IndEAA

Streamlining course review processs

Frinze Erin Lapuz and Michael Nefiodovas

System Health Lab

Table of contents

1. IndEAA Documentation

Welcome to the IndEAA documentation!

2. Coordinator

2.1 Introduction

3. Reviewer

3.1 Reviewer Documentation

This is the documentation for reviewers detailing how to complete a review.

3.2 Quickstart

1. Login with your google account. Click the G+ icon
2. At the top right (in the navigation bar of the website), click on the "Choose your role" dropdown and select "Reviewer". Select "reviewer" role from dropdown in top right
3. You should be automatically redirected to the reviews overview page where you can see all/any courses your account has been assigned to review. Image of the "Review Courses" page which shows the different reviews the user has permission to view
4. Click on the review you wish to begin or continue, you should be redirected to the page where you can complete your review for the particular course you selected. Image of the specific review page for the course MECH1234
5. A review is divided into 4 stages: "Overview & Eoc", "Read Documents", "Review Course" and "Review & Submit", it is not necessary that you do these in order, however, once you submit you will lose the ability to edit your review.
6. During the first stage, "Overview & Eoc", you are expected to familiarise yourself with the different element of competencies (EOCs) for this course, you can view the descriptions of each EOC by expanding the dropdowns and then selecting the specific EOC you wish to view. EOC 1: Knowledge and Skill Base expanded and EOC 1.6 is selected
7. Click the "Next" button to proceed to the second stage, "Read Documents". The Read Documents page
8. In this stage you should view the different documents provided to you by the course's coordinator(s). You can view documents by clicking on the "View" which will open the document in a new tab. To keep track of documents you have already viewed, you can click the "Mark as viewed" button. Once you've looked through the document, clicking the "Add Comment" button will allow you to enter a written comment for the document. Adding a comment to a document Documents page after writing some comments
9. Click the "Next" button to proceed to the third stage, "Review Course". Review Course page
10. The "Review Course" stage is where you will complete your actual review of the course. You are expected to provide a comment, rating and ideas for improvement for each EOC. To begin your review of a particular EOC, you should click on the "View" button for the EOC you're interested in. Review Course page viewing some uncompleted eocs
11. After clicking "View" you should see a window pop up with different information corresponding to the EOC and some fields you are able to fill in. This window is broken up into 3 major sections: The coordinator's review (top left), your review (bottom left) and the documents which the coordinator has labelled as applicable for this EOC (right). Review popup showing the Justification of Coordinators, Documents section and Reviewer rating area
12. The documents section is divided into two parts: "General Documents" and "Specific Documents". General documents are assigned to an entire class of EOCs and the coordinator believes that these are appropriate for a large range of EOCs. Specific Documents have been selected as being directly applicable to this EOC and may be more directly relevant to your review. Review popup with the specific documents tab selected
13. At any time you may click "Save" to save your progress or "Cancel" to lose your unsaved progress.
14. Click the "Next" button to proceed to the final stage, "Review & Submit".
15. In this stage is divided into two main parts: an overview of everything you've completed so far so that you can double-check if you've missed any reviews or steps in the process. If you scroll down to the bottom of the page, there is a text box where you should enter your final general comment which summarises your impression of the course in totality (Make sure to click "Save" to ensure your general comment is saved). Review and Submit page
16. Once you have double-checked your review and provided your general comment you can choose to submit by clicking "Submit" **submitting revokes your edit access for this course**, so make sure you've added everything you want (you may contact your coordinators to reinstate your edit access manually). Are you sure? Dialog after pressing the submit button

4. Administrator

4.1 Introduction

5. Developer

5.1 Developer Documentation

This is the documentation for developer used to maintain this software.

5.1.1 Tech Stack

Frontend

CORE FRONTEND TECHNOLOGIES

ReactJS

React documentation
React is a UI library in JavaScript that utilizes the power of Client-Side Rendering (CSR) for Single Page Application (SPA). It makes development easier by using the concept of component modularisation.

NextJS

Nextjs documentation
Nextjs is a library on top of React that a lot of extra features for React. However, it may have some difficulty when it comes to server-side rendering. Hence, will be under consideration.

COMPONENT LIBRARIES

Material-UI

Material-UI documentation
Material-UI is a component library for React which provides a large number of pre-made components styled in accordance with the material design standard. This allows for rapid development as you don't have to focus on design considerations as much.

Material Kit React

Material Kit React documentation
Material Kit React is built on top of Material UI and provides extra theming for components. This will further accelerate development and provide a consistent look and feel to the design.

Frontend + Backend Integration - Feathers-Redux

Redux documentation
Redux is a state management library for React that handles high-level state management that is separate from the structure

Feathers-redux documentation
IndEAA specific feathers-redux information
Feathers-redux is a library that maps the Feathers API to Redux reducers to abstract base HTTP calls to function calls in order to sync the state of the frontend, and the data stored in the backend.

Backend - Feathersjs/Express

Express documentation
Express is a library on Nodejs (a platform to run JavaScript outside of the browser) to easily create HTTP servers.

Feathersjs documentation
Feathersjs is a wrapper for application in Express, essentially to make development easier by loosely enforcing developers on creating RESTFul API (following uniformity of HTTP methods). It makes development easier by making route to database call uniform.

Database - Mongoose/MongoDB

MongoDB documentation
MongoDB will be used for its loosely defined and flexible data structure.

Mongoose documentation
Mongoose will be used to enforce some schema to control the shape of the data.

Deployment - Docker

Docker documentation
Docker will be used to orchestrate the frontend and the backend into two main containers as well as to host a temporary database for development.

https://stackoverflow.com/questions/51011552/mongodb-on-with-docker-failed-to-connect-to-server-localhost27017-on-first-c

5.2 Coding Standards

The IndEAA is bootstrapped by two developers working remotely. Along with the development, due to the difference of coding standards and background, this documentation has been established for better communication while working remotely.

Leniency

This coding standard is established for the purpose of more fluid communication. However, this is followed for the usual governance of the codebase, but can be overruled given a reason.

5.2.1 Import Order

It is important to have an order of import for code readability. There is a specific order of package, and if in the same category, it is arranged alphabetically. Each category is divided a space.

Frontend Import Order

The order is:

1. React + Redux + Functionality Imports
2. Own Components Import
3. Utilities Import
4. Material Kit import
5. Material Ui Import
6. Icons
7. Styles Import (both Material UI usestyles, style imports and style definition)

Absolute Import

It is recommended to use absolute import as it makes it easier to compare source code file imports. The frontend is configured to use absolute import from the root folder `client`. This is not the same for `server` at the moment.

Frontend Import Order Example

```
// This is reordered of administrator/index.js

import { useEffect, useState } from "react"
import { useDispatch, useSelector } from "react-redux"
import { services } from "store/feathersClient"

// Own Components
import UserModal from "components/administrator/UserModal"
import CreateUserModal from "components/administrator/CreateUserModal"

// Helper
import { getAvailablePermissionsOfUser, roleIcons } from "utils"

// Material Kit
import Card from "components/MaterialKit/Card/Card.js";
import CardBody from "components/MaterialKit/Card/CardBody.js";
import CardHeader from "components/MaterialKit/Card/CardHeader.js";
import Button from "components/MaterialKit/CustomButtons/Button.js";
import Grid from "components/MaterialKit/Grid/GridContainer.js";
import GridItem from "components/MaterialKit/Grid/GridItem.js";

// Material UI
import Tooltip from "@material-ui/core/Tooltip";

// Icons
import Placeholder from "@material-ui/icons/Mood";

//Styles
import { makeStyles } from "@material-ui/core/styles";
import styles from "assets/jss/nextjs-material-kit/pages/landingPage";
const useStyles = makeStyles(styles);
```

Backend Import Order

The order is:

1. NPM imports
2. Project Imports

5.2.2 Naming of Variables, Files and Database

This is no different to usual coding standards, but it is worth noting to put name of variable as sensible names.

General

Consistent Variable Names

Consistency of variable names for variables that refers to the same property, but different context.

Data from the store and data from the form state

Good

```
 useEffect(() => {
    setCourseId(courseData?.courseId || '');
    setReviewDescription(courseData?.reviewDescription || '' );
    setDueDate(courseData?.dueDate || '');
}, [courseData]);
```

Another Good one

This uses a single `formState` instead of separating.

```
 useEffect(() => {
    setFormState({
        courseId: courseData?.courseId || '',
        reviewDescription: courseData?.reviewDescription || '',
        dueDate: courseData?.dueDate || '',
    })
}, [courseData]);
```

Bad

```
 useEffect(() => {
    setCode(courseData?.courseId || '');
    setDescription(courseData?.reviewDescription || '' );
    setDueDate(courseData?.dueDate || '');
}, [courseData]);
```

This is part of the Evaluation Modal where a form state is declared and initialised with values from the data store.

If you look at the bad example, you will notice these: - what is the difference of terminology between `code` and `courseId` - between `description` and `reviewDescription`, there isn't that much of a problem. However, that begs the question, why change the first terminology in the first place? - the advantage that you get if you don't change terminology is that you can use the usual JavaScript `...` operator due to same name.

If you see a bad naming, but is consistent, you have 2 choices: - replace the entire reference of the bad name - use the bad name, and replace it later with find and replace.

This is the main power of being consistent is that you can always tie it to other references, although with different naming, you still can but you have to do it manually.

Consistent File Naming

Make sure that the name of the default function is the same name as the filename. If it is not the same name, have a good reason.

Filename consistency

Good

```
// AreYouSureButton.js
export default function AreYouSureButton({...})
```

Bad

```
// AreYouSureButton.js
export default function Modal({...})
```

Frontend Custom Hooks

All custom hooks should have `use` as prefix. Refer to the original documentation for the source of this coding standard.

Frontend Props

Usually there are 2 main reason as to why a react component is divided into 2 reasons: readability - by splitting the code into subcomponents, and reusability - to be able to reuse components.

GENERIC REUSED COMPONENT

For reuse component, always use generic variable name in the component, but the props passing should solve this.

Props with different name to the variable value it was assigned to

```
// AreYouSureButton.js
export default function AreYouSureButton({
    title = 'Are you sure?',
    description = 'Are you sure you want to do this?',
    action,
    actionParameter = [],
    buttonProps = {},
    children,
}) { ... }

// Some other file that will use this
<AreYouSureButton
    buttonProps={{}}
    description={
        'You are about to submit a review. Upon submission of a review, you will lose the ability to edit your review. If you have to edit a review, you will
have to contact the coordinator of this unit.'
    }
    action={handleSubmit}
>
Submit
</AreYouSureButton>
```

As you can see from this one, `action` is just the action to be executed by the modal when a the confirm button is pressed. However, the function name that was passed to a prop is called `handleSubmit`. The reason why `action` does not have to be named `handleSubmit` because you could have times where you might want to pass `handleDelete` or something else.

SPECIFIC CONTEXT COMPONENT

There are times where context is very important or when components are only divided for readability, but not for the purpose of reusability.

Props with Almost same name to the variable it was assigned to

```
const DocumentViewer = ({
    documents = [],
    course_id = null,
    review_id = null,
    eocBeingViewed = null,
    isReviewer = false,
    isReadOnly = false,
}) => {...}

// Being used in another file
<DocumentViewer
    course_id={course?._id}
    review_id={review?._id}
    documents={course?.documents}
    eocBeingViewed={eocGeneralAndSpecific}
    isReviewer
    isReadOnly={isReadOnly}
/>
```

As you can see from here, the name of the props being passed is almost the same name as the variable it was being assigned to.

OVERSPLITTING OR TOO DRY

As much as you want to have "DRY" (Dont Repeat Yourself) code, if it interferes with the readability of the code and adds more line of code instead of reducing, then it is not worth even splitting it.

In majority of cases, splitting is correctly, but be mindful especially because once you split the code, it may become less flexible.

Oversimplified Code

Let say I have a function that needs to be executed twice for two different context.

```
const someFunction = (requiredParam, optionalParam="optional") => ... // Maybe DB operation

// Correct way
someFunction("some value1")
someFunction("some value2","some options")

// Overly simplified code
[
    {requiredParam: "some value1"},
    {requiredParam: "some value2", optionalParam:"some options"}
].forEach(({requiredParam,optionalParam})=> someFunction(requiredParam,optionalParam) )
```

At this example, the "correct way" has been executed twice, but the code that follows "DRY" has more lines of code and is more complex. The only time the "DRY" code is accepted is if you need to call this function, enormous amount of times (maybe around 4 or more times. Use with discretion).

UTILS

If you have functions that you think are too generic, but it can be used for many situation depending on context. Put it in the `utils` folder.

_ID SUFFIX

`_id` suffix is a standard of Mongo Object ID. This codebase uses Pascal Case for all cases, so using `_id` is an exception of the rule which indicates that this is a mongo Object ID.

urseID and course_id

This is one of the pitfalls of coding standard that needs to be resolved. Currently `courseID` means differently in different context:

- routing - this corresponds to `course_id`
- anywhere else - this corresponds to `courseID` the unit code of the course.

Furthermore, `_id` fields in the backend are treated as objects. So, if you need to compare IDs, refer here.

5.3 Data Engineering

This covers the section all about data in the system: how it flows from and out of the system to the users, and how the data is stored. This dictates fundamentally how the system would work.

See diagram below for illustration.

5.3.1 Data Flow Diagram (DFD)

This illustrates the processes within the system and how data is converted from one form to another, as well as which data store a data will be located in..

5.3.2 Entity Relationship Diagram (ERD)

This illustrates how the data is structured with relation to one another.

5.3.3 Diagram

5.4 Database Support Scripts

These are scripts that helps in the development whenever data is involved and needs to be filled in the database.

This can be found under `server/support/*`

5.4.1 How to use?

Connect to the docker container

Connect to the docker container by typing this command

```
docker exec -it indeaa_server bash
```

What does this command do?

This executes a bash script in the docker container that can receive commands

Execute Script

When you are inside the docker container, use the scripts by the following command

```
node support/[scriptName]
```

eg.

```
node support/insertSampleCourses.js
```

support/index.js

This file will execute all the sample scripts. For ease of use of setup for development.

```
node support/index.js
```

5.4.2 Adding more scripts

1. Create your own script
2. Import the script to `index.js` by adding this line

```
_ = require("./SCRIPTNAME")
```

5.5 Known Issues

This section is for documenting known bugs developers may encounter and any temporary workarounds found.

5.5.1 Can't log in after deleting user records from database

Error message in terminal

```
server_1  | error: NotFound: No record found for id '6110af36a195d000504258e4'
server_1  |     at new NotFound (/app_code/node_modules/@feathersjs/errors/lib/index.js:114:17)
server_1  |     at /app_code/node_modules/feathers-mongoose/lib/service.js:147:17
server_1  |     at processTicksAndRejections (internal/process/task_queues.js:97:5)
```

Workaround

Clear your browser's local storage for the IndEAA site. Guide on how to do this: Google Chrome, Firefox

You can also use the site in a private browser session for a temporary fix.

Causes

When working with IndEAA it's possible to delete a user's account from the database while they are logged in (e.g. using a tool such as Mongodb Compass).

When this user is logged out they will experience this bug.

The user's session stores their account's `_id` field in local storage (even when logged out).
When a user attempts to log in with a given Google account, the browser will report the `_id` field which was in local storage. However, because the account was deleted from the database, the `_id` field will find no matches. This causes the error listed above.

5.6 Frontend

5.6.1 Component vs Page

In NextJS, you could either have a "Page" component that is loaded as the top-level component that also has special properties such as server-side initial data (not currrently used).

Terminology

In this documentation page, "component" is referred to as component that is loaded from another piece of code. A "Page" is a component that is loaded as the top-level component. It also refers to as a page in a web application.

What should be the difference in code?

PAGES WILL HANDLE ANY LOADING

It is often that when a page load, data loads, especially if that data is used in multiple places. Hence this is the component that is responsible for loading data. This also makes sense since for optimisation purpose, you don't want to hit the server multiple times for the same data because there's limited endpoint.

Coordinator Page

The intended design of the coordinator page should have been separated on each tab. However, this is not the case and should be changed. (TODO)

5.6.2 Transport / Integration Layer - Feathers-Redux

This is the technology used to sync/transfer data from client to server by abstracting HTTP calls to function calls.

Keywords:

- Dispatch - the action to modify the state/store
- Reducers - group of functions that has a determines a specific action to modify state
- State/Store - state/store management structure for data that is accessible through a (Data) Provider

Typical File Structure

This shows the typical file structure in the client folder.

```
client
├── middleware # contains all about middleware on data transport. Code running before main execution of transport
├── actions # contains all function regarding on functions that modifies the state/store by dispatching a change of state (reducer)
├── reducers # contains all function regarding on functions that modifies the state/store (called by action)
├── store # contains all about middleware on data transport. Code running before main execution of transport (creates store with reducer and middlewares)
│
├── feathersClient.js # This can be embedded in `app.js`, but this contains connection parameters to integrate Feathers with redux
```

Redux Developer Tools

Whenever working with this, ensure that you have the Redux Developer tools installed which will ultimately help in debugging state related problems.

Setup codes

```javascript
// ======= reducers/index.j s=======
// This is where you will combine both custom and feathers reducers

import { combineReducers } from 'redux';

export default function (reduxifiedServices) {
  return combineReducers({
    users: reduxifiedServices.users.reducer,
    todo: reduxifiedServices.todo.reducer
  });
}

// ======= middleware/index.js =======
// This is a configuration for all the middleware for Redux

import reduxThunk from 'redux-thunk';
import reduxPromiseMiddleware from 'redux-promise-middleware';
// import { routerMiddleware } from 'react-router-redux';
// import { browserHistory } from 'react-router';
import logger from './logger';

export default [
  reduxThunk, // Thunk middleware for Redux
  reduxPromiseMiddleware, // Resolve, reject promises with conditional optimistic updates
  // routerMiddleware(browserHistory), // !! IMPORTANT for location.href changes
  logger, // A basic middleware logger
];

// ======= store/index.js =======
// This is where both middleware and reducers are combined together to be created

import { createStore, applyMiddleware } from 'redux';
import rootReducer from '../reducers';
import middlewares from '../middleware';

export default function configureStore(reduxifiedServices, initialState) {
    // Engage the Chrome extension "Redux DevTools" if it is installed on the browser.
    // This extension watches reducers and logs their invocations, actions and changing state.
    // It caches activity so you can 'time travel' through state changes.
    // It runs in an extension reducing the size of your app bundle.
    // This interface can be left in prod bundles and the extension activated in the field as needed.
    // https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?utm_source=chrome-app-launcher-info-dialog
    const createStoreWithDevTools = (typeof window !== 'undefined' && window.devToolsExtension)
        ? window.devToolsExtension()(createStore)
        : createStore

    const createStoreWithMiddlewares = applyMiddleware(...middlewares)(createStoreWithDevTools);

    return createStoreWithMiddlewares(rootReducer(reduxifiedServices), initialState);
}

// ======= feathersClient.js =======
// This is where feathers connection is establish
// This is also where the store is created

import configureStore from "../store"
import io from "socket.io-client"
import feathers from '@feathersjs/feathers';
import socketio from '@feathersjs/socketio-client';
import Realtime from "feathers-offline-realtime"
import reduxifyServices, { getServicesStatus } from "feathers-redux"

// Configure Socket and Feathers Connection
export const socket = io(process.env.REACT_APP_BACKEND_URL});
export const feathersClient = feathers()
    .configure(socketio(socket));

// Configure Redux
export const services = reduxifyServices(feathersClient, ['users', 'message']); // Replace the array with the services name
const store = configureStore(services);
export default store;

// Configure realtime & connect it to services
const messages = feathersClient.service('/messages'); // Replace this with a specific service to configure realtime connection
const messagesRealtime = new Realtime(messages, { sort: Realtime.sort('text') });

messagesRealtime.on('events', (records, last) => {
  store.dispatch(services.messages.store({ connected: messagesRealtime.connected, last, records }));
});

// Enable realtime. It will start with a snapshot.
messagesRealtime.connect()
    .then(() => console.log('Realtime replication started'));
```

Realtime Feathers Update Configuration

With Feathers, it could connect with various methods such as sockets. With sockets, it enables the update of the store in realtime. See below for the configuration. More information can be seen in the feathers-redux documentation.

Feathers Realtime Update Configuration

```
// feathersClient.js
// Configure Redux with Feathers
export const serviceNames = [
    'users',
    'course-evaluation'
]
export const rawServices = reduxifyServices(feathersClient, serviceNames,{
    idField: "_id", // This is to ensure that realtime update matching uses that attribute
});

// Realtime Feathers Update Confguration
serviceNames.forEach(serviceName=>{
    const currentSelectedService = feathersClient.service(`/${serviceName}`)

    currentSelectedService.on('created', (data) => {
        store.dispatch(rawServices[serviceName].onCreated(data));
    })
    currentSelectedService.on('updated', (data) => {
        store.dispatch(rawServices[serviceName].onUpdated(data));
    })
    currentSelectedService.on('patched', (data) => {
        store.dispatch(rawServices[serviceName].onPatched(data));
    })
    currentSelectedService.on('removed', (data) => {
        store.dispatch(rawServices[serviceName].onRemoved(data));
    })
})
```

Pay attention, that this configuration allows realtime update for each of the services. If you don't need for each services, you can change `serviceNames` to a list of services that you would want to receive update.

Using Feathers Query and Actions

There are two types of services that can be fetched from `feathersClient.js`: - `rawServices` this are services that are not binded to the state which means that calls made from here will not affect the state - `services` (binded with dispatch)

How is this configured

```
import reduxifyServices, { getServicesStatus, bindWithDispatch } from "feathers-redux"
import configureStore from "../store"

...

    // Configure Redux
export const serviceNames = [
    'users',
    'course-evaluation'
]
export const rawServices = reduxifyServices(feathersClient, serviceNames);

const store = configureStore(rawServices);
export default store;

export const services = bindWithDispatch(store.dispatch, rawServices)
```

Most of the time, you will want to update the state upon entering a page. Be wary, to select only data that you think you will need at this time.

Administrator View Fetching data of users and course name

useEffect(() => { services.users.find() services["course-evaluation"].find({ query: { $select: ["courseId"] } }) }, [])

te: Feathers Routes does not work with Camel Casing

Feathers routes does not work with camel casing because of the convention of how internet URLs are not case sensitive, hence the use of kebab case.

This is very important to know!!! Because services name will be named after it.

Using and Creating Custom Action and Reducer

An example of a custom action and reducer is authentication. There are mainly 3 parts that you need to do to create a custom action and reducer for state management, see below as follow.

In the `AuthGuard` component, the following useEffect code is used to prevent accessing of page without Authentication.

```
// Store Actions and Redux
import { useDispatch } from "react-redux"
import { signIn } from "actions/auth"

...

    useEffect(() => {
        // Authentication Setup
        dispatch(signIn())
    }, [])
```

`useDispatch` is a function that takes a function (`signIn` is a function that returns a function with setup action dispatch)

1. signIn Action

```
export const signIn = () => async (dispatch, getState) => {
  try {
      const loginDetails = await feathersClient.reAuthenticate()
      return dispatch({
          type: "SIGNIN_SUCCESS",
          ...loginDetails
      })
  }
  catch (error) {
      // Cant Authenticate
      return dispatch({
          type: "SIGNIN_ERROR",
          error
      })
  }
}
```

This will contain intermediary action prior to modification of a state via the reducer.

Authentication Reducer

```
const initState = {
  user: null,
  error: null
}

export const authenticateReducer = (state = initState, action) => {
    switch (action.type) {
        case "SIGNIN_SUCCESS":
            return {
                ...state,
                user: action.user
            }
        case "SIGNIN_ERROR":
            return {
                ...state,
                error: action.error
            }
        case "SIGNOUT_SUCCESS":
            return initState
        default:
            return state
    }
}
```

This will outline actions towards the state. Note that any action that calls a reducer, all of its parameters are passed to the action object. Hence `action.type`, `action.user`, and `action.error`. The correct names that follows from the action should be followed.

Combine Reducer

```
import { combineReducers } from 'redux';

// Custom Reducers
import { authenticateReducer } from "reducers/auth"

export default function (reduxifiedServices) {
  return combineReducers({
    users: reduxifiedServices.users.reducer,
    "course-evalution": reduxifiedServices["course-evaluation"].reducer,
    auth: authenticateReducer
  });
}
```

This combines the existing reducer that is setup to the new reducer created.

Fetching data from State

An example is getting the current user that has login

Getting User Name

```
import { useSelector } from "react-redux"

...

const user = useSelector(state => state.auth.user)

console.log(user.name)
// This will print out the name of the user or it will error (if user is null)
```

Accessing the user can be accessed from the reducer auth. The auth reducer has access to its state which just happens to store the current user login.

5.6.3 Notifications with Redux Saga

Redux Saga is a library that aims to make application side effects (more info in the documentation) mainly for the following:

- easier to manage after effects
- better error handling

In IndEAA, this is used for notifications handling whenever a feathers request finished.

How does it work? (Simplified)

As you know, whenever you are dispatching an action to the store, Redux goes through the middleware that is setup with it.

Logger

This is how the logger functions to know which action has been dispatched.

Redux Saga essentially takes advantage of that to know which action has been dispatched, and from there, it can do whatever.

Configuration

Here are the steps to configure Redux Saga. Assuming you have already installed it

1. Add Saga Middleware

For IndEAA, this can be found in `client/middleware/index.js`

```
import createSagaMiddleware from "redux-saga"
...
// This needs to be activated after applying it to the storep
export const sagaMiddleware = createSagaMiddleware()
...
export default[
    ...
     sagaMiddleware,
    ...
]
```

2. Run the middleware after the store has been created

For IndEAA, this can be found in `client/store/feathersClient.js`

```
import { sagaMiddleware } from "../middleware"
import feathersSaga from "./feathersSaga" // This will be all your custom Saga Actions
...

const store = configureStore(rawServices);

// Run all Saga Middlewares
sagaMiddleware.run(feathersSaga)
```

With these two steps, Redux Saga is ready for action! See more below on how to use it.

What do you have to do to use it?

To use it, there are two main steps below

DEFINING A SAGA ACTION

To use it, you have to define a Saga action in `client/store/feathersSaga.js`

The most common structure will look something like this:

Common Structure

```
yield takeEvery(services.users.types.SERVICES_USERS_CREATE_FULFILLED, function* (action) {
        const { payload } = action
        // Payload is the dispatch action items
        yield put(addNotificationMessageParams({
            message: `User (${payload.email}) has been successfully created`
        }))
    });
```

The first parameter is the "type". When dealing with feathers types please refer to information here where types and Redux Saga is mentioned. The second parameter is a generator callback function (it uses `yield` instead of `return`)

`put` is a custom dispatch action for Redux Saga

Available Types in Feathers-Redux

Referring to the Feathers Redux documentation, the main types are as follows

- PENDING
- FULFILLED
- REJECTED

SERVICE CALLS

When the saga action is defined, service calls that can have notifications are easy peasy.

Example of Service Call that will have notification

```
// Found in the User Creation Modal
  const createUser = async (email) => {
    try {
        const response = await services.users.create({ email })
        closeModal();
        setCurrentUserSelected(response.value)
    } catch (error) {
        // Handled by Redux Saga
    }
}
```

It is a simple as that. Do your call, and of course catch the error (because what if you have BadRequest or lost internet connection). You don't need to do anything after the error unless you want to, but essentially Redux Saga will handle error reporting for you when you setup that.

Notications Component

The notification component uses a library called notistack which enables notification to be stacked. Check out the documentation for more information.

SETUP

There is 1 library component imported (`SnackbarProvider`), and 1 custom component `Notification` created to make this happen. See `components/Layout/ContentWrapper.js`

SnackbarProvider

It can be seen that the components are wrapped:

```
const ContentWrapper = ({ children }) => (
<SnackbarProvider maxSnack={3}>
    <Notification></Notification>
   ...
</SnackbarProvider>
)
```

This is essentially a component that uses context to display notifications which means that this is separate from the Redux.

Context and Redux

Context is almost the same thing as redux, both are used for state management, but context is built in React. The point of this information is to point that the `SnackbarProvider` data store/context is different from the Redux Store. Hence, it needs to be reconciled or coordinated when we used them both.

Notification Custom Component

The notification component, is special such that it doesn't render anything. It is simply a shell component that enables code to run upon modification of the notification redux store in order to modify the notistack context data store.

Why are we using Redux for managing notification

notistack library only enables `enqueueSnackbar` either using class-component or functional component by hooks. These hooks cannot be called whenever using Redux Saga, because those are not components.

5.7 Backend

5.7.1 Permission

Permission refers to the authority of the user to access or modify information. In business rules, the only reason why a user has to have permission if they need to have access to that information or authority to have action. This means that if a user does not need to access that information, they should not have it.

In IndEAA, this is essential in restricting each of the users to perform only actions that they have permission over. The permission has 2 main elements:

- `role` (what sets of permission does the user have?)
- `course_id` (which `course_evaluation` where this role persists?)

Relationship to Frontend

Although this part of the documentation applies for both the backend and frontend, this documentation is mainly for backend as it is where most business rules will apply. Permissions for frontend mainly apply only for the convenience of the user, but does not protect certain information or action from being exposed. Hence, all data-driven action (CRUD operations) should be protected by the backend.

Role

ADMINISTRATOR

Administrators have the capability to adjust all the permissions of the user in the system, and even has the ability to delete users from the system.

COORDINATOR

Coordinators have the capability adjust everything related to the `course_evaluation` and should be able to view details about other coordinators and reviewers of their `course_evaluation`

REVIEWER

Reviewers have the capability to adjust everything related to their own `review` for a `course_evaluation`, and should be able to view details about coordinators for the `course_evaluation` they are assigned to.

Hooks

ROLE-BASED RESTRICTION

This hook restricts actions/service methods only to users with a specific role regardless of `course_id`.

Examples of Times where you want this

- Admiinistrator Role

The administrator role does not need `course_id`. Hence, should be used for it

- Coordinator Role

Creating a `course_evaluation` requires a Coordinator role, but does not need to check for `course_id`

FILTER-BASED RESTRICTION

This hook restricts query (`GET` and `FIND` service methods) to limit (`FIND`) based on query or restricts access (`GET`).

ROLE-AND-COURSE BASED RESTRICTION

This hook is a stricter version of `Role-Based Restrictions` as this also applies with `course_id` that the user has access to.

Example of Times where you want this

- Multi-Role Coordinator

  A coordinator should not have access to other `course_evaluation` they do not have access on.

- Multi-Role Reviewer

  A reviewer should not have access to `course_evaluation` they do not have access on, and should also not be able to view `review` that they are not the review person for.

5.7.2 Services

Services refers to endpoints of the backend that interfaces directly with the database and may have relations to other service (as opposed to microservices).

See below for the services that IndEAA has.

Users `/users`

This is responsible for data-driven actions related to users.

Course Evaluation `/course_evaluation`

This is responsible for data-driven actions related to course that is being evaluated.

METADATA

This service has two metadata field that are not mapped in the model.

`COORDINATORS`

This is an array of users object that has a permission `course_id` for this `course_evaluation` and `role` as "Coordinator. The object has these following fields:

- `_id`
- `name`
- `email`

`REVIEWERS`

This is an array of users object that has a permission `course_id` for this `course_evaluation` and `role` as "Reviewer". The object has the same following fields as `coordinators` object structure.

> **Model != Service**

There is some difference to a service to a model field, hence should not be relied upon if deciding on which endpoint to use.

Review `/review`

This is responsible for data-driven actions related to reviews for a specific course.

5.7.3 Report Generation

The report generation feature is used for generating reports for the Coordinators.

Implementation

The report is generated everytime a new change that has been done for a Course Evaluation (either the details changed or reviews are filled).

Node Background Process

Report generation for every change normally increases server load. However, it is the easiest implementation. Also, it doesn't affect the performance of the server application as much because it runs as an after hook that is not awaited.

With Pandoc, the report generation usually takes only a couple of seconds to generate <50 pages.

Do not Await!

As said, it runs as an after hook that is not awaited so don't put an `await` there.

MARKDOWN TO WORD DOCUMENT PANDOC

The feature is implemented using the package node-pandoc-promise.

Requirement: Pandoc

This package requires installation of node-pandoc. You can easily install it by typing `apt-get install pandoc` (this is sadly not documented in the original documentation)

ACCESS FROM FRONTEND

The server application uses the `/public/documents` path to store the generated report. This storage is for the purpose of easily acceessing reports from the frontend.

Security

As of the implementation of the feature, the security authentication is not established.

Currently, it is deemed secure enough because to access a file, the user/application will need the **exact** path of the file they want to access. The format will be in `public/documents/${course_id}/IndEAA-Report-{courseCode}`. The hardest bit here to guess is the `course_id` because that refers to the MongoDB Object ID. Guessing this will probably be almost the same thing as trying to guess the JWT authentication token.

6. Changelog

This is the changelog for IndEAA.

7. v1.2.0 - Jul 15, 2021

New

- Implemented Cypress and Automated Testing Pipeline (#14)

Improvements

- Pagination Default Changes from 20 to 100 (#48)
- Cleanups of Import (partial progress) (#42)

Bugfixes

- Broken css documentation fix (#125)

8. v1.1.0 - Mar 3, 2021

Improvements

- Introduction Documents + Some Optimisation o f Data Fetching Optimisation Related (#82, #108)
- Permission protection + metadata on course-evaluation page (#24, #71)

Bugfixes

- Fix for adding permission to reviewer from coordinator view (#110)

9. v1.0.0 (Initial Deployment) - Feb 5, 2021

First initial deployment

High level summary of first deployment

- Project setup (#4)
- Basic administration functionality (#8)
- Basic coordinator functionality (#19, #65)
- Basic reviewer functionality (#30)
- Google Oauth authentication (#7)
- Production environment setup (#73)