Flight Fares Calendar - Technical Challenges

Objective

Provide an overview of challenges that were encountered while developing the application. Also provide a highlight on areas of improvement.

Technical Challenges

1. SkyScanner API

The document - `Assignment - A.pdf` document **description** - *Your task is to find flight **status** for a month* was a bit misleading and led to some confusion while exploring the Skyscanner API. While the intent is not to pin-point typos in the assignment, just stating that there was some confusion because of this.

The RapidAPI site has very little documentation on how to query Flight fares. Although it provides an excellent Swagger like platform to test the APIs, It fails to provide the sequence to query for flight fares.

For example, in order to **Browse Flight Fares**, you dont need to create a session, query response header for session, or poll for data. Instead you can directly call the **Browser Flight Fares** api with your **Rapid API key**. Information on Flight fares were obtained from the official SkyScanner Developer Documentation site

---

2. Calendar which is responsive to be viewed on Mobile Device.

My Aim in completing this assignment, was to ensure that the **UI is both viewable and accessible** on both **Desktop** (large screens) as well as on **Mobile handsets** (small screens). However, this posed a challenge while rendering Calendar on mobile screens. Infact, the **official** Skyscanner website's calendar view to browse flight fare is not responsive enough when viewed on Mobile screens.

I went with React Calendar with the primary aim to support Mobile devices. so as a disadvantage, on Larger screens, it led to difficulty in reading Fares due to the Calendar not adapting to the big screen.

In order to compensate for this, I designed a Flight fares details component which shows up if you click a particular date *(Just saying that this was a UI challenge to support large screens).*

---

3. Library for Dates *vs* MVP Design

On purpose, I did not use any library to process Date / Month data (like: Moment / Timer / Twitter Time SDK) as these libraries are large in size and quite an overkill for the assignment. I instead used an ENUM / Javascript Constant List which has all the dates for each month in string format *(Take a look at **month-details.ts** in the frontend project and you will understand what I mean* 😃*)*

Similarly, I did not use any extra library for form data processing (Formik / Redux Forms) (again the reason being such libraries are an overkill addition for just 3 input fields in the entire application).

I relied on React's core modules like - useState, useReducer & Context API instead of adopting state management libraries like Redux / Flow / Flux (again project size overload)

Areas of Improvement

i. API / Node JS Backend

1. Leverage Typescript on Node JS Backend Server / Use Nest JS

One area of improvement was to also setup Node JS backend server on Typescript. I am currently learning Nest JS which is a nice alternative to Using **Typescript + Express**

2. Better Unit Testing / Code Coverage (CC)

Although both Backend & Frontend projects have unit testing configured and stub test scripts written, given more time I could have shown better unit testing / code coverage

3. Node JS Security / Environment Variables

Usage of Environment Variables to store sensitive information (like - Rapid API key / secret) is a better alternative and provides good security

4. CI / CD Pipeline on Github

Given more time, I would have demonstrated setting up CI / CD pipeline on github

---

ii. React, Typescript, Frontend Application

1. Performance Enhancement (Caching) - Do not load / fetch the XHR data again if already loaded for the selected month

Currently in the application, you can keep hitting the **Find Fares** button repeatedly and the application keeps requesting for the same data from the server. This is obviously not an optimal method.

Caching previously hit XHR requests responses / Using RxJS switch map to cancel current request if User hits the Find fares button again are some of the performance improvements

2. Better Unit Testing / Code Coverage (CC)

Although both Backend & Frontend projects have unit testing configured and stub test scripts written, given more time I could have shown better unit testing / code coverage

3. Helper Module to dynamically generate dates for each month

A Better alternative to `month-details.ts` would be to come up with a class which dynamcially populates dates for a selected month (maybe with / without the help of date libraries). This way all the query dates need not be hard-coded as in `month-details.ts` module.

---

5. Promise.all vs Promise.allSettled challenge

Recently observed (18th & 19th September 2019) - that the Browse Flights API was failing for most of the dates. However, because the `Promise.all()` method would only work *IF ALL REQUESTS PASSED* would mostly end up with no calendar being shown

Solution - Use Promise.allSettled instead

I simple switched the Promise.all() method with the design pattern for Promise.allSettled. Other alternatives was use `core-js` allSettled / `bluebird` allSettled / `q` allSettled libraries.

```
const reflect = (promise: Promise<any>): any => {
    return promise.then(
      (v) => {
        return { status: 'fulfilled', value: v };
      },
      (error) => {
        return { status: 'rejected', reason: error };
      }
    );
};

Promise.all(query.promiseList.map(reflect))
    .then ((results:any[]): void => {
      results.map((result: any): void => {
        if (result.status === 'fulfilled') {
          temp.push(result.value);
        } else {
          console.log('Error fetching data: ', result.reason);
          appContext.addNotification('ERROR', `Error while fetching data`,
'danger', 2000);
        }
      });
      setFlightDisplay(true);
      setCalendarResult(temp);
      appContext.addNotification(`${origin} > ${destination}`, `showing fares for
the month: ${monthNames[new Date(query.sDate).getMonth()]}`, 'success', 5000);
      setCalendarDate(query.sDate);
      appContext.removeNotification(noticeId);
      // console.log('response is: ', result);
    }, error => {
      appContext.removeNotification(noticeId);
      appContext.addNotification('ERROR', `Error while fetching data`, 'danger',
5000);
      console.log(error);
    });
```