Redux

Issues with Prop Drilling

Prop drilling is a common problem in React applications where data is passed down through multiple layers of components via props. Prop drilling can lead to several issues:

1. **Storage Issue:** When large amounts of data is passed down through many layers of components via props, It can lead to issues with data storage and retrieval, as well as code maintainability and performance.
2. **Predictability of data**: Prop drilling can also make it difficult to predict where data comes from and how it will be used. It can be difficult to keep track of where the data is being used and where it is being modified. This can lead to issues with data consistency and can make it difficult to debug issues.
3. **Flow of Data:** Prop drilling can also make it difficult to pass data back up the component hierarchy.
4. **Data from multiple sources:** In complex applications, data may come from multiple sources, such as APIs or external services. Prop drilling can make it difficult to manage data and adds complexity to the application.

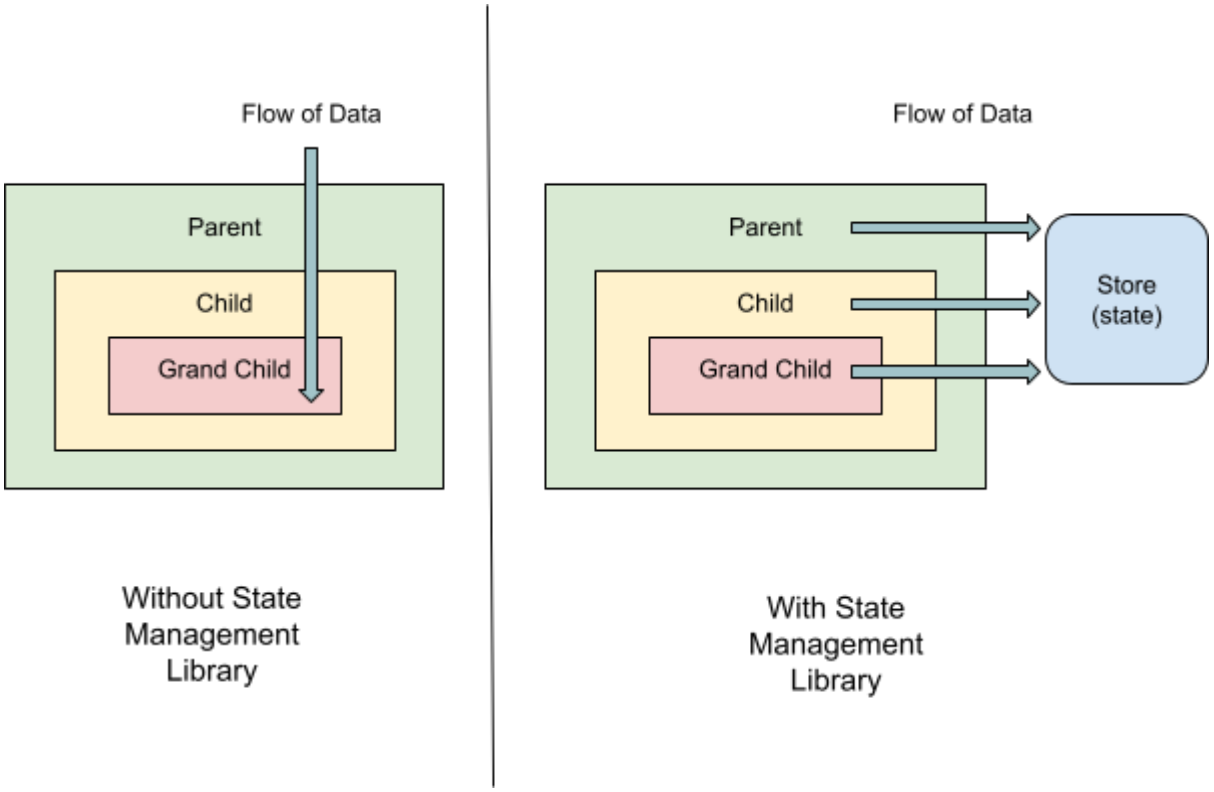
State Management

State management is a way to facilitate communication and sharing of data across components. State management libraries are tools used to manage and organize the state of an application predictably and efficiently. These libraries provide a set of rules and techniques for storing, retrieving, and updating application states. Advantages of using state management libraries are:

- **Centralized state:** State management libraries typically use a centralized store to manage the application state. This store is often implemented as a

JavaScript object that can be accessed and modified by components throughout the application.

- **Unidirectional data flow:** Data flows in a single direction in state management libraries, from the store to components. Components can update the store, but they cannot update other components directly.
- **Predictable state updates:** State management libraries provide a set of rules for updating the state, which helps to ensure that state changes are predictable and consistent across the application.
- **Immutable state:** Many state management libraries encourage the use of immutable data structures, which can help to prevent unintended side effects and make state updates more predictable.



Context API

Context API is a feature in React that provides a way to pass data through the component tree without having to pass props down manually at every level. It allows you to create a global state that can be accessed and modified by any component in the tree without the need for prop drilling. Context API can be useful for managing

states in cases where a small amount of data needs to be shared across multiple components, but is not ideal for larger and more complex state management needs.

Limitations

- **Overuse of context:** Overusing context can lead to a complex and difficult-to-manage application. Context should be used sparingly and only for data that needs to be shared across multiple components.
- **Designed for static content:** Context is designed for passing static data through the component tree, so it may not be the best choice for managing a dynamic state that changes frequently.
- **Re-renders the Context Consumers:** Whenever the value of the context changes, all the components that consume that context will re-render. This can lead to performance issues if the context value changes frequently.
- **Performance:** Context can cause performance issues if the context value is deeply nested and needs to be updated frequently.
- **Difficult to debug:** When an issue arises, debugging can be difficult since the data flow is not always clear. It can be difficult to trace where data is being passed and where it is being modified.
- **Difficult to extend and scale:** As an application grows in size and complexity, context can become difficult to manage and maintain.

Currying

Currying is defined as changing a function having multiple arguments into a sequence of functions with a single argument.
When currying a function in JavaScript, closures are used to retain the values of previous arguments that have been passed to the curried function. This is because each time a new argument is passed, a new function is returned that has access to the previous arguments.

For Example:

```
function sum(x){
    return function(y){
        return function(z){
            return x+y+z;
```

```
        }
    }
}

const sumXResult = sum(2);
const sumYResult = sumXResult(4);
const sumZResult = sumYResult(6);
console.log(sumZResult);
```

sum is a curried function that takes one argument x and returns another function that takes one argument y, which returns a third function that takes one argument z. The final function returns the sum of **x, y, and z**.

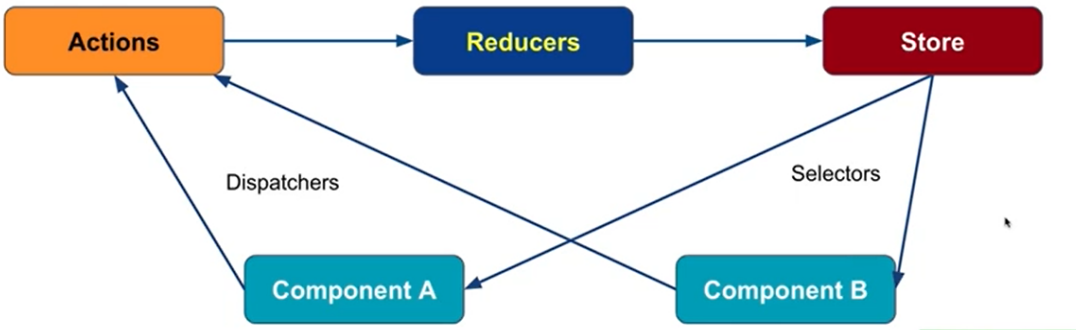
- When **sum(2)** is called, it returns a new function that takes one argument y. This returned function is assigned to **sumXResult**.
- When **sumXResult(4)** is called, it returns a new function that takes one argument z. This returned function is assigned to **sumYResult**.
- When **sumYResult(6)** is called, the final function is invoked, and it returns the sum of 2, 4, and 6, which is 12.

Finally, the result of **sumZResult** is printed to the console, which outputs 12.

Benefits of Currying

- Reusability: Currying allows you to create reusable functions by breaking down a function that takes multiple arguments into smaller functions that can be reused.
- Readability: Currying can improve the readability of the code by reducing the number of arguments passed to a function.
- Function Composition: Currying is useful for function composition, where the output of one function is the input to another function. It makes it easier to chain multiple functions together and create new functions that perform complex operations.

Redux Architecture



In Redux, the state of an application is represented as a single JavaScript object called the "store". The store is responsible for managing the application's state and updating the view when the state changes. The data flow in Redux is unidirectional, meaning that the data flows in one direction, from the view to the store and back to the view. This makes it easier to reason about the application's state and simplifies debugging.

Redux uses a "one-way data flow" app structure.

- The state describes the condition of the app at a point in time, and UI renders based on that state
- When something happens in the app:
    - The UI dispatches an action
    - The store runs the reducers, and the state is updated based on what occurred
    - The store notifies the UI that the state has changed
- The UI re-renders based on the new state

Installation of Redux

Run the following command to install Redux as a dependency for your project:

```
npm install redux
```

Components of the Redux Architecture

Store

The store is the central place where the application's state is stored. It is created using a reducer function that determines how the state should be updated based on the actions dispatched to the store.

For Example:

```
// store
const redux = require('redux');
const store = redux.createStore(todoApp);
```

Actions

Actions are plain JavaScript objects that describe what should happen in the application. They are created by calling action creator functions and are dispatched to the store using the store.dispatch() method. Payload is the data that needs to be transferred.

For Example:

```
// action types
const ADD_TODO = 'ADD_TODO';
const TOGGLE_TODO = 'TOGGLE_TODO';

// action creators
const addTodo = (text) => ({ type: ADD_TODO, text });
const toggleTodo = (index) => ({ type: TOGGLE_TODO, index });
```

Reducers

Reducers are pure functions that take the current state and an action as arguments and return a new state. They are responsible for updating the application's state based on the actions dispatched to the store.

For Example:

```
// reducer
```

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return {
        ...state,
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false,
          },
        ],
      };
    case TOGGLE_TODO:
      return {
        ...state,
        todos: state.todos.map((todo, i) => {
          if (i === action.index) {
            return {
              ...todo,
              completed: !todo.completed,
            };
          }
          return todo;
        }),
      };
    default:
      return state;
  }
}
```

View

The view is responsible for displaying the current state of the application to the user. It subscribes to changes in the store and updates the view when the state changes.

Dispatchers

a dispatcher is a function that receives an action object and dispatches it to the Redux store, triggering a state change. The dispatcher then passes the action object to the store's reducer function, which applies the update to the application state according to the rules defined by the application's business logic.

For Example:

```
// dispatch actions
store.dispatch(addTodo('Learn Redux'));
store.dispatch(addTodo('Build an app'));
store.dispatch(toggleTodo(0));

// get the current state
console.log(store.getState());
```

Selectors

Selectors are functions that extract specific data from the application's state. They provide a way to access and transform the data stored in the store.

For Example:

```
// selector
const getCompletedTodos = (state) => {
    return state.todos.filter((todo) => todo.completed === true);
    };
```

Principles of Redux

- Global app state is kept in a single store
- The store state is read-only to the rest of the app
- Reducer functions are used to update the state in response to actions

Summarizing it

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

- Learned about Issues with Prop Drilling.
- Learned about State Management.

- Learned about Context API.
- Learned about Redux and Architecture.
- Learned about components of Redux.
- Learned about the Principles of Redux.

Some References:

- Context: link
- Redux Documentation: link