# Advanced Redux

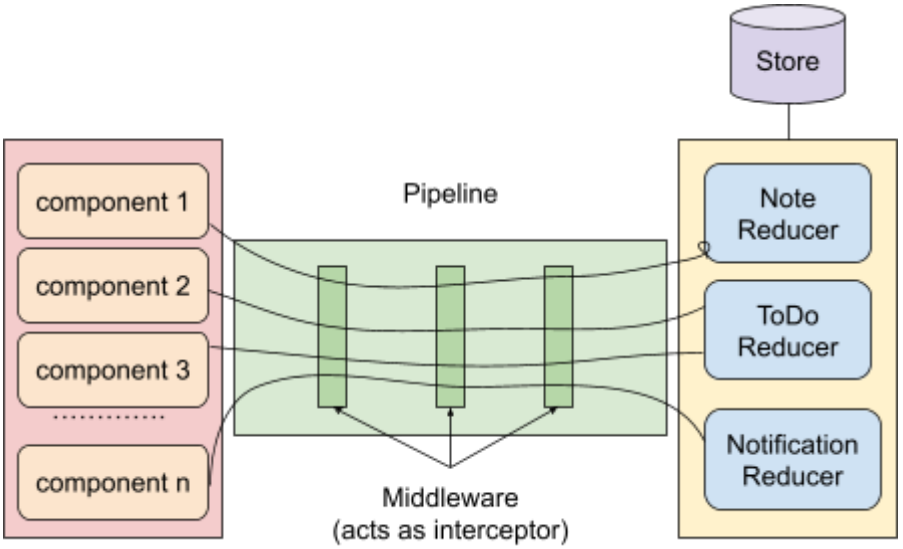
## The Problem

In large projects, it is important to keep track of what is happening inside the application at all times. This is where loggers come into play. A logger is a utility that captures information about various events that occur during the application's runtime, such as user actions, server responses, errors, and warnings.
In Redux, actions are dispatched from components to the store to update the state. Logging every action dispatched from the components to the store using console.log() can help debug and track the application flow. However, modifying every reducer to add console.log statements is not an ideal approach, as it can become difficult to manage when there are many components and reducers. One solution to this problem is to use middleware in Redux.

## Middleware - as a solution,

Middleware in Redux intercepts actions as they are dispatched to the store and can perform some additional logic on them before they reach the reducer.
One such middleware that can be used to log all actions is the loggerMiddleware.
In the case of Redux, middleware is added to the store as a pipeline, and each middleware in the pipeline can access the store, the next middleware in the pipeline, and the action being dispatched. When an action is dispatched from the component, it first passes through the middleware pipeline before reaching the reducer. Each middleware in the pipeline has the option to modify the action, dispatch additional actions, or perform other logic before passing it on to the next middleware using the next pointer. The concept of closure allows the middleware to access the Redux store and the next function even after the middleware function has completed execution.
It is important to note that every middleware in the pipeline should call the next function with the action as its argument to pass it along to the next middleware. This

ensures that all middleware in the pipeline can process the action before it reaches the reducer.




To solve the problem of logging every action, we can create a custom middleware that logs the action before passing it on to the next middleware. To use the middleware, we can add it to the middleware array in the Redux store.

```
export const loggerMiddleware = (store)=>{
  return function(next){
      return function(action){
          // log actions
          console.log("[LOG]: "+action.type+ " "+ new Date().toString());
          // call next middleware in pipeline.
          next(action);
          // log the modified state of app.
          console.log(store.getState());
      }
  }
}
```

## Calling an API

### Fetch function

To make an asynchronous call to an API in React using the fetch function, you can wrap the fetch function in a useEffect hook. Since the fetch function is asynchronous

and returns a promise, you can use either then and catch or async/await to handle the promise.

For Example, we make an API call to fetch todos from the server running on http://localhost:4100/api/todos. We then convert the response to JSON using the json() function, which also returns a promise. Finally, we log the parsed JSON data to the console.

```
useEffect(() => {
  fetch("http://localhost:4100/api/todos")
    .then(res=>res.json())
      .then(parsedJson=>{
        console.log(parsedJson);
      })
}, []);
```

## Axios Function

Axios is a commonly used library for making HTTP requests to an API. Axios provides an easy-to-use interface for making asynchronous requests, and it can be used in combination with Redux to manage state and handle API responses.

You can run `npm install axios` to install Axios.

For example, the useEffect hook is used to make an HTTP GET request to an API using Axios. The axios.get method takes the API URL as its argument and returns a promise that resolves with the response data. Once the response is received, the data is logged into the console. The [] as the second argument to useEffect ensures that the effect runs only once when the component mounts.

```
useEffect(() => {
  axios.get("http://localhost:4100/api/todos")
    .then(res=>
        {
          console.log(res.data);
        }
      );
}, []);
```

## How to manage API Data?

To manage API data in React Redux, there are two ways: either in the components or in the Redux itself.

### Using Components

If you choose to manage API calls in the components, you can use the useEffect hook to fetch the initial data from the API, and then once you have received the data, dispatch an action to update the Redux store.

For Example,

You can specify an action to update the Redux Store.

```
const todoSlice = createSlice({
  name:'todo',
  initialState:initialState,
  reducers:{
      setInitialState:(state, action)=>{
          state.todos=action.payload;
      },
 ...........
```

Then you can dispatch the action, once you receive data from the API.

```
useEffect(() => {
  axios.get("http://localhost:4100/api/todos")
    .then(res=>
        {
          console.log(res.data);
          disptach(actions.setInitialState(res.data));
        }
      );
}, []);
```

### Using Redux

When managing API data in Redux, it's important to note that we cannot make asynchronous calls from our reducer actions. This is because reducers are designed to be pure functions, meaning they should not have any side effects. They should only handle state updates based on the actions they receive. Instead, we can use the createAsyncThunk function provided by Redux Toolkit to handle async calls and update the Redux store accordingly. This allows you to manage API calls and state

updates in a centralized location in the Redux store, making it easier to manage your application's state and data flow.

## Create AsyncThunk

createAsyncThunk is a utility function provided by the Redux Toolkit that generates a Redux thunk. A thunk is a function that can be dispatched like a regular Redux action, but it can also contain asynchronous logic, such as fetching data from an API. The generated thunk function can be dispatched to trigger the async operation. When the async operation completes, it will automatically dispatch the appropriate action type based on the result.

For Example, When calling createAsyncThunk, you provide it with two parameters: a string that represents the base action type and a callback function that performs the asynchronous operation. The callback function passed to createAsyncThunk should be an async function. It should return the result of the operation as its resolved value or throw an error if the operation fails.

```
export const getInitialState = createAsyncThunk("todo/getInitialState",
    async (_,thunkAPI)=>{
    // async calls.
    try{
        const res = await axios.get("http://localhost:4100/api/todos")
        thunkAPI.dispatch(actions.setInitialState(res.data));
    }catch(err){
        console.log(err);
    }

})
```

When using createAsyncThunk, you may encounter an issue with the middleware in the Redux store. By default, the Redux Toolkit adds some middleware to the store, including the thunk middleware that enables using thunks like createAsyncThunk. However, if you have other middleware in your store that modifies the action or dispatch behavior, it may interfere with the behavior of createAsyncThunk. By using getDefaultMiddleware and spreading it into your middleware array, you ensure that

the necessary middleware for createAsyncThunk is included while also allowing you to add any other middleware that you need to the store.

```
middleware:[...getDefaultMiddleware(),loggerMiddleware]
```

### Advantages

- **Separation of Async Code:** By using createAsyncThunk, you can separate the async logic from the component code and move it to the Redux actions. This keeps the components lightweight and easier to manage and also simplifies testing.
- **Consistent pattern:** By using createAsyncThunk, you establish a consistent pattern for representing async operation states in your Redux store. This makes it easier to reason about your code and to debug it when issues arise.
- **Flexibility:** createAsyncThunk is flexible enough to allow you to customize its behavior if needed. For example, you can provide your own middleware to modify the behavior of the async operation or to add additional logic to the action dispatching process.

## Summarizing it

Let's summarize what we have learned in this Lecture:

- Learned about logging in projects.
- Learned what middlewares are?
- Learned how to call an API.
- Learned how to manage API data.
- Learned about createAsyncThunk.

## Some References:

- createAsyncThunk: link
- Axios: link