GeekBrains

Virtual DOM. Material UI. PropTypes

React

Оглавление

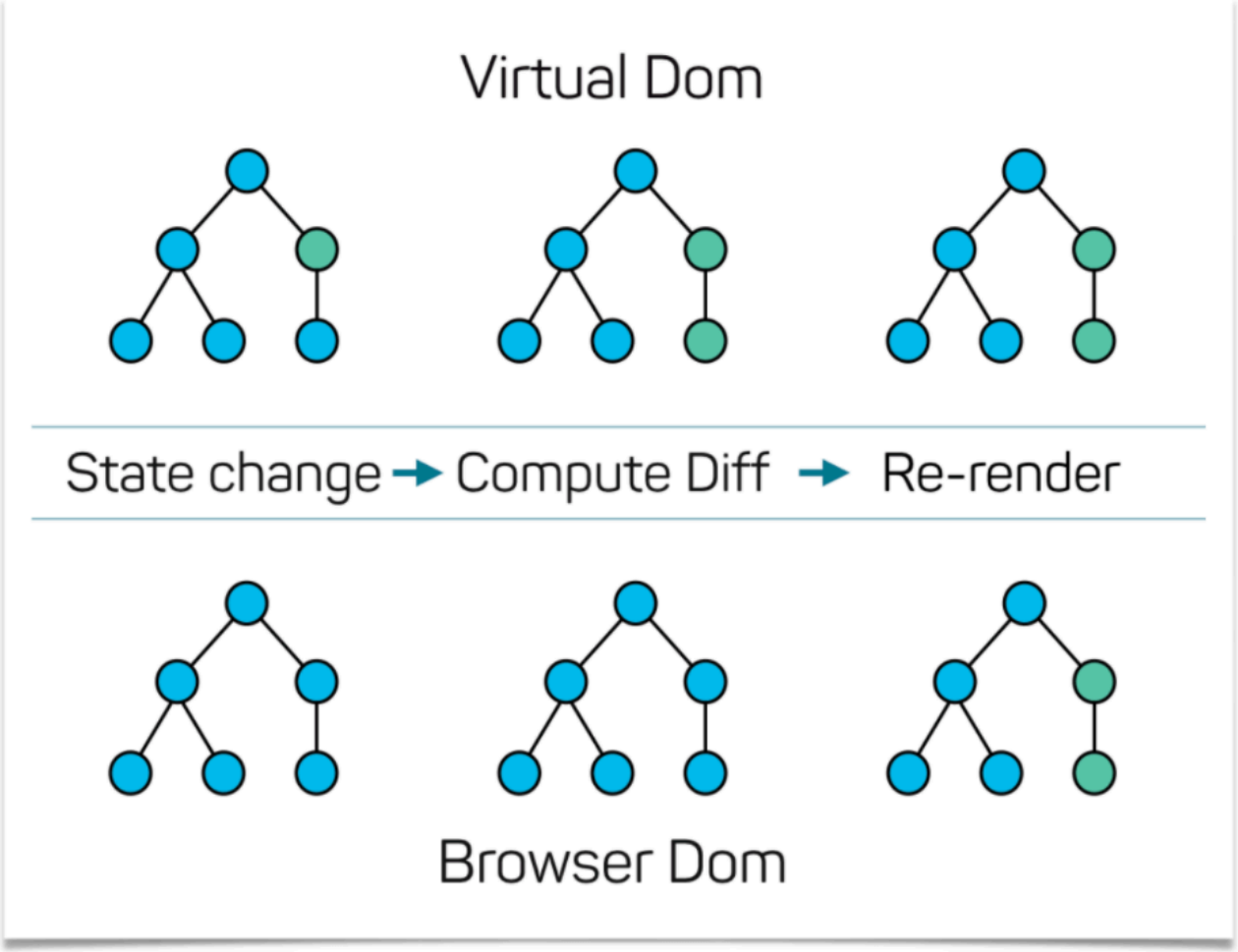
Виртуальный DOM

Реакт, при правильном его использовании, может работать весьма быстро. Это достигается за счет определенных внутренних механизмов оптимизации, одним из главных принципов которых является сокращение количества дорогостоящих операций. Под дорогостоящими операциями имеются в виду, в первую очередь, обращения и взаимодействия с DOM.

Когда пользователь первый раз открывает ваше приложение, Реакт создает объектное представление иерархии компонентов, называемое Virtual DOM (не путайте с shadow DOM). Virtual DOM, по сути - хранящийся в памяти объект, описывающий дерево элементов со всеми их свойствами. Актуальная версия Virtual DOM всегда хранится в памяти.

Примерно так выглядит объектное представление одного из компонентов - обратите внимание, что у элемента есть свойство **props**, описывающее все пропсы компонента, и свойство props.children - это специальный проп, сюда попадают все элементы, заключенные в тег элемента.

Когда один из компонентов обновляется (после получения новых пропсов или обновления стейта), Реакт вызывает необходимые методы жизненного цикла или соответствующие коллбэки хуков, и с помощью результирующего элемента (полученного как возвращаемый результат из функционального компонента или из метода render классового) строит новое дерево элементов - новый Virtual DOM. После того, как новый виртуальный DOM получен, Реакт проводит сравнение старой версии VirtualDOM с новой (обратите внимание - сравнение проводится между двумя версиями Virtual DOM, реальный DOM в этом механизме не используется). Проведя сравнение, Реакт определяет, какие элементы изменились, и производит обновление только тех элементов реального DOM, которые были изменены.



Таким образом, сводится к необходимому минимуму взаимодействие с DOM и перерисовка.

Согласование (reconciliation)

Алгоритм сравнения двух деревьев может быть достаточно сложным (на сегодняшний день сложность таких алгоритмов - не менее O(n3), то есть для относительно небольшого дерева из 100 элементов потребуется порядка миллиона операций). Для ускорения процесса сравнения Реакт использует эвристический алгоритм, построенный на следующих принципах:

1. Если корневой элемент некоторого поддерева изменяется - старое поддерево удаляется, вместо него создается новое. То есть, если один тип одного из элементов меняется (к примеру, вместо div - span), то все элементы, находившиеся внутри div, будут размонтированы, а затем смонтированы все элементы внутри span.

2. Элемент будет сохранен, если не изменился его проп key

Свойство key

Проп key - специальный проп, не влияющий на отображение элемента. Реакт использует его, чтобы определить, какие элементы не изменились - если при обновлении компонента key остался неизменным, элемент не будет размонтирован и смонтирован заново, а только обновится.

Свойство key используется, в основном, при рендере списков. Оно должно быть уникальным внутри данного списка - обычно используются id элементов массива. Допускается использовать индекс, если элементы списка не изменяют относительный порядок - то есть не будут переставлены с одной позиции на другую, не будет производится удаление или добавление элементов, кроме как из конца или в конец списка.

```
export function MessagesList() {
 const [messages, setMessages] = useState([
   { id: 'id1', text: "message 1" },
   { id: 'id2', text: "message 2" },
   { id: 'id3', text: "message 2" },
 ]);

 return messages.map((message) => <div
key={message.id}>{message.text}</div>);
}
```

Теперь, даже при удалении, к примеру, второго элемента из массива, React сможет переиспользовать уже отрендеренные элементы сообщений.

Material UI

Библиотеки компонентов

Одним из преимуществ переиспользуемых компонентов является возможность применять такие компоненты в совершенно различных местах одного проекта и даже различных проектах. Существует множество готовых библиотек таких

компонентов для React, самые популярные из них - AntDesign, React-Bootstrap, MaterialUI. Мы рассмотрим использование таких библиотек на примере MaterialUI.

Material UI. Установка и использование

Для установки данной библиотеки требуется ввести в терминале следующую команду:

npm install @mui/material @emotion/react @emotion/styled

Внимание!
Перед установкой библиотек проверяйте актуальную версию и порядок установки на официальном сайте или в официальном репозитории библиотеки

После этого мы можем использовать компоненты этой библиотеки в нашем коде:

```
import { useState, useCallback } from "react";
import { TextField } from '@mui/material';

export default function Example() {
 const [value, setValue] = useState("");

 const handleChange = useCallback((e) => {
   setValue(e.target.value);
 }, []);

 return (
   <TextField
     style={{ margin: '20px' }}
     id="outlined-basic"
     label="Outlined"
     variant="outlined"
     value={value}
     onChange={handleChange}
   />
 );
```

```
}
```

На официальном сайте представлено подробное описание доступных в библиотеке компонентов - их внешний вид, пропсы, которые они могут принимать и др. Аналогично можно использовать различные кнопки и другие элементы

Используя такие готовые компоненты, можно достаточно быстро создать сложный и красивый интерфейс.

Помимо готовых компонентов, Material UI также предоставляет возможность “темизации” вашего приложения - создания темы, то есть палитры цветов, которые будут использоваться компонентами Material UI, и применения этой темы.

Для использования тем необходимо сделать следующее:

1. Создать объект темы

```
import {
 ThemeProvider,
 useTheme,
 createMuiTheme,
} from "@material-ui/core/styles";
const theme = createMuiTheme({
 palette: {
   primary: {
     main: "#FF9800",
   },
   secondary: {
     main: "#0098FF",
   },
 },
});
```

2. Применить тему к приложению - для этого требуется обернуть корневой компонент вашего приложения в ThemeProvider

```
export default function Example() {
 const [value, setValue] = useState("");
```

```
  const handleChange = useCallback((e) => {
    setValue(e.target.value);
  }, []);

  return (
    <ThemeProvider theme={theme}>
      <TextField
        style={{ margin: "20px" }}
        id="outlined-basic"
        label="Outlined"
        variant="outlined"
        value={value}
        onChange={handleChange}
      />
      <SomeNestedChild />
      <Fab variant="extended">Click</Fab>
    </ThemeProvider>
  );
}
```

3. Получить объект темы в компоненте, используя хук useTheme:

```
function SomeNestedChild() {
  const theme = useTheme();
  return (
    <Button
      style={{
        backgroundColor: theme.palette.primary.main,
        borderColor: theme.palette.secondary.main,
      }}
    >
      Text
    </Button>
  );
```

```
}
```

При использовании темы MaterialUI и компонентов MaterialUI объект темы обязательно должен быть создан с использованием функции createMuiTheme и иметь как минимум свойство pallete.primary.main.

Компоненты можно удобно “подключать” к теме, используя функции makeStyles и createStyles.

```
const useStyles = makeStyles((theme) => createStyles({
 root: {
   backgroundColor: theme.palette.primary.main
 }
}));

export default function Example() {
 const classes = useStyles();
 return (
   <header className={classes.root}>Header</header>
 );
}
```

Функция makeStyles принимает аргументом другую функцию, которая возвращает результат вызова createStyles. В createStyles передается объект, содержащий стили для некоторого элемента. Обратите внимание, что в результирующем объекте будут использованы значения из темы, установленной ранее для вашего приложения через MuiThemeProvider.

При необходимости можно использовать пропсы компонента для кастомизации возвращенного объекта classes:

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';

const useStyles = makeStyles({
 root: {
   backgroundColor: 'red',
   color: props => props.color,
```

```
  },
});

export default function MyComponent(props) {
 const classes = useStyles(props);
 return <div className={classes.root} />;
}
```

Через объекты стилей Material UI возможно переопределять глобальные стили:

```
import { createStyles, makeStyles } from '@material-ui/core';

const useGlobalStyles = makeStyles(() =>
 createStyles({
   '@global': {
     html: {
       '-webkit-font-smoothing': 'antialiased',
       height: '100%',
       width: '100%'
     },
     body: {
       height: '100%',
       width: '100%'
     },
     '#root': {
       height: '100%',
       width: '100%'
     }
   }
 })
);
```

Используя специальный ключ @global, мы можем установить необходимые стили глобально для всего документа. Один из вариантов использования таких глобальных стилей - пустой компонент, подключающий их в приложение:

```
const GlobalStyles = () => { useGlobalStyles(); return null; };

const App = () => {
 return (
   <MuiThemeProvider theme={theme}>
       <GlobalStyles />
       <AnyComponents />
   </MuiThemeProvider>
 );
};
```

Компоненты Material UI используют свои, предопределенные стили и заранее определенные имена классов. Стили, применяемые к этим классам, мы можем переопределять, к примеру, с помощью тем. В некоторых случаях требуется переопределить конкретный предустановленный стиль для конкретного элемента. Для этого можно использовать специальное свойство overrides:

```
const theme = createMuiTheme({
 overrides: {
   MuiButton: {
     text: {
       color: 'white',
     },
   },
 },
});
```

При использовании такого свойства в теме цвет текста кнопки, импортированной из Material UI, будет заменен на белый.

При использовании overrides необходимо соблюдать следующую структуру темы: название стилей элемента (MuiButton) -> название правила стилей (text) -> стили (color: ‘white’).

Другой способ переопределить стиль элемента - свойство classes.

```
<Button
```

```
 classes={{
   root: newClasses.root,
   label: newClasses.label,
 }}
>
 ButtonLabel
</Button>
```

У компонента Button есть свои, предустановленные имена классов и стили как для корневого элемента, так и для текста. Однако, при такой записи стили, применяемые к этим классам, изменятся на переданные из newClasses (хотя сами имена классов, установленных для элементов, при этом останутся прежними).

Статические анализаторы кода. Статическая типизация

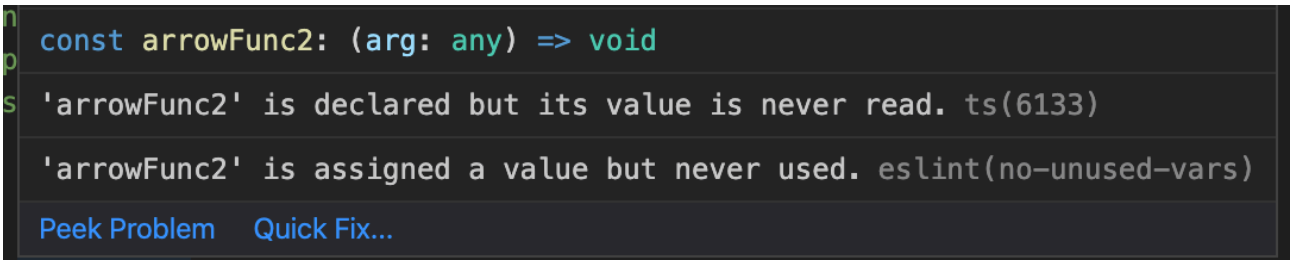
Статические анализаторы кода

Программисты придерживаются различных взглядов на форматирование кода - к примеру, одни предпочитают писать скобки вокруг аргумента стрелочной функции, а другие - нет:

```
const arrowFunc1 = (arg) => {};
const arrowFunc2 = arg => {};
```

Также могут возникать разногласия по поводу порядка импортов, использования деструктуризации, и т.п.

Для унификации кода и повышения его читабельности при работе в команде обычно используются т.н. линтеры - инструменты для статического анализа кода, которые помогают разработчикам придерживаться одного стиля кода. Самым популярным на сегодняшний день линтером в js является eslint - в приложениях, созданных через CRA он уже установлен и настроен. Вы могли видеть подобные предупреждения в IDE (здесь eslint указывает на то, что разработчик создал неиспользуемую переменную):



Помимо внедрения единого codestyle, eslint может выполнять, до определенной степени, проверку правильности самого кода - к примеру, правило eslint/exhaustive-deps проверяет, правильно ли указаны зависимости хуков.

Также часто в паре с eslint используется библиотека prettier - она также проводит статический анализ кода с целью внедрения единого стиля кода. Prettier предназначен для отслеживания таких вещей как ширина отступов, длина строки, наличие переносов, точек с запятой в конце строк и т.п.

Статическая типизация

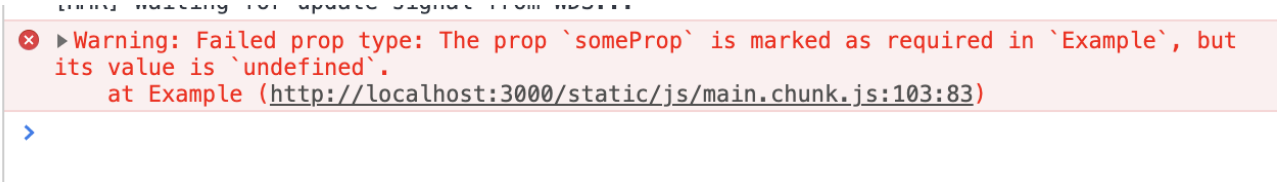
JavaScript - язык с динамической типизацией. Это значит, что до момента начала исполнения кода проверка правильности типов переменных невозможна. Одним из способов добавления статической типизации в JS является использование typescript. Typescript - достаточно сложный инструмент, и в этом курсе он не рассматривается.

Другим, более простым (но менее мощным) способом добавить проверку типов в приложениях на React является библиотека PropTypes. Использование ее выглядит следующим образом:

```
export default function Example(props) {...}

Example.propTypes = {
 someProp: PropTypes.bool.isRequired,
}
```

Компоненту устанавливается свойство propTypes (обратите внимание, название свойства начинается с маленькой буквы). В этом объекте перечисляются все пропсы, которые должен получать компонент, и указываются их типы. Если необходимо указать, что проп обязательный, используется свойство isRequred. При несовпадении типа полученного пропса с типом, указанным в propTypes (или отсутствии пропса, отмеченного как обязательный) React выбросит исключение:



PropTypes - достаточно быстрый и простой способ обеспечить проверку типов по крайней мере для пропсов компонентов.

Значение пропсов по умолчанию

Аналогично указанию типов пропсов, можно указывать их значения по умолчанию - если значение такого пропса будет undefined, подставится дефолтное значение:

```
Example.defaultProps = {
 color: 'white'
};
```

Такая запись может использоваться как с функциональным, так и с классовым компонентом (для классового также существует возможность указывать defaultProps и propTypes как статическое свойство класса).

Однако, в последнее время, с возрастающей популярностью функциональных компонентов, преимущественно встречается использование js-синтаксиса для указания значений аргументов функции по умолчанию:

```
export default function Example({ someProp, color = 'white' }) {
 /* ... */
}
```

Здесь аргумент функционального компонента - props - деструктурируется, и для отдельных пропсов указываются значения по умолчанию, т.е., если не передавать компоненту аргумент color, то он примет значение 'white'. Эта запись эквивалентна записи с использованием defaultProps из предыдущего примера.

Рефы

В прошлом уроке кратко рассматривался хук для создания рефа. Реф (ref - сокращение от reference - ссылка) - специальный объект с мутируемым свойством current. React гарантирует неизменность объекта рефа, при этом свойство current можно изменять, не вызывая при этом обновления компонента. Обычно рефы используются для создания ссылок на компоненты Реакт или элементы DOM:

```
const Input = (props) => {
 const inputRef = useRef(null);

 //...

 return (
   <input ref={inputRef} />
 )
}
```

После получения ссылки на элемент мы имеем возможность взаимодействовать с этим элементом - к примеру, в случае input, мы можем вызвать метод focus:

```
useEffect(() => {
 inputRef.current?.focus();
}, []);
```

Как правило, взаимодействие с элементами DOM через рефы - не лучшая практика. Вызов метода на элементе DOM - не декларативный, а императивный подход, тогда как Реакт предполагает написание кода именно в декларативном стиле. Кроме того, большинство операций с DOM проще и удобнее проводить через атрибуты элементов в JSX.

Исключение составляют фокус на элементе и установка скролла - зачастую такие действия необходимо произвести императивно, в ответ на некоторое событие.

Заметьте, что на схеме жизненного цикла компонента обновление рефов указано после рендера. До первого рендера значение свойства current объекта реф будет равно начальному значению - т.е., значению, переданному в useRef (или undefined, если аргумента нет). Кроме того, значение current может быть равно null, если элемент, на который должен ссылаться реф, не был отрисован в DOM (не был

смотирован или был размонтирован). Этим вызвана необходимость дополнительной проверки на существование current с помощью optional chaining.

Значение, переданное аргументом в useRef, а также ссылка на элемент DOM устанавливается именно в свойство current. Т.е., даже до первого рендера объект inputRef существует и имеет свойство current, однако оно будет равно (в приведенном выше примере) null.

Также в функциональных компонентах рефы зачастую используются для сохранения значения некоторых переменных, не влияющих напрямую на отображение компонента. К примеру, при такой записи

```
useEffect(() => {
 console.log('i will be called on every render!');
})
```

Коллбэк useEffect будет выполняться на каждом рендере, включая самый первый (на стадии монтирования). Однако, нам может быть необходимо выполнять данный колбэк именно при обновлении, но не при монтировании компонента. Для этого можно использовать флаг, установленный с помощью рефа:

```
const isFirstRender = useRef(true);

useEffect(() => {
 if (!isFirstRender) {
   console.log("i will be called on every render except first!");
 }
});

useEffect(() => {
 isFirstRender.current = false;
}, []);
```

Также в качестве примера такого использования рефа может послужить кастомный хук usePrevious

Для получения рефа на input, созданный с помощью TextField из Material-UI следует использовать проп inputRef.

Итоги урока

Что мы можем заметить, конечно работа с React очень вдохновляет и содержит в себе огромное количество полезных особенностей, конечно создание сложного проекта подразуемевает какой-то внешний вид и тут как раз нам на помощь приходит Material UI, получается что создание проекта выходит на новый уровень, осталось только добавить необходимый контент и результат превзойдет все ожидания.