Connecting the Backend and Frontend Using the Axios

In this worksheet we will retrieve data from our running API and also send data to it to update what is stored in the database. To do this we will employ the **Axios API** implemented in modern browsers.

Our tasks and goals in the worksheet include the following:

1. Connect landing page to retrieve all articles from the Backend API.
2. Retrieve individual post article to render in the main grid on the landing page using fetch.
3. Set up Cross Origin Resource Sharing (CORS) management on the API
   - This is needed to enable browser clients to receive data from the API endpoints (it was not needed for the Postman client that we used before).
4. Enhance the UX in the main page.
5. Add data validation to the form.
   - We wish to check things such as whether the email is valid etc.
6. Set up custom registration submission

Axios

As different components in the SPA connecting with the same Backend API, it is suggested to create a common folder to handle the base URL of the API and the headers. In the `src` folder, create a common folder and add a need `http-common.ts` within this folder. The source code is shown as follow:

```
export const api = {
  uri: "https://..." Your API Link
}
```

You can restore the API site which you have done in previous laboratories and update the link above.

Now, head back to the `article.tsx`, update the source code as follow:

```
import React from 'react';
import { Link } from 'react-router-dom';
import { Card, Col, Row } from 'antd';
import { api } from './common/http-common';
import axios from 'axios';

const Article = () => {
  const [articles, setArticles] = React.useState(null)
  React.useEffect(()=> {
    axios.get(`${api.uri}/articles`)
    .then((res)=>{
      setArticles(res.data)
    })
  }, [])

  if(!articles){
    return (
      <div>There is no article available now.</div>
    )
  } else {
    return (
      <Row justify="space-around">
      {
        articles &&
          articles.map(({id, title, alltext})=> (
            <Col span={8} key={id}>
              <Card  title={title} style={{ width: 300 }} bordered={true}>
                <p>{alltext}</p>
                <p></p>
                <Link to={`/a/${id}`}>Details</Link>
              </Card>
            </Col>))
      }
      </Row>);
  }
}
export default Article;
```

For the request with basic authorization, you can try with follow:

```
axios.get('https://api.github.com/user', {
  headers: {
    'Authorization': `Basic ${access_token}`
  }
})
```

Hence, the token should be converted to base-64 format as follow:

```
const access_token = Buffer.from(`${username}:${password}`,
'utf8').toString('base64')
```

Set up Cross Origin Resource Sharing on the API

Unless you have an unusual development environment setup, or have previously set up CORS on the API, then the above preview will fail. Check the console for details of the error, which will mention the CORS response.

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the
remote resource at …/api/v1/articles. (Reason: CORS header ‘Access-Control-
Allow-Origin’ missing). Status code: 200.
```

To add CORS functionality to the API first install `@koa/cors` as follows in the backend project environment:

```
$ npm install @koa/cors
```

Then in the main `index.ts` application file, include the library:

```
import cors from '@koa/cors';
```

To use it, after the import section add the imported CORS middleware to the application:

```
app.use(cors());
```

This suffices, without changing the defaults, for getting the API requests to work from the SPA in the browser.

Now, test the SPA again. However, you may find out that you cannot redirect to the details article component.

Worksheet Task 1

1. Based on the main article, try to update the detail articles component to show the detail of each article.
2. Try to implement a 404 Not Found component and redirect all illegal access route to this component.

Improve User Experience

Once you have complete both tasks, you may find that some unusual return display if the Backend response is not ready. In Ant Design, there are some “animated” components that can let the client know the data is currently loading from “somewhere” and preparing.

```
import React from 'react';
import { Link } from 'react-router-dom';
import { Card, Col, Row, Spin } from 'antd';
import axios from 'axios';
import http from '../common/http-common';
import { LoadingOutlined } from '@ant-design/icons';


const Article= () => {
  const [loading, setLoading] = React.useState(true);
  const [articles, setArticles] = React.useState(null)
  React.useEffect(()=> {
    axios.get(`${api.uri}/articles`)
    .then((res: any)=>{
      setArticles(res.data)
    }).then(()=>{setLoading(false)})
  }, [])

  if(loading){
    const antIcon = <LoadingOutlined style={{ fontSize: 48 }} spin />;
    return (<Spin indicator={antIcon} />);
  } else {
    if(!articles){
      return (
        <div>There is no article available now.</div>
      )
    } else {
      return (
        <Row justify="space-around">
        {
          articles &&
            articles.map(({id, title, alltext})=> (
              <Col span={8} key={id}>
                <Card  title={title} style={{ width: 300 }} bordered={true}>
                  <p>{alltext}</p>
                  <p></p>
                  <Link to={`/a/${id}`}>Details</Link>
                </Card>
              </Col>))
        }
        </Row>);
    }
  }
}
export default Article;
```

Set up a new article form

Recall that our API offers a public endpoint (no authentication required) at `/api/v1/users` for POSTing new user data. This is therefore a URI that will allow user registration. Our first step is to create a form that will capture the relevant data from the user in the SPA.

Basic form page

Complete the following steps to create a new article page.

- Create a new file `newarticle.tsx` in the components folder for the new component.
- Put in some placeholder code that renders `<h1>New Article</h1>` and export the component.
- Update `App.tsx` to add a `react-router-dom <Route>` component matching the path `/newarticle`. This should conditionally render the new placeholder component.
- Add a link to the new route in the nav bar component.
- Run your SPA and check that the page link in the nav is working and that you see the (empty) registration page rendered.

You can now add the form for adding an article. We begin by just adding the required fields. Form and form items are imported from Ant Design for convenience. At this stage the `<NewArticle>` component will just render the form layout:

```
import React from "react";
import { Form, Input, Button } from 'antd';

import { Form, Input, Button } from 'antd';
```

```
const { TextArea } = Input

const NewArticle = () => {
  return (
    <Form name="article">
      <Form.Item name="title" label="Title" >
        <Input />
      </Form.Item>
      <Form.Item name="context" label="Context" >
        <TextArea rows={4} />
      </Form.Item>
      <Form.Item>
        <Button type="primary" htmlType="submit">Submit</Button>
      </Form.Item>
    </Form>);
}

export default NewArticle;
```

Check that this works by previewing the page in the browser.

Add a submission handler

To get useful data from the form we need a handler. You will see in the <Form> API in the Ant Design docs that a suitable prop is available called `onFinish`.

- Define an `onFinish` method in your `<NewArticle>` class that logs its first argument to the browser console.
- Add the handler as the value of the prop `onFinish` to the `<Form>` component.

Sample script as below:

```
...

const { TextArea } = Input

const NewArticle = () => {
  const handleFormSubmit = (values: any) =>{
    const title = values.title;
    const context = values.context;
    console.log(values, title, context);
  }

  return (
    <Form name="article" onFinish={(values)=>handleFormSubmit(values)}>
      <Form.Item name="title" label="Title" >

...
```

Now when you preview the screen, also open the developer tools and load the console tab. Put some values in the fields and click on the 'Register' button. You should see the field values logged to the console.

Add form validation and UI feedback

By reusing Ant Design form components, we gain the ability to add props for data validation. The validation should generally match the rules in effect on the API that will ultimately receive the data. This will avoid having to call the API with data that fails validation (although the API validation is still essential for the integrity of the API data).

Add the following rule definitions after the layout definitions.

```
const contentRules = [
    {required: true, message: 'Please input a context' }
];
```

As you see, by reusing Ant Design components we get a lot of functionality for free. For basic validation such as being required, string pattern format, length of input, etc. there are pre-defined properties we can use in the rules such as `required:`, `type:` and so on. For custom validation we can pass in a `callback` function which returns an object containing a `validator callback` method. In it we can define any custom logic for validation we need.

- To implement the validation on the form, add a prop called `rules` to each of the `<Form.Item>` components that contains an `<Input>`, and set the prop's value to be the corresponding validation rule from above. For example, the title form item becomes `<Form.Item name="title" label="Title" rules={contentRules} >`
- Finally add the prop `scrollToFirstError` to the container `<Form>` component. This tells the component to scroll to fields with validation errors when the viewport is too small to show the whole form.

Now preview the form again and notice that the validation rules trigger UI feedback when the rules are not satisfied. The form components gain warning message text and, in the case of the password fields, icons to inform about the validation failures.

Worksheet Task 2

1. Try to implement with `axios` POST request to complete the create user.
2. Try to implement a login component and login authorization.

Hints for using basic authorization:

1. Install `@types/node` in your project;
2. Create an access token with username and password, separate with a colon and convert to `Base64;`
3. Add an Authorization header with the token;

Sample as shown below (with using GET request):

```
const access_token = Buffer.from(`${username}:${password}`,
'utf8').toString('base64');

axios.get('https://api.github.com/user', {
   headers: {
      'Authorization': `Basic ${access_token}`
   }
})
```