Тема-10. React

Что тут есть?
React: что это; основная идея; о компонентах; JSX; props; стандартные хуки: useState, useEffect, useRef, useCallback, useMemo, useReducer, useContext; кастомные хуки (простые и с контекстом); React.Fragment; HOC; решение некоторых типичных подзадач; как создать приложение на React.

Что такое React

React – это библиотека (впрочем, иногда её называют и фреймворком, потому что она существенно влияет на организацию кода приложения) для построения фронтенда на JavaScript.

Библиотека создана разработчиками из компании, которую ~~с недавних пор нельзя называть~~, можно, но надо какую-то приписку специальную делать, я не прогуглил как.

Доступна по лицензии MIT. Разрабатывается с 2011 года. 19 млн. скачиваний в неделю.

Версии React

Библиотека до версии 16.8 и после существенно отличается по функциональности. Код, написанный на старых и новых версиях библиотеки может существенно отличаться.

Поэтому в рамках курса будет рассматриваться стиль программирования, используемый после версии 16.8.

Альтернативы React

Есть много других библиотек, которые можно использовать вместо React. Например, Vue, Ember.

Здесь другие библиотеки не будут рассматриваться, но в конце есть ссылка на статью по сравнению. Можно почитать.

React в других средах

Несмотря на то, что React предназначен в первую для использования на фронтенде веб-приложений, он также может быть использован и в других средах.

Например, React можно использовать вместе с Electron в десктопных приложениях.

Electron – фреймворк для написания десктоп приложений средствами HTML, CSS и JS.

Есть также библиотека React Native, предназначенная для мобильной разработки (Android и iOS).

Идея React

Основные идеи React:

- Декларативность – код на React описывает части интерфейса декларативно в виде иерархической структуры, обновление и управление состоянием React часто берет на себя.

- Компонентность – приложение на React состоит из компонентов – частей кода, независимых друг от друга, которые можно переиспользовать без изменения.

- Децентрализованность состояния – состояние хранится распределённо и независимо от DOM и каждый компонент использует только то, что ему нужно.

React: компоненты

Компонент – это базовое понятие React. Это часть интерфейса, которую можно переиспользовать.

Всё приложение на React состоит в большей мере именно из компонентов. Помимо них оно также может иметь чистые функции и хуки. О хуках далее.

React: компоненты

Каждый компонент имеет некоторое представление – то, что будет отображаться в интерфейсе.

Опционально, он может иметь код логики представления, иметь свою подсистему управления состоянием и использовать внешнее состояние.

React: компоненты

Как правило, компонент представляется обыкновенной функцией (в старых версиях использовались классы, но сейчас это реже встречается, так как функции имеют сильно меньше кода и их работа проще для понимания).

Простейший компонент может выглядеть так:

```
export default function () => "Hello world";
```

Это просто функция.

React: JSX

Компонент позволяет писать представление не только на JS, но и на специальном языке, близком к HTML по синтаксису, но позволяющем в то же время писать на JS – JSX.

Пример:

```
export const Hello = () => <h1>Hello world</h1>;
```

То, что возвращается, не является HTML в прямом смысле. Это синтаксический сахар для создания элементов в DOM.

```
export const Hello = () => React.createElement('h1', {}, 'Hello, world!');
```

React: JSX

Код на JSX можно не только возвращать из компонентов, но и хранить в переменных, вкладывать друг в друга и т. д. Пример:

```
export const Main = () => {
  const title = <h1>Hello world</h1>;

  return <div className="main-container">
    <div className="title-container">
      {title}
    </div>
    <div className="content-container">...</div>
  </div>;
};
```

React: JSX

Особенности JSX:

- атрибут class записывается как className (class зарезервировано в JS);

- все атрибуты из нескольких слов записываются в camelCase (onchange → onChange, stroke-width → strokeWidth и т. д.);

- inline-стили в JSX записываются как объекты на JS: <h1 style={{color: "red"}}>Hello world</h1>;

- чтобы использовать JS внутри JSX нужно заключить JS в фигурные скобки.

React: JSX

Пример использования JS внутри JSX:

```
const listItems = ["one", "two", "three", "four", "five", "six"];
export const SomeComponent = () => {
  const handleClick = event => alert(event.target.id);

  return <ul>
    {listItems.map(item =>
      <li
        id={`list-item-${item}`}
        onClick={handleClick}
      >{item}</li>
    )}
  </ul>
};
```

React: JSX

Компоненты можно вызывать друг из друга:

```
function Title () {
  const titleText = "Title of the page";
  return <h1>{titleText}</h1>;
}

export const Main = () => {
  Return <div className="main">
    <Title/>
  </div>
};
```

React: props

В компоненты можно пробрасывать данные извне. Они называются свойствами (properties, или props):

```
function Title (props) {
  return <h1 style={{color: props.color}}>{props.text}</h1>;
}

export const Main = () => {
  Return <div className="main">
    <Title color="#13466B" text="Title of the page"/>
  </div>
};
```

React: props

Свойства можно сразу разобрать:

```
function Title ({color, text}) {
  return <h1 style={{color}}>{text}</h1>;
}

export const Main = () => {
  Return <div className="main">
    <Title color="#13466B" text="Title of the page"/>
  </div>
};
```

React hooks: useState

Компонент может постоянно хранить некоторую динамическую информацию внутри себя. Для этого используется один из так называемых хуков. Хук (с англ. крючок) – это специальная функция, которая позволяет „подцепиться“ к логике, находящейся в каком-то отдельном месте.

Хуки могут быть как стандартными (некоторые из них будут рассмотрены), так и самописными (пример также будет рассмотрен). Объявить хук можно только на верхнем уровне компонента или кастомного хука (о них далее).

Первый стандартный хук, который будет рассмотрен – хук состояния useState (имена всех хуков начинаются с use).

React hooks: useState

Компонент может постоянно хранить некоторую динамическую информацию внутри себя. Для этого и используется хук useState:

```
import React, {useState} from "react";

export const Counter = () => {
  const [value, setValue] = useState(0);

  const increaseCounter = () => setValue(value + 1);

  return <span onClick={increaseCounter}>{value}</span>;
}
```

React hooks: useState

Состояние, которое хранит useState, временное. Если перезагрузить страницу, такое состяние будет стерто.

Чтобы хранить информацию постоянно, необходимо записывать его в localStorage или использовать какое-либо другое хранилище.

При изменении состояния компонент перерендеривается (то есть подхватывает новые данные и отображает их на странице).

Компонент также перерендеривается, если изменились его props или перерендерился его родитель.

React hooks: useEffect

Иногда нужно сделать так, чтобы при обновлении компонента, поменялось что-то в DOM документа или был выполнен какой-то другой сторонний эффект (side effect). Или наоборот нужно, чтобы часть логики вычислялась только при изменении определённых данных в компоненте, а не всех, какие в нем есть.

Для таких задач существует стандартный хук useEffect.

React hooks: useEffect

Синтаксис useEffect следующий:

```
useEffect(() => {
  //side effect code
}, [/*dependency array*/]);
```

В функции, которая передается первым аргументом, пишется код того, что нужно сделать. Второй аргумент содержит массив тех данных, при изменении которых нужно выполнять этот эффект.
Если массив не передать, эффект будет выполняться каждый перерендер. Если передать пустой, эффект выполнится один раз.

React hooks: useEffect

Пример эффекта, который выполняется только один раз:

```
const [serverData, setServerData] = useState();

useEffect(() => {
  axios.get("/api/v1/data").then(data => setServerData(data));
}, []);
```

Здесь отправляется запрос, как только компонент отрендерился, и результат запроса записывается в состояние компонента.

React hooks: useEffect

Пример эффекта, который выполняется каждый раз:

```
useEffect(() => {
  document.title = `Вы нажали ${count} раз`;
});
```

Всякий раз, когда состояние компонента меняется в заголовок документа записывается новое число нажатий (код кнопки и состояния опущен здесь, но он подобен примеру выше).

React hooks: useEffect

Пример эффекта, который выполняется иногда:

```
export const Result = ({a, b, name, ...props}) => {
  const [functionResult, setFunctionResult] = useState(0);

  useEffect(() => {
    setFunctionResult(computeSomeHeavyCalculations(a, b));
  }, [a, b]);

  return <div styles={props.styles}>
    <span>Значение функции {name} = {functionResult}</span>
  </div>;
};
```

Если изменится name или style, тяжелого перевычисления не произойдет.

React hooks: useRef

Хук useRef позволяет записать некоторое изменяемое значение в переменную или константу (значение записывается в специальный объект, поэтому и можно использовать константу).

Фактически запись:

```
const data = useRef({someKey: 3});
```

это то же самое, что и просто создание объекта

```
const data = {someKey: 3};
```

С той лишь разницей, что useRef будет хранить ссылку на один и тот же объект, а не каждый ререндер на разный.

React hooks: useRef

Упомянутый выше способ использования useRef не самый частый. Чаще useRef используется, чтобы сохранить некоторый DOM-узел в переменную, чтобы иметь к нему доступ из кода напрямую, без использования функций наподобие getElementById.

```
const inputRef = useRef(null);
...
return <div>
  <input ref={inputRef} .../>
</div>
```

React hooks: useCallback

Хук useCallback позволяет создать мемоизованный коллбек.

Мемоизация – это "заморозка" функции или результатов её вычисления. То есть при мемоизации функции будут созданы дампы всех внешних данных, которые в ней используются. То есть она не будет пересоздаваться каждый перерендер.

Коллбек (или функция обратного вызова) – это функция, которая передается как параметр в другую функцию.

React hooks: useCallback

Использовать useCallback достаточно просто:

```
const getSomething = useCallback(() => {
   //тут код, в котором используются переменные: a, b, data
}, [a, b, data]);
```

В данном случае можно вызвать функцию getSomething() напрямую, но она будет пересоздаваться только если хотя бы одна из переменных a, b, data изменится.

React hooks: useMemo

Хук useMemo подобен предыдущему, но используется для мемоизации значения, то есть чтобы хранить результат выполнения функции, а не саму. Например:

```
const memoizedValue = useMemo(() => getValue(a, b), [a, b]);
```

Функция не будет перевычисляться при ререндере, если переменные a и b не изменились.

useCallback(fn, deps) — это эквивалент useMemo(() => fn, deps)

React hooks: useReducer

В тех случаях, когда нужно хранить в компоненте какое-то сложное состояние (например, объект объектов), удобно использовать вместо хука useState хук useReducer.

Этот хук позволяет задать специальную функцию, управляющую обновлением состоянием и, соответственно, обновлять состояние по своим правилам, передавая различные команды (которые реализуются в функции, управляющей состоянием).

React hooks: useReducer

Пример использования useReducer:

```
function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return <>
    Count: {state.count}
    <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    <button onClick={() => dispatch({type: 'increment'})}>+</button>
  </>;
}
```

React hooks: useReducer

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}
```

React hooks

Есть и другие стандартные хуки, но они не будут рассмотрены здесь (кроме хука useContext, о котором далее). С ними можно ознакомиться в документации по React, ссылка на которую есть в конце презентации.

React custom hooks

React позволяет создать свои собственные хуки. Фактически кастомный хук – это компонент, но который возвращает вместо части интерфейса функции и данные для использования в других компонентах. Соответственно, кастомные хуки нельзя отрендерить.

Имя кастомного хука всегда должно начинаться с use, иначе у React могут возникнуть проблемы с интерпретацией.

React custom hooks

Кастомные хуки часто используются в следующих случаях:

- когда нужно вынести логику из компонента отдельно (чтобы скрыть или улучшить читабельность кода);

- когда нужно разделить логику или данные между компонентами, которые не находятся друг с другом в отношении „предок-потомок“ (то есть, нельзя использовать props, чтобы что-то передать).

React custom hooks

Кастомные хуки можно разделить на два вида:
- простые;
- с контекстом.

Простые хуки можно считать обычными функциями, которые выносят логику отдельно. При использовании таких хуков для каждого компонента, который использует хук, будет создана своя копия хука (копии хука не будут связаны друг с другом).

React custom hooks

Пример простого хука:

```
export default function useUserText () {
   const [userText, setUserText] = useState("");
   const [userColor, setUserColor] = useState("white");

   const changeUserText = event => setUserText(event.target.value);
   const changeUserColor = event => setUserColor(event.target.value);

   return {userText, changeUserText, userColor, changeUserColor}
}
...
const {userText, changeUserText, userColor, changeUserColor} = useUserText();
```

React custom hooks

Кастомный хук с контекстом хранит данные глобально, то есть их можно использовать по всему приложению в разных компонентах и состояние из этого хука будет одно и тоже.

Соответственно, если в одном компоненте обновить состояние хука, все дургие компоненты, которые его используют, получат это новое состояние.

React custom hooks

Чтобы создать хук с контекстом используется следующая конструкция:

```
const UserViewBoxContext = React.createContext();

export const UserViewBoxProvider = ({children}) => {
   const [get, set] = useState("0 0 0 0");
   return <UserViewBoxContext.Provider value={{get, set}}>
      {children}
   </UserViewBoxContext.Provider>;
};

export const useUserViewBox = () => React.useContext(UserViewBoxContext);
```

React custom hooks

Вызвать такой хук в компоненте можно, например, так:

```
const userViewBox = useUserViewBox();
...
userViewBox.set(...); // установка значения
const viewBox = userViewBox.get; // получение значения
```

React hooks: useContext

Контекст – это специальный объект в React, который предназначен для хранения данных, общих для всех компонентов. Создается он так (код тот же, что и в примере с кастомным хуком).

```
const UserViewBoxContext = React.createContext();
```

React hooks: useContext

Чтобы контекст можно было использовать в компонентах, их необходимо обернуть в так называемый провайдер контекста. Для этого нужно поместить компонент внутри провайдера:

```
<UserViewBoxContext.Provider value={{get, set}}>
   {components}
</UserViewBoxContext.Provider>;
```

В качестве параметра value передается та информацию, которая будет доступна компонентам через этот контекст.

React hooks: useContext

Поэтому, чтобы использовать кастомный хук с контекстом, необходимо обернуть корневой компонент, потомки которого будут использовать хук, в провайдер этого контекста.

React.Fragment

Иногда бывает нужно возвратить несколько объектов из компонента. Для этого можно использовать специальный элемент React.Fragment:

```
...
return <React.Fragment>
    <div></div>
    <div></div>
</React.Fragment>;
```

Вместо <React.Fragment></React.Fragment> можно писать <></>

React: High Ordered Component

Иногда бывает нужно добавить какую-то функциональность ко многим компонентам сразу, не меняя их логику. Так, например, можно добавить подсказки к элементам.

Для такой задачи существуют так называемый High Ordered Component (HOC, компонент высшего порядка) – компонент, принимающий другой компонент в качестве параметра и возвращающий новый компонент, образованный из старого и чего-то дополнительного.

React: High Ordered Component

Выглядит подобный компонент, например, так:

```
export function hintModeTooltipHOC(ChildComponent, hintModeKey) {
   return props => {
      return <Tooltip content={hintModeKey}>
        <ChildComponent {...props}/>
      </Tooltip>;
   }
}
```

React: High Ordered Component

Использовать HOC можно так:

```
function FinalButton(props) {
    return <input type="button"/>;
}

export default hintModeTooltipHOC(FinalButton, "final button hint");
```

Разработка на React

Есть некоторые шаблонные задачи, которые часто возникают при разработке на React и под которые придуманы типичные решения. Здесь будут рассмотрены некоторые из них.

Разработка на React

Часто бывает нужно создать так называемые управляемые элементы (controlled elements). Это такие элементы, ввод информации пользователем в которые сохраняется в приложении и доступен для программной работы.

Например, пользователь вводит информацию в поле „логин“ и нужно проверить, можно ли вводить такой логин.

Разработка на React

Как это сделать:

```
export const LoginInput (props) {
  const [userInput, setUserInput] = useState();

  const updateUserInput = event => {
    if (isLoginValid(event.target.value)) {
      setUserInput(event.target.value);
    }
  };

  return <input type="text" value={userInput} onChange={updateUserInput}/>;
}
```

Разработка на React

Иногда бывает нужно, чтобы какие-то данные были нужны в двух соседних компонентах, в одном из которых эти данные создаются, а во втором используются. В этом случае используется прием, называемый поднятием состояния до ближайщего общего предка (но если данные используются много где, лучше написать кастомный хук с контекстом).

Например, есть компонент меню, к в котором устанавливаются фильтры, и есть компонент, который отражает контент по этим фильтрам.

Разработка на React

Как это сделать? Создаем корневой элемент, который будет хранить состояние. Передаем из него состояние фильтра в компонент, который отображает контент (ContentContainer). Передаем состояние фильтра и функцию обновления фильтра в компонент, который настраивает фильтр (Menu).

```
function Page () {
  const [filter, setFilter] = useState({});
  return <>
    <Menu filter={filter} setFilter={setFilter}/>
    <ContentContainer filter={filter}/>
  </>;
}
```

Разработка на React

Создаем компонент управления фильтрацией:

```
function Menu ({filter, setFilter}) {
  const updateName = event => setFilter({...filter, name: event.target.value});
  const updatePrice = event => setFilter({...filter, price: event.target.value});

  return <div>
    <input type="text" value={filter.name || ""} onChange={updateName}/>
    <input type="number" value={filter.price || ""} onChange={updatePrice}/>
  </div>;
}
```

Разработка на React

Создаем компонент, отображающий контент:

```
function ContentContainer ({filter}) {
  return <div>
    {allContent.map(item =>
      item.name.startsWith(filter.name)
      && item.price < filter.price
      && <ContentItem data={item}/>
    )}
  </div>;
}
```

Разработка на React

Иногда неизвестно, какими будут дочерние элементы у компонента. В этом случае применяется прием render props (отрендерить свойства).

Например, имеется компонент – контейнер для элементов (BeautifulContainer). Элементами могут быть книги (рендерятся через компоненты Book), научные статьи (Paper) или оцифрованные древние письменные источники (AncientSource).

Разработка на React

Как это может быть сделано? Создается компонент контейнера, который просто рендерит что-то, что передается извне, оборачивая это в какую-то свою обертку.

```
const BeautifulContainer = ({children}) => {
  return <div className="super-beautiful-style">
    {children}
  </div>;
}
```

Разработка на React

Использоваться это может так:

```
const PageContent = () => {
  return <>
    <BeautifulContainer>
      {books.map(book => <Book data={book}/>)}
    </BeautifullContainer>
    <BeautifulContainer>
      {papers.map(paper => <Paper data={paper}/>)}
    </BeautifullContainer>
    <BeautifulContainer>
      {ancientSourcer.map(ancientSource => <AncientSource data={ancientSource}/>)}
    </BeautifullContainer>
  </>;
}
```

Разработка на React

А где children? Это стандартное название свойства. Оно всегда содержит то, что было передано в качестве потомков компонента.

```
<BeautifulContainer>
  {/*Всё, что тут есть, будет в children у BeautifulContainer*/}
  {books.map(book => <Book data={book}/>)}
</BeautifullContainer>
```

Как создать приложение на React

Чтобы компоненты рендерились в DOM и соответственно, отображались в документе, нужно указать корневому компоненту, где он будет находиться. Это делается следующим образом:

```
import ReactDOM from "react-dom";

ReactDOM.render(
   <RootComponent/>,
   document.getElementById("root")  // элемент #root должен существовать
);
```

Полезные ссылки

- https://reactjs.org/ – сайт React;
- https://kyleshevlin.com/use-encapsulation/ – статья про инкаспуляцию с помощью кастомных хуков;
- https://create-react-app.dev/ – простой способ создать приложение на React;
- https://kruschecompany.com/ember-jquery-angular-react-vue-what-to-choose/ – сравнение разных библиотек, которые делают примерно то же, что и React.

Следующая лекция тоже про React (но более расширенно, про использование разных библиотек, упрощающих типичные задачи)