Labwork 3

Course	Introduction to Mobile App Design and Development R0334-3007
Name	Richard Zilahi
CourseID	R0334-3007
Course Name	Introduction to Mobile App Design and Development
Module	Module 3 – Labwork-3: Demo Mobile App with Ionic Framework
Date	11.09.2022 – 20:00

Table of contents

Introduction

This was the third assignment in the course, which was mainly focusing on API handling within an application, more specifically an an `Ionic` driven mobile application.

For this assignment, I've decided to write a very basic, dead simple Email client application. It's rather a fake email client application, because there's no proper `MTA` (`Message Transfer Agent`) behind it.

However, as the main aspect of this assignment was different data processing via an `API` interface, i've decided to create my own api for this project, which mocks the most essential email sending features:

- email sending

- email receiving

Local development env

Since there is an `API` which was not the scope of this assignment, but the application still needs to be started for evaluation, you have to do some workarounds.

There is an example `.env` environment file in the root folder of the project, which you have to make a copy of. In order to do that, please execute the following command in the project root:

```
cp .env.example .env
```

Please note, that this command above is in `bash`, if you don't have bash on your computer, or you are on a different `OS` use the method you see fit.

Once you have the `.env` file ready, you have to add the deployed `API` endpoint into the file:

```
PRODUCTION_API=<PROD_API_URL_HAS_TO_COME_HERE>
```

The reason behind this, is that this `API` runs on my personal `AWS` instance, which I am paying the bill for. Since there is no any type of authentication implemented on the `API` , i didn't want to enclose it publicly, to avoid abuse, and potential request number peeks. So this is just for being on the safe side.

To run the application locally, you'll need to have at least the latest `LTS` version of node, which is the moment of writing this documentation is `>=16`. However, the applicatoin was written using `Node 17`.

Implementation

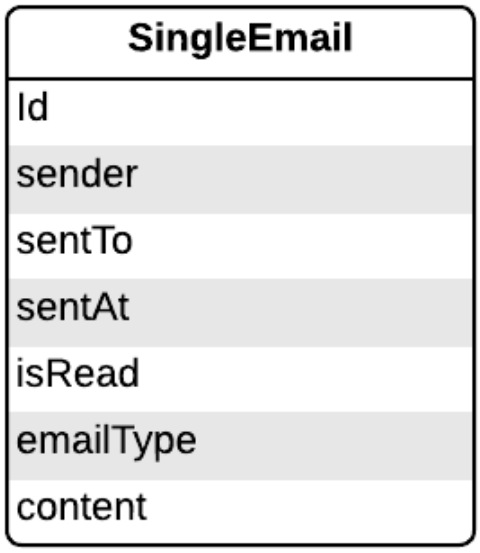
Api

This assignment does't not required to write any `API` on my own, however as I've done it, which is an essential part of the assignment itself, I've decided to document it in this learning reflection document.

The `API` itself implemented in `typescript` using `ExpressJS`. Locally, the application runs on a single instance of `serverles` function, which mocks an actual serverless environment, which is in this case `AWS Lambda`

The `API` stores it's data in a cloud hosted `MongoDB` instance, provided by `Atlas`. It's running on a free trier.

The `MongoDB` data model builds up from the following attributes:



As the model of the data that represent a single email shows, this is *very* far from an actual emailing service, but the goal here is to have a working `API` than can be integrated into the Mobile App, and as my idea was to build an email client, I needed something very simple.

This data model implements the following interface on the backend:

```
interface SingleEmail {
  sender: string;
  sentAt: number;
  content: string;
  isRead: boolean;
  emailType: EmailType;
}
```

Where the `emailType` is a simple `type`:

```
export enum EmailType {
  INCOMING = 'INCOMING',
  OUTGOING = 'OUTGOING',
}
```

The reasoning behind this was to be able to mock incoming and outgoing `API` requests inside the mobile app.

The `API` implements just a handful of routes:

```
GET /all/:type where `type` represents an EmailType
GET /:id where `id` represents an email with it's `ObjectId`
POST /
POST /:id where `id` represent an email with it's `ObjectId`
POST /incoming
```

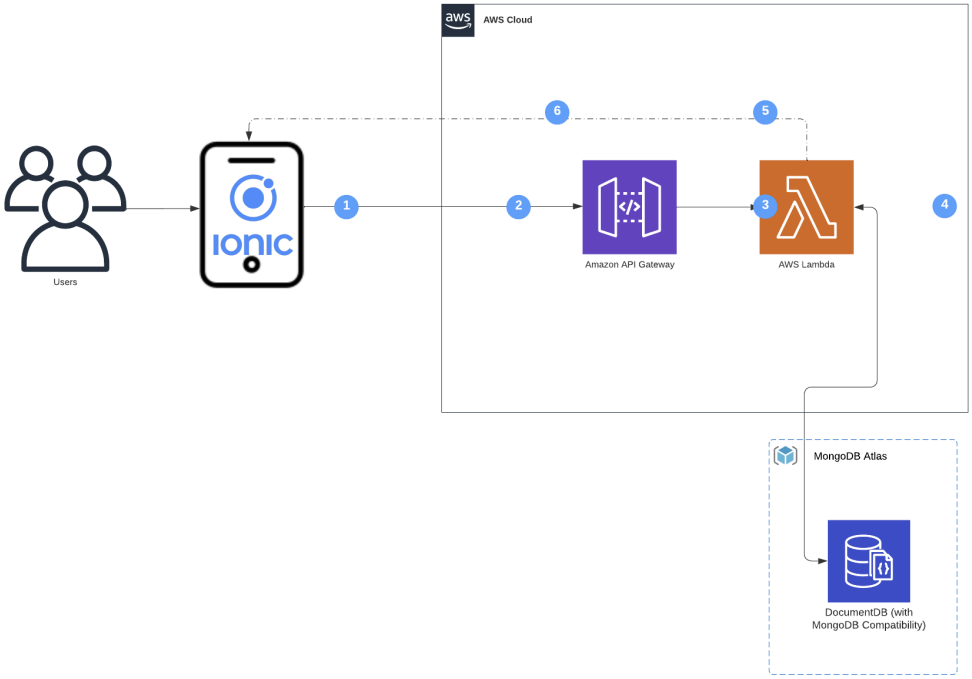
Every endpoint mocks a simple functionality of an email service:

API Endpoints

endpoint	description	method
`/all/:type`	getting all the email belonging to a specific type (either INCOMING, or OUTGOING)	GET
`/:id`	returns an object represented by the id given in the parameters	GET
`/:id`	sets an email as red represented by the id given in the parameters	POST
`/`	sends out a single email, passed in the body of the request	POST
`/incoming`	mocks a request that returns new arrived emails	POST

`API` Architecture

The following diagram vaguely describes the architectural connection between the client `Ionic` mobile app, and the `API`:



▼ Note: If you want to run the backend locally, please refer to the github repository, which I made available to everyone.

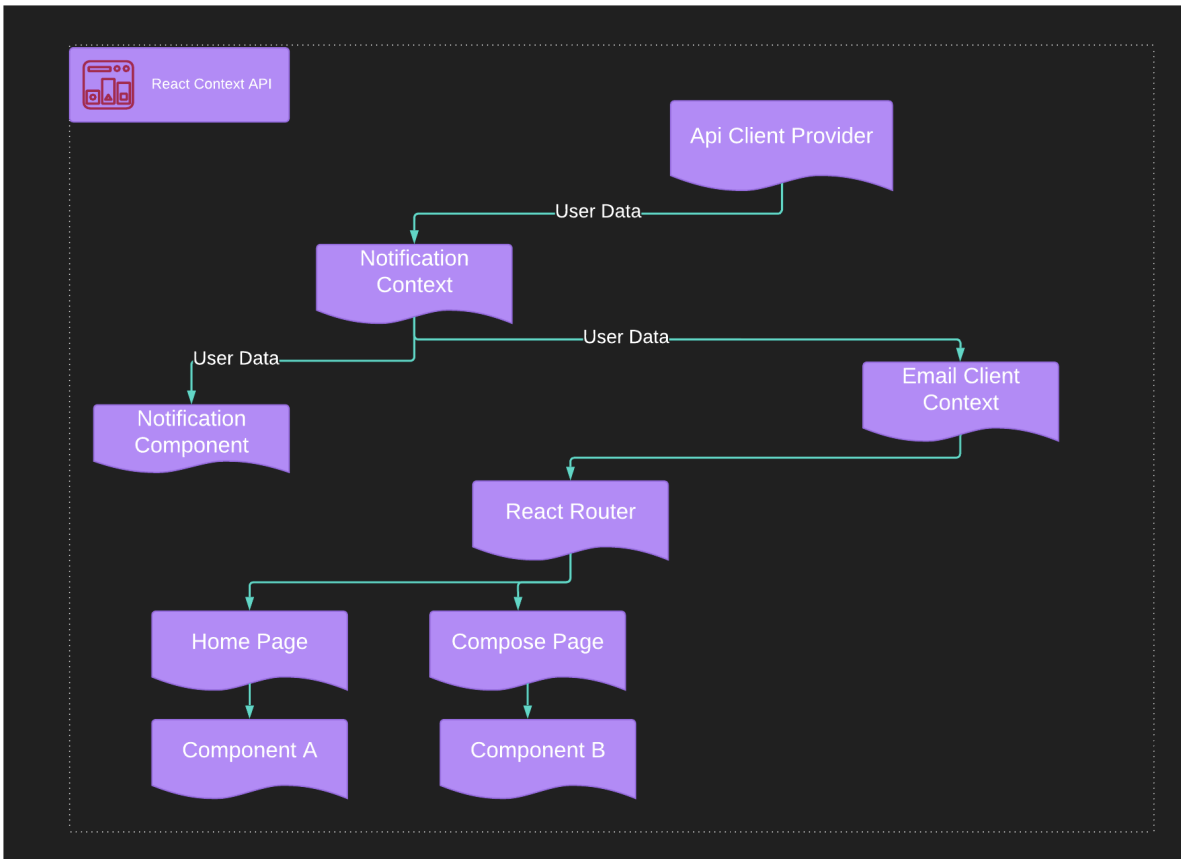
Mobile App

When implementing the application for this assignment i've used an alternate version of the `Data First` developing mindset. First i've designed how the data going to *flow* in the application so I implemented the state management first.

State management

As I am using `React` do avoid any heavy usage of third dependencies, such as `redux`, or `flow`, I've decide to go along with the built-in `Context` of `React`.

The high level `Context` tree looks as the following:



The `Api Client Provider` wraps the entire apps, whcih makes it possible for the `API` data to be passed down to the child components. Before it reaches the two-way data binding `React` philosophy, it flow's through a few other providers, which handles their own functionality.

In `React` this implementation looks as the following:

```
<ApiClientProvider>
      <RootContextProvider>
        <NotificationProvider>
          <EmailClient>
            <App />
          </EmailClient>
        </NotificationProvider>
        </RootContextProvider>
    </ApiClientProvider>
  </React.StrictMode>
```

▼ Reference to the file on github: `src/App.ts` https://github.com/zilahir/R0334-3007_3/blob/master/src/App.tsx

Root Context

The `RootContextProvider` implemens a very simple `React Context` that handles the `HTTP` requests towards the `API`.

The `Context` uses a `React hook` which exposes functions of a getter, and a setter, that stores the emails returned by the `API`

```
const [emails, setEmails] = useState<SingleEmail[]>([])
```

And an other pair of getter and setter which are holding data when composing a new email:

```
const [newEmail, composeNewEmail] = useState<NewEmail>({} as NewEmail)
```

▼ For reference the file on github can be found at: `src/src/api/context/index.tsx`:
https://github.com/zilahir/R0334-3007_3/blob/master/src/api/context/index.tsx

Notification Context

Following the logic mentioned above, the `Notification Context` implements some methods handling notifications related features:

- dispatching an action which creates a new notificaion
- dispatching an action which removes a notification by id

```
function addNewNotificaion(newNotification: SingleNotification): void {
        setNotifications([...notifications, newNotification])
    }
```

A notification implements the following interface:

```
interface SingleNotification {
    message: string,
    severity: Severity,
    timestamp: number,
}
```

Where `Severity` is takes up the following type:

```
type Severity = "success" | "danger"
```

As it's visible in the interface, the timestamp behaves as the `id` of the `Notificaion` which can be then later used, to remove them:

```
function removeNotificationByTimeStamp(timeStampToRemove: number): void {
        const filtered = notifications.filter(({ timestamp }) =>  timestamp !== timeStampToRemove);
        setNotifications(filtered);
    }
```

▼ For reference the file on: `src/components/common/Notification/context/index.tsx`: https://github.com/zilahir/R0334-3007_3/blob/master/src/components/common/Notification/context/index.tsx

Email Context

The most important context this app implements is the `Email Context`. The email context contains the functions that mocks the features of an actual Email client.

It fetches the emails from the `API` stores them in the context's values, and makes it available to it's child components.

The actual `API` calls are implemented in a reusable `React hook`. This is a function, that implements a simple `React state` using the `useState` hook for handling a boolean state for

waiting the `API` call to be full-filled:

```
const [isLoading, toggleLoading] = useState<boolean>(false);
```

Besides that, there's a couple of exposed functions as well, which are handling the `HTTP` requests themselves:

```
GET: async function getEmail()
POST: asyn function sendEmail()
```

The `EmailContext` itself implements two `React useState` methods, for getting and storing the email returned by the `API`,

```
const [emails, setEmails] = useState<SingleEmail[]>([])
const [newEmail, composeNewEmail] = useState<NewEmail>({} as NewEmail)
```

Then, lower in the component tree combining the `useEmail` hook, together with the `EmailContext`, we can orchestrate when the `API` should be called, and the stored list of emails should be updated:

```
const { getEmail, randomIncomingEmail } = useEmail()
const { setEmails } = useContext(EmailContext)
```

▼ For reference, the file containing the implementation of the `Email Context` can be found on github at `src/api/context/index.tsx`: https://github.com/zilahir/R0334-3007_3/blob/master/src/api/context/index.tsx

▼ For reference, the file containing the implementation of the `useEmail` hook can be found on github at: `src/hooks/useEmail.ts`: https://github.com/zilahir/R0334-3007_3/blob/master/src/hooks/useEmail.ts

React useQuery

The `API` calls are implemented using the `react-query` which is a rich and powerful asynchronous state management engine. For reference, you can find the documentation of `react-query` behind the following URL: https://tanstack.com/query/v4

To make the `HTTP` request, I was using `axios`. Axios is a *promisified* `HTTP Client`. For reference, you can find the documentation of `axios` behind the following URL: https://github.com/axios/axios

`Axios` provides an `API` to create a reusable instance of `axios` pointing to the root `URL` of the backend:

```
const apiClient = axios.create({
    baseURL: API_ROOT["dev"],
    withCredentials: false,
})
```

Combining `react-query` with axios makes the following implementation in `React`:

```
useQuery(["getAllEmail"], getEmailRequest, {
    enabled: true,
    retry: false,
    onSuccess: (data => setEmails(data)),
})
```

Where the `getEmailRequest` is a function, which returns an `AxiosPromise`, which then resolves with the `Response` coming from the `API`.