Flights App Project

Dezvoltat de Dinca Andrei-Gabriel

Github: github.com/Andreid27/FlightsApp

Bun venit la FlightsApp

Despre aplicație:

FlightsApp este o aplicație web în dedicată zborurilor și rezervării de zboruri. Aceasta vine cu un set complet de funcționalități precum căutarea unui zbor disponibil din baza de date, rezervarea unui zbor pentru 1 sau mai multe persoane. După deschiderea perioadei de Check-in utilizatorul *poate efectua check-in online* completând datele sale și ale pasagerilor pentru care a făcut rezervarea și poate **descărca un fișier pdf cu biletul** generat de API.

Aplicație a fost gandită, pregătită pentru modul de utilizare ca administrator sau moderator, calilitate care aduce o serie de facilități precum: adăugarea de zboruri, suprascrierea manuală a check-in-ului, modificarea unei rezervări, etc. Aceste implementări, precum și multe altele, vor veni în versiuni viitoare ale aplicației.

1.Baza de date

Tehnologia folosita pentru crearea bazei de date: **MySql**. – această alegere a fost făcută datorita numărului mare de relații între informațiile stocate în această bază de date.

„**MySql** este un sistem de gestiune a bazelor de date relaționale, produs de compania **MySQL AB**. Este cel mai popular sistem de gestionare de acest fel. Se găsește de obicei alături de limbajul **PHP**, însă cu **MySql** se pot construi aplicații in orice limbaj. ”

-https://ro.wikipedia.org/wiki/MySQL

Am realizat o baza de date in **MySQL WORKBENCH** cu motor InnoDB (InnoDB este un motor de stocare pentru baza de date MySQL. Acesta este folosit pentru a gestiona tabelele și indexurile în baza de date. InnoDB oferă suport pentru chei externe, care permit legarea între tabele prin intermediul unei chei comune. De asemenea, InnoDB oferă suport pentru tranzacții, ceea ce înseamnă că poți efectua mai multe operații asupra bazei de date într-un singur lot și poți anula întregul lot dacă se întâmplă ceva în timpul procesului.) , ea conținând 5 tabele principale și 2 tabele secundare(tabele de istoric - cărora li se face update odată la 24h de ore printr-o procedură PL/SQL) : Totul pleacă de la asocierea **users** cu **zboruri**. Asocierea dintre ele este de tipul M:N , ceea ce presupune crearea unei tabele intermediare, pe care am numit-o **rezervari**. La rândul lor tabelele zboruri și rezervari au asociate tabelele **aeronave**, respectiv **persoane_cin**(datele persoanelor care și-au făcut check-in, care se deschide de către API cu 24h înainte de zbor și are valorile ”closed” sau ”open”). Pe baza acestor persoane_cin se GENEREAZĂ apoi locul în avion și biletul complet.

Cele 2 tabele de istoric se numesc **istoric_zboruri** și **istoric_rezervari** în care rămân stocate doar informațiile esențiale(o parte din informații fiind șterse) și sunt actualizate odată la 24h, printr-o procedură PL/SQL cu numele *update_istoric_zboruri* și *update_istoric_rezervări*.

Scopul păstrării acestor date este pentru a avea informațiile necesare în cazul unei plângeri sau restituirii unor rezervări/zboruri anulate(păstrând și numărul tranzacției).Mai multe detalii și atribute ale fiecărei tabele le puteți găsi in digrama de mai jos în Fig. 1.

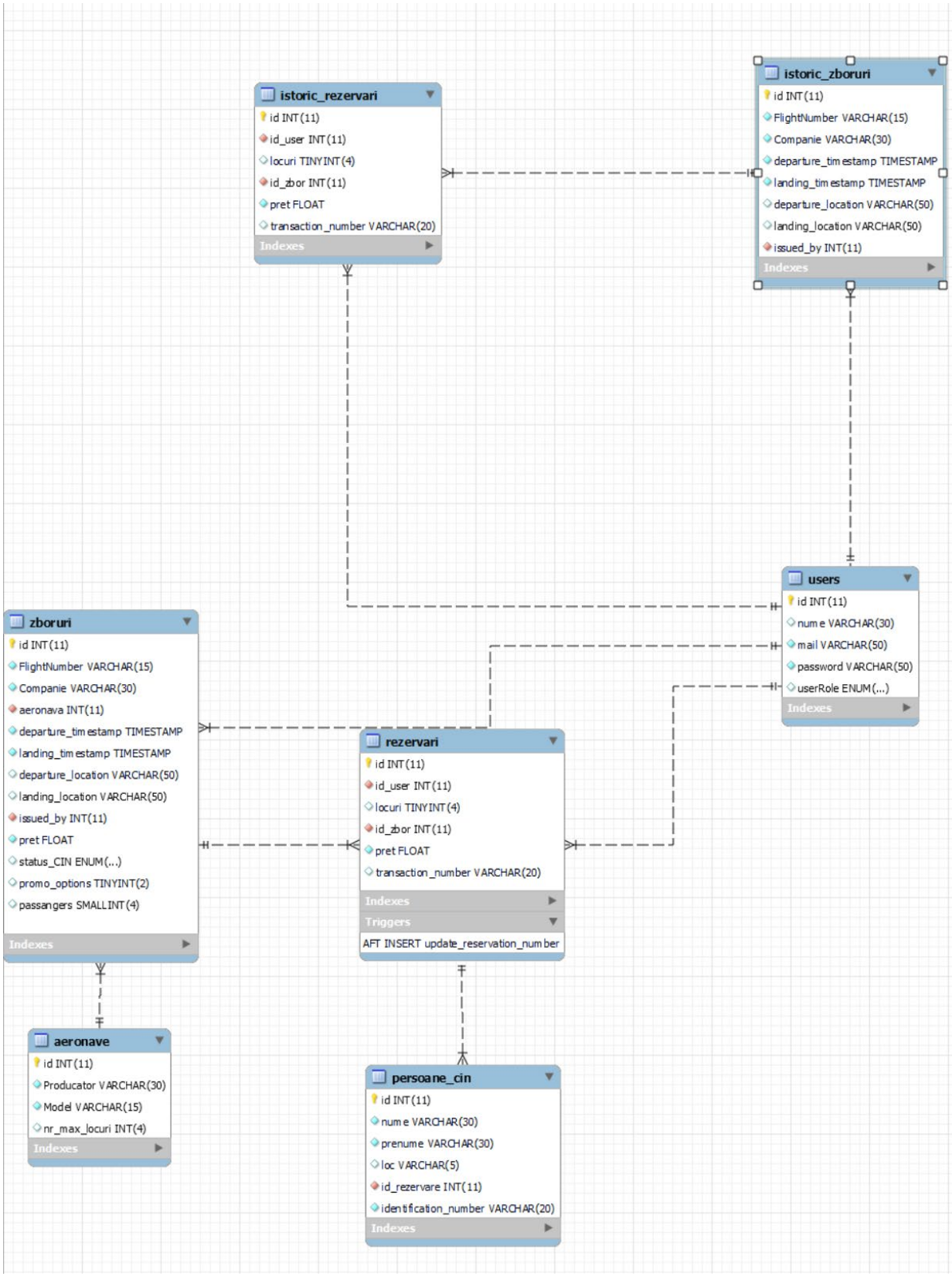


Fig. 1

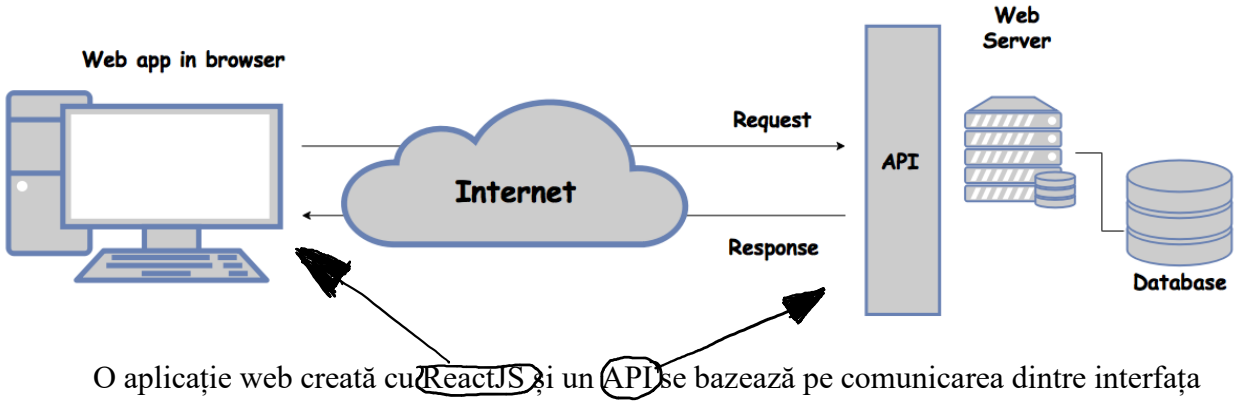
Tehnologii utilizate

Pentru interfața am folosi **Frontend** *(React, Redux-Toolkit(state manager tool), Package manager(npm, yarn), Material-UI)* și **Backend** *(Java API cu Maven* și *httpclient(Maven dependency)* pentru a crea un server Tomcat *) și cum am descris mai sus MySQL pentru baza de date)*. In acest mod putem edita si vizualiza datele pe care le comunicăm cu API-ul.

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

La proiectarea backend-ului s-a încercat păstrarea cât mai mult a codului Java și utilizarea **câtor mai puține** librării și framework-uri deoarece a fost un proiect dezvoltat pentru cursul de Java 1 Professional. Din acest motiv s-au folosit doar câteva pachete suplimentarea cum ar fi: Gson(pentru crearea și decodificarea JSON), XML bind & Openhtmltopdf(pentru generarea biletelor de îmbarcare în format PDF). Trebuie menționat de început că acesta a fost un proiect didactic, iar din acest motiv securitatea nu a fost unul din scopurile principale. Pentru ***securitate*** sporită vezi **RetetaTa.ro** unde am folosit Spring Security și JWT tokens.

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



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

1. Backend

API-ul RetetaTa.ro este construit după regulile arhitecturii Spring MVC(Model-View-Controller). Spring MVC o așa zisă arhitectură pentru Spring Boot care permite separarea logicii

aplicației, afișării datelor și controlului fluxului de date, într-un mod organizat și scalabil. Acesta oferă funcționalități precum gestionarea rutelor, cererilor și răspunsurilor, precum și suport pentru diferite formate de date. API-ul implementează logica din figura 2 de mai jos :

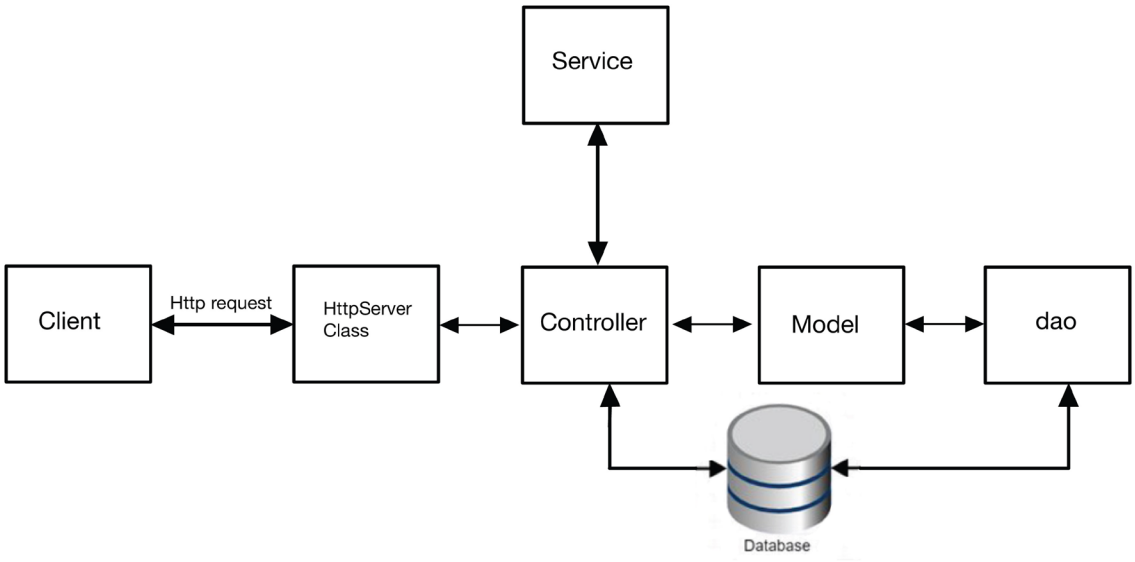


Fig. 2

Vom lua fiecare layer al MVC și îl vom analiza. Pe parcurs vom observa că fiecare din layere descrise în MVC se vor regăsi în pachete separate(iar logica nu este mult diferită de Spring MVC). Pentru ușurința explicațiilor ne vom raporta la Spring MVC(Fig. 3) pentru a explica logica API-ului nostru.

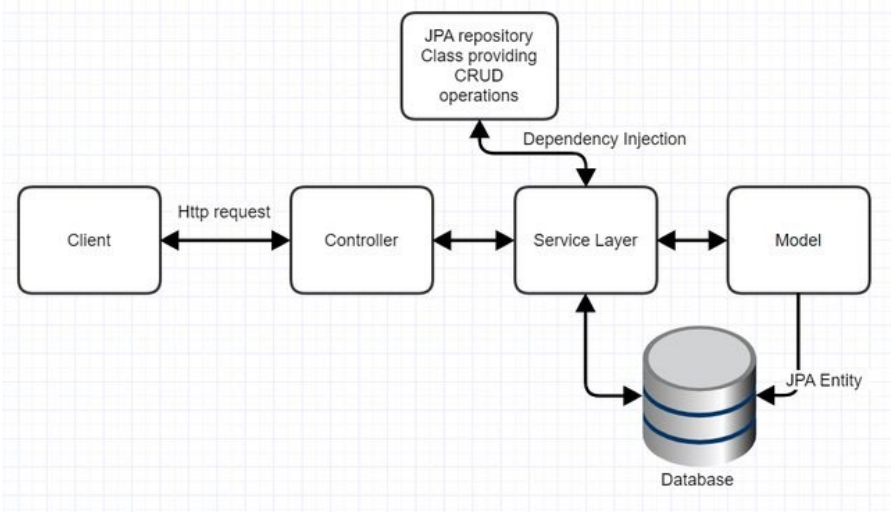
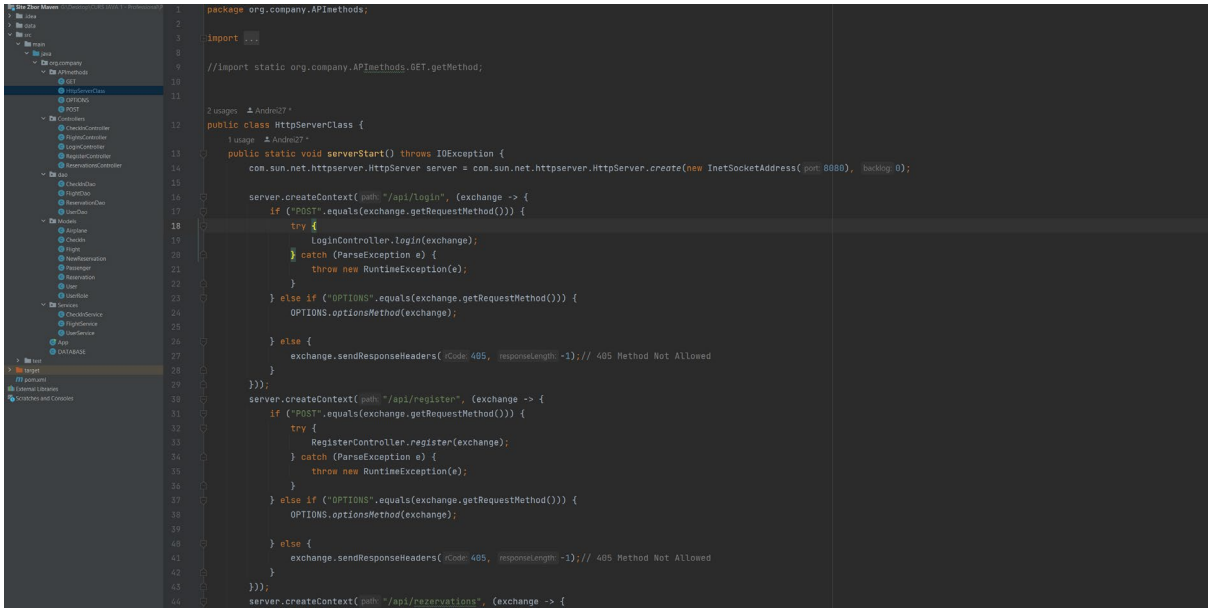


Fig. 3

Primul nostru layer se numește **HttpServerClass**. A fost preferată această metodă deși vine cu serie de dezavantaje datorită HttpServer-ului care este mai ușor de manipulat în acest context. Aici se regăsesc endpoint-urile accesibile de Frontend, iar odată primită cererea o avansează către următorul layer. Această clasă este similară unui **Controller** universal din Spring MVC. Controller-ul este responsabil pentru a determina care metodă din clasă trebuie

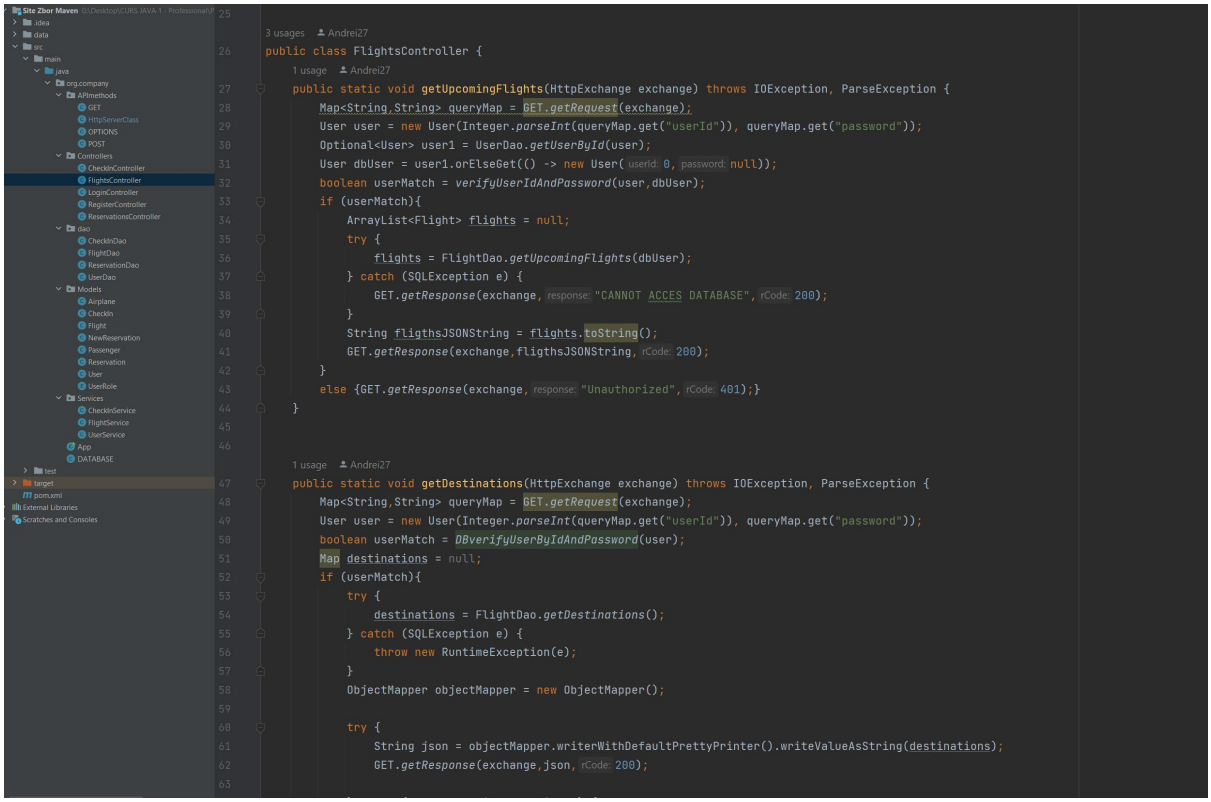
să răspundă la cererea respectivă, bazată pe adresa URL și metoda HTTP (GET, POST, etc.). Metoda selectată din controler este apoi apelată, care poate prelucra cererea(exemplu: la post creează din JSON o instanță a obiectului nostru(vom vedea la Model ce înseamnă) sau din URL extrage anumiți parametrii, etc.) și trimite către metoda din service layer care prelucrează cererea făcută acum de controller. Vezi Fig. 4.



```
package org.company.APImethods;

import ...

//import static org.company.APImethods.GET.getMethod;


2 usages  Andrei27 *
public class HttpServerClass {
    1 usage  Andrei27 *
    public static void serverStart() throws IOException {
        com.sun.net.httpserver.HttpServer server = com.sun.net.httpserver.HttpServer.create(new InetSocketAddress( port: 8080),  backlog: 0);

        server.createContext( path: "/api/login", (exchange -> {
            if ("POST".equals(exchange.getRequestMethod())) {
                try {
                    LoginController.login(exchange);
                } catch (ParseException e) {
                    throw new RuntimeException(e);
                }
            } else if ("OPTIONS".equals(exchange.getRequestMethod())) {
                OPTIONS.optionsMethod(exchange);

            } else {
                exchange.sendResponseHeaders( rCode: 405,  responseLength: -1);// 405 Method Not Allowed
            }
        }));
        server.createContext( path: "/api/register", (exchange -> {
            if ("POST".equals(exchange.getRequestMethod())) {
                try {
                    RegisterController.register(exchange);
                } catch (ParseException e) {
                    throw new RuntimeException(e);
                }
            } else if ("OPTIONS".equals(exchange.getRequestMethod())) {
                OPTIONS.optionsMethod(exchange);

            } else {
                exchange.sendResponseHeaders( rCode: 405,  responseLength: -1);// 405 Method Not Allowed
            }
        }));
        server.createContext( path: "/api/rezervations", (exchange -> {
```

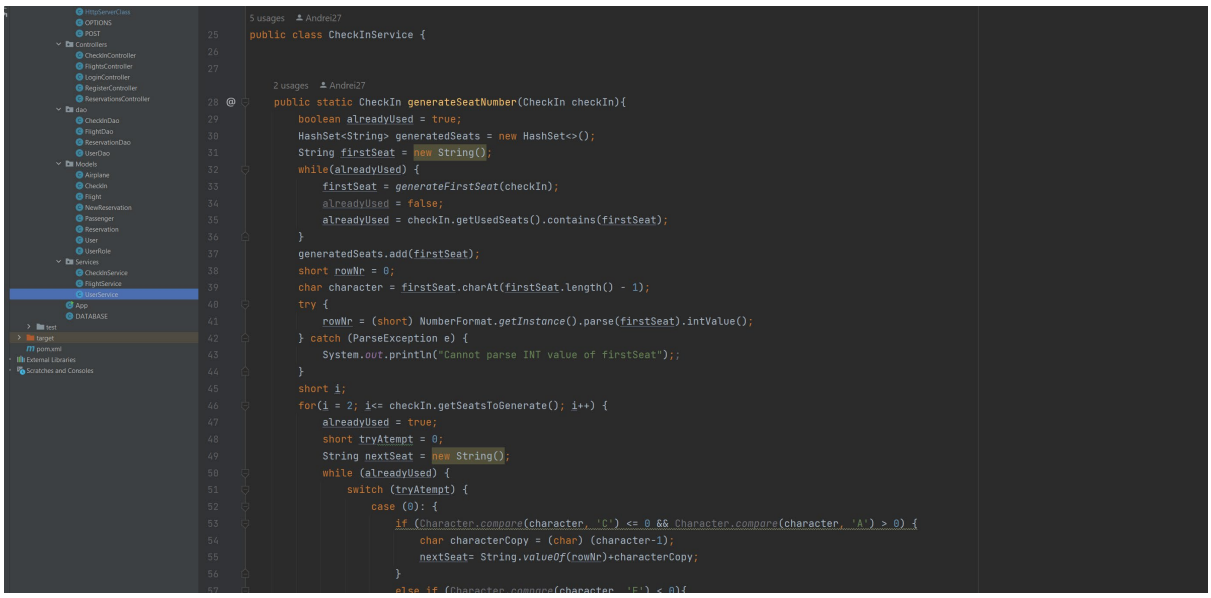
Fig. 4

Al doilea layer din API-ul nostru se numește **Controller**(Fig. 5), și este locul unde ajung request-urile pentru procesare, odată ce au fost sortate. Acest layer este foarte similar layer-ului **Service** din Spring MVC. Service Layer-ul din Spring MVC procesează logica aplicației, oferind metode specifice pentru operațiuni cum ar fi accesarea bazei de date, comunicarea cu alte sisteme sau orice altă logica necesara. Practic aici se face procesarea efectivă de care este responsabil API-ul nostru. În afară de procesarea efectivă a aplicație aceasta se ocupă și de apelarea părții de **dao** care se ocupă de manipularea datelor și comunicarea cu baza de date. Putem observa și un **Service** layer (Fig. 6) în această configurație. Acesta conține metode suplimentare care au fost extrase din **Controller**, deoarece am dorit ca acesta să conțină metodele clare și ușor de înțeles pentru funcționarea aplicației. Metodele de verificare a unor parametrii sau algoritmul de generare a locului din avion sa fie separat, iar acestea sa poate fi apelate din controller care să primească rezultatul final.



```
3 usages  Andrei27
public class FlightsController {
    1 usage  Andrei27
    public static void getUpcomingFlights(HttpExchange exchange) throws IOException, ParseException {
        Map<String,String> queryMap = GET.getRequest(exchange);
        User user = new User(Integer.parseInt(queryMap.get("userId")), queryMap.get("password"));
        Optional<User> user1 = UserDao.getUserById(user);
        User dbUser = user1.orElseGet(() -> new User( userId: 0, password: null));
        boolean userMatch = verifyUserIdAndPassword(user,dbUser);
        if (userMatch){
            ArrayList<Flight> flights = null;
            try {
                flights = FlightDao.getUpcomingFlights(dbUser);
            } catch (SQLException e) {
                GET.getResponse(exchange, response: "CANNOT ACCES DATABASE", rCode: 200);
            }
            String fligthsJSONString = flights.toString();
            GET.getResponse(exchange,fligthsJSONString, rCode: 200);
        }
        else {GET.getResponse(exchange, response: "Unauthorized", rCode: 401);}
    }


    1 usage  Andrei27
    public static void getDestinations(HttpExchange exchange) throws IOException, ParseException {
        Map<String,String> queryMap = GET.getRequest(exchange);
        User user = new User(Integer.parseInt(queryMap.get("userId")), queryMap.get("password"));
        boolean userMatch = DBverifyUserByIdAndPassword(user);
        Map destinations = null;
        if (userMatch){
            try {
                destinations = FlightDao.getDestinations();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
            ObjectMapper objectMapper = new ObjectMapper();

            try {
                String json = objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(destinations);
                GET.getResponse(exchange,json, rCode: 200);
```

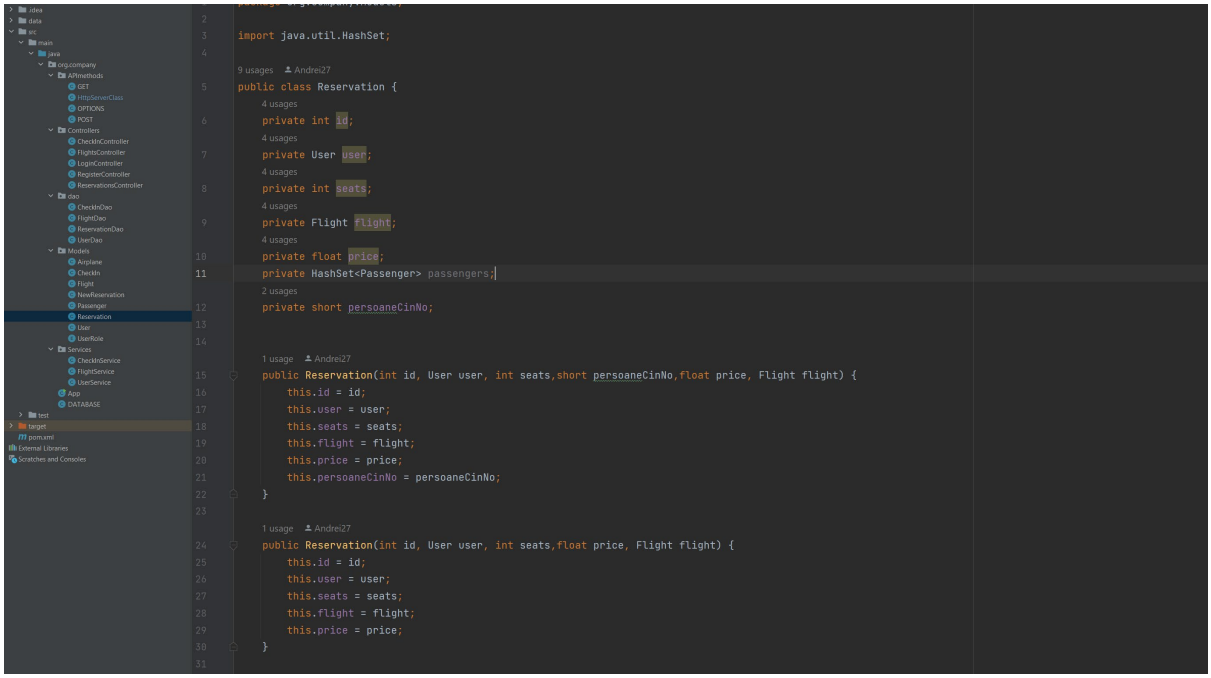
Fig. 5



```
5 usages  Andrei27
public class CheckInService {


    2 usages  Andrei27
    public static CheckIn generateSeatNumber(CheckIn checkIn){
        boolean alreadyUsed = true;
        HashSet<String> generatedSeats = new HashSet<>();
        String firstSeat = new String();
        while(alreadyUsed) {
            firstSeat = generateFirstSeat(checkIn);
            alreadyUsed = false;
            alreadyUsed = checkIn.getUsedSeats().contains(firstSeat);
        }
        generatedSeats.add(firstSeat);
        short rowNr = 0;
        char character = firstSeat.charAt(firstSeat.length() - 1);
        try {
            rowNr = (short) NumberFormat.getInstance().parse(firstSeat).intValue();
        } catch (ParseException e) {
            System.out.println("Cannot parse INT value of firstSeat");;
        }
        short i;
        for(i = 2; i<= checkIn.getSeatsToGenerate(); i++) {
            alreadyUsed = true;
            short tryAtempt = 0;
            String nextSeat = new String();
            while (alreadyUsed) {
                switch (tryAtempt) {
                    case (0): {
                        if (Character.compare(character, 'C') <= 0 && Character.compare(character, 'A') > 0) {
                            char characterCopy = (char) (character-1);
                            nextSeat= String.valueOf(rowNr)+characterCopy;
                        }
                        else if (Character.compare(character, 'F') < 0){
```

Fig. 6

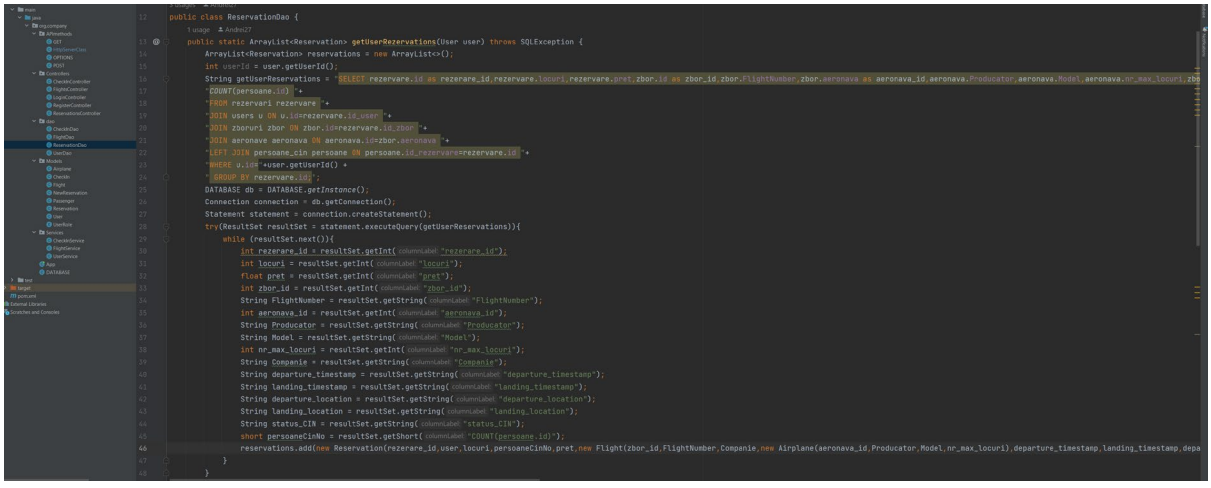
Al treilea layer este layer-ul **Model** unde similar Spring MVC sunt definite obiectele care sunt folosite în aplicație. Un exemplu putem găsi în figura 7.



```java
import java.util.HashSet;

9 usages  Andrei27
public class Reservation {
    4 usages
    private int id;
    4 usages
    private User user;
    4 usages
    private int seats;
    4 usages
    private Flight flight;
    4 usages
    private float price;
    private HashSet<Passenger> passengers;
    2 usages
    private short persoaneCinNo;


    1 usage  Andrei27
    public Reservation(int id, User user, int seats,short persoaneCinNo,float price, Flight flight) {
        this.id = id;
        this.user = user;
        this.seats = seats;
        this.flight = flight;
        this.price = price;
        this.persoaneCinNo = persoaneCinNo;
    }

    1 usage  Andrei27
    public Reservation(int id, User user, int seats,float price, Flight flight) {
        this.id = id;
        this.user = user;
        this.seats = seats;
        this.flight = flight;
        this.price = price;
    }
```

Fig. 7

Al patrulea layer este **dao** care este layer-ul unde se face comunicarea cu baza de date. Funcția acestui layer este similară **Repository** layer-ului din Spring MVC, dar implementarea este total diferită. În aceste layer fiecare metodă apelează baza de date trimițându-i un cod SQL (poate îndeplini oricare din funcțiile CRUD), iar aceasta întoarce sau nu, depinzând de caz, un ResultSet pe care îl putem parcurge și obține datele pe care le dorim din baza de date. Un exemplu putem vedea în figura 8.

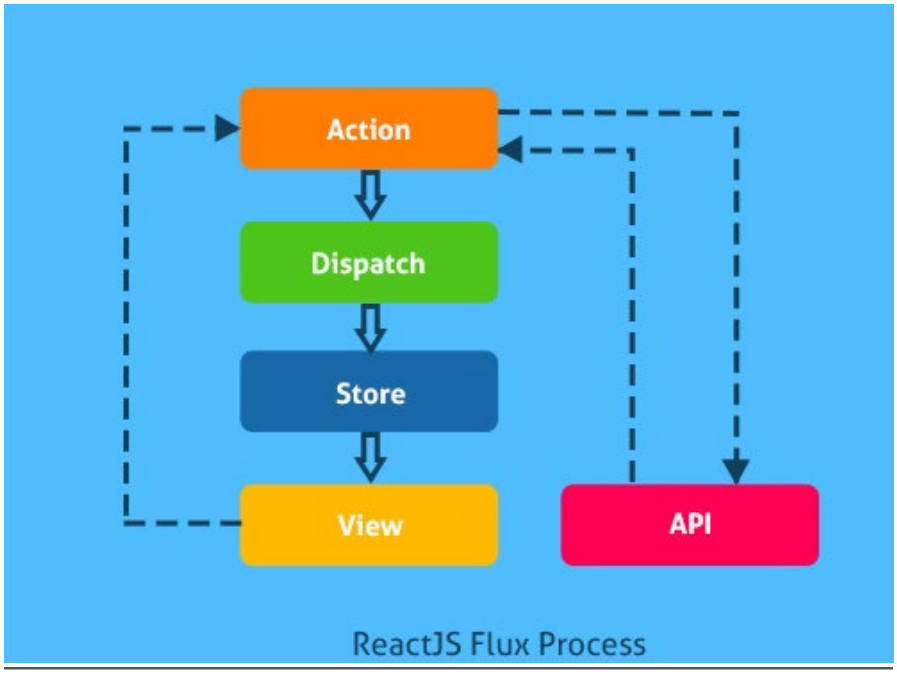


```java
public class ReservationDao {
    1 usage  Andrei27
    public static ArrayList<Reservation> getUserRezervations(User user) throws SQLException {
        ArrayList<Reservation> reservations = new ArrayList<>();
        int userId = user.getUserId();
        String getUserReservations = "SELECT rezervare.id as rezerare_id,rezervare.locuri,rezervare.pret,zbor.id as zbor_id,zbor.FlightNumber,zbor.aeronava as aeronava_id,aeronava.Producator,aeronava.Model,aeronava.nr_max_locuri,zbo
        "COUNT(persoane.id) "+
        "FROM rezervari rezervare "+
        "JOIN users u ON u.id=rezervare.id_user "+
        "JOIN zboruri zbor ON zbor.id=rezervare.id_zbor "+
        "JOIN aeronave aeronava ON aeronava.id=zbor.aeronava "+
        "LEFT JOIN persoane_cin persoane ON persoane.id_rezervare=rezervare.id "+
        "WHERE u.id ="+user.getUserId() +
        " GROUP BY rezervare.id;";
        DATABASE db = DATABASE.getInstance();
        Connection connection = db.getConnection();
        Statement statement = connection.createStatement();
        try(ResultSet resultSet = statement.executeQuery(getUserReservations)){
            while (resultSet.next()){
                int rezerare_id = resultSet.getInt( columnLabel: "rezerare_id");
                int locuri = resultSet.getInt( columnLabel: "locuri");
                float pret = resultSet.getInt( columnLabel: "pret");
                int zbor_id = resultSet.getInt( columnLabel: "zbor_id");
                String FlightNumber = resultSet.getString( columnLabel: "FlightNumber");
                int aeronava_id = resultSet.getInt( columnLabel: "aeronava_id");
                String Producator = resultSet.getString( columnLabel: "Producator");
                String Model = resultSet.getString( columnLabel: "Model");
                int nr_max_locuri = resultSet.getInt( columnLabel: "nr_max_locuri");
                String Companie = resultSet.getString( columnLabel: "Companie");
                String departure_timestamp = resultSet.getString( columnLabel: "departure_timestamp");
                String landing_timestamp = resultSet.getString( columnLabel: "landing_timestamp");
                String departure_location = resultSet.getString( columnLabel: "departure_location");
                String landing_location = resultSet.getString( columnLabel: "landing_location");
                String status_CIN = resultSet.getString( columnLabel: "status_CIN");
                short persoaneCinNo = resultSet.getShort( columnLabel: "COUNT(persoane.id)");
                reservations.add(new Reservation(rezerare_id,user,locuri,persoaneCinNo,pret,new Flight(zbor_id,FlightNumber,Companie,new Airplane(aeronava_id,Producator,Model,nr_max_locuri),departure_timestamp,landing_timestamp,depa
            }
        }
```

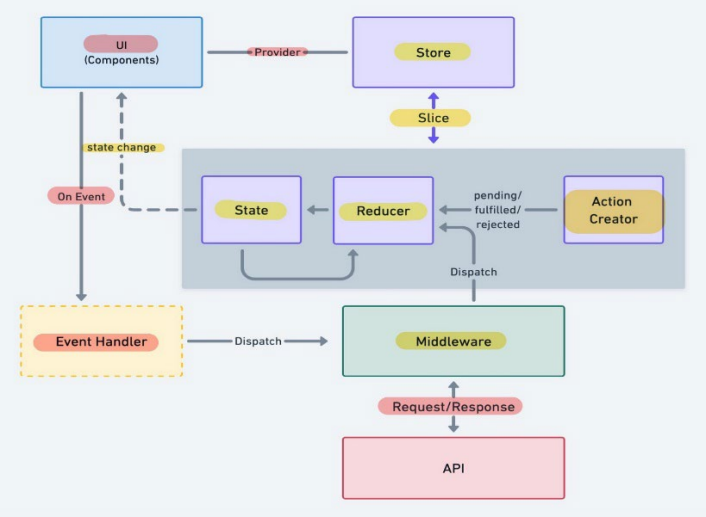
Fig. 8

2. Frontend

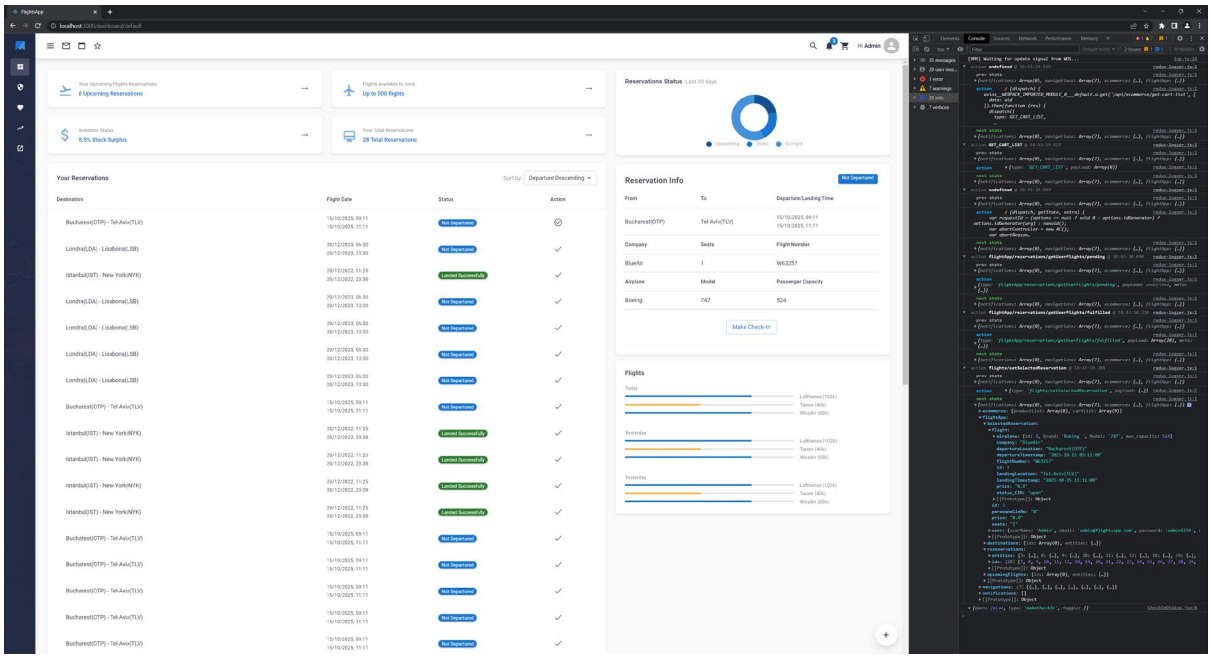
ReactJS este un framework JavaScript open-source pentru crearea de aplicații web cu interfață utilizator interactivă. Principiul de bază al acestui framework se bazează pe conceptul de componente, care permit dezvoltatorilor să construiască aplicații web mai eficient prin modularizarea și reutilizarea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



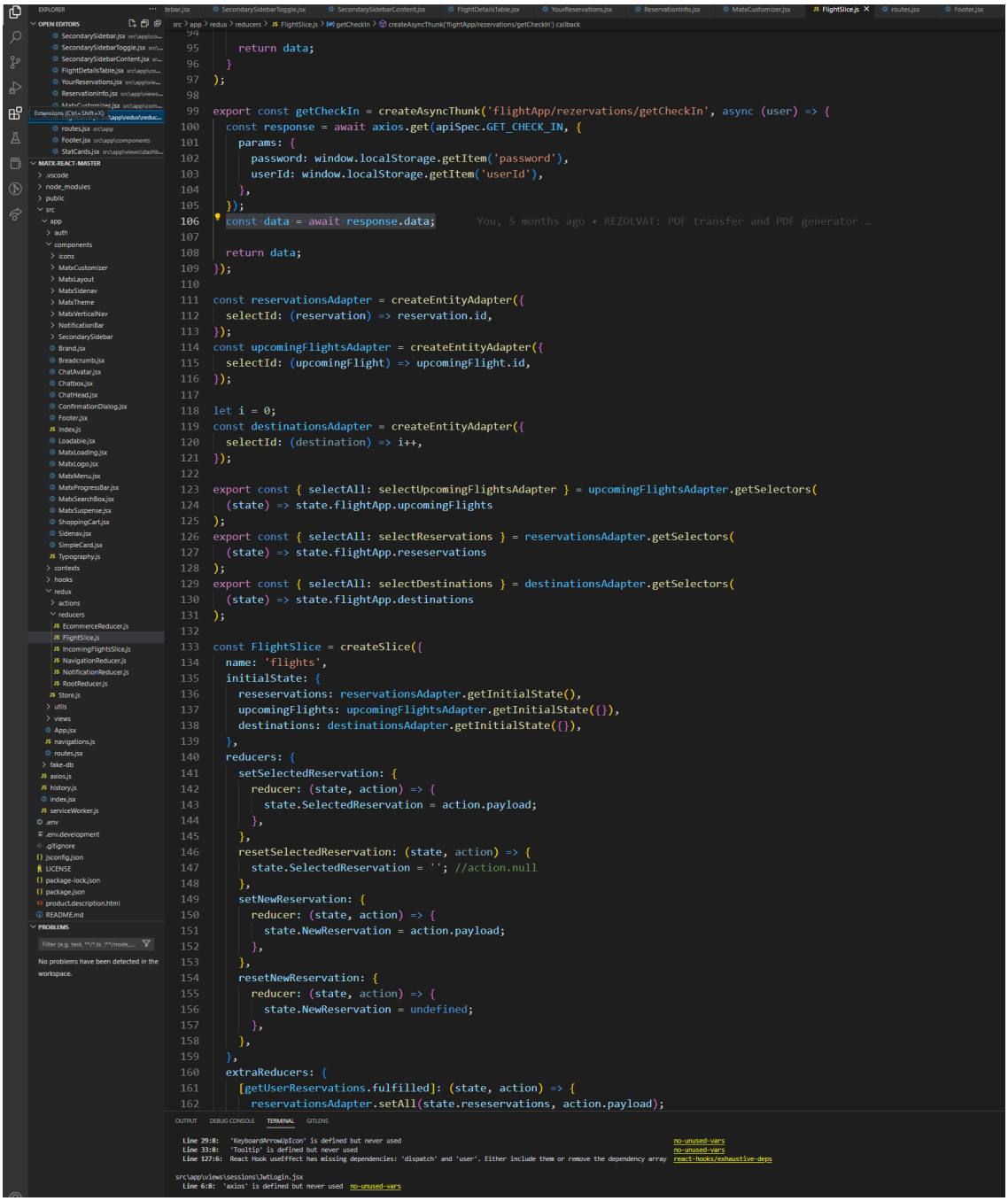
În aplicațiile React, termenul "action" se referă la acțiunile care descriu schimbările de date care au loc în aplicație. Aceste acțiuni sunt vizibile prin intermediul unor obiecte JavaScript simple, care conțin informații despre schimbările de date. Procesul de transmitere a acțiunilor către un "store"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



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



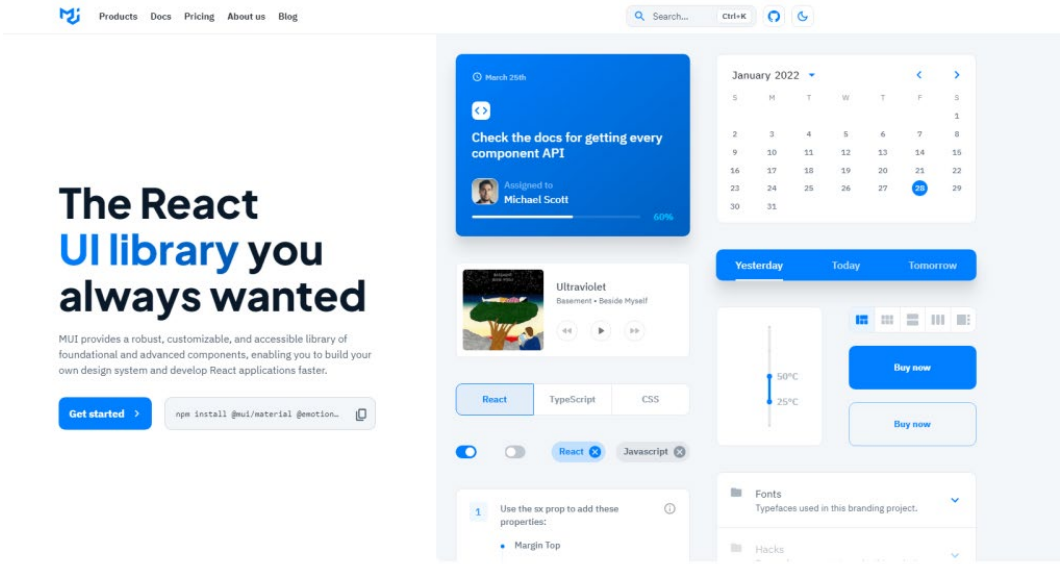
Fiecare stare conține toți parametri necesari pe care componentele React le pot accesa cu ajutorul unui metode in Redux Toolkit și anume useSelect. Cu acești selectori pot accesa starea curentă. Mai jos avem „slice”-ul unui ingredient. Un slice este o bucată constituentă a „store”-ului. Acest lucru ne-a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



```
export const getCheckIn = createAsyncThunk('flightApp/rezervations/getCheckIn', async (user) => {
  const response = await axios.get(apiSpec.GET_CHECK_IN, {
    params: {
      password: window.localStorage.getItem('password'),
      userId: window.localStorage.getItem('userId'),
    },
  });
  const data = await response.data;       You, 5 months ago • REZOLVAT: PDF transfer and PDF generator …

  return data;
});

const reservationsAdapter = createEntityAdapter({
  selectId: (reservation) => reservation.id,
});
const upcomingFlightsAdapter = createEntityAdapter({
  selectId: (upcomingFlight) => upcomingFlight.id,
});

let i = 0;
const destinationsAdapter = createEntityAdapter({
  selectId: (destination) => i++,
});

export const { selectAll: selectUpcomingFlightsAdapter } = upcomingFlightsAdapter.getSelectors(
  (state) => state.flightApp.upcomingFlights
);
export const { selectAll: selectReservations } = reservationsAdapter.getSelectors(
  (state) => state.flightApp.reseservations
);
export const { selectAll: selectDestinations } = destinationsAdapter.getSelectors(
  (state) => state.flightApp.destinations
);

const FlightSlice = createSlice({
  name: 'flights',
  initialState: {
    reseservations: reservationsAdapter.getInitialState(),
    upcomingFlights: upcomingFlightsAdapter.getInitialState({}),
    destinations: destinationsAdapter.getInitialState({}),
  },
  reducers: {
    setSelectedReservation: {
      reducer: (state, action) => {
        state.SelectedReservation = action.payload;
      },
    },
    resetSelectedReservation: (state, action) => {
      state.SelectedReservation = ''; //action.null
    },
    setNewReservation: {
      reducer: (state, action) => {
        state.NewReservation = action.payload;
      },
    },
    resetNewReservation: {
      reducer: (state, action) => {
        state.NewReservation = undefined;
      },
    },
  },
  extraReducers: {
    [getUserReservations.fulfilled]: (state, action) => {
      reservationsAdapter.setAll(state.reseservations, action.payload);
```

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

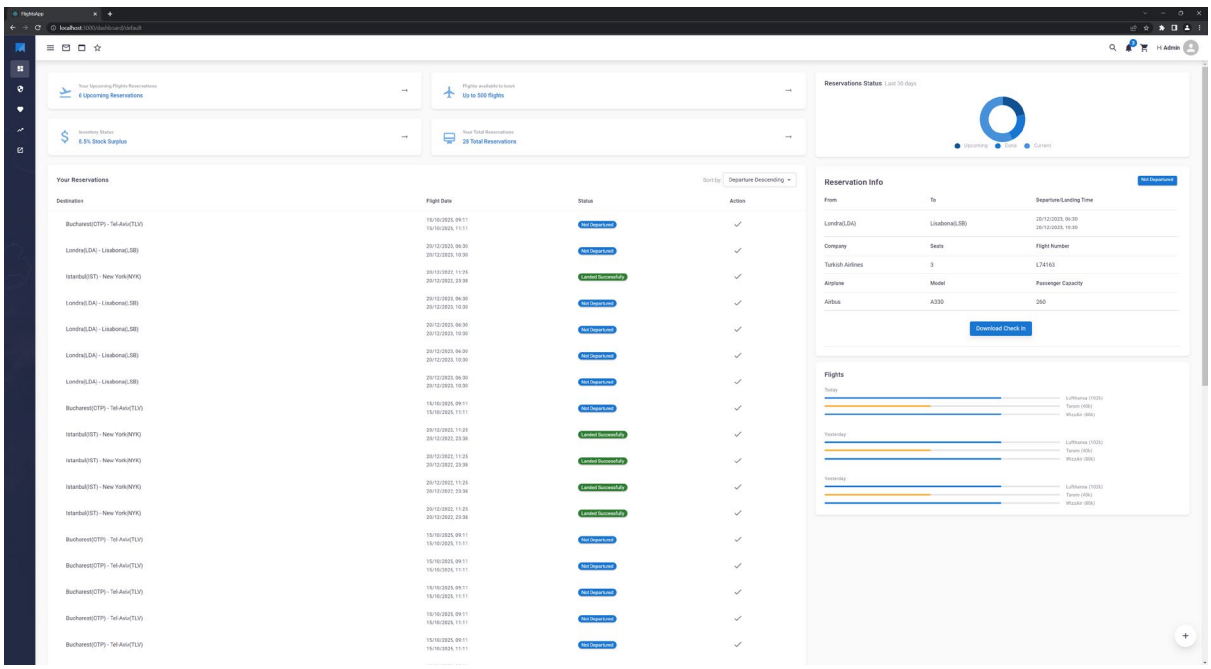
Elementele au fost majoritar folosite din Material-UI care ne permite sa grabim procesul de dezvoltare a front-end-ului.



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

Un tur complet al aplicație se găsește chiar aici.

(sau în repository-ul aplicației de Github)



Bibliografie:

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