9. State management

Introduction

State management has always been a vital yet somewhat dreadful aspect for working with React.
While managing state locally inside one class component was pretty much easy and straightforward since the beginning of the React, managing global state for multiple parts of the application was really painful. The main problem was that the React by default did not specify how to manage the data globally across the application.
Because of this, third party libraries like Redux or MobX came into being. If you wanted to use React in real applications you have to also learn these libraries and if you changed jobs, it was possible that the company was using some other library for the state management and you have to learn it again.

React solved these issues in version 16.3.0 (new context API) and in my opinion simplified this solution in version 16.8.0 (Hooks). Today's React solved, in my opinion, the state management using the context and the hooks and you don’t need other libraries anymore.

But, because these changes are relatively new (hooks are from february 2019) it exists a lot of legacy code out there using older versions of React. It is possible that you come across some of it in your React career.

That is because in today's lecture we will be talking mainly about the new concept of how to manage the data in your application, but we will speak also about the most used third party libraries. The new concept is the correct way and if you can choose, because you are starting a new project, then you should definitely use context API. In the end of the lecture, we are going to look at the most used third party library Redux, so you understand what a pain it was only a couple of years ago.

Local State Management

By the local state management we understand working with the data important only for one or a small number of components, for example the data inside some input.

Class

To create a local state the first option is to set this state inside a class. To initialize a state we can define it directly inside a constructor. After this you should never mutate **this.state** directly.

```
// You should never do this!!!
badExampleFunction = () => {
  this.state = { counter: 5 };
};
```

To change **this.state** we will be using a function **this.setState**, this function doesn't change the state immediately but creates a pending state transition. Thanks to this, when you synchronously change the state multiple times, these changes are coupled together and the component will change its state and re-render only once.
The function **this.setState** takes as an argument a new state or a function, this function is called with the old state and returns a new state. If you want to modify the state based on its previous data, you should use **this.setState** with a function as an argument**.**
Another important thing is, that if you want to use **this.setState** inside the function of a Class, the function must be bound to Class's **this**. This is because a classic function has its own **this**, therefore you must redirect it. With arrow function you don't have this problem, because arrow function inherits **this** from a parent.

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    // You can set a state like this only when you initialize it
    this.state = { counter: 0 };

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      counter: state.counter + 1,
    }));
  }

  handleArrowDoubleClick = () => {
    this.setState(state => ({
      counter: state.counter + 1,
    }));
    this.setState(state => ({
      counter: state.counter + 1,
    }));
  };

  render() {
    return (
      <div>
        <h2>You clicked {this.state.counter} times!</h2>
        <button onClick={this.handleClick}>Click!</button>
        <button onClick={this.handleArrowDoubleClick}>Double Click!</button>
      </div>
    );
  }
}
```

Hook

If you don't want to use classes inside your application (which I recommend), here's another example, how to create a state using the **useState** hook.

This is our example with a counter rewritten to a function with hooks:

```
const Counter = () => {
  const [counter, setCounter] = useState(0);

  function handleClick() {
    setCounter(counter => counter + 1);
  }

  const handleArrowDoubleClick = () => {
    setCounter(counter => counter + 1);
    setCounter(counter => counter + 1);
  };

  return (
    <div>
      <h2>You clicked {counter} times!</h2>
      <button onClick={handleClick}>Click!</button>
      <button onClick={handleArrowDoubleClick}>Double Click!</button>
    </div>
  );
};
```

As you can see this version is a little bit shorter and you don't have to bind functions.
The useState hook returns an array, where the first attribute is a state and the second one is a function which is used to modify the state, just like with a **this.state**, the function takes as an argument a new state or a function which is called with the old state and returns the new state.

Global State Management

By global state management, you can imagine some data that are important for the whole application or at least a big part of it. For example, a theme from material-ui is such a thing, or a localization state, that your app is using.
A little example with simple localization solution:

```
// src/localization/cs.ts
export default {
  'app.title': 'Príklad aplikácie',
  'app.description': 'Tento príklad slúži na pochopenie lokalizácie.',
  'switcherLabel': 'Změnit jazyk',
};

// src/localization/en.ts
export default {
  'app.title': 'Example application',
  'app.description': 'This example is used to understand localization.',
  'switcherLabel': 'Change language',
};

// src/localization/index.ts
import cs from './cs';
import en from './en';

export default { cs, en };
```

With this settings, you can use localization like this:

```
import texts from 'src/localization';
import Switcher from 'src/components/switcher;

export function LocalizationSwitcher {
  const [localization, setLocalization] = useState<'cs' | 'en'>();

  return (
    <Switcher
```

```
      onClick={()=> setLocalization(loc => loc === 'cs' ? 'en': 'cs')}
      label={texts[localization]['switcherLabel']}
    />
  )
}
```

If you click on a Switcher, the setLocalization function will change the language and the label property will change too.

But there is a problem, the information about what language is used is stored only in a function component LocalizationSwitcher. We need to somehow send this localization information to every component, that shows some text.
The problem is that in react, we ALWAYS send property data from the root component to its children, never the opposite way.

This is commonly called a **“top-down”** or **“unidirectional”** data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components “below” them in the tree.

That means, that if we want to send localization information to the whole database we must **lift state up.**

Lifting State Up

By lifting state up you can imagine storing a state information inside a first common ancestor of all the components that need this information. Even when this component has nothing to do with the state. In a previous example, which was localization, the common ancestor is going to be the *App.tsx*, because every part of the application needs the localization info.

Prop Drilling Problem

As you can imagine, this solution makes code messy, so it is harder to understand what the code does. For example in our tic-tac-toe application, if we want to add the localization into the app, to correctly show the game status, we need to drill the localization props from *App.tsx* all the way to *Status.tsx*.

App -> Home -> Board -> Status

The tic-tac-toe is a very small application, in a typical size app, there could be a path of more than 10 components. And there could be a couple of different global state data, which are needed in the app.

There must be a better way.

React Context

Until react's version 16.3.0 there was only an unstable version of react context, that was recommended not to use, because creators of the react were aware that there will be a breaking changes in this functionality. Developers have to use 3rd party libraries like Redux or to use an unstable react context and be ready to rework the whole functionality when the context stabilizes.
This happened in version 16.3.0. The concept is pretty easy and straightforward. When you need to create a global state, you'll find the common ancestor of all the components that need this state and create the state there. This is the same way as in a previous example. But the main difference is that you don't send this state to props of all the children of the component, instead you wrap these children components into a **ContextProvider**. This provider is holding the global state information and when some descendant needs this information, it uses a **ContextConsumer** to get this data, no matter how deep in the component hierarchy it is.

Here is a little code example: In order to use **ContextProvider** and **ContextConsumer**, you have to define this custom context that you will be using. So we create a file in the **src/localization** directory called LocalizationCtx.tsx:

```
// src/localization/LocalizationCtx.tsx
import { createContext } from 'react';

interface LocalizationCtxType {
  localization?: 'cs' | 'en';
  setLocalisation?: React.Dispatch<React.SetStateAction<'cs' | 'en'>>;
}
const defaultContext: LocalizationCtxType = {};
export const LocalizationCtx = createContext<LocalizationCtxType>(
  defaultContext,
);
```

Then all you need to do is to use this new **LocalizationCtx** context in your App.tsx:

```
import { LocalizationCtx } from './localization/LocalizationCtx';

const App: FC = () => {
  const classes = useStyles();
  const user = useLoggedInUser();
```

```
  const [localization, setLocalization] = useState<'cs' | 'en'>('cs');
  return (
    <MuiThemeProvider theme={ourTheme}>
      <LocalizationCtx.Provider value={{ localization, setLocalization }}>
        ...
      </LocalizationCtx.Provider>
    </MuiThemeProvider>
  );
};
```

Provider

You can also create a component called LocalizationProvider, where you extract state for which the context will be responsible. In this way you remove unnecessary code from the App.tsx:

```
// src/localization/LocalizationProvider.tsx
import React, { useState } from 'react';
import { LocalizationCtx } from './LocalizationCtx';

export function LocalizationProvider({
  children,
}: {
  children: React.ReactNode;
}) {
  const [localization, setLocalization] = useState<'cs' | 'en'>('cs');

  return (
    <LocalizationCtx.Provider value={{ localization, setLocalization }}>
      {children}
    </LocalizationCtx.Provider>
  );
}
```

The App.tsx:

```
import { LocalizationProvider } from './localization/LocalizationProvider';

const App: FC = () => {
  const classes = useStyles();
  const user = useLoggedInUser();
  return (
    <MuiThemeProvider theme={ourTheme}>
      <LocalizationProvider>...</LocalizationProvider>
    </MuiThemeProvider>
  );
};
```

It is possible that you are now thinking that this optimization is not an optimization at all. But when the global state data will be much more complex than just one useState and there will be a couple of Providers in the App.tsx you will understand that this way the App.tsx will stay as clean as possible.

Consumer

Now let's look at how the consumer part is done, again there are a couple of options how you can use the consumer.

The most straightforward way is to use Context.Consumer:

```
const Status: FC<Props> = ({ player, winner, onRestart, onConcede }) => {
  const classes = useStyles({ winner });
  useEffect(() => {
    /* ... */
  }, [player, winner]);
  return (
    <LocalizationCtx.Consumer>
      {({ localization = 'cs' }) => (
        <Paper className={classes.paper} square>
          {winner === 'Tie' ? (
            <Typography variant="h6" className={classes.text}>
              {texts[localization]['status.tie']} {/* <- use of localization */}
            </Typography>
          ) : (
            <>
              <Typography variant="h6" className={classes.text}>
                {texts[localization]['status.winner']}{' '}
                {/* <- use of localization */}
              </Typography>
              <PlayerIcon color={winner ? 'white' : 'black'}>
                {winner ?? player}
              </PlayerIcon>
            </>
          )}
        </Paper>
      )}
    </LocalizationCtx.Consumer>
  );
};
```

The main problem with this approach is that sometimes, you want to use global data before the JSX part of a component. With this solution you would have to create another component, where you send this data through props or start to write the application logic inside the JSX part, which is a really bad habit and I strongly do not recommend it.

```
const ExampleComponent: FC = () => {
  // I want to use the global data here, but I can't
  return (
    <ExampleCtx.Consumer>
      {props => {
        //Finally Now I can use the props
        // But i'm creating a complicated logic inside a JSX
        return (
          <>
            {/* So I have to create another Component */}
            <AnotherComponentWith data={props} />
          </>
        );
      }}
    </ExampleCtx.Consumer>
  );
};

const AnotherComponent: FC<{ data: ExampleCtxType }> = ({ data }) => {
  // Finally data are accessed in a correct part of a Function Component
  return <>{/* ... */}</>;
};
```

Hook useContext

As you can see, it is a lot of work just to extract data from a context, it must be a better way to do it, and fortunately there is one - using a **useContext hook**:

```
const ExampleComponent: FC = () => {
  // I want to use the global data here, so I use useContext.
  const props = useContext(ExampleCtx);
  return <></>;
};
```

IMPORTANT!

We can compare the way consumers work to an error throwing and catching.
When an error in an application appears, it starts to propagate up the hierarchy until it finds an error catch, if it doesn't find any catch handler, the application will collapse.

Context works very similarly.
When you use a **Consumer** or a **useContext** hook, it starts to look for a corresponding provider up the component hierarchy all the way to the root **App.tsx**, when it finds the provider it returns the data stored in this provider. If it doesn't find any corresponding provider, it returns the state that is defined as the property of a **createContext** function.
This can come handy when you need to work with a slightly different data of a global state in part of the application. You don't have to define identical context, you just create a different instance of the same context.
For example, your whole application is using a color theme consisting of a red color as primary and a blue color as a secondary. But now on one specific page, you have to switch these colors, so you define another MuiThemeProvider with a different theme.

On the next page, there is a code example with multiple providers and consumers. A couple of remarks regarding the code:

Let's assume that the Example Component is used in the App.tsx and there aren't any other Providers.

1. The useContext is used before any Provider is specified, so **the firstProps** contain only { index: 0}.
2. **The secondProps** are just under the first Provider, so they contain {index: 1, setIndex }.
3. **The thirdProps** are under the second Provider, so they contain {index: 2, setIndex }.
4. **The fourthProps** are the same as the secondProps.

If you look closely, the second provider has permanently defined the index as number 2, so there isn't any way to change this value.
Another thing is, that the setIndex is always reused from a previous provider, so no matter what setIndex you use, you always change the index value defined in the useState hook.

```tsx
interface ExampleCtxType {
  index: number;
  setIndex?: React.Dispatch<React.SetStateAction<number>>;
}

export const ExampleCtx = createContext<ExampleCtxType>({ index: 0 });

const Example: FC = () => {
  const [index, setIndex] = useState<number>(1);
  const firstProps = useContext(ExampleCtx);
  return (
    <ExampleCtx.Provider value={{ index, setIndex }}>
      <ExampleCtx.Consumer>
        {secondProps => (
          <div>
            <div>
              <ExampleCtx.Provider
                value={{ index: 2, setIndex: secondProps.setIndex }}
              >
                <ExampleCtx.Consumer>
                  {thirdProps => (
                    <div>
                      {thirdProps.index}
                      <ExampleCtx.Provider
                        value={{ index, setIndex: secondProps.setIndex }}
                      >
                        <ExampleCtx.Consumer>
                          {fourthProps => <div>{fourthProps.index}</div>}
                        </ExampleCtx.Consumer>
                        {/* ... */}
    </ExampleCtx.Provider>
  );
};
```

To sum it up, when you need to create a global state, follow this simple steps:

1. Create a file where you define a context, which will be storing global state data.
2. Create a ContextProvider for better readability of your code.
3. When you need to read (consume) the data stored in the context, I prefer to use a useContext hook over a Context.Consumer.

3rd party libraries

And now let's look at how was the global state management before the react's version 16.3.0. using the most popular third party library called **Redux**.
Because it is the *most popular*, *most used* and in my opinion one of the *best documented* solutions for a state management, there is the highest probability of you encountering it in your future react career. But if you come across a different library, don't worry because the way these libraries work is very similar. If you don't understand some parts of how Redux works, or you are just curious about it, look at the documentation. It is really well written, everything you find in this document about Redux is from that documentation.
Another great thing about Redux is that it has the ReduxDevTools Extension for your browser. The Redux DevTools Extension shows a history of the changes to the state in your Redux store over time. This allows you to debug your applications effectively, including using powerful techniques like "time-travel debugging".

Redux

Redux is a pattern and library for managing and updating application state, using events called "actions". It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.
The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur. Redux helps you deal with shared state management, but like any tool, it has tradeoffs. There are more concepts to learn, and more code to write. It also adds some indirection to your code, and asks you to follow certain restrictions. It's a trade-off between short term and long term productivity.

Let's start with a small recapitulation:

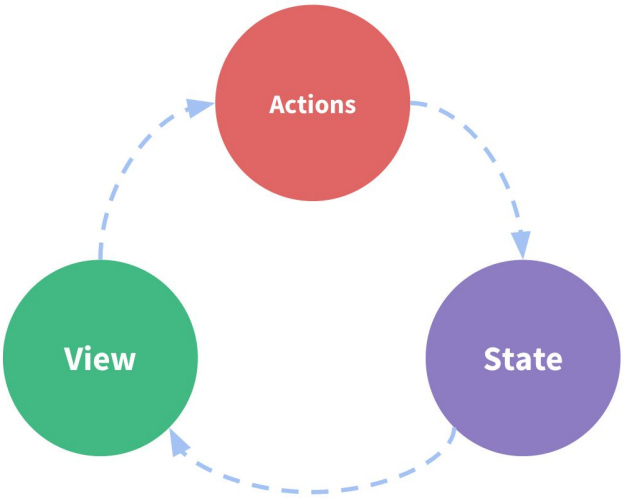
```
function Counter() {
  // State: a counter value
  const [counter, setCounter] = useState(0);

  // Action: code that causes an update to the state when something happens
  const increment = () => {
    setCounter(prevCounter => prevCounter + 1);
  };

  // View: the UI definition
  return (
    <div>
      Value: {counter} <button onClick={increment}>Increment</button>
    </div>
  );
}
```

- **The state**, the source of truth that drives our app;
- **The view**, a declarative description of the UI based on the current state
- **The actions**, the events that occur in the app based on user input, and trigger updates in the state

This is a small example of "**one-way data flow**", where **the state** describes the condition of the app at a specific point in time, **the UI** is rendered based on the state and when something happens (user clicks button), **the action** updates the state. The UI re-renders based on the new state.


Actions
View
State

However, the simplicity can break down when we have multiple components that need to share and use the same state, especially if those components are located in different parts of the application.
One way to solve this is to extract the shared state from the components, and put it into a centralized location outside the component tree. With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!
By defining and separating the concepts involved in state management and enforcing rules that maintain independence between views and states, we give our code more structure and maintainability.
This is the basic idea behind Redux: a single centralized place to contain the global state in your application, and specific patterns to follow when updating that state to make the code predictable.
One of these patterns is that Redux expects that all state updates are done immutably. We have to remember that, if we want to use Redux correctly.

Terminology

There are some important Redux terms that you'll need to be familiar with before we continue:

Actions

An action is a plain JavaScript object that has a **type** field. You can think of an action as an event that describes something that happened in the application.
The **type** field should be a string that gives this action a descriptive name, like "todos/todoAdded". We usually write that type string like "domain/eventName", where the first part is the feature or category that this action belongs to, and the second part is the specific thing that happened.
An action object can have other fields with additional information about what happened. By convention, we put that information in a field called payload.

A typical action object might look like this:

```
const addTodoAction = {
  type: 'todos/todoAdded',
  payload: 'Buy milk',
};
```

Action Creators

An **action creator** is a function that creates and returns an action object. We typically use these so we don't have to write the action object by hand every time:

```
const addTodo = (text: string) => {
  return {
    type: 'todos/todoAdded',
    payload: text,
  };
};
```

Reducers

A **reducer** is a function that receives the current *state* and an *action* object, decides how to update the state if necessary, and returns the new state: *(state, action) => newState*. You can think of a reducer as an event listener which handles events based on the received action (event) type.

Reducers must always follow some specific rules:

- They should only calculate the new state value based on the *state* and *action* arguments
- They are not allowed to modify the existing *state*. Instead, they must make immutable updates, by copying the existing *state* and making changes to the copied values.
- They must not do any asynchronous logic, calculate random values, or cause other "side effects"

The logic inside reducer functions typically follows the same series of steps:

- Check to see if the reducer cares about this action
  - If so, make a copy of the state, update the copy with new values, and return it
- Otherwise, return the existing state unchanged

Here's a small example of a reducer, showing the steps that each reducer should follow:

```
const initialState = { value: 0 };
function counterReducer(state = initialState, action) {
  // Check to see if the reducer cares about this action
  if (action.type === 'counter/increment') {
    // If so, make a copy of `state`
    return {
      ...state,
      // and update the copy with the new value
      value: state.value + 1,
```

```
    };
  }
  // otherwise return the existing state unchanged
  return state;
}
```

Store

The current Redux application state lives in an object called the **store**.

The store is created by passing in a reducer, and has a method called *getState* that returns the current state value:

```
import { configureStore } from '@reduxjs/toolkit';

const store = configureStore({ reducer: counterReducer });
console.log(store.getState()); // {value: 0}
```

Dispatch

The Redux store has a method called **dispatch**. The only way to update the state is to call *store.dispatch()* and pass in an action object. The store will run its reducer function and save the new state value inside, and we can call *getState()* to retrieve the updated value:

```
store.dispatch({ type: 'counter/increment' });
console.log(store.getState()); // {value: 1}
```

You can think of dispatching actions as "triggering an event" in the application. Something happened, and we want the store to know about it. Reducers act like event listeners, and when they hear an action they are interested in, they update the state in response.

We typically call action creators to dispatch the right action:

```
const increment = () => {
  return {
    type: 'counter/increment',
  };
```

```
};

store.dispatch(increment());
console.log(store.getState()); // {value: 2}
```

Selectors

Selectors are functions that know how to extract specific pieces of information from a store state value. As an application grows bigger, this can help avoid repeating logic as different parts of the app need to read the same data:

```
const selectCounterValue = state => state.value

const currentValue = selectCounterValue(store.getState())
console.log(currentValue); // 2
```

Redux Application Data Flow

Earlier, we talked about "one-way data flow", which describes this sequence of steps to update the app:

- State describes the condition of the app at a specific point in time
- The UI is rendered based on that state
- When something happens (such as a user clicking a button), the state is updated based on what occurred
- The UI re-renders based on the new state

For Redux specifically, we can break these steps into more detail:

Initial setup:

- A Redux store is created using a root reducer function
- The store calls the root reducer once, and saves the return value as its initial *state*
- When the UI is first rendered, UI components access the current state of the Redux store, and use that data to decide what to render. They also subscribe to any future store updates so they can know if the state has changed.

Updates:

1. Something happens in the app, such as a user clicking a button
2. The app code dispatches an action to the Redux store, like *dispatch({type: 'counter/increment'})*
3. The store runs the reducer function again with the previous *state* and the current *action*, and saves the return value as the new *state*
4. The store notifies all parts of the UI that are subscribed that the store has been updated
5. Each UI component that needs data from the store checks to see if the parts of the state they need have changed.
6. Each component that sees its data has changed forces a re-render with the new data, so it can update what's shown on the screen