



ReactJS vol. 2

Функциональные компоненты

- Представляет собой функцию, принимающую на вход пропы и возвращающую JSX элемента.

```
function Welcome(props) {  
  return <h1>Привет, {props.name}</h1>;  
}
```

```
const Welcome = ({name}) => {  
  return <h1>Привет, {name}</h1>;  
}
```

Загрузка по требованию

- `React.lazy()` принимает функцию, которая должна вызвать динамический `import()`. Результатом возвращённого Promise является модуль, который экспортирует по умолчанию React-компонент (`export default`).

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

- Функция автоматически загрузит бандл, содержащий `OtherComponent`, когда этот компонент будет впервые отрендерен.
- Компонент с «ленивой загрузкой» должен рендериться внутри компонента `Suspense`, который позволяет нам показать запасное содержимое (например, индикатор загрузки) пока происходит загрузка ленивого компонента.



```
import React, { Suspense } from 'react';
```

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

```
function MyComponent() {  
  return (  
    <div>  
      <Suspense fallback={<div>Загрузка...</div>}>  
        <OtherComponent />  
      </Suspense>  
    </div>  
  );  
}
```

Предохранители

- Если какой-то модуль не загружается (например, из-за сбоя сети), это вызовет ошибку. Вы можете обрабатывать эти ошибки для улучшения пользовательского опыта с помощью Предохранителей. После создания предохранителя, его можно использовать в любом месте над ленивыми компонентами для отображения состояния ошибки.
- Предохранители — это компоненты React, которые отлавливают ошибки JavaScript в любом месте деревьев их дочерних компонентов, сохраняют их в журнале ошибок и выводят запасной UI вместо рухнувшего дерева компонентов.
- Классовый компонент является предохранителем, если он включает хотя бы один из следующих методов жизненного цикла: `static getDerivedStateFromError()` или `componentDidCatch()`. Используйте `static getDerivedStateFromError()` при рендеринге запасного UI в случае отлова ошибки. Используйте `componentDidCatch()` при написании кода для журналирования информации об отловленной ошибке.



Порталы

- Обычно, когда вы возвращаете элемент из рендер-метода компонента, он монтируется в DOM как дочерний элемент ближайшего родительского узла, но иногда требуется поместить потомка в другое место в DOM:

```
render() {  
  ...  
  return ReactDOM.createPortal(  
    this.props.children,  
    domNode );  
}
```




Контекст

- Контекст позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на промежуточных уровнях.
- Создание контекста (useContext для использования в функциональных компонентах):

```
const MyContext = React.createContext(defaultValue);
```

- Когда React рендерит компонент, который подписан на этот объект, React получит текущее значение контекста из ближайшего подходящего Provider выше в дереве компонентов.
- Аргумент defaultValue используется только в том случае, если для компонента нет подходящего Provider выше в дереве.



Context.Provider

`<MyContext.Provider value={/* некоторое значение */}>`

- Каждый объект Context используется вместе с Provider компонентом, который позволяет дочерним компонентам, использующим этот контекст, подписаться на его изменения.
- Компонент Provider принимает проп value, который будет передан во все компоненты, использующие этот контекст и являющиеся потомками этого компонента Provider. Один Provider может быть связан с несколькими компонентами, потребляющими контекст. Так же компоненты Provider могут быть вложены друг в друга, переопределяя значение контекста глубже в дереве.

Context.Consumer

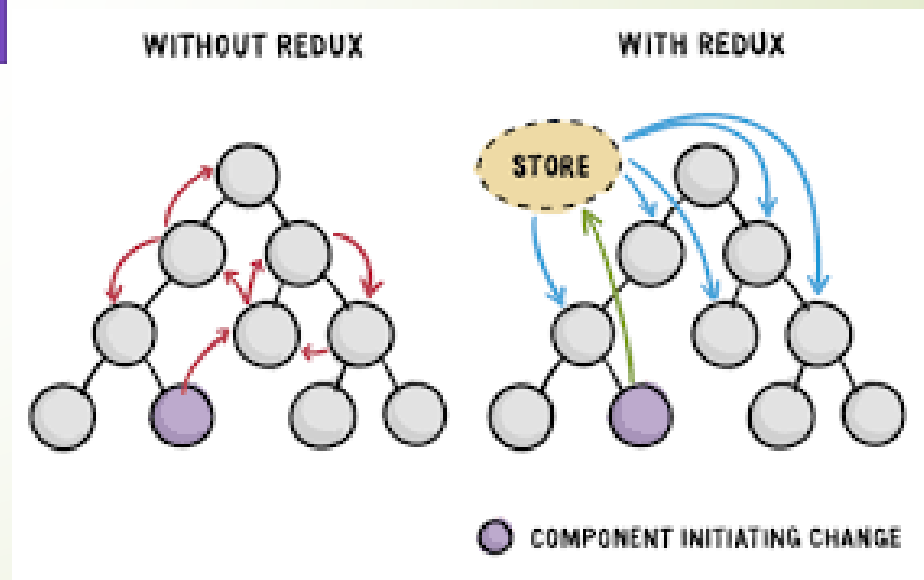
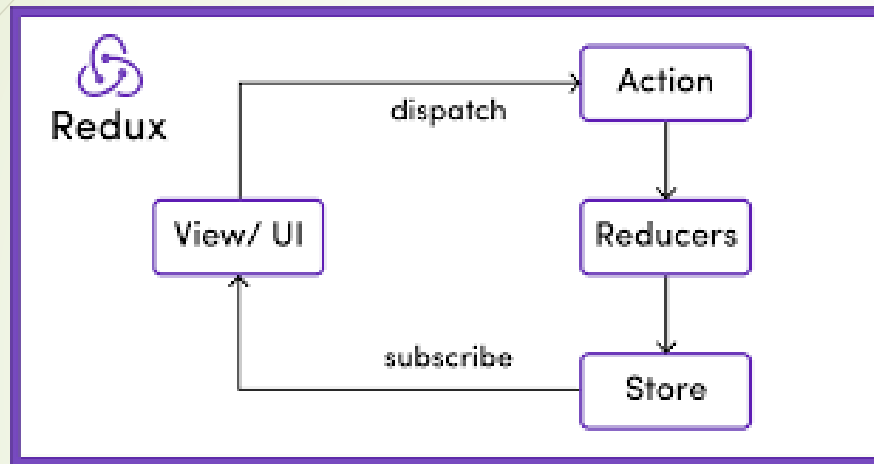
- Consumer — это React-компонент, который подписывается на изменения контекста. В свою очередь, это позволяет вам подписаться на контекст в функциональном компоненте.
- Consumer принимает функцию в качестве дочернего компонента. Эта функция принимает текущее значение контекста и возвращает React-компонент.

```
<MyContext.Consumer>
```

```
  {value => /* отрендерить что-то, используя значение контекста */}
```

```
</MyContext.Consumer>
```

Redux (а еще лучше mobx)





Ref (рефы)


- Ремы дают возможность получить доступ к DOM-узлам или React-элементам, созданным в рендер-методе.
- Ситуации, в которых использование рефов является оправданным:
 1. Управление фокусом, выделение текста или воспроизведение медиа.
 2. Императивный вызов анимаций.
 3. Интеграция со сторонними DOM-библиотеками.
- Избегайте использования рефов в ситуациях, когда задачу можно решить декларативным способом!

- 
- Создание рефа (useRef для функциональных компонентов):

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.myRef = React.createRef();  
  }  
  render() {  
    return <div ref={this.myRef} />;  
  }  
}
```

- Доступ:

```
this.myRef.current.focus();
```



Хуки (hooks)

- Хуки — это функции, с помощью которых можно использовать состояние и методы жизненного цикла React из функциональных компонентов. Хуки не работают внутри классов — они дают возможность использовать React без классов.
- Правила использования:
 1. Хуки следует вызывать только на верхнем уровне. Не вызывайте хуки внутри циклов, условий или вложенных функций.
 2. Хуки следует вызывать только из функциональных компонентов React. Не вызывайте хуки из обычных JavaScript-функций. Есть только одно исключение, откуда можно вызывать хуки — это пользовательские хуки.

Хук СОСТОЯНИЯ

- Вызов `useState` возвращает две вещи: текущее значение состояния и функцию для его обновления. Единственный аргумент `useState` — это начальное состояние.

```
const [count, setCount] = useState(0);
```

- Хук состояния можно использовать в компоненте более одного раза
- Обращение к состоянию:

```
<div>{count}</div>
```




Хук эффекта

- Хук эффекта даёт возможность выполнять побочные эффекты в функциональном компоненте. Побочными эффектами в React-компонентах могут быть: загрузка данных, оформление подписки и изменение DOM вручную (хук `useEffect` представляет собой совокупность методов `componentDidMount`, `componentDidUpdate`, и `componentWillUnmount`)

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

- Если эффект возвращает функцию, React выполнит её только тогда, когда наступит время сбросить эффект



Пропуск эффектов



```
componentDidUpdate(prevProps, prevState) {  
  if (prevState.count !== this.state.count) {  
    document.title = `Вы нажали ${this.state.count} раз`;  
  }  
}
```

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

Пользовательские хуки

- Пользовательский хук — это JavaScript-функция, имя которой начинается с «use», и которая может вызывать другие хуки.

```
function useFriendStatus(friendID) {  
  const [isOnline, setIsOnline] = useState(null);  
  useEffect(() => {  
    function handleStatusChange(status) {setIsOnline(status.isOnline);}  
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);  
    return () => {  
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);  
    };  
  });  
  return isOnline;  
}
```



```
function FriendStatus(props) {  
  const isOnline = useFriendStatus(props.friend.id);  
  if (isOnline === null) {  
    return 'Загрузка...';  
  }  
  return isOnline ? 'В сети' : 'Не в сети';  
}
```



Мемоизация

```
const fib = n => {  
  if (n <= 1) return 1;  
  return fib(n - 1) + fib(n - 2);  
}
```

```
const fib = (n, memo) => {  
  memo = memo || {};  
  
  if (memo[n]) return memo[n];  
  
  if (n <= 1) return 1;  
  return memo[n] = fib(n-1, memo) + fib(n-2, memo);  
}
```

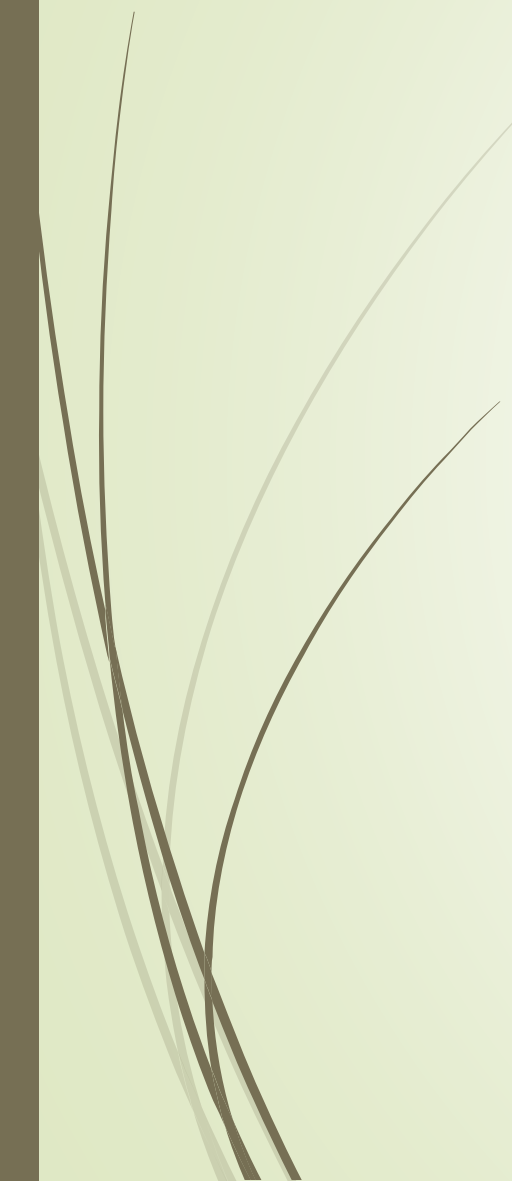



```
function App(){
  const [count, setCount] = useState(0);
  const [caths, setCaths] = useState(['IUK4']);

  const handleIncrement = () => setCount(count => count+1);
  const addCath = () => setCaths([...caths, 'New cath']);
  console.log("Parent render");

  return (
    <div className="App">
      <button onClick={() => handleIncrement()}>Increment</button>
      <h2>{count}</h2>
      <OtherComponent caths={caths} addCath={addCath} />
    </div>
  )
}

export default const OtherComponent = ({caths, addCath}) => {
  console.log("Child render")
  return (<>
    {caths.map((cath, index) => <p key={index}>{cath}</p>)}
    <button onClick={addCath}></button>
  </>)
}
```

```
function App(){
  const [count, setCount] = useState(0);
  const [caths, setCaths] = useState(['IUK4']);

  const handleIncrement = () => setCount(count => count+1);
  const addCath = useCallback(() => setCaths([...caths, 'New cath']), [caths])
  // const addCath = useMemo(() => () => setCaths([...caths, 'New cath']), [caths])
  console.log("Parent render");

  return (
    <div className="App">
      <button onClick={() => handleIncrement()}>Increment</button>
      <h2>{count}</h2>
      <OtherComponent caths={caths} addCath={addCath} />
    </div>
  )
}

export default const OtherComponent = ({caths, addCath}) => {
  console.log("Child render")
  return (<>
    {caths.map((cath, index) => <p key={index}>{cath}</p>)}
    <button onClick={addCath}></button>
  </>)
}
```