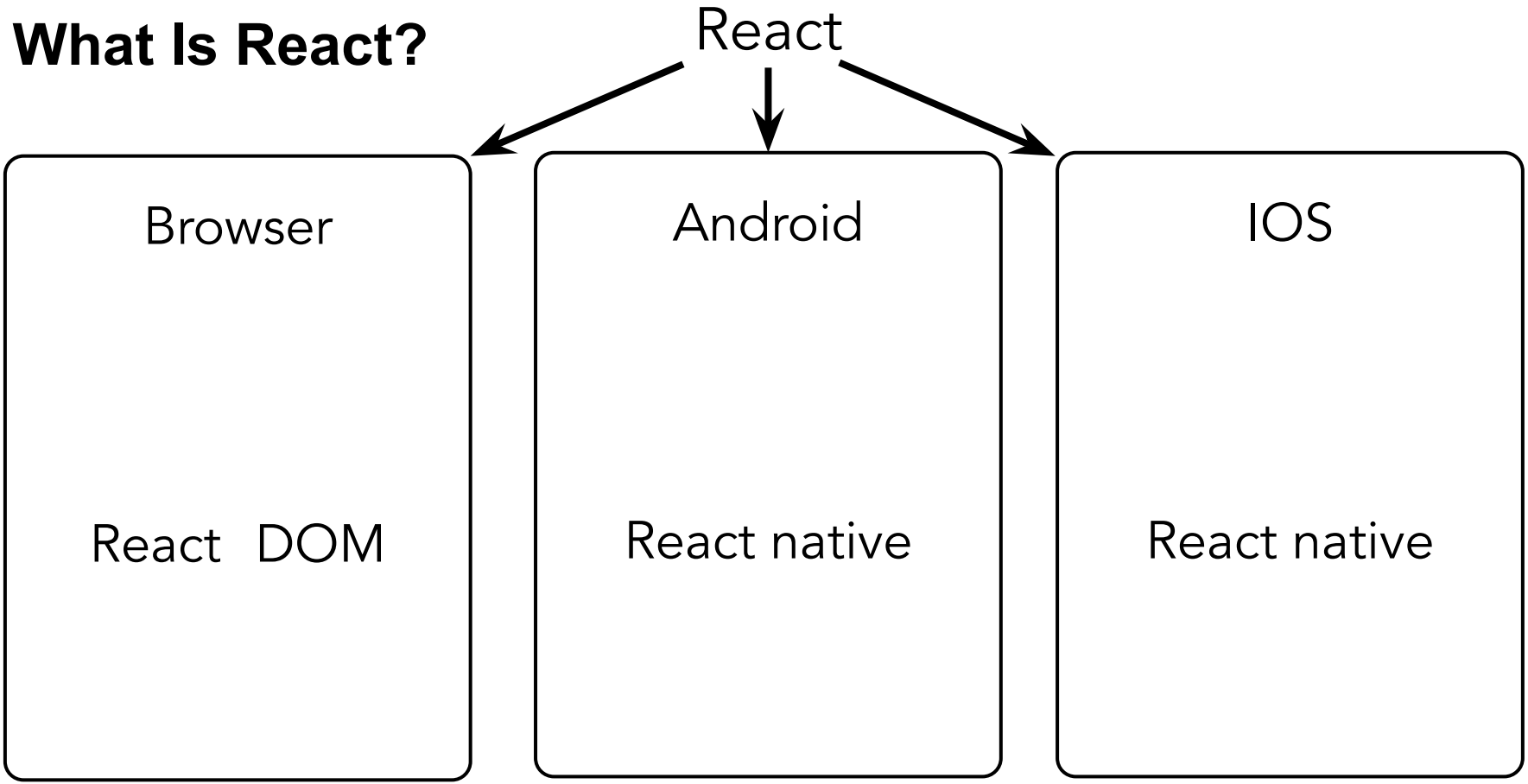
React Basics

Introduction

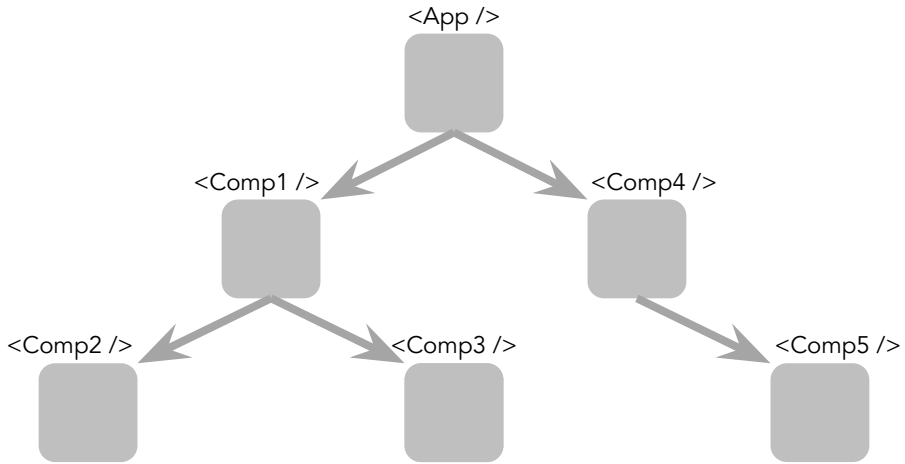
What Is React?



The Concept Of React

В React используется концепция компонентов. Все приложение представляет собой набор строительных блоков (компонентов), сложенных один в другой (образующих композицию).

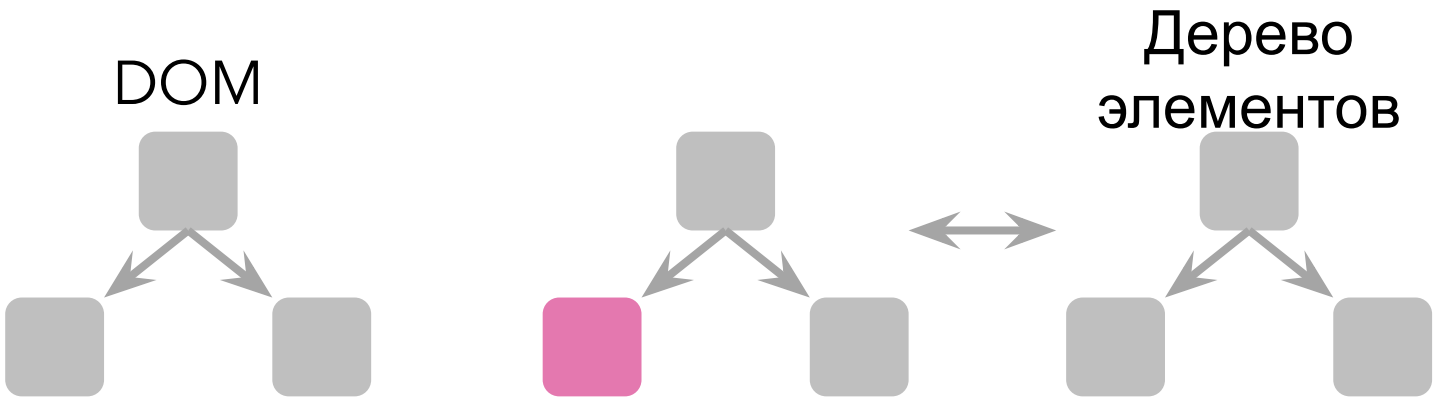
В приложении всегда присутствует корневой компонент (на рисунке - App).



Elements Tree

React использует под капотом виртуальное дерево элементов. Это более легкая и быстрая версия DOM дерева в контексте браузера.

В момент совершения какого-либо изменения React строит обновленное дерево и сравнивает с предыдущим. После нахождения разницы происходит рендеринг изменений. Каждое такое изменение имеет свой приоритет.



Setting Up A Project

Для создания проекта мы будем использовать create-react-app - это официально поддерживаемый способ создания одностраничных приложений React без предварительной настройки. Для использования необходимо иметь установленный node.js. Итак, для создания приложения необходимо использовать команду npx create-react-app <project name> --template typescript. Она генерирует директорию проекта со следующей структурой:

```
--node_modules/
--public/
--src/
----App.css
----index.css
----App.tsx
----App.test.tsx
----index.tsx
----logo.svg
--.gitignore
--package-lock.json
--package.json
--README.md
```

package.json:

```
{
  ...
  },
  "scripts": {
    "start": "react-scripts start",        - запуск сервера для разработки на  http://localhost:3000
    "build": "react-scripts build",        - сборка приложения
    "test": "react-scripts test",          - запуск тестов
    "eject": "react-scripts eject"         - разблокировка глубоких настроек (нельзя откатить)
  },
  ...
}
```

Meeting With Components

Первый компонент находится в /src/App.js:

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Для начала, немного сократим его:

```
import React from 'react'
function App() {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  )
}
export default App
```

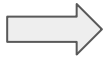
Компонент является обычной JS функцией. Компоненты такого вида называются функциональными.

Компонент возвращает HTML-подобный код. Он называется JSX.

JSX

JSX — расширение языка JavaScript, позволяющее писать HTML-подобный код, обладающий рядом преимуществ. Одно из них - интерполяция:

```
import React from 'react'
const title = 'React'
function App() {
  return (
    <div>
      <h1>Hello {title}</h1>
    </div>
  )
}
export default App
```



http://localhost:3000

Так как JSX является расширением JS, в нем запрещено использовать зарезервированные слова, такие как class и for. Вместо них используются соответственно className и htmlFor:

```
<h1 className="greetings">Hello {title}</h1>
```

Basics

Lists

JSX (TSX для TypeScript) позволяет использовать для создания пользовательских интерфейсов всю мощь языка JavaScript. Например, отрисовывать списки можно с помощью стандартного метода массивов - map. Для этого необходимо преобразовать каждый элемент массива в необходимый объект TSX:

```tsx
import React from 'react'
const list = [1, 2, 3, 4, 5]
function App() {
  return (
    <div>
      {list.map(el => (
        <h1 key={el}>{el}</h1>
      ))}
    </div>
  )
}
export default App
```

1

2

3

4

5

Чтобы React мог правильно отслеживать изменения, происходящие с элементами списка (например, изменение порядка элементов), необходимо передавать им атрибут key, уникальный для каждого элемента.

Another React Component

Всё приложение не может быть написано всего лишь на одном компоненте. Переиспользуемые блоки кода необходимо выносить в отдельные компоненты. Усложним элементы списка и реализуем компонент List:

```
interface IPost {
  title: string
  url: string
  author: string
  commentsCount: number
  points: number
  objectID: number
}
```

```
const List = () => {
  return list.map(function (el) {
    return (
      <div key={el.objectID}>
        <span>
          <a href={el.url}>{el.title}</a>
        </span>
        <span>{el.author}</span>
        <span>{el.commentsCount}</span>
        <span>{el.points}</span>
      </div>
    )
  })
}
```

Чтобы использовать компонент, необходимо его импортировать (если он находится в другом модуле) и указать как обычный html тег:

```
function App() {
  return (
    <div>
      <List />
    </div>
  )
}
```

React DOM

Изучив компоненты, перейдем к рассмотрению библиотеки ReactDOM. В файле src/index.tsx она импортируется вместе с React:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'

const root = ReactDOM.createRoot(document.getElementById('root') as HTMLElement)
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

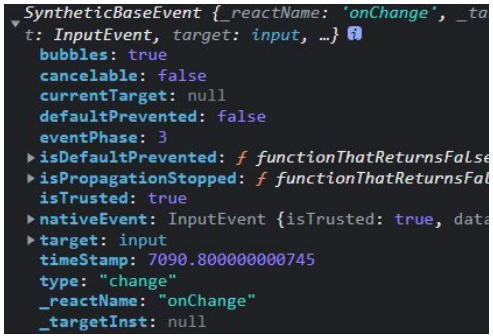
Функция createRoot() принимает на вход HTML-элемент, в котором будет отрендерено приложение, и создает его корневую точку.

Метод render() корневой точки принимает на вход само приложение и рендерит его в переданном в createRoot() HTML-элементе.

Handler Function

Почти любое приложение имеет интерактивные элементы, события которых необходимо обрабатывать. Рассмотрим работу с элементов input. Чтобы обрабатывать событие change, необходимо написать соответствующий обработчик и передать его в качестве атрибута onChange:

```
function App() {
  const logInputEvent: ChangeEventHandler<HTMLInputElement> = (evt) => console.log(evt)
  return (
    <div>
      <input type="text" onChange={logInputEvent} />
      <List />
    </div>
  )
}
```



Просмотрев консоль, можно заметить отличия объекта evt от обычных событий JavaScript. Такие события называются синтетическими. Они представляют собой кроссбраузерную обертку над стандартными событиями и обеспечивает одинаковую работу событий во всех браузерах.

Props

Сейчас компонент List берет список из глобальной области видимости. Это неправильно. Используя пропсы (props), мы можем передавать информацию от одного компонента к другому. Перенесем список в компонент App и передадим компоненту List:

```
function App() {
  const posts: IPost[] = [/*...*/]
  const logInputEvent = /*...*/
  return (
    <div>
      <input type="text" onChange={logInputEvent} />
      <List list={posts} />
    </div>
  )
}
```

Компонент - это функция. И в качестве параметра она принимает пропсы (props). Поэтому, чтобы использовать переданный список в дочернем компоненте, необходимо обратиться к параметру props:

```
const List = (props: { list: IPost[] }) => {
  return props.list.map(function (el) {
    return (
      <div key={el.objectID}>
        /*...*/
      </div>
    )
  })
}
```

Описать функциональный компонент можно с помощью стандартного интерфейса FC:

```
const List: FC<{ list: IPost[] }> = (props) => {
  /*...*/
}
```

State

Пропсы используются для передачи информации вниз по дереву компонентов. Состояния же (states) используются для того, чтобы сделать приложение интерактивным. Для работы с состояниями существует специальная функция useState(). Она принимает в качестве аргумента начальное состояние и возвращает массив из 2 элементов: состояния и сеттера для него:

```
function App() {
    const posts: IPost[] = [/*...*/]
    const [searchTerm, setSearchTerm] = React.useState('')
    const handleChange: /*Type*/ = (evt) => setSearchTerm(evt.target.value)
    return (
        <div>
            <input type="text" value={searchTerm} onChange={handleChange}/>
            <p>Searching for <strong>{searchTerm}</strong>.</p>
            <List list={posts} />
        </div>
    )
}
```

Когда пользователь меняет значение input, событие изменения попадает в обработчик, там новое значение устанавливается в качестве значения состояния, и мы видим обновленное значение в теге p.

Без установки состояния в обработчике значения поля ввода всегда будет ".

Такое поведение называется двусторонним связыванием: при изменении значения поля ввода меняется состояние, и при изменении состояния меняется значение поля ввода.

Callback Handlers

Элементы поиска разрослись, теперь их можно вынести в отдельный компонент Search:

```
function App() {
  const posts: IPost[] = [/*...*/]
  return (
    <div>
      <Search />
      <List list={posts} />
    </div>
  )
}
```

```
const Search: FC = () => {
  const [searchTerm, setSearchTerm] = React.useState('')
  const handleChange: /*Type*/ = (evt) =>
    setSearchTerm(evt.target.value)
  return (
    <div>
      <input type="text" onChange={handleChange} />
      <p>Searching for <strong>{searchTerm}</strong>.</p>
    </div>
  )
}
```

Сейчас Search просто отображает введенный текст и не делится им с другими компонентами. В таком виде он бесполезен.

Чтобы сделать компонент Search полезным, можно воспользоваться концепцией callback handlers:

```
type IHandleSearch = (val: string) => void
function App() {
  const posts: IPost[] = [/*...*/]
  const handleSearch: IHandleSearch = (term) => console.log(term)
  return (
    <div>
      <Search onSearch={handleSearch} />
      <List list={posts} />
    </div>
  )
}
```

```
const Search: FC<{ onSearch: IHandleSearch }> = () => {
  const [searchTerm, setSearchTerm] = React.useState('')
  const handleChange: /*Type*/ = (evt) =>
    setSearchTerm(evt.target.value)
    props.onSearch(evt.target.value)
  return /*...*/
}
```

Теперь из родительского компонента мы можем отслеживать изменения дочернего.

Lifting State

Предыдущий пример показывает, как отслеживать изменения в дочернем компоненте. Но хотелось бы иметь реактивное значение поискового запроса. Для этого перенесем состояние поискового запроса в место, где оно действительно нужно:

```
function App() {
  const posts: IPost[] = [/*...*/]
  const [searchTerm, setSearchTerm] = React.useState('')
  const handleSearch: IHandleSearch = (term) => setSearchTerm(term)
  const searchPosts = posts.filter(el => el.title.includes(searchTerm))
  return (
    <div>
      <Search onSearch={handleSearch} />
      <List list={searchPosts} />
    </div>
  )
}
```

Теперь у нас есть список, отображающийся компонентом List, который зависит от поискового запроса, динамически изменяемого компонентом Search. Каждый компонент решает свою задачу.

Осталась одна небольшая проблема. Поиск зависит от регистра. Чтобы решить этот вопрос, достаточно использовать стандартный строковый метод toLowerCase():

```
const searchPosts = posts.filter(el => {
    return el.title.toLowerCase().includes(searchTerm.toLowerCase())
})
```

Controlled Components

Если задать состоянию поискового запроса начальное значение, мы увидим отсортированный список, но значением поля ввода запроса будет пустая строка. Чтобы решить эту проблему, необходимо сделать компонент Search управляемым (controlled component). Для этого передадим ему в качестве пропса состояние searchTerm:

```
interface ISearchProps {
  term: string,
  onSearch: IHandleSearch
}
```

```
function App() {
  /*...*/
  return (
    <div>
      <Search term={searchTerm} onSearch={handleSearch} />
      <List list={searchPosts} />
    </div>
  )
}
```

```
const Search: FC<ISearchProps> = props => {
  /*...*/
  return (
    <div>
      <input value={props.term} onChange={handleChange} />
    </div>
  )
}
```

Теперь компонент Search создается с правильным начальным значением. Мы реализовали двустороннее связывание - то, что раньше делали с элементом input.

Side-Effects

Реализуем сохранение последнего поискового запроса localStorage:

```
const [searchTerm, setSearchTerm] = React.useState(
    localStorage.getItem('search') || 'React'
  )
  const handleSearch: IHandleSearch = (term) => {
    setSearchTerm(term)
    localStorage.setItem('search', term)
  }
```

При создании компонента начальное состояние задается значением из локального хранилище либо устанавливается в React. При каждом изменении состояния значение также сохраняется в localStorage.

Использование хранилища можно рассматривать как побочный эффект (side-effect), поскольку мы работаем за пределами домена React, используя API браузера.

В данном решении есть существенный недостаток: функция обработчика должна выполнять только свою работу, сейчас же на нее возложена работа с хранилищем. Также, если мы захотим использовать setSearchTerm где-то еще, значение в хранилище не обновится.

Effects

Для решения проблемы можно использовать React-функцию useEffect.

```
const [searchTerm, setSearchTerm] = React.useState(
  localStorage.getItem('search') || 'React'
)
React.useEffect(() => {localStorage.setItem('search', searchTerm)}, [searchTerm])
const handleSearch: IHandleSearch = (term) => {
  setSearchTerm(term)
}
```

Функция ожидает на вход функцию-эффект, выполняющуюся при каждом обновлении зависимостей, вторым же параметром она принимает сами зависимости в виде массива. Если массив не передан, эффект вызывается при каждом обновлении компонента. Если массив пустой, эффект вызывается только при создании компонента. Если эффект возвращает функцию, она вызовется при уничтожении компонента.

Hooks

Хуки — это технология, которая перехватывает вызовы функций. Мы уже работали с некоторыми из них: useState() и useEffect(). Теперь, используя эти хуки, напишем собственный:

```
const useSemiPersistentState = (key: string, initialState = '') => {
  const [value, setValue] = React.useState(localStorage.getItem(key) || initialState)
  React.useEffect(() => {
    localStorage.setItem(key, value)
  }, [value, key])
  return [value, setValue] as const
}
```

Имена хуков должны начинаться с use.

```
function App() {
  /*...*/
  const [searchTerm, setSearchTerm] = useSemiPersistentState('search')
  /*...*/
}
```

Пользовательские хуки позволяют инкапсулировать нетривиальные детали реализации и переиспользовать их.

Fragments

Одна из особенностей JSX/TSX - в компоненте должен присутствовать оберточный элемент. В данном примере - div:

```
const Comp = props => {
  /*...*/
  return (
    <div>
      <div className="1"></div>
      <div className="2"></div>
      <div className="3"></div>
    </div>
  )
}
```

Если необходимо иметь несколько элементов верхнего уровня, необходимо обернуть их в массив и каждому из них присвоить уникальный ключ:

```
const Comp = props => {
  /*...*/
  return [
    <div className="1" key="1"></div>
    <div className="2" key="2"></div>
    <div className="3" key="3"></div>
  ]
}
```

Альтернативным способом является использование фрагментов (fragments). По синтаксису они напоминают тег без имени:

```
const Comp = props => {
  /*...*/
  return <>
    <div className="1"></div>
    <div className="2"></div>
    <div className="3"></div>
  </>
}
```

Children

Элементы, помещаемые внутрь компонентов, передаются им как пропс с именем children. Используем это и добавим label в компонент поиска:

```tsx
interface ISearchProps {
  term: string
  onSearch: IHandleSearch
  id: string
  children: string
}
```

```tsx
const Search: FC<ISearchProps> = props => {
  /*...*/
  return (
    <div>
      <label htmlFor={props.id}>{props.children}</label>
      <input id={props.id} value={props.term} onChange={handleChange} />
    </div>
  )
}
```

```tsx
function App() {
  /*...*/
  return (
    <div>
      <Search
        id="search"
        term={searchTerm}
        onSearch={handleSearch}
      >Search</Search>
      <List list={searchPosts} />
    </div>
  )
}
```

В текущем виде компонент получился универсальным и переиспользуемым. В дальнейшем его можно будет использовать не только как поле поиска, поэтому его следует переименовать в InputWithLabel.

useRef

useRef() - хук, возвращающий изменяемый ref-объект. Его свойство current инициализируется переданным аргументом. Обычно useRef используется для императивного взаимодействия с элементами. Чтобы связать элемент с ref-объектом, используется атрибут ref:

```
const Comp: FC = () => {
  const ref = React.useRef<HTMLInputElement>(null)
  return <>
    <input ref={ref} type="text" />
  </>
}
```

В свойстве current переменной ref хранится ссылка на элемент input, как если бы использовался querySelector.

Если требуется передать ref дочернему компоненту, его необходимо обернуть в forwardRef, чтобы React знал, с каким элементом связывать ссылку:

```
const ChildComp = React.forwardRef<HTMLInputElement, IChildCompProps>((props, ref) => <>
  <input ref={ref} type="text" />
</>)
```

Inline Handler

До этого момента список являлся лишь обычной переменной. Работы с реальными данными нет. Чтобы получить контроль над списком, необходимо сделать его stateful, используя useState():

```
const initPosts: IPost[] = [/*...*/]

function App() {
  const [posts, setPosts] = useState(initPosts)
  /*...*/
}
```

Теперь можно реализовать логику изменения списка. Например, удаление элементов:

```
function App() {
  const [posts, setPosts] = useState(initPosts)
  const handleRemovePost = (post: IPost) => {
    setPosts(posts.filter((p) => p.objectID !== post.objectID))
  }
  /*...*/
  return (
      /*...*/
      <List list={searchPosts} onRemoveItem={handleRemovePost} />
  )
}
```

```
interface IListProps {
  list: IPost[]
  onRemoveItem: (p: IPost) => void
}

const List: FC<IListProps> = props => <>{
  props.list.map(function (el) {
    return (
      <div key={el.objectID}>
        /*...*/
        <button onClick={() => props.onRemoveItem(el)}>delete</button>
      </div>
    )
  })
}</>
```

Asynchronous Data

В реальных приложениях обычно данные приходится загружать из внешних источников. Представим, что список постов возвращается извне:

```
const [posts, setPosts] = useState<IPost[]>([])
const getAsyncPosts = () => new Promise<{ data: { posts: IPost[] } }>((resolve) => {
  setTimeout(() => resolve({data: {posts: initPosts}}), 2000)
})
```

Через 2 секунды результатом промиса будет список постов

Для работы с асинхронной загрузкой данных можно использовать хук useEffect:

```
  useEffect(() => {
    getAsyncPosts().then(result => setPosts(result.data.posts))
  }, [])
```

Так как массив зависимостей пуст, эффект выполнится только при создании компонента.

Conditional Rendering

В реальном приложении довольно часто приходится обрабатывать всевозможные состояния и в зависимости от них менять контент. Рассмотрим работу с состоянием загрузки:

```
const [posts, setPosts] = useState<IPost[]>([])
const [isLoading, setIsLoading] = useState(false)

useEffect(() => {
  setIsLoading(true)
  getAsyncPosts().then(result => {
    setPosts(result.data.posts)
    setIsLoading(false)
  })
}, [])
```

С помощью хука useState() создается состояние загрузки. В момент создания компонента оно устанавливается в true, и когда загрузка завершена - в false.

Теперь необходимо показать пользователю, когда происходит загрузка. React позволяет использовать всю гибкость JS, поэтому можно воспользоваться, например, тернарным оператором:

```
function App() {
  /*...*/
  return (
    <div>
      /*...*/
      {
        isLoading
          ? <p>Loading...</p>
          : <List list={searchPosts} onRemoveItem={handleRemovePost} />
      }
    </div>
  )
}
```

Advanced State: useReducer

useReducer() - хук, являющийся альтернативой useState() и позволяющий реализовать более сложную логику управления состояниями. Первое, с чего необходимо начать - функция reducer. Она реализуется вне компонента. На вход функция всегда получает состояние и действие, и зависимости от них должна возвращать новое состояние:

```
type IPostsReducerAction = {
  type: 'SET',
  payload: IPost[]
} | {
  type: 'DELETE',
  payload: IPost
}

const postsReducer: Reducer<IPost[], IPostsReducerAction> = (state, action) => {
  switch (action.type) {
    case 'SET':
      return action.payload
    case 'DELETE':
      return state.filter(p => p.objectID !== action.payload.objectID)
  }
}
```

Advanced State: useReducer

Воспользоваться редьюсером можно, передав его в useReducer. Вторым параметром этот хук принимает начальное состояние. Также задать начальное состояние можно, передав вторым параметром аргументы для функции-инициализатора, а третьим - сам инициализатор.

```
function App() {
  /*...*/
  const [posts, dispatchPosts] = useReducer(postsReducer, [])

  useEffect(() => {
    setIsLoading(true)
    getAsyncPosts().then(result => {
      dispatchPosts({type: 'SET', payload: result.data.posts})
      setIsLoading(false)
    })
  }, [])

  const handleRemovePost = (post: IPost) => {
    dispatchPosts({type: 'DELETE', payload: post})
  }
  /*...*/
}
```

Impossible States

Невозможное состояние - состояние, которое может возникнуть при ошибке в работе с асинхронными данными. Уменьшить вероятность их появления можно, объединив логику работы состояний с данными:

```
type IPostsReducerAction =
  {
    type: 'FETCH_INIT',
  } | {
  type: 'FETCH_SUCCESS',
  payload: IPost[]
} | {
  type: 'FETCH_FAILURE',
} | {
  type: 'DELETE',
  payload: IPost
}

interface IPostsState {
  data: IPost[],
  isLoading: boolean
  isError: boolean
}
```

```
const postsReducer: Reducer<IPostsState, IPostsReducerAction> = (state, action) => {
  switch (action.type) {
    case 'FETCH_INIT':
      return {data: [], isLoading: true, isError: false}
    case 'FETCH_SUCCESS':
      return {data: action.payload, isLoading: false, isError: false}
    case 'FETCH_FAILURE':
      return {...state, isLoading: false, isError: true}
    case 'DELETE':
      return {...state, data: state.data.filter(p => p.objectID !== action.payload.objectID)}
  }
}
```

Impossible States

Объединенную логику использовать намного проще, а вероятность получить невозможное состояние снижается:

```
function App() {
  /*...*/
  const [posts, dispatchPosts] = useReducer(postsReducer, {data: [], isLoading: false, isError: false})
  useEffect(() => {
    dispatchPosts({type: 'FETCH_INIT'})
    getAsyncPosts().then(result => {
      dispatchPosts({type: 'FETCH_SUCCESS', payload: result.data.posts})
    })
  }, [])

  const handleRemovePost = (post: IPost) => {
    dispatchPosts({type: 'DELETE', payload: post})
  }
  /*...*/
  return (
    <div>
      /*...*/
      {posts.isError && <p>Something went wrong ...</p>}
      {
        posts.isLoading
          ? <p>Loading...</p>
          : <List list={searchPosts} onRemoveItem={handleRemovePost} />
      }
    </div>
  )
}
```

Data Fetching

До текущего момента приложение работало с моками - ненастоящими данными. Рассмотрим работу с API на примере Hacker News API:

```
interface IApiPost {
  title: string | null
  url: string | null
  author: string
  points: number
  objectID: string
}

interface IApiResponse {
  hits: IApiPost[]
}
```

```
const API_URL = 'https://hn.algolia.com/api/v1/search'

function App() {
  /*...*/
  useEffect(() => {
    dispatchPosts({type: 'FETCH_INIT'})
    fetch(API_URL)
    .then(res => res.json())
    .then((result: IApiResponse) => {
      dispatchPosts({type: 'FETCH_SUCCESS', payload: result.hits})
    })
    .catch(() => dispatchPosts({type: 'FETCH_FAILURE'}))
  }, [])
  /*...*/
}
```

В компонент списка необходимо внести соответствующие изменения для корректного отображения постов с url или title в значении null.

Data Re-Fetching

Сейчас загрузка данных происходит единоразово. В реальном приложении часто приходится повторно запрашивать данные. Например, при поиске:

```
function App() {
  /*...*/
  useEffect(() => {
    dispatchPosts({type: 'FETCH_INIT'})
    fetch(`${API_URL}?query=${searchTerm}`)
    .then(res => res.json())
    .then((result: IApiResponse) => {
      dispatchPosts({type: 'FETCH_SUCCESS', payload: result.hits})
    })
    .catch(() => dispatchPosts({type: 'FETCH_FAILURE'}))
  }, [searchTerm])
  /*...*/
  return (
    <div>
      /*...*/
      {
        posts.isLoading
          ? <p>Loading...</p>
          : <List list={posts.data} onRemoveItem={handleRemovePost} />
      }
    </div>
  )
}
```

При таком подходе задачу поиска перекладывается на сервер, и логика, связанная с searchPosts больше не нужна.

Memoization: useCallback, useMemo

useCallback - хук, принимающий на вход функцию и массив зависимостей. Он возвращает мемоизированную версию принятой функции, которая изменяется только при изменении зависимости:

```
const handlePosts = useCallback(() => {
  dispatchPosts({type: 'FETCH_INIT'})
  fetch(`${API_URL}?query=${searchTerm}`)
  .then(res => res.json())
  .then((result: IApiResponse) => {
    dispatchPosts({type: 'FETCH_SUCCESS', payload: result.hits})
  })
  .catch(() => dispatchPosts({type: 'FETCH_FAILURE'}))
}, [searchTerm])

useEffect(handlePosts, [handlePosts])
```

Такой подход позволяет не создавать функцию при каждом обновлении компонента, а также избежать ситуаций с бесконечным циклом useEffect, когда эффект при создании обновляет состояние, изменение заставляет обновиться компонент и пересоздать эффект, который изменяет состояние…

React также предоставляет похожий хук useMemo(). Он, в отличие от useCallback(), возвращает результат выполнения полученного коллбэка и обновляет каждый раз при изменении зависимостей, что позволяет избежать лишних дорогостоящих вычислений.

Class Components

За время существования React компоненты претерпели множество изменений. Изначально был 1 вид компонентов - классовые:

```
class ClassInputWithLabel extends React.Component<InputWithLabelProps> {
  render() {
    return <>
      <div>
        <label htmlFor={this.props.id}>{this.props.children}</label>
        <input
          id={this.props.id}
          value={this.props.term}
          onChange={(evt) => this.props.onSearch(evt.target.value)}
        />
      </div>
    </>
  }
}
```

Стандартный классовый компонент - класс с обязательным методом render(), возвращающим JSX. Для доступа к основным свойствам компонента, класс наследует React.Component. Доступ к пропсам осуществляется через this.

Если компонент не должен был обладать эффектами и состояниями, можно было использовать функциональный подход.

Class Components: State

До появления хуков классовые компоненты обладали бóльшим функционалом - в них можно было использовать состояния:

```
interface IAppState {searchTerm: string}

class ClassApp extends React.Component<any, IAppState> {
  constructor(props: any) {
    super(props)
    this.state = {
      searchTerm: 'React',
    }
  }

  render() {
    const {searchTerm} = this.state
    return (
      <ClassInputWithLabel
        id="search"
        term={searchTerm}
        onSearch={(val) => this.setState({searchTerm: val})}
      >Search</ClassInputWithLabel>
    )
  }
}
```

Для работы с состояниями React.Class предоставляет специальный API - this.state и this.setState.

С добавлением хуков, функциональные компоненты перестали уступать классовым. Сообщество перешло на функциональные компоненты за счет простоты синтаксиса, а классовые стали считаться устаревшими.

CSS

Чтобы начать работать со стилями, достаточно создать простой css файл и импортировать его:

```
import './App.css'

function App() {
  /*...*/
  return (
    <div className="app">
      <InputWithLabel
        id="search"
        term={searchTerm}
        onSearch={handleSearch}
      >Search</InputWithLabel>
      {posts.isError && <p>Something went wrong ...</p>}
      {
        posts.isLoading
          ? <p>Loading...</p>
          : <List list={posts.data} onRemoveItem={handleRemovePost} />
      }
    </div>
  )
}
```

```
.app {
  margin: 100px auto;
  max-width: 1200px;
  padding: 30px;
  border: 5px dashed grey;
  border-radius: 15px;
}
```

CSS Modules

CSS-модули - более продвинутый способ работы со стилями. Проблема, которую они решают - конфликт имен. CSS-модуль представляет собой обычный CSS-файл с именем в формате <name>.module.css. При таком подходе мы импортируем не просто css файл, а его классы:

```
import classes from './App.module.css'

function App() {
  /*...*/
  return (
    <div className={classes.app}>
      <InputWithLabel
        id="search"
        term={searchTerm}
        onSearch={handleSearch}
      >Search</InputWithLabel>
      {posts.isError && <p>Something went wrong ...</p>}
      {
        posts.isLoading
          ? <p>Loading...</p>
          : <List list={posts.data} onRemoveItem={handleRemovePost} />
      }
    </div>
  )
}
```

Под капотом для каждого класса генерируется уникальное имя:

```
<div id="root">
▸ <div class="App_app__Hg1VR">
</div>
```

Styled Components

Существует еще один способ работы со стилями - Styled Components. Они не поддерживаются из коробки, поэтому придется установить зависимость. Для работы на TypeScript также понадобится установка типов:

```
npm install styled-components
```

```
npm i --save-dev @types/styled-components
```

```
import styled from 'styled-components'

const StyledApp = styled.div`
  margin: 100px auto;
  max-width: 800px;
  padding: 30px;
  border: 5px dashed grey;
  border-radius: 15px;
`
```

```
function App() {
  /*...*/
  return (
    <StyledApp>
    /*...*/
    </StyledApp>
  )
}
```

```
<div id="root">
▸ <div class="sc-bczRLJ cFfJQc">
</div>
```

Принцип работы styled components основан на технологии tagged templates. Результатом вызова функций является компонент, содержащий в себе указанные стили.