Airwise Documentation

1. Description

The project began with a mix of personal travel experiences and a vision to make flying easier and more accessible. It was inspired by personal travel experiences during a summer in Italy, where challenges regarding air travel arose. Drawing inspiration from apps like Skyscanner and Kiwi, the goal was to make something similar reflecting what travelers need. The path to creating this app was full of turns, from early experiments with web scraping to figuring out how to get around outdated information. The breakthrough came with the integration of APIs, providing real-time flight data.

2. Objectives

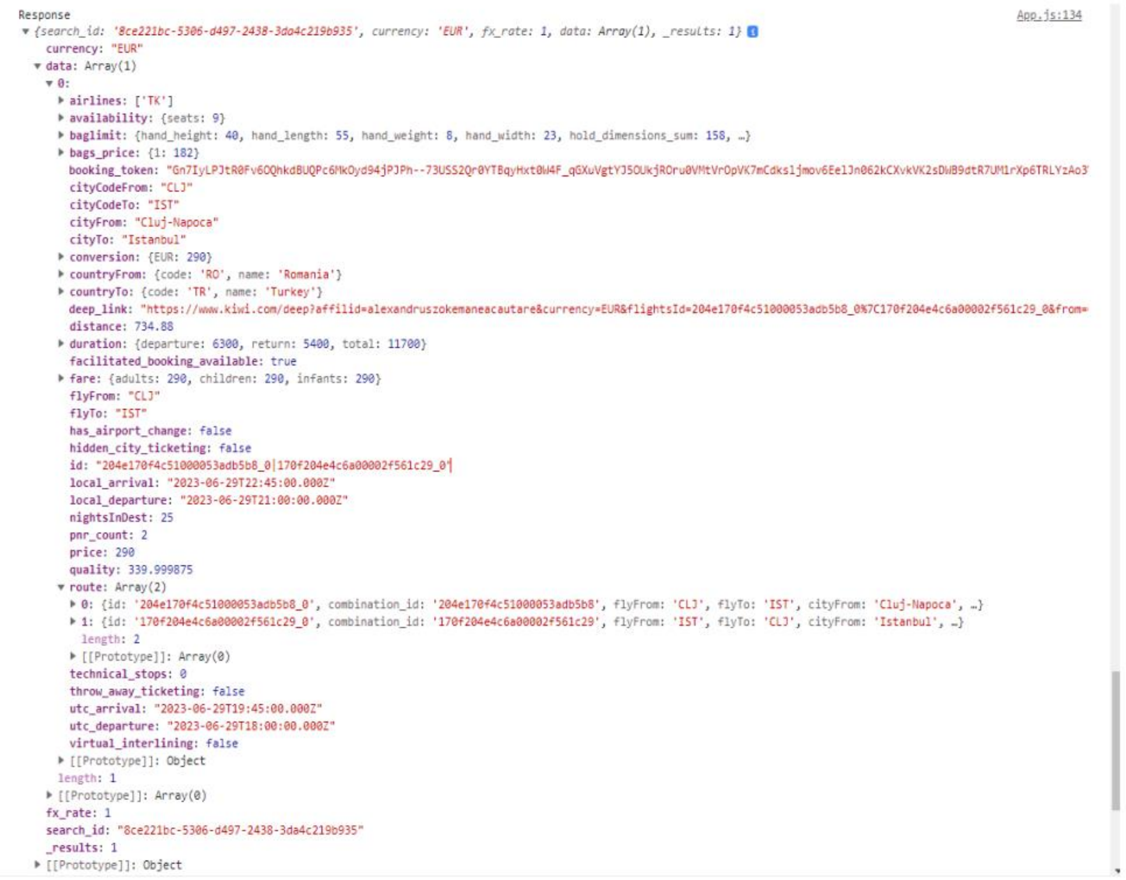
The goal of this project is to make the process of finding means of travel by air easier.
The main objectives are as follows:

- Improve User Experience
- Provide Real-time Flight Data
- Promote Economic Travel
- Incorporate Innovative Solutions
- Enable Flexibility

3. Technologies

To produce a good final result, the app included state-of-the-art web development technologies in the latest iteration of the project. With an emphasis on React.js, a front-end JavaScript library, the aim is to achieve seamless functionality throughout the framework. As a crucial component of the webpage presentation, HTML was selected along with CSS, which was deemed ideal for bringing style and visual appeal into play.

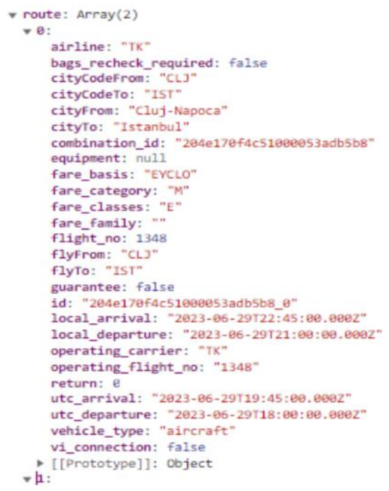
Approach - In response to numerous challenges encountered with previous approaches, the issue was addressed by implementing an API. The first step was to identify an API to fetch the flight data. After a comprehensive search, Kiwi's API located here https://tequila.kiwi.com/ was found. Kiwi is a well-known travel company known for its flight search engine, and they provide a freely accessible API from which the user can get information about flights. Kiwi's direct process made it easy to use their API. The initial step taken to use the API was to register an account on their developer platform called Tequila. Once the account was created a solution was generated. After the solution was created, Kiwi generated an API key, and this unique key was vital for the application to interact with the API. This enabled the fetching of real-time data about flight schedules, prices, and durations for integration into the application. The accessibility of Kiwi's API accelerated the development process and mitigated the need to build the flight search system from scratch. Because of the access to the flight information, a major part of the work was designing a suitable user interface. It was important to present this information in an organized, and user-friendly manner. The visual design of the application was also an important factor. A smooth user journey was prioritized, hence the effort to mirror Kiwi's color palette and aesthetic in the application. The goal was to make a seamless transition between this application and Kiwi’s website. It is essential that users experience a smooth transition when they move from this site to Kiwis' platform for their booking, aiming in this way to enhance the user experience. To provide an example of the vast information provided by the API, I would like to present next a typical response in the JSON format:



```
Response                                                                                                App.js:134
▾ {search_id: '8ce221bc-5306-d497-2438-3da4c219b935', currency: 'EUR', fx_rate: 1, data: Array(1), _results: 1}
    currency: "EUR"
  ▾ data: Array(1)
    ▾ 0:
      ▸ airlines: ['TK']
      ▸ availability: {seats: 9}
      ▸ baglimit: {hand_height: 40, hand_length: 55, hand_weight: 8, hand_width: 23, hold_dimensions_sum: 158, …}
      ▸ bags_price: {1: 182}
        booking_token: "Gn7IyLPJtR0Fv6OQhkdBUQPc6MkOyd94jPJPh--73USS2Qr0YTBqyHxt0W4F_qGXuVgtYJ5OUkjROru0VMtVrOpVK7mCdks1jmov6Ee1Jn062kCXvkVK2sDWB9dtR7UM1rXp6TRLYzAo3
        cityCodeFrom: "CLJ"
        cityCodeTo: "IST"
        cityFrom: "Cluj-Napoca"
        cityTo: "Istanbul"
      ▸ conversion: {EUR: 290}
      ▸ countryFrom: {code: 'RO', name: 'Romania'}
      ▸ countryTo: {code: 'TR', name: 'Turkey'}
        deep_link: "https://www.kiwi.com/deep?affilid=alexandruszokemaneacautare&currency=EUR&flightsId=204e170f4c51000053adb5b8_0%7C170f204e4c6a00002f561c29_0&from=
        distance: 734.88
      ▸ duration: {departure: 6300, return: 5400, total: 11700}
        facilitated_booking_available: true
      ▸ fare: {adults: 290, children: 290, infants: 290}
        flyFrom: "CLJ"
        flyTo: "IST"
        has_airport_change: false
        hidden_city_ticketing: false
        id: "204e170f4c51000053adb5b8_0|170f204e4c6a00002f561c29_0"
        local_arrival: "2023-06-29T22:45:00.000Z"
        local_departure: "2023-06-29T21:00:00.000Z"
        nightsInDest: 25
        pnr_count: 2
        price: 290
        quality: 339.999875
      ▾ route: Array(2)
        ▸ 0: {id: '204e170f4c51000053adb5b8_0', combination_id: '204e170f4c51000053adb5b8', flyFrom: 'CLJ', flyTo: 'IST', cityFrom: 'Cluj-Napoca', …}
        ▸ 1: {id: '170f204e4c6a00002f561c29_0', combination_id: '170f204e4c6a00002f561c29', flyFrom: 'IST', flyTo: 'CLJ', cityFrom: 'Istanbul', …}
          length: 2
        ▸ [[Prototype]]: Array(0)
        technical_stops: 0
        throw_away_ticketing: false
        utc_arrival: "2023-06-29T19:45:00.000Z"
        utc_departure: "2023-06-29T18:00:00.000Z"
        virtual_interlining: false
      ▸ [[Prototype]]: Object
      length: 1
    ▸ [[Prototype]]: Array(0)
    fx_rate: 1
    search_id: "8ce221bc-5306-d497-2438-3da4c219b935"
    _results: 1
  ▸ [[Prototype]]: Object
```

Figure 1 – API Response

Inside the route section one can find information regarding the flights, and layovers, as it is available bellow:



```
▾ route: Array(2)
  ▾ 0:
      airline: "TK"
      bags_recheck_required: false
      cityCodeFrom: "CLJ"
      cityCodeTo: "IST"
      cityFrom: "Cluj-Napoca"
      cityTo: "Istanbul"
      combination_id: "204e170f4c51000053adb5b8"
      equipment: null
      fare_basis: "EYCLO"
      fare_category: "M"
      fare_classes: "E"
      fare_family: ""
      flight_no: 1348
      flyFrom: "CLJ"
      flyTo: "IST"
      guarantee: false
      id: "204e170f4c51000053adb5b8_0"
      local_arrival: "2023-06-29T22:45:00.000Z"
      local_departure: "2023-06-29T21:00:00.000Z"
      operating_carrier: "TK"
      operating_flight_no: "1348"
      return: 0
      utc_arrival: "2023-06-29T19:45:00.000Z"
      utc_departure: "2023-06-29T18:00:00.000Z"
      vehicle_type: "aircraft"
      vi_connection: false
    ▸ [[Prototype]]: Object
  ▾ 1:
```

Figure 2 – Route Contents

Functionalities - Before focusing on the design and the actual coding process, let's have a look at some of the key functionalities inside this application:

• **Trip Types**: The application allows users to switch between "Return" and "One‐way". This appeals to different travel needs, whether the users plan a round trip or an open journey.

• **Passengers and Class**: This feature allows users to customize their traveling, specifying the number of adults, children, and infants. Also, travelers can choose their travel class by selecting it and is suitable to everybody, from budget-conscious customers to those looking for luxury.

• **Stopovers**: With this feature, users can customize their journey by specifying the maximum number of layovers. From choosing direct ”Non-stop” flights to limiting layovers, this functionality can alter the travel time and experience.
• **Times**: Through intuitive sliders, users can choose their preferred departure and arrival times, helping to match their travel plans with their personal schedules.
• **Currency**: This application also provides a global-friendly interface, supporting multiple currencies. Users can view prices in the currency they're most comfortable with reducing the need of converting by themselves.
• **Clear Filters Button**: To enhance usability, the ”Clear Filters” button allows users to reset all their search parameters instantly, simplifying repeated searches or major travel plan changes.
• **Search Bar**: This is the heart of the application where users input their departure and arrival airports, and select their travel dates. The system has been designed to handle flexible date ranges, incorporating last-minute changes or flexible travel plans. For one-way trips, a single date input is displayed, keeping the interface clean and intuitive.
• **Results Display and Purchase**: Each search result provides detailed information about the flight, including layovers, waiting times, travel dates, cabin bags weights and even the number of available seats (when provided by the airline). Each result includes a ”Book” button that redirects users to Kiwi's website to complete the booking process.
• **Sorting**: Users can sort flight results based on their preferences, choosing between the "Cheapest" and "Fastest" options. This sorting functionality helps users identify the most suitable flight options.
• **Booking Link**: A booking link that corresponds to the search parameters opens in a new window when users click the search button. This feature provides a perfect experience by allowing users to estimate the total price of their trips.
• **Responsive Design**: The application adapts to screen size changes, ensuring a consistent user experience across various devices by providing relevant messages to the users as they interact with the application.
• **Kiwi’s Widget**: Prior to initiating any search, users can view a widget displaying the cheapest flights from their location to popular cities. This can inspire spontaneous travel plans.
• **Loading Animation**: An engaging animation plays while the flight data is being fetched, providing a visually pleasing experience and keeping users informed about the application's progress.
The implementation of these functionalities offers an intuitive and versatile user experience making the process of searching for and booking flights really easy. Let’s now explore the design and coding aspects of this project.

4. Design and Code

The design and coding process of this application required careful planning and numerous iterations. While designing the application, the decision to use the two-tier architecture model was made. This choice was fitting for this Single Page Application (SPA) because it separates the client-side presentation from the server-side. In the case of this search, the React application represents the front-end client tier, while the Kiwi API serves as the data source or server tier. This distinction allows for better maintainability and scalability, as updates or modifications can be made to one tier without influencing the other.

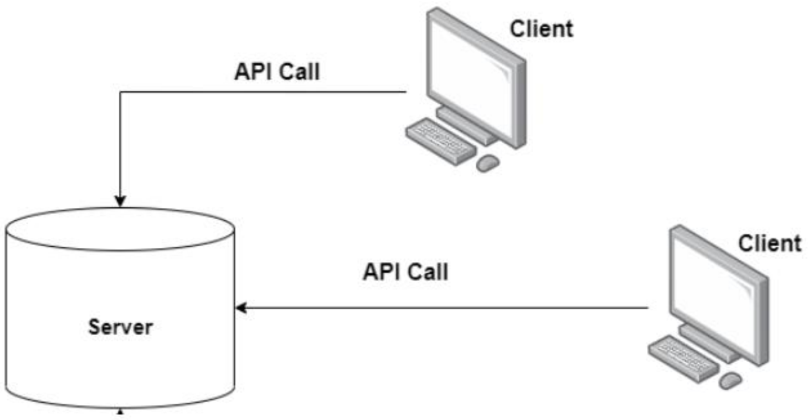
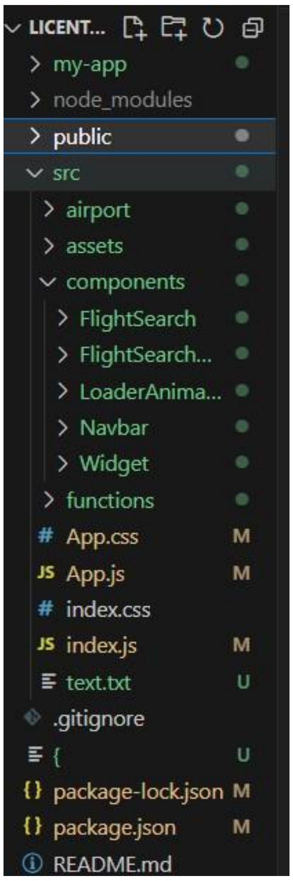


Figure 3 – Used Architecture

A lot of technologies were utilized during the development of this application. For example, React.js was used to create the User Interface components, while JavaScript eased the management of logic inside the application. Furthermore, using HTML and CSS helped in designing an organized structure with an appealing visual. After multiple iterations the project structure now looks like this:



```
LICENT...
  > my-app
  > node_modules
  > public
  v src
    > airport
    > assets
    v components
      > FlightSearch
      > FlightSearch...
      > LoaderAnima...
      > Navbar
      > Widget
    > functions
    # App.css            M
    JS App.js            M
    # index.css
    JS index.js          M
    text.txt             U
  .gitignore
  {                      U
  {} package-lock.json   M
  {} package.json        M
  README.md
```

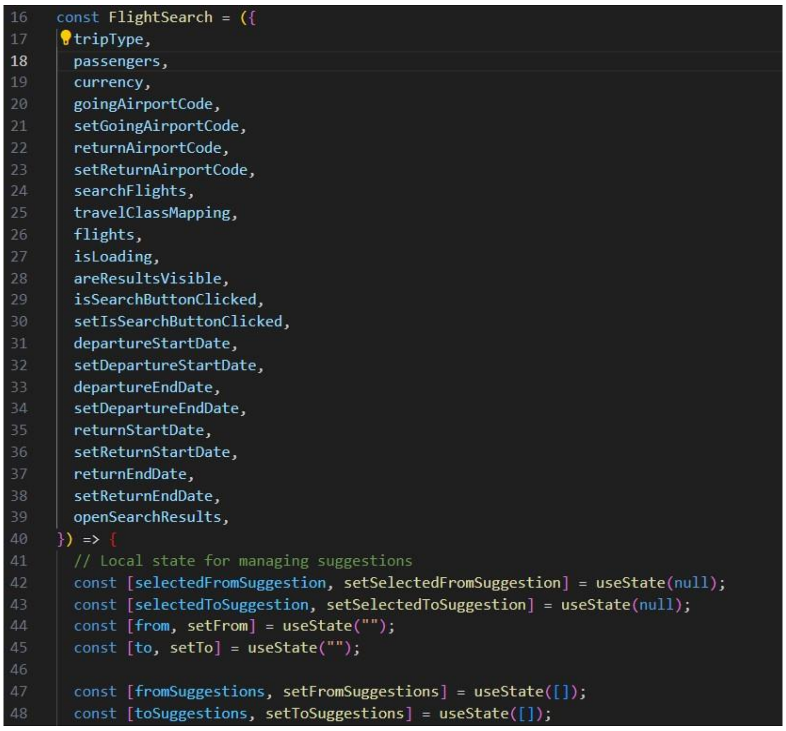
Figure 4 - Project structure

In concordance to the best practices and aiming for maintainability, the React code was divided into components, with each component not exceeding 300 lines of code. Adopting this approach not only keeps the code clear and understandable but also considerably improves the application's extensibility. The benefit of managing fewer components is that they are simpler to handle.

To enhance maintainability, a lot of consideration was given during the design phase of this structure ensuring that each component follows the single responsibility principle. By identifying both components and their functionalities it simplifies the overall management the project.
The primary objective behind implementing this project structure was maintainability of the code. Now its' time to explore the specific design and code elements that contribute towards making the application efficient.

The development of the application began with coding the ”FlightSearch” component. The ”FlightSearch” component is the base of the application and it is where users will input their travel details to search for flights. The component makes use of various hooks from React to manage its state and effects.

The code for the ”FlightSearch” component starts with defining a set of state variables using the useState hook. These variables store the values of the various inputs in the form, such as the departure and arrival locations or the dates of travel.

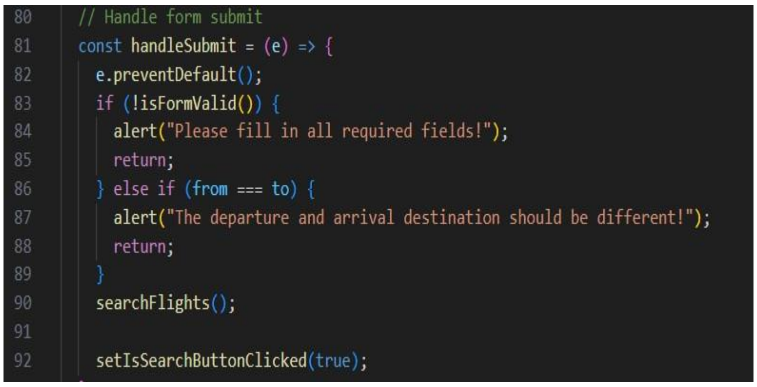


```
const FlightSearch = ({
  tripType,
  passengers,
  currency,
  goingAirportCode,
  setGoingAirportCode,
  returnAirportCode,
  setReturnAirportCode,
  searchFlights,
  travelClassMapping,
  flights,
  isLoading,
  areResultsVisible,
  isSearchButtonClicked,
  setIsSearchButtonClicked,
  departureStartDate,
  setDepartureStartDate,
  departureEndDate,
  setDepartureEndDate,
  returnStartDate,
  setReturnStartDate,
  returnEndDate,
  setReturnEndDate,
  openSearchResults,
}) => {
  // Local state for managing suggestions
  const [selectedFromSuggestion, setSelectedFromSuggestion] = useState(null);
  const [selectedToSuggestion, setSelectedToSuggestion] = useState(null);
  const [from, setFrom] = useState("");
  const [to, setTo] = useState("");

  const [fromSuggestions, setFromSuggestions] = useState([]);
  const [toSuggestions, setToSuggestions] = useState([]);
```

Figure 6 – The „FlightSearch” component state variables

The “useEffect” hook is used to attach and clean up event listeners. For instance, there is an event listener set up to close the airport suggestions list when the user clicks outside of it. There are also “useEffect” hooks set up to log the airport codes and to open the flight search results once they are updated, which were mainly used for testing purposes.

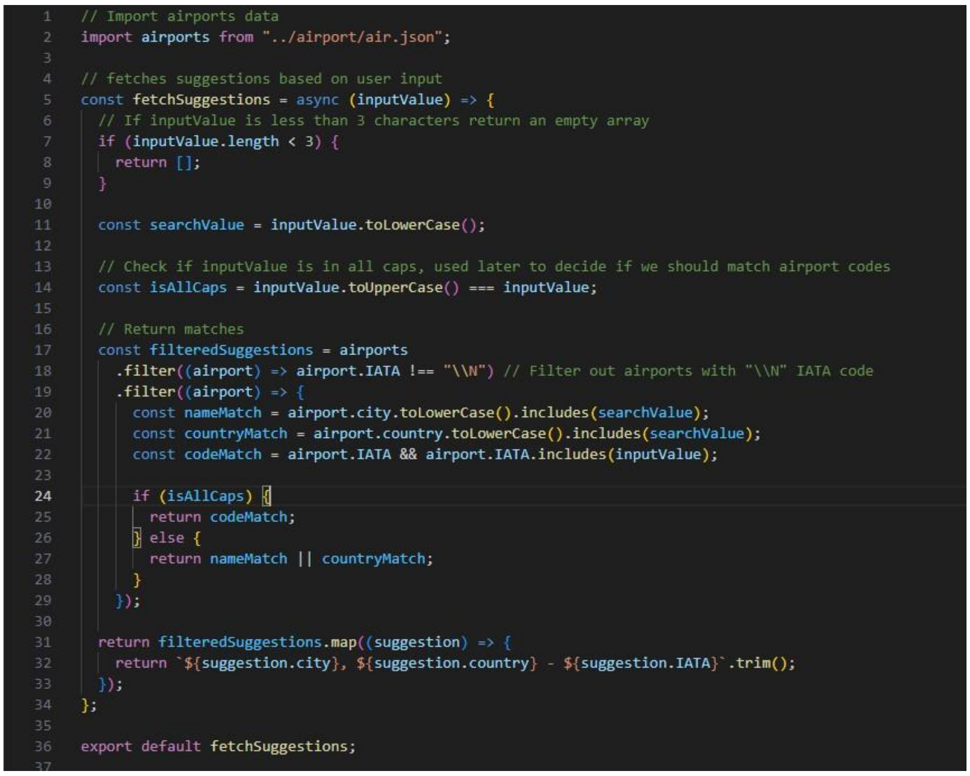
The form uses controlled inputs, which means that the React state is the source for the input elements. The value of every input is connected to a state variable. Whenever the user types something into the input an “onChange” event handler is triggered, causing the state to update accordingly. Upon the submission of the form a “handleSubmit” function is executed and before proceeding with the search, the function checks if the form is valid



```
  // Handle form submit
  const handleSubmit = (e) => {
    e.preventDefault();
    if (!isFormValid()) {
      alert("Please fill in all required fields!");
      return;
    } else if (from === to) {
      alert("The departure and arrival destination should be different!");
      return;
    }
    searchFlights();

    setIsSearchButtonClicked(true);
```

Figure 7 – handleSubmit function

A key feature of this component is the auto-suggestion feature for the departure and arrival input fields. The feature uses the “fetchSuggestions” function, which is triggered on the “onChange” event of the input fields. This function retrieves a list of airport suggestions based on the current value of the input field. If the input has less than three characters, the function will return an empty list elsewise if the input has three or more characters, the function will filter a list of airports and search for matches with the city, country, or IATA code. Once it has a list of matches, it formats each match to include the city, country, and IATA code and then returns that list. This list of matching airport names is then used to update the state variables, “fromSuggestions” and “toSuggestions”, providing the user with a list of possible airports based on their input.



```javascript
// Import airports data
import airports from "../airport/air.json";

// fetches suggestions based on user input
const fetchSuggestions = async (inputValue) => {
  // If inputValue is less than 3 characters return an empty array
  if (inputValue.length < 3) {
    return [];
  }

  const searchValue = inputValue.toLowerCase();

  // Check if inputValue is in all caps, used later to decide if we should match airport codes
  const isAllCaps = inputValue.toUpperCase() === inputValue;

  // Return matches
  const filteredSuggestions = airports
    .filter((airport) => airport.IATA !== "\\N") // Filter out airports with "\\N" IATA code
    .filter((airport) => {
      const nameMatch = airport.city.toLowerCase().includes(searchValue);
      const countryMatch = airport.country.toLowerCase().includes(searchValue);
      const codeMatch = airport.IATA && airport.IATA.includes(inputValue);

      if (isAllCaps) {
        return codeMatch;
      } else {
        return nameMatch || countryMatch;
      }
    });

  return filteredSuggestions.map((suggestion) => {
    return `${suggestion.city}, ${suggestion.country} - ${suggestion.IATA}`.trim();
  });
};

export default fetchSuggestions;
```

Figure 8 – Fetch suggestions function

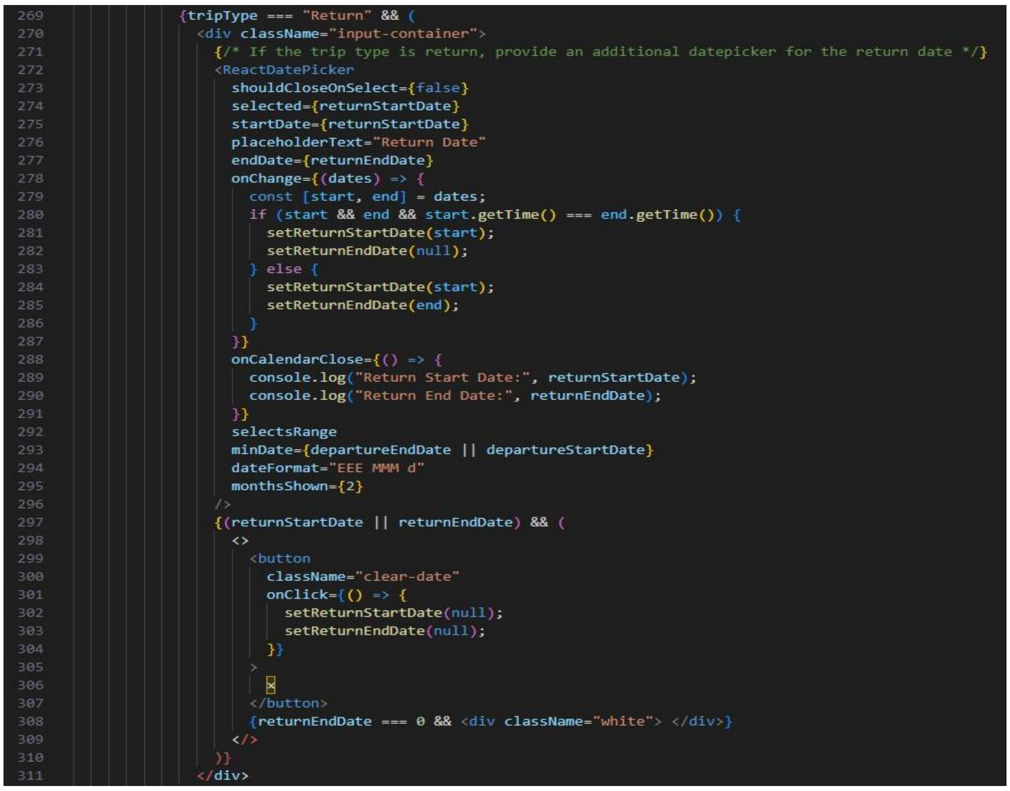
Another notable feature is the use of the “ReactDatePicker” for selecting the departure and return dates. There are two separate instances of the “ReactDatePicker” component in the code snippet. One instance is used for the departure date, while the other serves to manage the return date. Worth mentioning is that the return date picker is conditionally rendered based on whether the trip type is “Return”. Also, a noticeable feature of these date pickers is their capacity to handle date ranges. This functionality, denoted by the “selectsRange” prop, allows users to choose an interval of dates instead of a single one. When the user selects a date range, the “onChange” prop function is triggered. This function takes in an array that contains both the start and end dates. The date picker utilizes two separate state variables, one for the start date and another for the end date. These variables are updated within the “onChange” prop function, and the function dissects the selected dates into “start” and “end” and assigns these values to the corresponding state variables.

The application also considers the scenario where a user may select a single date. When encountering such scenarios, it’s necessary to check if the start and end dates are identical. If they turn out to be so the end date state variable is assigned a null value. This approach effectively treats the input as a single date selection instead of considering it as a range. Furthermore, a “clear date” button is integrated within each date picker and on clicking this button, both the start and end date state variables are reset to null enabling users to easily clear their previously selected date range.

To provide an optimal date selection experience, restrictions are imposed on the range of the dates through the “minDate” and “maxDate” props. The “minDate” for the departure date picker is set as the current date, preventing users from selecting a past date. The

“maxDate”, on the other hand, is contingent on the trip type. If a return trip is chosen, the “maxDate” for the departure date picker is configured to be the start date of the return trip.

This ensures the departure date does not exceed the return date. For the return date picker, the “minDate” is set to the departure start or end date, making sure users don't select a return date prior to their departure.



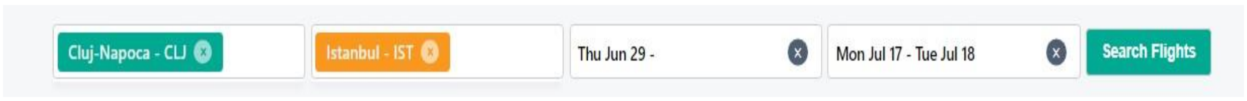
```
{tripType === "Return" && (
  <div className="input-container">
    {/* If the trip type is return, provide an additional datepicker for the return date */}
    <ReactDatePicker
      shouldCloseOnSelect={false}
      selected={returnStartDate}
      startDate={returnStartDate}
      placeholderText="Return Date"
      endDate={returnEndDate}
      onChange={(dates) => {
        const [start, end] = dates;
        if (start && end && start.getTime() === end.getTime()) {
          setReturnStartDate(start);
          setReturnEndDate(null);
        } else {
          setReturnStartDate(start);
          setReturnEndDate(end);
        }
      }}
      onCalendarClose={() => {
        console.log("Return Start Date:", returnStartDate);
        console.log("Return End Date:", returnEndDate);
      }}
      selectsRange
      minDate={departureEndDate || departureStartDate}
      dateFormat="EEE MMM d"
      monthsShown={2}
    />
    {(returnStartDate || returnEndDate) && (
      <>
        <button
          className="clear-date"
          onClick={() => {
            setReturnStartDate(null);
            setReturnEndDate(null);
          }}
        >
          ×
        </button>
        {returnEndDate === 0 && <div className="white"> </div>}
      </>
    )}
</div>
```

Figure 9 - Handling the return case in the Flight Search component.

Finally, depending on whether the application is loading or not, it either displays a loading spinner or the flight search results.

The “FlightSearchResults” component is responsible for displaying the search results, which it receives as props from” FlightSearch”. In the initial stages of application development, the state of the components was managed inside each individual component but as the development progressed, issues passing state between sibling components were encountered. It was decided to lift the state up to their common parent component, App.js. This made it easier to manage and share state among different components, as well as making the overall structure of the application more scalable.

After implementing the CSS into the structure, the flight search component will look like below providing an aesthetic and pleasing appearance.



Cluj-Napoca - CLJ

Istanbul - IST

Thu Jun 29 -

Mon Jul 17 - Tue Jul 18

Figure 10 – Flight Search component view

Following the creation of the “FlightSearch” component, the second component, “Navbar”, was implemented. “Navbar” is a versatile component that is responsible for keeping several dropdowns, each providing a specific functionality within the flight search application.

To make the code easier to organize each dropdown was developed as an individual component. This approach made sense because each dropdown has a specific functionality and state that needs to be handled independently. Also, it allowed us to keep the code for each dropdown isolated and easy to manage.

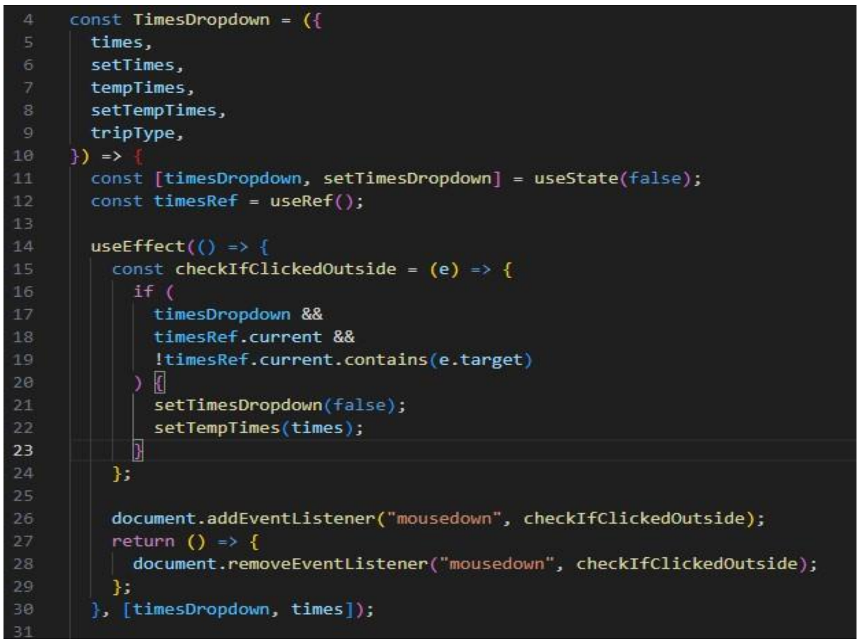
In the “Navbar”, several dropdowns were included, which will be enumerated below:

- **ReturnDropdown**: This dropdown is responsible for handling the trip type and whether the user wants to book a one-way trip or a return trip.
- **PassengerDropdown**: This dropdown manages the count of passengers and whether it's adults, children, or infants, the “PassengerDropdown” component keeps track of it.
- **CabinTypeDropdown**: This dropdown provides the cabin or travel class selection. If the user wants to travel in economy, business, or first class, this component handles it.
- **StopoversDropdown**: Here, the user can specify their preference for stopovers. It can be anything from non-stop flights to a maximum of ten stopovers.
- **TimesDropdown**: The “TimesDropdown” lets users specify their preferred departure and arrival times, which is important for many travelers. If it's a return trip, the users can also specify their preferred times for the return journey.

- **CurrencyDropdown**: Lastly, the “CurrencyDropdown” allows the user to specify the currency in which they want to view the flight options.

Now we’ll get deeper into how these dropdown components are integrated within the “Navbar” component.

Let's begin with the “TimesDropdown”. This component has its own state for whether the dropdown is currently open (timesDropdown). It also stores a reference to its root element for interaction handling (through the timesRef ref), and to handle the click outside functionality, which closes the dropdown if the user clicks outside its edge.



```
const TimesDropdown = ({
  times,
  setTimes,
  tempTimes,
  setTempTimes,
  tripType,
}) => {
  const [timesDropdown, setTimesDropdown] = useState(false);
  const timesRef = useRef();

  useEffect(() => {
    const checkIfClickedOutside = (e) => {
      if (
        timesDropdown &&
        timesRef.current &&
        !timesRef.current.contains(e.target)
      ) {
        setTimesDropdown(false);
        setTempTimes(times);
      }
    };

    document.addEventListener("mousedown", checkIfClickedOutside);
    return () => {
      document.removeEventListener("mousedown", checkIfClickedOutside);
    };
  }, [timesDropdown, times]);

```

Figure 11 - State variables and click outside function

When the dropdown is open, it displays a panel containing a set of sliders that allow the user to specify the time ranges for the departure and arrival times. If the trip type is "Return", additional sliders for the return departure and arrival times are displayed as you can see below.

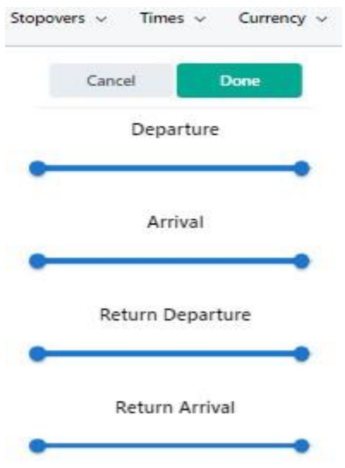


Figure 12 - Times Dropdown

The “Cancel” button resets any changes made within the dropdown, while the “Done” button saves those changes.

The “PassengerDropdown” follows a similar pattern, and it has its own state for whether the dropdown is open (passengersDropdown), and a ref to its root element for interaction handling (passengersRef).

Inside the dropdown, the number of adults, children, and infants can be adjusted. Each group of passengers has its own row, complete with buttons to increase or decrease the count, and an input field that displays the current count. The “Cancel” button will revert any changes to the passenger counts that were made since the dropdown was last opened, while the “Done” button will save those changes.

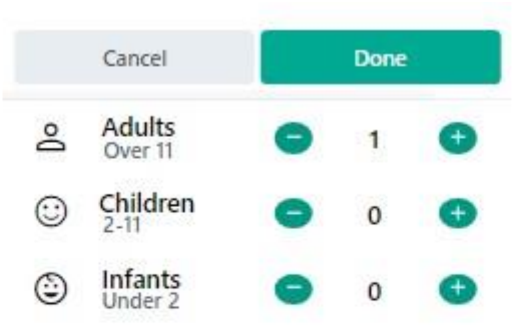


Figure 13 - Passengers Dropdown view

These components use a pattern that provides isolation, making it easier to maintain and scale the code. For instance, “TimesDropdown” is only concerned with time selection logic, while “PassengerDropdown” handles the number of passengers. This way, each dropdown component handles its own logic and state, making the codebase cleaner and easier to read.

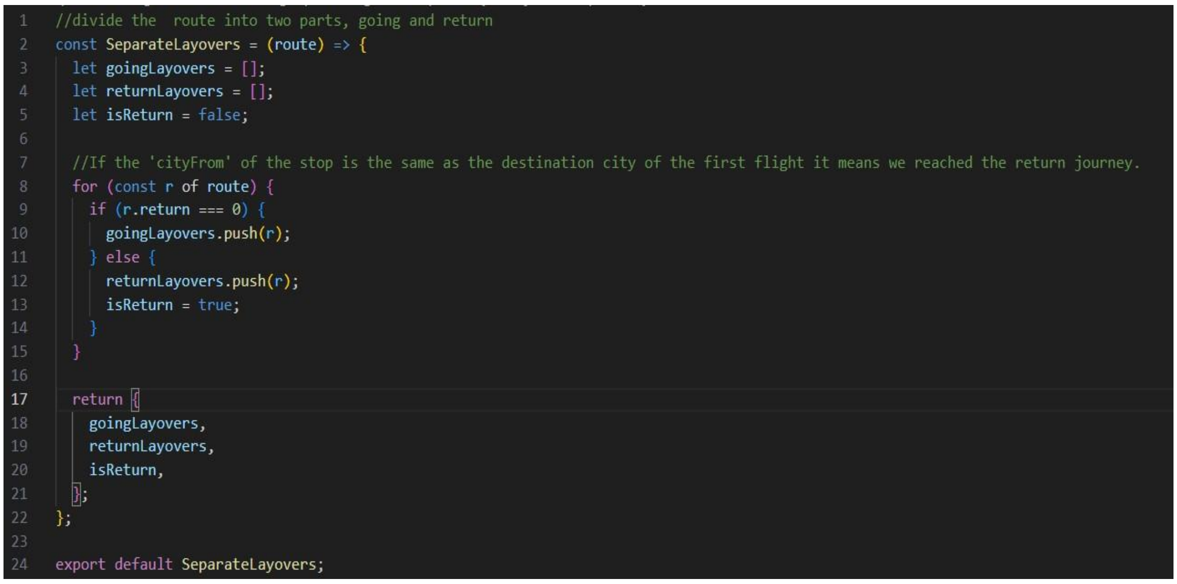
By utilizing this pattern and integrating these individual dropdown components within the main Navbar component, we have a modular and maintainable structure that effectively handles the complexity of various flight search parameters.

After establishing the search parameters through the Navbar component, the design for the “FlightSearchResults” component was next. The “FlightSearchResults” component is in charge of displaying the available flights according to the search parameters.
In order to manage the complexity and to improve readability of the code, the “FlightSearchResults” component was divided into smaller sub-components and utilities:

• **SortOptions**: This is a component that handles the sorting of flight options, and it gives the users the ability to sort the flights according to price and duration.

• **RenderFlights**: This component is in charge of rendering the list of available flights on the screen and it's designed to take various parameters, such as the list of flights, sorting order, and passenger details, and display the relevant information.

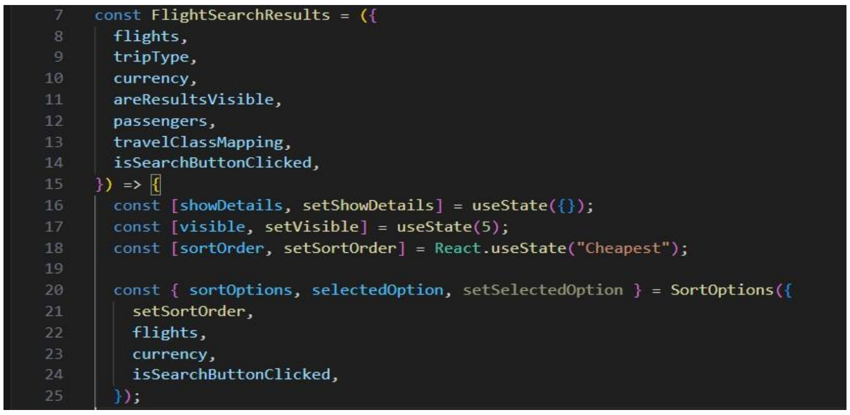
• **SeparateLayovers**: This is a component used within the “RenderFlights” component to handle the division of a flight's route into two parts, the going and the return layovers. It takes the flight's route as an argument, and based on this data, returns two separate arrays representing the going and return layovers. This is necessary to properly calculate and display the durations of each part of the trip separately.


```
//divide the  route into two parts, going and return
const SeparateLayovers = (route) => {
  let goingLayovers = [];
  let returnLayovers = [];
  let isReturn = false;

  //If the 'cityFrom' of the stop is the same as the destination city of the first flight it means we reached the return journey.
  for (const r of route) {
    if (r.return === 0) {
      goingLayovers.push(r);
    } else {
      returnLayovers.push(r);
      isReturn = true;
    }
  }

  return {
    goingLayovers,
    returnLayovers,
    isReturn,
  };
};

export default SeparateLayovers;
```
Figure 14 - Separate Layovers Component

• **RenderLayovers**: This component takes as arguments the layovers for each part of the journey (going and returning), whether the current layovers are for a return flight, the total duration of the layovers, and a function to find airline details. This component is used to render the details of each layover, such as the departure and arrival times, the airline, and the total duration. This information is displayed whenever the user chooses to view the flight details.

• **formatDate and extractTime**: These are utility functions for handling the date and time data of the flights. These functions are used to ensure that the date and time information is displayed in a constant format. The "FlightSearchResults" component begins by initializing its local state variables. These include the "showDetails" state, that holds information about which flight's details should be displayed to the user, the visible state variable which controls the "Load More" functionality and determines the number of flight results that should be displayed at a time and the "sortOrder" state which keeps track of the user's preferred sorting order.

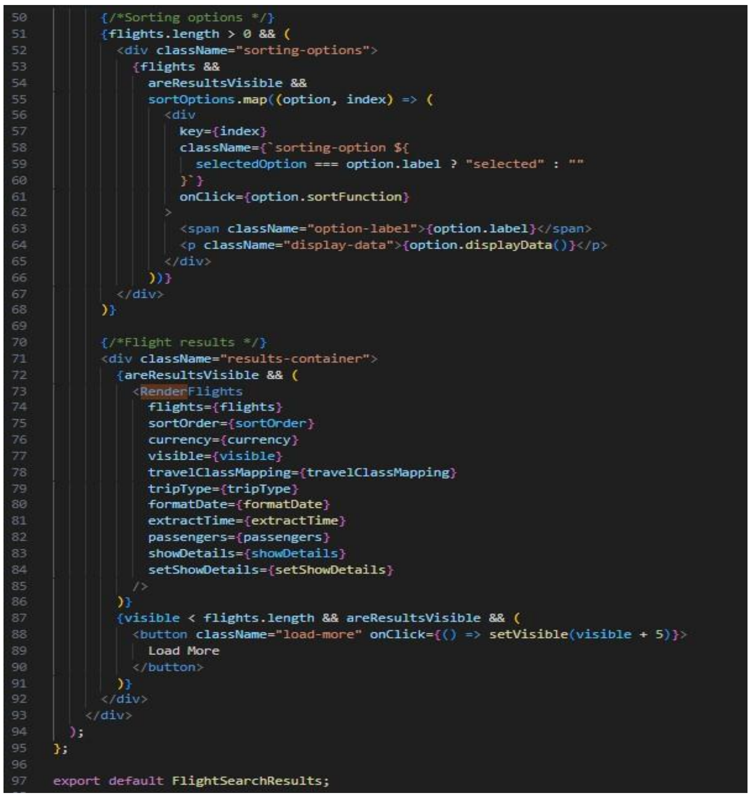
Once the states are initialized, the "SortOptions" component is used to set up the sorting options, and it is provided with the required parameters including the "setSortOrder" function that allows it to update the "sortOrder" state of the "FlightSearchResults" component.



```jsx
const FlightSearchResults = ({
  flights,
  tripType,
  currency,
  areResultsVisible,
  passengers,
  travelClassMapping,
  isSearchButtonClicked,
}) => {
  const [showDetails, setShowDetails] = useState({});
  const [visible, setVisible] = useState(5);
  const [sortOrder, setSortOrder] = React.useState("Cheapest");

  const { sortOptions, selectedOption, setSelectedOption } = SortOptions({
    setSortOrder,
    flights,
    currency,
    isSearchButtonClicked,
  });
```

Figure 15 - The state variables inside the "FlightSearchResults" component

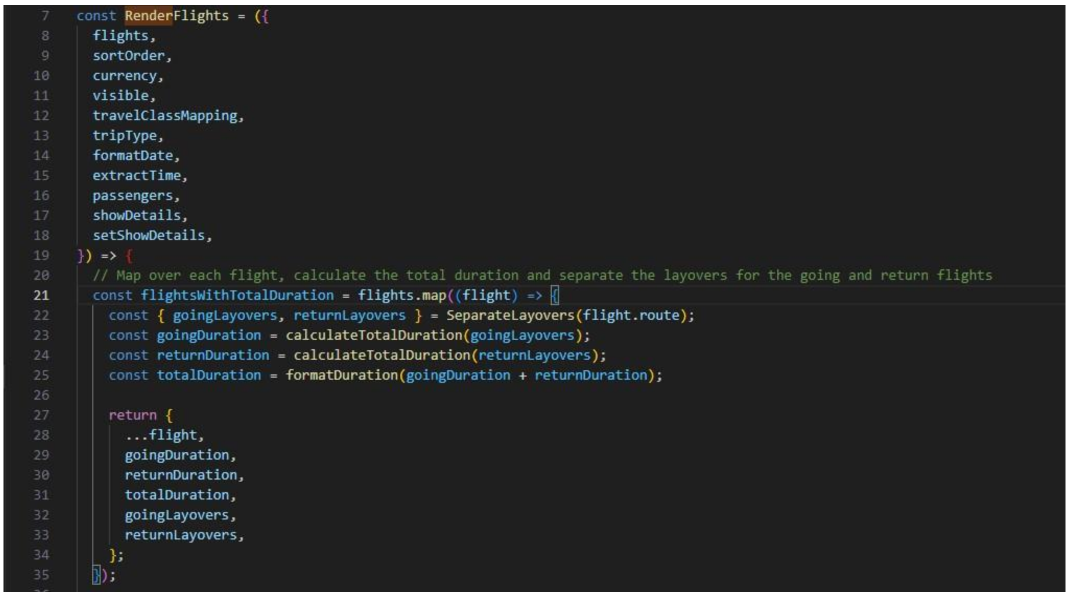
The component then returns the JSX, which is the user interface for the "FlightSearchResults" component. Depending on whether there are any flights to display and whether the search button was clicked, it either displays a message indicating that no flights were found, or it displays the sorting options and the flight results.



```jsx
        {/*Sorting options */}
        {flights.length > 0 && (
          <div className="sorting-options">
            {flights &&
              areResultsVisible &&
              sortOptions.map((option, index) => (
                <div
                  key={index}
                  className={`sorting-option ${
                    selectedOption === option.label ? "selected" : ""
                  }`}
                  onClick={option.sortFunction}
                >
                  <span className="option-label">{option.label}</span>
                  <p className="display-data">{option.displayData()}</p>
                </div>
              ))}
          </div>
        )}

        {/*Flight results */}
        <div className="results-container">
          {areResultsVisible && (
            <RenderFlights
              flights={flights}
              sortOrder={sortOrder}
              currency={currency}
              visible={visible}
              travelClassMapping={travelClassMapping}
              tripType={tripType}
              formatDate={formatDate}
              extractTime={extractTime}
              passengers={passengers}
              showDetails={showDetails}
              setShowDetails={setShowDetails}
            />
          )}
          {visible < flights.length && areResultsVisible && (
            <button className="load-more" onClick={() => setVisible(visible + 5)}>
              Load More
            </button>
          )}
        </div>
      </div>
    );
};

export default FlightSearchResults;
```

Figure 16 - The code related with the "SortingOptions" and "FlightResults"

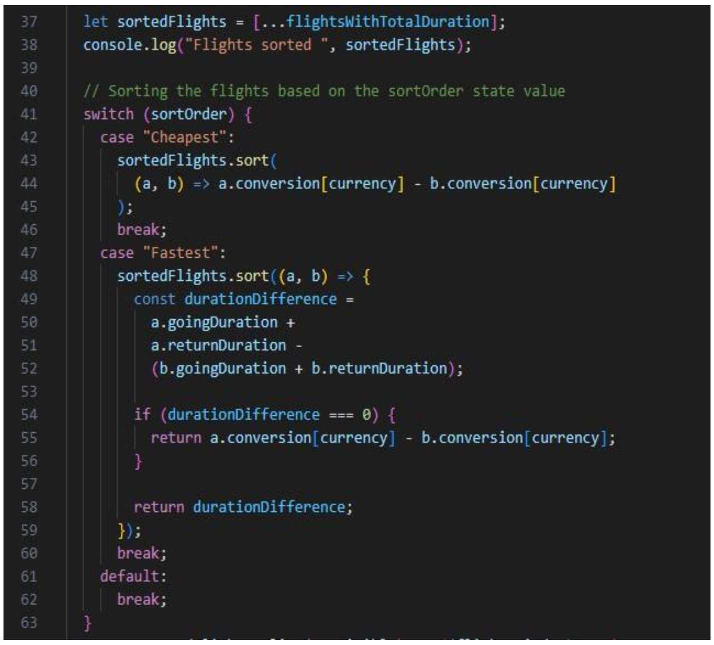
If there are flights, the sorting options are rendered dynamically for each option and returned by the "SortOptions" component. Clicking on an option updates the selected sorting option and sorts the list of flights.
In this section the user interface for each flight is generated. This will be explained further. The user interface includes various props, such as "flights" "sortOrder" "currency" "visible" "travelClassMapping" "tripType" "formatDate" "extractTime" "passengers" "showDetails " and "setShowDetails." This component begins by processing each flight from the "flights" prop. The "SeparateLayovers" function is utilized here to differentiate between the going and return flights. Then the total flight duration is calculated using the "calculateTotalDuration" function and is presented through the "formatDuration" function



```
const RenderFlights = ({
  flights,
  sortOrder,
  currency,
  visible,
  travelClassMapping,
  tripType,
  formatDate,
  extractTime,
  passengers,
  showDetails,
  setShowDetails,
}) => {
  // Map over each flight, calculate the total duration and separate the layovers for the going and return flights
  const flightsWithTotalDuration = flights.map((flight) => {
    const { goingLayovers, returnLayovers } = SeparateLayovers(flight.route);
    const goingDuration = calculateTotalDuration(goingLayovers);
    const returnDuration = calculateTotalDuration(returnLayovers);
    const totalDuration = formatDuration(goingDuration + returnDuration);

    return {
      ...flight,
      goingDuration,
      returnDuration,
      totalDuration,
      goingLayovers,
      returnLayovers,
    };
  });
```

Figure 17 - Displaying the use of the separate layovers and durations functions.

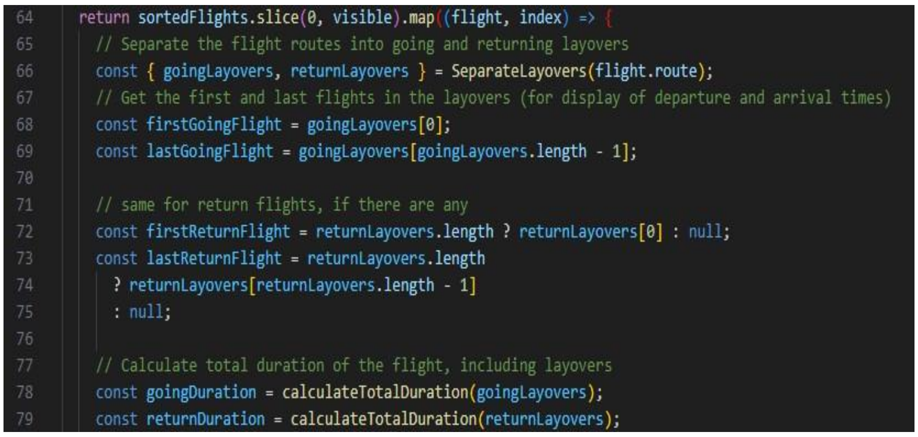
Depending on the state of "sortOrder", flights are then sorted and in the case of "Cheapest", flights are organized based on the lowest price, while for "Fastest", duration is considered. If two flights have the same duration, the price is then used to sort them. Also, the visible prop dictates the number of flights displayed on the screen at any given time, which is implemented by slicing the sorted flights array.



```
let sortedFlights = [...flightsWithTotalDuration];
console.log("Flights sorted ", sortedFlights);

// Sorting the flights based on the sortOrder state value
switch (sortOrder) {
  case "Cheapest":
    sortedFlights.sort(
      (a, b) => a.conversion[currency] - b.conversion[currency]
    );
    break;
  case "Fastest":
    sortedFlights.sort((a, b) => {
      const durationDifference =
        a.goingDuration +
        a.returnDuration -
        (b.goingDuration + b.returnDuration);

      if (durationDifference === 0) {
        return a.conversion[currency] - b.conversion[currency];
      }

      return durationDifference;
    });
    break;
  default:
    break;
}
```

Figure 18 – Sorting the flights

The component further processes the sorted flights by breaking down the flight routes into going and return layovers, identifying the initial and ending flights within the layovers, and then calculating the total flight duration with layovers included.

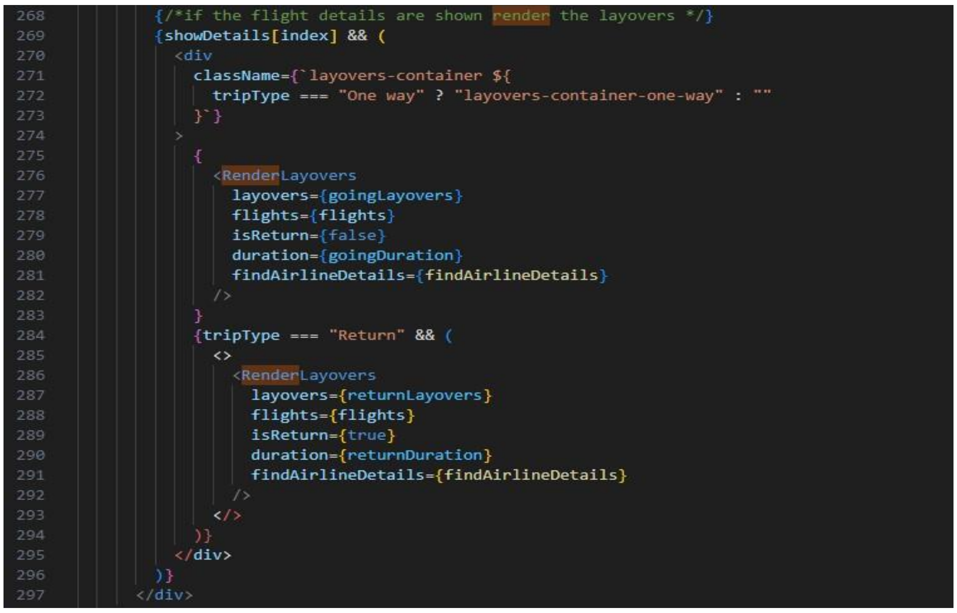


```
return sortedFlights.slice(0, visible).map((flight, index) => {
  // Separate the flight routes into going and returning layovers
  const { goingLayovers, returnLayovers } = SeparateLayovers(flight.route);
  // Get the first and last flights in the layovers (for display of departure and arrival times)
  const firstGoingFlight = goingLayovers[0];
  const lastGoingFlight = goingLayovers[goingLayovers.length - 1];

  // same for return flights, if there are any
  const firstReturnFlight = returnLayovers.length ? returnLayovers[0] : null;
  const lastReturnFlight = returnLayovers.length
    ? returnLayovers[returnLayovers.length - 1]
    : null;

  // Calculate total duration of the flight, including layovers
  const goingDuration = calculateTotalDuration(goingLayovers);
  const returnDuration = calculateTotalDuration(returnLayovers);
```

Figure 19 – Separating the flights and retriving the durations

The “RenderFlights” component then employs JSX to manifest the flight data. The displayed information contains departure and arrival times, cities, baggage limit (if any), travel class, and nights in the destination (in case of return flights). Furthermore, a container for the price and booking details is included in the JSX, with the booking link.

In addition to this, there is a toggle button to manage the visibility of flight details. When expanded, the “RenderLayovers” component comes into play, providing the details of each layover. The final output of the “RenderFlights” component is a JSX list, with each 'flight-result' div symbolizing a single flight. If there are more flights available than what is currently being shown, a "Load More" button is displayed. Clicking on this button increases the number of flights being displayed.



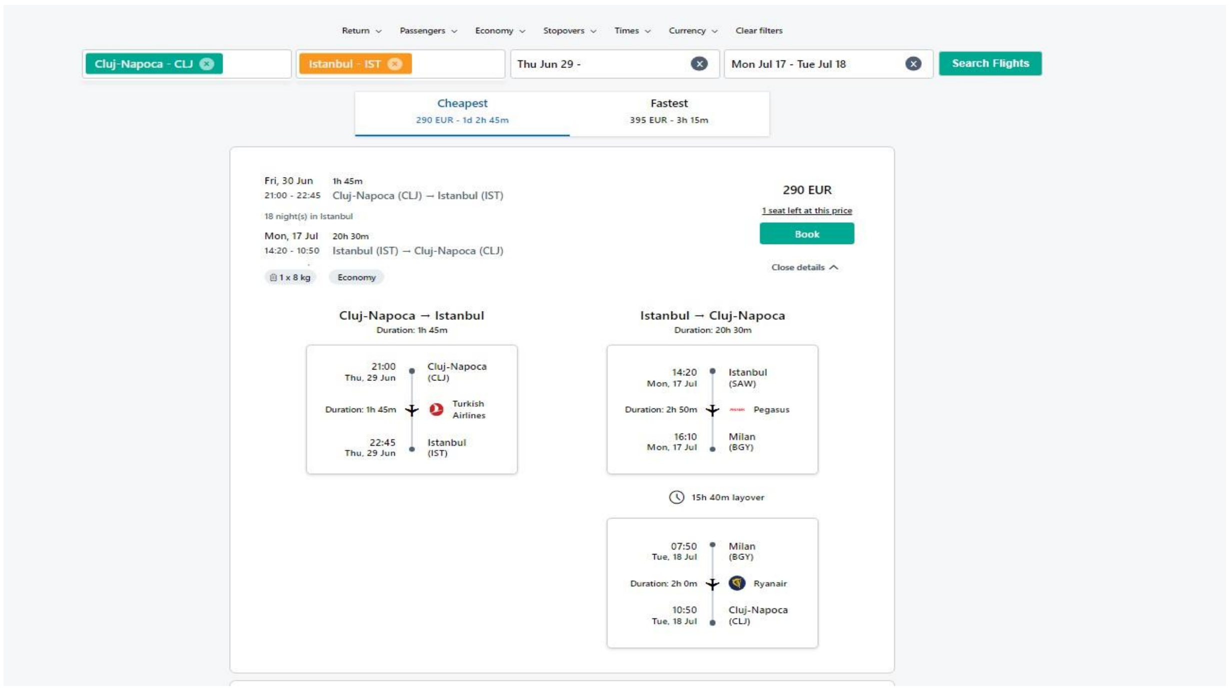
```
      {/*if the flight details are shown render the layovers */}
      {showDetails[index] && (
        <div
          className={`layovers-container ${
            tripType === "One way" ? "layovers-container-one-way" : ""
          }`}
        >
          {
            <RenderLayovers
              layovers={goingLayovers}
              flights={flights}
              isReturn={false}
              duration={goingDuration}
              findAirlineDetails={findAirlineDetails}
            />
          }
          {tripType === "Return" && (
            <>
              <RenderLayovers
                layovers={returnLayovers}
                flights={flights}
                isReturn={true}
                duration={returnDuration}
                findAirlineDetails={findAirlineDetails}
              />
            </>
          )}
        </div>
      )}
    </div>
```

Figure 20 – Featuring the show details button

Also, the two components mentioned previously, “RenderLayovers” and “SeparateLauyovers” are integrated into this larger component. The “RenderLayovers” component displays the layover details for each flight and is utilized to show the layover information when flight details are expanded while “SeparateLayovers” function, differentiates the going and return layovers and it is used to segregate the layovers for each flight.

By dividing the “FlightSearchResults” component into sub-components and utility functions, the code was kept clean, organized, and easy to read. Each sub-component is only responsible for its specific task, allowing for easy debugging and future updates. Furthermore, the use of utility functions for date and time formatting ensures that these operations are consistent across the application.

Before moving to the final component, let’s take a look at the visual appearance of this component, which can be seen below:


Return ⌄ Passengers ⌄ Economy ⌄ Stopovers ⌄ Times ⌄ Currency ⌄ Clear filters

Cluj-Napoca - CLJ | Istanbul - IST | Thu Jun 29 - | Mon Jul 17 - Tue Jul 18 | Search Flights

Cheapest 290 EUR - 1d 2h 45m | Fastest 395 EUR - 3h 15m

Fri, 30 Jun 1h 45m
21:00 - 22:45 Cluj-Napoca (CLJ) → Istanbul (IST)
18 night(s) in Istanbul
Mon, 17 Jul 20h 30m
14:20 - 10:50 Istanbul (IST) → Cluj-Napoca (CLJ)
1 x 8 kg Economy

290 EUR
1 seat left at this price
Book
Close details

Cluj-Napoca → Istanbul
Duration: 1h 45m
21:00 Thu, 29 Jun Cluj-Napoca (CLJ)
Duration: 1h 45m Turkish Airlines
22:45 Thu, 29 Jun Istanbul (IST)

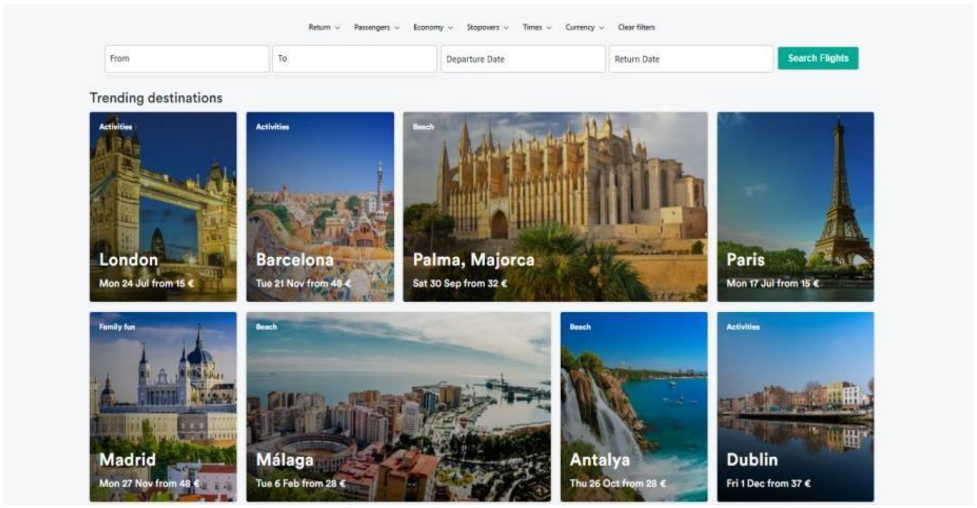
Istanbul → Cluj-Napoca
Duration: 20h 30m
14:20 Mon, 17 Jul Istanbul (SAW)
Duration: 2h 50m Pegasus
16:10 Mon, 17 Jul Milan (BGY)
15h 40m layover
07:50 Tue, 18 Jul Milan (BGY)
Duration: 2h 0m Ryanair
10:50 Tue, 18 Jul Cluj-Napoca (CLJ)

Figure 21 – An example of a real-time flight from Cluj-Napoca to Istanbul

Another important feature that contributes to the versatility and user-friendliness of the application is the integration of the Kiwi Widget. It has its own component, Widget.js, and it serves as a compact and visually appealing tool that delivers quick access to search for the cheapest destinations.

The Widget component primarily uses the “useEffect” hook to execute a side effect in the component. In this case, upon mounting, it dynamically creates a script and a div element. The script element then points to the Kiwi widget's JavaScript file.

This widget adds an additional layer of functionality to the application and increases user engagement by showing the cheapest destinations available. It is directly sourced from Kiwi's Tequila solution, making it easy to integrate into the application. The easy configuration and out-of-the-box functionality of the widget further exemplifies the benefits of leveraging third-party solutions.


Return ⌄ Passengers ⌄ Economy ⌄ Stopovers ⌄ Times ⌄ Currency ⌄ Clear filters

From | To | Departure Date | Return Date | Search Flights

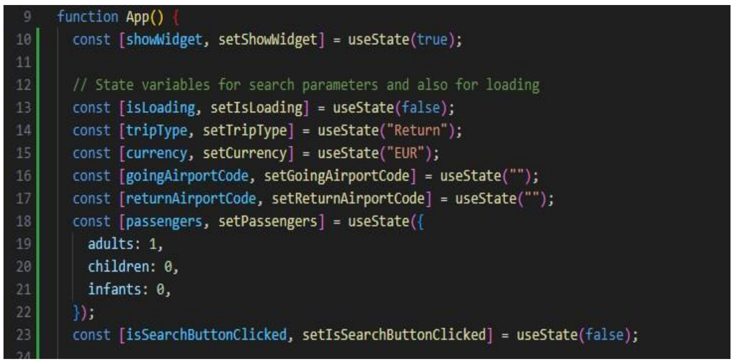
Trending destinations

Activities — London — Mon 24 Jul from 15 €
Activities — Barcelona — Tue 21 Nov from 48 €
Beach — Palma, Majorca — Sat 30 Sep from 32 €
Paris — Mon 17 Jul from 15 €
Family fun — Madrid — Mon 27 Nov from 48 €
Beach — Málaga — Tue 6 Feb from 28 €
Beach — Antalya — Thu 26 Oct from 28 €
Activities — Dublin — Fri 1 Dec from 37 €

Figure 22 – The visuals of the widget in my application

The last, but most crucial, component of the application is App.js. This is where the majority of the application's state management happens, following the principle of "lifting the

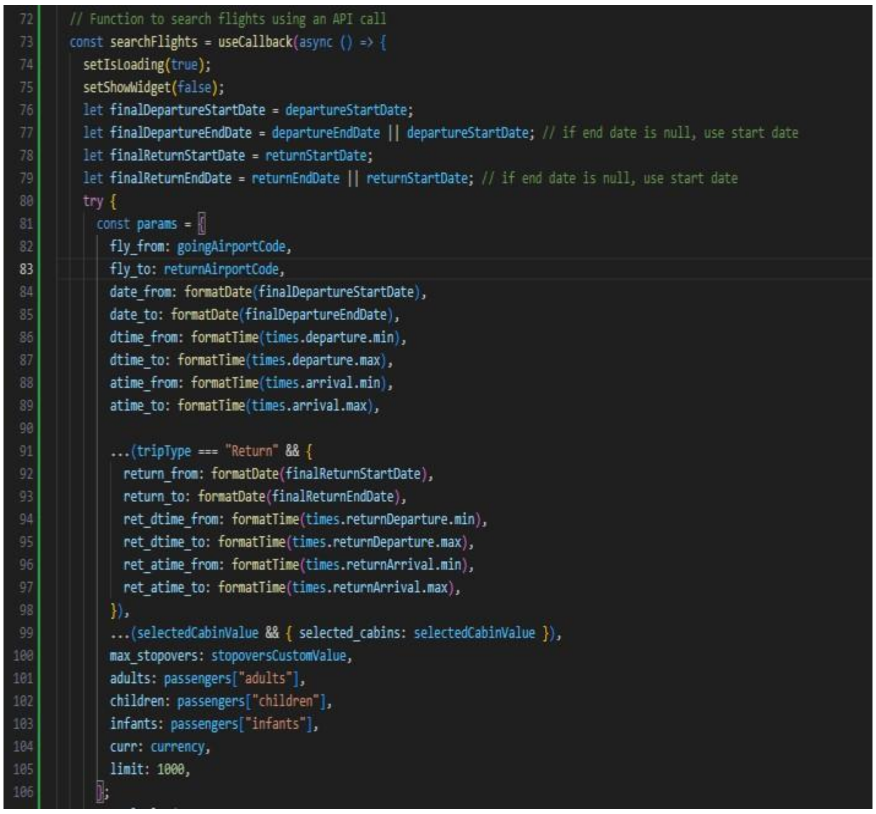
state up" in React. The purpose of lifting state up is to facilitate communication between sibling components.
In App.js, several states are defined, such as “isLoading”, “showWidget”, “flights”, and others. By placing these states in App.js, they can be efficiently managed and accessed by various child components. These states are then passed down to child components as props, simplifying state handling and reducing complexity.



```
function App() {
  const [showWidget, setShowWidget] = useState(true);

  // State variables for search parameters and also for loading
  const [isLoading, setIsLoading] = useState(false);
  const [tripType, setTripType] = useState("Return");
  const [currency, setCurrency] = useState("EUR");
  const [goingAirportCode, setGoingAirportCode] = useState("");
  const [returnAirportCode, setReturnAirportCode] = useState("");
  const [passengers, setPassengers] = useState({
    adults: 1,
    children: 0,
    infants: 0,
  });
  const [isSearchButtonClicked, setIsSearchButtonClicked] = useState(false);
```

Figure 23 – State variables for which the state has been lifted

The “searchFlights” function, for instance, perfectly illustrates the lifting of state. When a user starts a search, “isLoading” is set to true and “showWidget” to false, indicating that the search process has begun. Then, a series of parameters for the API call are prepared based on the application's state.



```
// Function to search flights using an API call
const searchFlights = useCallback(async () => {
  setIsLoading(true);
  setShowWidget(false);
  let finalDepartureStartDate = departureStartDate;
  let finalDepartureEndDate = departureEndDate || departureStartDate; // if end date is null, use start date
  let finalReturnStartDate = returnStartDate;
  let finalReturnEndDate = returnEndDate || returnStartDate; // if end date is null, use start date
  try {
    const params = {
      fly_from: goingAirportCode,
      fly_to: returnAirportCode,
      date_from: formatDate(finalDepartureStartDate),
      date_to: formatDate(finalDepartureEndDate),
      dtime_from: formatTime(times.departure.min),
      dtime_to: formatTime(times.departure.max),
      atime_from: formatTime(times.arrival.min),
      atime_to: formatTime(times.arrival.max),

      ...(tripType === "Return" && {
        return_from: formatDate(finalReturnStartDate),
        return_to: formatDate(finalReturnEndDate),
        ret_dtime_from: formatTime(times.returnDeparture.min),
        ret_dtime_to: formatTime(times.returnDeparture.max),
        ret_atime_from: formatTime(times.returnArrival.min),
        ret_atime_to: formatTime(times.returnArrival.max),
      }),
      ...(selectedCabinValue && { selected_cabins: selectedCabinValue }),
      max_stopovers: stopoversCustomValue,
      adults: passengers["adults"],
      children: passengers["children"],
      infants: passengers["infants"],
      curr: currency,
      limit: 1000,
    };
```

Figure 24 – Snippet of the „searchFlights” function

Upon a successful API call, the flight data is set into the flights state. This is a powerful aspect of state lifting because the user can handle data in one centralized location and distribute it to components that require it. By doing this, the state data is kept consistent preventing potential bugs. The “searchFlights” function also uses “useCallback” to memorize the function itself, preventing unnecessary re-renders and optimizing performance and it is recomputed only when any of the values in the dependency array.

Similarly, in the “openSearchResults” function, state variables are used to construct a URL for the booking and then open it in a new tab. This again demonstrates how the state variables defined and updated in ` App.js are used in the functions. The “openSearchResults” function also stands out as a significant feature of the application because once a user completes a search, this function generates a booking link with specific dates, offering users a direct way of estimating the cost of their trip. It's a good feature that increases the usability of the flight search engine.

As for responsiveness, this concern was addressed by providing a clear message to mobile users, as seen in the image below.

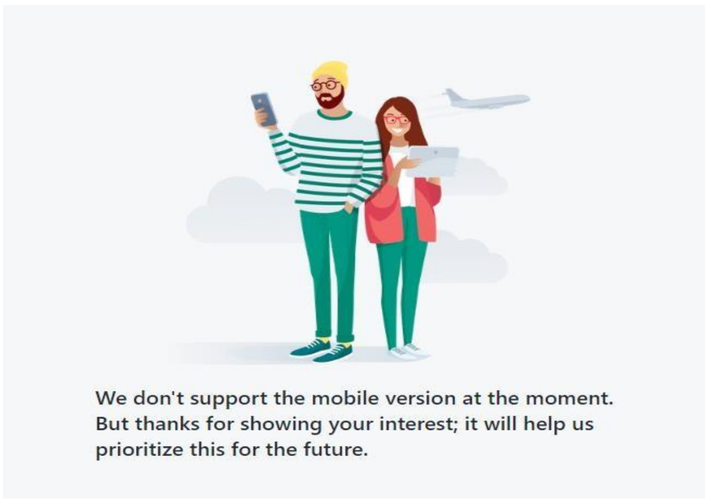

We don't support the mobile version at the moment. But thanks for showing your interest; it will help us prioritize this for the future.

Figure 25 – Mobile Responsiveness

By providing a responsive design or, in this case, a clear message to users about the limitations of the app's design, the user experience is enhanced.
In summary, the App.js component efficiently utilizes the concept of lifting state up and passing props down, ensuring that the application's state is consistently managed and that all components can access the data they require. In addition to introducing important features like ”searchFlights” and “openSearchResults”, various capabilities of the application were emphasized that ultimately improve the user experience.

Flow

Up next, how the application operates is discussed, by focusing on a few key steps:

When a user opens the application, they are greeted by the interface that is rendered by the App.js component. This main component contains the application's state and is responsible for rendering the ”Navbar”, ”FlightSearch” components and the responsive images.

The user fills out their desired search parameters in the ”FlightSearch” component. These parameters are controlled inputs, meaning their values are tied to the application's state in App.js. As the user interacts with these inputs, the state in App.js is updated accordingly.

Once the user clicks on the "Search" button, the ”searchFlights” function is invoked. This function creates an API request based on the current state of the application. When this process occurs, the loading state is activated to inform the user that their search is being processed.

The ”searchFlights” function sends a request to the Kiwi API with the parameters specified by the user. If the request is successful, the APIs response, which includes the flight data, is then set into the flights state.

The flights state is passed down to the ”FlightSearchResults” component as a prop. The ”FlightSearchResults” component, in turn, maps over the flight data to render a list of ”FlightCard” components, each ”FlightCard” displaying information about a specific flight.

After the search, the user will take advantage of the ”openSearchResults" function because a new tab will open automatically. This function constructs a URL for booking with the specific dates and details of the user's search, and opens this link, offering the user an immediate way to book or estimate the cost of their trip.

Throughout this entire process, the state is lifted up in the App.js component, and various states are passed down as props to child components. This structure of ”lifting state up” and ”passing props down” promotes efficient communication between components, and allows for a more manageable code. Here is a high-level overview of the flow:

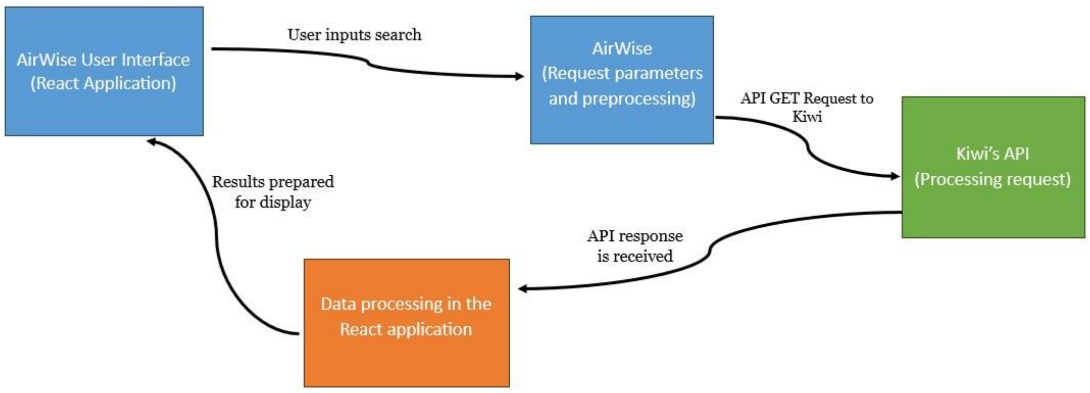

AirWise User Interface (React Application)
User inputs search
AirWise (Request parameters and preprocessing)
API GET Request to Kiwi
Kiwi's API (Processing request)
API response is received
Data processing in the React application
Results prepared for display

Figure 26 - Application Flow

5. Testing and validation

The first layer of the testing was **functional testing**. Each feature, from the trip types and passengers class to the stopovers and search bar, was individually tested to ensure their proper functioning. For instance, a variety of inputs were entered into the search bar, including various airport codes and dates, to verify that it was producing the correct output. Similarly, the “Book” button was repeatedly tested to confirm that it was redirecting users to Kiwi's website accurately.

In the functional testing, the React date picker was rigorously examined. The testing approach had a dual focus. Firstly, it ensured that past dates could not be selected by the users. This was crucial as it is not possible to book flights for dates in the past. Various date inputs were used in this testing process, ranging from today's date to dates back in the past. It was confirmed that past dates were correctly unselectable, and that dates from today onwards were available for selection. Secondly, it tested whether the selected dates were accurately parsed and sent to the API. This is crucial for the correct retrieval of available flights based on the user's intended travel date. Various future dates were selected, and the API request parameters were observed to verify that the selected dates were correctly sent to the API.

In addition to the date picker and search bar, there was a focus on testing each of the components in “FlightSearchResults”. For instance, the functionality of the “SortOptions” component was validated by checking if the sorting of flight options was working accurately. After each sort option, "cheapest" and "fastest," the order of the flights was observed to ensure they matched the selected sorting criteria.

Similarly, the “RenderFlights” component underwent various tests to verify its correct functionality. The accuracy of the flight data rendered onto was verified. Also, it was ensured that the component correctly parsed flight data to differentiate between the going and return flights using the “SeparateLayovers” component.

Tests were conducted to ensure that the “RenderLayovers” component was functioning correctly. This involved confirming that it accurately displayed the detailed information for each layover when the user chose to view flight details.

In all, each feature was tested independently for functionality to ensure an overall smooth navigation.

Moving ahead involved conducting browser console testing. This approach involves analyzing the browser's console log to reveal any possible errors or issues residing within the application. Since

React was used for developing this application, any code-related concerns like errors or warnings would appear on the console.

This method was particularly effective for identifying and rectifying issues in real-time, ensuring a smooth and efficient debugging process.

It involved testing the JavaScript functionalities of the application, such as data fetching from the Kiwi API, state management using React hooks, and dynamic rendering of results. By examining the browser's console log, any errors or anomalies in these areas were promptly identified and corrected.

Integration testing was undertaken to ensure the efficient operation between various components of my application, including its distinctive components like the user interface and Kiwi's API. The process involved confirming that the data fetched from the Kiwi API was correctly interpreted and displayed. Below there is an example of how this was done.

6. Users manual

Hardware and Software Requirements:

To successfully run this application, the following minimum requirements are recommended:

Hardware

- CPU: 1.6 GHz or faster.
- RAM: Minimum 4GB, but 8GB is recommended for a better performance.
- Disk Space: 2GB or more to accommodate the application, Node modules, and any necessary dependencies.

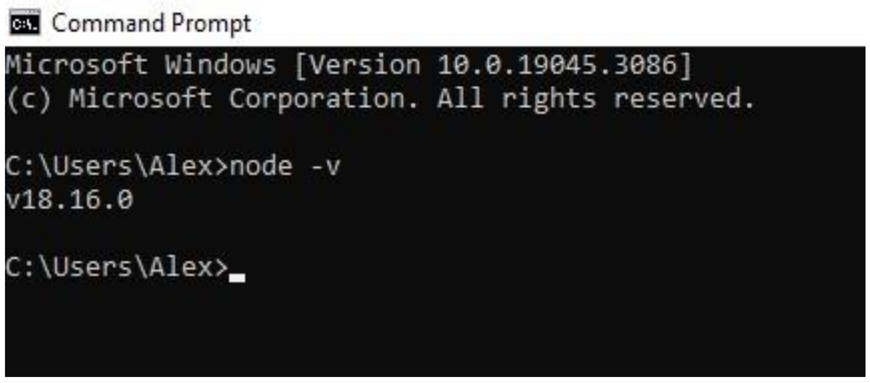
Software

- Operating System: Windows 10, macOS Sierra or higher, or a modern Linux distribution.
- Node.js: v14.0.0 or later.
- Npm (Node Package Manager): v6.0.0 or later.
- IDE (Integrated Development Environment): Visual Studio Code is recommended.
- A modern web browser such as Chrome, Firefox, or Edge.

Installation and Deployment:

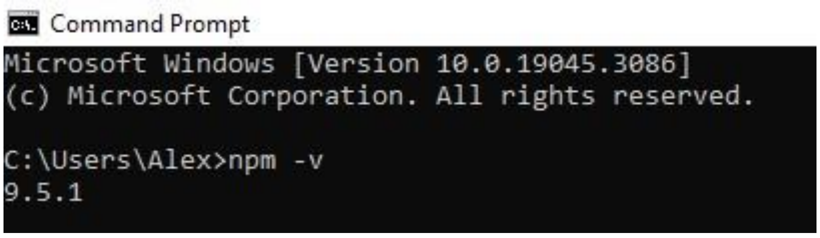
Install Node.js and npm

- Visit the Node.js website (https://nodejs.org) to download and install the latest LTS (Long Term Support) version of Node.js which comes with npm. Verify the installation by typing node -v and npm -v in your terminal/command prompt. The system should display the installed versions of Node.js and npm respectively.



```
Microsoft Windows [Version 10.0.19045.3086]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Alex>node -v
v18.16.0

C:\Users\Alex>_
```

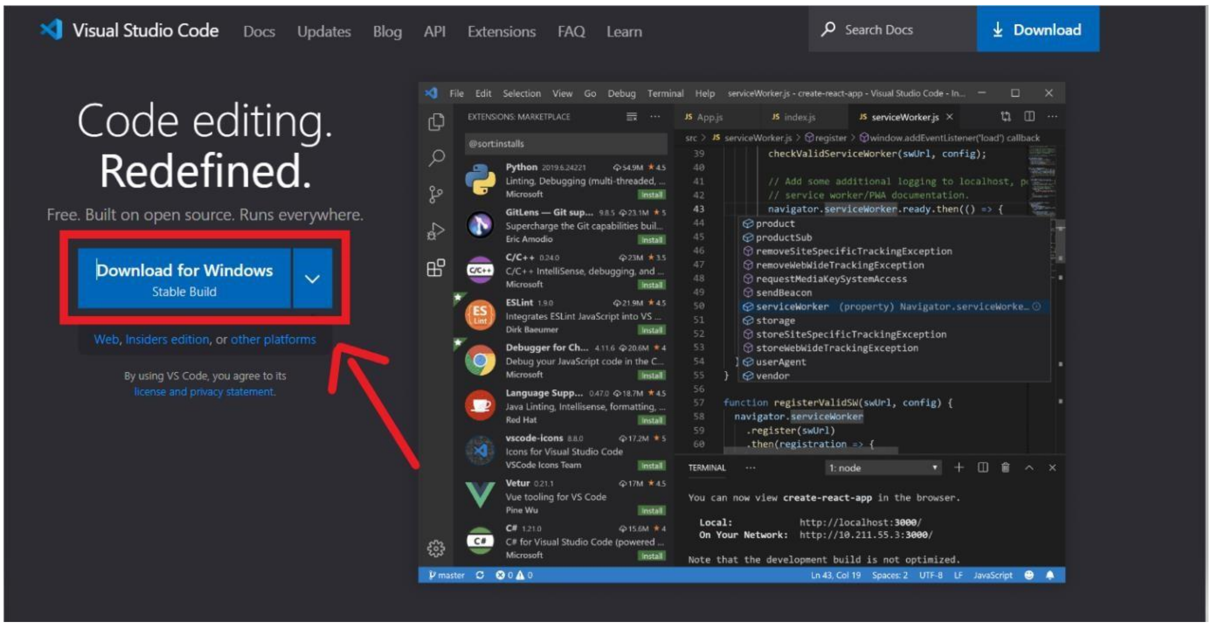
Figure 27 - Node version



```
Microsoft Windows [Version 10.0.19045.3086]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Alex>npm -v
9.5.1
```

Figure 28 – Npm version

- Install Visual Studio Code: Download and install Visual Studio Code from the official website (https://code.visualstudio.com/). Choose the correct distribution, if you are on macOS, choose that version.

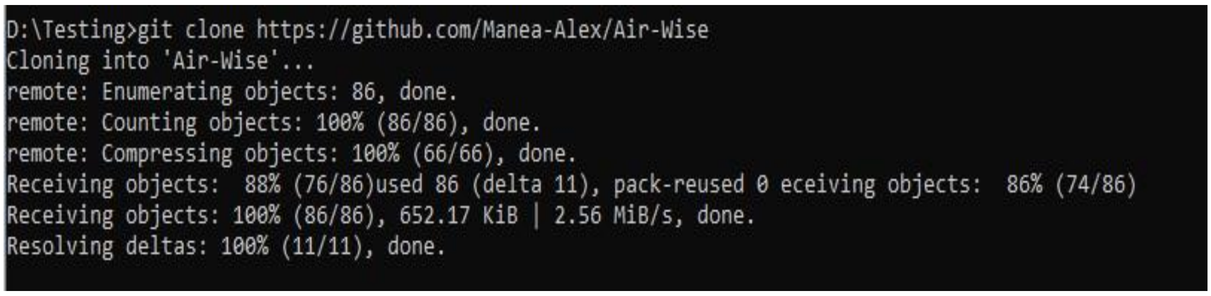


```
        checkValidServiceWorker(swUrl, config);

        // Add some additional logging to localhost, p
        // service worker/PWA documentation.
        navigator.serviceWorker.ready.then(() => {
    }
  }

function registerValidSW(swUrl, config) {
  navigator.serviceWorker
    .register(swUrl)
    .then(registration => {
```

```
You can now view create-react-app in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://10.211.55.3:3000/

Note that the development build is not optimized.
```

Figure 29 - Visual Studio Code installation page

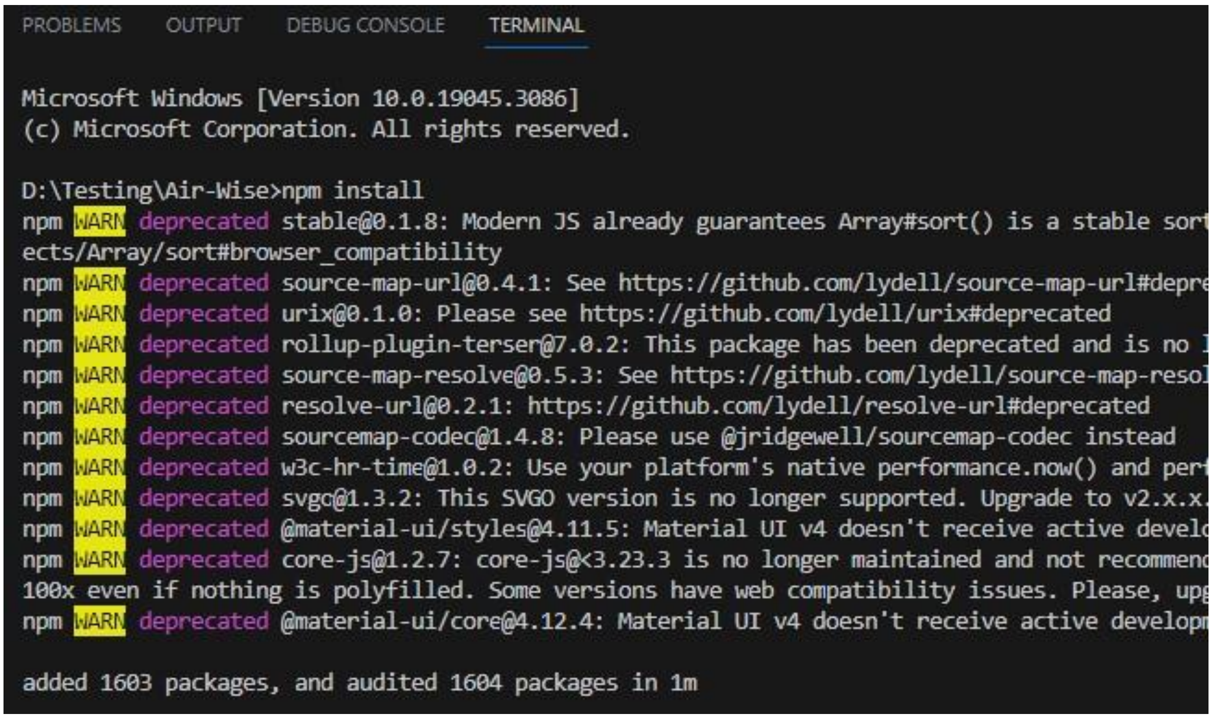
- Clone the project repository: Use Git to clone the project repository to your local machine. In your terminal, navigate to the directory where you want to clone the project and run the command "git clone https://github.com/Manea-Alex/Air-Wise"



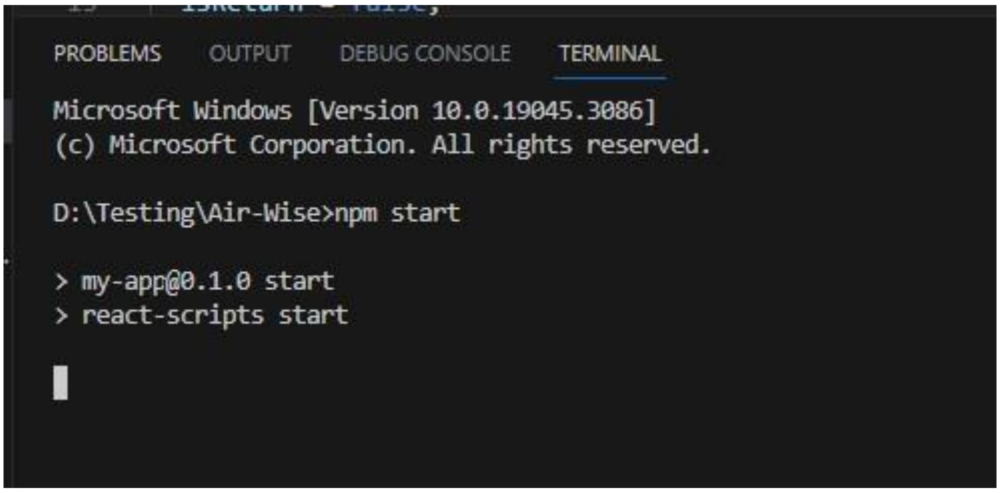
```
D:\Testing>git clone https://github.com/Manea-Alex/Air-Wise
Cloning into 'Air-Wise'...
remote: Enumerating objects: 86, done.
remote: Counting objects: 100% (86/86), done.
remote: Compressing objects: 100% (66/66), done.
Receiving objects:  88% (76/86)used 86 (delta 11), pack-reused 0 eceiving objects:  86% (74/86)
Receiving objects: 100% (86/86), 652.17 KiB | 2.56 MiB/s, done.
Resolving deltas: 100% (11/11), done.
```

Figure 30 - Cloning the repository.

- Install Project Dependencies: Navigate to the project directory in the terminal using cd and run npm install. This will install all necessary project dependencies. This can also be done from the terminal in Visual Studio Code.


```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Microsoft Windows [Version 10.0.19045.3086]
(c) Microsoft Corporation. All rights reserved.

D:\Testing\Air-Wise>npm install
npm WARN deprecated stable@0.1.8: Modern JS already guarantees Array#sort() is a stable sort
ects/Array/sort#browser_compatibility
npm WARN deprecated source-map-url@0.4.1: See https://github.com/lydell/source-map-url#depre
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated rollup-plugin-terser@7.0.2: This package has been deprecated and is no l
npm WARN deprecated source-map-resolve@0.5.3: See https://github.com/lydell/source-map-resol
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
npm WARN deprecated sourcemap-codec@1.4.8: Please use @jridgewell/sourcemap-codec instead
npm WARN deprecated w3c-hr-time@1.0.2: Use your platform's native performance.now() and perf
npm WARN deprecated svgo@1.3.2: This SVGO version is no longer supported. Upgrade to v2.x.x.
npm WARN deprecated @material-ui/styles@4.11.5: Material UI v4 doesn't receive active develo
npm WARN deprecated core-js@1.2.7: core-js@<3.23.3 is no longer maintained and not recommend
100x even if nothing is polyfilled. Some versions have web compatibility issues. Please, upg
npm WARN deprecated @material-ui/core@4.12.4: Material UI v4 doesn't receive active developm

added 1603 packages, and audited 1604 packages in 1m
```
Figure 31 – Installing dependencies

- Start the Application: Still in the project directory, type npm start. This will start the development server, and the application should open in your default browser, usually at http://localhost:3000.


```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Microsoft Windows [Version 10.0.19045.3086]
(c) Microsoft Corporation. All rights reserved.

D:\Testing\Air-Wise>npm start

> my-app@0.1.0 start
> react-scripts start
```
Figure 32 - Running the application.

User Guide:

This application is designed to be intuitive and user-friendly. Here's how the user can navigate it:

- **Home Page**: When the user opens the application, it'll land on the home page where the search bar can be seen.

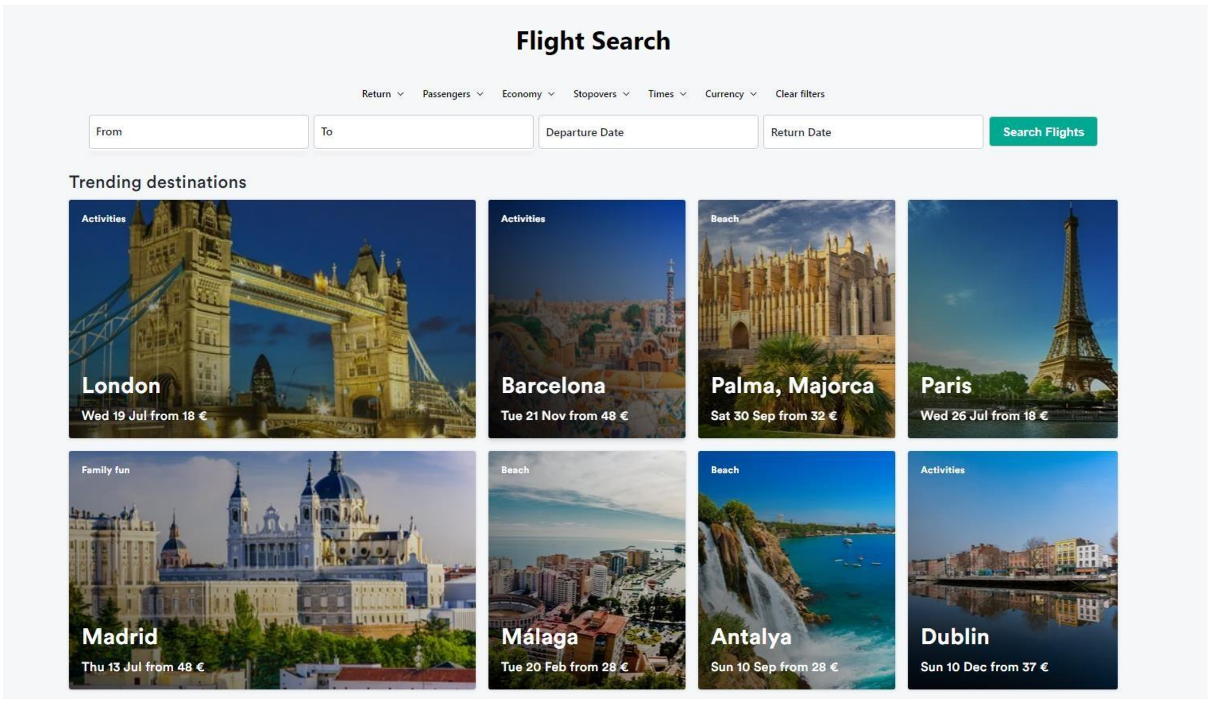
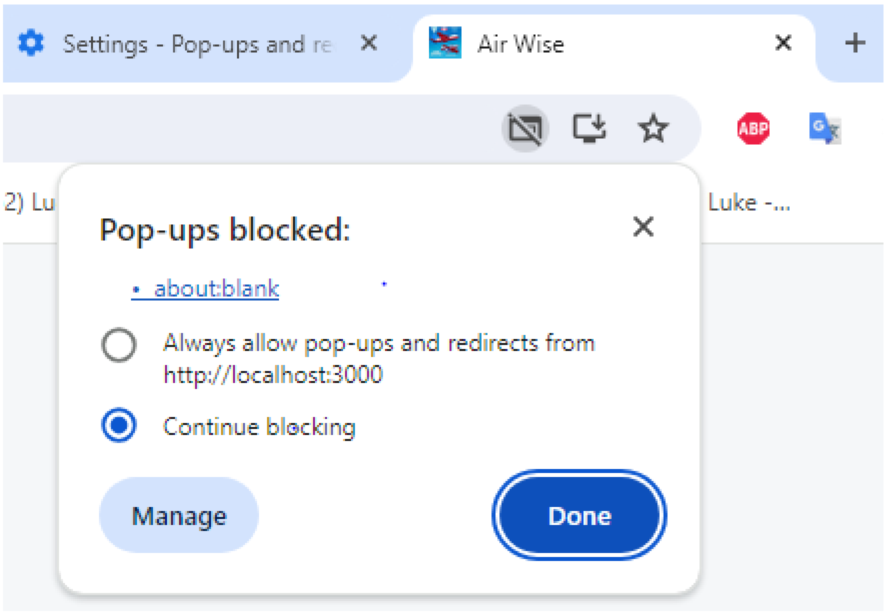


Figure 33 – Home Page

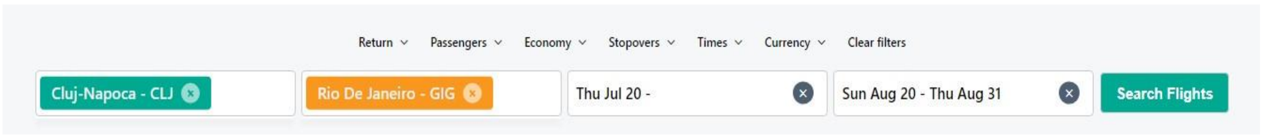
- **NOTE**: Before searching please disable the pop ups otherwise the search won't work and nothing will be displayed or you will be met by an error. Click on „Always allow pop-ups" and then on done.



Always allow pop-ups and redirects from http://localhost:3000 [✓ Continue blocking]

Figure 34 – Allow Pop-ups

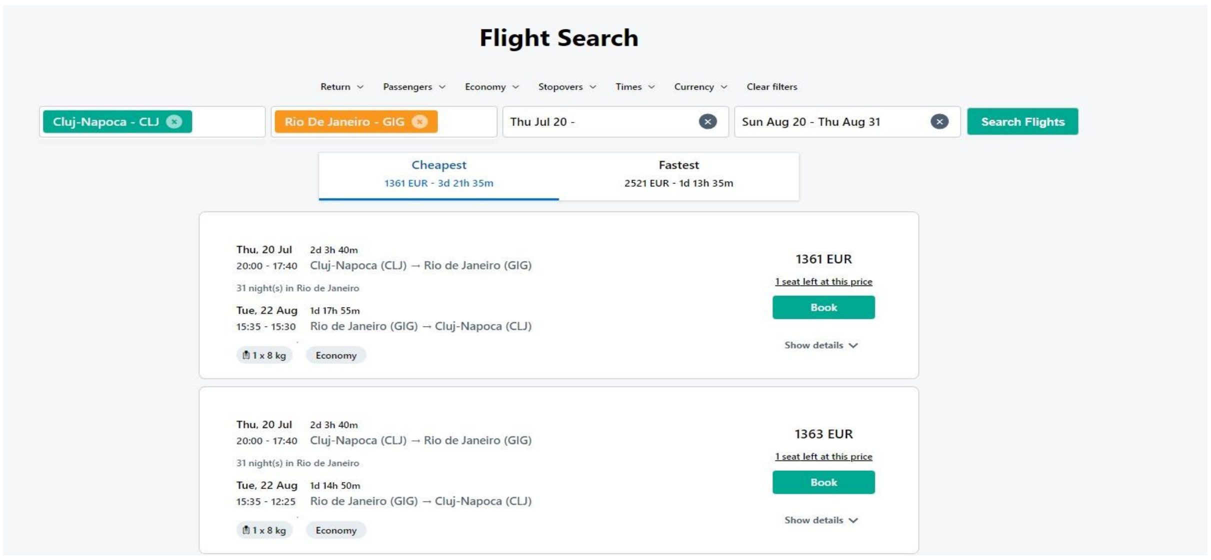
- **Search Bar**: Enter your desired search parameters, such as departure and arrival airports, dates, passenger count, and trip type. Click the "Search" button to see available flights.



Cluj-Napoca - CLJ | Rio De Janeiro - GIG | Thu Jul 20 - | Sun Aug 20 - Thu Aug 31

Figure 35 - Search parameters for a flight from Cluj-Napoca to Rio de Janeiro

- **Flight Results**: A list of flights matching your search parameters will be displayed. Each flight card will display key information such as departure, arrival times, duration, and price. These can be sorted by price and duration.


Flight Search

Return · Passengers · Economy · Stopovers · Times · Currency · Clear filters

Cluj-Napoca - CLJ | Rio De Janeiro - GIG | Thu Jul 20 - | Sun Aug 20 - Thu Aug 31 | Search Flights

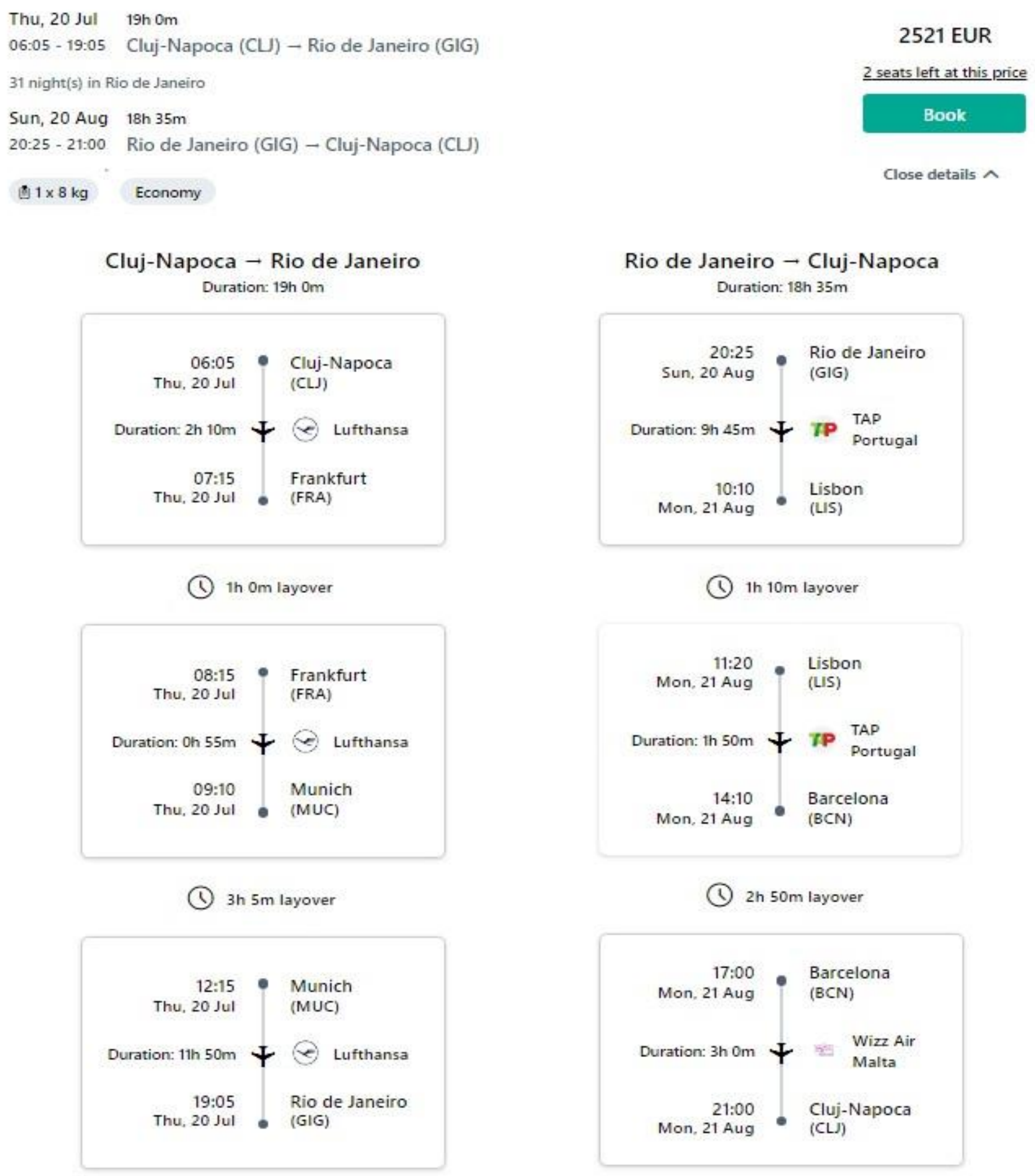
Cheapest	Fastest
1361 EUR - 3d 21h 35m	2521 EUR - 1d 13h 35m

Thu, 20 Jul 2d 3h 40m
20:00 - 17:40 Cluj-Napoca (CLJ) → Rio de Janeiro (GIG)
31 night(s) in Rio de Janeiro
Tue, 22 Aug 1d 17h 55m
15:35 - 15:30 Rio de Janeiro (GIG) → Cluj-Napoca (CLJ)
1 x 8 kg · Economy
1361 EUR
1 seat left at this price
Book
Show details

Thu, 20 Jul 2d 3h 40m
20:00 - 17:40 Cluj-Napoca (CLJ) → Rio de Janeiro (GIG)
31 night(s) in Rio de Janeiro
Tue, 22 Aug 1d 14h 50m
15:35 - 12:25 Rio de Janeiro (GIG) → Cluj-Napoca (CLJ)
1 x 8 kg · Economy
1363 EUR
1 seat left at this price
Book
Show details

Figure 36 - Results

- **Flight Details**: Click on a flight card to view more details about the flight, such as layover information and durations.


Thu, 20 Jul 19h 0m
06:05 - 19:05 Cluj-Napoca (CLJ) → Rio de Janeiro (GIG)
31 night(s) in Rio de Janeiro
Sun, 20 Aug 18h 35m
20:25 - 21:00 Rio de Janeiro (GIG) → Cluj-Napoca (CLJ)
1 x 8 kg · Economy
2521 EUR
2 seats left at this price
Book
Close details

Cluj-Napoca → Rio de Janeiro
Duration: 19h 0m

06:05 Thu, 20 Jul — Cluj-Napoca (CLJ)
Duration: 2h 10m — Lufthansa
07:15 Thu, 20 Jul — Frankfurt (FRA)

1h 0m layover

08:15 Thu, 20 Jul — Frankfurt (FRA)
Duration: 0h 55m — Lufthansa
09:10 Thu, 20 Jul — Munich (MUC)

3h 5m layover

12:15 Thu, 20 Jul — Munich (MUC)
Duration: 11h 50m — Lufthansa
19:05 Thu, 20 Jul — Rio de Janeiro (GIG)

Rio de Janeiro → Cluj-Napoca
Duration: 18h 35m

20:25 Sun, 20 Aug — Rio de Janeiro (GIG)
Duration: 9h 45m — TAP Portugal
10:10 Mon, 21 Aug — Lisbon (LIS)

1h 10m layover

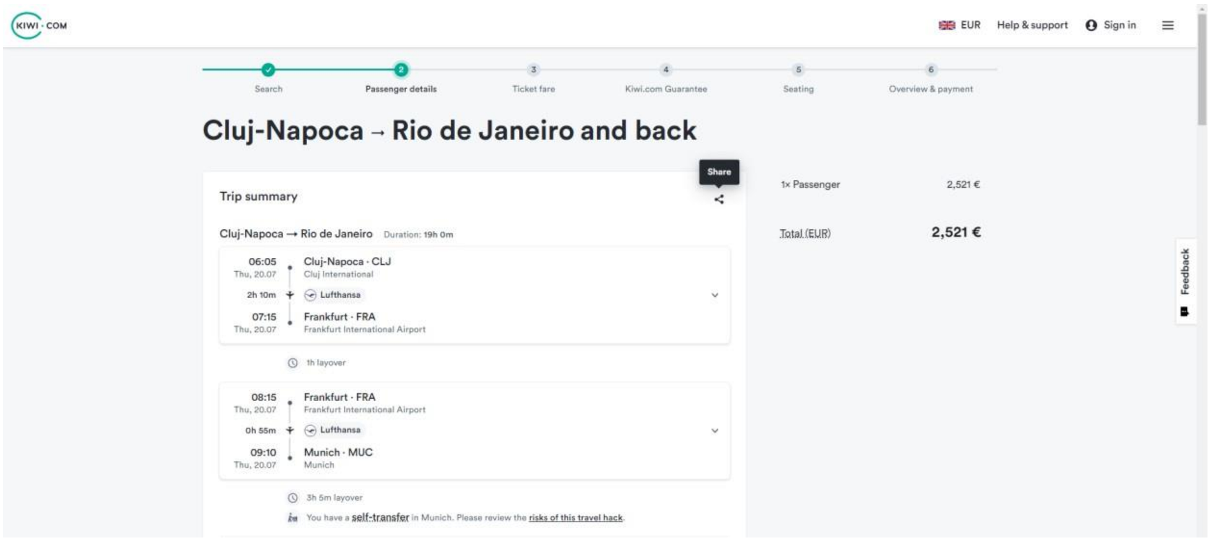
11:20 Mon, 21 Aug — Lisbon (LIS)
Duration: 1h 50m — TAP Portugal
14:10 Mon, 21 Aug — Barcelona (BCN)

2h 50m layover

17:00 Mon, 21 Aug — Barcelona (BCN)
Duration: 3h 0m — Wizz Air Malta
21:00 Mon, 21 Aug — Cluj-Napoca (CLJ)

Figure 37 - Flight Details

- **Book a Flight**: Once a flight is selected, click on the "Book" button. This will redirect you to Kiwi's website for booking completion.



Cluj-Napoca → Rio de Janeiro and back

Trip summary

Cluj-Napoca → Rio de Janeiro Duration: 19h 0m

06:05 Thu, 20.07 Cluj-Napoca · CLJ Cluj International
2h 10m Lufthansa
07:15 Thu, 20.07 Frankfurt · FRA Frankfurt International Airport

1h layover

08:15 Thu, 20.07 Frankfurt · FRA Frankfurt International Airport
0h 55m Lufthansa
09:10 Thu, 20.07 Munich · MUC Munich

3h 5m layover
You have a self-transfer in Munich. Please review the risks of this travel hack.

1× Passenger 2,521 €
Total (EUR) 2,521 €

Figure 38 - Booking page.

This guide should help the user navigate through the application without problems.