**Τεχνική αναφορά**

**Περιεχόμενα**

Κεφ.1: Σκοπός της τεχνικής αναφοράς

Κεφ.2: Παράδειγμα setup του framework FastAPI σε εικονικό περιβάλλον ενός συστήματος

Κεφ.3: Τα κύρια χαρακτηριστικά της βιβλιοθήκης React μέσω παραδειγμάτων:

Παράδειγμα ενός component

Παράδειγμα state με useState

Παράδειγμα props

Παράδειγμα του hook, useEffect

Παράδειγμα χρήσης του Context API

Παράδειγμα Rooter

Παράδειγμα setup της React και επικοινωνία με το backend (FastAPI)

Κεφ.4: Εγκατάσταση της PostgreSQL σε ένα σύστημα

Κεφ.5: Χρήση του Git μέσω κοινών εντολών

Κεφ.6: Χρήση του GitHub

Κεφ.7: Setup του project και η δομή του κώδικα (backend)

Κεφ.7.1:Modules και παραδείγματα API Endpoints

Κεφ.8: Setup του project και η δομή του κώδικα (frontend)

Κεφ.8.1: Κύρια Components και προεπεξεργασία δεδομένων

Κεφ.9: Περιγραφή UI

Κεφ.10: Δοκιμές (Testing)

Κεφ.10.1: Το format του κωδικού της συνταγής και του υλικού

Κεφ.10.2: Παραδείγματα Δοκιμών

Κεφ.10.2.1: Καταχώρηση νέου υλικού

Κεφ.10.2.1.1: Περίπτωση ελιάς

Κεφ.10.2.1.2: Περίπτωση φάβας

Κεφ.10.2.1.3: Περίπτωση μοσχαρίσιου κιμά

Κεφ.10.2.1.4: Περίπτωση καυκαλήθρες

Κεφ.10.3: Καταχώρηση νέας συνταγής

Κεφ.10.3.1: Περίπτωση παστίτσιο

Κεφ.10.3.2: Περίπτωση τσαλαφούτι

**Κεφ.1: Σκοπός της τεχνικής αναφοράς**

Σκοπός του τεύχους της τεχνικής αναφοράς είναι να λειτουργήσει συμπληρωματικά με το κύριο κείμενο της πτυχιακής, αναδεικνύοντας κάποια σημαντικά σημεία μέσω παραδειγμάτων και μέσω μιας πιο εμπεριστατωμένης ανάλυσης σε κομμάτια κώδικα που χρησιμοποιούνται στην εφαρμογή. Με αυτό τον τρόπο επιτυγχάνεται μια ακόμα πιο ολοκληρωμένη τεκμηρίωση σχετικά με τη λογική που ακολουθήθηκε και τις επιμέρους τεχνικές που εφαρμόστηκαν ώστε να παραχθεί το τελικό προϊόν.

**Κεφ.2: Παράδειγμα setup του framework FastAPI σε εικονικό περιβάλλον ενός συστήματος**

Ακολουθεί ένα παράδειγμα με όλες τις οδηγίες και τα βήματα που χρειάζονται για να στηθεί ένα περιβάλλον FastAPI που χρησιμοποιεί τις βιβλιοθήκες Pydantic, SQLAlchemy και Uvicorn καθώς και το εργαλείο της αυτόματης τεκμηρίωσης Swagger UI.

1. **Δημιουργία και ρύθμιση ενός εικονικού περιβάλλοντος για το έργο**

Αρχικά θα πρέπει η Python να είναι ήδη εγκατεστημένη στο σύστημα. Αν είναι εγκατεστημένη τότε θα πρέπει να δημιουργηθεί ένα εικονικό περιβάλλον για το project, να ενεργοποιηθεί και έπειτα να εγκατασταθούν σε αυτό οι απαραίτητες βιβλιοθήκες (dependencies) όπως περιγράφεται στις παρακάτω εντολές.

# Δημιουργία εικονικού περιβάλλοντος

python -m venv myenv

# Ενεργοποίηση εικονικού περιβάλλοντος (Windows)

myenv\Scripts\activate

# Ενεργοποίηση εικονικού περιβάλλοντος (Unix or MacOS)

source myenv/bin/activate

# Εγκατάσταση βιβλιοθηκών

pip install fastapi uvicorn sqlalchemy pydantic

2. **Δημιουργία των αρχείων του project**

Μία ενδεικτική δομή των φακέλων και των αρχείων του project μπορεί να είναι αυτή που φαίνεται παρακάτω:

mkdir fastapi\_project

cd fastapi\_project

mkdir app

touch app/main.py

touch app/models.py

touch app/database.py

touch app/schemas.py

touch app/crud.py

touch app/\_\_init\_\_.py

3. **Ρύθμιση της βάσης δεδομένων (app/database.py)**

Στο αρχείο database.py, πρέπει να ρυθμιστεί η σύνδεση με τη βάση δεδομένων χρησιμοποιώντας την βιβλιοθήκη SQLAlchemy.

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "sqlite:///./sql\_app.db"

# SQLALCHEMY\_DATABASE\_URL = "postgresql://user:password@postgresserver/db"

engine = create\_engine(

SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False}

)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

Σε αυτό το σημείο μπορεί να χρησιμοποιηθεί όπως φαίνεται είτε η sqlite είτε η postgresql. Εδώ χρησιμοποιείται η πρώτη.

4. **Ορισμός μοντέλων (app/models.py)**

Στο αρχείο models.py, θα οριστούν τα μοντέλα της βάσης δεδομένων.

from sqlalchemy import Column, Integer, String

from .database import Base

class Item(Base):

\_\_tablename\_\_ = "items"

id = Column(Integer, primary\_key=True, index=True)

title = Column(String, index=True)

description = Column(String, index=True)

5. **Ορισμός σχημάτων (app/schemas.py)**

Στο αρχείο schemas.py, θα οριστούν τα Pydantic models για την επικύρωση των δεδομένων.

from pydantic import BaseModel

class ItemBase(BaseModel):

title: str

description: str

class ItemCreate(ItemBase):

pass

class Item(ItemBase):

id: int

class Config:

orm\_mode = True

6.  **Δημιουργία CRUD λειτουργιών (app/crud.py ή services.py)**

Στο αρχείο crud.py ή services.py θα υλοποιηθούν όλες οι λειτουργίες για τη διαχείριση των δεδομένων.

from sqlalchemy.orm import Session

from . import models, schemas

def get\_item(db: Session, item\_id: int):

return db.query(models.Item).filter(models.Item.id == item\_id).first()

def get\_items(db: Session, skip: int = 0, limit: int = 10):

return db.query(models.Item).offset(skip).limit(limit).all()

def create\_item(db: Session, item: schemas.ItemCreate):

db\_item = models.Item(title=item.title, description=item.description)

db.add(db\_item)

db.commit()

db.refresh(db\_item)

return db\_item

7. **Δημιουργία των endpoints (app/main.py)**

Στο αρχείο main.py, θα δημιουργηθούν τα API endpoints χρησιμοποιώντας FastAPI.

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from . import models, schemas, crud

from .database import SessionLocal, engine

models.Base.metadata.create\_all(bind=engine)

app = FastAPI()

# Dependency

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

@app.post("/items/", response\_model=schemas.Item)

def create\_item(item: schemas.ItemCreate, db: Session = Depends(get\_db)):

return crud.create\_item(db=db, item=item)

@app.get("/items/", response\_model=list[schemas.Item])

def read\_items(skip: int = 0, limit: int = 10, db: Session = Depends(get\_db)):

items = crud.get\_items(db, skip=skip, limit=limit)

return items

@app.get("/items/{item\_id}", response\_model=schemas.Item)

def read\_item(item\_id: int, db: Session = Depends(get\_db)):

db\_item = crud.get\_item(db, item\_id=item\_id)

if db\_item is None:

raise HTTPException(status\_code=404, detail="Item not found")

return db\_item

8. **Εκτέλεση της eφαρμογής**

Τέλος, χρησιμοποιώντας το Uvicorn, η εφαρμογή εκτελείται με την εντολή.

uvicorn main:app –reload

9. **Πρόσβαση στο Swagger UI**

Όταν εκτελεστεί η εφαρμογή, υπάρχει η δυνατότητα πρόσβασης στο Swagger UI μέσω της διεύθυνσης http://127.0.0.1:8000/docs. Αυτή η σελίδα θα παρέχει μια πλήρη τεκμηρίωση των endpoints της εφαρμογής που θα επιτρέπει τη δοκιμή τους.

Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα του εργαλείου Swagger UI του FastAPI:

A screenshot of a computer

Description automatically generatedΠηγή: <https://fastapi.tiangolo.com/>

Με αυτές τις εντολές και τα αρχεία, έχει στηθεί ένα περιβάλλον FastAPI που χρησιμοποιεί Pydantic για την επικύρωση δεδομένων, SQLAlchemy για τη διαχείριση της βάσης δεδομένων και Uvicorn για την εκτέλεση του server.

**Κεφ.3: Τα κύρια χαρακτηριστικά της βιβλιοθήκης React μέσω παραδειγμάτων**

Τα κύρια χαρακτηριστικά της βιβλιοθήκης React θα μπορούσαν συνοπτικά να αποτελέσουν τα παρακάτω:

**1. Components:** Η React βασίζεται σε μια αρχιτεκτονική που χρησιμοποιεί components. Τα components είναι αυτοτελή και επαναχρησιμοποιούμενα κομμάτια κώδικα που αντιπροσωπεύουν τα διάφορα μέρη της διεπαφής χρήστη π.χ ένα κουμπί ή ένα footer. Κάθε component μπορεί να έχει τη δική του λογική και κατάσταση (state) και μπορεί να συνδυάζεται με άλλα components για τη δημιουργία πιο σύνθετων διεπαφών.

**Παράδειγμα ενός component**

α. To component Greeting κάνει display το μήνυμα Hello, World.

import React from 'react';

function Greeting() {

return <h1>Hello, World</h1>;

}

export default Greeting;

β. Χρήση του component Greeting από το App.

import React from 'react';

import Greeting from './ Greeting;

function App() {

return (

<div className="App">

<header className="App-header">

**< Greeting />**

</header>

</div>

);

}

export default App;

γ. Γίνεται το rendering του App στο DOM της εφαρμογής.

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

**2. Virtual DOM:** Το **Virtual DOM** είναι μια αναπαράσταση του πραγματικού DOM (Document Object Model) που υπάρχει στη μνήμη. Σε αντίθεση με τον πραγματικό DOM, το Virtual DOM δεν χειρίζεται απευθείας τα στοιχεία της σελίδας (HTML elements). Αντίθετα, είναι ένα ελαφρύ αντίγραφο του πραγματικού DOM που χρησιμοποιείται από τη React για να παρακολουθεί τις αλλαγές στην κατάσταση της εφαρμογής.

A diagram of different types of information

Description automatically generatedΠηγή: <https://www.syncfusion.com/blogs/post/react-virtual-dom>

Όταν η εφαρμογή React φορτώνεται, δημιουργείται ένα Virtual DOM που αντιπροσωπεύει τη δομή του user interface. Αντιγράφεται στο Virtual DOM το πραγματικό DOM. Όταν αλλάζει η κατάσταση (state) της εφαρμογής, η React δημιουργεί ένα νέο Virtual DOM που αντιπροσωπεύει την ενημερωμένη κατάσταση της εφαρμογής. Η React συγκρίνει το νέο Virtual DOM με το προηγούμενο Virtual DOM χρησιμοποιώντας έναν αλγόριθμο σύγκρισης (diffing algorithm). Ο αλγόριθμος αυτός εντοπίζει τις αλλαγές μεταξύ των δύο δέντρων DOM. Η React υπολογίζει τις ελάχιστες δυνατές αλλαγές που χρειάζεται να γίνουν τώρα στο πραγματικό DOM και τις εκτελεί σε αυτό.

**3. State και Props:** Το state είναι ένα αντικείμενο που αντιπροσωπεύει την εσωτερική κατάσταση ενός component, ενώ τα props είναι παράμετροι που περνούν δεδομένα από το ένα component στον άλλο. Με αυτό τον τρόπο μπορούν τα διάφορα components να επικοινωνούν μεταξύ τους.

**Παράδειγμα state με useState**

Το state τροποποιείται είτε μέσω της μεθόδου setState (class components) είτε μέσω του hook useState (functional components). Τα hooks εξετάζονται και πιο κάτω.

//Γίνεται import του hook, useState

import React, { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

return (

<div>

<p>You clicked {count} times</p>

// το state του count αυξάνεται κατά 1, κάθε φορά που πατιέται το κουμπί

<button onClick={() => setCount(count + 1)}>Click me</button>

</div>

);

}

export default Counter;

**Παράδειγμα props**

import React from 'react';

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

function App() {

return (

// το Welcome χρησιμοποιείται 2 φορές από το App στο οποίο περνιούνται hardcoded οι τιμές του property του Welcome με όνομα name.

<div>

<Welcome name="Alice" />

<Welcome name="Bob" />

</div>

);

}

export default App;

**4. Hooks:** Τα hooks είναι μια νέα προσθήκη στη React και χρησιμοποιούνται από την έκδοση 16.8 και μετά. Επιτρέπουν την χρήση του state που αναφέρθηκε παραπάνω αλλά και άλλων features της React, χωρίς τη δημιουργία κλάσεων. Τα πιο γνωστά hooks που χρησιμοποιούνται είναι κυρίως τα useState και useEffect. Το useState αναφέρθηκε και παραπάνω στο παράδειγμα state.

**Παράδειγμα του hook, useEffect**

import React, { useState, useEffect } from 'react';

function Example() {

const [count, setCount] = useState(0);

useEffect(() => {

document.title = `You clicked ${count} times`;

}, [count]); // Τρέχει μόνο όταν το count αλλάζει

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}

**5. State Management:** Για πιο σύνθετες εφαρμογές, η διαχείριση της κατάστασης (state) μπορεί να γίνει με τη χρήση βιβλιοθηκών όπως το Redux ή το Context API της React. Το Context API επιτρέπει την μετάδοση δεδομένων μέσω του component tree χωρίς να χρειάζεται να περάσουν μέσω props.

**Παράδειγμα χρήσης του Context API**

Ακολουθεί ένα παράδειγμα χρήσης του Context API που διαχειρίζεται το state μιας εφαρμογής:

α. **Δημιουργία του Context**:

import React, { createContext, useState, useContext } from 'react';

// Δημιουργία του context

const MyContext = createContext();

// Δημιουργία ενός provider component

export const MyProvider = ({ children }) => {

const [state, setState] = useState('Αρχική Τιμή');

return (

<MyContext.Provider value={{ state, setState }}>

{children}

</MyContext.Provider>

);

};

// Custom hook για εύκολη χρήση του context

export const useMyContext = () => {

return useContext(MyContext);

};

β. **Χρήση του Provider στο Component Tree**:

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import { MyProvider } from './MyContext';

ReactDOM.render(

<React.StrictMode>

<MyProvider>

<App />

</MyProvider>

</React.StrictMode>,

document.getElementById('root')

);

γ. **Κατανάλωση του Context**:

import React from 'react';

import { useMyContext } from './MyContext';

const MyComponent = () => {

const { state, setState } = useMyContext();

return (

<div>

<p>Η τρέχουσα κατάσταση είναι: {state}</p>

<button onClick={() => setState('Νέα Τιμή')}>Αλλαγή Κατάστασης</button>

</div>

);

};

export default MyComponent;

δ. **Χρήση του Component**:

import React from 'react';

import MyComponent from './MyComponent';

const App = () => {

return (

<div>

<h1>Παράδειγμα Χρήσης του Context API</h1>

<MyComponent />

</div>

);

};

export default App;

6. **React Router:** Το React Router είναι μια βιβλιοθήκη για την προσθήκη πλοήγησης και δρομολόγησης (routing) σε μια εφαρμογή React. Επιτρέπει τη δημιουργία διαφορετικών σελίδων και τη διαχείριση της πλοήγησης μεταξύ αυτών.

**Παράδειγμα Rooter**

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Home from './Home';

import About from './About';

function App() {

return (

<Router>

<Switch>

<Route path="/" exact component={Home} />

<Route path="/about" component={About} />

</Switch>

</Router>

);

}

export default App;

**Παράδειγμα setup της React και επικοινωνία με το backend (FastAPI)**

Ακολουθεί ένα παράδειγμα που αναδεικνύει τον τρόπο με τον οποίο μπορεί να στηθεί μία εφαρμογή React και τον τρόπο που αυτή μπορεί και επικοινωνεί με το backend μέσω των endpoints.

1. **Δημιουργία ενός νέου έργου React**

Αρχικά πρέπει στο σύστημα να υπάρχουν ήδη εγκατεστημένα τα Node.js και npm (Node Package Manager). Έπειτα γίνεται χρήση της παρακάτω εντολής ώστε να δημιουργηθεί το project με όνομα myapp.

npx create-react-app myapp

cd myapp

2. **Δομή Φακέλων**

Η δομή των φακέλων που δημιουργήθηκαν θα έχει την παρακάτω μορφή

myapp/

├── node\_modules/

├── public/

├── src/

│ ├── components/

│ ├── App.css

│ ├── App.js

│ ├── App.test.js

│ ├── index.css

│ ├── index.js

│ └── serviceWorker.js

├── .gitignore

├── package.json

├── README.md

└── yarn.lock

3. **Δημιουργία Components**

myapp/src/components/Header.js

import React from 'react';

function Header() {

return (

<header>

<h1>My App</h1>

</header>

);

}

export default Header;

myapp/src/components/Footer.js

import React from 'react';

function Footer() {

return (

<footer>

<p>&copy; 2023 My App</p>

</footer>

);

}

export default Footer;

myapp/src/components/ItemList.js

import React from 'react';

function ItemList({ items }) {

return (

<ul>

{items.map(item => (

<li key={item.id}>{item.title}</li>

))}

</ul>

);

}

export default ItemList;

4. **Χρήση Components στο App.js**

myapp/src/App.js

import React, { useState, useEffect } from 'react';

import './App.css';

import Header from './components/Header';

import Footer from './components/Footer';

import ItemList from './components/ItemList';

function App() {

const [items, setItems] = useState([]);

useEffect(() => {

// Fetch data from the backend (FastAPI)

fetch('http://localhost:8000/items/')

.then(response => response.json())

.then(data => setItems(data));

}, []);

return (

<div className="App">

<Header />

<main>

<ItemList items={items} />

</main>

<Footer />

</div>

);

}

export default App;

5. **Σύνδεση με το backend**

Όπως αναφέρεται στο παραπάνω κώδικα, η εφαρμογή React κάνει αιτήματα HTTP στο backend που δημιουργήθηκε με το FastAPI (βλέπε ενότητα Backend: FastAPI) για να κάνει retrieve τα δεδομένα από τον server. Αυτό επιτυγχάνεται μέσω του fetch API. Στην περίπτωση αυτή, το frontend επικοινωνεί με το backend χρησιμοποιώντας το endpoint http://localhost:8000/items/, όπως ακριβώς ορίστηκε στο main.py.

@app.get("/items/", response\_model=list[schemas.Item])

def read\_items(skip: int = 0, limit: int = 10, db: Session = Depends(get\_db)):

items = crud.get\_items(db, skip=skip, limit=limit)

return items

**Κεφ.4: Εγκατάσταση της PostgreSQL σε ένα σύστημα**

Ακολουθεί ένα παράδειγμα που αναδεικνύει τον τρόπο με τον οποίο μπορεί να γίνει η εγκατάσταση της PostgreSQL σε ένα σύστημα σε οποιοδήποτε λειτουργικό αλλά και η σύνδεσή της στο backend μέσω της βιβλιοθήκης sqlalchemy που αναφέρθηκε σε προηγούμενη ενότητα.

1. **Εγκατάσταση της PostgreSQL**

Η εγκατάσταση της PostgreSQL μπορεί να πραγματοποιηθεί μέσω διαφόρων μεθόδων ανάλογα με το λειτουργικό σύστημα που έχει το σύστημα.

Για Ubuntu:

sudo apt update

sudo apt install postgresql postgresql-contrib

Για MacOS (χρησιμοποιώντας Homebrew):

brew update

brew install postgresql

brew services start postgresql

Για Windows:

Μέσω του installer από το [επίσημο site της PostgreSQL](https://www.postgresql.org/download/windows/) και ακολουθώντας τις οδηγίες εγκατάστασης.

2. **Εγκατάσταση GUI pgAdmin 4**

Προκειμένου να γίνεται εφικτή με έναν ευκολότερο τρόπο η επικοινωνία με τη ΒΔ, η εκτέλεση των ερωτημάτων προς της βάση αλλά και η οπτικοποίηση των δεδομένων αυτών, μπορεί να χρησιμοποιηθεί το GUI, pgAdmin 4.

Έχοντας εγκαταστήσει την εφαρμογή, είναι εύκολο να δημιουργηθεί τώρα μία νέα βάση δεδομένων με όνομα myDatabase.

A screenshot of a computer

Description automatically generated

3. **Σύνδεση της PostgreSQL με το SQLAlchemy**

Για την σύνδεση της PostgreSQL από την εφαρμογή του FastAPI, χρησιμοποιείται το SQLAlchemy με το κατάλληλο dependency (psycopg2):

pip install psycopg2-binary

Στο αρχείο app/database.py:

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "postgresql://myuser:mypassword@localhost/**myDatabase**"

engine = create\_engine(SQLALCHEMY\_DATABASE\_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

4. **Δημιουργία πινάκων στη ΒΔ**

Εφόσον έχει γίνει η σύνδεση της ΒΔ με το FastAPI με τη βοήθεια της βιβλιοθήκης sqlalchemy, όταν η εφαρμογή τρέξει τότε αυτόματα θα δημιουργηθούν οι πίνακες που έχουν οριστεί στο αρχείο app/models.py.

from sqlalchemy import Column, Integer, String

from .database import Base

class Item(Base):

\_\_tablename\_\_ = "items"

id = Column(Integer, primary\_key=True, index=True)

title = Column(String, index=True)

description = Column(String, index=True)

Στο συγκεκριμένο παράδειγμα δημιουργείται ο πίνακας items μέσα στη ΒΔ myDatabase της PostgreSQL.

**Κεφ.5: Χρήση του Git μέσω κοινών εντολών**

Ακολουθούν μερικές από τις πιο κοινές εντολές που χρησιμοποιούνται στο Git:

Ρύθμιση Git σε ένα νέο έργο:

Στο directory που βρίσκεται το έργο, μπορούν να εκτελεστούν οι παρακάτω εντολές.

# Αρχικοποίηση ενός νέου αποθετηρίου Git

git init

# Προσθήκη αρχείων στο αποθετήριο

git add .

# Δημιουργία πρώτου commit

git commit -m "Initial commit"

Δημιουργία και συγχώνευση branches:

# Δημιουργία νέου κλάδου

git branch new-feature

# Μετακίνηση σε νέο κλάδο

git checkout new-feature

# Συγχώνευση κλάδου στην κύρια γραμμή

git checkout main

git merge new-feature

**Κεφ.6: Χρήση του GitHub**

Ακολουθούν μερικές από τις πιο κοινές ενέργειες που πραγματοποιούνται στο GitHub:

Δημιουργία αποθετηρίου και push στον GitHub:

# Δημιουργία νέου αποθετηρίου στο GitHub μέσω της ιστοσελίδας

# Σύνδεση του τοπικού αποθετηρίου με το GitHub

git remote add origin https://github.com/username/repository.git

# Αποστολή του τοπικού αποθετηρίου στον GitHub

git push -u origin main

Δημιουργία pull request:

1. Αρχικά, πραγματοποιείται ένα commit και έπειτα ένα push τις αλλαγές που έγιναν σε έναν νέο κλάδο:

git checkout -b new-feature

git add .

git commit -m "Add new feature"

git push origin new-feature

2. Στη συνέχεια, μέσω της ιστοσελίδας του GitHub ολοκληρώνεται το pull request με τις παρακάτω ενέργειες:

* Μετάβαση στο αποθετήριο.
* Κλικ στο κουμπί "Compare & pull request".
* Προσθήκη περιγραφής και εκτέλεση του pull request.

**Κεφ.7: Setup του project και η δομή του κώδικα (backend)**

Τα παρακάτω βήματα αφορούν στο στήσιμο (setup) του project στο backend:

1. Αρχικά θα πρέπει να είναι εγκαταστημένη στο σύστημα η Python. Αν δεν είναι, τότε θα πρέπει να γίνει εγκατάστασή της. Η έκδοση της Python που χρησιμοποιήθηκε για το εν λόγω project είναι η Python 3.12.3. Η έκδοση μπορεί να φανεί εκτελώντας στο terminal την εντολή

python --version

Αν το output επιστρέψει για παράδειγμα

Python 3.12.3

τότε η Python είναι εγκατεστημένη ήδη στο σύστημα.

2. Το project θα πρέπει να φιλοξενηθεί σε ένα directory σε κάποιο σημείο στον υπολογιστή δίνοντας ένα όνομα, όπως π.χ recipe-treasure-foodex-fullstack-project. Στο path C:\Users\tziac\Workspace εκτελείται η εντολή

C:\Users\tziac\Workspace> mkdir recipe-treasure-foodex-fullstack-project

και με την εντολή

C:\Users\tziac\Workspace> cd recipe-treasure-foodex-fullstack-project

γίνεται η μεταφορά μέσα στο directory του project.

3. Για να διασφαλιστεί ότι οι εξαρτήσεις (dependencies) του project είναι απομονωμένες από το υπόλοιπο σύστημα, θεωρείται καλή πρακτική να δημιουργηθεί ένα εικονικό περιβάλλον Python. Με την εντολή

C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project> python -m venv venv

δημιουργείται το εικονικό περιβάλλον venv.

Το εικονικό περιβάλλον θα πρέπει να ενεργοποιηθεί, κάτι που επιτυγχάνεται με την παρακάτω εντολή

C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project>.\venv\Scripts\activate

To output στο terminal είναι

(venv) PS C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project>

4. Έπειτα, γίνεται η εγκατάσταση όλων των απαραίτητων για το project dependencies, με την εντολή

(venv) PS C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project>pip install –r requirements.txt

Το αρχείο requirements.txt περιέχει όλα τα dependencies που χρειάζονται στην εφαρμογή και είναι καλή πρακτική να δημιουργείται για να μπορέσει να εκτελεστεί η παραπάνω εντολή. Το αρχείο με όλα τα dependencies που χρησιμοποιήθηκαν, βρίσκεται εδώ

<https://github.com/tziachar/recipe-treasure-foodex-fullstack-project/blob/master/requirements.txt>

οπότε μπορεί να αντιγραφεί στο path C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project για να τρέξει σωστά η εντολή pip install –r requirements.txt.

5. Έχοντας ολοκληρώσει την εγκατάσταση των dependencies, ακολουθεί η δημιουργία της βασικής δομής του project χρησιμοποιώντας τις παρακάτω εντολές

(venv) PS C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project>

mkdir backend

(venv) PS C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project>

cd backend

(venv) PS C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project\ backend>

# Δημιουργία βασικών φακέλων και αρχείων του backend

mkdir source

touch database.py

touch main.py

touch models.py

touch schemas.py

touch services.py

touch similarity\_search\_service.py

touch secret.env

Στο directory source θα πρέπει να περιέχονται κάποια αρχεία, τα οποία μπορούν να γίνουν download από εδώ

<https://github.com/tziachar/recipe-treasure-foodex-fullstack-project/tree/master/backend/source>

Το περιεχόμενο του file secret.env μπορεί να αποτελείται από τα εξής:

ADMIN\_USERNAME=admin

ADMIN\_PASSWORD=admin

ADMIN\_EMAIL=admin@example.com

Η δομή των φακέλων και των αρχείων θα πρέπει να είναι όπως στην εικόνα που ακολουθεί.

A screenshot of a computer

Description automatically generated

Με αυτά τα βήματα έχει ολοκληρωθεί το βασικό structure του project στο backend.

Σημείωση: Το \_\_pycache\_\_ μέσα στον φάκελο backend, είναι ένας φάκελος που δημιουργείται αυτόματα όταν τρέξει η εφαρμογή και εκτελεστεί ο κώδικας. Περιέχει αρχεία bytecode που έχουν μεταγλωττιστεί από τον Python interpreter. Αυτά τα αρχεία έχουν την επέκταση .pyc και είναι μεταγλωττισμένες εκδόσεις των αρχείων .py (Python source files).

**Κεφ.7.1:Modules και παραδείγματα API Endpoints**

Στην ενότητα αυτή θα εξεταστούν ορισμένα από τα βασικότερα modules της εφαρμογής δηλαδή τα διακριτά τμήματα του κώδικα μέσα στη δομή του project που εξυπηρετούν συγκεκριμένες λειτουργίες.

**Database**

Στο FastAPI είναι εφικτό να μπορεί να χρησιμοποιηθεί οποιαδήποτε βάση δεδομένων η οποία υποστηρίζεται από τη βιβλιοθήκη SQLAlchemy. Κατά την ανάπτυξη του project, αρχικά χρησιμοποιήθηκε η SQLite δοκιμαστικά και σε δεύτερο χρόνο επιλέχτηκε τελικώς η PostgreSQL η οποία παρέχει μεγαλύτερο χώρο και καλύτερη διαχείριση μεγάλων όγκων δεδομένων σε σύγκριση με την SQLite.

Στο αρχείο **database.py** χρησιμοποιείται η βιβλιοθήκη SQLAlchemy της Python που παρέχει μια πλήρη λύση ORM (Object Relational Mapping) για τη διαχείριση της βάσης δεδομένων με αντικειμενοστραφή τρόπο, καθώς και ένα σύνολο εργαλείων για τη δημιουργία της σύνδεσης με τη βάση.

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker, declarative\_base

DATABASE\_URL = "postgresql://postgres:ΧΧΧΧΧΧΧ@localhost:5432/RecipesTreasure"

# Σύνδεση με τη βάση δεδομένων

engine = create\_engine(DATABASE\_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Βάση για τη δημιουργία των κλάσεων

Base = declarative\_base()

# Dependency

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

Επισημάνσεις για το Dependency:

* db = SessionLocal(): Δημιουργεί μια νέα συνεδρία της βάσης δεδομένων.
* yield db: Επιστρέφει τη συνεδρία στη λειτουργία που την καλεί και "παγώνει" την εκτέλεση της συνάρτησης get\_db.
* finally: db.close(): Όταν η καλούσα λειτουργία ολοκληρωθεί, η συνάρτηση get\_db συνεχίζει και κλείνει τη συνεδρία της βάσης δεδομένων.

Οι κλάσεις των models που θα δημιουργηθούν στην συνέχεια, κληρονομούν την κλάση Base που δημιουργήθηκε παραπάνω.

**Models**

Σε αυτό το σημείο θα χρησιμοποιηθεί η κλάση Base προκειμένου να δημιουργηθούν τα SQLAlchemy models. Τα models αναπαριστούν του πίνακες όπως αυτοί ορίστηκαν στο σχήμα της ΒΔ σε προηγούμενο κεφάλαιο. Η αντιστοίχιση αυτή πραγματοποιείται στο αρχείο models.py όπως φαίνεται παρακάτω. Σημειώνεται ότι έχουν προστεθεί σχόλια για την καλύτερη κατανόηση του αναγνώστη.

from sqlalchemy import Column, Float, Integer, String, ForeignKey, Table, Boolean, JSON, Text # Για τον ορισμό των πεδίων των πινάκων και του τύπου δεδομένων τους

from sqlalchemy.orm import relationship # Για τον ορισμό της σχέσης μεταξύ των πινάκων

from database import Base # Αποτελεί την κλάση την οποία κληρονομούν οι κλάσεις που αναπαριστούν τους πίνακες της βάσης

# Δημιουργία κλάσης του βοηθητικού πίνακα ingredient\_recipe όπως προκύπτει από την σχέση πολλά προς πολλά μεταξύ των πινάκων recipes και ingredients.

class IngredientRecipe(Base):

\_\_tablename\_\_ = 'ingredient\_recipe'

recipe\_id = Column(Integer, ForeignKey("recipes.id"), primary\_key=True)

ingredient\_id = Column(Integer, ForeignKey("ingredients.id"), primary\_key=True)

amount = Column(String, nullable=False)

# Δημιουργία κλάσης του βοηθητικού πίνακα processed\_ingredient\_recipe όπως προκύπτει από την σχέση πολλά προς πολλά μεταξύ των πινάκων recipes και processed\_ingredients.

class ProcessedIngredientRecipe(Base):

\_\_tablename\_\_ = 'processed\_ingredient\_recipe'

recipe\_id = Column(Integer, ForeignKey("recipes.id"), primary\_key=True)

processed\_ingredient\_id = Column(Integer, ForeignKey("processed\_ingredients.id"), primary\_key=True)

# Δημιουργία κλάσης του πίνακα users και των πεδίων του και η έκφραση της σχέσης ένα προς πολλά μεταξύ του users με τους πίνακες ingredients, recipes και processed\_ingredients αντίστοιχα.

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True, index=True)

username = Column(String, unique=False, index=True)

email = Column(String, unique=True, index=True)

hashed\_password = Column(String)

is\_admin = Column(Boolean, default=False)

ingredients = relationship("Ingredient", back\_populates="owner")

recipes = relationship("Recipe", back\_populates="owner")

processed\_ingredients = relationship("ProcessedIngredient", back\_populates="owner")

# Δημιουργία κλάσης του πίνακα ingredients και των πεδίων του και η έκφραση της σχέσης ένα προς πολλά μεταξύ του ingredients με τον πίνακα users και της σχέσης πολλά προς πολλά με τον πίνακα recipes.

class Ingredient(Base):

\_\_tablename\_\_ = "ingredients"

id = Column(Integer, primary\_key=True, index=True)

ingredient\_name = Column(String, index=True)

ingredient\_base\_term = Column(String, nullable=True)

ingredient\_base\_term\_name = Column(String, nullable=True)

ingredient\_with\_facets = Column(JSON, nullable=True)

place\_of\_origin = Column(String, nullable=True)

is\_POP = Column(Boolean, nullable=True)

is\_PGE = Column(Boolean, nullable=True)

latin\_name = Column(String, nullable=True)

greek\_name = Column(String, nullable=True)

greek\_alter\_name = Column(String, nullable=True)

description = Column(JSON, nullable=True)

final\_Foodex\_code = Column(String, nullable=True)

owner\_id = Column(Integer, ForeignKey("users.id"))

foodex\_Code\_for\_recipes = Column(String, index=True)

owner = relationship("User", back\_populates="ingredients")

recipes = relationship("Recipe", secondary="ingredient\_recipe", back\_populates="ingredients", lazy='subquery')

# Δημιουργία κλάσης του πίνακα processed\_ingredients και των πεδίων του και η έκφραση της σχέσης ένα προς πολλά μεταξύ του processed\_ingredients με τον πίνακα users και της σχέσης πολλά προς πολλά με τον πίνακα recipes.

class ProcessedIngredient(Base):

\_\_tablename\_\_ = "processed\_ingredients"

id = Column(Integer, primary\_key=True, index=True)

processed\_ingredient\_name = Column(String, index=True)

processed\_ingredient\_base\_term = Column(String, nullable=True)

processed\_ingredient\_base\_term\_name = Column(String, nullable=True)

processed\_ingredient\_with\_facets = Column(JSON, nullable=True)

description = Column(JSON, nullable=True)

final\_Foodex\_code = Column(String, nullable=True)

owner\_id = Column(Integer, ForeignKey("users.id"))

owner = relationship("User", back\_populates="processed\_ingredients")

recipes = relationship(

"Recipe",

secondary="processed\_ingredient\_recipe",

back\_populates="processed\_ingredients",

lazy='subquery'

)

# Δημιουργία κλάσης του πίνακα recipes και των πεδίων του και η έκφραση της σχέσης ένα προς πολλά μεταξύ του recipes με τον πίνακα users και της σχέσης πολλά προς πολλά με τους πίνακες ingredients και processed\_ingredients αντίστοιχα.

class Recipe(Base):

\_\_tablename\_\_ = "recipes"

id = Column(Integer, primary\_key=True, index=True)

recipe\_name = Column(String, index=True)

recipe\_base\_term = Column(String, nullable=True)

recipe\_base\_term\_name = Column(String, nullable=True)

recipe\_with\_facets = Column(JSON, nullable=True)

place\_of\_origin = Column(String, nullable=True)

is\_POP = Column(Boolean, nullable=True)

is\_PGE = Column(Boolean, nullable=True)

mediterranean\_diet\_category = Column(String, nullable=True)

nova\_system\_category = Column(String, nullable=True)

latin\_name = Column(String, nullable=True)

greek\_name = Column(String, nullable=True)

greek\_alter\_name = Column(String, nullable=True)

instructions = Column(String, nullable=True)

description = Column(JSON, nullable=True)

processed\_ingredients\_info = Column(JSON, nullable=True)

final\_Foodex\_code = Column(String, nullable=True)

owner\_id = Column(Integer, ForeignKey("users.id"))

owner = relationship("User", back\_populates="recipes")

ingredients = relationship("Ingredient", secondary="ingredient\_recipe", back\_populates="recipes", lazy='subquery')

processed\_ingredients = relationship(

"ProcessedIngredient",

secondary="processed\_ingredient\_recipe",

back\_populates="recipes",

lazy='subquery'

)

Με αυτό τον τρόπο επετεύχθη το mapping μεταξύ των κλάσεων και των πινάκων του σχήματος ορίζοντας ως attributes των κλάσεων τα αντίστοιχα πεδία των πινάκων ενώ παράλληλα ορίστηκαν οι σχέσεις μεταξύ των, με τη βοήθεια της λειτουργίας relationship που παρέχεται από την βιβλιοθήκη sqlalchemy.

**Schemas**

Στο αρχείο schemas.py ορίζονται με την χρήση της βιβλιοθήκης pydantic τα Pydantic models που χρησιμοποιούνται για την επικύρωση - validation και τη δημιουργία και το διάβασμα των δεδομένων που εισάγονται ή εξάγονται μέσω των endpoints του FastAPI από άλλες εφαρμογές (όπως η εφαρμογή της React που αποτελεί το frontend του project και που θα εξεταστεί στην επόμενη ενότητα αυτού του κεφαλαίου). Με άλλα λόγια τα models αυτά βοηθούν στο να εξασφαλιστεί ότι τα δεδομένα έχουν το σωστό σχήμα και τύπο, μειώνοντας τα σφάλματα και βελτιώνοντας την αξιοπιστία της εφαρμογής.

Για την αποφυγή της σύγχυσης μεταξύ των models SQLAlchemy και των models Pydantic, είναι καλή πρακτική να δημιουργούνται δύο διαφορετικά αρχεία, το αρχείο models.py με τα models SQLAlchemy και το αρχείο schemas.py με τα models Pydantic.

Ο ορισμός των Pydantic models που χρησιμοποιήθηκαν στην εφαρμογή φαίνονται στον κώδικα που ακολουθεί.

# Το pydantic.BaseModel αποτελεί τη βάση από όπου τα models κληρονομούν την κλάση BaseModel κατά την αρχικοποίησή τους, το EmailStr αποτελεί έναν ειδικό τύπο δεδομένων για την ορθή διαχείριση των emails και το Field για τον ορισμό επιπλέον παραμέτρων των πεδίων των μοντέλων.

from pydantic import BaseModel, Field, EmailStr

from typing import Dict, List, Optional

class UserBase(BaseModel):

username: str

email: EmailStr

class UserCreate(UserBase):

password: str

class Facet(BaseModel):

name: str

code: str

class FacetForRecipe(Facet):

amount: str

class Description(BaseModel):

user\_input: str

class Processed\_Ingredient\_Info(BaseModel):

processed\_ingredient\_name: str

# amount: Optional[str] = None

final\_Foodex\_code: str

class IngredientBase(BaseModel):

ingredient\_name: str

ingredient\_base\_term: Optional[str] = None

ingredient\_base\_term\_name: Optional[str] = None

ingredient\_with\_facets: List[Facet] = Field(default\_factory=list)

place\_of\_origin: Optional[str] = None

is\_POP: Optional[bool] = None

is\_PGE: Optional[bool] = None

latin\_name: Optional[str] = None

greek\_name: Optional[str] = None

greek\_alter\_name: Optional[str] = None

description: Optional[List[Description]] = Field(default\_factory=list)

class IngredientCreate(IngredientBase):

pass

class IngredientOut(IngredientBase):

id: int

final\_Foodex\_code: Optional[str] = None

foodex\_Code\_for\_recipes: str

class Config:

from\_attributes = True

class ProcessedIngredientBase(BaseModel):

processed\_ingredient\_name: str

processed\_ingredient\_base\_term: Optional[str] = None

processed\_ingredient\_base\_term\_name: Optional[str] = None

processed\_ingredient\_with\_facets: List[Facet] = Field(default\_factory=list)

description: Optional[List[Description]] = Field(default\_factory=list)

class ProcessedIngredientCreate(ProcessedIngredientBase):

pass

class ProcessedIngredientOut(ProcessedIngredientBase):

id: int

final\_Foodex\_code: Optional[str] = None

class Config:

from\_attributes = True

class RecipeBase(BaseModel):

recipe\_name: str

recipe\_base\_term: Optional[str] = None

recipe\_base\_term\_name: Optional[str] = None

recipe\_with\_facets: List[FacetForRecipe] = Field(default\_factory=list)

place\_of\_origin: Optional[str] = None

is\_POP: Optional[bool] = None

is\_PGE: Optional[bool] = None

mediterranean\_diet\_category: Optional[str] = None

nova\_system\_category: Optional[str] = None

latin\_name: Optional[str] = None

greek\_name: Optional[str] = None

greek\_alter\_name: Optional[str] = None

instructions: Optional[str] = None

description: Optional[List[Description]] = Field(default\_factory=list)

processed\_ingredients\_info: Optional[List[Processed\_Ingredient\_Info]] = Field(default\_factory=list)

class RecipeCreate(RecipeBase):

pass

class RecipeOut(RecipeBase):

id: int

ingredients: List[IngredientOut] = []

processed\_ingredients: List[ProcessedIngredientOut] = []

final\_Foodex\_code: Optional[str] = None

class Config:

from\_attributes = True

class UserOut(UserBase):

id: int

recipes: List[RecipeOut] = []

ingredients: List[IngredientOut] = []

processed\_ingredients: List[ProcessedIngredientOut] = []

class Config:

from\_attributes = True

Για τον ορισμό ενός model ή schema λοιπόν, ακολουθείται η ίδια λογική. Παρακάτω εξηγείται η λογική της δημιουργίας όλων των models που δημιουργήθηκαν για τη διαχείριση των δεδομένων που αφορούν στον user.

Αρχικά, δημιουργείται το UserBase μοντέλο το οποίο κληρονομεί από το BaseModel όπως αναφέρθηκε προηγουμένως. Το UserBase αποτελείται από τα βασικά attributes, username και email. Έπειτα δημιουργείται το μοντέλο UserCreate που χρησιμοποιείται για τη δημιουργία ενός user. Όπως φαίνεται, το UserCreate κληρονομεί το model UserBase και επιπλέον προστίθεται ένα ακόμη attribute, το password το οποίο είναι χρήσιμο για τη δημιουργία του user. Έτσι, το password δεν ανήκει σε κανένα άλλο μοντέλο, συνεπώς όταν θα διαβάζεται ένας user στην εφαρμογή, το API δεν θα εμφανίζει στον χρήστη του συστήματος την πληροφορία για το password στα αντίστοιχα responses, αποκρύπτοντας με αυτό τον τρόπο την συγκεκριμένη ευαίσθητη πληροφορία. Το παραπάνω επιτυγχάνεται με τον ορισμό ενός τρίτου model, του UserOut το οποίο κληρονομεί το UserBase ενώ παράλληλα προστίθενται σε αυτό μερικά ακόμα attributes τα οποία είναι θεμιτό να εμφανίζονται στον χρήστη του συστήματος.

Το Swagger UI παρέχει τη δυνατότητα να ελεγχθούν όλα τα Pydantic models ή schemas που χρησιμοποιούνται στο API, όπως αποτυπώνεται στην παρακάτω εικόνα.

A screenshot of a computer

Description automatically generated

Όπως φαίνεται, τα models UserCreate και UserOut διαχειρίζονται τα δεδομένα του user κατά τη δημιουργία ενός user και κατά το διάβασμα των users. Στις παρακάτω εικόνες από το Swagger, γίνεται ακόμα πιο εμφανές το έργο που επιτελούν τα models που εξετάζονται στο παράδειγμα.

Πιο συγκεκριμένα κατά τη δημιουργία του χρήστη, παρατηρείται ότι το request body που στέλνεται στο API, αντιπροσωπεύει το model UserCreate που δημιουργήθηκε.

A screenshot of a computer

Description automatically generated

Πατώντας το κουμπί Execute, επιστρέφεται το response από το API που ακολουθεί τον ορισμό του model UserOut.

A screenshot of a computer

Description automatically generated

Το ίδιο model εφαρμόζεται και όταν πραγματοποιείται ένα αίτημα GET που αναμένεται να επιστραφούν ως response, όλα τα δεδομένα των users που είναι αποθηκευμένοι στο σύστημα. Έτσι, το response σε αυτή την περίπτωση είναι μία λίστα από json αντικείμενα που κάθε αντικείμενο είναι σχηματισμένο βάσει του model UserOut, έχοντας την εξής μορφή.

[ { "username": "Charis T.", "email": "charis@example.com", "id": 4, "recipes": [], "ingredients": [], "processed\_ingredients": [] } ]

**Main Application Module**

Το αρχείο main.py αποτελεί το κύριο αρχείο στο οποίο μεταξύ άλλων ορίζεται η εφαρμογή, τα endpoints και το middleware. Ακολουθούν βασικά σημεία του κώδικα τα οποία εξηγούνται αναλυτικά.

from typing import List

from fastapi import FastAPI, Depends, HTTPException, Query

from sqlalchemy.orm import Session

import torch

from database import engine, Base, get\_db

import schemas

import services

import similarity\_search\_service as search

from fastapi.middleware.cors import CORSMiddleware

from sqlalchemy.orm.exc import ObjectDeletedError

from transformers import BertTokenizer, BertModel

Παραπάνω φαίνονται οι βιβλιοθήκες, τα dependencies και τα εργαλεία που χρησιμοποιούνται καθώς και τα imports όλων των προηγούμενων modules που εξετάστηκαν προηγουμένως. Μερικά από τα σημαντικότερα, αφορούν στα εξής:

* FastAPI: Η κύρια κλάση για τη δημιουργία της εφαρμογής.
* Session: Εισάγεται η κλάση Session για τη διαχείριση των συνεδριών της βάσης δεδομένων.
* CORSMiddleware: Χρησιμοποιείται για τη διαχείριση των CORS (Cross-Origin Resource Sharing), επιτρέποντας αιτήσεις από άλλες εφαρμογές.

app = FastAPI()

origins = [

"<http://localhost>",

"<http://localhost:3000>",

"<http://localhost:3000/login>"

]

app.add\_middleware(

CORSMiddleware,

allow\_origins=origins,

allow\_credentials=True,

allow\_methods=["GET", "POST", "PUT", "DELETE"],

allow\_headers=["\*"],

)

# Δημιουργία όλων των πινάκων της βάσης

Base.metadata.create\_all(bind=engine)

Στον παραπάνω κώδικα το app όπως ειπώθηκε αποτελεί το κύριο αντικείμενο της εφαρμογής στο οποίο ορίζονται όλες οι λειτουργίες του συστήματος. Η λίστα origins περιέχει όλες τις προελεύσεις (την React εν προκειμένω που by default "ακούει” στο port 3000) από όπου μπορούν να γίνουν requests στο API ενώ στις παραμέτρους του middleware CORSMiddleWare που προστίθεται στο app, καθορίζονται μεταξύ άλλων, τα origins και οι HTTP μέθοδοι ("GET", "POST", "PUT", "DELETE") που θα επιτρέπονται από τα συγκεκριμένα origins ως αιτήσεις - requests.

Τέλος, η χρήση του Base.metadata.create\_all(bind=engine) επιτρέπει τη δημιουργία όλων των πινάκων της ΒΔ όπως ορίζονται από τα μοντέλα SQLAlchemy που αναφέρθηκαν προηγουμένως. Με άλλα λόγια ο παραπάνω κώδικας δημιουργεί και ρυθμίζει μια εφαρμογή FastAPI, επιτρέποντας σε συγκεκριμένες προελεύσεις να κάνουν αιτήσεις μέσω του CORS, διασφαλίζοντας ότι όλοι οι πίνακες της βάσης δεδομένων είναι δημιουργημένοι πριν η εφαρμογή αρχίσει να δέχεται αιτήσεις.

Επιπλέον, στην εφαρμογή έχουν υλοποιηθεί αρκετά endpoints τα οποία επιτελούν διάφορες λειτουργίες που σχετίζονται με τους χρήστες - users, τα συστατικά - ingredients, τα επεξεργασμένα συστατικά - proccesed\_ingredients και τις συνταγές - recipes. Ο τρόπος που ορίζεται ένα endpoint στο FastAPI είναι ο εξής:

Για να εξηγηθεί ο τρόπος δημιουργίας ενός endpoint, χρησιμοποιείται ως παράδειγμα το παρακάτω endpoint.

@app.post("/users/", response\_model=schemas.UserOut, tags=["users"])

def create\_user(user: schemas.UserCreate, db: Session = Depends(get\_db)):

return services.create\_user(db, user)

Βήμα 1: Ορισμός της διαδρομής (Path definition)

@app.post("/users/", response\_model=schemas.UserOut, tags=["users"])

* @app.post("/users/"): Το @app.post είναι ένα decorator που δηλώνει ότι αυτή η λειτουργία (function) θα χειρίζεται HTTP POST αιτήματα στη διαδρομή "/users/". Το FastAPI παρέχει διάφορους decorators για διαφορετικούς τύπους αιτημάτων, όπως @app.get, @app.put, @app.delete, κ.λπ.
* response\_model=schemas.UserOut: Η επιλογή response\_model ορίζει το Pydantic model που θα χρησιμοποιηθεί για την επικύρωση και την τεκμηρίωση της απόκρισης - response. Στο παράδειγμα αυτό δηλαδή, το schemas.UserOut είναι ένα Pydantic model που καθορίζει το format των δεδομένων που θα επιστραφούν.
* tags=["users"]: Οι ετικέτες (tags) χρησιμοποιούνται για την οργάνωση και την τεκμηρίωση του API, όπως εμφανίζονται στο Swagger UI. Στο παράδειγμα αυτό, η διαδρομή αυτή θα ενταχθεί στην κατηγορία "users".

Βήμα 2: Ορισμός της λειτουργίας (Function definition)

def create\_user(user: schemas.UserCreate, db: Session = Depends(get\_db)):

* def create\_user(...): Η function έχει το όνομα create\_user. Αυτό το όνομα χρησιμοποιείται εσωτερικά στον κώδικα και δεν επηρεάζει την εξωτερική συμπεριφορά του API.
* user: schemas.UserCreate: Η παράμετρος (user) είναι του τύπου δεδομένων schemas.UserCreate δηλαδή ενός Pydantic model για την επικύρωση και την αναπαράσταση των δεδομένων εισόδου. Το FastAPI θα διαβάσει τα δεδομένα από το σώμα του αιτήματος (request body) και θα προσπαθήσει να τα ταιριάξει με το σχήμα UserCreate.
* db: Session = Depends(get\_db): Η παράμετρος (db) αφορά σε μια συνεδρία (session) της βάσης δεδομένων που δημιουργείται χρησιμοποιώντας το FastAPI Depends. Το get\_db είναι μια συνάρτηση που εξηγεί πώς δημιουργείται η συνεδρία της βάσης δεδομένων.

Βήμα 3: Λογική της λειτουργίας (Function logic)

return services.create\_user(db, user)

Η συνάρτηση καλεί την services.create\_user (που θα αναλυθεί παρακάτω, στο σημείο που εξετάζεται το services module), περνώντας ως ορίσματα τη συνεδρία της βάσης δεδομένων (db) και το user αντικείμενο σε αυτή. Το αποτέλεσμα αυτής της κλήσης επιστρέφεται ως response του API σε αυτόν που την καλεί.

Ακολουθώντας την παραπάνω λογική, ορίστηκαν τα ακόλουθα endpoints για την εφαρμογή:

**Endpoints Χρηστών (Users)**

1. Login

@app.post("/login", tags=["users"])

* Περιγραφή: Επιτρέπει σε έναν χρήστη να συνδεθεί.
* Επιστρέφει: user\_id και is\_admin (true ή false).

1. Create User

@app.post("/users/", response\_model=schemas.UserOut, tags=["users"])

* Περιγραφή: Δημιουργεί έναν νέο χρήστη.
* Επιστρέφει: Τον χρήστη που δημιουργήθηκε.

**Endpoints λειτουργιών του Admin**

1. Read Users

@app.get("/users/", response\_model=List[schemas.UserOut], tags=["admin"])

* Περιγραφή: Προβάλει όλους τους χρήστες.
* Επιστρέφει: Λίστα χρηστών.

1. Delete User

@app.delete("/users/{user\_id}", tags=["admin"])

* Περιγραφή: Διαγράφει έναν χρήστη βάσει του id του.
* Επιστρέφει: Μήνυμα επιτυχίας διαγραφής ή σφάλμα αν δεν βρεθεί ο χρήστης.

1. Delete Ingredient

@app.delete("/ingredients/{ingredient\_id}", tags=["admin"])

* Περιγραφή: Διαγράφει ένα συστατικό.
* Επιστρέφει: Μήνυμα επιτυχίας διαγραφής ή σφάλμα αν δεν βρεθεί το συστατικό.

1. Delete Processed Ingredient

@app.delete("/processed\_ingredients/{processed\_ingredient\_id}", tags=["admin"])

* Περιγραφή: Διαγράφει ένα επεξεργασμένο συστατικό.
* Επιστρέφει: Μήνυμα επιτυχίας διαγραφής ή σφάλμα αν δεν βρεθεί το επεξεργασμένο συστατικό.

1. Delete Recipe

@app.delete("/recipes/{recipe\_id}", tags=["admin"])

* Περιγραφή: Διαγράφει μια συνταγή βάσει του id της.
* Επιστρέφει: Μήνυμα επιτυχίας διαγραφής ή σφάλμα αν δεν βρεθεί η συνταγή.

1. Update ingredient

@app.put("/ingredients/{ingredient\_id}",response\_model=schemas.IngrtedientOut, tags=["admin"])

* Περιγραφή: Ανανέωση ενός συστατικού βάσει του id του.
* Επιστρέφει: Μήνυμα επιτυχίας ανανέωσης ή σφάλμα αν δεν βρεθεί το συστατικό.

**Endpoints Συστατικών (Ingredients)**

1. Create Ingredient

@app.post("/ingredients/", response\_model=schemas.IngredientOut, tags=["ingredients"])

* Περιγραφή: Δημιουργεί ένα νέο συστατικό.
* Επιστρέφει: Το συστατικό που δημιουργήθηκε.

1. Update Ingredient Description

@app.put("/ingredients/{ingredient\_id}/description/{description}", response\_model=schemas.IngredientOut, tags=["ingredients"])

* Περιγραφή: Ενημερώνει την περιγραφή ενός συστατικού.
* Επιστρέφει: Το ενημερωμένο συστατικό.

1. Read Ingredient

@app.get("/ingredients/{ingredient\_id}", response\_model=schemas.IngredientOut, tags=["ingredients"])

* Περιγραφή: Επιστρέφει ένα συγκεκριμένο συστατικό βάσει του id του.
* Επιστρέφει: Το συστατικό.

1. Read Ingredients by User

@app.get("/ingredients/user/{user\_id}", response\_model=List[schemas.IngredientOut], tags=["ingredients"])

* Περιγραφή: Επιστρέφει όλα τα συστατικά ενός συγκεκριμένου χρήστη.
* Επιστρέφει: Λίστα συστατικών.

**Endpoints Επεξεργασμένων Συστατικών (Processed Ingredients)**

1. Create Processed Ingredient

@app.post("/processed\_ingredients/",response\_model=schemas.ProcessedIngredientOut, tags=["processed\_ingredients"])

* Περιγραφή: Δημιουργεί ένα νέο επεξεργασμένο συστατικό.
* Επιστρέφει: Το επεξεργασμένο συστατικό που δημιουργήθηκε.

1. Read Processed Ingredient

@app.get("/processed\_ingredients/{user\_id}",response\_model=List[schemas.ProcessedIngredientOut], tags=["processed\_ingredients"])

* Περιγραφή: Επιστρέφει όλα τα επεξεργασμένα συστατικά ενός συγκεκριμένου χρήστη.
* Επιστρέφει: Λίστα επεξεργασμένων συστατικών.

**Endpoints Συνταγών (Recipes)**

1. Create Recipe

@app.post("/recipes/", response\_model=schemas.RecipeOut, tags=["recipes"])

* Περιγραφή: Δημιουργεί μια νέα συνταγή.
* Επιστρέφει: Τη συνταγή που δημιουργήθηκε.

1. Update Recipe Description

@app.put("/recipes/{recipe\_id}/description/{description}", tags=["recipes"])

* Περιγραφή: Ενημερώνει την περιγραφή μιας συνταγής.
* Επιστρέφει: Το ενημερωμένο συστατικό.

1. Read Recipes

@app.get("/recipes/{user\_id}", response\_model=List[schemas.RecipeOut], tags=["recipes"])

* Περιγραφή: Επιστρέφει όλες τις συνταγές ενός συγκεκριμένου χρήστη.
* Επιστρέφει: Λίστα συνταγών.

1. Read Recipes from Region

@app.get("/recipes/from\_region/{place\_of\_origin}",response\_model=List[schemas.RecipeOut], tags=["recipes"])

* Περιγραφή: Επιστρέφει όλες τις συνταγές από μια συγκεκριμένη περιοχή.
* Επιστρέφει: Λίστα συνταγών.

**Endpoints Αναζήτησης (Search)**

1. Search Terms

@app.get("/search\_terms/{term}", tags=["search\_terms"])

* Περιγραφή: Αναζητά όρους και επιστρέφει αποτελέσματα βασισμένα στο μοντέλο αναζήτησης sentence-transformers/paraphrase-multilingual-mpnet-base-v2.
* Επιστρέφει: Λίστα αποτελεσμάτων αναζήτησης.

1. Search Ingredient

@app.get("/search\_ingredient/", tags=[" search\_terms "])

* Περιγραφή: Αναζητά συστατικά στη βάση δεδομένων και επιστρέφει τα ενσωματωμένα αναπαραστάσεις του κειμένου από το Greek-BERT.
* Επιστρέφει: Λίστα συστατικών.

1. Search Processed Ingredient

@app.get("/search\_processed\_ingredient/", tags=[" search\_terms "])

* Περιγραφή: Αναζητά επεξεργασμένα συστατικά στη βάση δεδομένων και επιστρέφει τα ενσωματωμένα αναπαραστάσεις του κειμένου από το Greek-BERT.
* Επιστρέφει: Λίστα επεξεργασμένων συστατικών.

1. Search Recipe

@app.get("/search\_recipe/", tags=[" search\_terms "])

* Περιγραφή: Αναζητά συνταγές στη βάση δεδομένων και επιστρέφει τα ενσωματωμένα αναπαραστάσεις του κειμένου από το Greek-BERT.
* Επιστρέφει: Λίστα συνταγών.

Παρακάτω στις δύο εικόνες, με τη βοήθεια του Swagger UI, φαίνονται συνοπτικά όλα τα endpoints της εφαρμογής, όπως περιγράφηκαν προηγουμένως.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Similarity search service Module**

Στο αρχείο similarity\_search\_service.py υλοποιείται μια διαδικασία που διαχειρίζεται όρους (terms) από ένα αρχείο Excel (που αφορά σε όλα δεδομένα της EFSA) και δημιουργεί δείκτες (indexes) που βοηθούν στην αποδοτικότερη αναζήτηση των όρων που δίνει ο χρήστης ως παράμετρο κάθε φορά όταν καλεί το συγκεκριμένο API, χρησιμοποιώντας τη βιβλιοθήκη FAISS (Facebook AI Similarity Search) και το Sentence Transformer μοντέλο για την εξαγωγή των embeddings. Έτσι ο χρήστης μπορεί να δώσει ως λέξη κλειδί έναν όρο στα ελληνικά π.χ αυγό και μέσω της διαδικασίας θα του επιστραφεί η λίστα με τους 5 κοντινότερους όρους (στα αγγλικά) σε σχέση με αυτόν που έδωσε. Το endpoint που καλεί ο χρήστης είναι το παρακάτω:

# API endpoint to search terms

@app.get("/search\_terms/{term}", tags=["search\_terms"])

async def search\_terms(term: str):

query\_vector = search.model.encode([term])

D, I = search.term\_index.search(query\_vector, 5)

return [f'{search.alltermcodes[i]}: {search.alltermnames[i]}: {search.alldescriptions[i]}' for i in I[0]]

Η συνάρτηση search\_terms λαμβάνει το όρισμα term από το endpoint και το χρησιμοποιεί για να κάνει την αναζήτηση. Ο όρος term κωδικοποιείται χρησιμοποιώντας το Sentence Transformer μοντέλο (search.model.encode). Εδω το search είναι ένα alias που χρησιμοποιείται στο αρχείο main.py.

import similarity\_search\_service as search

Το αποτέλεσμα είναι μια ενσωμάτωση (embedding) που αναπαριστά τον όρο σε έναν πολυδιάστατο χώρο - vector. Το FAISS χρησιμοποιείται για να εκτελέσει αναζήτηση του embedding (query\_vector) στον search.term\_index. Η συνάρτηση search της βιβλιοθήκης FAISS επιστρέφει δύο πίνακες:

* D: Αποστάσεις των πλησιέστερων όρων από τον όρο αναζήτησης.
* I: Δείκτες (indices) των πλησιέστερων όρων στον πίνακα δεδομένων.

Η αναζήτηση ψάχνει τους 5 πιο σχετικούς όρους (μέσω της παραμέτρου 5) και τελικά χρησιμοποιώντας τους δείκτες από το I[0], δημιουργείται μια λίστα με τους αντίστοιχους κωδικούς από τους πίνακες search.alltermcodes, search.alltermnames και search.alldescriptions και κάθε αποτέλεσμα διαμορφώνεται στη μορφή '{termCode}: {termName}: {description}' και επιστρέφεται ως λίστα από strings.

Ακολουθούν τα βήματα που εξηγούν το πως επετεύχθη η ολοκλήρωση της διαδικασίας των αναζητήσεων:

1. Φόρτωση του Sentence Transformer μοντέλου

# Load the sentence model

model = SentenceTransformer('sentence-transformers/paraphrase-multilingual-mpnet-base-v2')

Τα βασικά dependencies που γίνοται import είναι τα εξής:

import pandas as pd

from sentence\_transformers import SentenceTransformer

import faiss

import os

1. Ανάγνωση του αρχείου Excel

# Function to read Excel file

def read\_excel(file\_path):

return pd.read\_excel(file\_path, sheet\_name=None)

Η συνάρτηση read\_excel διαβάζει ένα αρχείο Excel και επιστρέφει ένα λεξικό (dictionary) όπου τα κλειδιά είναι τα ονόματα των φύλλων (sheets) και οι τιμές είναι τα δεδομένα σε μορφή DataFrame.

1. Επεξεργασία των όρων

# Function to process terms

def process\_terms(term\_df):

term\_df = term\_df[term\_df['deprecated'] == 0]

alltermcodes = term\_df['termCode'].tolist()

alltermnames = (term\_df['termExtendedName'].fillna('') + ' ' + term\_df['termShortName'].fillna('') + ' ' + term\_df['commonNames'].fillna('') + ' ' + term\_df['scientificNames'].fillna('')).tolist()

alldescriptions = term\_df['termScopeNote'].fillna('').tolist()

return alltermcodes, alltermnames, alldescriptions

Η συνάρτηση process\_terms επεξεργάζεται τους όρους από το DataFrame:

* Αφαιρεί όρους που έχουν χαρακτηριστεί ως "deprecated" .
* Δημιουργεί τρεις λίστες:
  1. alltermcodes: Περιέχει όλους τους κωδικούς των όρων.
  2. alltermnames: Περιέχει τα ονόματα των όρων που συνδυάζονται από τα πεδία termExtendedName, termShortName, commonNames, και scientificNames.
  3. alldescriptions: Περιέχει τις περιγραφές των όρων από το πεδίο termScopeNote.

1. Δημιουργία ή φόρτωση του FAISS δείκτη

# Function to load or create FAISS index

def load\_or\_create\_index(index\_file\_path, data, model):

if os.path.exists(index\_file\_path):

return faiss.read\_index(index\_file\_path)

else:

sentence\_embeddings = model.encode(data)

d = sentence\_embeddings.shape[1]

index = faiss.IndexFlatL2(d)

index.add(sentence\_embeddings)

faiss.write\_index(index, index\_file\_path)

return index

Η συνάρτηση load\_or\_create\_index:

* Ελέγχει αν υπάρχει ήδη αποθηκευμένος FAISS δείκτης (index) στον καθορισμένο φάκελο.
* Αν υπάρχει, τον φορτώνει χρησιμοποιώντας το faiss.read\_index.
* Αν δεν υπάρχει, δημιουργεί έναν νέο δείκτη:
  + Κωδικοποιεί τα δεδομένα (data) χρησιμοποιώντας το Sentence Transformer μοντέλο για να δημιουργήσει τις ενσωματώσεις προτάσεων (sentence\_embeddings).
  + Δημιουργεί έναν FAISS δείκτη που βασίζεται σε αυτές τις ενσωματώσεις χρησιμοποιώντας μετρική faiss.IndexFlatL2.
  + Προσθέτει τις ενσωματώσεις στον δείκτη και τον αποθηκεύει στον φάκελο source.

1. Αρχικοποίηση και δημιουργία δεικτών

# Initialize indexes (you need to adjust the 'folder' variable)

folder = 'source/'

file\_path = folder + 'MTX15.xlsx'

dfs = read\_excel(file\_path)

term\_df = dfs['term']

alltermcodes, alltermnames, alldescriptions = process\_terms(term\_df)

term\_index\_file\_path = folder + '/termindex.idx'

term\_index = load\_or\_create\_index(term\_index\_file\_path, alltermnames, model)

desc\_index\_file\_path = folder + '/termdescindex.idx'

desc\_index = load\_or\_create\_index(desc\_index\_file\_path, alldescriptions, model)

* Φορτώνει τα δεδομένα από το αρχείο Excel.
* Επεξεργάζεται τα δεδομένα των όρων για να πάρει τις σχετικές λίστες.
* Δημιουργεί ή φορτώνει δύο FAISS indexes:

1. term\_index: Δείκτης για τα ονόματα των όρων (alltermnames).
2. desc\_index: Δείκτης για τις περιγραφές των όρων (alldescriptions).

Παρακάτω φαίνεται ένα παράδειγμα αναζήτησης, με τη βοήθεια του Swagger UI.

A screenshot of a computer

Description automatically generated

Σημείωση:

Στην εφαρμογή, παρόλο που έχουν δημιουργηθεί δύο indexes, ένα με βάση τον όρο - term και ένα με βάση την περιγραφή - description, χρησιμοποιείται μέσω του endpoint μόνο ο πρώτος καθώς όπως διαπιστώθηκε από τις δοκιμές που έγιναν, δίνει πιο ακριβή αποτελέσματα. Για λόγους παρουσίασης λοιπόν, δεν έχει αφαιρεθεί από τον κώδικα, η δημιουργία του δεύτερου index.

**Services Module**

Στο αρχείο services.py αναπτύσσεται όλη η λογική του τρόπου με τον οποίο γίνεται η αλληλεπίδραση των δεδομένων με την ΒΔ. Περιλαμβάνει μεταξύ άλλων τα CRUD operations, διάφορες βοηθητικές συναρτήσεις και συναρτήσεις που εφαρμόζουν τα NLP μοντέλα τα οποία χρησιμοποιούνται στις αναζητήσεις. Παρακάτω ακολουθούν κάποια από τα βασικότερα σημεία του κώδικα εξηγώντας για κάθε κομμάτι, τον λόγο και τον σκοπό που εξυπηρετούν στη λειτουργικότητα της εφαρμογής.

Σχετικά με τη διαχείριση του user και την ταυτοποίησή του από το σύστημα και για να ελεγχθεί εάν ο user που εισέρχεται στην εφαρμογή είναι ο admin ή όχι ώστε να αποδοθούν σε αυτόν τα σωστά δικαιώματα, χρησιμοποιούνται μεταξύ άλλων και οι βιβλιοθήκες

from dotenv import load\_dotenv

import os

from pathlib import Path

Παράλληλα δημιουργείται το αρχείο secret.env στο οποίο ορίζονται κάποιες μεταβλητές περιβάλλοντος που αποτελούν τα ευαίσθητα δεδομένα, τα credentials του admin.

Οι μεταβλητές περιβάλλοντος φορτώνονται με τον κώδικα που ακολουθεί:

dotenv\_path = Path('./secret.env')

load\_dotenv(dotenv\_path=dotenv\_path)

admin\_username = os.getenv("ADMIN\_USERNAME")

admin\_password = os.getenv("ADMIN\_PASSWORD")

admin\_email=os.getenv("ADMIN\_EMAIL")

* **dotenv\_path = Path('./secret.env')**: Ορίζει τη διαδρομή του αρχείου .env που περιέχει τις μεταβλητές περιβάλλοντος. Σε αυτό το παράδειγμα, το αρχείο βρίσκεται στον ίδιο φάκελο με το module και ονομάζεται secret.env.
* **load\_dotenv(dotenv\_path=dotenv\_path)**: Φορτώνει τις μεταβλητές περιβάλλοντος από το αρχείο .env στη μνήμη, ώστε να είναι διαθέσιμες για χρήση στο πρόγραμμα.
* **admin\_username = os.getenv("ADMIN\_USERNAME")**: Αποθηκεύει τη μεταβλητή περιβάλλοντος ADMIN\_USERNAME στην Python μεταβλητή admin\_username.
* **admin\_password = os.getenv("ADMIN\_PASSWORD")**: Αποθηκεύει τη μεταβλητή περιβάλλοντος ADMIN\_PASSWORD στην Python μεταβλητή admin\_password.
* **admin\_email = os.getenv("ADMIN\_EMAIL")**: Αποθηκεύει τη μεταβλητή περιβάλλοντος ADMIN\_EMAIL στην Python μεταβλητή admin\_email.

Εν συνεχεία με την χρήση της συνάρτησης:

def authenticate\_user(db: Session, username: str, password: str):

if username == admin\_username and password == admin\_password:

db\_user = db.query(User).filter(User.email == admin\_email).first()

if db\_user:

db\_user.is\_admin = True

db.commit()

db.refresh(db\_user)

return db\_user, True

else:

return None, False

user = db.query(User).filter(User.username == username).first()

if user and bcrypt.checkpw(password.encode('utf-8'), user.hashed\_password.encode('utf-8')):

return user, user.is\_admin # Επιστρέφει το αντικείμενο του χρήστη και το is\_admin=False εφόσον πρόκειται για κανονικό user

return None, False # Αν δεν βρεθεί χρήστης με τα δεδομένα διαπιστευτήρια, επιστρέφει False για το is\_admin

επιχειρείται η ταυτοποίηση του user από το σύστημα ενώ ταυτόχρονα διενεργείται έλεγχος για το αν ο user είναι ο admin ή όχι. Η συνάρτηση επιστρέφει δύο τιμές σε κάθε περίπτωση, το αντικείμενο user και μία boolean τιμή (True or False). Θα επιστραφεί True λοιπόν στην περίπτωση που η συνθήκη

if username == admin\_username and password == admin\_password

ικανοποιείται. Έπειτα, διενεργείται ένα query στη βάση που αναζητά τον χρήστη που το email του είναι ίσο με την τιμή της μεταβλητής admin\_email που φορτώθηκε από το αρχείο secret.env

db\_user = db.query(User).filter(User.email == admin\_email).first()

Στον συγκεκριμένο χρήστη γίνεται update στο attribute is\_admin από False που είναι by default (τιμή που παίρνει κατά τη δημιουργία του, όπως ορίστηκε στο models.py) σε True. Οπότε ο χρήστης αυτός θα αναγνωρίζεται πλέον από το σύστημα ως admin.

Η συνάρτηση authenticate\_user του αρχείου services.py καλείται μέσω του endpoint που βρίσκεται στο αρχείο main.py

@app.post("/login", tags=["users"])

def login(username: str, password: str, db: Session = Depends(get\_db)):

user, is\_admin = services.**authenticate\_user**(db, username, password)

if not user:

raise HTTPException(status\_code=401, detail="Invalid credentials")

return {"user\_id": user.id, "is\_admin": is\_admin}

και μέσω της συνάρτησης login όπως φαίνεται παραπάνω στην οποία γίνεται και η διαχείριση του exception στην περίπτωση που η authenticate\_user δεν επιστρέψει κάποιο αντικείμενο user. Στην περίπτωση επιτυχούς κλήσης, επιστρέφεται ως response ένα json με keys το user\_id και το is\_admin. Την τιμή is\_admin διαχειρίζεται στην συνέχεια το frontend αναλόγως. Αν είναι True τότε στην οθόνη Home page της εφαρμογής εμπεριέχονται κάποιες extra λειτουργίες που πρόσβαση σε αυτές έχει μόνο ο admin. Στην επόμενη ενότητα που αναλύονται οι λειτουργίες της React, θα εξεταστεί πιο αναλυτικά ο τρόπος που γίνεται η διαχείριση της τιμής is\_admin στο περιβάλλον της.

Η απεικόνιση της συγκεκριμένης λειτουργικότητας που περιγράφηκε παραπάνω μπορεί να γίνει με τη βοήθεια του Swagger UI εξετάζοντας τρεις περιπτώσεις χρηστών που προσπαθεί το σύστημα να ταυτοποιήσει.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Για τη δημιουργία του χρήστη αντίστοιχα, η συνάρτηση create\_user που έχει οριστεί στο ίδιο αρχείο των services

def create\_user(db: Session, user: UserCreate):

# Check if email already exists

db\_user = db.query(User).filter(User.email == user.email).first()

if db\_user:

raise HTTPException(status\_code=400, detail="Email already registered")

# Check if password already exists

users\_with\_same\_password = db.query(User).all()

for existing\_user in users\_with\_same\_password:

if bcrypt.checkpw(user.password.encode('utf-8'), existing\_user.hashed\_password.encode('utf-8')):

raise HTTPException(status\_code=400, detail="Password already in use")

# Hash the new user's password

hashed\_password = bcrypt.hashpw(user.password.encode('utf-8'), bcrypt.gensalt())

# Create the new user

db\_user = User(username=user.username, email=user.email, hashed\_password=hashed\_password.decode('utf-8'))

db.add(db\_user)

db.commit()

db.refresh(db\_user)

return db\_user

καλείται από το αντίστοιχο endpoint του αρχείου main μέσω της συνάρτησης create\_user (το όνομα των συναρτήσεων και στα δύο αρχεία είναι το ίδιο, άλλα αυτό δεν παίζει κάποιο ρόλο καθώς έχουν οριστεί σε διαφορετικά modules)

@app.post("/users/", response\_model=schemas.UserOut, tags=["users"])

def create\_user(user: schemas.UserCreate, db: Session = Depends(get\_db)):

return services.**create\_user**(db, user)

και επιστρέφει το αντικείμενο user που δημιουργήθηκε, σύμφωνα με τη δομή που ορίστηκε στο αρχείο schemas, δηλαδή την κλάση UserOut. Οι παράμετροι της συνάρτησης create\_user των services είναι το αντικείμενο user (όπως αυτό έχει οριστεί στο module schemas με την κλάση UserCreate) και το αντικείμενο db (αναλύθηκε σε προηγούμενη ενότητα). Εντός της συνάρτησης διενεργείται η λογική της προσθήκης του user στη ΒΔ αφού προηγουμένως έχει γίνει έλεγχος των δεδομένων που αφορούν στις πληροφορίες του user που έχουν περαστεί στο request body. Αρχικά ελέγχεται αν στη ΒΔ υπάρχουν άλλοι users με το ίδιο email και έπειτα με το ίδιο password. Αν δεν υπάρχουν τότε με τη βοήθεια της βιβλιοθήκης bcrypt γίνεται hashed του password του user που πρόκειται να δημιουργηθεί και να γίνει η εισαγωγή στη ΒΔ όπου το password πλέον είναι encrypted. Επιπλέον, γίνεται διαχείριση των exceptions στην περίπτωση που οι συνθήκες είναι αναληθής όπου σε αυτή την περίπτωση το σύστημα στέλνει στο σώμα του response κατάλληλο μήνυμα σε μορφή json και status code = 400 στον browser.

Στην συνέχεια εξετάζονται τρία ακόμα services που υλοποιούν τη λειτουργία της καταχώρησης ενός υλικού - ingredient στη ΒΔ, τον σχηματισμό του Foodex Code του υλικού και το update του description ενός υλικού που ήδη υπάρχει στη ΒΔ.

Ο σκοπός είναι να αναδειχθεί ο τρόπος με τον οποίο γίνονται οι συγκεκριμένες API κλήσεις από το UI στο FastAPI και πώς επιτυγχάνεται η επικοινωνία μεταξύ του frontend και του backend. Πως δηλαδή ένα exception σε μία κλήση μπορεί να πυροδοτήσει την επόμενη API κλήση. Αν για παράδειγμα το υλικό υπάρχει στη ΒΔ τότε γίνεται raise ένα exception, το οποίο λαμβάνει το frontend ως not ok response. Μέσα στο αντικείμενο του exception περιέχεται εκτός του status\_code και ένα μήνυμα με τη μορφή json που λέγεται detail και που περιέχει το message και το id του υλικού που υπάρχει ήδη στη ΒΔ. Το frontend λοιπόν, ενημερώνεται με αυτό τον τρόπο για το ότι το υλικό υπάρχει και έπειτα χρησιμοποιεί το συγκεκριμένο id του υλικού κάνοντας εκ νέου μία άλλη API κλήση που αφορά στο update του description που θα παρουσιαστεί στην συνέχεια. Με αυτό τον τρόπο επιτυγχάνεται η επικοινωνία μεταξύ του frontend και του backend κάθε φορά.

Η συνάρτηση που διαχειρίζεται την εισαγωγή του υλικού στη ΒΔ είναι η create\_ingredient όπως φαίνεται παρακάτω

def create\_ingredient(db: Session, ingredient: IngredientCreate, user\_id: int):

user = db.query(User).filter(User.id == user\_id).first()

if not user:

raise HTTPException(status\_code=404, detail="User not found")

final\_foodex\_code = construct\_final\_foodex\_code(ingredient.ingredient\_base\_term, ingredient.ingredient\_with\_facets)

existing\_ingredient = db.query(Ingredient).filter(

Ingredient.final\_Foodex\_code == final\_foodex\_code,

Ingredient.place\_of\_origin == ingredient.place\_of\_origin

).first()

print(existing\_ingredient)

if existing\_ingredient:

raise HTTPException(

status\_code=400,

detail={"message": "Ingredient already exists with the same final\_Foodex\_code and place of origin", "ingredient\_id": existing\_ingredient.id}

)

ingredient\_with\_facets = [facet.dict() for facet in ingredient.ingredient\_with\_facets]

description=[desc.dict() for desc in ingredient.description] if ingredient.description else None

if "#" in final\_foodex\_code or "$" in final\_foodex\_code:

foodex\_code\_for\_recipes = "F26.A07XE"

else:

foodex\_code\_for\_recipes = "F04."+final\_foodex\_code

db\_ingredient = Ingredient(

ingredient\_name=ingredient.ingredient\_name,

ingredient\_base\_term=ingredient.ingredient\_base\_term,

ingredient\_base\_term\_name=ingredient.ingredient\_base\_term\_name,

ingredient\_with\_facets=ingredient\_with\_facets,

place\_of\_origin=ingredient.place\_of\_origin,

is\_POP=ingredient.is\_POP,

is\_PGE=ingredient.is\_PGE,

latin\_name=ingredient.latin\_name,

greek\_name=ingredient.greek\_name,

greek\_alter\_name=ingredient.greek\_alter\_name,

description=description,

owner\_id=user\_id,

final\_Foodex\_code=final\_foodex\_code,

foodex\_Code\_for\_recipes = foodex\_code\_for\_recipes

)

try:

db.add(db\_ingredient)

db.commit()

db.refresh(db\_ingredient)

logging.debug(f"Created new ingredient: {db\_ingredient}")

print(f"Created new ingredient: {db\_ingredient}")

print(db\_ingredient)

return db\_ingredient

except IntegrityError:

db.rollback()

raise HTTPException(status\_code=400, detail="Ingredient already exists")

η οποία καλείται μέσω της συνάρτησης του endpoint όπου γίνεται η κλήση με τη μέθοδο POST μέσω μιας άλλης συνάρτησης με το ίδιο όνομα create\_ingredient.

@app.post("/ingredients/", response\_model=schemas.IngredientOut, tags=["ingredients"])

def create\_ingredient(ingredient: schemas.IngredientCreate, user\_id: int, db: Session = Depends(get\_db)):

return services.**create\_ingredient**(db, ingredient, user\_id)

Αν ολοκληρωθεί επιτυχώς η εισαγωγή του υλικού στη ΒΔ τότε επιστρέφεται ως response το αντικείμενο IngredientOut.

Η συνάρτηση έχει ως παράμετρο εκτός του αντικειμένου ingredient και του db session, το user\_id του χρήστη που χρησιμοποιείται ως query parameter στο endpoint /ingredients/?user\_id. Το user\_id καθορίζεται στο frontend, on the fly, ανάλογα με τον χρήστη που είναι logged in στο σύστημα και που επιχειρεί να καταχωρήσει ένα υλικό (στην επόμενη ενότητα που αφορά στο frontend θα διαπιστωθεί το πως επιτυγχάνεται αυτό με την χρησιμοποίηση του component UserContext). Έτσι λοιπόν, εντός της συνάρτησης create\_ingredient γίνεται έλεγχος αρχικά με ένα query στη ΒΔ αν ο χρήστης που επιχειρεί να καταχωρήσει το υλικό, υπάρχει ή όχι. Στην συνέχεια, ορίζεται η μεταβλητή final\_foodex\_code που αρχικοποιείται με την τιμή του αποτελέσματος της βοηθητικής συνάρτησης construct\_final\_foodex\_code που χρησιμοποιείται στα services, η λειτουργία της οποίας είναι να φτιάξει το foodex code του προς καταχώρηση υλικού με το σωστό format.

def construct\_final\_foodex\_code(base\_term: str, facets: list) -> str:

facet\_codes = [facet.code for facet in facets]

joined\_codes = '$'.join(facet\_codes)

if len(facet\_codes) > 0 and facet\_codes[0] != '':

return f"{base\_term}#{joined\_codes}"

else:

return base\_term

Η συνάρτηση αυτή έχει ως παραμέτρους ένα string και μία λίστα από json αντικείμενα με keys το name και το code. Έπειτα, απομονώνει σε μία λίστα μόνο τα keys με το όνομα code του κάθε αντικειμένου και ενώνει τα codes αυτά μεταξύ τους βάζοντας ανάμεσά τους το σύμβολο “$”. Τέλος, η επιστρεφόμενη τιμή θα είναι ένα string ανάλογα την περίπτωση. Αν η λίστα είναι γεμάτη θα επιστρέψει το 1ο όρισμα συνδυαστικά με τα codes που έγιναν join με το σύμβολο “#” να παρεμβαίνει μεταξύ τους αλλιώς θα επιστραφεί μόνο το 1ο όρισμα.

Η final\_foodex\_code χρησιμοποιείται για τον καθορισμό της τιμής που θα πάρει και το calculated πεδίο foodex\_code\_for\_recipes του πίνακα των ingredients και επιπλέον για τον έλεγχο που γίνεται στη ΒΔ, για το αν υπάρχει ο συγκεκριμένος κωδικός ήδη στη βάση σε συνδυασμό με το ίδιο place\_of\_origin, σε άλλο υλικό.

Στην περίπτωση που το υλικό δεν υπάρχει, τότε το υλικό καταχωρείται επιτυχώς στη ΒΔ ενώ στην περίπτωση που το προς καταχώρηση υλικό έχει το ίδιο foodex code σε συνδυασμό με το ίδιο place\_of\_origin με ένα άλλο υλικό που είναι ήδη αποθηκευμένο στη ΒΔ, τότε η συνθήκη

if existing\_ingredient:

raise HTTPException(

status\_code=400,

detail={"message": "Ingredient already exists with the same final\_Foodex\_code and place of origin", "ingredient\_id": existing\_ingredient.id}

)

είναι αληθής με αποτέλεσμα να στέλνεται ως response ένα exception στον browser. Σε αυτό το σημείο, το frontend διαχειρίζεται το exception αυτό και χρησιμοποιώντας το id του υλικού που υπάρχει καλεί την παρακάτω API κλήση του FastaPI

@app.put("/ingredients/{ingredient\_id}/description/{description}", response\_model=schemas.IngredientOut, tags=["ingredients"])

def update\_ingredient\_description(ingredient\_id: int, description: str, db: Session = Depends(get\_db)):

return services.update\_ingredient\_description(db, ingredient\_id, description)

που αφορά στο update του description του εν λόγω υλικού. Με την σειρά του το endpoint καλεί την συνάρτηση στα services

def update\_ingredient\_description(db: Session, ingredient\_id: int, user\_input: str):

ingredient = db.query(Ingredient).get(ingredient\_id)

if ingredient:

logging.debug(f"Updating ingredient ID {ingredient\_id} with new description: {user\_input}")

print(f"Updating ingredient ID {ingredient\_id} with new description: {user\_input}")

if ingredient.description is None:

ingredient.description = []

logging.debug(f"Current description before update: {ingredient.description}")

print(f"Current description before update: {ingredient.description}")

updated\_description = ingredient.description + [{'user\_input': user\_input}]

try:

ingredient.description = updated\_description

db.commit()

db.refresh(ingredient)

logging.debug(f"Updated ingredient: {ingredient.description}")

print(f"Updated ingredient: {ingredient.description}")

return ingredient

except IntegrityError:

db.rollback()

raise HTTPException(status\_code=400, detail="Failed to update ingredient description")

else:

raise HTTPException(status\_code=404, detail="Ingredient not found")

στην οποία γίνεται τελικά η προσθήκη ενός νέου description από τον χρήστη που επιχείρησε αρχικά να καταχωρήσει ένα νέο υλικό. Όπως παρατηρείται, η συνάρτηση update\_ingredient\_description θα πάρει ως όρισμα το νέο description και το id του υλικού που υπάρχει στη βάση το οποίο θα το αναζητήσει με ένα query για να προχωρήσει στην συνέχεια με το update του attribute description που έχει οριστεί ως μία λίστα αντικειμένων τύπου json με key το user\_input και value το string που εισάγει ο χρήστης ως νέο description στο πεδίο της φόρμας του UI.

Αντίστοιχα με τη λειτουργία που επιτελούν οι προηγούμενες συναρτήσεις σε σχέση με την καταχώρηση ενός υλικού, οι συναρτήσεις create\_recipe, update\_recipe\_description και construct\_final\_foodex\_code\_with\_exisiting\_ingredients αφορούν στην καταχώρηση μιας νέας συνταγής στη ΒΔ και στην ανανέωση του description με την προσθήκη ενός επιμέρους σχολίου, στην περίπτωση που η προς καταχώρηση συνταγή υφίσταται ήδη στη βάση. Επιπλέον, Στην περίπτωση που η νέα συνταγή που καταχωρείται, περιέχει κάποιο υλικό ή επεξεργασμένο υλικό από τους πίνακες των ingredients και processed\_ingredients αντίστοιχα τότε εντός της συνάρτησης υλοποιείται η λογική που αφορά στην ενημέρωση των βοηθητικών πινάκων ingredient\_recipe και processed\_recipe\_ingredient που αποτελούν τις εκφράσεις των σχέσεων πολλά προς πολλά μεταξύ των. Έτσι , η συνάρτηση

def create\_recipe(db: Session, recipe: RecipeCreate, user\_id: int):

που καλείται από το endpoint

@app.post("/recipes/", response\_model=schemas.RecipeOut, tags=["recipes"])

def create\_recipe(recipe: schemas.RecipeCreate, user\_id: int, db: Session = Depends(get\_db)):

return services.create\_recipe(db, recipe, user\_id)

αρχικά ελέγχει αν στο πεδίο recipe\_with\_facets που είναι μία λίστα από αντικείμενα Facet εμπεριέχεται κάποιο ingredient που να ήδη καταχωρημένο στον πίνακα των ingredient,

# Check if any ingredient in recipe\_with\_facets already exists in ingredients table

existing\_ingredient\_list = []

for facet in recipe.recipe\_with\_facets:

existing\_ingredient = db.query(Ingredient).filter(

Ingredient.ingredient\_name == facet.name,

Ingredient.ingredient\_base\_term == facet.code

).first()

if existing\_ingredient:

existing\_ingredient\_list.append(existing\_ingredient)

final\_foodex\_code = construct\_final\_foodex\_code\_with\_existing\_ingredients(

recipe.recipe\_base\_term,

recipe.recipe\_with\_facets,

existing\_ingredient\_list

)

αν περιέχεται τότε με τη βοήθεια της συνάρτησης construct\_final\_foodex\_code\_with\_exisiting\_ingredients που χρησιμοποιεί τη μεταβλητή exisiting\_ingredient\_list ως όρισμα

def construct\_final\_foodex\_code\_with\_existing\_ingredients(base\_term: str, facets: list, existing\_ingredient: List[Ingredient]) -> str:

facet\_codes = [facet.code for facet in facets]

for item in existing\_ingredient:

if item.ingredient\_base\_term in facet\_codes:

index = facet\_codes.index(item.ingredient\_base\_term)

facet\_codes[index] = item.foodex\_Code\_for\_recipes

joined\_codes = '$'.join(facet\_codes) if facet\_codes else ''

if facet\_codes and facet\_codes[0] == '':

return base\_term

return f"{base\_term}#{joined\_codes}" if joined\_codes else base\_term

σχηματίζεται αυτόματα ο κωδικός του πεδίου foodex\_code. Στην περίπτωση που δεν χρησιμοποιείται κανένα υπάρχον υλικό τότε για τον σχηματισμό του κωδικού καλείται η συνάρτηση construct\_final\_foodex\_code που παρουσιάστηκε προηγουμένως. Στην συνέχεια η μεταβλητή final\_foodex\_code χρησιμοποιείται για να γίνει ένα query στη βάση ώστε να ελεγχθεί αν η προς καταχώρηση συνταγή υπάρχει στη ΒΔ. Στην περίπτωση που υπάρχει τότε διενεργείται μία νέα API κλήση στο endpoint

@app.put("/recipes/{recipe\_id}/description/{description}", tags=["recipes"])

def update\_recipe\_description\_endpoint(recipe\_id: int, description: str, db: Session = Depends(get\_db)):

return services.update\_recipe\_description(db, recipe\_id, description)

μέσω του οποίου καλείται η update\_recipe\_description

def update\_recipe\_description(db: Session, recipe\_id: int, user\_input: str):

recipe = db.query(Recipe).get(recipe\_id)

if recipe:

logging.debug(f"Updating recipe ID {recipe\_id} with new description: {user\_input}")

if recipe.description is None:

recipe.description = []

logging.debug(f"Current description before update: {recipe.description}")

updated\_description = recipe.description + [{'user\_input': user\_input}]

try:

recipe.description = updated\_description

db.commit()

db.refresh(recipe)

logging.debug(f"Updated recipe: {recipe.description}")

return recipe

except IntegrityError:

db.rollback()

raise HTTPException(status\_code=400, detail="Failed to update recipe description")

else:

raise HTTPException(status\_code=404, detail="Recipe not found")

στην οποία υλοποιείται η ανανέωση του πεδίου description του υπάρχοντος υλικού με την προσθήκη ενός νέου σχολίου στη λίστα με τα αντικείμενα Description.

Στην συνέχεια, ορίζεται ένα νέο αντικείμενο Recipe τα attributes του οποίου αρχικοποιούνται με τις τιμές όπως αυτές εστάλησαν στο request body κατά την API κλήση και τελικά το αντικείμενο καταχωρείται στη ΒΔ. Ακολουθούν τα updates των πινάκων ingredient\_recipe και processed\_recipe\_ingredient της βάσης

# Update the many-to-many relationship between recipes and ingredients

for facet in recipe\_with\_facets:

ingredient = db.query(Ingredient).filter(

Ingredient.ingredient\_name == facet['name'],

Ingredient.ingredient\_base\_term == facet['code']

).first()

if ingredient:

association = IngredientRecipe(

recipe\_id=db\_recipe.id,

ingredient\_id=ingredient.id,

amount=facet['amount']

)

db.add(association)

db.commit()

if processed\_ingredients\_info is not None:

# Update the many-to-many relationship between recipes and processed ingredients

for info in processed\_ingredients\_info:

processed\_ingredient = db.query(ProcessedIngredient).filter(

ProcessedIngredient.final\_Foodex\_code == info['final\_Foodex\_code']

).first()

if not processed\_ingredient:

logging.warning(f"ProcessedIngredient with final\_Foodex\_code {info['final\_Foodex\_code']} not found")

continue # Skip this processed ingredient and continue with the others

if processed\_ingredient:

association = ProcessedIngredientRecipe(

recipe\_id=db\_recipe.id,

processed\_ingredient\_id=processed\_ingredient.id,

# amount=info['amount']

)

db.add(association)

db.commit()

αφού αρχικά διενεργηθούν οι κατάλληλοι έλεγχοι, ώστε να εντοπιστεί αν η συνταγή χρησιμοποιεί ή όχι, υλικά ή επεξεργασμένα υλικά που ήδη βρίσκονται αποθηκευμένα στη βάση.

Ως προς τη λειτουργία των αναζητήσεων που εκτελούνται στην εφαρμογή, για τις αποθηκευμένες συνταγές, τα αποθηκευμένα υλικά και τα επεξεργασμένα υλικά στη ΒΔ, η λογική που έχει υλοποιηθεί παρουσιάζεται στην συνέχεια.

@app.get("/search\_ingredient/", tags=["search\_terms"])

async def analyze\_text(search\_item: str, db: Session = Depends(get\_db)):

# Αναζήτηση στη βάση δεδομένων

ingredients = services.search\_ingredients(db, search\_item)

if len(ingredients) == 0:

raise HTTPException(status\_code=404, detail="Ingredient not found")

return ingredients

Το endpoint στον παραπάνω κώδικα καλείται μέσω της συνάρτησης analyze\_text η οποία δέχεται ως query parameter το search\_item (τη λέξη που θέλει να βρει ο χρήστης). Η επιστρεφόμενη τιμή είναι μία λίστα που περιέχει όλα τα υλικά που εμπεριέχουν μέρος του προς αναζήτηση όρου. Η μεταβλητή ingredients αρχικοποιείται με την τιμή που επιστρέφει η βοηθητική συνάρτηση search\_ingredient κατά την κλήση της.

# Search ingredients function

def search\_ingredients(db: Session, search\_item: str):

normalized\_search\_item = strip\_accents\_and\_lowercase(search\_item)

ingredients = db.query(Ingredient).all()

normalized\_ingredients = [

ingredient for ingredient in ingredients

if normalized\_search\_item in strip\_accents\_and\_lowercase(ingredient.ingredient\_name)

or normalized\_search\_item in strip\_accents\_and\_lowercase(ingredient.greek\_name)

or normalized\_search\_item in strip\_accents\_and\_lowercase(ingredient.greek\_alter\_name)

or any(

normalized\_search\_item in strip\_accents\_and\_lowercase(desc['user\_input'])

for desc in ingredient.description

)

]

return normalized\_ingredient

Η συνάρτηση search\_ingredients κάνει χρήση μιας άλλης βοηθητικής συνάρτησης όπως φαίνεται παραπάνω, της strip\_accents\_and\_lowercase η οποία είναι υπεύθυνη για το normalization το κείμενο, αφαιρώντας τόνους (accents) και μετατρέποντας όλα τα γράμματα σε πεζά. Αυτό είναι χρήσιμο για την εξάλειψη των διαφορών μεταξύ κειμένων που μπορεί να έχουν τόνους ή διαφορετική χρήση κεφαλαίων και πεζών γραμμάτων. Έτσι, τόσο στη λέξη κλειδί search\_item όσο και στους όρους των πεδίων του πίνακα όπου γίνεται η αναζήτηση γίνονται normalized μέσω της συνάρτησης.

# Strip accents and convert to lowercase

def strip\_accents\_and\_lowercase(s: str) -> str:

return ''.join(c for c in unicodedata.normalize('NFD', s)

if unicodedata.category(c) != 'Mn').lower()

Τα αποτελέσματα του query αποθηκεύονται στη μνήμη, στη μεταβλητή ingredients και έπειτα με τη βοήθεια της Python με την εντολή in γίνεται ο έλεγχος κατά τη δημιουργία της λίστας που αρχικοποιεί τη μεταβλητή normalized\_ingrdients που τελικά επιστρέφεται στην κληθείσα συνάρτηση ως αποτέλεσμα.

Στην ενότητα αυτή παρουσιάστηκαν τα βήματα που ακολουθήθηκαν για τη δημιουργία του περιβάλλοντος του framework FastAPI καθώς και τα modules όπως αυτά υλοποιήθηκαν και προσαρμόστηκαν στη δομή του project από την πλευρά του backend. Στην επόμενη ενότητα αναλύονται αντιστοίχως τα κύρια σημεία που αφορούν στο setup ενός νέου React project το οποίο ενσωματώνεται στο τελικό προϊόν επικοινωνώντας με το backend, στη δομή και στη δημιουργία των διάφορων components που χρησιμοποιήθηκαν με σκοπό την ολοκλήρωση του εν λόγω έργου.

**Κεφ.8: Setup του project και η δομή του κώδικα (frontend)**

Τα βήματα για τη δημιουργία του frontend που υλοποιείται με την χρήση της βιβλιοθήκης React στο σύστημα είναι τα εξής:

1. Αρχικά στο σύστημα θα πρέπει να είναι εγκατεστημένο το runtime environment Node.js που επιτρέπει την εκτέλεση κώδικα javascript καθώς και το npm δηλαδή του εργαλείου με τη βοήθεια του οποίου γίνεται η διαχείριση των πακέτων και των dependencies. Οι εκδόσεις των Node.js και npm στο παρόν σύστημα που χρησιμοποιήθηκαν είναι η v20.12.2 και 10.5.0 αντίστοιχα.

2. Στην συνέχεια κάτω από το ίδιο directory δηλαδή στο path C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project που βρίσκεται και το backend της εφαρμογής, με την χρήση της εντολής

C:\Users\tziac\Workspace\recipe-treasure-foodex-fullstack-project> npx create-react-app frontend

γίνεται η εγκατάσταση του React application εντός του directory με όνομα frontend μαζί με όλα τα dependencies που χρειάζεται η νέα αυτή εφαρμογή. Η δομή του έργου σε αυτό το σημείο, μετά την επιτυχή εκτέλεση της παραπάνω εντολής, απεικονίζεται παρακάτω:

A screenshot of a computer

Description automatically generated

Στην παραπάνω εικόνα παρατηρείται ότι έχουν εγκατασταθεί όλα τα απαραίτητα αρχεία και τα directories που είναι χρήσιμα για την ανάπτυξη της εφαρμογής στην συνέχεια. Στο directory src θα φιλοξενηθεί ολόκληρη η λογική που εφαρμόζεται μέσω της κατασκευής διάφορων components που θα παρουσιαστούν στην συνέχεια της ενότητας. Για τον λόγο αυτό, δημιουργείται ένα νέο directory με όνομα components και ένα directory με όνομα data, εντός του directory src, όπως φαίνεται στην επόμενη εικόνα. Να σημειωθεί ότι δεν χρειάζεται να γίνει κάποια επιπρόσθετη παραμετροποίηση στα αρχεία του περιβάλλοντος.

A screenshot of a computer program

Description automatically generated

Με αυτά τα απλά βήματα ολοκληρώνεται το setup του περιβάλλοντος του frontend. Στην συνέχεια παρουσιάζονται τα βασικότερα components της εφαρμογής, το routing των pages καθώς και η διαχείριση των δεδομένων που αντλήθηκαν από την EFSA, επεξεργάστηκαν και τελικά χρησιμοποιήθηκαν στην εφαρμογή.

**Κεφ.8.1: Κύρια Components και προεπεξεργασία δεδομένων**

Πριν από την εξήγηση των components που κατασκευάστηκαν για την εφαρμογή κρίνεται σκόπιμο να γίνει αναφορά στην προεπεξεργασία των δεδομένων που χρησιμοποιήθηκαν. Πηγή των δεδομένων αποτέλεσε το xml αρχείο MTX\_FULL\_12\_0.xml το οποίο αντλήθηκε από την EFSA, που βρίσκεται αποθηκευμένο στο directory με όνομα handle-json-files.

Για τη διαχείριση λοιπόν των δεδομένων δημιουργήθηκε ένα directory εντός του project, το οποίο περιλάμβανε αρχικά εκτός του αρχείου MTX\_FULL\_12\_0.xml, τα αρχεία convertor.py και json\_handling.py, όπως φαίνεται παρακάτω.

A screenshot of a computer

Description automatically generated

Με την εκτέλεση του convertor.py

import xmltodict

import json

# # Read the XML file

with open('handle-json-files\\MTX\_FULL\_12\_0.xml', 'r', encoding='utf-8') as xml\_file:

xml\_data = xml\_file.read()

# # Convert XML to JSON

json\_data = json.dumps(xmltodict.parse(xml\_data), indent=4)

# # Write the JSON data to a new file

with open('handle-json-files\\converted.json', 'w') as json\_file:

json\_file.write(json\_data)

with open("handle-json-files\\converted.json", "r") as file:

data = json.load(file)

# Print the list

print("List of items:")

print(data['message']['catalogue']['catalogueTerms']['term'][0])

γίνεται η μετατροπή του xml αρχείου σε json με την ονομασία converted.json το οποίο αυτόματα αποθηκεύεται και αυτό στο ίδιο directory όπου εκτελέστηκε το παραπάνω script.

Το format του αρχείου converted.json φαίνεται στην επόμενη εικόνα.

A screen shot of a computer

Description automatically generated

Το παραπάνω αρχείο τροποποιήθηκε εκ νέου manually, επιχειρώντας να κρατηθεί μόνο η απαραίτητη πληροφορία που αφορά στο key catalogueTerms. Ουσιαστικά κρατήθηκε η λίστα (το value του key term) κάθε στοιχείο της οποίας είναι ένα json αντικείμενο.

A screenshot of a computer program

Description automatically generated

Με αυτό τον τρόπο δημιουργήθηκε το νέο αρχείο με όνομα converted2.json το οποίο περιέχει την παραπάνω λίστα, με όλα τα json αντικείμενα μέσα στα οποία υπάρχουν σημαντικές πληροφορίες όπως το termCode, το termExtendedName κλπ.

Στην συνέχεια το script, json\_handling.py χρησιμοποιεί το αρχείο converted2.json για την παραγωγή της τελικής μορφής των δεδομένων σε json format που αποθηκεύονται στο αρχείο final\_extracted\_data.json το οποίο έχει πλέον την παρακάτω μορφή.

A screenshot of a computer program

Description automatically generated

Το παραπάνω αρχείο είναι λοιπόν το output μετά την εκτέλεση του παρακάτω script.

import json

# Read JSON data from file

with open('handle-json-files\\converted2.json', 'r') as file:

data = json.load(file)

# Define a list to store the extracted data

extracted\_data = []

# Define the list of specific hierarchy codes

specific\_hierarchy\_codes = [

"part", "source", "racsource", "process", "ingred", "medium", "sweet",

"fort", "qual", "cookext", "gen", "prod", "packformat", "packmat",

"state", "fat", "alcohol", "dough", "partcon", "place", "targcon",

"use", "riskingred", "fpurpose", "replev", "animage", "gender", "legis"

]

# Iterate over each term object

for term in data:

# Extract required attributes

extracted\_term = {

"termCode": term['termDesc']['termCode'],

"termExtendedName": term['termDesc']['termExtendedName'],

}

# Check if termScopeNote exists before accessing it

if 'termScopeNote' in term['termDesc']:

extracted\_term["termScopeNote"] = term['termDesc']['termScopeNote']

parent\_code\_found = False

term\_type\_found = False

# Find parentCode and other hierarchy codes from hierarchyAssignments

hierarchy\_assignments = term['hierarchyAssignments']['hierarchyAssignment']

if isinstance(hierarchy\_assignments, list):

for assignment in hierarchy\_assignments:

if assignment['hierarchyCode'] == "MTX":

extracted\_term["parentCode"] = assignment['parentCode']

parent\_code\_found = True

elif assignment['hierarchyCode'] == "report":

extracted\_term["reportable"] = assignment['reportable']

elif assignment['hierarchyCode'] in specific\_hierarchy\_codes:

extracted\_term["facet"] = assignment['hierarchyCode']

else: # If hierarchyAssignment is not a list but a dictionary

if hierarchy\_assignments['hierarchyCode'] == "MTX":

extracted\_term["parentCode"] = hierarchy\_assignments['parentCode']

parent\_code\_found = True

elif hierarchy\_assignments['hierarchyCode'] == "report":

extracted\_term["reportable"] = hierarchy\_assignments['reportable']

elif hierarchy\_assignments['hierarchyCode'] in specific\_hierarchy\_codes:

extracted\_term["facet"] = hierarchy\_assignments['hierarchyCode']

# Find termType from implicitAttributes where attributeCode is "detailLevel"

implicit\_attributes = term['implicitAttributes']['implicitAttribute']

if isinstance(implicit\_attributes, list):

for attribute in implicit\_attributes:

if attribute['attributeCode'] == "detailLevel":

extracted\_term["termType"] = attribute['attributeValues']['attributeValue']

term\_type\_found = True

else: # If implicitAttribute is not a list but a dictionary

if implicit\_attributes['attributeCode'] == "detailLevel":

extracted\_term["termType"] = implicit\_attributes['attributeValues']['attributeValue']

term\_type\_found = True

# Check if both parentCode and termType are found

if parent\_code\_found and term\_type\_found:

# Add the extracted term to the list

extracted\_data.append(extracted\_term)

# Write the extracted data to a new JSON file

with open('handle-json-files\\extracted\_data.json', 'w') as outfile:

json.dump(extracted\_data, outfile, indent=4)

# Print the number of objects in the extracted\_data list

print(f"Extraction completed. Number of objects extracted: {len(extracted\_data)}")

import json

# Read the extracted\_data from the JSON file

with open('handle-json-files\\extracted\_data.json', 'r') as file:

extracted\_data = json.load(file)

# Define the mapping for facet replacement

facet\_mapping = {

"part": "F02",

"source": "F01",

"racsource": "F27",

"process": "F28",

"ingred": "F04",

"medium": "F06",

"sweet": "F08",

"fort": "F09",

"qual": "F10",

"cookext": "F17",

"gen": "F26",

"prod": "F21",

"packformat": "F18",

"packmat": "F19",

"state": "F03",

"fat": "F07",

"alcohol": "F11",

"dough": "F12",

"partcon": "F20",

"place": "F22",

"targcon": "F23",

"use": "F24",

"riskingred": "F25",

"fpurpose": "F29",

"replev": "F30",

"animage": "F31",

"gender": "F32",

"legis": "F33"

}

# Replace the facet values based on the mapping

for term in extracted\_data:

if 'facet' in term:

facet\_value = term['facet']

if facet\_value in facet\_mapping:

term['facet'] = facet\_mapping[facet\_value]

# Write the updated data to a new JSON file

with open('handle-json-files\\final\_extracted\_data.json', 'w') as outfile:

json.dump(extracted\_data, outfile, indent=4)

# Print the number of objects in the final\_extracted\_data list

print(f"Facet replacement completed. Number of objects in the final data: {len(extracted\_data)}")

Το παραπάνω script, εκτελεί δύο κύρια βήματα για την επεξεργασία και μετασχηματισμό των δεδομένων από το αρχείο converted2.json.

1. Ανάγνωση και Εξαγωγή Δεδομένων από το JSON (Πρώτο Μέρος)

* **Ανάγνωση του αρχείου JSON**: Το πρόγραμμα ξεκινάει με την ανάγνωση του converted2.json που περιέχει μια λίστα από αντικείμενα.
* **Αρχικοποίηση λίστας για αποθήκευση δεδομένων**: Δημιουργείται μια κενή λίστα (extracted\_data) για να αποθηκευτούν τα δεδομένα που θα γίνουν εξαγωγή.
* **Ορισμός συγκεκριμένων hierarchy codes**: Δημιουργείται μια λίστα με εκείνα τα "hierarchy codes" που θα χρειαστεί να έχουν ως χαρακτηριστικό τα δεδομένα που θα επεξεργαστούν και θα αποθηκευτούν.
* **Εξαγωγή δεδομένων από κάθε αντικείμενο**: Για κάθε αντικείμενο στο JSON, το script εξάγει συγκεκριμένα πεδία:
  + **termCode και termExtendedName**: Από το πεδίο termDesc.
  + **termScopeNote**: Μόνο αν υπάρχει.
  + **parentCode και facet**: Ανάλογα με τις τιμές των hierarchyAssignment και implicitAttributes.
* **Έλεγχος και αποθήκευση**: Αν βρεθούν τα πεδία parentCode και termType, το αντικείμενο αποθηκεύεται στη λίστα extracted\_data.
* **Αποθήκευση εξαγόμενων δεδομένων**: Τέλος, τα εξαγόμενα δεδομένα αποθηκεύονται σε ένα νέο αρχείο JSON (extracted\_data.json).

2. Αντικατάσταση Facet Values και Δημιουργία Τελικού JSON (Δεύτερο Μέρος)

* **Ανάγνωση του εξαγόμενου JSON**: Το δεύτερο μέρος του κώδικα ξεκινάει με την ανάγνωση του αρχείου extracted\_data.json.
* **Ορισμός ενός χάρτη αντιστοίχισης (mapping)**: Δημιουργείται ένας πίνακας facet\_mapping που αντιστοιχίζει τα αρχικά facet values σε νέα values της μορφής F01, F02 κ.ο.κ.
* **Αντικατάσταση τιμών**: Για κάθε αντικείμενο στο extracted\_data, αν υπάρχει το πεδίο facet, γίνεται αντικατάσταση της τιμής του με βάση τον πίνακα facet\_mapping.
* **Αποθήκευση του τελικού JSON**: Τα ενημερωμένα δεδομένα αποθηκεύονται σε ένα νέο αρχείο JSON (final\_extracted\_data.json).

Το τελικό αρχείο γίνεται copy paste στο directory data κάτω από το directory src του React application με σκοπό να χρησιμοποιηθεί στην υλοποίηση ενός σημαντικού component με όνομα TreeComponent.js που θα εξεταστεί στην συνέχεια αυτής της ενότητας.

Η τελική δομή του directory handle-json-files μετά τις εκτελέσεις των scripts και την manually επεξεργασία των αρχείων έχει την παρακάτω μορφή.

A screenshot of a computer

Description automatically generated

Τα components που κατασκευάστηκαν για την επίτευξη της θεμιτής λειτουργικότητας της εφαρμογής, φαίνονται συγκεντρωτικά στις εικόνες που ακολουθούν.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

Όλα τα components βρίσκονται αποθηκευμένα στο directory με όνομα components κάτω από το directory src του React application.

Τα components αυτά καταναλώνονται από το App component που αποτελεί το root component της εφαρμογής και λειτουργεί ως το κύριο container. Μέσα στο App, όλες οι σελίδες (components) και η πλοήγηση της εφαρμογής είναι ορισμένες μέσω των Routes και Route. Πιο συγκεκριμένα, στον παρακάτω κώδικα

import React from 'react';

import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

import { UserProvider } from './components/UserContext';

import Login from './components/Login';

import Register from './components/Register';

import HomePage from './components/HomePage';

import AddIngredient from './components/AddIngredient';

import AddRecipe from './components/AddRecipe';

import AddProcessedIngredient from './components/AddProcessedIngredient';

import ShowMyRecipes from './components/ShowMyRecipes';

import DeleteUser from './components/DeleteUser';

import DeleteRecipe from './components/DeleteRecipe';

import DeleteIngredient from './components/DeleteIngredient';

import DeleteProcessedIngredient from './components/DeleteProcessedIngredient';

import UpdateIngredient from './components/UpdateIngredient';

import RecipesByRegion from './components/RecipesByRegion';

import ShowMyIngredients from './components/ShowMyIngredients';

import ShowMyProcessedIngredients from './components/ShowMyProcessedIngredients';

import RecipesByName from './components/RecipesByName';

import ShowUsers from './components/ShowUsers';

const App = () => {

return (

<UserProvider>

<Router>

<Routes>

<Route path="/" element={<Login />} />

<Route path="/login" element={<Login />} />

<Route path="/register" element={<Register />} />

<Route path="/home" element={<HomePage />} />

<Route path="/add-ingredient" element={<AddIngredient />} />

<Route path="/add-recipe" element={<AddRecipe />} />

<Route path="/add-processed-ingredient" element={<AddProcessedIngredient />} />

<Route path="/my-recipes" element={<ShowMyRecipes />} />

<Route path="/delete-user" element={<DeleteUser />} />

<Route path="/delete-ingredient" element={<DeleteIngredient />} />

<Route path="/delete-recipe" element={<DeleteRecipe />} />

<Route path="/delete-processed-ingredient" element={<DeleteProcessedIngredient />} />

<Route path="/update-ingredient" element={<UpdateIngredient />} />

<Route path="/recipes-per-region" element={<RecipesByRegion />} />

<Route path="/my-ingredients" element={<ShowMyIngredients />} />

<Route path="/my-processed-ingredients" element={<ShowMyProcessedIngredients />} />

<Route path="/recipes-per-name" element={<RecipesByName />} />

<Route path="/users/" element={<ShowUsers />} />

</Routes>

</Router>

</UserProvider>

);

};

export default App;

το component **UserProvider** είναι ένα context provider που ουσιαστικά τυλίγει όλη την εφαρμογή. Παρέχει δεδομένα και λειτουργικότητα (state management, methods, κλπ.) σχετικά με τον χρήστη σε όλα τα components που βρίσκονται "μέσα" στο UserProvider. Αυτό γίνεται μέσω του React Context API, το οποίο επιτρέπει τη διαχείριση του state της εφαρμογής και το μοίρασμα δεδομένων μεταξύ διαφορετικών components χωρίς να χρειάζεται η προώθηση props μέσω πολλαπλών επιπέδων.

Το Router είναι το component που εισάγεται από τη βιβλιοθήκη react-router-dom. Συγκεκριμένα, χρησιμοποιείται το BrowserRouter με alias ως Router. Είναι υπεύθυνο για τη διαχείριση του routing της εφαρμογής , δηλαδή τον τρόπο με τον οποίο γίνεται η πλοήγηση μεταξύ των διαφορετικών σελίδων/components της εφαρμογής.

Το Routes είναι επίσης ένα component από τη βιβλιοθήκη react-router-dom. Περιέχει τα Route components και είναι υπεύθυνο για την απόδοση του κατάλληλου Route component, με βάση την τρέχουσα διαδρομή (path) στο URL. Αν το URL είναι /home, το Routes θα βρει το Route που έχει path "/home" και θα αποδώσει το HomePage component.

Το Route είναι το component που καθορίζει ποιο component θα εμφανιστεί για μια συγκεκριμένη διαδρομή (URL path). Κάθε Route έχει ένα path prop που αντιστοιχεί σε μια διαδρομή του URL και ένα element prop που καθορίζει ποιο component θα αποδοθεί όταν η διαδρομή αυτή είναι ενεργή. Για τη διαδρομή "/home", το Route αποδίδει το HomePage component.

Όλα αυτά τα components συνεργάζονται για να διαχειριστούν την πλοήγηση, τη διαχείριση κατάστασης (state management), και την απόδοση των σωστών components στην εφαρμογή.

Στο αρχείο index.js βρίσκεται ο κώδικας που αποτελεί το σημείο εισόδου για την εφαρμογή React.

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import 'bootstrap/dist/css/bootstrap.min.css';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

Αυτός ο κώδικας παίρνει το κύριο component της εφαρμογής (App), το οποίο περιέχει όλη τη λογική και τη δομή της εφαρμογής, και το αποδίδει μέσα στο HTML element, ένα div με id="root" στο αρχείο index.html που βρίσκεται μέσα στο directory public της React εφαρμογής. Αυτό σημαίνει ότι όλο το περιεχόμενο της εφαρμογής θα εμφανιστεί μέσα σε αυτό το στοιχείο στη σελίδα. Τέλος, το React.StrictMode χρησιμοποιείται για να βοηθήσει στον εντοπισμό προβλημάτων κατά την ανάπτυξη της εφαρμογής.

Στην συνέχεια επιλέχτηκαν κάποια ακόμα βασικά components της εφαρμογής, τα οποία εξετάζονται σε ένα μεγαλύτερο βάθος, προκειμένου να υπογραμμιστούν ο τρόπος που αυτά επικοινωνούν μεταξύ τους αλλά και η λειτουργικότητα που επιτελούν στην εφαρμογή.

Ο παρακάτω κώδικας αποτελεί το component TreeComponent το οποίο υλοποιήθηκε με τη βοήθεια της βιβλιοθήκης react-treebeard η οποία χρησιμοποιείται για την εμφάνιση και τη διαχείριση της δεντρικής δομής των δεδομένων. Τα δεδομένα αφορούν στο json αρχείο που κατασκευάστηκε όπως αναφέρθηκε προηγουμένως στην αρχή της ενότητας.

import React, { useState } from 'react';

import { Treebeard } from 'react-treebeard';

const buildTree = (data) => {

const map = {};

data.forEach(node => {

map[node.termCode] = { ...node, name: `${node.termExtendedName} (${node.termCode})`, children: [] };

});

data.forEach(node => {

if (node.parentCode && map[node.parentCode]) {

map[node.parentCode].children.push(map[node.termCode]);

}

});

const tree = [];

data.forEach(node => {

if (!node.parentCode || !map[node.parentCode]) {

tree.push(map[node.termCode]);

}

});

return { name: "Foodex catalogues", termCode: "root", children: tree };

};

const collapseTree = (node) => {

if (node.children) {

node.children.forEach(child => collapseTree(child));

node.toggled = false;

}

};

const TreeComponent = ({ data, onSubmitData, onSubmitFacet }) => {

const [cursor, setCursor] = useState(null);

const [treeData, setTreeData] = useState(buildTree(data));

const onToggle = (node, toggled) => {

if (cursor && cursor.active) {

cursor.active = false;

}

node.active = true;

if (node.children) {

node.toggled = toggled;

}

setCursor(node);

setTreeData({ ...treeData });

};

const handleSelectBaseTerm = () => {

if (cursor.termType !== "M" && cursor.termType !== "C" && cursor.termType !== "E" && cursor.termType !== "F") {

alert("This node cannot be selected.");

return;

}

if (cursor) {

onSubmitData([{

name: cursor.termExtendedName,

code: cursor.termCode

}]);

}

collapseTree(treeData);

setTreeData({ ...treeData });

setCursor(null);

};

const handleSelectFacet = () => {

if (cursor.termType !== "M" && cursor.termType !== "C" && cursor.termType !== "E" && cursor.termType !== "F") {

alert("This node cannot be selected.");

return;

}

if (cursor && cursor.facet) {

const facetCode = `${cursor.facet}.${cursor.termCode}`;

onSubmitFacet({

name: cursor.termExtendedName,

code: facetCode

});

} else {

alert("This node does not have a facet.");

}

collapseTree(treeData);

setTreeData({ ...treeData });

setCursor(null);

};

const customStyles = {

tree: {

base: {

listStyle: 'none',

backgroundColor: '#333',

margin: 0,

padding: 0,

color: '#9DA5AB',

fontFamily: 'Lato, sans-serif',

fontSize: '12px'

},

node: {

base: {

position: 'relative'

},

link: {

cursor: 'pointer',

position: 'relative',

padding: '0 5px',

display: 'block',

color: 'white'

},

activeLink: {

background: '#31363F',

color: 'white'

},

toggle: {

base: {

position: 'relative',

display: 'inline-block',

verticalAlign: 'middle',

marginLeft: '-5px',

height: '10px',

width: '10px'

},

wrapper: {

position: 'absolute',

top: '50%',

left: '50%',

margin: '-7px 0 0 -7px',

height: '14px'

},

height: 10,

width: 10,

arrow: {

fill: '#9DA5AB',

strokeWidth: 0

}

},

header: {

base: {

display: 'inline-block',

verticalAlign: 'middle',

color: 'white'

},

connector: {

width: '2px',

height: '12px',

borderLeft: 'solid 2px #9DA5AB',

borderBottom: 'solid 2px #9DA5AB',

position: 'absolute',

top: '0px',

left: '-21px'

},

title: {

lineHeight: '12px',

verticalAlign: 'middle'

}

},

subtree: {

listStyle: 'none',

paddingLeft: '19px'

},

loading: {

color: '#E2C089'

},

selectable: {

backgroundColor: '#dc143c',

borderRadius: '3px',

padding: '2px'

}

}

}

};

return (

<div>

<Treebeard

data={treeData}

onToggle={onToggle}

style={customStyles}

/>

{cursor && cursor.termCode !== "root" && cursor.termCode !== "all\_lists" && (

<div style={{ fontSize: '12px', padding: '10px', backgroundColor: '#f7f7f7', borderRadius: '5px', marginTop: '10px' }}>

<h3 style={{ fontSize: '14px', marginBottom: '5px' }}>Selected Node</h3>

<p><strong>Term Code:</strong> {cursor.termCode}</p>

<p><strong>Extended Name:</strong> {cursor.termExtendedName}</p>

<p><strong>Scope Note:</strong> {cursor.termScopeNote}</p>

{cursor.facet && <p><strong>Facet:</strong> {cursor.facet}</p>}

<button onClick={handleSelectBaseTerm} style={{ fontSize: '15px', padding: '3px 8px', marginLeft: '10px', marginRight: '10px', width: '40%'}}>Select Base Term</button>

<button onClick={handleSelectFacet} style={{ fontSize: '15px', padding: '3px 8px', marginLeft: '10px', marginRight: '10px', width: '40%'}}>Select Facet</button>

</div>

)}

</div>

);

};

export default TreeComponent;

Το buildTree function δημιουργεί μια δεντρική δομή από τα δεδομένα που περνιούνται ως όρισμα. Χρησιμοποιεί τη μεταβλητή map για να αποθηκεύσει τους κόμβους (nodes) και έπειτα συνδέει κάθε κόμβο με τον γονέα του μέσω της parentCode μιας ιδιότητας του αντικειμένου node.

Από την άλλη πλευρά η collapseTree συνάρτηση κλείνει (collapse) όλους τους κόμβους του δέντρου, απενεργοποιώντας την toggled ιδιότητα για κάθε κόμβο που έχει παιδιά.

Οι παραπάνω συναρτήσεις καλούνται όπως φαίνεται μέσα στο βασικό component TreeComponent το οποίο χρησιμοποιεί δύο useState hooks, το cursor για να κρατά τον επιλεγμένο κόμβο και το treeData για να αποθηκεύει τη δεντρική δομή των δεδομένων. Εντός του component η συνάρτηση onToggle διαχειρίζεται το expand και το collapse των κόμβων του δέντρου καθώς ο χρήστης κάνει navigate σε αυτό. Η συνάρτηση handleSelectBaseTerm εκτελείται όταν επιλέγεται ένας βασικός όρος από το δέντρο μέσω του κουμπιού Select Base Term και επικυρώνει τον σωστό τύπο του κόμβου προτού καλέσει την onSubmitData. Αν το attribute termType του αντικειμένου είναι ο θεμιτός τότε διενεργείται το submit των δεδομένων name και code του αντικειμένου, επιτυχώς, διαφορετικά ενημερώνει τον χρήστη ότι ο συγκεκριμένος κόμβος δεν μπορεί να επιλεγεί και να γίνει συνεπώς submit. Η συνάρτηση handleSelectFacet λειτουργεί παρόμοια με την προηγούμενη. Το component μπορεί να καταναλωθεί από ένα άλλο component μέσω των props, data (αρχείο με δεδομένα που ακολουθούν ένα ιεραρχικό fromat - final\_extracted\_data.json) και των callback functions onSubmitData και onSubmitFacet ενημερώνοντας το state κάποιας μεταβλητής με τις τιμές που γίνονται submit από το δέντρο.

Στη μεταβλητή customStyles αποθηκεύεται ένα JavaScript αντικείμενο που προσδίδει ένα συγκεκριμένο στυλ στο TreeComponent. Αυτό το στυλ καθορίζει την εμφάνιση των διαφόρων τμημάτων του δέντρου, όπως η λίστα, οι κόμβοι, οι σύνδεσμοι, οι τίτλοι, και τα βελάκια που δείχνουν την κατάσταση (ανοιχτό/κλειστό) ενός υποδέντρου.

Σχετικά με το rendering του δέντρου που επιτυγχάνεται μέσω του statement return όταν το component καταναλωθεί από ένα άλλο component, τότε εμφανίζεται στην οθόνη πρώτον το δέντρο που κατασκευάστηκε με τα δεδομένα και το ορισμένο στυλ που του αποδόθηκε και δεύτερον ένα element div κατά περίπτωση. Αν υπάρχει επιλεγμένος κόμβος και το attribute termCode του κόμβου δεν είναι "root" ή "all\_lists", τότε στο περιεχόμενο του div παρέχονται πληροφορίες για τον επιλεγμένο κόμβο και εμφανίζονται επιπλέον δύο κουμπιά για τη λήψη ενεργειών σχετικά με αυτόν τον κόμβο.

Στο component UserContext που ακολουθεί, παρουσιάζεται μία σημαντική τεχνική που χρησιμοποιείται στην React για τη διαχείριση και την επικοινωνία δεδομένων μεταξύ components χωρίς να απαιτείται η χρήση props. Με την εφαρμογή αυτής της τεχνικής διευκολύνεται η διαχείριση των κοινών δεδομένων μιας εφαρμογής.

import React, { createContext, useState, useEffect } from 'react';

export const UserContext = createContext();

export const UserProvider = ({ children }) => {

const [user, setUser] = useState(() => {

const savedUser = localStorage.getItem('user');

return savedUser ? JSON.parse(savedUser) : null;

});

useEffect(() => {

if (user) {

localStorage.setItem('user', JSON.stringify(user));

} else {

localStorage.removeItem('user');

}

}, [user]);

return (

<UserContext.Provider value={{ user, setUser }}>

{children}

</UserContext.Provider>

);

};

Ο παραπάνω κώδικας λοιπόν δημιουργεί ένα React Context μέσω της συνάρτησης createContext που χρησιμοποιείται για τη διαχείριση της κατάστασης του user στην εφαρμογή. Το context object περιέχει δύο βασικές ιδιότητες:

1. Provider: Είναι ένα component που χρησιμοποιείται για την παροχή του context σε όλα τα nested components. Ο Provider δέχεται μια value prop, η οποία καθορίζει τα δεδομένα που είναι διαθέσιμα σε όλα τα components που καταναλώνουν το context. Παράδειγμα της χρήσης του Provider αναφέρθηκε στην περιγραφή του root component App προηγουμένως.
2. Consumer: Είναι ένα component που χρησιμοποιείται για την κατανάλωση του context. Μέσω του Consumer, τα components μπορούν να έχουν πρόσβαση στην τρέχουσα τιμή του context. Ωστόσο, με τη χρήση του hook useContext, η χρήση του Consumer είναι λιγότερο συνήθης. Παρουσιάζεται παράδειγμα της χρήσης του hook useContext σε επόμενα components πιο κάτω.

Το UserContext είναι λοιπόν το context που θα χρησιμοποιηθεί για να μοιράζει δεδομένα γύρω από την εφαρμογή σχετικά με το state του user. Ενώ το UserProvider είναι ένα component που παρέχει το UserContext σε όλα τα παιδιά - children components που βρίσκονται μέσα σε αυτό. Το hook useState χρησιμοποιείται για την διαχείριση της κατάστασης του χρήστη (user), αρχικοποιώντας την από το localStorage αν υπάρχει ενώ το useEffect hook παρακολουθεί τις αλλαγές στο state του χρήστη και ενημερώνει το localStorage αναλόγως.

Το component Login για παράδειγμα,

import React, { useState, useContext } from 'react';

import axios from 'axios';

import { useNavigate } from 'react-router-dom';

import { UserContext } from './UserContext';

import styles from './AuthForm.module.css';

const Login = () => {

const [username, setUsername] = useState('');

const [password, setPassword] = useState('');

const [errorMessage, setErrorMessage] = useState('');

const navigate = useNavigate();

const { setUser } = useContext(UserContext);

const handleSubmit = async (e) => {

e.preventDefault();

try {

const response = await axios.post('http://localhost:8000/login', null, {

params: { username, password },

});

const { user\_id, is\_admin } = response.data;

setUser({ username, id: user\_id, isAdmin: is\_admin });

navigate('/home');

} catch (error) {

if (error.response && error.response.status === 401) {

setErrorMessage('Invalid username or password. Please try again. Or register');

} else {

setErrorMessage('An error occurred during login.');

}

}

};

return (

<div className={styles.container}>

<div className={styles.form}>

<h2>Login</h2>

<form onSubmit={handleSubmit}>

<div className={styles.inputGroup}>

<label>Username:</label>

<input

type="text"

value={username}

onChange={(e) => setUsername(e.target.value)}

/>

</div>

<div className={styles.inputGroup}>

<label>Password:</label>

<input

type="password"

value={password}

onChange={(e) => setPassword(e.target.value)}

/>

</div>

<button type="submit">Login</button>

</form>

{errorMessage && <p className={styles.errorMessage}>{errorMessage}</p>}

<button onClick={() => navigate('/register')} className={styles.registerButton}>

Go to Register

</button>

<h10>If you do not have an account, please register</h10>

</div>

</div>

);

};

export default Login;

κάνει χρήση του component UserContext μέσω του hook useContext για να αποκτήσει πρόσβαση σε αυτό. Η συνάρτηση setUser που παρέχεται από το context χρησιμοποιείται για να ενημερώσει την κατάσταση του χρήστη (κατά την είσοδό του). Έτσι τα νέα δεδομένα του χρήστη (το username, το id και το is\_admin πιο συγκεκριμένα) μπορούν και διαμοιράζονται πλέον και στα υπόλοιπα components επιτρέποντας με αυτό τον τρόπο και την χρήση των υπόλοιπων σελίδων σε αυτά τα δεδομένα αν αυτά είναι valid. Σε αυτή την περίπτωση όπως φαίνεται στον παραπάνω κώδικα γίνεται navigate στην σελίδα home της εφαρμογής ακριβώς επειδή τα δεδομένα είναι έγκυρα.

Για τη δημιουργία του component Login, χρησιμοποιείται το hook useState το οποίο ορίζει τρία states για τη διαχείριση του username, του password και του errorMessage. Επιπλέον, χρησιμοποιεί τα hooks useNavigate παρέχοντας τη δυνατότητα στον χρήστη να ανακατευθυνθεί σε άλλες σελίδες της εφαρμογής και useContext(UserContext) για την ενημέρωση του state του user που αναλύθηκε προηγουμένως.

Η συνάρτηση handleSubmit καλείται όταν ο χρήστης υποβάλει τη φόρμα. Κατά την υποβολή - submit της φόρμας, πραγματοποιείται η API κλήση στον server με παραμέτρους τα credentials του χρήστη. Αν το response είναι OK τότε οι τιμές user\_id και is\_admin που επιστρέφονται, χρησιμοποιούνται στην συνάρτηση setUser ενημερώνοντας με αυτό τον τρόπο το context, δηλαδή το state του user που μπήκε στο σύστημα ενώ παράλληλα ο χρήστης ανακατευθύνεται στην σελίδα /home. Αν υπάρξει σφάλμα, δηλαδή δεν βρεθεί στο backend ο συγκεκριμένος χρήστης γίνεται ενημέρωση του state errorMessage, με ένα μήνυμα λάθους το οποίο ειδοποιεί ανάλογα τον χρήστη της εφαρμογής για μη εξουσιοδοτημένο χρήστη.

Σχετικά με το rendering, την αναπαράσταση δηλαδή της φόρμας που βλέπει στην οθόνη του ο χρήστης, η φόρμα login περιλαμβάνει πεδία για το όνομα χρήστη και τον κωδικό πρόσβασης. Όταν ο χρήστης υποβάλλει τη φόρμα, καλείται η συνάρτηση handleSubmit που εξηγήθηκε παραπάνω. Αν υπάρχει μήνυμα σφάλματος, εμφανίζεται κάτω από τη φόρμα και τέλος παρέχεται ένα κουμπί για την ανακατεύθυνση στη σελίδα εγγραφής - registration για χρήστες που δεν έχουν λογαριασμό.

Επίσης, στο Login component γίνεται χρήση της βιβλιοθήκης axios για την πραγματοποίηση του HTTP αιτήματος από το frontend προς τον server (FastAPI). H λειτουργία του axios είναι να στείλει τα διαπιστευτήρια (όνομα χρήστη και κωδικό πρόσβασης) στον server και να περιμένει για μια απάντηση - response, η οποία θα καθορίσει αν η σύνδεση του χρήστη ήταν επιτυχής ή όχι. Σε άλλα components γίνεται η χρήση του fetch αντί του axios όπως θα φανεί παρακάτω.

Στο σημείο αυτό αξίζει να αναφερθεί ότι η συγκεκριμένη βιβλιοθήκη θα πρέπει να είναι ήδη εγκατεστημένη στα πακέτα της εφαρμογής για να μπορεί να χρησιμοποιηθεί. Η εγκατάστασή της γίνεται με την παρακάτω εντολή, στο path που βρίσκεται η React εφαρμογή, δηλαδή στο directory frontend.

npm install axios

Συμπερασματικά, το Context API της React επιτρέπει στον προγραμματιστή να ορίσει μία global μεταβλητή, στην οποία μπορούν να έχουν πρόσβαση και να αλλάζουν την τιμή της τα υπόλοιπα components. Έτσι, όταν αλλάζει η τιμή της μεταβλητής user (μέσω του setUser) σε ένα component, η αλλαγή αυτή γίνεται άμεσα ορατή σε όλα τα άλλα components που καταναλώνουν το ίδιο context. Αυτό διευκολύνει τη διαχείριση της κατάστασης σε πολύπλοκες εφαρμογές, εξαλείφοντας την ανάγκη χρησιμοποίησης props και επιτρέποντας μια πιο ευέλικτη διαχείριση του state.

Στο component HomePage που ακολουθεί, μεταξύ άλλων δίνεται έμφαση στον τρόπο λειτουργίας των dropdowns που χρησιμοποιούνται στο navigation bar της εφαρμογής.

import React, { useContext, useState, useEffect } from 'react';

import { Link, useNavigate } from 'react-router-dom';

import { UserContext } from './UserContext';

import backgroundImage from '../data/grandma.jpg';

const HomePage = () => {

const { user, setUser } = useContext(UserContext);

const navigate = useNavigate();

const [isDropdownOpen, setIsDropdownOpen] = useState(false);

const [imageLoaded, setImageLoaded] = useState(false);

const [adminDropdownOpen, setAdminDropdownOpen] = useState(false);

const [searchDropdownOpen, setSearchDropdownOpen] = useState(false);

useEffect(() => {

const img = new Image();

img.src = backgroundImage;

img.onload = () => {

setImageLoaded(true);

};

}, []);

const handleLogout = () => {

setUser(null); // Clear the user information from the context

localStorage.removeItem('user'); // Clear the user information from localStorage

console.log(`user ${user ? user.username : ''} with id: ${user.id} is logged out`);

navigate('/login');

};

const toggleDropdown = () => {

setIsDropdownOpen(!isDropdownOpen);

};

const toggleAdminDropdown = () => {

setAdminDropdownOpen(!adminDropdownOpen);

};

const toggleSearchDropdown = () => {

setSearchDropdownOpen(!searchDropdownOpen);

};

if (!imageLoaded) {

return null; // Εμφανίζει το κενό μέχρι η εικόνα να φορτώσει

}

return (

<div style={pageContainerStyle}>

<div style={containerStyle}>

<h2 style={welcomeStyle}>Welcome <strong style={{ color: 'green' }}>{user ? user.username : ''}</strong> to the Recipe Treasure App</h2>

<nav style={navStyle}>

<div style={navButtonsContainer}>

<Link to="/add-ingredient">

<button style={buttonStyle}>Add a New Ingredient</button>

</Link>

<Link to="/add-recipe">

<button style={buttonStyle}>Add a New Recipe</button>

</Link>

{/\* Dropdown for My inventory \*/}

<div style={{ position: 'relative', display: 'inline-block' }}>

<button onClick={toggleDropdown} style={buttonStyle}>

My inventory

</button>

{isDropdownOpen && (

<div style={dropdownStyle}>

<Link to="/my-recipes" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyle,

backgroundColor: 'green',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightgreen';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'green';

}}

>

Show my recipes

</button>

</Link>

<Link to="/my-ingredients" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyle,

backgroundColor: 'green',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightgreen';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'green';

}}

>

Show my ingredients

</button>

</Link>

<Link to="/my-processed-ingredients" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyle,

backgroundColor: 'green',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightgreen';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'green';

}}

>

Show my processed ingredients

</button>

</Link>

</div>

)}

</div>

<div style={{ position: 'relative', display: 'inline-block' }}>

<button onClick={toggleSearchDropdown} style={buttonStyle}>

Search recipes

</button>

{searchDropdownOpen && (

<div style={dropdownStyle}>

<Link to="/recipes-per-region" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyle,

backgroundColor: 'green',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightgreen';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'green';

}}

>

Recipes per region

</button>

</Link>

<Link to="/recipes-per-name" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyle,

backgroundColor: 'green',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightgreen';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'green';

}}

>

Recipes per name

</button>

</Link>

</div>

)}

</div>

<button onClick={handleLogout} style={logoutButtonStyle}>Logout</button>

{user && user.isAdmin && (

<div style={{ position: 'relative', display: 'inline-block' }}>

<button onClick={toggleAdminDropdown} style={buttonStyleAdmin}>

Admin

</button>

{adminDropdownOpen && (

<div style={dropdownStyle}>

<Link to="/users/" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyleAdmin,

backgroundColor: 'blue',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightblue';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'blue';

}}

>

Show users

</button>

</Link>

<Link to="/delete-user" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyleAdmin,

backgroundColor: 'blue',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightblue';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'blue';

}}

>

Delete User

</button>

</Link>

<Link to="/delete-ingredient" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyleAdmin,

backgroundColor: 'blue',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightblue';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'blue';

}}

>

Delete Ingredient

</button>

</Link>

<Link to="/delete-recipe" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyleAdmin,

backgroundColor: 'blue',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightblue';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'blue';

}}

>

Delete Recipe

</button>

</Link>

<Link to="/delete-processed-ingredient" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyleAdmin,

backgroundColor: 'blue',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightblue';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'blue';

}}

>

Delete Processed Ingredient

</button>

</Link>

<Link to="/update-ingredient" style={{ textDecoration: 'none' }}>

<button style={dropdownItemStyleAdmin}>Update Ingredient</button>

</Link>

<Link to="/update-recipe" style={{ textDecoration: 'none' }}>

<button style={dropdownItemStyleAdmin}>Update Recipe</button>

</Link>

</div>

)}

</div>

)}

</div>

</nav>

</div>

</div>

);

};

const pageContainerStyle = {

position: 'relative',

minHeight: '100vh',

backgroundImage: `url(${backgroundImage})`,

backgroundSize: 'cover',

backgroundPosition: 'center',

backgroundRepeat: 'no-repeat',

};

const containerStyle = {

textAlign: 'center',

padding: '20px',

backgroundColor: 'black',

borderRadius: '5px',

position: 'fixed',

top: 0,

left: 0,

width: '100%',

boxSizing: 'border-box',

zIndex: 1000,

};

const welcomeStyle = {

fontSize: '18px',

color: 'white',

margin: 0,

};

const navStyle = {

backgroundColor: 'black',

padding: '10px',

borderRadius: '5px',

boxShadow: '0 4px 8px rgba(0, 0, 0, 0.1)',

marginTop: '10px',

};

const navButtonsContainer = {

display: 'flex',

justifyContent: 'center',

flexWrap: 'wrap',

};

const buttonStyle = {

fontSize: '14px',

padding: '10px 20px',

margin: '10px',

backgroundColor: 'green',

color: 'white',

border: 'none',

borderRadius: '5px',

cursor: 'pointer',

transition: 'background-color 0.3s, transform 0.3s',

};

const buttonStyleAdmin = {

fontSize: '14px',

padding: '10px 20px',

margin: '10px',

backgroundColor: 'blue',

color: 'white',

border: 'none',

borderRadius: '5px',

cursor: 'pointer',

transition: 'background-color 0.3s, transform 0.3s',

};

const dropdownStyle = {

position: 'absolute',

backgroundColor: 'white',

boxShadow: '0px 8px 16px 0px rgba(0,0,0,0.2)',

zIndex: 1,

borderRadius: '5px',

overflow: 'hidden',

};

const dropdownItemStyle = {

display: 'block',

width: '100%',

padding: '10px',

backgroundColor: 'green',

color: 'white',

textAlign: 'left',

border: 'none',

borderRadius: '0',

cursor: 'pointer',

transition: 'background-color 0.3s',

};

const dropdownItemStyleAdmin = {

display: 'block',

width: '100%',

padding: '10px',

backgroundColor: 'blue',

color: 'white',

textAlign: 'left',

border: 'none',

borderRadius: '0',

cursor: 'pointer',

transition: 'background-color 0.3s',

};

const logoutButtonStyle = {

backgroundColor: 'red',

color: 'white',

fontSize: '14px',

padding: '10px 20px',

margin: '10px',

width: '100px',

border: 'none',

borderRadius: '5px',

cursor: 'pointer',

transition: 'background-color 0.3s, transform 0.3s',

};

export default HomePage;

Στον παραπάνω κώδικα, όπως και στα προηγούμενα components που εξετάστηκαν, για τον ορισμό του component γίνονται τα κατάλληλα imports και ακολουθεί ο ορισμός του μέσω των hooks που χρησιμοποιούνται σε αυτό. Στην χρήση του hook useState έχει γίνει αναφορά για τον τρόπο με τον οποίο λειτουργεί, επομένως είναι θεμιτό να εξεταστεί σε αυτό το component το hook useEffect, το οποίο χρησιμοποιείται για την φόρτωση της εικόνας που εμφανίζεται στην Home page. Έτσι λοιπον, το useEffect εκτελείται εδώ μόνο μία φορά όταν το component φορτώνεται. Φορτώνει την εικόνα backgroundImage και όταν αυτή φορτωθεί, ενημερώνει το state imageLoaded σε true.

Η συνάρτηση handleLogout διαχειρίζεται την αποσύνδεση του χρήστη από το σύστημα. Αρχικά, καθαρίζει το state του χρήστη (user) θέτοντάς το σε null, χρησιμοποιώντας τη συνάρτηση setUser που παρέχεται από το UserContext. Παράλληλα, αφαιρεί τα δεδομένα του χρήστη από το localStorage χρησιμοποιώντας τη μέθοδο removeItem, διαγράφοντας έτσι τις πληροφορίες του χρήστη από τον τοπικό αποθηκευτικό χώρο. Τέλος, η συνάρτηση κατευθύνει τον χρήστη στη σελίδα εισόδου (/login) χρησιμοποιώντας τη navigate από το React Router.

Μέσω των συναρτήσεων toggleDropdown, toggleAdminDropdown και toggleSearchDropdown γίνεται η διαχείριση των dropdowns που εμφανίζονται στην σελίδα. Πιο συγκεκριμένα, κάθε dropdown έχει ένα state που ελέγχει αν είναι ανοιχτό (open) ή κλειστό (closed). Αυτό το state ορίζεται με τη χρήση του React hook useState.

Στο παρακάτω παράδειγμα,

const [isDropdownOpen, setIsDropdownOpen] = useState(false);

το isDropdownOpen είναι μια boolean τιμή (true ή false) που καθορίζει αν το dropdown είναι ανοιχτό. Το setIsDropdownOpen είναι η συνάρτηση που αλλάζει αυτήν την τιμή. Η συνάρτηση toggleDropdown, αλλάζει το state σε true αν είναι false (δηλαδή ανοίγει το dropdown), και το αντίστροφο (κλείνει το dropdown αν είναι ανοιχτό). Μέσω της τεχνικής του conditional rendering, το περιεχόμενο του dropdown εμφανίζεται μόνο όταν η τιμή του isDropdownOpen είναι true. Αν το isDropdownOpen είναι false, η React δεν θα εμφανίσει τίποτα στην σελίδα. Αν είναι true, θα εμφανίσει το περιεχόμενο που περιέχεται στο παρακάτω div.

<button onClick={toggleDropdown} style={buttonStyle}>

My inventory

</button>

{isDropdownOpen && (

<div style={dropdownStyle}>

<Link to="/my-recipes" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyle,

backgroundColor: 'green',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightgreen';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'green';

}}

>

Show my recipes

</button>

</Link>

<Link to="/my-ingredients" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyle,

backgroundColor: 'green',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightgreen';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'green';

}}

>

Show my ingredients

</button>

</Link>

<Link to="/my-processed-ingredients" style={{ textDecoration: 'none' }}>

<button

style={{

...dropdownItemStyle,

backgroundColor: 'green',

transition: 'background-color 0.3s',

cursor: 'pointer',

}}

onMouseEnter={(e) => {

e.target.style.backgroundColor = 'lightgreen';

}}

onMouseLeave={(e) => {

e.target.style.backgroundColor = 'green';

}}

>

Show my processed ingredients

</button>

</Link>

</div>

)}

Στον κώδικα παρατηρείται και η χρήση του component Link. Το Link είναι ένα React component που παρέχεται από τη βιβλιοθήκη react-router-dom, και χρησιμοποιείται για τη δημιουργία συνδέσμων (links) μέσα στην εφαρμογή React, οι οποίοι επιτρέπουν την πλοήγηση μεταξύ διαφορετικών routes χωρίς να χρειάζεται η σελίδα να φορτώνεται ξανά. Έτσι, ο χρήστης πατώντας το dropdown Μy inventory ανοίγουν κάποιες επιλογές. Μία από αυτές είναι το κουμπί Show my recipes που αν ο χρήστης το επιλέξει τότε μεταφέρεται στην σελίδα /my-recipes της εφαρμογής.

Όπως έχει αναφερθεί, η επικοινωνία μεταξύ των components μπορεί να επιτευχθεί με δύο βασικούς τρόπους. Είτε με την χρήση του context API που εξετάστηκε πριν, είτε με την βοήθεια των props. Στο επόμενο component που θα εξεταστεί λοιπόν μεταξύ άλλων αναδεικνύεται ακριβώς αυτό, πως δηλαδή μέσω των props επιτυγχάνεται η επικοινωνία και η ανταλλαγή δεδομένων μεταξύ δύο διαφορετικών components, όταν ένα parent component δηλαδή καταναλώνει δεδομένα μέσα από ένα child. Ο παρακάτω κώδικας αφορά στο component AddIngredient. Γενικά, το component αυτό επιτρέπει στον χρήστη να προσθέσει ή να ενημερώσει ένα υλικό - ingredient. Χρησιμοποιεί hooks για τη διαχείριση των καταστάσεων (state) και των ελέγχων φόρμας, και επικοινωνεί με το backend server μέσω HTTP αιτημάτων.

import React, { useState, useContext, useRef } from 'react';

import { UserContext } from './UserContext';

import styles from './AddIngredients.module.css';

import TreeComponent from './TreeComponent';

import SearchIngredient from './SearchIngredient';

import data from '../data/final\_extracted\_data.json';

import { Link, useNavigate } from 'react-router-dom';

import PlaceOfOriginDropdown from './PlaceOfOriginDropdown';

const AddIngredient = () => {

const navigate = useNavigate();

const { user, setUser } = useContext(UserContext);

const [existingIngredient, setExistingIngredient] = useState(null);

const [showUpdateDescription, setShowUpdateDescription] = useState(false);

const [ingredient, setIngredient] = useState({

ingredient\_name: '',

ingredient\_base\_term: '',

ingredient\_base\_term\_name: '',

ingredient\_with\_facets: [],

place\_of\_origin: '',

is\_POP: false,

is\_PGE: false,

latin\_name: '',

greek\_name: '',

greek\_alter\_name: '',

description: [{ user\_input: '' }]

});

const dropdownRef = useRef(null); // Create a ref for the dropdown component

const handleChange = (e) => {

const { name, value, type, checked } = e.target;

if (name === 'description') {

setIngredient({

...ingredient,

description: [{ user\_input: value }]

});

} else {

setIngredient({

...ingredient,

[name]: type === 'checkbox' ? checked : value

});

}

};

const handleFacetChange = (index, e) => {

const { name, value } = e.target;

const newFacets = ingredient.ingredient\_with\_facets.map((facet, i) =>

i === index ? { ...facet, [name]: value } : facet

);

setIngredient({ ...ingredient, ingredient\_with\_facets: newFacets });

};

const addFacet = () => {

setIngredient({

...ingredient,

ingredient\_with\_facets: [...ingredient.ingredient\_with\_facets, { name: '', code: '' }]

});

};

const handleSubmit = async (e) => {

e.preventDefault();

if (user) {

console.log('User ID:', user.id); // Log the user ID to ensure it's being retrieved correctly

try {

const response = await fetch(`http://localhost:8000/ingredients/?user\_id=${user.id}`, {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(ingredient)

});

const data = await response.json();

if (!response.ok) {

alert(data.detail['message'] || 'Failed to update description.'); // Show error message in alert

setShowUpdateDescription(true);

setExistingIngredient(data.detail['ingredient\_id']);

alert('Ingredient already exists. You can update the description.');

throw new Error('Network response was not ok');

}

alert('Ingredient added successfully!');

handleClear(); // Clear the form after successful submission

} catch (error) {

console.error('There was a problem with your fetch operation:', error);

}

} else {

alert('User is not logged in.');

}

};

const handleUpdateDescription = async (e) => {

e.preventDefault();

if (user && existingIngredient) {

try {

const response = await fetch(`http://localhost:8000/ingredients/${existingIngredient}/description/${ingredient.description[0].user\_input}`, {

method: 'PUT',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ user\_input: ingredient.description[0].user\_input })

});

const data = await response.json();

if (!response.ok) {

alert(data.detail || 'Failed to update description.'); // Show error message in alert

throw new Error('Network response was not ok');

}

// Handle successful response

alert('Description updated successfully!');

setShowUpdateDescription(false);

handleClear(); // Clear the form after successful update

} catch (error) {

console.error('There was a problem with your fetch operation:', error);

}

} else {

alert('User is not logged in or no existing ingredient found.');

}

};

const handleClear = () => {

setIngredient({

ingredient\_name: '',

ingredient\_base\_term: '',

ingredient\_base\_term\_name: '',

ingredient\_with\_facets: [],

place\_of\_origin: '',

is\_POP: false,

is\_PGE: false,

latin\_name: '',

greek\_name: '',

greek\_alter\_name: '',

description: [{ user\_input: '' }]

});

dropdownRef.current.resetDropdown(); // Reset the dropdown component

};

const handleLogout = () => {

setUser(null); // Clear the user information from the context

localStorage.removeItem('user'); // Clear the user information from localStorage

console.log(`user ${user ? user.username : ''} with id: ${user.id} is logged out`);

navigate('/login');

};

const handleSubmitData = (jsonData) => {

if (jsonData.length > 0) {

setIngredient({

...ingredient,

ingredient\_base\_term: jsonData[0].code,

ingredient\_base\_term\_name: jsonData[0].name

});

}

};

const handleSubmitFacet = (facetData) => {

setIngredient((prevState) => {

const newFacets = [...prevState.ingredient\_with\_facets];

const lastFacetIndex = newFacets.length - 1;

newFacets[lastFacetIndex] = {

name: facetData.name,

code: facetData.code

};

return {

...prevState,

ingredient\_with\_facets: newFacets

};

});

};

const handleSelectBaseTerm = (selectedData) => {

setIngredient({

...ingredient,

ingredient\_base\_term: selectedData.code,

ingredient\_base\_term\_name: selectedData.name

});

};

const handleSelectRegion = (region) => {

setIngredient((prevState) => ({

...prevState,

place\_of\_origin: region

}));

};

const handleCancelFacet = (index) => {

const newFacets = [...ingredient.ingredient\_with\_facets];

newFacets.splice(index, 1);

setIngredient({ ...ingredient, ingredient\_with\_facets: newFacets });

};

return (

<div>

<div style={{ display: 'flex', justifyContent: 'space-between', alignItems: 'center', backgroundColor: 'black'}}>

<h8 style={{color:'white'}}>Welcome <strong style={{color:'green'}}>{user ? user.username : ''}</strong> to the Recipe Treasure App</h8>

<h1 style={{ textAlign: 'center', marginTop: '10px', flex: '1' , color: 'white'}}>Add Ingredient Page</h1>

<Link to="/home" style={{ textDecoration: 'none' }}>

<button style={{ margin: '10px', width: '60%', backgroundColor:'blue', color: 'white', marginRight: '80px'}}>Home</button>

</Link>

<button onClick={handleLogout} style={{ margin: '10px', width: '5%', backgroundColor:'red'}}>Logout</button>

</div>

<div style={{ display: 'flex', justifyContent: 'space-between', alignItems: 'center' }}>

<SearchIngredient onSelectBaseTerm={handleSelectBaseTerm} />

</div>

<div className={styles.container}>

<div className={styles.treeContainer}>

<TreeComponent data={data} onSubmitData={handleSubmitData} onSubmitFacet={handleSubmitFacet} /> {/\* Render the tree component \*/}

</div>

<div className={styles.formContainer}>

<form onSubmit={handleSubmit}>

<div className={styles.inputGroup}>

<label>Ingredient Name:</label>

<input type="text" name="ingredient\_name" value={ingredient.ingredient\_name} onChange={handleChange} required />

</div>

<div className={styles.inputGroup}>

<label>Ingredient Base Term:</label>

<input type="text" name="ingredient\_base\_term" value={ingredient.ingredient\_base\_term} onChange={handleChange} required readOnly />

</div>

<div className={styles.inputGroup}>

<label>Ingredient Base Term Name:</label>

<input type="text" name="ingredient\_base\_term\_name" value={ingredient.ingredient\_base\_term\_name} onChange={handleChange} required readOnly />

</div>

<div className={styles.inputGroup}>

<br></br>

<PlaceOfOriginDropdown onSelectRegion={handleSelectRegion} ref={dropdownRef} /> {/\* Use the ref here \*/}

<br></br>

</div>

<div className={styles.inputGroup}>

<label className={styles.label}>

Is POP:

<input type="checkbox" name="is\_POP" checked={ingredient.is\_POP} onChange={handleChange} className={styles.checkbox} />

</label>

<label className={styles.label}>

Is PGE:

<input type="checkbox" name="is\_PGE" checked={ingredient.is\_PGE} onChange={handleChange} className={styles.checkbox} />

</label>

</div>

<div>

</div>

<div className={styles.inputGroup}>

<label>Latin Name:</label>

<input type="text" name="latin\_name" value={ingredient.latin\_name} onChange={handleChange} />

</div>

<div className={styles.inputGroup}>

<label>Greek Name:</label>

<input type="text" name="greek\_name" value={ingredient.greek\_name} onChange={handleChange} required />

</div>

<div className={styles.inputGroup}>

<label>Greek Alter Name:</label>

<input type="text" name="greek\_alter\_name" value={ingredient.greek\_alter\_name} onChange={handleChange} required />

</div>

<div className={styles.inputGroup}>

<label>Description:</label>

<textarea

name="description"

value={ingredient.description[0].user\_input}

onChange={handleChange}

style={{ height: '50px', width: '100%', resize: 'vertical' }}

/>

</div>

<div className={styles.inputGroup}>

<label>Facets:</label>

{ingredient.ingredient\_with\_facets.map((facet, index) => (

<div key={index} className={styles.facetGroup}>

<input

type="text"

name="name"

placeholder="Facet Name"

value={facet.name}

onChange={(e) => handleFacetChange(index, e)}

required readOnly

/>

<input

type="text"

name="code"

placeholder="Facet Code"

value={facet.code}

onChange={(e) => handleFacetChange(index, e)}

required readOnly

/>

<button style={{ margin: '5px', width: '60%', backgroundColor:'red', color: 'white', marginRight: '15px'}} type="button" onClick={() => handleCancelFacet(index)} className={styles.cancelButton}>Remove</button>

</div>

))}

<button type="button" onClick={addFacet} style={{ margin: '5px', width: '50%', backgroundColor:'green', color: 'white', marginRight: '15px', fontSize: '0.8rem', padding: '5px 10px', borderRadius: '5px'}}>Add Another Facet</button>

</div>

<div className={styles.buttonGroup}>

<button type="submit" style={{ margin: '5px', width: '60%', backgroundColor:'blue', color: 'white', marginRight: '15px'}}>Add Ingredient</button>

<button type="button" onClick={handleClear} style={{ margin: '5px', width: '30%', backgroundColor:'blue', color: 'white', marginRight: '5px'}}>Clear</button>

</div>

</form>

{showUpdateDescription && (

<div>

<h5>Ingredient already exists. Add another description:</h5>

<form onSubmit={handleUpdateDescription}>

<div className={styles.inputGroup}>

<label>New Description:</label>

<textarea

name="description"

value={ingredient.description[0].user\_input}

onChange={handleChange}

style={{ height: '50px', width: '100%', resize: 'vertical' }}

/>

</div>

<div className={styles.buttonGroup}>

<button type="submit" style={{ margin: '5px', width: '60%', backgroundColor:'blue', color: 'white', marginRight: '15px'}}>Add Description</button>

</div>

</form>

</div>

)}

</div>

</div>

</div>

);

};

export default AddIngredient;

Για μια πιο ολοκληρωμένη και οργανωμένη παρουσίαση του component αυτού θα αναλυθεί ο κώδικας κομμάτι - κομμάτι.

................................................................

Η παρουσίαση κάποιων βασικών components της εφαρμογής της ενότητας αυτής, αποσκοπεί στην κατανόηση, του τρόπου με τον οποίο τα components μπορούν να επικοινωνήσουν μεταξύ τους, της χρήσης των κυριότερων hooks και κάποιων επιμέρους βιβλιοθηκών που χρησιμοποιούνται στην React ενώ παράλληλα καλείται να τεκμηριώσει τη λογική που εφαρμόστηκε ώστε να επιτευχθεί η θεμιτή λειτουργικότητα της εφαρμογής.

**Κεφ.9: Περιγραφή UI**

Στην παρούσα ενότητα παρουσιάζεται εν γένει η εφαρμογή και οι λειτουργίες που επιτελεί. Η παρουσίαση επιχειρείται με τη βοήθεια διάφορων screenshots. Σε αυτό το σημείο θα πρέπει να επισημανθεί ότι τα user interfaces διαφοροποιούνται ανάλογα με τους ρόλους των χρηστών. Για λόγους πληρότητας, θα χρησιμοποιηθεί ο ρόλος του admin, στο UI του οποίου προβάλλονται τόσο οι λειτουργίες του απλού user όσο και κάποιες επιπρόσθετες λειτουργίες διαχείρισης που αφορούν στον admin. Σενάρια χρήσης και δοκιμές θα εξεταστούν στο 6ο κεφάλαιο. Σκοπός της ενότητας είναι να αναδείξει λοιπόν, τις βασικές οθόνες από τις οποίες αποτελείται η εφαρμογή με στόχο την εξοικείωση του μελλοντικού χρήστη με το περιβάλλον του συστήματος.

1. Login page

A screen shot of a computer

Description automatically generated

Ο χρήστης δίνει τα credentials του (username και password), και εισέρχεται στην εφαρμογή πατώντας το κουμπί Login. Αν δεν έχει ήδη έναν λογαριασμό τότε μπορεί να πατήσει το κουμπί Go to registration για να δημιουργήσει έναν νέο λογαριασμό.

2. Register page

A screenshot of a login form

Description automatically generated

Ο χρήστης δίνει τα credentials που του ζητούνται (username, email και password) και πατώντας το κουμπί Register ολοκληρώνει την εγγραφή του. Πλέον, μπορεί να επιστρέψει στη Login page και να εισέλθει στο σύστημα.

3. Home page

A person's hands kneading dough

Description automatically generated

Η Home page περιέχει ένα μήνυμα καλωσορίσματος αποτελούμενο από το username του χρήστη που έχει εισέλθει στην εφαρμογή (εδώ είναι το admin). Το navigation bar αποτελείται από έξι διαφορετικά κουμπιά. Τα κουμπιά My inventory, Search recipes και Admin αποτελούνται από κάποια επιμέρους κουμπιά που εμφανίζονται ως επιλογές σε dropdowns, όπως φαίνεται στην εικόνα. Τα κουμπιά με τις υποκατηγορίες αυτών είναι τα εξής:

1. Add a New Ingredient
2. Add a New Recipe
3. My inventory

3.1 Show my recipes

3.2 Show my ingredients

3.3 Show my processed ingredients

1. Search recipes

4.1 Recipes per region

4.2 Recipes per name

1. Logout
2. Admin

6.1 Show users

6.2 Delete User

6.3 Delete Ingredient

6.4 Delete Processed Ingredient

6.5 Update Ingredient

6.6 Update Recipe

Στην συνέχεια θα δούμε όλες τις βασικές οθόνες στις οποίες ο χρήστης μπορεί να περιηγηθεί ανάλογα με το κουμπί που επιλέγει να πατήσει.

4. Add a New Ingredient

A screenshot of a computer

Description automatically generated

Από την Home page πατώντας το κουμπί Add a New Ingredient ο χρήστης μεταφέρεται στην Add a New Ingredient page. Σε αυτή την οθόνη ο χρήστης μπορεί να προσθέσει ένα νέο συστατικό συμπληρώνοντας την φόρμα η οποία αποτελείται από πεδία, dropdowns και radio buttons. Τα πεδία Ingredient Name, Latin Name, Greek Name, Alter Greek Name και Description συμπληρώνονται manually ενώ τα πεδία Ingredient Base Term, Ingredient Base Term Name και τα Facet Name και Facet Code της κατηγορίας Facets συμπληρώνονται αυτόματα ανάλογα με την επιλογή του node που θα κάνει ο χρήστης από το δέντρο (TreeComponent) και το κουμπί που θα πατήσει ανάμεσα στα Select Base Term και Select Facet. Το κουμπί Add Another Facet δίνει την δυνατότητα της προσθήκης επιπλέον Facets. Οπότε όταν πατηθεί ανοίγουν δύο νέα πεδία Facet Name και Facet Code. Το κουμπί Remove δίπλα από κάθε συνδυασμό των πεδίων Facet Name και Facet Code, όταν πατηθεί αφαιρεί τα πεδία αυτά. Επιπλέον, το κουμπί Add Ingredient όταν πατηθεί έχει ως αποτέλεσμα την προσθήκη του νέου συστατικού στη ΒΔ. Το κουμπί clear καθαρίζει την φόρμα ενώ το κουμπί Home πάνω δεξιά επιστρέφει τον χρήστη στην Home page και τέλος το κουμπί Logout βγάζει τον χρήστη από το σύστημα. Στο επόμενο κεφάλαιο που αφορά στο στάδιο των δοκιμών, εκτελούνται διάφορα σενάρια που αποσαφηνίζουν τον τρόπο με τον οποίο εισάγεται ένα νέο συστατικό. Εκεί θα αναφερθεί και ο ρόλος του component Ingredient search machine που βρίσκεται επάνω και αριστερά στην οθόνη.

5. Add a New Recipe

A screenshot of a computer

Description automatically generated

Η ίδια λογική με την οθόνη του Add a New Ingredient ακολουθείται και στην παραπάνω, με τη διαφορά ότι έχουν προστεθεί κάποια επιπλέον στοιχεία, όπως το κουμπί Go to process και τα components που αφορούν στην αναζήτηση συστατικών και επεξεργασμένων συστατικών. Το κουμπί Go to process κατευθύνει τον χρήστη σε μία νέα οθόνη που αφορά στην προσθήκη ενός επεξεργασμένου υλικού, ενώ στα πεδία Search ingredient και Search processed ingredient μπορεί ο χρήστης να εισάγει το όνομα ενός υλικού και ενός επεξεργασμένου υλικού και έπειτα πατώντας το κουμπί Find να προβληθούν τα αποτελέσματα των αναζητήσεων. Επιλέγοντας από τα αποτελέσματα αναζήτησης μπορεί να προσθέτει Facets ή και Processed ingredients για την συνταγή που δημιουργεί, αντλώντας την πληροφορία απευθείας από τη ΒΔ και χωρίς να χρειάζεται να γίνεται επιλογή ενός Facet από το δέντρο (στην περίπτωση των Facets) ή να χρειάζεται να δημιουργηθεί on the fly ένα νέο επεξεργασμένο υλικό (στην περίπτωση των Processed ingredients). Τα παραπάνω αποτελούν και τις βασικές διαφορές σε σχέση με την οθόνη του Add a New Ingredient.

6. Add a New Processed Ingredient

A screenshot of a computer

Description automatically generated

Ο χρήστης μεταφέρεται σε αυτή την οθόνη μέσω της προηγούμενης που εξετάστηκε (της Add a New Recipe) όταν πατηθεί το κουμπί Go to process. Στην οθόνη αυτή ακολουθείται η ίδια λογική που ακολουθείται και στις οθόνες Add a New Recipe και Add a New Ingredient με τη διαφορά ότι όταν πατηθεί το κουμπί Add Processed Ingredient ο χρήστης μεταφέρεται ξανά πίσω στην οθόνη Add a New Recipe για να ολοκληρώσει την εισαγωγή της συνταγής. Η παραπάνω οθόνη δεν είναι βασική αλλά προαιρετική και συμμετέχει κατά την εισαγωγή μιας συνταγής.

7. My inventory ---> Show my recipes

A screenshot of a computer

Description automatically generated

Στην παραπάνω οθόνη ο χρήστης μπορεί να μεταβεί πατώντας από το κεντρικό μενού της Home page, το κουμπί My inventory και έπειτα το κουμπί Show my recipes. Τότε εμφανίζεται η λίστα με τις συνταγές που έχουν καταχωρηθεί από τον χρήστη σε έναν πίνακα. Έπειτα ο χρήστης μπορεί να επιλέξει την συνταγή που επιθυμεί (τα rows του πίνακα είναι επιλέξιμα). Αν οι συνταγές είναι πάρα πολλές, τότε με τα κουμπιά Previous και Next, ο χρήστης μπορεί να περιηγηθεί μεταξύ των επιλογών, μέχρι να βρει την συνταγή που τον ενδιαφέρει. Οι πληροφορίες της συνταγής που επιλέχτηκε εμφανίζονται στα δεξιά της οθόνης όπως φαίνεται στην παραπάνω εικόνα και ο χρήστης έχει τη δυνατότητα να εκτυπώσει αυτές τις πληροφορίες πατώντας το κουμπί Print. Το αποτέλεσμα της επιλογής Print είναι ένα αρχείο pdf που ανοίγει αυτόματα με το πρόγραμμα εκτύπωσης που παρέχεται από το λειτουργικό σύστημα και είναι έτοιμο να εκτυπωθεί, όπως φαίνεται στην επόμενη εικόνα.

A screenshot of a computer

Description automatically generated

8. My inventory ---> Show my ingredients

A screenshot of a recipe

Description automatically generated

Στην παραπάνω οθόνη ο χρήστης μπορεί να μεταβεί πατώντας από το κεντρικό μενού της Home page, το κουμπί My inventory και έπειτα το κουμπί Show my ingredients. Στο UI εμφανίζεται ένας πίνακας που περιέχει όλα τα υλικά που έχουν καταχωρηθεί στο σύστημα από τον συγκεκριμένο χρήστη.

9. My inventory ---> Show my processed ingredients

A screenshot of a computer

Description automatically generated

Στην παραπάνω οθόνη ο χρήστης μπορεί να μεταβεί πατώντας από το κεντρικό μενού της Home page, το κουμπί My inventory και έπειτα το κουμπί Show my processed ingredients. Στο UI εμφανίζεται ένας πίνακας που περιέχει όλα τα επεξεργασμένα υλικά που έχουν καταχωρηθεί στο σύστημα από τον συγκεκριμένο χρήστη.

10. Search recipes ---> Recipes per region

A screenshot of a computer

Description automatically generated

Στην παραπάνω οθόνη ο χρήστης μπορεί να μεταβεί πατώντας από το κεντρικό μενού της Home page, το κουμπί Search recipes και έπειτα το κουμπί Recipes per region. Ο χρήστης μπορεί να επιλέξει από το dropdown Place of origin το γεωγραφικό διαμέρισμα που τον ενδιαφέρει. Πατώντας το κουμπί search θα προκύψουν αποτελέσματα με τα ονόματα των συνταγών που έχουν καταχωρηθεί από όλους τους χρήστες του συστήματος και που αντιστοιχούν στο origin που επέλεξε, σε έναν πίνακα, όπως φαίνεται στην παραπάνω εικόνα. Ο χρήστης στην συνέχεια μπορεί να επιλέξει το row με την συνταγή που τον ενδιαφέρει και τότε εμφανίζονται όλες οι πληροφορίες της επιλεγμένης συνταγής στο δεξί μέρος της οθόνης όπως φαίνεται και στην εικόνα που ακολουθεί.

A screenshot of a computer

Description automatically generated

Στην συνέχεια ο χρήστης μπορεί να προχωρήσει πατώντας το κουμπί Print για να εκτυπώσει την συγκεκριμένη συνταγή.

11. Search recipes ---> Recipes per name

A screenshot of a computer

Description automatically generated

Στην ίδια λογική με την προηγούμενη οθόνη, ο χρήστης μπορεί να πραγματοποιήσει μία αναζήτηση συνταγής με το όνομα της συνταγής, γράφοντας στο πεδίο ένα μέρος της λέξης που περιγράφει το όνομα μιας συνταγής, όπως φαίνεται στην παραπάνω εικόνα. Στην συνέχεια πατώντας το κουμπί Find, εμφανίζονται οι συνταγές που εμπεριέχουν τον όρο που έδωσε ο χρήστης ως λέξη κλειδί. Οι συνταγές που εμφανίζονται στον πίνακα των αποτελεσμάτων αφορούν συνταγές που έχουν καταχωρηθεί από όλους τους χρήστες του συστήματος. Και σε αυτή την οθόνη, ο χρήστης μπορεί επιλέγοντας την συνταγή που τον ενδιαφέρει να δει τις πληροφορίες σχετικά με αυτήν και έπειτα παρέχεται στον χρήστη η δυνατότητα εκτύπωσής της.

12. Admin ---> Show users

A screenshot of a computer

Description automatically generated

Ο Admin μπορεί να ελέγξει την τρέχουσα κατάσταση των χρηστών του συστήματος (ποιοι χρήστες είναι active) πατώντας το κουμπί Admin και έπειτα το κουμπί Show users. Ο πίνακας που εμφανίζεται προβάλλει κάποιες πληροφορίες των χρηστών, όπως το id, το username και το email του κάθε χρήστη, ακριβώς όπως φαίνεται στην παραπάνω εικόνα.

13. Admin ---> Delete user

Οι επόμενες τρεις οθόνες αφορούν σε λειτουργίες του deletion ενός user, ενός υλικού ή ενός επεξεργασμένου υλικού και έχουν ακριβώς το ίδιο UI. Οπότε δεν χρειάζεται να παρουσιαστούν και οι τρεις οθόνες ξεχωριστά αφού το μόνο που χρειάζεται κάθε φορά ως input από τον admin είναι τo id του προς διαγραφή χρήστη, το id προς διαγραφή υλικού και το id του προς διαγραφή επεξεργασμένου υλικού. Και οι τρεις οθόνες είναι πανομοιότυπες και έχουν το layout που φαίνεται στην επόμενη εικόνα.

A white background with black border

Description automatically generated

Ο admin μεταφέρεται στην παραπάνω οθόνη πατώντας το κουμπί Admin και έπειτα το κουμπί Delete user. Εισάγει στο πεδίο το id του χρήστη που θέλει να διαγράψει από τη ΒΔ και πατάει το κουμπί Delete user. Το σύστημα ενημερώνει τον admin με ένα popup message ότι ο user με το id που έδωσε ο admin έχει διαγραφεί, όπως φαίνεται στην εικόνα που ακολουθεί.

A screenshot of a computer

Description automatically generated

14. Admin ---> Update ingredient

Ο admin μεταφέρεται στην παραπάνω οθόνη πατώντας το κουμπί Admin και έπειτα το κουμπί Update ingredient. Εισάγει στο πεδίο το id του υλικού που θέλει να ανανεώσει και έπειτα πατώντας το κουμπί search, το σύστημα επιστρέφει μία φόρμα με τις πληροφορίες του προς ανανέωση υλικού. Ο admin μπορεί να τροποποιήσει, να διαγράψει ή και να προσθέσει μία νέα πληροφορία και στο τέλος να πατήσει το κουμπί Update ingredient ώστε να αποθηκευτούν οι αλλαγές στη ΒΔ. Παρακάτω φαίνεται η οθόνη του Update ingredient η οποία ακολουθεί την ίδια λογική της οθόνης Add a New Ingredient που παρουσιάστηκε παραπάνω.

A screenshot of a computer

Description automatically generated

Στην ενότητα αυτή παρουσιάστηκαν συνοπτικά οι βασικές οθόνες της εφαρμογής ώστε ο μελλοντικός χρήστης του συστήματος να αποκτήσει μία πρώτη εμπεριστατωμένη εικόνα σε σχέση με τις λειτουργίες και τις δυνατότητες που προσφέρει η εφαρμογή. Στο επόμενο κεφάλαιο που αφορά στις δοκιμές διαφόρων σεναρίων χρήσης, ο χρήστης του συστήματος μέσα από παραδείγματα θα έχει τη δυνατότητα να εξοικειωθεί περαιτέρω με τις βασικές λειτουργίες της εφαρμογής.

**Κεφ.10: Δοκιμές (Testing)**

Στο κεφάλαιο επιχειρούνται κάποιες βασικές δοκιμές που βασίζονται σε σενάρια χρήσης του συστήματος. Πιο συγκεκριμένα δοκιμάζονται στις βασικές οθόνες που παρουσιάστηκαν στο προηγούμενο κεφάλαιο οι εξής λειτουργίες:

1. Εισαγωγή νέου υλικού.
2. Εισαγωγή νέας συνταγής.
3. Εισαγωγή επεξεργασμένων υλικών.

**Κεφ.10.1: Το format του κωδικού της συνταγής και του υλικού**

Για την πληρέστερη κατανόηση των βασικών λειτουργιών της εφαρμογής, καταχώρηση νέου υλικού και καταχώρηση νέας συνταγής, κρίνεται σκόπιμο να γίνει αναφορά στο πως προκύπτει ο κωδικός του υλικού και της συνταγής που εισάγεται στο σύστημα βάσει της ανάλυσης των απαιτήσεων. Για κάθε κωδικό συνταγής που σχηματίζεται θα πρέπει να ακολουθείται η εξής φόρμα σύμφωνα με το πρότυπο της Foodex:

Generic-base-term#Process-facet$Generic-term-facet1$Generic-term-facet2$...$ Generic-term-facetΝ

Όπου Generic-base-term = ο βασικός όρος που μπορεί να περιγράψει καλύτερα το υλικό ή την συνταγή,

Process-facet = η επεξεργασία που έχει υποστεί το υλικό ή η συνταγή,

Ανάμεσα στο Generic-base-term και στο Process facet υπάρχει το σύμβολο “#”.

Generic-term-facet = αφορά σε επιμέρους facets από διάφορες κατηγορίες που προστίθενται στο υλικό ή στη συνταγή κατά τον σχηματισμό του κωδικού. Ο κωδικός με αυτό τον τρόπο επεκτείνεται δίνοντας με αυτό τον τρόπο μια πληρέστερη περιγραφή του εκάστοτε υλικού ή συνταγής.

Ανάμεσα στο Process-facet και στο Generic-term-facet υπάρχει πάντα το σύμβολο “$”. Για κάθε ένα facet που προστίθεται από εκεί και πέρα παρεμβάλλεται πάντοτε το “$” ως διαχωριστικό μεταξύ των facets.

Το παραπάνω format αφορά σε συνταγές που μπορούν να χαρακτηριστούν και ως πιάτα και τα οποία αποτελούνται από επιμέρους υλικά που προστίθενται ως facets. Υπάρχει όμως και άλλη μία περίπτωση συνταγών που χαρακτηρίζονται ως σύνθετα τρόφιμα. Έτσι για παράδειγμα ως πιάτο μπορεί να χαρακτηριστεί το παστίτσιο ενώ αντίστοιχα ως σύνθετο τρόφιμο το τσαλαφούτι. Σε αυτό το σημείο να σημειωθεί ότι τα σύνθετα τρόφιμα μπορούν να χρησιμοποιηθούν με την σειρά τους ως υλικά μιας συνταγής, Δηλαδή το τσαλαφούτι που είναι ένα σύνθετο τρόφιμο μπορεί να χρησιμοποιηθεί σε μία συνταγή για μακαρονάδα. Η διάκριση αυτή μεταξύ πιάτου και σύνθετου τροφίμου αλλάζει και το format του κωδικού. Έτσι λοιπόν, το format για τον σχηματισμό του κωδικού ενός σύνθετου τροφίμου είναι το εξής:

Generic-base-term#Source-facet$Process-facet1$Process-facet2... $Process-facetΝ

Η κύρια διαφορά μεταξύ των δύο έγκειται στο γεγονός ότι η επιλογή του Generic-base-term στην περίπτωση του πιάτου γίνεται από μία συγκεκριμένη κατηγορία του δέντρου, την Composite dishes (A03VA), ενώ στην περίπτωση του σύνθετου τροφίμου δεν υπάρχει κάποιο Generic-base-term ήδη στον κατάλογο της EFSA που να το περιγράφει με ακρίβεια και σαφήνεια, οπότε θα πρέπει να δημιουργηθεί με την χρήση μιας γενικότερης κατηγορίας ως Generic-base-term συμπληρωματικά με ένα πρωτογενές υλικό, δηλαδή ένα Source-facet. Στις επόμενες ενότητες εξετάζονται οι περιπτώσεις καταχώρησης συνταγών, του παστίτσιου και του τσαλαφουτιού αντίστοιχα ώστε όλα τα παραπάνω να γίνουν κατανοητά μέσω των παραδειγμάτων.

Σχετικά με το format του κωδικού για κάθε υλικό που εισάγεται στο σύστημα αυτό διαφοροποιείται ανάλογα με το αν χρειάζονται ή όχι επιμέρους facets για τον σχηματισμό του αλλά και από το αν υπάρχει ή δεν υπάρχει αντίστοιχα πρωτογενές υλικό. Όπως θα παρουσιαστεί στην συνέχεια, διακρίνουμε κάποιες περιπτώσεις καταχώρησης υλικών κατά τις δοκιμές που εφαρμόστηκαν όπου ανάλογα με την περίπτωση, ακολουθείται και διαφορετικό format σχηματισμού κωδικού του κάθε υλικού. Για παράδειγμα, στην περίπτωση που το Generic-base-term ενός υλικού υπάρχει ήδη στον κατάλογο της EFSA και αυτό αρκεί από μόνο του για να προσδιορίσει σαφώς ένα υλικό όπως είναι η ελιά ή η φάβα τότε το format αποτελείται μόνο από το Generic-base-term. Σε αυτή την περίπτωση ο κωδικός Foodex του υλικού ταυτίζεται με το Generic-base-term ενώ η διάκριση μεταξύ των ειδών της ελιάς που θα έχουν το ίδιο Generic-base-term θα γίνεται μέσω άλλων χαρακτηριστικών που αφορούν στο υλικό όπως για παράδειγμα ο τόπος παραγωγής τους.

Από την άλλη πλευρά, όταν δεν υπάρχει ήδη κάποιο Generic-base-term στον κατάλογο που θα μπορεί να περιγράψει το υλικό με σαφήνεια, τότε η χρήση των facets είναι αναγκαία. Επιπλέον, γίνεται διάκριση στα υλικά για τα οποία υπάρχει πρωτογενές υλικό ή όχι. Ως εκ τούτου, ο τρόπος με τον οποίο γίνεται η περιγραφή του υλικού σε αυτή την περίπτωση διαφοροποιείται. Έτσι το format που ακολουθείται στην περίπτωση που υπάρχει πρωτογενές υλικό είναι το εξής:

Generic-base-term#Source-facet$Generic-term-facet:other

Αντίστοιχα στην περίπτωση που δεν υπάρχει πρωτογενές υλικό, το format είναι:

Generic-base-term#Generic-term-facet:other

και εφόσον δεν υπάρχει η πηγή προέλευσης του υλικού στον κατάλογο, αυτή θα πρέπει να δίνεται περιγραφικά σε ένα πεδίο manually από τον χρήστη, Source-facet = something.

**Κεφ.10.2: Παραδείγματα Δοκιμών**

Στα παραδείγματα που ακολουθούν εξετάζονται κάποια βασικά σενάρια χρήσης προκειμένου να αποσαφηνιστούν τα διακριτά βήματα που πρέπει να ακολουθηθούν για την επιτυχή ολοκλήρωση των διαδικασιών και των λειτουργιών του συστήματος. Με τη μελέτη των δοκιμών αυτών, ο μελλοντικός χρήστης θα είναι σε θέση αφενός να κατανοήσει σε μεγαλύτερο βάθος τη λειτουργικότητα της εφαρμογής και τους περιορισμούς όπως προκύπτουν από τις απαιτήσεις και την ανάλυση των προδιαγραφών, αφετέρου να εξοικειωθεί περισσότερο με το UI εφαρμόζοντας τα παραδείγματα αυτά.

**Κεφ.10.2.1: Καταχώρηση νέου υλικού**

Υπάρχουν υλικά όπως αναφέρθηκε παραπάνω, που για τον σχηματισμό του κωδικού τους δεν απαιτούνται facets (όψεις) και άλλα υλικά που απαιτούνται. Επιπλέον, για ορισμένα υλικά δεν υπάρχει στον κατάλογο της EFSA το αντίστοιχο Generic-base-term αλλά υπάρχει το πρωτογενές υλικό το οποίο συμπληρώνεται με τον συνδυασμό επιμέρους facets. Επιπρόσθετα, για κάποια υλικά δεν υπάρχει ούτε το πρωτογενές υλικό, οπότε ως Generic-base-term χρησιμοποιείται ένας ακόμη πιο γενικός όρος που μπορεί να ανήκει σε μια κατηγορία πιο γενική μέσα στον κατάλογο. Τέλος, κάποια υλικά δύναται να αναφέρονται με το ίδιο όνομα αλλά ουσιαστικά πρόκειται για διαφορετικά υλικά.

Στα επόμενα σενάρια εξετάζονται όλες οι περιπτώσεις εισαγωγής νέων υλικών που αντιστοιχούν σε καθένα από τους περιορισμούς που αναφέρθηκαν παραπάνω. Ως εκ τούτου, για την καλύτερη κατανόηση γίνεται διάκριση των παρακάτω περιπτώσεων:

1. Περίπτωση ελιάς (δεν χρειάζονται facets)
2. Περίπτωση φάβας (δεν χρειάζονται facets, έχουν κοινό όνομα άλλα πρόκειται για διαφορετικά υλικά)
3. Περίπτωση μοσχαρίσιος κιμάς (χρειάζονται facets και έχουν πρωτογενές υλικό)
4. Περίπτωση καυκαλήθρας (χρειάζονται facets και δεν έχουν πρωτογενές υλικό)

**Κεφ.10.2.1.1: Περίπτωση ελιάς**

Στην περίπτωση καταχώρησης της “ελιάς” ο κωδικός EFSA είναι κοινός για όλα τα είδη ελιάς. Ωστόσο η διάκριση μεταξύ των υλικών που εισέρχονται στο σύστημα με το όνομα “ελιά” θα γίνεται βάσει του τόπου παραγωγής τους.

Ο χρήστης μεταφέρεται στην σελίδα Add Ingerdient Page προκειμένου να εισάγει ένα νέο υλικό με όνομα “ελιά”.

1ο βήμα:

Στο πεδίο Ingredient Name της φόρμας, ο χρήστης συμπληρώνει το όνομα “Ελιά”.

2ο βήμα:

Ο χρήστης στο βήμα αυτό, θα πρέπει να αναζητήσει το Generic-base-term που περιγράφει πληρέστερα το προς καταχώρηση υλικό. Στην περίπτωση της ελιάς, είναι το “Table olives ready for consumption” με αντίστοιχο κωδικό A01BQ. Το Generic-base-term μπορεί να αναζητηθεί με δύο διαφορετικούς τρόπους. Είτε μέσω της αναζήτησης στο Ingredient search machine δίνοντας τον όρο “ελιά” είτε μέσω της περιήγησης στο δέντρο Foodex catalogues. Ο πρώτος τρόπος δεν επιστρέφει για όλους τους ελληνικούς όρους έγκυρα και ακριβή αποτελέσματα, ωστόσο η αναζήτηση ελληνικών όρων μέσω αυτής της λειτουργικότητας μπορεί να φανεί χρήσιμη σε αρκετές περιπτώσεις. Ο δεύτερος τρόπος είναι ο πιο ασφαλής. Ωστόσο είναι πιο αργός και προϋποθέτει ότι ο χρήστης γνωρίζει τον τρόπο που μπορεί να περιηγηθεί στο δέντρο ώστε να επιλέξει τον σωστό όρο.

1ος τρόπος αναζήτησης του Generic-base-term:

A screenshot of a recipe

Description automatically generated

Ο χρήστης ψάχνει τον όρο “ελιά” και εμφανίζονται τα αποτελέσματα της αναζήτησης. Έπειτα ο χρήστης επιλέγει τη γραμμή με το αποτέλεσμα που επιθυμεί και στην συνέχεια μπορεί να πατήσει το κουμπί Select Base Term όπως φαίνεται στην επόμενη εικόνα.

A screenshot of a computer

Description automatically generated

Αυτόματα τα πεδία Ingredient Base Term και Ingredient Base Term Name της φόρμας θα συμπληρωθούν, όπως φαίνεται παρακάτω.

A screenshot of a computer

Description automatically generated

2ος τρόπος αναζήτησης του Generic-base-term:

Ο χρήστης περιηγείται στο δέντρο αναζητώντας το Generic-base-term για την “ελιά”.

A screenshot of a computer

Description automatically generated

Αφού εντοπίσει το Generic-base-term που επιθυμεί, το επιλέγει.

A screenshot of a computer

Description automatically generated

Όπως φαίνεται στην παραπάνω εικόνα, η επιλογή του χρήστη εμφανίζεται σε ένα πλαίσιο, δίνοντας πληροφορίες για τον εν λόγω όρο και μαζί με τις πληροφορίες εμφανίζονται και τα κουμπιά Select Base Term και Select Facet. Ο χρήστης επιλέγει το κουμπί Select Base Term και τα πεδία Ingredient Base Term και Ingredient Base Term Name της φόρμας θα συμπληρωθούν αυτόματα, όπως φαίνεται παρακάτω.

A screenshot of a computer

Description automatically generated

3ο βήμα:

Αφού επιλέχτηκε το Generic-base-term, στην συνέχεια ο χρήστης καλείται να συμπληρώσει τα υπόλοιπα στοιχεία της φόρμας, όπως φαίνεται στην επόμενη εικόνα.

A screenshot of a computer

Description automatically generated

Τέλος, πατώντας το κουμπί Add Ingredient το υλικό καταχωρείται επιτυχώς στη ΒΔ και ο server πληροφορεί τον χρήστη για την επιτυχή καταχώρηση του υλικού, όπως φαίνεται στην επόμενη εικόνα.

A screenshot of a computer

Description automatically generated

Στο προσωπικό inventory του χρήστη, έχει προστεθεί επιτυχώς η καταχώρηση, όπως δείχνει η επόμενη εικόνα.

A screenshot of a recipe

Description automatically generated

Παρατήρηση:

Για την εισαγωγή του συγκεκριμένου υλικού δεν χρειάστηκε να χρησιμοποιηθούν άλλα facets.

Αντιμετώπιση διπλότυπων

Όλα τα είδη ελιάς που πρόκειται να εισαχθούν στο σύστημα θα έχουν τον ίδιο κωδικό. Η διάκρισή τους θα γίνεται βάσει του place\_of\_origin. Στην περίπτωση δηλαδή που ο χρήστης επιχειρήσει να εισάγει το υλικό “ελιά” που ως τόπο παραγωγής έχει την Καλαμάτα, το σύστημα δεν θα επιτρέψει να γίνει η καταχώρηση. Αντί για αυτό, το σύστημα θα ενημερώσει τον χρήστη ότι μπορεί να εισάγει μία επιπλέον πληροφορία (ένα νέο description δηλαδή) εμπλουτίζοντας με αυτό τον τρόπο τις πληροφορίες που αφορούν στο εν λόγω υλικό.

Έτσι λοιπόν, έστω ότι ο χρήστης επιχειρεί να καταχωρήσει το ίδιο υλικό, όπως φαίνεται στην εικόνα.

A screenshot of a computer

Description automatically generated

Το σύστημα ειδοποιεί τον χρήστη ότι το υλικό υπάρχει ήδη και προτείνει στον χρήστη να προσθέσει μία νέα πληροφορία.

A screenshot of a computer

Description automatically generated

Πατώντας το ΟΚ, ο χρήστης μπορεί να συμπληρώσει κάποια νέα πληροφορία στο πλαίσιο που εμφανίζεται.

A screenshot of a computer

Description automatically generated

Στην συνέχεια ο χρήστης μπορεί να πατήσει το κουμπί Add Description και η νέα πληροφορία καταχωρείται επιτυχώς.

A screenshot of a computer

Description automatically generated

Το αποτέλεσμα της νέας πληροφορίας που προστέθηκε και που αφορά στο ήδη υπάρχον υλικό, αποτυπώνεται στην παρακάτω εικόνα.

A screenshot of a recipe

Description automatically generated

Στην περίπτωση που ο χρήστης επιχειρήσει να καταχωρήσει μια “ελιά” με διαφορετικό origin τότε το σύστημα θα ολοκληρώσει επιτυχώς την ενέργεια αυτή, όπως διαπιστώνεται στην επόμενη δοκιμή.

A screenshot of a computer

Description automatically generated

Το σύστημα επιτρέπει την καταχώρηση της ελιάς με τόπο παραγωγής τα Χανιά.

A screenshot of a computer

Description automatically generated

Και το inventory πλέον περιέχει τόσο την ελιά Καλαμών όσο και την ελιά Χανίων, όπως φαίνεται παρακάτω.

A screenshot of a recipe

Description automatically generated

Παρατήρηση:

Όταν ο χρήστης σε δεύτερο χρόνο επιχειρήσει να καταχωρήσει μια νέα συνταγή η οποία περιέχει το υλικό ελιά, τότε μέσω της αναζήτησης του inventory που θα παρουσιαστεί και στην συνέχεια, μπορεί να δώσει τον όρο “ελιά” προς αναζήτηση, οπότε στα αποτελέσματα θα εμφανιστούν όλα τα υλικά που περιέχουν την λέξη “ελιά”, όπως φαίνεται στην επόμενη εικόνα.

A screenshot of a computer

Description automatically generated

Ο χρήστης θα μπορεί να επιλέξει ως facet επεκτείνοντας τον κωδικό της νέας συνταγής, όποιο είδος ελιάς επιθυμεί βάσει του origin.

**Κεφ.10.2.1.2: Περίπτωση φάβας**

Στην περίπτωση της φάβας όπως στην περίπτωση της ελιάς που παρουσιάστηκε προηγουμένως, δεν χρειάζονται facets για τον σχηματισμό του κωδικού. Ωστόσο, η ιδιαιτερότητα που έχει το υλικό αυτό έγκειται στο γεγονός ότι υπάρχουν πολλές διαφορετικές ονομασίες - ποικιλίες ανά την Ελλάδα, που ουσιαστικά αφορούν στο ίδιο υλικό. Έτσι λοιπόν, η φάβα μπορεί να εμφανιστεί με το όνομα κουκί, λαθούρι, μπιζέλι κ.α. Κάθε μία από τις ονομασίες έχει όμως διαφορετικό κωδικό EFSA. Κατά συνέπεια, ο χρήστης θα πρέπει να είναι αρκετά προσεκτικός κατά τις καταχωρήσεις ειδών φάβας όπου θα πρέπει να μεριμνήσει να δώσει πέραν του βασικού ονόματος του υλικού π.χ κουκί και την πληροφορία ότι πρόκειται για την φάβα, στο πεδίο Greek alter name ή στο description. Με αυτό τον τρόπο η ΒΔ είναι σωστά ενημερωμένη οπότε όταν ο χρήστης σε δεύτερο χρόνο θα δοκιμάσει να καταχωρήσει μια νέα συνταγή που περιέχει το υλικό φάβα, κατά την αναζήτηση με τον όρο φάβα, θα επιστραφούν όλα τα αποτελέσματα των διαφορετικών ονομασιών που αναφέρθηκαν παραπάνω. Η εφαρμογή έχει υλοποιηθεί με τέτοιο τρόπο, που στις αναζητήσεις ψάχνει σε όλη τη βάση για τον όρο φάβα, οπότε ο χρήστης έχει μία πλήρη εικόνα των υλικών που θα επιλέξει ως facet για την συνταγή που θέλει να καταχωρήσει.

Στη ΒΔ υπάρχουν χάριν παραδείγματος το λαθούρι και το μπιζέλι όπως φαίνεται στην επόμενη εικόνα. Τα υλικά καταχωρήθηκαν ακριβώς με την ίδια διαδικασία που παρουσιάστηκε στην υποενότητα Περίπτωση ελιάς.

A screenshot of a recipe

Description automatically generated

Σε μία αναζήτηση με τον όρο φάβα κατά την καταχώρηση μιας νέας συνταγής που χρησιμοποιεί το υλικό φάβα, στα αποτελέσματα θα περιέχονται και οι δύο εγγραφές, οπότε ο χρήστης θα μπορεί να επιλέξει όποια καλύπτει τις ανάγκες του.

A screenshot of a computer

Description automatically generated

Παρατήρηση:

Σχετικά με τα διπλότυπα, η αντιμετώπιση του συστήματος στην περίπτωση της φάβας είναι ακριβώς η ίδια με την περίπτωση της ελιάς. Δεν θα επιτραπεί η καταχώρηση ενός υλικού εν γένει με το ίδιο Foodex code και το ίδιο origin. Σε αυτή την περίπτωση προστίθεται στο υπάρχον υλικό, ένα extra note ή description από τον χρήστη και με αυτό τον τρόπο ολοκληρώνεται η διαδικασία καταχώρησης.

**Κεφ.10.2.1.3: Περίπτωση μοσχαρίσιου κιμά**

Στην περίπτωση του μοσχαρίσιου κιμά ο κωδικός EFSA που θα σχηματιστεί αποτελείται από ένα σύνολο facets και αυτό γιατί δεν υπάρχει ήδη στον κατάλογο κάποιο Generic-base-term που να μπορεί να αποδώσει με ακρίβεια την περιγραφή του μοσχαρίσιου κιμά. Εδώ, το Generic-base-term αφορά σε πρωτογενές υλικό, το μοσχαρίσιο κρέας. Στην δοκιμή που ακολουθεί παρουσιάζονται τα διακριτά βήματα που ακολουθούνται για τον σχηματισμό του κωδικού.

Ο χρήστης μεταφέρεται στην σελίδα Add Ingerdient Page προκειμένου να εισάγει ένα νέο υλικό με όνομα “μοσχαρίσιος κιμάς”.

1ο βήμα:

Στο πεδίο Ingredient Name της φόρμας, ο χρήστης συμπληρώνει το όνομα “Μοσχαρίσιος κιμάς”.

2ο βήμα:

Ο χρήστης με τη βοήθεια του δέντρου αναζητά το Generic-base-term σε έναν από τους κόμβους του.

A screenshot of a computer screen

Description automatically generated

Επιλέγει τον κόμβο και στην συνέχεια πατάει το κουμπί Select Base Term,

A screenshot of a computer

Description automatically generated

και τα πεδία Ingredient Base Term και Ingredient Base Term Name της φόρμας θα συμπληρωθούν αυτόματα, όπως φαίνεται παρακάτω.

A screenshot of a computer

Description automatically generated

3ο βήμα:

Αφού επιλέχτηκε το Generic-base-term, στην συνέχεια ο χρήστης καλείται να συμπληρώσει τα υπόλοιπα στοιχεία της φόρμας, όπως φαίνεται στην επόμενη εικόνα.

A screenshot of a computer

Description automatically generated

4ο βήμα:

Σε αυτό το σημείο ο χρήστης πρέπει να επιλέξει τα facets. Οπότε, πατώντας το κουμπί Add Another Facet θα ανοίξουν δυναμικά δύο νέα πεδία (Facet Name, Facet Code) τα οποία συνοδεύονται με ένα νέο κουμπί με όνομα Remove.

A screenshot of a computer

Description automatically generated

Τα πεδία αυτά συμπληρώνονται αυτόματα. Αρκεί ο χρήστης να περιηγηθεί στο δέντρο μέχρις ότου βρει το κατάλληλο facet ψάχνοντας στην κατηγορία Process. Στην περίπτωση του μοσχαρίσιου κιμά το facet που μπορεί να χρησιμοποιηθεί από την κατηγορία Process είναι το Mincing/ chopping / cutting καθώς το συγκεκριμένο περιγράφει με σαφήνεια την επεξεργασία (τον τεμαχισμό πιο συγκεκριμένα) του πρωτογενούς υλικού “μοσχαρίσιο κρέας” που χρησιμοποιείται ως Generic-base-term.

A screenshot of a computer

Description automatically generated

Ο χρήστης επιλέγει τον κόμβο και αυτή την φορά πατάει το κουμπί Add Facet, όπως φαίνεται παρακάτω.

A screenshot of a computer

Description automatically generated

Τα πεδία Facet Name και Facet Code συμπληρώνονται αυτόματα.

A screenshot of a computer

Description automatically generated

Στην ίδια λογική ο χρήστης μπορεί να προσθέσει και άλλα facets η προσθήκη των οποίων θα δώσει ακόμα μία πιο εμπεριστατωμένη περιγραφή του υλικού. Αυτό μπορεί ο χρήστης να το πετύχει επαναλαμβάνοντας τη διαδικασία του 4ου βήματος όσες φορές θέλει.

A screenshot of a computer

Description automatically generated

Επιπλέον, το κουμπί Remove διαγράφει το αντίστοιχο facet. Στο συγκεκριμένο παράδειγμα θα διαγραφεί το facet Soaking πατώντας το κουμπί Remove και στο επόμενο βήμα θα γίνει η αποθήκευση του.

Πριν να γίνει η αποθήκευση όμως, δεν θα πρέπει ο χρήστης να ξεχάσει να προσθέσει άλλο ένα facet, το **Other** (F26.A07XE) ακολουθώντας με αυτό τον τρόπο τις οδηγίες σχετικά με το format που πρέπει να ακολουθείται, όπως τονίστηκε στην πρώτη ενότητα του κεφαλαίου.

A screenshot of a computer

Description automatically generated

5ο βήμα:

Ο χρήστης πατάει το κουμπί Add Ingredient και το υλικό μοσχαρίσιος κιμάς αποθηκεύεται επιτυχώς στην ΒΔ.

A screenshot of a computer

Description automatically generated

Ελέγχοντας το inventory επιβεβαιώνεται ότι το υλικό έχει καταχωρηθεί επιτυχώς.

A screenshot of a recipe

Description automatically generated

Αξίζει να παρατηρηθεί το format του κωδικού του νέου υλικού που μόλις προστέθηκε στη ΒΔ, που φαίνεται highlighted στην παραπάνω εικόνα.

Τέλος, να αναφερθεί ότι το κουμπί Clear όταν πατηθεί καθαρίζει όλη την φόρμα και ο χρήστης καλείται να επαναλάβει τη διαδικασία καταχώρησης του υλικού από την αρχή. Κατά τη διάρκεια των δοκιμών το κουμπί Clear φαίνεται να λειτουργεί σωστά.

**Κεφ.10.2.1.4: Περίπτωση καυκαλήθρες**

Για τον σχηματισμό του κωδικού των καυκαληθρών, ακολουθείται ακριβώς η ίδια μεθοδολογία που ακολουθήθηκε και στην περίπτωση του μοσχαρίσιου κιμά, με τη διαφορά ότι δεν υπάρχει κάποια πηγή (πρωτογενές υλικό) που να μπορεί να χρησιμοποιηθεί ως facet συμπληρωματικά με το Generic-base-term. Στην περίπτωση αυτή το Source-facet δηλώνεται περιγραφικά στο πεδίο description.

Στις δύο επόμενες εικόνες φαίνονται η καταχώρηση των καυκαληθρών και η ανανέωση του inventory του χρήστη με τα ingredients όπως διαμορφώθηκε αμέσως μετά την εισαγωγή του εν λόγω υλικού.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Κεφ.10.3: Καταχώρηση νέας συνταγής**

Στην ενότητα αυτή παρουσιάζονται τα βήματα που ακολουθούνται για τον σχηματισμό του κωδικού μιας συνταγής κατά την καταχώρησή της στο σύστημα. Πιο συγκεκριμένα παρατίθεται ένα παράδειγμα που αφορά στην καταχώρηση της συνταγής παστίτσιο (πιάτο) και στην καταχώρηση του τυριού τσαλαφούτι (σύνθετο τρόφιμο).

Η μεθοδολογία που ακολουθείται για την καταχώρηση μιας συνταγής ως πιάτο είναι πολύ συγκεκριμένη. Στην περίπτωση του πιάτου παστίτσιο, η αναζήτηση του Generic-base-term γίνεται σε μία συγκεκριμένη κατηγορία του καταλόγου της EFSA, την Composite dishes (A03VA). Έπειτα επιλέγεται το facet από την κατηγορία process που δείχνει τον τρόπο μαγειρέματος ενώ στην συνέχεια μπορούν να προστεθούν και άλλα facets καθένα από τα οποία αντιπροσωπεύει ένα υλικό που χρησιμοποιεί η συνταγή όπως για παράδειγμα είναι τα μακαρόνια, το τυρί κλπ. Για κάθε υλικό που χρησιμοποιείται όμως στην συνταγή, θα πρέπει να γίνεται μια περαιτέρω περιγραφή σχετικά με την επεξεργασία που υπέστη στην συγκεκριμένη συνταγή το υλικό που συμμετέχει σε αυτή. Σε αυτό το σημείο να σημειωθεί ότι η επιλογή των υλικών μπορεί να γίνει με δύο τρόπους. Είτε από το δέντρο, είτε από τη ΒΔ που περιέχει καταχωρημένα υλικά όπως τα υλικά που εισήχθησαν στα παραδείγματα της προηγούμενης ενότητας (μοσχαρίσιος κιμάς, ελιά Καλαμών, κουκιά κλπ.). Στην συνέχεια, και αφού τα υλικά έχουν επιλεχτεί, μπορούν να προστεθούν και άλλα facets που αφορούν σε πληροφορίες όπως το target group στο οποίο απευθύνεται η συνταγή για παράδειγμα.

Στην ίδια λογική κινείται και η διαδικασία που αφορά στην καταχώρηση ενός σύνθετου τροφίμου όπου στον σχηματισμό του κωδικού συμμετέχουν facets που αποτελούν άλλα υλικά και διαδικασίες επεξεργασίας.

**Κεφ.10.3.1: Περίπτωση παστίτσιο**

Ο χρήστης μεταφέρεται στην σελίδα Add Recipe Page προκειμένου να εισάγει μία νέα συνταγή με όνομα “Παστίτσιο με κιμά και διάφορα μυρωδικά από το Αιγαίο”. Ο χρήστης μπορεί να δώσει ένα αρκετά περιγραφικό όνομα για την συνταγή που πρόκειται να καταχώρησει.

1ο βήμα:

Στο πεδίο Recipe Name της φόρμας, ο χρήστης συμπληρώνει το όνομα “Παστίτσιο με κιμά και διάφορα μυρωδικά από το Αιγαίο”.

2ο βήμα:

Ο χρήστης αναζητά ένα Generic-base-term από την κατηγορία Composite dishes στο δέντρο. Στην περίπτωση του παστίτσιου επιλέγει, το Pasta, filled with meat mix, cooked (A040V) από το δέντρο και στην συνέχεια πατά το κουμπί Select Base Term.

A screenshot of a computer

Description automatically generated

3ο βήμα:

Τα πεδία Recipe Base Term και Recipe Base Term Name συμπληρώνονται αυτόματα και ο χρήστης συνεχίζει με την συμπλήρωση της φόρμας, όπως φαίνεται στην επόμενη εικόνα.

A screenshot of a computer

Description automatically generated

4ο βήμα:

Ο χρήστης στην συνέχεια προσθέτει τα facets στην συνταγή. Πατώντας το κουμπί Add Another Facet αυτόματα ανοίγουν κάποια πεδία, Facet Name, Facet Code, Amount και Foodex Final. Τα πεδία αυτά (πλην του Amount) συμπληρώνονται αυτόματα ακολουθώντας τη διαδικασία που περιγράφεται παρακάτω.

Αρχικά, ο χρήστης επιλέγει το facet που σχετίζεται με το process του τύπου μαγειρέματος. Στην περίπτωση του παστίτσιου είναι μαγείρεμα στον φούρνο οπότε ο χρήστης επιλέγει από το δέντρο το facet Baking και πατάει το κουμπί Select Facet.

A screenshot of a computer

Description automatically generated

Ο χρήστης συνεχίζει με την προσθήκη των υλικών. Οπότε πατάει ξανά το κουμπί Add Another Facet και πάλι αυτόματα ανοίγουν κάποια νέα πεδία. Μετά την επιλογή του process και το πάτημα του κουμπιού Add Another Facet, η οθόνη είναι όπως φαίνεται παρακάτω.

A screenshot of a computer

Description automatically generated

Η διαδικασία επιλογής των υλικών γίνεται με δύο τρόπους. Είτε με τη βοήθεια του δέντρου είτε με τη μηχανή αναζήτησης των ingredients που βρίσκεται στο δεξί μέρος της οθόνης.

A screenshot of a computer

Description automatically generated

Στο παράδειγμα θα επιλεγούν υλικά και με τους δύο τρόπους. Ο χρήστης λοιπόν, θέλει να επιλέξει τα μακαρόνια οπότε περιηγείται στο δέντρο, τα επιλέγει και στην συνέχεια πατάει Select Facet.

A screenshot of a computer

Description automatically generated

Τα μακαρόνια προστέθηκαν επιτυχώς στη λίστα με τα facets. Στην συνέχεια πρέπει να συμπληρωθεί το πεδίο Amount με την ποσότητα που αφορά στα μακαρόνια.

A screenshot of a computer

Description automatically generated

Ο χρήστης επαναλαμβάνει την ίδια διαδικασία, όμως αυτή τη φορά θα επιλέξει το επόμενο υλικό από τη μηχανή αναζήτησης και συγκεκριμένα τον μοσχαρίσιο κιμά. Οπότε πατάει το κουμπί Add Another Facet και εκ νέου ανοίγουν κάποια πεδία. Έπειτα στη μηχανή αναζήτησης εισάγει τη λέξη “Μοσχαρίσιος κιμάς” ή μπορεί να γράψει μέρος της λέξης που θέλει να αναζητήσει όπως για παράδειγμα “μοσχαρ”.

A screenshot of a computer

Description automatically generated

Εμφανίζονται τα αποτελέσματα της αναζήτησης και τότε ο χρήστης μπορεί να επιλέξει το υλικό που επιθυμεί πατώντας το κουμπί Select facet. Το υλικό προστίθεται στη λίστα των facets κάτω από τα προηγούμενα και ο χρήστης συμπληρώνει και εδώ το Amount (1 κιλό). Οποιαδήποτε στιγμή μπορεί να διαγράψει κάποιο facet από τα επιλεγμένα, πατώντας το κουμπί Remove, οπότε το facet αφαιρείται από τη λίστα.

Μπορούν να επιλεγούν και άλλα υλικά με τους δύο εναλλακτικούς τρόπους που παρουσιάστηκαν παραπάνω ακολουθώντας τα ίδια ακριβώς βήματα. Στο παράδειγμα ας θεωρηθεί ότι χρησιμοποιούνται αυτά τα δύο υλικά. Ωστόσο, για λόγους πληρότητας κρίνεται σκόπιμο να επιλεχθεί άλλο ένα facet που περιγράφει το σημείο παρασκευής του. Ακολουθώντας την ίδια λογική ο χρήστης επιλέγει από το δέντρο το facet Home prepared.

Στην οθόνη μέχρι αυτή την στιγμή θα πρέπει να εμφανίζονται τα εξής facets.

A screenshot of a computer program

Description automatically generated

5ο βήμα:

Στο βήμα αυτό ο χρήστης για κάθε ένα υλικό που περιλαμβάνεται στην λίστα του, μπορεί να προχωρήσει στην περιγραφή της επεξεργασίας που αφορά στο καθένα από αυτά τα υλικά εάν και εφόσον η επεξεργασία λαμβάνει χώρα. Ένα υλικό μπορεί να μην τυγχάνει περαιτέρω επεξεργασίας αλλά κάποιο άλλο υλικό όπως στα υλικά του παραδείγματος έχουν περάσει ένα στάδιο επεξεργασίας στην συγκεκριμένη συνταγή. Έτσι, τα μακαρόνια έχουν περάσει από την επεξεργασία του βρασμού πριν μπουν στον φούρνο και αντίστοιχα ο κιμάς έχει σωταριστεί στο τηγάνι. Αυτές τις δύο extra επεξεργασίες μπορεί ο χρήστης να περιγράψει ακολουθώντας την παρακάτω διαδικασία.

Πατώντας το κουμπί Go to process δίπλα από το υλικό Dried Pasta, μεταφέρεται σε μία νέα σελίδα, την Processed Ingredient Page, όπως φαίνεται στην εικόνα που ακολουθεί.

A screenshot of a computer

Description automatically generated

Τα πεδία Processed Base Term και Processed Base Term Name είναι προσυμπληρωμένα όπως φαίνεται παραπάνω. Ο χρήστης θέλει να περιγράψει τη διαδικασία του βρασμού (ουσιαστικά να δημιουργήσει ένα νέο processed ingredient) για το εν λόγω υλικό. Με την ίδια λογική που έχει παρουσιαστεί ως τώρα, ο χρήστης συμπληρώνει την φόρμα και επιλέγει από το δέντρο το facet Boiling.

A screenshot of a computer

Description automatically generated

Πατώντας το κουμπί Add Processed Ingredient καταχωρείται στη ΒΔ το επεξεργασμένο υλικό “Μακαρόνια βρασμένα” το οποίο μπορεί να χρησιμοποιηθεί σε μελλοντική συνταγή που περιέχει το ίδιο υλικό το οποίο έχει υποστεί την ίδια ακριβώς επεξεργασία, χωρίς να χρειάζεται κάθε φορά να δημιουργείται. Η αναζήτηση για τα “βρασμένα μακαρόνια” γίνεται με τον ίδιο τρόπο που εφαρμόστηκε στη μηχανή αναζήτησης για τα υλικά όπως παρουσιάστηκε πριν στην περίπτωση του “μοσχαρίσιου κιμά”. Υπάρχει δηλαδή και μία δεύτερη μηχανή αναζήτησης που αφορά στα επεξεργασμένα.

A screenshot of a computer

Description automatically generated

Με το πάτημα του κουμπιού OK μέσα στο μήνυμα που στέλνει ο server επιβεβαιώνοντας την επιτυχή προσθήκη του επεξεργασμένου υλικού, ο χρήστης μεταφέρεται πίσω στην σελίδα της καταχώρησης της συνταγής προκειμένου να συνεχίσει από το σημείο που είχε μείνει.

A screenshot of a computer

Description automatically generated

Το αποτέλεσμα της παραπάνω διαδικασίας είναι η προσθήκη μιας νέας γραμμής στη λίστα των Processed Ingredients με την πληροφορία “Βρασμένα μακαρόνια” και τον αντίστοιχο κωδικό του επεξεργασμένου υλικού. Ο χρήστης μπορεί αν θέλει να διαγράψει τη γραμμή πατώντας το κουμπί Χ. Στο παράδειγμα διατηρούμε τη γραμμή ως έχει.

Στην συνέχεια ο χρήστης ακολουθεί την ίδια ακριβώς διαδικασία και για το υλικό του μοσχαρίσιου κιμά όπως την εφάρμοσε προηγουμένως για τα μακαρόνια. Πατώντας λοιπον, το κουμπί Go to process που βρίσκεται δίπλα στο υλικό μοσχαρίσιος κιμάς, μεταφέρεται εκ νέου στην οθόνη Processed Ingredient Page.

A screenshot of a computer

Description automatically generated

Ο χρήστης προσθέτει τα facets που αφορούν στην επεξεργασία του εν λόγω υλικού και συμπληρώνει την φόρμα και τέλος πατάει το κουμπί Add Processed Ingredient ώστε να μεταφερθεί και πάλι στην προηγούμενη οθόνη. Η λίστα με τα facets και τα processed ingredients στην οθόνη του Recipe page,θα πρέπει να έχει διαμορφωθεί ως εξής.

A screenshot of a computer

Description automatically generated

6o βήμα:

Ο χρήστης πατάει το κουμπί Add Recipe ώστε να καταχωρηθεί η συνταγή.

A screenshot of a computer

Description automatically generated

O server ενημερώνει τον χρήστη για την επιτυχή προσθήκη της συνταγής στη ΒΔ.

Ο χρήστης από τη Home Page μπορεί να επιλέξει My inventory -> Show my recipes ώστε να επιβεβαιώσει ότι η συνταγή του έχει καταχωρηθεί κανονικώς. Πράγματι, όπως μαρτυρά η παρακάτω εικόνα, η συνταγή “αστίτσιο με κιμά και διάφορα μυρωδικά από το Αιγαίο” είναι πλέον στο inventory του.

A screenshot of a recipe

Description automatically generated

Τέλος, αξίζει να σημειωθεί ότι και σε αυτή την περίπτωση η διαχείριση για τα διπλότυπα εφαρμόζεται με την ίδια λογική όπως παρουσιάστηκε και στην καταχώρηση των υλικών.

**Κεφ.10.3.2: Περίπτωση τσαλαφούτι**

Για τα σύνθετα τρόφιμα όπως για παράδειγμα είναι τα τυριά, τα αλλαντικά και άλλα ελληνικά τρόφιμα καθώς και το τσαλαφούτι που εξετάζεται ως παράδειγμα, η διαδικασία καταχώρησης και κατά προέκταση ο σχηματισμός του κωδικού της Foodex, βασίζεται στην ίδια μεθοδολογία που ακολουθήθηκε σε προηγούμενες κατηγορίες τροφίμων. Μάλιστα, τα σύνθετα τρόφιμα θα μπορούσε να ειπωθεί ότι αποτελούν μία κατηγορία που συνδυάζουν τις μεθοδολογίες των άλλων κατηγοριών που εξετάστηκαν σε ξεχωριστές ενότητες. Πιο συγκεκριμένα συνδυάζει τις μεθοδολογίες που εφαρμόστηκαν στην περίπτωση του μοσχαρίσιου κιμά και στο παστίτσιο. Ως εκ τούτου, θα θεωρείτο πλεονασμός μία λεπτομερής αναφορά σχετικά με τα βήματα που ακολουθούνται για τη δημιουργία αυτής της συνταγής, μιας και οι προηγούμενες δύο περιπτώσεις καλύπτουν και με τα παραπάνω την περίπτωση τσαλαφούτι.

Στο κεφάλαιο παρουσιάστηκαν οι τρεις βασικές λειτουργίες της εφαρμογής, της καταχώρησης υλικού, της καταχώρησης συνταγής και της καταχώρησης επεξεργασμένου υλικού. Αναλύθηκαν σε διακριτά βήματα οι διαδικασίες που ο χρήστης θα πρέπει ακολουθήσει στο σύστημα προκειμένου να ολοκληρώσει τις λειτουργίες αυτές ενώ έγινε μία εκτενής ανάλυση σχετικά με τις προδιαγραφές που υποδεικνύουν τον τρόπο με τον οποίο σχηματίζονται οι αντίστοιχοι κωδικοί των τροφίμων.

γ