

Контекст. Компоненты высшего порядка. Знакомство с Redux

React



Оглавление

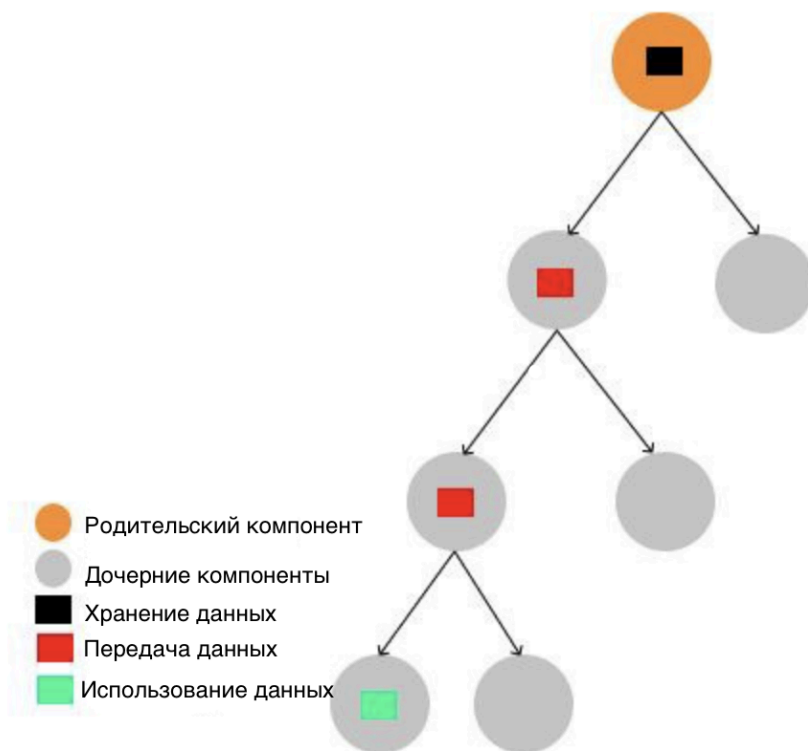
Введение	3
React.Context	3
Компоненты высшего порядка	7
Redux	8
Составляющие элементы Redux	11
Используем redux в компоненте	13
React-redux	16
Redux-devtools	19
Другие state managers	20
Подведем итоги	21
Домашнее задание	21
Дополнительные материалы	21
Использованная литература	21

Введение

В этом уроке мы углубимся в ключевые аспекты разработки на React, начиная с React.Context, обеспечивающего более эффективную передачу данных в компонентах; перейдем к пониманию и применению компонентов высшего порядка для создания более гибких и модульных приложений; изучим Redux для управления состоянием, овладеем созданием экшенов, редюсеров и миддлваров (да-да, все это сможем разобрать в уроке) для управления данными; научимся внедрять Redux в компоненты, обеспечивая четкую связь с состоянием; познакомимся с React-Redux и эффективным интегрированием Redux в React-приложения; и, наконец, ознакомимся с другими менеджерами состояния, что даст вам обширное представление о современных решениях управления состоянием в React, всё это обогатит ваши навыки и позволит создавать мощные и масштабируемые веб-приложения.

React.Context

В настоящий момент в нашем приложении все данные хранятся в стейте компонента Router. При этом, если какому-то компоненту, находящемуся глубоко в иерархии, требуются данные о чатах, их приходится передавать через все промежуточные компоненты:



В нашем приложении это (пока) не критично - данные передаются на 2-3 уровня вниз. Однако, при разрастании приложения это может стать проблемой - представьте, что данные о некотором чате придется передавать через 7-8 промежуточных уровней, на которых компоненты эти данные не используют, а лишь передают дочерним:

```
function GrandParent() {  
  return <Parent message="Hello from the top!" />  
};  
  
function Parent(props) {  
  return <Child message={props.message} />  
}  
  
function Child(props) {  
  return <GrandChild message={props.message} />;  
}  
  
// ...  
  
function GrandGrandGrandChild(props) {  
  return <span>{props.message}</span>;  
}
```

Такое явление называется props drilling. Оно приводит к необходимости писать дублирующийся код (строка `message={props.message}` передаёт значение и вызывает дублирование кода), а также нарушает один из принципов React, согласно которому компонент должен получать лишь те данные, которые ему нужны (промежуточные компоненты не используют данные о чате, а лишь передают их; сами по себе они могли бы существовать и без этих данных).

Для решения этой проблемы React предлагает инструмент под названием Context (не путать с контекстом вызова в JS). С помощью такого подхода мы можем

получать данные в компоненте на любом уровне вложенности, не передавая их через промежуточные компоненты, а получая напрямую из контекста.

Рассмотрим использование контекста.


Сперва контекст необходимо создать:

```
export const MyThemeContext = React.createContext({theme: 'dark'});
```

Функция createContext принимает аргументом значение переменной контекста по умолчанию.

Затем нужно обернуть корневой компонент в провайдер контекста (не обязательно корневой, но самый верхний из тех, которым требуются данные контекста), и указать prop value - то значение, которое и будет передаваться в компоненты:

```
export function App() {  
  return (  
    <MyContext.Provider value={{ theme: 'dark' }}>  
      <Router />  
    </MyContext.Provider>  
  )  
}
```

 Если компонент, использующий контекст, не обернут в подходящий провайдер, то будет использоваться значение переменной контекста по умолчанию - то, которое было передано в createContext.

Наконец, в компоненте, которому необходимы эти данные, нужно получить их. Для функциональных компонентов самый простой способ - использование хука useContext:

```
import { useContext } from "react";  
import { MyThemeContext } from "../../context";
```

```
export default function Example() {
  const contextValue = useContext(MyThemeContext);
  console.log(contextValue); // { theme: 'dark' }
  return <span>Example</span>;
}
```

Существуют и другие способы получения данных из контекста, они подробно описаны в официальной документации.

Контекст предназначен, в первую очередь, для передачи одних и тех же данных компонентам на различном уровне вложенности - с его помощью удобно использовать, к примеру, тему или язык приложения. На основе контекста работают многие библиотеки для React, в т.ч. react-redux, а также темы из MaterialUI.

Количество используемых контекстов никак не ограничено. Вы можете использовать один объект контекста для локализации приложения, другой - для темы, третий - для хранения данных.

```
export function App() {
  return (
    <MyThemeContext.Provider value={{ theme: "dark" }}>
    <DataContext.Provider value={{ messages: ["hello world!"] }}>
    <LocalizationContext value="en">
    <Router />
    </LocalizationContext>
    </DataContext.Provider>
    </MyThemeContext.Provider>
  );
}
```

В данном курсе напрямую контекст не используется, однако понимание принципов работы с ним позволяет лучше освоить работу с библиотеками, использующими его “под капотом”.

Компоненты высшего порядка

В js есть такие паттерны “декоратор”, принцип которых основан на использовании функций высшего порядка (можно почитать дополнительно [ссылка](#))

```
function withLogger(fn) {  
  return function (...args) {  
    console.log(args);  
    return fn(...args);  
  }  
}  
  
const foo = (a, b) => a + b;  
const bar = withLogger(foo);  
  
bar(1, 2);
```

Аналогично, можно использовать этот паттерн с компонентами Реакт:

```
function Example(props) {  
  return <span>{props.message}</span>;  
}  
  
const withLoggerHOC = function (Component) {  
  return (props) => {  
    console.log(props);  
    return <Component {...props} />  
  }  
}  
  
export default withLoggerHOC(Example);
```

Здесь создана функция, принимающая компонент Реакт, и возвращающая этот же компонент, но с некоторой дополнительной функциональностью. В приведенном выше примере это просто логирование пропсов.

Такие компоненты (принимающие и возвращающие компонент) называются компонентами высшего порядка, или HOC (higher order components). Это достаточно часто используемый паттерн в Реакт, с помощью которого удобно, к примеру, передавать в компонент данные из контекста:

```
function Example({theme}) {  
  
  const color = useMemo(() => theme === THEME.DARK ? 'black' :  
    'grey', [theme]);  
  
  return <span style={{ color }}>Example</span>;  
}  
  
const withContext = function(Component) {  
  return (props) => {  
    const contextValue = useContext(MyThemeContext);  
    return <Component {...props} theme={contextValue} />  
  }  
}  
  
export default withContext(Example);
```

Обратите внимание, как изменился код компонента Example. С помощью withContext мы можем подключить любой компонент к контексту, а компонентам теперь знать о контексте не требуется - все данные они получают из пропсов. Такие компоненты намного легче переиспользовать и тестировать.

Redux

Несмотря на то, что сам по себе контекст удобен для “пробрасывания” данных на глубокие уровни иерархии компонентов, большие объемы данных хранить в нем не

представляется удобным. Для этих целей, как правило, используются сторонние библиотеки, одна из которых - Redux.

Redux предоставляет так называемое глобальное хранилище, или глобальный стор (store) для данных - упрощенно говоря, это большой объект. Каждый компонент может получить из него необходимую ему часть данных, либо вызвать их изменение.

Для наилучшего понимания процесса работы с Redux следует учитывать следующие принципы:

1. SST - single source of truth или единый источник истины. Все компоненты и функции, зависящие от некоторых данных, должны получать их из одного и того же источника, а также изменять их в только том же источнике.
2. Иммутабельность. Объект, состояние которого не может быть изменено после создания.

Рассмотрим такой пример:

```
const obj = {  
  x: 1  
}  
  
function updateObj(obj) {  
  obj.x = 2;  
  return obj;  
};
```

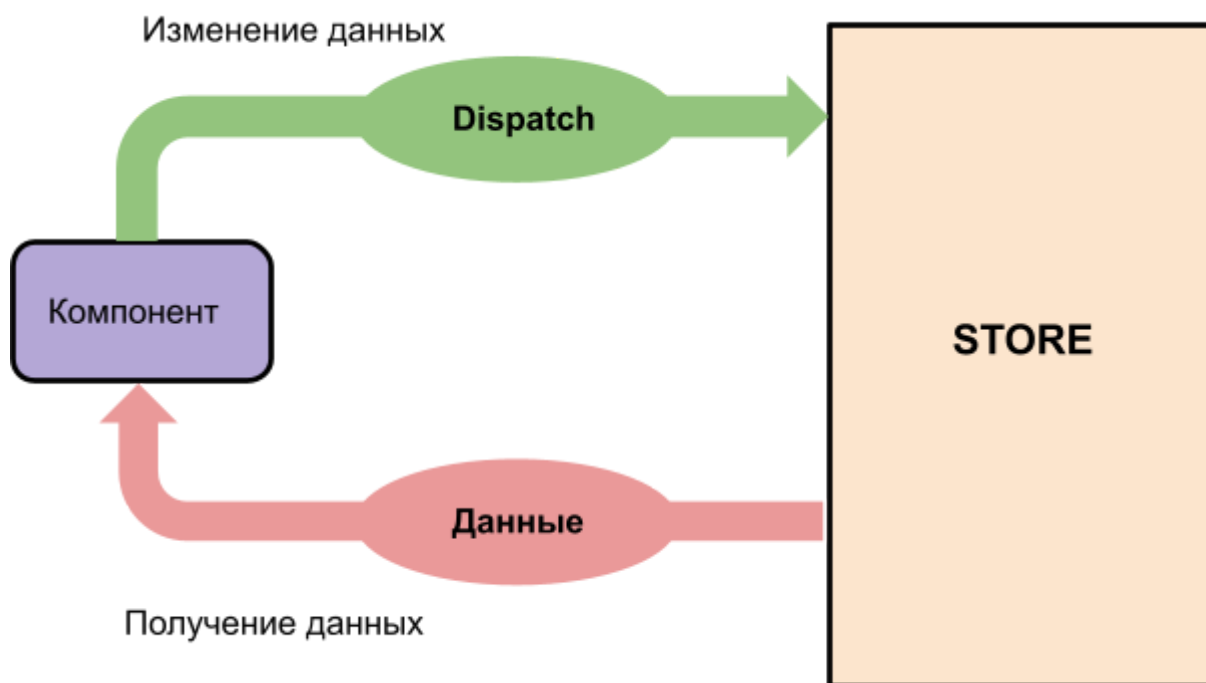
Здесь изменяется значение `obj.x`, однако сам объект, и, следовательно, ссылка на него, остаются теми же. Мутирование данных зачастую приводит к трудноотлаживаемым багам (к примеру, если `obj` используется несколькими компонентами или функциями, и одна из них мутирует его, то поведение остальных становится трудно предсказуемым. Более корректная реализация функции `updateObj` будет выглядеть так:

```
function updateObj(obj) {  
  const newObj = {  
    ...obj,
```

```
x: 2  
};  
  
return newObj;  
};
```

Здесь создается новый объект на основе предыдущего, и в нем изменяется необходимое свойство. Функция `updateObj` не изменяет переданный ей объект.

3. Однонаправленный поток данных. Для понимания этого принципа рассмотрим следующую схему:



Здесь компонент не имеет прямого доступа к хранилищу данных - из компонента нельзя напрямую изменить стор. Для этого требуется вызвать специальную функцию (`dispatch`), с помощью которой изменения и будут внесены в хранилище. После обновления стора компонент получит новые данные и сможет обновиться. Именно это и имеется ввиду под однонаправленностью потока данных - мы можем отправлять данные в стор только через специальные функции, но не можем изменять их напрямую (данные передаются только в направлении, указанном стрелками).

Составляющие элементы Redux

Рассмотрим структуру Redux более подробно.

Основным элементом в этой структуре является сам store - глобальное хранилище данных. Для его создания используется функция createStore (ее использование рассматривается далее). Это специальный объект - помимо самих данных он, в числе прочего, имеет два метода - dispatch и getState. Первый метод служит для изменения данных, второй - для получения. Метод dispatch принимает аргументом action.

Action (экшен), по сути, просто объект. Как правило, у него есть поле type, представляющее собой строку:

```
const EXAMPLE_ACTION = "EXAMPLE_ACTION";

export const exampleAction = {
  type: EXAMPLE_ACTION
}
```

С помощью типов экшенов определяют, какие именно изменения необходимо внести в стор при каждом конкретном экшене, поэтому типы необходимо делать уникальными. Как правило, типы сохраняют в строковые переменные.

Если вызвать метод dispatch с некоторым экшеном, то Redux передаст этот аргумент вместе с данными стора в специальную функцию - reducer (редьюсер). Редьюсер - это функция, которая описывает, какие изменения стора необходимо произвести в ответ на каждый из экшенов. Как правило, они используют switch..case:


```
const initialState = {
  showName: false,
  name: 'Default'
}

const profileReducer = (state = initialState, action) => {
```

```
switch (action.type) {  
  case EXAMPLE_ACTION:  
    return {  
      ...state,  
      showName: !state.showName  
    }  
  default:  
    return state  
  }  
}
```

Здесь объект стейта хранит две переменных - showName и name. Как указано выше, при вызове dispatch с некоторым экшеном Redux вызовет функцию-редьюсер, передав ей текущее состояние стора - state, и сам экшен. В редьюсере мы можем через тип экшена определить, какое именно действие требуется совершить с данными. После создания нового объекта данных (стейта) на основе предыдущего, необходимо вернуть этот объект.

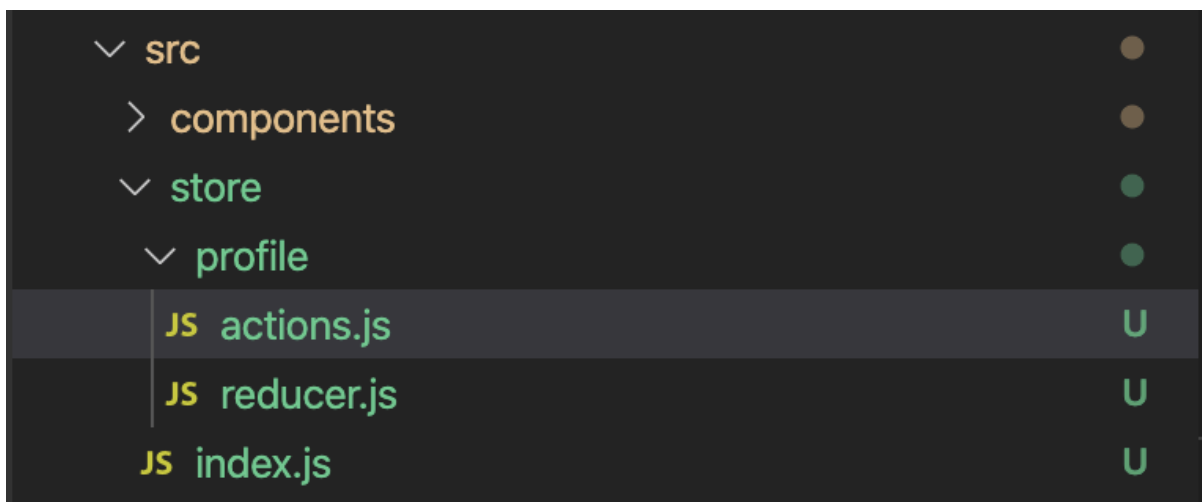
Заметьте, что при первом запуске redux (то есть, когда state === undefined) будет использован аргумент по умолчанию - initialState, начальное состояние нашего стора.

 Из редьюсера необходимо всегда возвращать стейт - новый объект после изменения стейта и старый стейт, если изменения стейта не было (случай default).

Редьюсер необходимо передать в функцию createStore при создании стора:

```
const store = createStore(profileReducer);
```

Как правило, для стора создается отдельная папка, в которой хранятся все файлы, связанные с redux:



Таким образом:

1. Данные хранятся в сторе
2. Для изменения данных необходимо вызвать `store.dispatch`, передав некоторый экшен (“задиспатчить экшен”)
3. Redux вызовет редьюсер, передав туда актуальное состояние данных стора и задиспатченный экшен
4. Редьюсер изменяет данные, возвращая новый объект состояния

Такой подход, хотя и может показаться искусственно усложненным, имеет несколько важных преимуществ:

1. Изменения данных предсказуемы, как правило - достаточно просты (один экшен выполняет одно изменение)
2. Облегчение отладки программы - достаточно просто отследить, какой из экшенов вызвал конкретное изменение в сторе
3. Возможность “путешествия во времени” - несложно откатить изменения, вызванные некоторыми экшенами

Используем redux в компоненте

В целом, функционала самого redux достаточно, чтобы начать использовать стор в нашем приложении. К примеру, на странице профиля мы можем учитывать значение поля `showName` из стора, чтобы определить, следует ли показать имя на данной странице:

```
import { useCallback } from "react";
```

```

import { store } from "../../store/index";
import { toggleShowName } from "../../store/profile/actions";

export default function Profile() {
  const { showName, name } = store.getState().profile;

  const setShowName = useCallback(() => {
    dispatch(toggleShowName);
  }, [dispatch]);

  return (
    <div>
      <h4>Profile</h4>
      {showName && <div>{name}</div>}
    </div>
  );
}

```

Обратите внимание, что используемый здесь объект store - это результат вызова функции createStore. Также мы можем изменить значение, сохраненное в сторе, с помощью dispatch.

Для этого добавим на страницу профиля checkbox:

```

return (
  <div>
    <h4>Profile</h4>
    <input
      type="checkbox"
      checked={showName}
      value={showName}
      onChange={setShowName}
    />
  </div>
);

```

```

/>
<span>Show Name</span>
{showName && <div>{name}</div>}
</div>
);

```

При нажатии на checkbox будем отправлять экшен в стор с помощью store.dispatch:

```

const setShowName = useCallback(() => {
  dispatch(toggleShowName);
}, [dispatch]);

```

Теперь, изменяя состояние чекбокса, мы можем сохранить данные в глобальном сторе. Однако, несмотря на то что стор изменился, компонент не отобразил обновленное имя, так как не произошло повторного вызова функции Profile (не изменились пропсы и стейт). В данном случае самый простой вариант вызвать обновление - вызвать изменение переменной стейта:

```

export default function Profile() {
  const [dummy, setDummy] = useState();
  const { showName, name } = store.getState().profile;
  const dispatch = store.dispatch;

  const setShowName = useCallback(() => {
    dispatch(toggleShowName);
    setDummy({});
  }, [dispatch]);

  return (
    <div>
      <h4>Profile</h4>

```

```

<input
  type="checkbox"
  checked={showName}
  value={showName}
  onChange={setShowName}
/>
<span>Show Name</span>
{showName && <div>{name}</div>}
</div>
);
}

```

Таким образом мы можем получить данные из стора, а также обновить их. Более того, сохраненные в сторе данные можно получать и изменять в любых других компонентах, и они сохранятся, даже если компонент Profile будет размонтирован - в отличие от переменных state компонента.

Однако для того чтобы увидеть обновленное состояние в компоненте, нам придется вручную обновлять компонент (вызывать `setDummy`, причем каждый раз с новым объектом). Это не самый простой и удобный способ.

React-redux

Redux предоставляет возможность более удобного связывания компонентов с данными - библиотека `react-redux`. Сперва установим ее:

```
npm i --save react-redux
```

Затем необходимо обернуть корневой компонент вашего приложения в компонент `Provider`, предоставляемый `react-redux`:

```

// other imports...

import { Provider } from "react-redux";

```



```
export function App() {
  return (
    <Provider store={store}>
      <Router />
    </Provider>
  );
}
```

Этот компонент принимает пропсом store - объект, который мы создали через createStore. Теперь в компоненте мы можем получать данные из стора, используя хук useSelector из react-redux. В этот хук необходимо передавать функцию-селектор. React-redux передаст в селектор стейт, а вернуть из нее необходимо нужную нам в компоненте часть стейта:

```
const { showName, name } = useSelector((state) =>
  state);
```

Вместо store.dispatch в компоненте мы можем теперь использовать хук useDispatch:

```
const dispatch = useDispatch();
```

Таким образом, вместо store.getState() используем useSelector, а вместо store.dispatch - useDispatch.

Помимо более лаконичной и понятной записи, а также отсутствия необходимости импортировать в компоненте объект store, данные хуки обеспечивают автоматическое обновление компонента при изменении данных, от которых он зависит. В этом можно убедиться, убрав искусственно добавленное обновление:

```
export default function Profile() {
```

```

const { showName, name } = useSelector((state) =>
state);

const dispatch = useDispatch();

const setShowName = useCallback(() => {

dispatch(toggleShowName);

}, [dispatch]);

return (

<div>

<h4>Profile</h4>

<input

type="checkbox"

checked={showName}

value={showName}

onChange={setShowName}

/>

<span>Show Name</span>

{showName && <div>{name}</div>}

</div>

```

```
) ;  
  
}
```

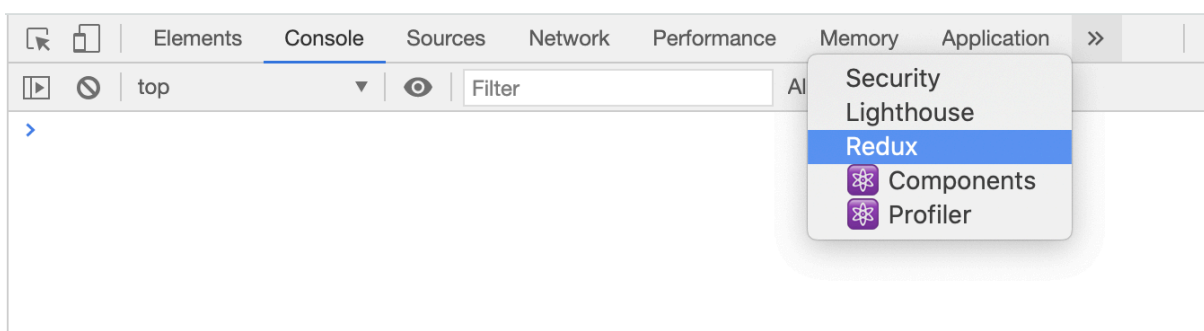
Redux-devtools

Состояние стора можно удобно просматривать при отладке приложения в браузере. Для этого необходимо использовать redux-devtools - расширение для браузера.

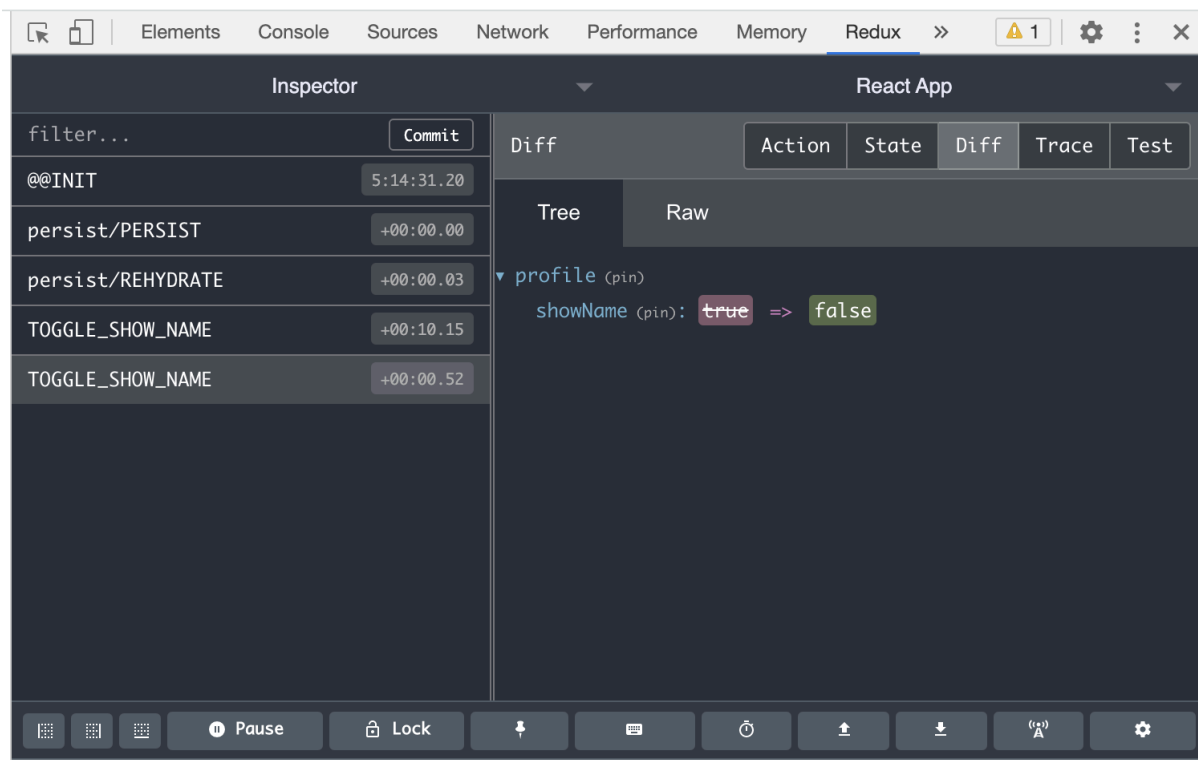
После установки расширения необходимо следующим образом изменить создание store:

```
const store = createStore(  
  profileReducer,  
  window.__REDUX_DEVTOOLS_EXTENSION__ &&  
    window.__REDUX_DEVTOOLS_EXTENSION__()  
);
```

Теперь, открыв инструменты разработчика, перейдем на вкладку redux:



В левой части открывшейся вкладки представлен список экшенов, обработанных стором. В правой - подробная информация о выбранном экшене (вкладка action), состоянии стора на момент попадания туда выбранного экшена (вкладка state), и изменении в сторе, вызванном выбранным экшеном (вкладка diff).



Вы можете изменить состояние чекбокса и убедиться, что redux devtools верно отображает экшны и состояние данных в любой момент времени.

Таким образом, redux позволяет нам сохранять данные в общем для всего приложения сторе, а redux-devtools - удобно просматривать состояние стора в любой момент времени.

Другие state managers

Redux является на сегодняшний день самым популярным стейт менеджером для приложений на Реакт. Однако, помимо него, популярностью пользуются и другие библиотеки:

MobX - стейт менеджер, работа которого основана на использовании паттерна observer. По сравнению с redux является более гибким инструментом, однако накладывает на разработчика дополнительные обязательства (к примеру, в отличие от redux, mobX никак не регламентирует создание стора).

Recoil - относительно “молодая” библиотека. Предлагает “атомарный” подход к стейту и API, основанный на хуках.

Hookstate - одна из менее популярных и простых библиотек. Относительно небольшой стейт менеджер (по размеру пакета), основанный на хуках.

Подведем итоги

В заключение, изучение описанных в методичке тем позволит вам уверенно войти в мир разработки на React, обогатив ваши навыки и понимание принципов создания современных веб-приложений. От React.Context до использования менеджеров состояния и React-Redux, вы узнали инструменты, способствующие более эффективной передаче данных, управлению состоянием и созданию модульных компонентов. Это знание станет вашим фундаментом для разработки масштабируемых и гибких приложений, открывая перед вами бесконечные возможности для создания современных веб-решений. Важно практиковаться, экспериментировать и продолжать углубляться в темы, так как развитие в сфере веб-разработки — это постоянный процесс, и с каждым новым шагом вы будете приближаться к мастерству.

Использованная литература

1. [Официальный сайт React Router](#)
2. [Словарь терминов React](#)