ДИСЦИПЛИНА	Фронтенд и бэкенд разработка
ИНСТИТУТ	Институт перспективных технологий и индустриального программирования
КАФЕДРА	Кафедра индустриального программирования
ВИД УЧЕБНОГО МАТЕРИАЛА	Методические указания к практическим занятиям
ПРЕПОДАВАТЕЛЬ	Астафьев Рустам Уралович
СЕМЕСТР	4 семестр, 2023-2024 учебный год

Практическое занятие №5

To-do list на ReactJS

Обратите внимание, что выполнение данного задания и является базовым примером по теме.

Файлы к данной инструкции располагаются в рабочей области дисциплины в СДО.

Практическое занятие 5 посвящено созданию веб-приложения «ToDo list» («Список задач»). ToDo list - это список дел, которые вам нужно выполнить или того, что вы хотите сделать. На примере этого приложения вы освоите подходы к созданию собственных React-компонентов.

1. Для начала работы с React на вашем компьютере должны быть установлены Node.js и create-react-app.
2. Создадим базовое React приложение. Запустим терминал командной строки («Пуск»-> «Служебные Windows» -> «Командная строка»). Введите в терминал команду

   create-react-app todo_list //нажимаем «Enter»

   Для запуска приложения последовательно выполните следующие команды:

   cd todo_list //нажимаем «Enter»

   npm start //нажимаем «Enter»

Сама папка с проектом находится в папке пользователя, (обычно, C:\Users\Имя_пользователя) Имя_пользователя у каждого своё. Вместо Users папка может называться «Пользователи».

todo_list это папка, которая содержит созданное нами приложение. Откроем эту папку в Visual Studio Code и приступим к созданию приложения.

Пошаговое создание веб-приложения

Приступим к созданию приложения «ToDo list» на React — приложения, которое позволяет пользователям добавлять, редактировать и удалять задачи, над которыми они хотят работать, а также отмечать задачи как выполненные, не удаляя их.

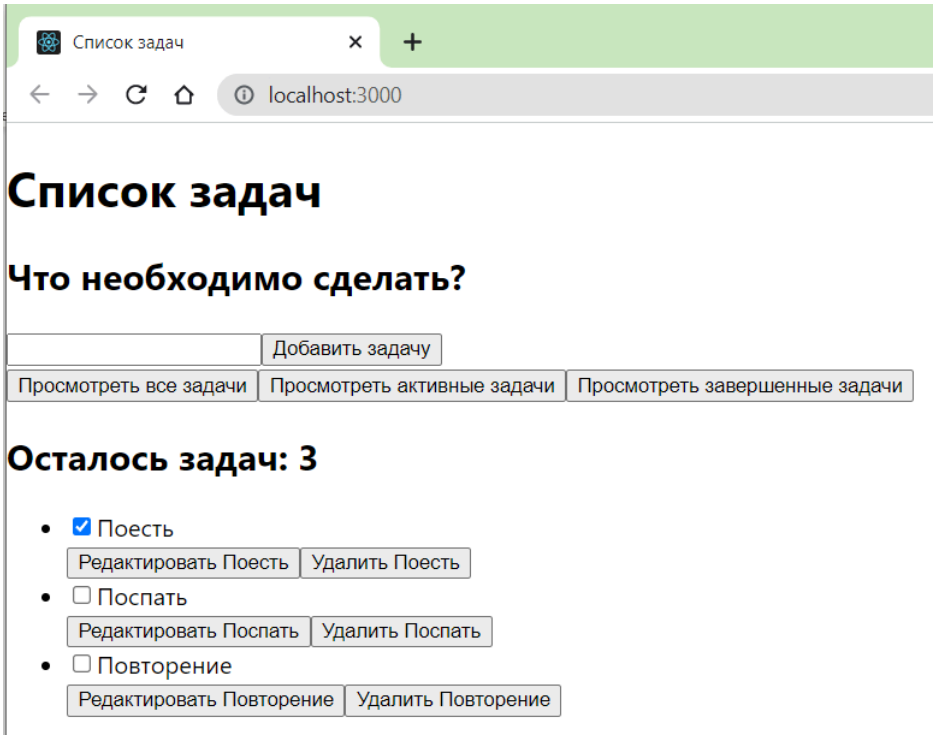
Создаваемое нами веб-приложение будет обладать следующим функционалом:

- прочитать списка задач.
- добавить задачи в список.
- пометить любую задачу как выполненную.
- удалить любую задачу.
- редактировать любую задачу.
- просмотреть определенное подмножество задач: все задачи, только активная задача или только выполненные задачи.

Откройте в Visual Studio Code файл App.js. Удалим 1 и 2 строки (import) из этого файла.

Функцию App() в файле App.js заменим новой функцией App(). (Код новой функции доступен в папке Файлы, скачанной с СДО. Код находится в файле app_func.js). По сути, это обычная HTML-форма с кнопками, галочками и другими элементами.

Теперь откройте public/index.html и измените <title>текст элемента на «Список задач». Приложение в браузере выглядит примерно так:



Пока оно не работает, кнопки не функционируют.

Рассмотрим содержимое функции App():

- У нас есть форма <form> и элемент <input type="text">для написания новой задачи, а также кнопка для отправки формы (“Добавить задачу”).
- У нас есть несколько кнопок, которые будут использоваться для фильтрации наших задач (Просмотреть все задачи, Просмотреть активные задачи, Просмотреть завершенные задачи).
- У нас есть заголовок, который говорит нам, сколько задач осталось.
- У нас есть 3 задачи, расположенные в неупорядоченном списке. Каждая задача представляет собой элемент списка ( ) и имеет кнопки для ее редактирования и удаления, а также флажок, чтобы отметить ее как выполненную.

То есть с помощью верхнего элемента формы мы сможем ставить задачи; кнопки ниже позволят нам фильтровать их; заголовок и список необходимы для просмотра задач. Пользовательский интерфейс для редактирования задачи пока отсутствует, будет сделан позже.

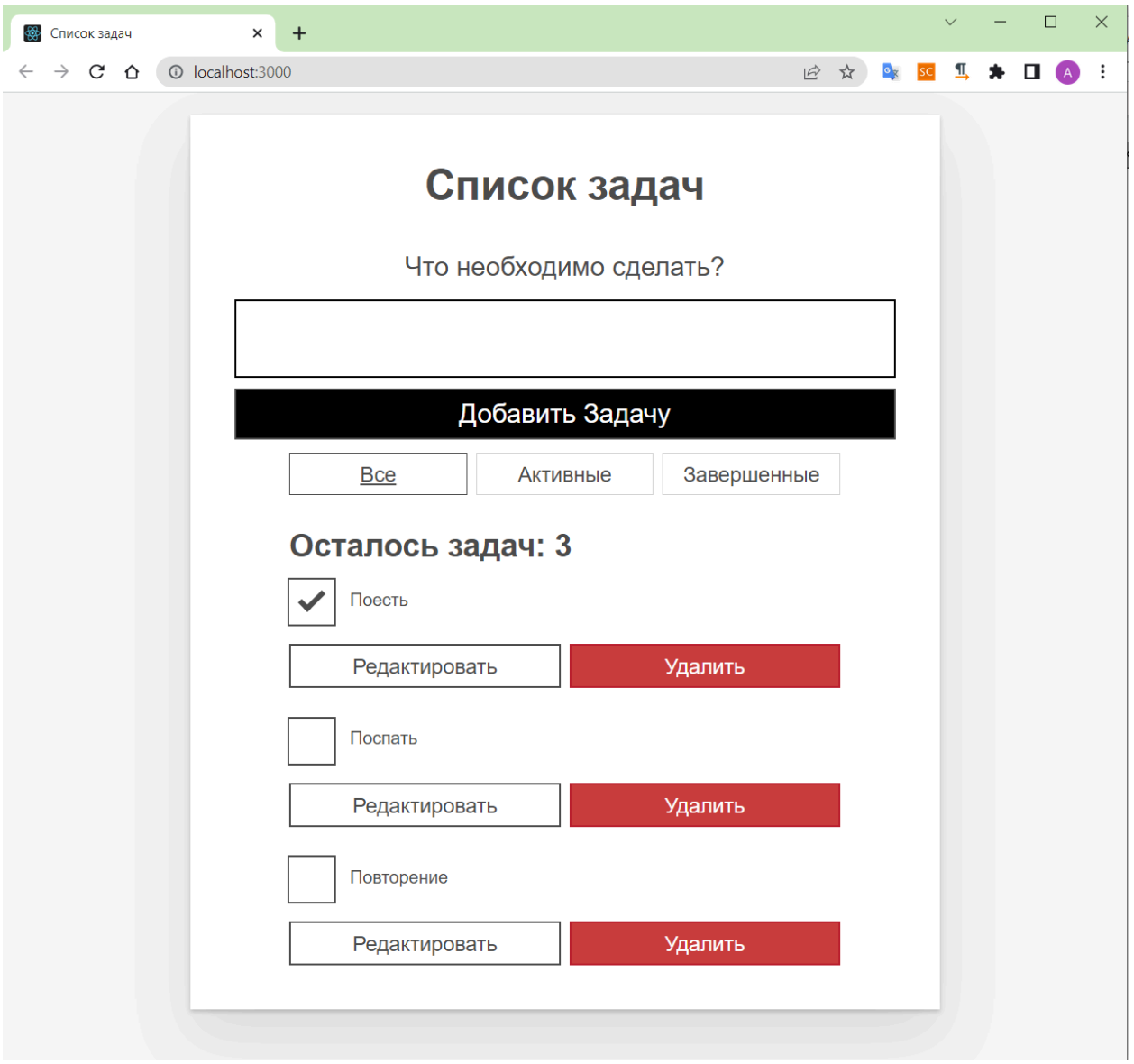
Привычные элементы кода (например, <input /> содержат новые атрибуты.

Например, атрибут defaultChecked в <input/ > теге сообщает React, что сначала нужно установить этот флажок. Если бы мы использовали checked, как в обычном HTML, React регистрировал бы в консоли браузера некоторые предупреждения, касающиеся обработки событий флажка, которых мы хотим избежать.

Атрибут htmlFor соответствует for атрибуту, используемому в HTML. Мы не можем использовать for в качестве атрибута в JSX, потому что for - это зарезервированное слово в JavaScript.

Добавим стили. Копируем содержимое файла index_style.css из папки Файлы, скачанной с СДО, в конец файла index.css.

Наше веб-приложение выглядит теперь гораздо лучше.



Однако пока кнопки не работают.

Создание компонента из обычных элементов пользовательского интерфейса позволяет изменять свой код в одном месте и видеть эти изменения везде, где используется этот компонент.

Создадим собственный компонент <Todo />. Для этого в папке src проекта создадим папку с названием components, а в ней пустой файл Todo.js. Этот файл в настоящее время пуст. Откройте его в Visual Studio Code и введите в него первую строку:

```
import React from "react";
```

Поскольку мы собираемся создать компонент с именем Todo, мы также можем начать добавлять код для него в Todo.js, как показано ниже:

```
export default function Todo() {
  return (
    // …
  );
}
```

Наполним содержимое команды return(). Перейдем к файлу App.js, копируем первый <li> из ненумерованного списка и вставим его в Todo.js так, чтобы функция Todo() выглядела следующим образом:

```
import React from "react";

export default function Todo() {
    return (
        <li className="todo stack-small">
        <div className="c-cb">
          <input id="todo-0" type="checkbox" defaultChecked={true} />
          <label className="todo-label" htmlFor="todo-0">
            Поесть
          </label>
        </div>
        <div className="btn-group">
          <button type="button" className="btn">
            Редактировать <span className="visually-hidden">Поесть</span>
          </button>
          <button type="button" className="btn btn__danger">
            Удалить <span className="visually-hidden">Поесть</span>
          </button>
        </div>
      </li>
    );
  }
```

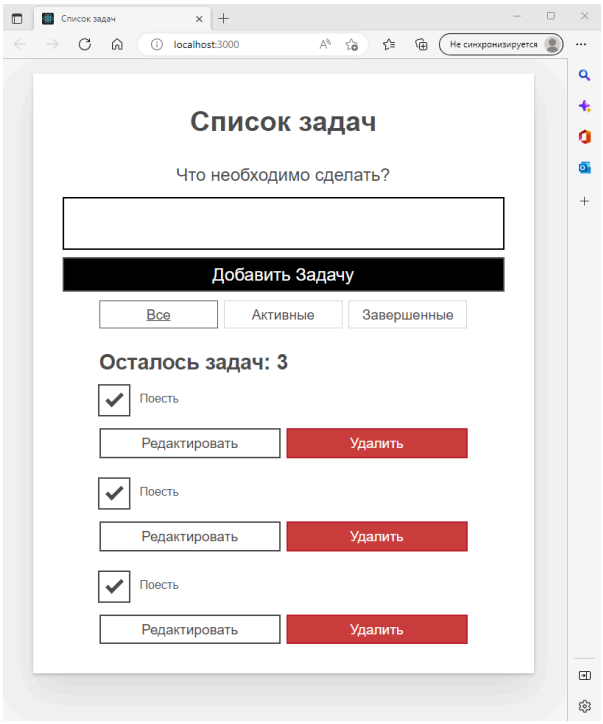
Наш Todo компонент завершен; теперь мы можем использовать его. В App.js, добавьте следующую строку вверху файла для импорта Todo:

```
import Todo from "./components/Todo";
```

Импортировав этот компонент, вы можете заменить все <li> элементы App.js вызовами <Todo /> компонентов. То есть список <ul> выглядит теперь так:

```
<ul
  role="list"
  className="todo-list stack-large stack-exception"
  aria-labelledby="list-heading"
>
  <Todo />
  <Todo />
  <Todo />
</ul>
```

В браузере мы видим следующее:



Сейчас в списке дел содержится только «Поесть» - трижды. Но в списке дел должны быть и другие дела. Рассмотрим, как разные вызовы компонентов заставить отображать уникальный контент.

Чтобы отслеживать имена задач, которые мы хотим выполнить, мы должны убедиться, что каждый <Todo />компонент отображает уникальное имя. В App.js для каждого элемента <Todo /> пропишем каждому <Todo /> атрибут name, значение которого будет соответствовать уникальной задаче:

```
<ul
  role="list"
  className="todo-list stack-large stack-exception"
  aria-labelledby="list-heading">
<Todo name="Поесть"/>
<Todo name="Поспать"/>
<Todo name="Повторить"/>
</ul>
```

Далее в файле Todo.js в строку, где мы объявляем функцию, т.е. в

```
export default function Todo() {
```

в круглые скобки добавим слово props:

```
export default function Todo(props) {
```

Через параметр props мы можем передавать внутрь компонента уникальный контент. В нашем случае таким контентом является значением атрибута name в <Todo /> из файла App.js.

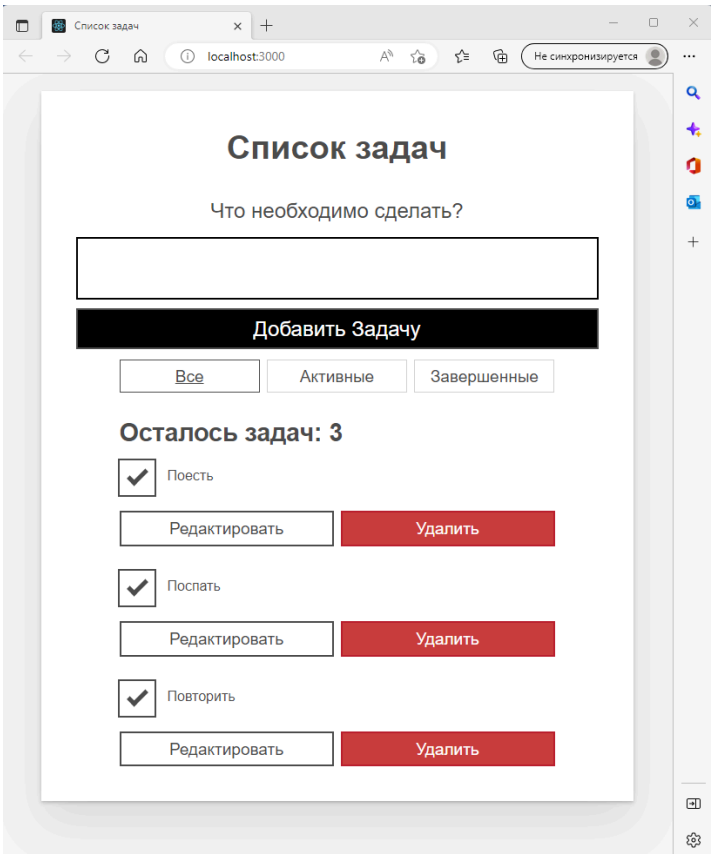
В коде файла Todo.js мы можем добраться до значения атрибута name путем применения команды {props.name}.

Т.е. слово Поесть в коде нужно заменить на {props.name} в 3 местах. Todo.js выглядит теперь так:

src > components > JS Todo.js > Todo

```
1   import React from "react";
2
3   export default function Todo(props) {
4       return (
5           <li className="todo stack-small">
6           <div className="c-cb">
7             <input id="todo-0" type="checkbox" defaultChecked={true} />
8             <label className="todo-label" htmlFor="todo-0">
9               {props.name}
10            </label>
11          </div>
12          <div className="btn-group">
13            <button type="button" className="btn">
14              Редактировать <span className="visually-hidden">{props.name}</span>
15            </button>
16            <button type="button" className="btn btn__danger">
17              Удалить <span className="visually-hidden">{props.name}</span>
18            </button>
19          </div>
20        </li>
21      );
22  }
```

Браузер показывает 3 уникальные задачи. Однако они все отмечены галочками, хотя ранее галочкой была отмечена только задача «Поесть».



Добавим в вызовы <Todo /> из App.js еще один атрибут completed, который будет принимать значения {true} или {false} в зависимости от того, поставлена галочка или нет:

```
<Todo name="Поесть" completed={true}/>
<Todo name="Поспать" completed={false}/>
<Todo name="Повторить" completed={false}/>
```

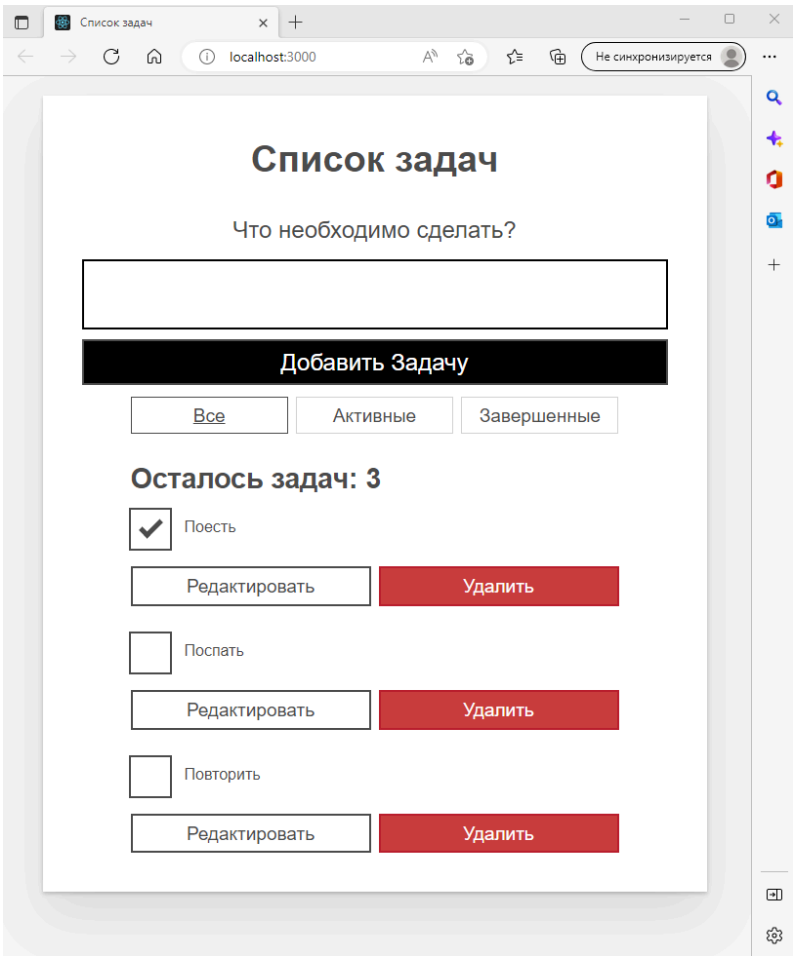
Далее в очередной раз отредактируем Todo.js, где строку

```
<input id="todo-0" type="checkbox" defaultChecked={true} />
```

заменим на

```
<input id="todo-0" type="checkbox" defaultChecked={props.completed} />
```

Обновляем браузер:



Далее. В последней строке, с которой мы работали прописано: id="todo-0". Это означает, что каждый созданный нами Todo-элемент имеет одинаковый id. Однако нам известно, что id у всех элементов должен быть разным. Дадим каждому экземпляру <Todo /> компонента в App.js идентификатор в формате todo-i, где i каждый раз увеличивается на единицу:

```
<ul
  role="list"
  className="todo-list stack-large stack-exception"
  aria-labelledby="list-heading">
 <Todo name="Поесть" completed={true} id="todo-0"/>
 <Todo name="Поспать" completed={false} id="todo-1"/>
 <Todo name="Повторить" completed={false} id="todo-2"/>
</ul>
```

Теперь вернемся в Todo.js и вновь поменяем строку

```
<input id="todo-0" type="checkbox" defaultChecked={props.completed} />
```

на строку:

```
<input id={props.id} type="checkbox" defaultChecked={props.completed} />
```

а также значение атрибута htmlFor у <label> в следующей строке:

```
<label className="todo-label" htmlFor={props.id}>
```

Подготовка к работе с произвольным количеством задач.

Сейчас наш список задач состоит из ровно 3 задач, для каждой из которой мы создали свой элемент <Todo />. Для хранения произвольного количества задач данные для формирования элементов <Todo /> (т.е. значения атрибутов name, completed и id) следует хранить в специальном массиве, который позволит добавлять в него новые записи, изменять существующие и т.д. Массив для имеющихся 3 элементов выглядит так:

```
const DATA = [
  { id: "todo-0", name: "Поесть", completed: true },
  { id: "todo-1", name: "Поспать", completed: false },
  { id: "todo-2", name: "Повторить", completed: false }
];
```

Добавим его в файл index.js (находится в папке src) сразу после последней строчки с import.

Строчку <App /> в этом же файле заменим на:

```
<App tasks={DATA}/>
```

Таким образом, файл index.js выглядит так:

src > JS index.js > ...

```
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  import './index.css';
4  import App from './App';
5  import reportWebVitals from './reportWebVitals';
6
7  const DATA = [
8    { id: "todo-0", name: "Поесть", completed: true },
9    { id: "todo-1", name: "Поспать", completed: false },
10   { id: "todo-2", name: "Повторить", completed: false }
11 ];
12
13 const root = ReactDOM.createRoot(document.getElementById('root'));
14 root.render(
15   <React.StrictMode>
16     <App tasks={DATA}/>
17   </React.StrictMode>
18 );
19
20 // If you want to start measuring performance in your app, pass a function
21 // to log results (for example: reportWebVitals(console.log))
22 // or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
23 reportWebVitals();
```

Далее подключим этот массив в App.js вннутрь функции App() перед строкой с вызовом return. Добавим перед return следующий код, который на основе массива будет формировать <Todo> - элементы:

```
const taskList = props.tasks.map((task) => (
    <Todo
      id={task.id}
      name={task.name}
      completed={task.completed}
      key={task.id}
    />
  )
);
```

Начало функции App() выглядит теперь так:

```
import logo from './logo.svg';
import './App.css';
import Todo from "./components/Todo";

function App(props) {
  const taskList = props.tasks.map((task) => (
    <Todo
      id={task.id}
      name={task.name}
      completed={task.completed}
      key={task.id}
    />
  )
);
  return (
    <div className="todoapp stack-large">
      <h1>Список задач  </h1>
```

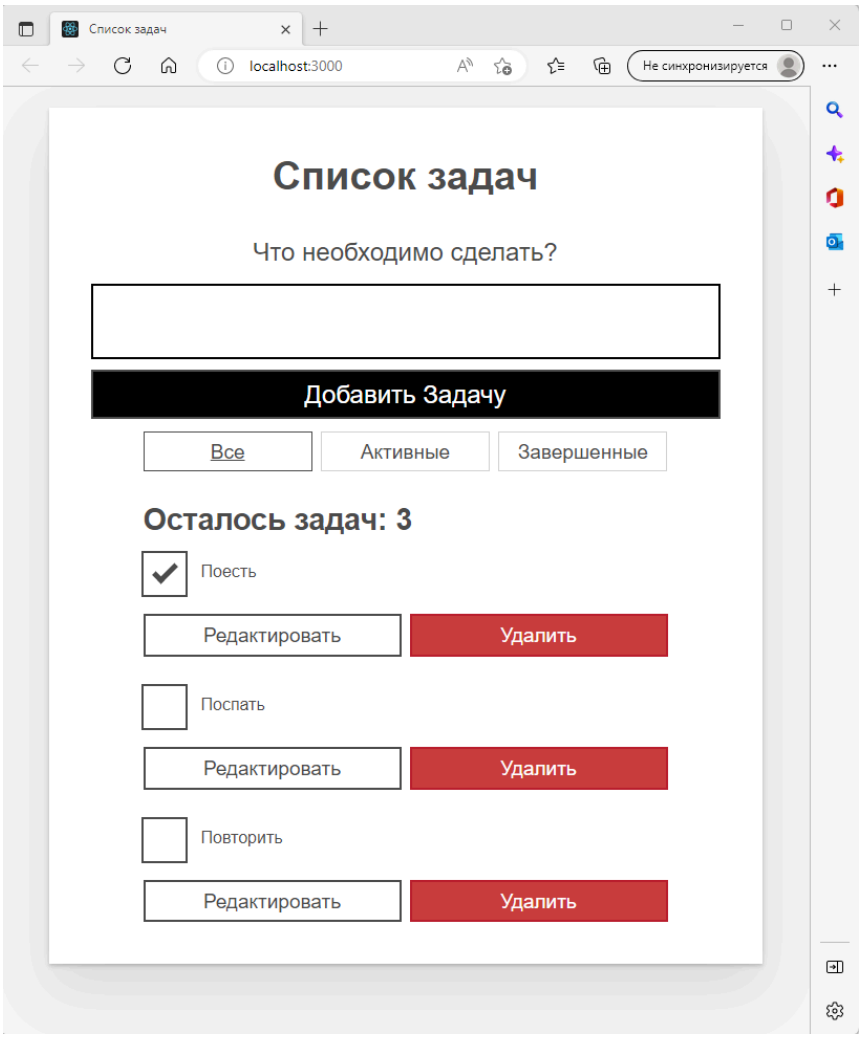
Внутри списка <ul> вместо трех элементов <Todo> добавим одну строку:

```
{taskList}
```

Теперь список <ul> выглядит так:

```
<ul
  role="list"
  className="todo-list stack-large stack-exception"
  aria-labelledby="list-heading">
 {taskList}
</ul>
```

В браузере картина следующая:



Создание компонента <Form/> для добавления новой задачи

В папке components создадим пустой файл Form.js.

В этом файле:

- Импортируйте React в начало файла, как мы это делали в Todo.js.
- Создайте новый Form() компонент с той же базовой структурой, что и Todo().
- Скопируйте теги <form> и все, что между ними, из App.js в Form.js после оператора return.

Содержимое Form.js следующее:

```
import React from "react";
function Form(props) {
  return (
    <form>
      <h2 className="label-wrapper">
        <label htmlFor="new-todo-input" className="label__lg">
          Что необходимо сдедать?
        </label>
      </h2>
```

```
      <input
        type="text"
        id="new-todo-input"
        className="input input__lg"
        name="text"
        autoComplete="off"
      />
      <button type="submit" className="btn btn__primary btn__lg">
        Добавить задачу
      </button>
    </form>
  );
}
export default Form;
export default Form;
```

Создание компонента <FilterButton> для реализации возможности фильтрации по списку задач

В папке components создадим пустой файл FilterButton.js.

Содержимое этого файла следующее (внутри return скопированы строки кода из App.js для первой кнопки):

```
import React from "react";
function FilterButton(props) {
  return (
    <button type="button" className="btn toggle-btn" aria-pressed="true">
      <span className="visually-hidden">Show </span>
      <span>all </span>
      <span className="visually-hidden"> tasks</span>
    </button>
  );
}
export default FilterButton;
```

Окончательное редактирование App.js

Заменим скопированные в пунктах 3.5 и 3.6 строки кода файла App.js на созданные нами компоненты <Form /> и < FilterButton>. App.js выглядит следующим образом:

```
import logo from './logo.svg';
import './App.css';
import Todo from "./components/Todo";
import Form from "./components/Form";
import FilterButton from "./components/FilterButton";
function App(props) {
  const taskList = props.tasks.map((task) => (
    <Todo
      id={task.id}
      name={task.name}
      completed={task.completed}
      key={task.id}
    />
  )
);
  return (
    <div className="todoapp stack-large">
      <h1>Список задач  </h1>
```

```
      <Form />
      <div className="filters btn-group stack-exception">
        <FilterButton />
        <FilterButton />
        <FilterButton />
      </div>
      <h2 id="list-heading">Осталось задач: 3</h2>
      <ul
        role="list"
        className="todo-list stack-large stack-exception"
        aria-labelledby="list-heading">
       {taskList}
      </ul>
    </div>
  );
}
export default App;
```

App.js стал заметно короче.

Перейдем в VS Code в файл Form.js

Между строками function Form(props) { и return(добавим следующий код:

```
 function handleSubmit(e) {
  e.preventDefault();
  alert(Привет!');
}
```

Чтобы использовать эту функцию в качестве обработчика, добавьте атрибут onSubmit внутрь тега <form>, и установите его значение равным {handleSubmit}. Строка теперь имеет такой вид:

```
<form onSubmit={handleSubmit}>
```

Нажмите на кнопку «Добавить Задачу» в браузере. Теперь появляется окошко со словом «Привет».

В приложениях React интерактивность редко ограничивается пределами одного компонента. События, происходящие в одном компоненте, влияют на другие части приложения. Когда мы создаем новые задачи в компоненте <Form />, то при нажатии на кнопку «Добавить Задачу» изменяется список задач, который отображается в компоненте <App />.

Мы хотим, чтобы наша handleSubmit() функция в конечном итоге помогла нам добавить задачу в список, поэтому нам нужен способ передачи информации из <Form /> в <App />.

Механизм, который используется в React, называется в программировании функцией обратного вызова, или callback функцией. Именно функцию

обратного вызова мы можем вызвать в компоненте <Form />, чтобы передать данные в компонент <App />.

В файле App.js после строки function App(props) { добавьте следующий код:

```
function addTask(name) {
  alert(name);
}
```

Теперь необходимо передать функцию addTask() в компонет <Form />, который был нами вставлен ранее внутрь функции App() в файле App.js, как атрибут.

Строка имеет вид:

```
<Form addTask={addTask}/>
```

Теперь можно использовать вызов addTask внутри функции handleSubmit() <Form /> компонента. Обновим код функции handleSubmit() в Form.js следующим образом:

```
function handleSubmit(e) {
  e.preventDefault();
  props.addTask("Hello");
}
```

Попробуйте нажать на кнопку «Добавить задачу» в приложении. Вы увидите окно alert(), которое формируется уже функцией, объявленной внутри App().

Состояния React-компонентов

Мы не можем заставить компоненты React формировать их собственные параметры, но можем отслеживать некоторые собственные данные компонента для дальнейшего использования в веб-приложении. Такие данные, которыми владеет сам компонент, называются состоянием. Состояние — еще один мощный инструмент React, поскольку компоненты не только владеют состоянием, но и могут обновлять его позже.

React имеет множество специальных функций, которые позволяют нам предоставлять новые возможности компонентам, например состояние. Эти функции называются хуками, а useState хук – это именно тот, который нам нужен, чтобы придать нашему компоненту какое-то состояние.

Чтобы использовать хук React, нам нужно импортировать его из модуля React. В Form.js, измените самую первую строку так, чтобы она читалась следующим образом:

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

Теперь мы можем работать с функцией useState() и наделять элементы нашего веб-приложением состоянием, которое можно изменять.

Над строкой function handleSubmit(e){ напишите следующий код:

```
const [name, setName] = useState('Используйте хуки!');
```

В этой строке мы добавили состояние, которое будет записываться в параметр name нашего компонента Form(). setName – это функция, которая позволяет обновлять параметр name (т.е. изменять его значение). Строка 'Используйте хуки!' – это начальное значение параметра name.

Чтобы увидеть это состояние добавим строку

```
value={name}
```

внутрь тега <input> компонента Form().

Тег <input> теперь выглядит так:

```
<input
  type="text"
  id="new-todo-input"
  className="input input__lg"
  name="text"
  autoComplete="off"
  value={name}
/>
```

Фраза «Используйте хуки!» теперь отображается в поле для ввода текста в веб-приложении:

Теперь изменим недавно добавленную строку

```
const [name, setName] = useState('Используйте хуки!');
```

удалив из нее значение по-умолчанию. Строка выглядит теперь так:

```
const [name, setName] = useState('');
```

Далее наверх функции Form(), т.е. после строки function Form(props) {

добавим функцию – обработчик события ввода пользователем символов в строку:

```
function handleChange(e) {
    setName(e.target.value);
  }
```

e.target.value – это как раз и есть тот текст, который введен пользователем.

Внутрь тега <input> после value={name} добавим строку:

```
onChange={handleChange}
```

Теперь тег <input> выглядит так:

```
<input
  type="text"
  id="new-todo-input"
  className="input input__lg"
  name="text"
  autoComplete="off"
  value={name}
  onChange={handleChange}
/>
```

Изменим функцию handleSubmit, срабатывающую по нажатии на кнопку. Функция выглядит так:

```
function handleSubmit(e) {
  e.preventDefault();
  props.addTask(name);
  setName("");
}
```

Попробуйте добавить текст в строку и нажмите кнопку. Теперь введенный вами текст отображается в окне alert().

Перейдем к разработке кода, который будет добавлять новую задачу в список.

Откроем в VS Code файл App.js.

После всех строк import добавим еще одну:

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

Актуальный список задач в файле App.js доступен через вызов props.tasks

tasks – это массив, в котором хранятся задачи. Создадим хук, который позаолит работать с этим массивом задач. Для этого после строки function App(props) {

в файле App.js добавим строку:

```
const [tasks, setTasks] = useState(props.tasks);
```

то есть мы будем работать с параметром tasks, функция для обновления списка задач называется setTasks, а props.tasks – это начальное состояние списка задач.

Объявление taskList внутри функции App() теперь должно выглядеть так:

```
const taskList = tasks.map((task) => (
    <Todo
      id={task.id}
      name={task.name}
      completed={task.completed}
      key={task.id}
    />
  )
);
```

Далее изменим код функции addTask, чтобы она добавляла задачу в список:

```
function addTask(name) {
    const newTask = { id: "id", name, completed: false };
    setTasks([...tasks, newTask]);
  }
```

Посмотрите веб-приложение в браузере. Теперь новая задача будет добавляться в список. Но есть проблема: функция addTask() дает сейчас задачам одинаковый id, что затруднит далее их обработку.

Обычным способом устранения этой проблемы является использование специализированных библиотек, которые умеют создавать уникальные идентификаторы. Одна из таких библиотек называется nanoid.

Для ее установки откроем еще одну командную строку (не закрывая имеющуюся).

Введем команду cd todo_list и нажмем «Enter».

Далее введем команду

```
npm install nanoid
```

Производится загрузка необходимых пакетов.

В App.js после всех import добавим строку:

```
import { nanoid } from "nanoid";
```

Теперь строку

```
const newTask = { id: "id", name, completed: false };
```

заменим на следующую:

```
const newTask = { id: `todo-${nanoid()}`, name, completed: false };
```

Сохраните всё. Обновите веб-приложение в браузере и проверьте его работу.

Обратите внимание: сколько бы мы задач не добавили, в приложении надпись

Осталось задач: 3

не изменяется.

Мы можем исправить это, подсчитав количество элементов в taskList и соответствующим образом изменив текст надписи веб-приложения.

Перед строкой return (добавьте:

```
const headingText = `Осталось задач: ${taskList.length}`;
```

Строку

```
<h2 id="list-heading">Осталось задач: 3</h2>
```

Замените на:

```
<h2 id="list-heading">{headingText}</h2>
```

Работа с отметками о выполнении задачи («галочки»).

Сейчас приложение работает так, что пользователь может ставить и снимать галочки. Однако поставленные или снятые галочки никак не меняют массив задач внутри React-приложения, т.е. React совсем не знает о том, поставил пользователь галочки или нет.

Чтобы исправить это, добавьте перед строчкой

```
const taskList = tasks.map((task) => (
```

следующий код:

```
function toggleTaskCompleted(id) {
    const updatedTasks = tasks.map((task) => {
      // if this task has the same ID as the edited task
      if (id === task.id) {
        // use object spread to make a new object
        // whose `completed` prop has been inverted
        return {...task, completed: !task.completed}
      }
      return task;
    });
    setTasks(updatedTasks);
  }
```

Кроме того, обновите const taskList = tasks.map((task) => (

Он должен быть таким:

```
  const taskList = tasks.map((task) => (
    <Todo
      id={task.id}
      name={task.name}
      completed={task.completed}
      key={task.id}
      toggleTaskCompleted={toggleTaskCompleted}
    />
  )
);
```

Теперь перейдем в файл Todo.js

Найдем строку:

```
<input id={props.id} type="checkbox" defaultChecked={props.completed} />
```

Замени ее на

```
<input
            id={props.id}
            type="checkbox"
            defaultChecked={props.completed}
            onChange={() => props.toggleTaskCompleted(props.id)}
          />
```

Теперь наше приложение способно отслеживать галочки.

Операция удаления задачи

По аналогии с созданием функции addTask для добавления задачи в список, создадим функцию deleteTask для удаления задачи из списка.

В App.js перед строкой

```
const taskList = tasks.map((task) => (
```

добавьте:

```
function deleteTask(id) {

  }
```

Эта функция пока ничего не делает.

Обновим const taskList = tasks.map((task) => (

Он должен выглядеть так:

```
  const taskList = tasks.map((task) => (
    <Todo
      id={task.id}
      name={task.name}
      completed={task.completed}
      key={task.id}
      toggleTaskCompleted={toggleTaskCompleted}
      deleteTask={deleteTask}
    />
  )
);
```

Теперь в Todo.js свяжем наш deleteTask с кнопкой «Удалить»

Заменим код кнопки

```
<button type="button" className="btn btn__danger">
            Удалить <span className="visually-hidden">{props.name}</span>
          </button>
```

на следующий код:

```
<button
            type="button"
            className="btn btn__danger"
            onClick={() => props.deleteTask(props.id)}
          >
            Удалить <span className="visually-hidden">{props.name}</span>
          </button>
```

Наконец, в App.js добавим код в созданную нами функцию deleteTask().

Функция должна выглядеть так:

```
function deleteTask(id) {
    const remainingTasks = tasks.filter((task) => id !== task.id);
    setTasks(remainingTasks);
  }
```

Попробуйте удалить задачи в веб-приложении.

Редактирование задач

Для редактирования задачи создадим в App.js функцию editTask() , по своей сути похожую на deleteTask()

Для этого добавим функцию editTask() в App.js перед строкой const taskList = tasks.map((task) => (

Скопируйте следующий код:

```
function editTask(id, newName) {
  const editedTaskList = tasks.map((task) => {
  // if this task has the same ID as the edited task
    if (id === task.id) {
      //
      return {...task, name: newName}
    }
    return task;
  });
  setTasks(editedTaskList);
}
```

Поместим editTask в <Todo /> компоненты в качестве параметра так же, как мы это делали с deleteTask. Код const taskList = tasks.map((task) => (

Должен выглядеть так:

```
const taskList = tasks.map((task) => (
  <Todo
    id={task.id}
    name={task.name}
    completed={task.completed}
    key={task.id}
    toggleTaskCompleted={toggleTaskCompleted}
    deleteTask={deleteTask}
    editTask={editTask}  //Добавленная строка
  />
)
);
```

Далее перейдем в файл Todo.js

Замените первую строку (import) на

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

После строки export default function Todo(props) {

добавьте:

```
const [isEditing, setEditing] = useState(false);
```

Далее сделаем так, чтобы у задачи в списке было 2 режима просмотра: обычный и режим редактирования.

После строки const [isEditing, setEditing] = useState(false);

поместите следующий большой код:

```
const editingTemplate = (
    <form className="stack-small">
      <div className="form-group">
        <label className="todo-label" htmlFor={props.id}>
          Новое имя для: {props.name}
        </label>
        <input id={props.id} className="todo-text" type="text" />
      </div>
      <div className="btn-group">
```

```
            <button type="button" className="btn todo-cancel">
              Закрыть
              <span className="visually-hidden">renaming {props.name}</span>
            </button>
            <button type="submit" className="btn btn__primary todo-edit">
              Сохранить
              <span className="visually-hidden">new name for {props.name}</span>
            </button>
          </div>
        </form>
      );
      const viewTemplate = (
        <div className="stack-small">
          <div className="c-cb">
              <input
                id={props.id}
                type="checkbox"
                defaultChecked={props.completed}
                onChange={() => props.toggleTaskCompleted(props.id)}
              />
              <label className="todo-label" htmlFor={props.id}>
                {props.name}
              </label>
            </div>
            <div className="btn-group">
              <button type="button" className="btn">
                Редактировать <span
classNameName="visually-hidden">{props.name}</span>
              </button>
              <button
                type="button"
                className="btn btn__danger"
                onClick={() => props.deleteTask(props.id)}
              >
                Удалить <span className="visually-hidden">{props.name}</span>
              </button>
            </div>
        </div>
      );
```

Теперь у нас есть две разные структуры шаблонов — «редактирование» и «просмотр», — определенные внутри двух отдельных констант.

Осталось изменить вызов return, который должен теперь возвратить для отрисовки один из двух шаблонов. Весь имеющийся код return

```
return (
    <li className="todo stack-small">
    <div className="c-cb">
    <input
        id={props.id}
        type="checkbox"
        defaultChecked={props.completed}
        onChange={() => props.toggleTaskCompleted(props.id)}
      />
      <label className="todo-label" htmlFor={props.id}>
        {props.name}
      </label>
    </div>
    <div className="btn-group">
      <button type="button" className="btn">
        Редактировать <span className="visually-hidden">{props.name}</span>
      </button>
      <button
        type="button"
        className="btn btn__danger"
        onClick={() => props.deleteTask(props.id)}
      >
        Удалить <span className="visually-hidden">{props.name}</span>
      </button>
    </div>
  </li>
);
```

замените на строку:

```
return <li className="todo">{isEditing ? editingTemplate :
viewTemplate}</li>;
```

Пока мы не видим интерфейс редактирования задач.

Добавим функционал, чтобы кнопка «Редактировать» переключала шаблоны просмотра. Для этого в Todo.js заменим код кнопки «Редактировать»

```
<button type="button" className="btn">
  Редактировать <span className="visually-hidden">{props.name}</span>
</button>
```

на следующий код:

```
<button type="button" className="btn" onClick={() => setEditing(true)}>
    Редактировать <span className="visually-hidden">{props.name}</span>
  </button>
```

Теперь обновим код кнопки «Закрыть» шаблона редактирования. Для этого заменим код

```
<button type="button" className="btn todo-cancel">
  Закрыть
  <span className="visually-hidden">renaming {props.name}</span>
</button>
```

на

```
<button
    type="button"
    className="btn todo-cancel"
    onClick={() => setEditing(false)}
  >
    Закрыть
    <span className="visually-hidden">renaming {props.name}</span>
  </button>
```

Посмотрите приложение. Теперь мы можем переключаться между режимами просмотра и редактирования для задачи, кнопки «Редактировать» и «Закрыть» работают.

Следующий шаг – реализовать возможность непосредственного редактирования формулировки задачи

Для этого в Todo.js создадим новый хук.

Ниже строки const [isEditing, setEditing] = useState(false);

добавим код:

```
const [newName, setNewName] = useState(props.name);
Сразу после этой строки добавим:
function handleChange(e) {
    setNewName(e.target.value);
  }
```

Чуть ниже найдем код <input id={props.id} className="todo-text" type="text" />

и заменим его на:

```
<input
    id={props.id}
    className="todo-text"
    type="text"
    value={newName}
    onChange={handleChange}
  />
```

После добавленного нами кода

```
function handleChange(e) {
  setNewName(e.target.value);
}
```

добавим еще одну функцию:

```
function handleSubmit(e) {
    e.preventDefault();
    props.editTask(props.id, newName);
    setNewName(newName);
    setEditing(false);
  }
```

Эта функция представляет собой обработчик события onSubmit формы редактирования, которое возникает при нажатии на кнопку «Сохранить».

Тег создания формы, расположенный на пару строчек ниже, следует изменить так:

```
<form className="stack-small" onSubmit={handleSubmit}>
```

Теперь вы можете редактировать формулировки задач непосредственно в веб-приложении.

Добавление фильтров

В файле App.js создадим новый хук для добавления фильтров. Для этого после строчки

```
const [tasks, setTasks] = useState(props.tasks);
```

добавим строку:

```
const [filter, setFilter] = useState('Все');
```

‘All’ означает видеть все задачи. Именно в таком виде мы хотим видеть наш список по умолчанию.

Вверху App.js после всех строчек import добавим:

```
const FILTER_MAP = {
  Все: () => true,
  Активные: (task) => !task.completed,
  Завершенные: (task) => task.completed
};
const FILTER_NAMES = Object.keys(FILTER_MAP);
```

- Фильтр All показывает все задачи, поэтому возвращаемся true по всем задачам.
- Фильтр Active показывает задачи, параметр completed которых равен false.
- Фильтр Completed показывает задачи, параметр completed которых равен true.

В App.js перед строкой const headingText = `Осталось задач: ${taskList.length}`;

добавьте следующий текст:

```
const filterList = FILTER_NAMES.map((name) => (
  <FilterButton
```

```
    key={name}
    name={name}
    isPressed={name === filter}
    setFilter={setFilter}
  />
));
```

Теперь мы заменим три повторяющихся <FilterButton /> несколько ниже на

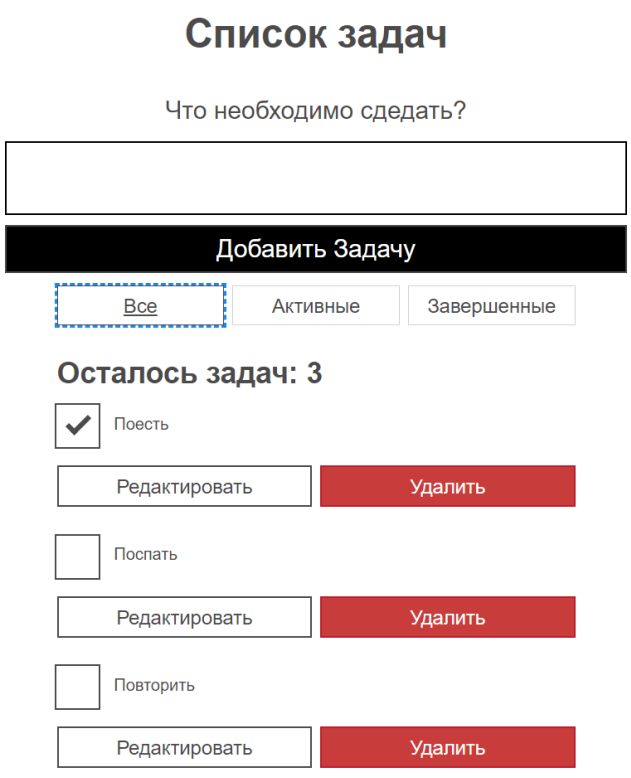
{filterList}

Теперь в VS Code перейдем к файлу FilterButton.js

Заменим содержимое на

```
import React from "react";
function FilterButton(props) {
  return (
    <button
      type="button"
      className="btn toggle-btn"
      aria-pressed={props.isPressed}
      onClick={() => props.setFilter(props.name)}
    >
      <span className="visually-hidden">Show </span>
      <span>{props.name}</span>
      <span className="visually-hidden"> tasks</span>
    </button>
  );
}
export default FilterButton;
```

Сейчас приложение выглядит так, но кнопки фильтров еще не работают:



Остался последний шаг:

В App.js замените код

```
const taskList = tasks.map((task) => (
  <Todo
    id={task.id}
    name={task.name}
    completed={task.completed}
    key={task.id}
    toggleTaskCompleted={toggleTaskCompleted}
    deleteTask={deleteTask}
    editTask={editTask}
  />
)
);
```

на следующий код:

```
const taskList = tasks
 .filter(FILTER_MAP[filter])
 .map((task) => (
   <Todo
     id={task.id}
     name={task.name}
     completed={task.completed}
     key={task.id}
     toggleTaskCompleted={toggleTaskCompleted}
     deleteTask={deleteTask}
     editTask={editTask}
   />
 ));
```

Теперь с учетом галочек, фильтры отбирают задачи.

Попробуйте поработать с получившимся React-приложением.