adapta la nevoile lor. În Spring Security 3.0, totul este un bean și aceasta a făcut ca implementarea de securitate să devină mai flexibilă și a adus un plus pentru mentenanța codului , dar poate fi mai dificil de înțeles și de configurat la început cu noile metode de configurare când singura metodă de îndrumare este documentația oficială. Mai jos vedem construcția bean-ului Security Filter Chain.



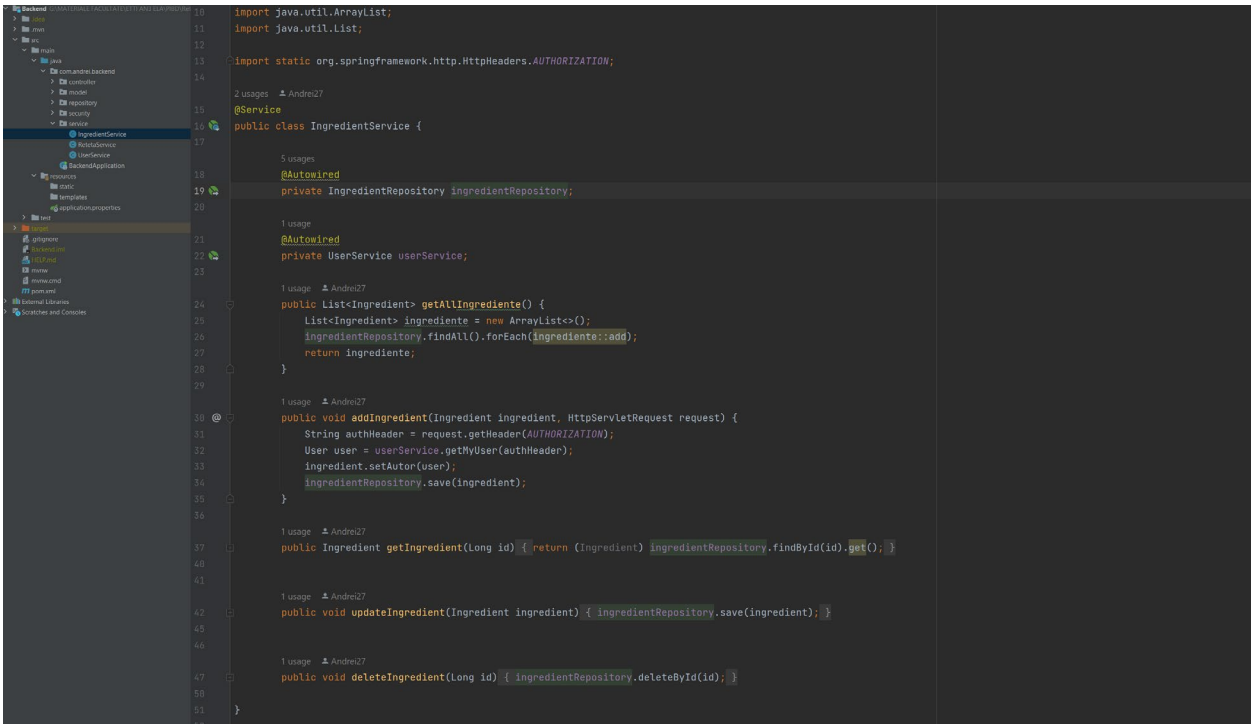
```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration{

    private final JwtAuthFilter jwtAuthFilter;

    @Autowired
    private UserService userService;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
            http
                    .csrf().disable()
                    .cors()
                    .and()
                    .authorizeHttpRequests()
                    .requestMatchers("/login")
                    .permitAll()
                    .requestMatchers("/register")
                    .permitAll()
                    .anyRequest()
                    .authenticated()
                    .and()
                    .sessionManagement()
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                    .and()
                    .authenticationProvider(authenticationProvider())
                    .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)
                    ;
    return http.build();
    }

    @Bean
    public AuthenticationProvider authenticationProvider() throws Exception {
        final DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userService);
        authenticationProvider.setPasswordEncoder(getNoOpPasswordEncoder());
        return authenticationProvider;
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {
```

Pentru menținerea utilizatorului logat pe întreaga durată a sesiunii sale(*Fig. 2, pasul 2*), am folosit JWT authentification. JWT (JSON Web Token) este un standard deschis pentru crearea și transmiterea de token-uri de autentificare. Acestea conțin informații despre utilizator și sunt semnate digital pentru a se asigura că sunt autentice. JWT-urile sunt plasate în header-ul cererilor HTTP ca un parametru "Authorization" cu valoarea "Bearer <JWT token>". Serverul poate valida semnătura JWT-ului și extrage informațiile despre utilizator pe care apoi îl trimite către Security Filter Chain pentru a decide dacă acesta are acces la resursele protejate. Metodele pentru codarea și decodarea token-ului dar și verificarea dacă acesta mai este valabil sau generarea unui refresh token sunt JwtUtil, iar metodele pentru accesarea user-ului in cazul nostru sunt in JwtAuthFilter care practic generează security chain-ului un user de care acesta să se ocupe mai departe.

Al doilea layer din Spring MVC se numește Controller. După ce un utilizator a trecut cu succes prin straturile de securitate în Spring Boot, cererea este redirecționată către layerul de control al aplicației. În acest layer, cererea este gestionată de un Controller, care este o clasă Java annotată cu @Controller. Controller-ul este responsabil pentru a determina care metodă din clasă trebuie să răspundă la cererea respectivă, bazată pe adresa URL și metoda HTTP (GET, POST, etc.). Metoda selectată din controler este apoi apelată, care poate prelucra cererea(exemplu: la post creează din JSON o instanță a obiectului nostru(vom vedea la Model ce înseamnă) sau din URL extrage anumiți parametrii, etc.) și trimite către metoda din service layer care prelucrează cererea făcută acum de controller.

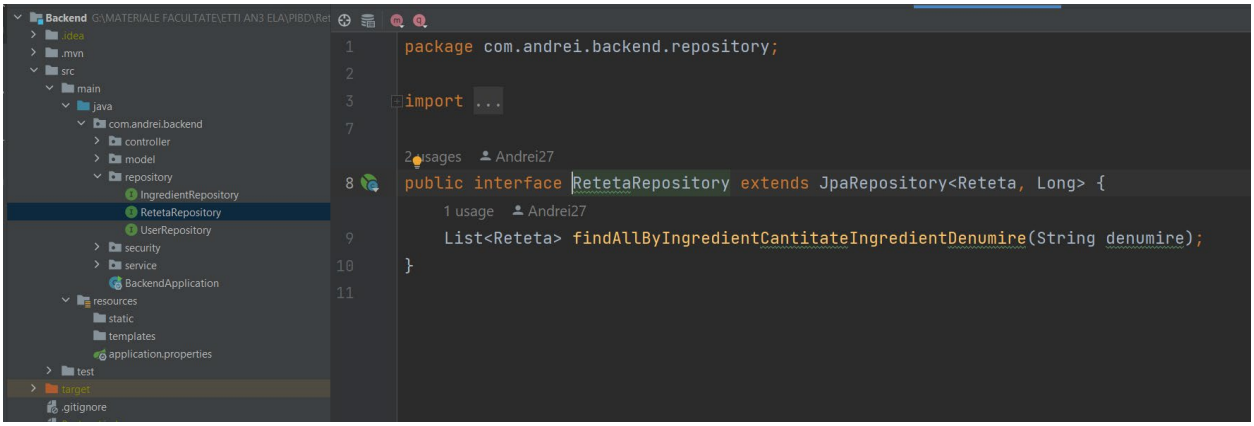


Al treilea layer care a fost apelat de către cererea noastră se numește Service Layer. Service Layer-ul din Spring MVC procesează logica aplicației, oferind metode specifice pentru operațiuni cum ar fi accesarea bazei de date, comunicarea cu alte sisteme sau orice altă logica necesara. Practic aici se face procesarea efectivă de care este responsabil API-ul nostru. În afară de procesarea efectivă a aplicație aceasta se ocupă și de apelarea părții de JPA care se ocupă de manipularea datelor, iar aceasta ne împinge către Repository Layer.



```java
import java.util.ArrayList;
import java.util.List;

import static org.springframework.http.HttpHeaders.AUTHORIZATION;

2 usages  Andrei27
@Service
public class IngredientService {

    5 usages
    @Autowired
    private IngredientRepository ingredientRepository;

    1 usage
    @Autowired
    private UserService userService;

    1 usage  Andrei27
    public List<Ingredient> getAllIngrediente() {
        List<Ingredient> ingrediente = new ArrayList<>();
        ingredientRepository.findAll().forEach(ingrediente::add);
        return ingrediente;
    }

    1 usage  Andrei27
    public void addIngredient(Ingredient ingredient, HttpServletRequest request) {
        String authHeader = request.getHeader(AUTHORIZATION);
        User user = userService.getMyUser(authHeader);
        ingredient.setAutor(user);
        ingredientRepository.save(ingredient);
    }

    1 usage  Andrei27
    public Ingredient getIngredient(Long id) { return (Ingredient) ingredientRepository.findById(id).get(); }


    1 usage  Andrei27
    public void updateIngredient(Ingredient ingredient) { ingredientRepository.save(ingredient); }


    1 usage  Andrei27
    public void deleteIngredient(Long id) { ingredientRepository.deleteById(id); }

}
```

Al patrulea layer din Spring MVC se referă la Repository Layer. Repository Layer-ul din Spring MVC este responsabil pentru gestionarea accesului la baza de date. Acesta este compus din clasele care implementează interfața JpaRepository sau alte interfete similare. JpaRepository oferă metode predefinite pentru operatiile CRUD(cum ar fi salvarea, actualizarea, ștergerea sau găsirea) unui obiect în baza de date. Se pot extinde aceste metode sau pot adăuga metode proprii pentru a se potrivi nevoilor aplicației. La implementarea sa se folosește și un obiect drept „Model”.



```
Backend G:\MATERIALE FACULTATE\ETTI AN3 ELA\PIBD\Ret
  .idea
  .mvn
  src
    main
      java
        com.andrei.backend
          controller
          model
          repository
            IngredientRepository
            RetetaRepository
            UserRepository
          security
          service
          BackendApplication
      resources
        static
        templates
        application.properties
    test
  target
  .gitignore
```

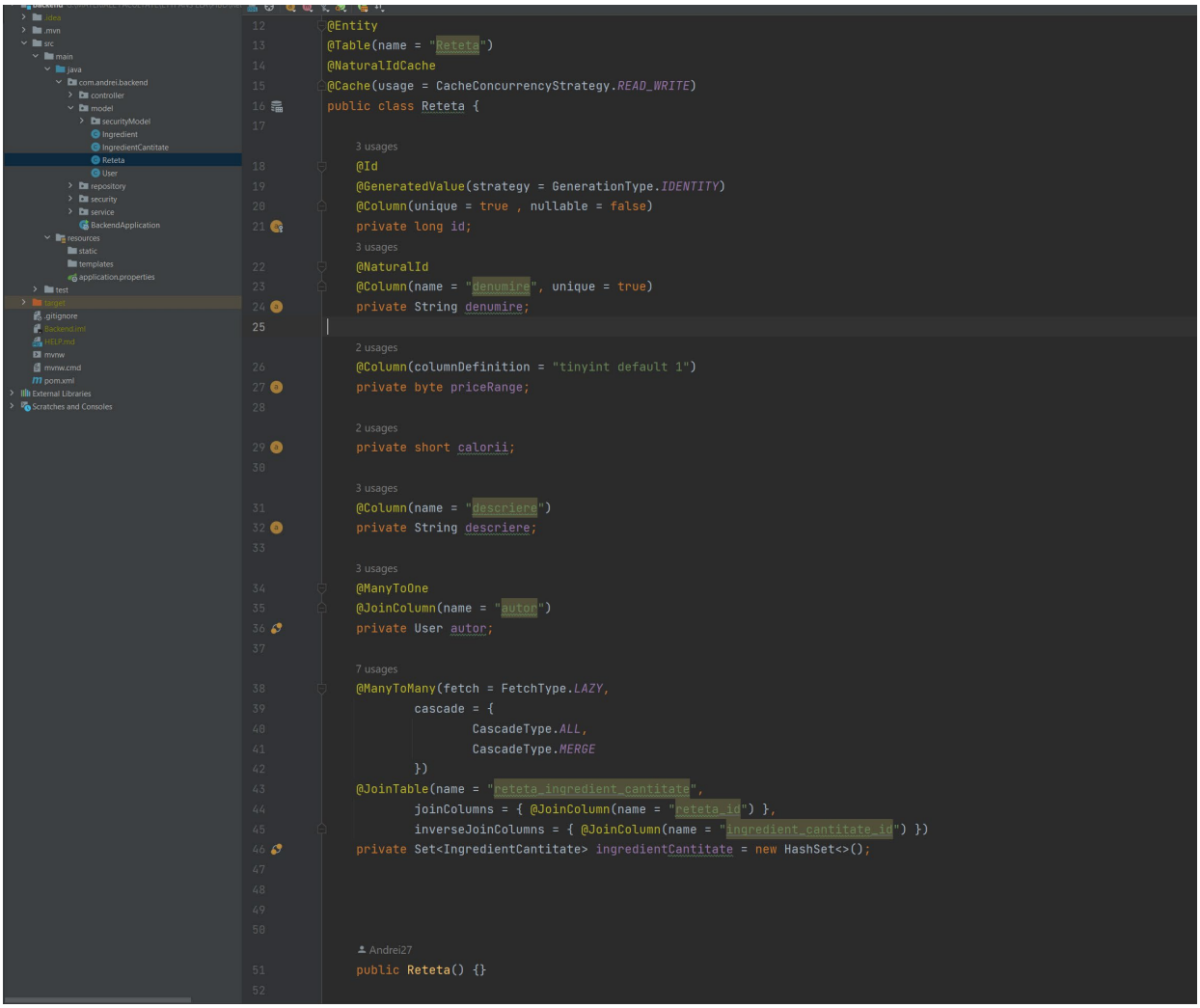
```java
package com.andrei.backend.repository;

import ...

2 usages  Andrei27
public interface RetetaRepository extends JpaRepository<Reteta, Long> {
    1 usage  Andrei27
    List<Reteta> findAllByIngredientCantitateIngredientDenumire(String denumire);
}
```

Ultimul așa zis „layer” din Spring MVC, nu este chiar un layer propriu zis ci mai degraba un model de care noi ne-am folosit pe întregul parcurs al aplicației. Acest „Model” poate fi accesat chiar din faza de controller când se face un POST, iar noi transmitem mai departe un obiect. Obiectul se bazează pe un model de obiect(nu voi relua tot conceptul de POO) cu niște proprietăți și metode specifice. Ce este particular la acest model este că după el este creată și baza de date.

Acest model se numește Spring Object Model și funcționează prin intermediul unui mecanism de mapare obiect-relațional (ORM) care permite dezvoltatorilor să lucreze cu baze de date utilizând obiecte Java în loc de cod SQL. În cazul utilizării JPA (Java Persistence API) împreună cu Spring, obiectele sunt mapate la tabelele din baza de date prin intermediul anotărilor @Entity.

Pentru a utiliza JPA împreună cu Spring, trebuie să configurați un EntityManagerFactory care poate fi creat prin intermediul clasei LocalEntityManagerFactoryBean. Acest EntityManagerFactory poate fi apoi utilizat pentru a crea un EntityManager care poate fi injectat în layer-ul service layer sau repository pentru a accesa și manipula datele din baza de date. Pentru fiecare entitate, se creeaza o clasa java care va fi mapata la o tabela din baza de date prin intermediul adnotării @Entity. Aceasta clasa va contine proprietatile care sunt coloanele din tabela respectiva. Adnotările @Id și @GeneratedValue sunt utilizate pentru a defini cheia primară a tabelei respective și modul în care este generată aceasta. Adnotările @Column sunt utilizate pentru a defini proprietățile care sunt mapate la coloanele din tabelul respectiv.

Prin intermediul acestui mecanism ORM putem accesa datele din baza de date prin intermediul obiectelor Java și putem efectua operațiunile CRUD , fără a mai scrie cod SQL.

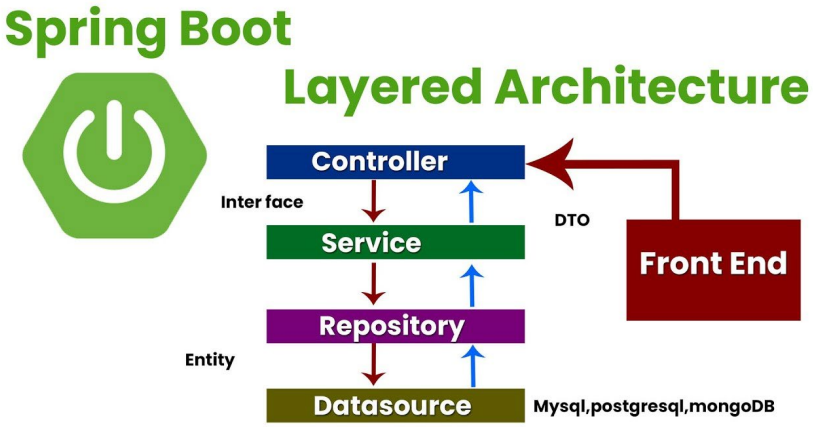


Deși tabelele din baza de date sunt create automat de JPA trebuie ținut cont de câteva aspecte precum:

- Relațiile între obiecte: JPA oferă mai multe opțiuni pentru a defini relațiile dintre tabele, cum ar fi @OneToOne pentru relația unu-la-unu între două tabele, @OneToMany pentru relația unu-la-mulți între două tabele, @ManyToOne pentru relația mulți-la-unu între două tabele și @ManyToMany pentru relația mulți-la-mulți între două tabele.
- Constângeri(constrains). JPA în mod automat va crea index-și pentru fiecare coloana din tabelă care are un tip de relație. ATENȚIE: JPA nu creează constrângeri(în special chei externe/străine). Pentru aceasta se poate un database-migration tool precum Flyway. Flyway este un instrument pentru managementul bazelor de date care automatizează procesul de adăugare a constrângerilor prin intermediul fișierelor SQL.(atenție trebuie folosit InnoDB)

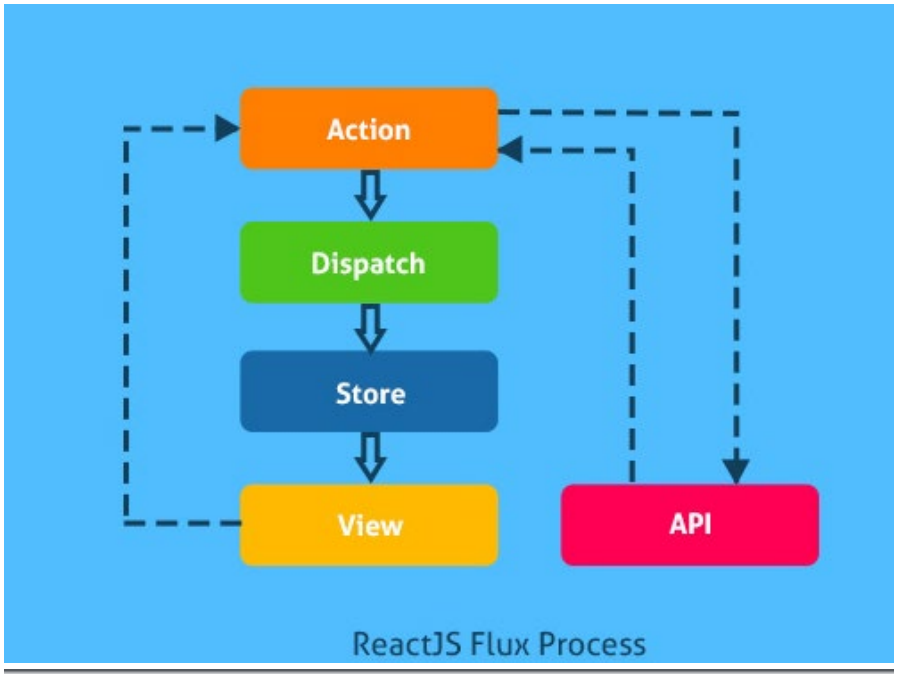
Metoda de lucru este în principiu următoarea: Se trimite un request HTTP cu POST de exemplu. Security Layer-ul analizează Header ul de autorizare, îl trimite către JwtAuthFilter care întoarce validitate logării sau nu. Dacă token-ul este corect atunci se continuă către controller-ul care deține URL la care s-a făcut request-ul. Contreller-ul apelează metoda care deține respectivul URL iar metoda primește ca parametru deja prestabilit(adică metoda primește parametru de tip Ingredient spre exemplu. Se va încerca decodificarea JSON-ului ca obiect de tip Ingredient) iar apoi apelează Service Layer-ul cu metoda sa de prelucrare căruia îi trimite obiectul Ingredient. Din acest moment și până la întoarcerea răspunsului metoda layer-ului așteaptă prelucrarea de metoda din service layer.

În service layer depinde de cel fel de procesare se întâmplă, dar în modul cel mai simplist se apelează repository-ul și acum și service layer-ul așteaptă răspunsul repository-ului. În mod normal repository-ul întoarce acknowledgement despre statusul operațiunii sau chiar un obiect către service layer. Service layer-ul adaugă sau nu procesare obiectului întors, pe care îl trimite înapoi către controller care aștepta răspunsul service layer-ului. Acum controller-ul va returna un către cel care a inițiat request-ul un cod HTTP și un payload-ul acestuia.(de obicei un fișier de tip JSON ce conține datele pe care le-a furnizat API-ul).

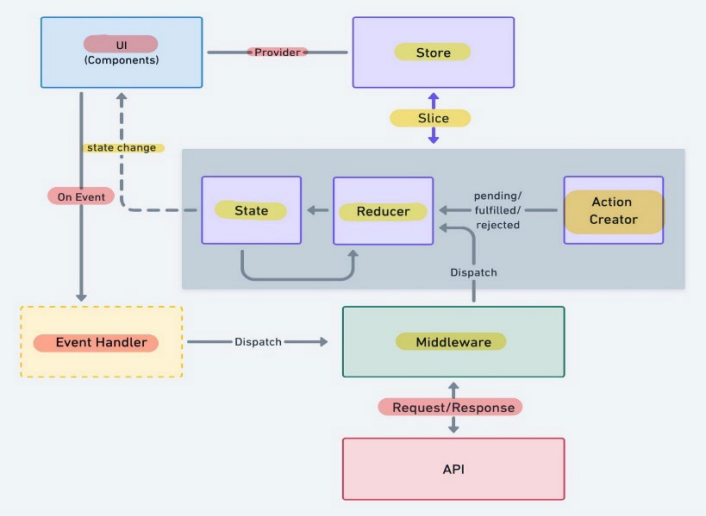


2) FRONTEND

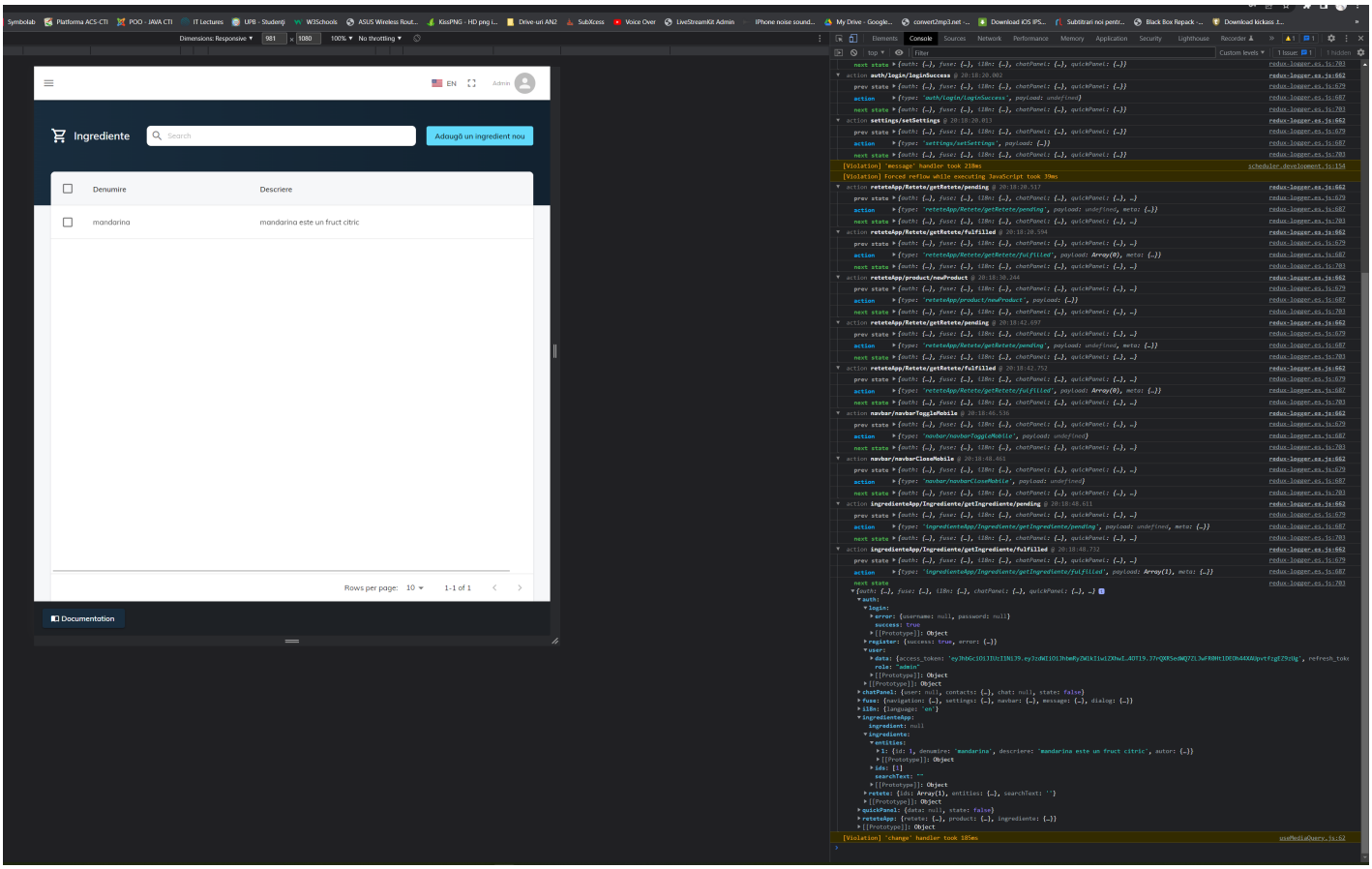
ReactJS este un framework JavaScript open-source pentru crearea de aplicații web cu interfață utilizator interactivă. Principiul de bază al acestui framework se bazează pe conceptul de componente, care permit dezvoltatorilor să construiască aplicații web mai eficient prin modularizarea și reutilizarea codului. ReactJS utilizează un mecanism de actualizare a afișării numit "re-rendering", care permite actualizarea eficientă a afișării pe ecran fără a reîncărca întreaga pagină web. Acest re-rendering poate fi declanșat automat de către actualizarea unor componente React sau manual/la modificare unor anumite variabile/acțiuni prestabilite de noi cu ajutorul unui „Hook” numit useEffect.



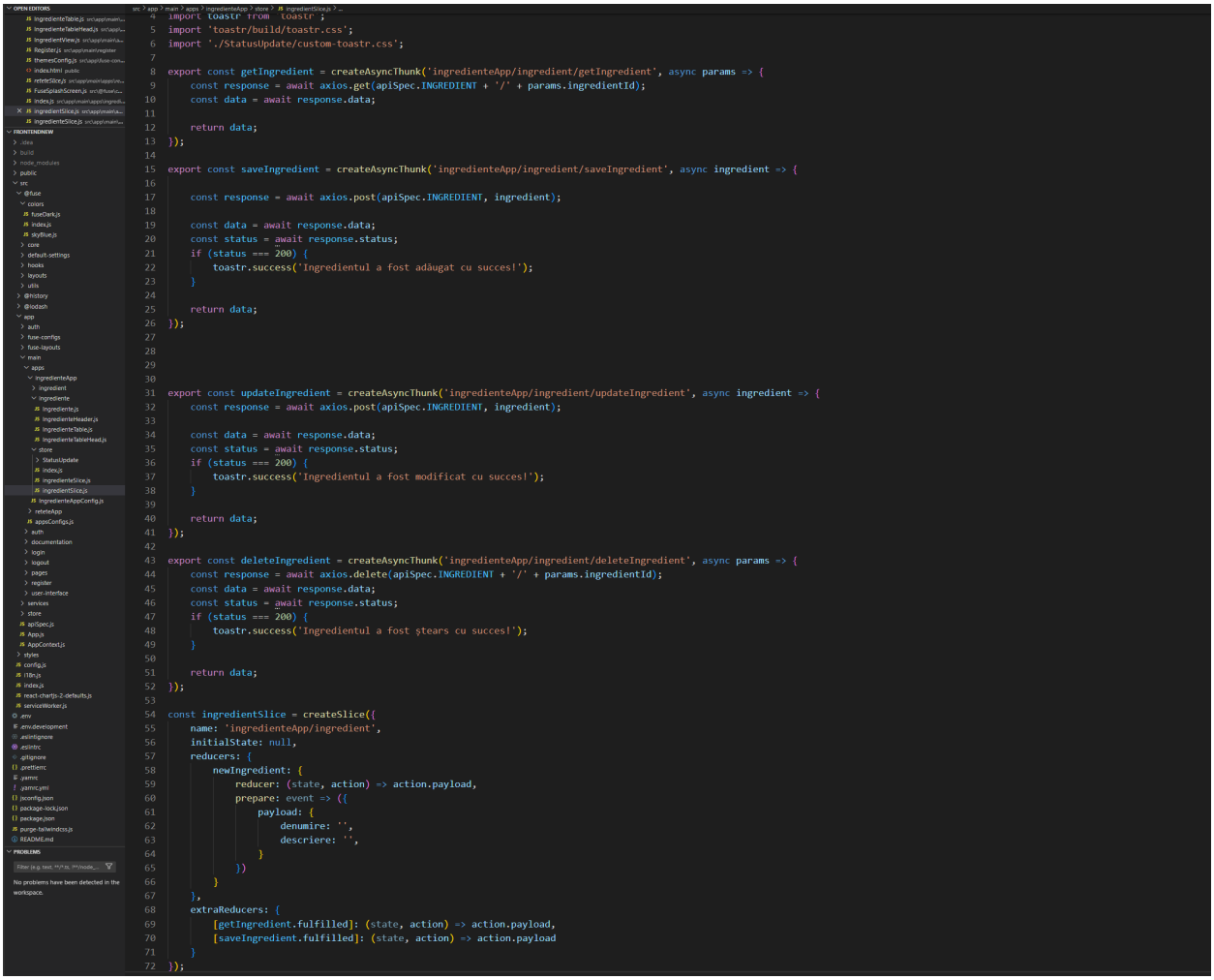
În aplicațiile React, termenul "action" se referă la acțiunile care descriu schimbările de date care au loc în aplicație. Aceste acțiuni sunt vizibile prin intermediul unor obiecte JavaScript simple, care conțin informații despre schimbările de date. Procesul de transmitere a acțiunilor către un "store" pentru a fi gestionate se numește "dispatch". "Store" este un container centralizat pentru starea aplicației, care gestionează și actualizează automat componentele care depind de acea stare(un fel de memorie de scurtă durată a aplicației web. Ea dispare la un refresh al paginii web). "View" este o reprezentare vizuală a stării din "store" pentru utilizator, exprimată prin intermediul componentelor React, care sunt actualizate automat atunci când starea din "store" se schimbă. Noi în această aplicație pentru a eficientiza procesul de salvare a stărilor(state-uri) am folosit o altă librărie suplimentară numita **Redux Toolkit.**



Redux Toolkit poate fi utilizat cu ReactJS pentru a gestiona state-ul aplicației. Acesta oferă o configurație implicită și un set de instrumente pentru a simplifica procesul de creare și gestionare a unui "store" Redux în aplicația React. Principalele beneficii ale utilizării Redux Toolkit cu ReactJS sunt: simplificarea procesului de configurare și utilizare a Redux în aplicația React, eliminarea boilerplate-ului și a codului redundante, precum și posibilitatea de a crea "store"-uri cu o configurație personalizată sau *"store"-uri reducer-based* care eficientizează și îmbunătățesc ușurința de înțelegere a codului sau.



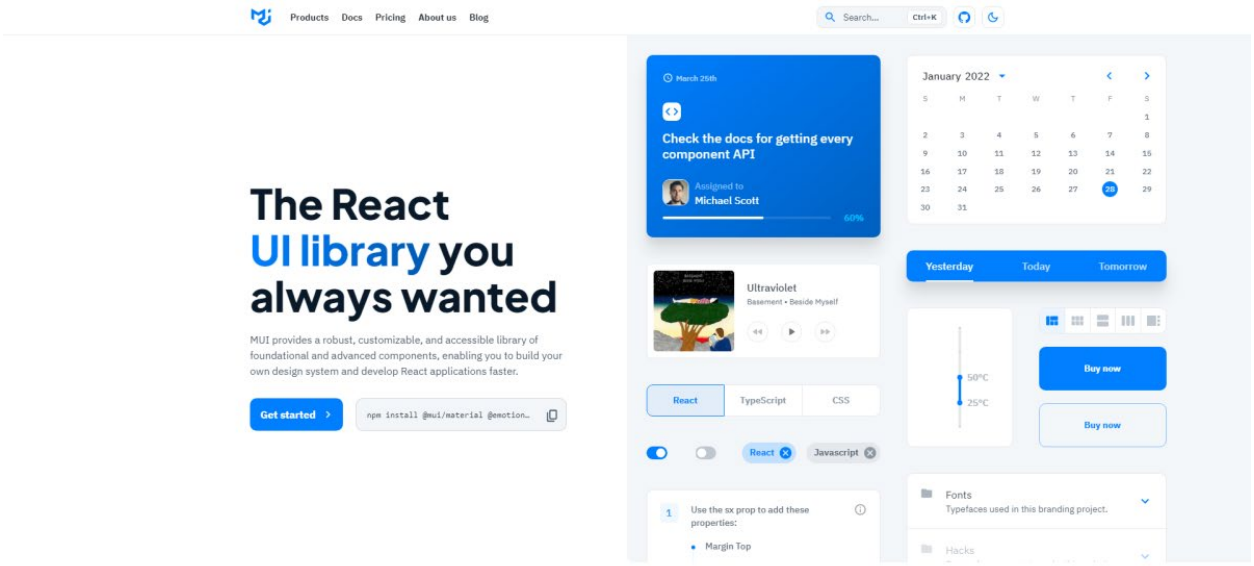
Fiecare stare conține toți parametri necesari pe care componentele React le pot accesa cu ajutorul unui metode in Redux Toolkit și anume useSelect. Cu acești selectori pot accesa starea curentă. Mai jos avem „slice”-ul unui ingredient. Un slice este o bucată constituentă a „store”-ului. Acest lucru ne-a permis sa avem mai multe „mini-store”-uri pentru fiecare pagina/aplicatie React pe care o avem.



```
import toastr from 'toastr';
import 'toastr/build/toastr.css';
import './StatusUpdate/custom-toastr.css';

export const getIngredient = createAsyncThunk('ingredienteApp/ingredient/getIngredient', async params => {
    const response = await axios.get(apiSpec.INGREDIENT + '/' + params.ingredientId);
    const data = await response.data;

    return data;
});

export const saveIngredient = createAsyncThunk('ingredienteApp/ingredient/saveIngredient', async ingredient => {

    const response = await axios.post(apiSpec.INGREDIENT, ingredient);

    const data = await response.data;
    const status = await response.status;
    if (status === 200) {
        toastr.success('Ingredientul a fost adăugat cu succes!');
    }

    return data;
});




export const updateIngredient = createAsyncThunk('ingredienteApp/ingredient/updateIngredient', async ingredient => {
    const response = await axios.post(apiSpec.INGREDIENT, ingredient);

    const data = await response.data;
    const status = await response.status;
    if (status === 200) {
        toastr.success('Ingredientul a fost modificat cu succes!');
    }

    return data;
});

export const deleteIngredient = createAsyncThunk('ingredienteApp/ingredient/deleteIngredient', async params => {
    const response = await axios.delete(apiSpec.INGREDIENT + '/' + params.ingredientId);
    const data = await response.data;
    const status = await response.status;
    if (status === 200) {
        toastr.success('Ingredientul a fost șters cu succes!');
    }

    return data;
});

const ingredientSlice = createSlice({
    name: 'ingredienteApp/ingredient',
    initialState: null,
    reducers: {
        newIngredient: {
            reducer: (state, action) => action.payload,
            prepare: event => ({
                payload: {
                    denumire: '',
                    descriere: '',
                }
            })
        }
    },
    extraReducers: {
        [getIngredient.fulfilled]: (state, action) => action.payload,
        [saveIngredient.fulfilled]: (state, action) => action.payload
    }
});
```

După cum se poate vedea pentru operațiile CRUD am ales să folosim **AXIOS**. Axios poate fi utilizat cu ReactJS pentru a realiza cereri HTTP și pentru a gestiona răspunsurile. Acesta este un client HTTP bazat pe promisiuni, care poate fi utilizat pentru a realiza cereri GET, POST, PUT, DELETE și altele, precum și pentru a gestiona erorile și alte aspecte ale răspunsurilor. Principalele beneficii ale utilizării Axios cu ReactJS sunt simplificarea procesului de realizare a cererilor HTTP, posibilitatea de a gestiona erorile în mod eficient, și posibilitatea de a realiza cereri asincrone și sincrone.

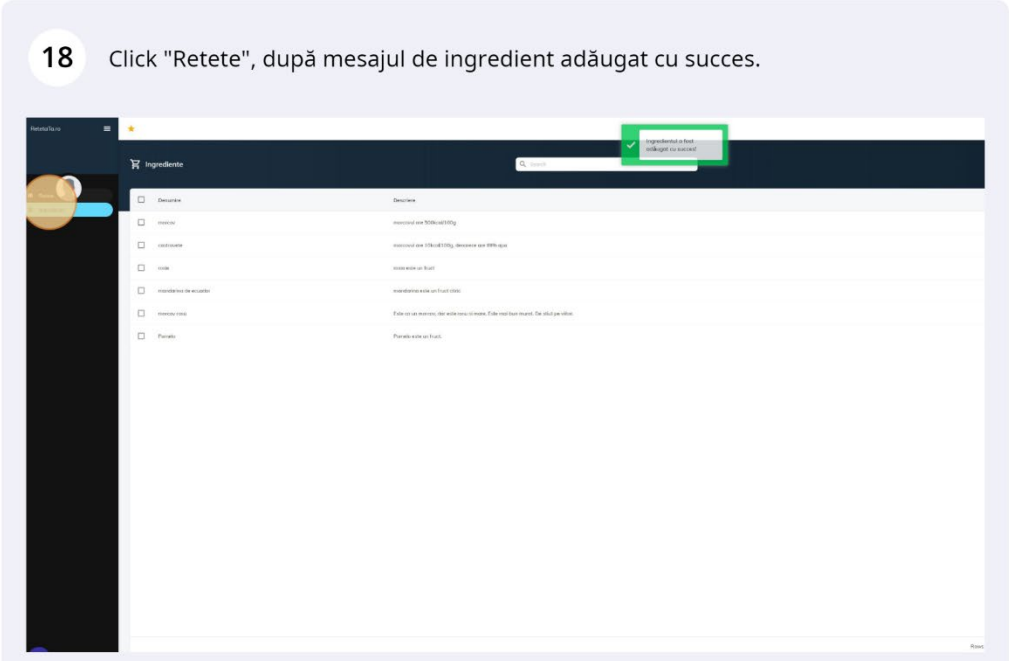
Elementele au fost majoritar folosite din Material-UI care ne permite sa grabim procesul de dezvoltare a front-end-ului.



Alaturi de Material-UI am folosit Fuse Framework care combina elemente de MUI in elemente Fuse care au deja funcționalități complexe.

Un tur complet al aplicație se găsește chiar [aici](#).

(sau în repository-ul aplicației de Github)



Bibliografie:

1. https://stackoverflow.com/
2. https://wikipedia.org/
3. https://spring.io/
4. https://javabrains.io/
5. https://www.seleniumexpress.com/
6. http://youtube.com/
7. https://mui.com/material-ui/getting-started/overview/