From Redux to Component-level State: Modernizing CMSX Architecture

Alex Kozik
December 19th, 2024

Table of contents

1: Introduction

1.1: Overview and Motivation

Over the past 2 semesters, I migrated the new UI from a Redux-facilitated global state architecture to a component-level state architecture. We had four main goals with the project:

1. Reducing the amount of data that is sent to the frontend on login
2. Ensuring that users see fresh data whenever they navigate to a new page
3. Improving the scalability of the frontend architecture, allowing for future work on the frontend
4. Reducing latency on the frontend

1.2: Phases of the project

Research phase (Jan - Late-Feb): During this phase, I researched the architecture of the frontend to figure out how all of the different components interact with each other.

Planning phase (Late-Feb, Mid-March): During this time, I had already gathered significant insight into how the new UI works, and was deciding between two possible approaches for improving the architecture - lazy global state and component-level state. I gave a presentation on March 4th detailing my findings from the research phase, and received feedback from everyone about which approach would be more optimal.

Implementation phase 1 (Mid-March - Early-May): After deciding to go with component-level state, I utilized my findings from the previous two phases to migrate the new UI to component-level state. By the end of this phase, we had a good prototype for how the new UI would with component state. However, it was far from perfect. The interface was glitchy, the backend was extremely inefficient, and there were numerous bugs.

Implementation phase 2 (September): I continued to work on refining the PR.

Reviewing phase and release (October - Mid-November): Noah, Rohen, and Colin did an initial review of 2229. Since it was a large PR, there were 200+ comments to address. During this time, I also fixed some bugs. so it took most over October and into November to polish everything. We merged the PR in mid November and deployed it to production!

Further development phase (Mid November - Mid December): After the initial release of component state, we identified some areas for improvement, such as an assignment overview builder, removing Redux form, removing `SessionFactory`, improving the loader, and other things. We implemented these things and deployed them at the end of finals week.

1.3: Related Pull Requests

PR Name	PR #	Purpose	Author
Component State Architecture for React	#2229	Implement the majority of the component state architecture. This document primarily focuses on this PR.	Alex
Rewrite Student Course API	#2452	Make `GetStudentCourseData` more efficient, get rid of `SessionFactory`, implement assignment overview builders.	Alex & Thomas
Remove Redux	#2485	After merging 2229, the only part of the Redux store that was left was the form slice. This PR removed that and made forms operate via component state.	Colin
Add Delay to Loader	#2502	Students were complaining that the website seemed slower. This PR fixes that by only making a spinner show up after an API request has been in progress for at least 500 milliseconds.	Alex

2: The Architecture before Component-level State

2.1: Redux Store

2.1.1: Redux Store Slices

Before implementing component-level state, the primary mechanism for storing data on the frontend was the Redux, store, which is a collection of 6 "slices", as follows:

2.1.2: Reducers

```
const rootReducer = combineReducers({
    privileges: privilegeReducer,
    status: statusReducer,
    ....
});
export const initialState: AppCentralState = {
    privileges: guestUserPrivileges,
    status: ....,
    ....
```

Each slice has a corresponding reducer, which is a function of type $\text{reducer} : \text{Store} \times \text{Action} \to \text{Store}$ that determines how frontend actions affect the condition of a particular slice of the Redux store.

Action creators are used to modify the Redux store. Here's an example:

```
export function resetSessionAction(): SetSessionAction {
    return {
        type: SET_SESSION,
        session: new Session(),
    };
}
```

2.1.3: Connecting Components with the Redux Store

In order to "link up" a component with the Redux store, we define two methods:

1. `mapStateToProps`, which "injects" parts of the Redux store into the props of the component
2. `mapDispatchToProps`, which "injects" action creators into the props of the component

Then, `connect` is used to apply `mapStateToProps` and `mapDispatchToProps` to the component

For example:

```
class App extends React.Component {
....
}
const mapStateToProps = ({ .... }: AppCentralState) => { .... }
```

```
const mapDispatchToProps = (dispatch: any) => { .... }
export default connect(mapStateToProps,mapDispatchToProps)(App);
```

2.2: Data Classes and Serialization

Data transfer from the backend to the new UI works as follows:

1. Java backend constructs a ***data class*** instance
2. Data class instance is serialized to JSON
3. JSON is sent to the frontend using HTTP
4. TypeScript frontend deserializes the JSON into a ***data class*** instance

Data class refers to a class that stores data to be sent to the frontend. In order to facilitate data transfer, the frontend and backend have an isomorphic set data classes. For example:

```
@JsonObject('UserData')
export class UserData {
    @JsonProperty('netId', String)
    public netId: string = 'guest';
    @JsonProperty('firstName', String)
    public firstName: string = 'A';
    ....
}
```

```
public class UserData {
    private final String netId;
    private final String firstName;
    ....
```

To serialize data classes on the backend, we use a library called Jackson.

To deserialize JSON to TypeScript objects we use a library called Json2Typescript. The `@JsonProperty` decorator defines how a property is deserialized, namely, what property name to look for in the JSON and what type to deserialize it to.

2.3: Data Transfer and APIs

`SessionFactory` has methods for constructing data class instances at every level of the tree.

Here's an example of an API

```
@EndpointActionName ("apigetsession")
public class GetSession implements Endpoint {
    @ApiMethod(HttpMethod.GET)
    public ActionResult doGet( (.... ) {
        SessionFactory sessionFactory = new SessionFactory(database, assignmentController, structs);
        Session session = sessionFactory.buildSession(principal);

        return ActionResult.json(session);
    }
}
```

`CMSXApiContext` is a class that contains all functions that fetch data from the backend.

```
export class CMSXApiContext {
    public async getSession(): ....
    public async getStudentEmailPrefs(): ....
    ....
}
```

Here's how `getSession` is implemented

```
public async getSession(): Promise<Session> {
    return this._callEndpoint(actions.getsession, RequestType.GET, {}, Session);
}
```

Here, `this._callEndpoint` is a helper function, and the arguments are as follows:

- `actions.getsession` is the particular endpoint we want to access
- `RequestType.GET` is the request type. It could also be `POST`
- `{}` is the JSON object we send to the api
- `Session` is the class to which we deserialize the JSON response

3: Remarks and Issues

3.1: The issues with the architecture before component-level state

When a user would log in, the entire ***session*** would be sent to the frontend after the `App` component mounts.

```
componentDidMount(): void {
    this.props.getSessionAction.bind(this)();
}
```

Session refers to the large collection of data encapsulating anything the user would need to see while using the website.

Issue 1 - Excessive Data Transfer - Upon experimenting, I found that the amount of data sent to the frontend upon login can easily exceed 1,500+ lines of JSON, which is excessive, considering that the overview page only display basic information about courses and assignments.

Issue 2 - Lack of Data Refreshing - Moreover, state refreshing was done very rarely, particularly only after actions like submitting an assignment. And, even then, lot's of unnecessary data was sent upon refreshing.

Issue 3 - Lack of Scalability - Given this architecture, it would be very difficult to modify the data that is sent to the frontend, since the structure of the Redux store is rigid and inflexible.

Issue 4 - Complex code - The code that deals with Redux is unnecessarily complex for our purposes.

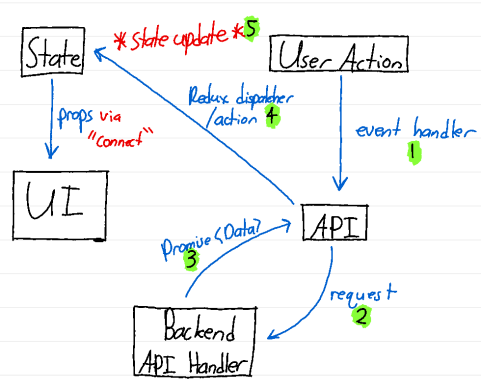
3.2: Lazy Global State and why we decided against it

After finishing the research phase of this project, I had come up with two prototype designs and I had to decide which one to implement. They were:

1. Lazy global state
2. Component-level state

Lazy global state would be similar to the original state management architecture, but it would only load the parts of the `sessionData` slice that were required.

Here is a diagram of how it would work:



We decided against this architecture because, although it would remedy the issue of sending unnecessary data to the frontend, changing the structure of the Redux store to send more/different data would remain very difficult. Therefore, this approach would not be scalable for future projects on the new UI.

Moreover, the code would become extremely complicated, as facilitating lazy state would require designing the data classes in a way that would allow "emptiness", which is highly prone to bugs and would make debugging very difficult.

3.3: Takeaways

Initially, I was going to implement lazy global state. In fact, when I gave the presentation detailing the two approaches, everyone except for Noah was in favor of lazy global state.

However, after a long back and forth with Noah and Ryan, I became convinced that implementing a component-level state architecture would be:

1. Simpler
2. More maintainable
3. Less prone to bugs
4. More conducive to future projects on the new UI

With this in mind, I began to implement component-level state.

4: Component-level State architecture

4.1: New Data Classes

In order to facilitate the sending of exactly the data we need on the frontend, I implemented some new data classes.

As with the other data classes, I implemented them in both Java (backend) and TypeScript (frontend) to facilitate serialization and deserialization.

They are as follows:

- `AssignmentOverview`
  - Contains basic information about an assignment that is displayed on the overview page.
  - Properties are
    - `name : string`
    - `shortAssignmentName : string`
    - `dueDate : Date`
    - `type : number`
    - `status : AssignStatus`
    - `submittedString: : string`
    - `lateSubmissionsDue : Date | null`
    - `assignmentID : number`
    - `courseID : number`
    - `courseDisplayCode : string`
    - `isCurrent : boolean`
    - `weight : number | null`
- `AssignmentOverviewExtended`
  - A subclass of `AssignmentOverview` that also contains. It is used to send data about assignments on a course menu.
  - Properties are (in addition to those inherited from `AssignmentOverview`)
    - `statistics : ScoreStatistics`
    - `score : number | null`
    - `slipDaysUsed : string`
- `StudentAssignmentDataAndRegrades`
  - Contains all data needed by an assignment menu page, which is all data about the assignment and all regrade data about the assignment.
  - Properties are
    - `studentAssignmentData : StudentAssignmentData`
    - `commentDownloadResponse : CommentDownloadResponse`
- `CourseOverview`
  - Contains basic information about a course that is displayed on the overview page and nav bar, such as
    - `displayCode : string`
    - `courseName : string`
    - `courseID : number`
    - `isPastEnrollment : boolean`
    - `isStaffCourse : boolean`
    - `semester : Semester`
- `SemesterOverview`
  - Contains basic information about courses, grouped into a single semester
  - Typically used to represent a student's enrollments in courses during a past semester
  - Properties are
    - `semester : Semester`
    - `courses : CourseOverview[]`

- `HeaderData`
  - Contains data that is required by the header
  - Properties are
    - `netID: string`
    - `firstName: string`
    - `lastName: string`
- `NavbarData`
  - Contains data that is required by the nav bar
  - Properties are
    - `currentSemester : Semester`
    - `courseOverviews : CourseOverview[]`
    - `isAdmin : boolean`
- `CourseContent`
  - Contains data that is required when viewing a course's content/categories
  - Properties are
    - `categories: Category[]`
    - `courseDisplayCode: string`
- `NavbarAndProfileData`
  - Contains all data needed for the nav bar and header
  - Properties are
    - `profileData: HeaderData`
    - `navbarData: GetNavbarDataResponse`
- `GetOverviewResponse`
  - Contains all data needed for the nav bar and header
  - Properties are
    - `assignmentOverviews: AssignmentOverview[]`
    - `courseOverviews: CourseOverview[]`
- `StudentCourseDataResponse
  - Used to send data about a course to a course menu
  - Properties are
    - `properties: Course`
    - `finalGrade: string | null`
    - `semester: Semester`
    - `announcements: Announcement[]`
    - `assignments: AssignmentOverviewExtended[]`
    - `totalStatistics: ScoreStatistics`
    - `totalScore: number | null`

4.2: New Data Class Builders

Consider the following code from `GetStudentCourseData`

```
assignmentOverviewExtended.add(new AssignmentOverviewExtended(
        assignment.getName(),
        assignment.getNameShort(),
        Optional.of(assignment.getDueDate()),
        assignment.getAllowLate(),
        assignment.getType(),
        assignment.getStatus(),
        GetAssignmentsOverview.makeSubmittedString(assignment, student, database.em),
        Optional.ofNullable(assignment.getLateDeadline()),
        assignment.id,
        course.id,
        course.getCode(),
        true,
        assignment.getWeight(),
        stats,
        score,
        slipDaysUsed
));
```

The problem here is that the constructor for `AssignmentOverviewExtended` takes 17 arguments, and we have to construct them all before we pass them in. Moreover, some of them are nullable, so we have to wrap them in `Optional<T>` before passing them in.

To make this code less bug prone and easier to read, I implemented builder classes for `AssignmentOverview` and `AssignmentOverviewExtended`.

Here is part of the builder class for `AssignmentOverview`

```
public static class Builder {
    String name;
    String shortAssignmentName;
    ....

    public Builder withName(String name) {
        this.name = name;
        return this;
    }

    public Builder withShortAssignmentName(String shortAssignmentName) {
        this.shortAssignmentName = shortAssignmentName;
        return this;
    }
    ....
}
```

Here is part of the builder for `AssignmentOverviewExtended`

```
public static class Builder extends AssignmentOverview.Builder {
    private Optional<ScoreStatistics> statistics = Optional.empty();
    ....

    // Override methods from AssignmentOverview.Builder
    @Override
    public Builder withName(String name) {
        this.name = name;
        return this;
    }
```

```
   ....

    // New methods specific to AssignmentOverviewExtended
public Builder withStatistics(ScoreStatistics statistics) {
    this.statistics = Optional.of(statistics);
    return this;
    }

    ....
```

A key thing to note here is that we are overriding each inherited method in order to change the return type from `AssignmentOverview.Builder` to `AssignmentOverviewExtended.Builder`.

With these builders, the original code in `GetStudentAssignmentData` becomes

```
AssignmentOverviewExtended.Builder assignmentBuilder = new
// Add the properties we know exist
AssignmentOverviewExtended.Builder()
        .withName(assignment.getName())
        .withShortAssignmentName(assignment.getNameShort())
        .withLateSubmissions(assignment.getAllowLate())
        ....

// Conditionally add properties that might not exist

if (assignment.getShowSlipDays()) {
    assignmentBuilder.withSlipDaysUsed(....));
}

....

assignmentOverviews.add(assignmentBuilder.build());
```

The `.build()` methods takes all of the arguments that have been accumulated in the builder and constructs an `AssignmentOverview` object from them.

I also used `AssignmentOverview.Builder` to make the code cleaner in `GetOverview`. The result is a much cleaner, more maintainable, and less error-prone approach to constructing `AssignmentOverview` and `AssignmentOverviewExtended` objects. By using builder classes, we simplify the process of setting properties, handle optional arguments more gracefully, and make the code more readable.

4.3: New APIs

4.3.1: Utilities

In `CMSXApiContext` I implemented a new function `_callEndpointGetArray`, which works the same way as `_callEndpoint`, except it deserializes the response into a list of objects, rather than a singleton.

4.3.2: New Endpoints

I implemented the following endpoints as Java classes on the backend

Endpoint	Serialized Return Type	Description	Parameters
`GetCurrentSemester`	`Semester`	Gets the current semester	
`GetOverview`	`GetOverviewResponse`	Gets all data needed for the overview page	
`GetCourseContent`	`GetCourseContentResponse`	Gets a course's content	
`GetNavbarAndProfileData`	`NavbarAndProfileData`	Gets all data needed for the nav bar and header	
`GetStudentAssignmentData`	`StudentAssignmentData`	Sends information about a specific assignment	`{courseid: number, assignid: number}`
`GetStudentAssignmentDataAndRegrades`	`StudentAssignmentDataAndRegrades`	Gets all information about a particular assignment and all regrade information for the assignment.	`{courseid: number, assignid: number}`
`GetCourseAnnouncements`	`Announcement[]`	Sends a list of announcements for a course	`{courseid: number}`

4.3.3: Wrappers for the new APIs

I built the following methods into `CMSXApiContext` as wrappers for the new APIs described above

Wrapper	Endpoint
`getCurrentSemester()`	`GetCurrentSemester`
`getSemester()`	`GetSemester`
`getOverview()`	`GetOverview`
`getCourseContent()`	`GetCourseContent`
`getNavbarAndProfileData()`	`GetNavbarAndProfileData`
`getStudentAssignmentData()`	`GetStudentAssignmentData`
`getStudentAssignmentDataAndRegrades()`	`GetStudentAssignmentDataAndRegrades`
`getCourseAnnouncements()`	`GetCourseAnnouncements`
`getStudentCourseData()`	`GetStudentCourseData`

Each wrapper method has roughly the same implementation. For example:

```
public async getCourseContent(courseid: number): Promise<CourseContentResponse> {
    return this._callEndpoint(
        actions.getcoursecontent,
        RequestType.GET,
        { courseid: courseid },
        CourseContentResponse,
```

```
    );
}
```

4.3.4: Using the new APIs

New APIs can be used as follows:

```
const Component = (props: ComponentProps) => {
    const [data, setData] = useState<Data>(new Data())

    useEffect(() => {
        apiContext.getData()
            .then(setData)
            .catch(alert)
    }, [])

    return ( .... )
}
```

And, in the case of a class the case of a class component, it looks like this

```
componentDidMount() {
    apiContext.getData()
        .then(data => {
            this.setState({
                data: data
            })
        })
        .catch(alert)
}
```

Originally, this is how we used the new APIs in 2229. However, there are two problems here

1. This pattern uses a default "empty" value for `data` before the API call is finished. This is really bad because we don't want any UI components to rely on this data.
2. This pattern does not take into consideration whether the API request is in progress, successful, or failed.

With these things in mind, I implemented a `Loader` component and a `useAPI` hook which solved these issues. They are discussed in detail in sections 4.4, 4.5, and 4.6.

4.4: Loader

One of the big considerations with component-level state is that when a user navigates to a new page, the data they want will not immediately be available to them, since the API takes some time to send the data. Therefore, we need a way to show the absence of data until it is received by the frontend.

Some of the considerations with this are

1. We want it to be obvious to the user that there isn't any data, but there should be data.
2. We want it to be obvious to the user that progress is being made.
3. We want the design to be subtile, and not too distracting from the main content of the page.
4. Each page makes 1 API request upon loading.

I implemented the following React component for the loader

```
export const Loader = ({ showLoader } : LoaderProps) => {
    if (showLoader) return (
        <div className="loader-container">
            <ClipLoader color={"gray"}
                size={75}
            />
        </div>
    );
    return <></>;
}
```

I utilized the loader in the following components:

- `CoursesSection`
- `StudentAssignmentMenu`
- `CourseDetails`
- `BaseStudentAssignment`
- `CourseContent`
- `PastSemesters`
- `GradingCommentsAndRegrades`
- `StudentSurveyMenu`
- `CurrentStudentAssignments`
- `Navbar`

4.5: APIStatus

One of the issues that came up was how we should handle missing data, for example, when we are unable to contact the backend. To solve this problem, I drew inspiration from algebraic data types in OCaml. I wanted something like this:

```
type 'a api_status =
    | Loading of bool (* whether we should render the loader *)
    | Failure of string (* the error message *)
    | Success of 'a (* the data from the API *)
```

Since TypeScript doesn't have a way to declare a sum type, like in OCaml, I made each constructor it's own type with a `type` field to pattern match between them, then made `APIStatus<T>` the union of the three constructors.

```
export type Loading = { type: 'Loading', showLoader: boolean };
export type Failure = { type: 'Failure', errors: string[] };
export type Success<T> = { type: 'Success'; value: T };
export type APIStatus<T> = Loading | Failure | Success<T>;
```

There are three possible "constructors" for a value of type `APIStatus<T>`.

- `Loading` indicates that there is no current data to display and we are still in the process of fetching the data. `showLoader` is whether we should render the loading circle on the page.
- `Failure` indicates that we tried fetching the data, but were unable to fetch it. the `errors` field has the list of errors from the backend.
- `Success<T>` indicates that we have successfully fetched data, and it is in the `value` field of the object.

Each constructor also has a `type` field, which is a constant string. The `type` field allows us to pattern match over which constructor we have.

For example

```
const data: APIStatus<T> = ....;

if (data.type === 'Success') {
    // data : Success<T>
    const value: T = data.value;
}

else if (data.type === 'Loading') {
    // data : Loading
}

else {
    // data.type === 'Failure' implies data : Failure
    const errors: string[] = data.errors;
}
```

The result of this is a clean way to represent whether we have data, are trying to get data, or have failed to get data, without the dangers of using `null` or `undefined`.

4.6: useAPI

I also implemented a new React hook called `useAPI`. The purpose of this hook is to call an API and return an `APIStatus<T>`, then update the `APIStatus<T>` in real-time as the status of the API call changes. `useAPI` also manages whether the loading overlay should be rendered.

Here is the signature of `useAPI`

```
export function useAPI<T>(
    fetchFunction: () => Promise<T>,
    onSuccess?: (result: T) => void,
    showErrorMessage: boolean = true,
    loaderDelay: number = 500
): [APIStatus<T>, () => void]
```

- `fetchFunction` is the method that calls the API wrapper. It must return a `Promise<T>`, where `T` is the data type we ultimately want to receive.
- `onSuccess` is an optional parameter. It is a callback function that will be applied to the data of type `T` whenever a successful API request occurs.
- `showErrorMessage` is an optional parameter. If an API request fails and `showErrorMessage` is `true`, a toast will be displayed with the error message (this is why we track errors in `APIStatus<T>`).
- `loaderDelay` is an optional parameter. Initially, we don't want to show a loading overlay. However, after `loaderDelay` milliseconds, we do want to show a loading overlay. So initially, the value of the data returned from `useAPI` will be a `Loading` object with `showLoader` equal to `false`. `useAPI` will set a timeout for `loaderDelay` milliseconds, and, if by the time the callback function executes, the API request is still in progress, `useAPI` will set the data to `Loading` with `showLoader` equal to `true`. The TLDR is that `loaderDelay` is how long we want to wait before displaying a loader.

The return values are as follows

- The 0-index element is an `APIStatus<T>` that represents the status of the API request. It will be modified in real time as updates are made to the API request.
- The 1-index element is a function that, when called, will call `fetchFunction` and update the 0-index element accordingly.

Here is an example usage of `useAPI`

```
const Overview: React.FC = () => {
    const [data, fetchData] = useAPI(() => apiContext.getOverview())
    ....
};
```

Here is an example of how we use `useAPI` and `Loader` together to concisely implement a stateful component:

```
const Overview: React.FC = () => {
    const [data] = useAPI(() => apiContext.getOverview())

    if (data.type === 'Failure') return (
        /* Render a failure message */
    )

    if (data.type === 'Loading') return (
        <Loader showLoader={data.showLoader}/>
    );

    return (
        /* The actual content of the page */
    )
};
```

Here, `data` is of type `APIStatus<T>` and `fetchData` is a function that recalls the API, and updates `data` accordingly.

This is now the general pattern that is used to implement stateful components on the new UI.

In a more complex component like `StudentAssignmentMenu`, we can leverage the `fetchData` return value as follows

```
const StudentAssignmentMenu: React.FC<Props> = ({ .... }) => {
    const [data, fetchState] = useAPI(() => ....));
    const declineGroupInvite = ( .... ) => {
    apiContext.declineGroupInvite(groupid)
        .then(() => {
            toast.success("Declined invitation");
            fetchState();
        })
        .catch(toast.error);
    };
    ....
}
```

Overall, having `useAPI` significantly reduces boilerplate, and is a concise way to call APIs and use their data.

4.7: Class components -> Function components

In the old version of the new UI, we had class components. Some of the issues with class components are

1. Verbose syntax
   - This makes class components harder to read and maintain
2. Complex state management using lifecycle methods
   - `componentDidMount`
   - `componentDidUpdate`
   - `componentWillUnmount`
   - `this.setState({ .... })`
3. The `this` keyword can be confusing
   - The need to bind methods to the component instance (`this`) can lead to confusion and bugs.
4. **Class components cannot use API hooks**
   - This is a big problem

So, I refactored all class components to be function components. There were about 10 to do.

The main benefits of this are

1. We now have a simpler syntax for React components, which makes them easier to read and understand.
2. Function components can use React hooks, which allows us to use the `useAPI` hook. This greatly simplifies making API requests and state management.

4.8: New Route for Staff

When a user clicks on a course link, one of two things happens

1. If they are a student, they are brought to the course menu on the new UI
2. If they are a staff, they are redirected to the course menu on the classic UI

Performing one of these two actions requires knowing whether the user is a student or staff in the course.

The way the `CourseMenu` component handled this before component-level state is as follows

```
render() {
    const course = this.lookupCourseById(courseid);
    /* .... */
    if (isStaffCourse) return <StaffCourseMenu .... />;
    else return <StudentCourseMenu .... />;
    /* .... */
}
```

This kind of approach would no longer work with component-level state because the `CourseMenu` component would not immediately know whether the user is a student or staff in the course, at the time of rendering.

I solved this issue by implementing a new route in `App` as follows:

```
<Route
    path={'/staff/:id'}
    exact
    render={(props) => {
        if (
            props.match.params.id !== undefined &&
            isPositiveInteger(props.match.params.id)
        ) {
            this.setActivePage(Page.COURSE);
            return (
                <StaffCourseMenu
                    courseid={parseInt(props.match.params.id)}
                />
            );
        } else
            return (
                <div className="content">
                    <h2>Error: Invalid Course id</h2>
                </div>            );
    }}
/>
```

Upon connecting to this URL, `StaffCourseMenu` renders, which redirects the user to the course page on the classic UI.

I then made links in `Navbar` and `CourseSection` redirect to `/staff/:id` when the user is a staff in the course.

The result is a seamless way to redirect to the classic UI for staff courses without `CourseMenu` needing to fetch any data at all.

4.9: Statuses with Toastify

In order to facilitate a more interactive user experience, Noah and I replaced `alert` with Toastify, a status library. Here's an example of how we can use Toastify:

```
const submitRegradeFromForm = (text: string, problems: number[]) => {
    apiContext.postStudentRegradeRequest(assignid, text, problems)
        .then(() => {
            toast.success("Regrade request posted");
            fetchState();
        })
        .catch(toast.error);

};
```

Here a success message is shown if the the `Promise` resolves, and an error message is shown if the `Promise` rejects.

5: Optimizing `GetStudentCourseData`

We previously had a class - `SessionFactory` which would construct a `Session` object to send to the frontend. This `Session` object would contain all information the client would ever need to know. As described in section 2, this is the basis of how our Redux-facilitated framework used to operate.

However, a few days before merging #2229, we realized that the `GetStudentCourseData` API was utilizing `SessionFactory` to construct `StudentCourseData` object as follows:

```
@EndpointActionName (RequestAction.API.General.GET_STUDENT_COURSE_DATA)
public class GetStudentCourseData implements Endpoint {
    @ApiMethod (HttpMethod.GET)
    public ActionResult doGet( .... ) {
        ....
        SessionFactory factory = new SessionFactory( .... );
        ....
        return ActionResult.json(new StudentCourseDataResponse(semester,
factory.buildStudentCourseData(user, course)));
    }
    ....
}
```

Here, `StudentCourseDataResponse` is defined as

```
private static class StudentCourseDataResponse {
    public Semester semester;
    public StudentCourseData studentCourseData;
    public StudentCourseDataResponse( .... ) { .... }
}
```

There are two main issues with the above implementation of `GetStudentCourseData`

`factory.builderStudentCourseData` constructs the `StudentCourseData` object.

1. A lot of the data in `StudentCourseData` is not needed on a student course menu. So, this API was resulting in a lot of latency when trying to visit the page.
2. We don't want to use `SessionFactory` anymore, since follows the old pattern of constructing `Session` objects.

I addressed this PR by redesigning `StudentCourseDataResponse` to be

```
private static class StudentCourseDataResponse {
    public CourseProperties properties;
    public String finalGrade;
    public Semester semester;
    public List<AnnouncementProperties> announcements;
    public List<AssignmentOverviewExtended> assignments;
    public ScoreStatistics totalStatistics;
    public Float totalScore;
    public StudentCourseDataResponse( .... ) { .... }
}
```

`AssignmentOverviewExtended` is a new data class. It contains the information about an assignment that we need to know in a student course menu. The reason that `AssignmentOverview` didn't suffice here is because it doesn't contain

1. What the student got on the assignment, if it was graded already
2. The scoring statistics of the assignment
3. The number of slip days used for the assignment

`AssignmentOverviewExtended` includes these additional three properties. In order to make creating an `AssignmentOverviewExtended` more efficient, I implemented builder classes for `AssignmentOverview` and `AssignmentOverviewExtended`. There are more details about the implementations of the data classes and the builders in sections 4.1 and 4.2.

6: Benchmarking

6.1: Intro

In the following experiment, we compare latencies and response sizes between the old and new versions of CMSX. The new version is our develop branch after merging #2229. The old version our develop branch at commit `5f23a5b0fe7c1c5967f9cc924c3195fcf60ba86e`.

All assignment specific actions were performed in the course CS 6110

6.2: Setup

Two courses

- CS 6110
  - 20 assignments
  - A ton of course content
- CS 6120
  - 5 assignments

Server: Dev-5
Client location: Cornell University West Campus

6.3: Old version statistics

- Time to load - average of 20
  - 173 ms
- Size of network requests to load
  - 1007kb

These are the only things I tested because there is 0 latency across the rest of the website. This is because all of the data is already stored in the session.

6.4: New version statistics

- Time to load overview - average of 10
  - 51 ms
- Size of network requests to load overview - always the same
  - 13 kb
- Time to load course - average of 10
  - 101.9 ms
- Time to load assignment - average of 10
  - 24.6 ms
- Time to load course content - average of 10
  - 70.3

6.5: Conclusions

In the old version, it takes 173ms to load overview. In the new version, it takes 51 ms to load overview. This is a 71% decrease in latency!

In the old version, it 1007 kb are sent from the backend when loading the page. In the new version, 13 kb are sent from the backend when loading the page. This is a 99% decrease in response size!

7: Future Potential and Goals

7.1: Staff Functionality on the New UI

Originally, my proposal for my CS 4999 project was to migrate grading from the classic UI to the new UI, which we quickly realized was too large of a project for one semester. However, the primary reason this was such an ambitious goal was because, at the time, our frontend didn't yet support the architecture necessary for this project.

Now that component-level state is implemented, modifying the data that is sent to the frontend is much simpler, and, therefore, staff functionality on the new UI becomes a feasible project.

7.2: Automatically Generating Data Classes

Data transfer from the backend to the new UI works as follows:

1. Java backend constructs a data class instance
2. Data class instance is serialized to JSON
3. JSON is sent to the frontend using HTTP
4. TypeScript frontend deserializes the JSON into a data class instance

This architecture requires that the data class instance is written twice - once in Java, and once in TypeScript. One enhancement I'd like to see in the future is a way to write the data class once and have it translated to both languages. This would reduce the probability of misalignments between the two languages and decrease the amount of code we have to write.

One possible approach for this would be ***AutoValue***, which is a library for generating source code for value objects or value-typed objects - https://www.baeldung.com/introduction-to-autovalue.

7.3: Customizable UI

In the future, I'd like to implement UI customizability, like a dark mode. User preferences could be stored on the backend, and there would be a menu for toggling between different color schemes/layouts.

8: Summary

8.1: Final remarks

Working on this project has been very challenging, enlightening, and, above all, rewarding. I entered Spring 2024 having a good foundation of React knowledge, but knowing little about how the new UI works. I had to spend the first 1.5-2 months of the semester researching how the frontend works, trying to understand the architecture.

In the beginning of the Spring 2024, migrating the new UI to component-level state seemed like an awesome idea, but I didn't even know where to start. By the end of the semester, we had a working beta version of it.

After not working on the project for a couple of months, I resumed in September and worked on it for the entirety of Fall 2024. The most difficult part of this project were the 3 weeks leading up to the merging and deployment to production, since I wanted to make sure that the code would perform with thousands of users.

After merging the project, it was amazing to see it working in production. It was so satisfying to see months of work come to fruition and positively impact the daily lives of our fellow students.

8.2: Thank you!

I'd like to thank the following people for their contributions to implementing component state, as this project would not have been possible without them:

- Professor Myers
  - Reviewed #2229
  - Reviewed #2452
- Noah
  - Reviewed #2229
  - Helped fix bugs and upgrade dependencies in #2229
  - Reviewed #2452
- Ryan
  - Implemented backend endpoints, which were merged into #2229
- Colin
  - Reviewed #2229
  - Removed Redux form
- Rohen
  - Reviewed #2229
- Thomas
  - Reviewed #2229
  - Reviewed and helped implement #2452
- The CMSX team
  - Helping test #2229 in a prod environment before merging!