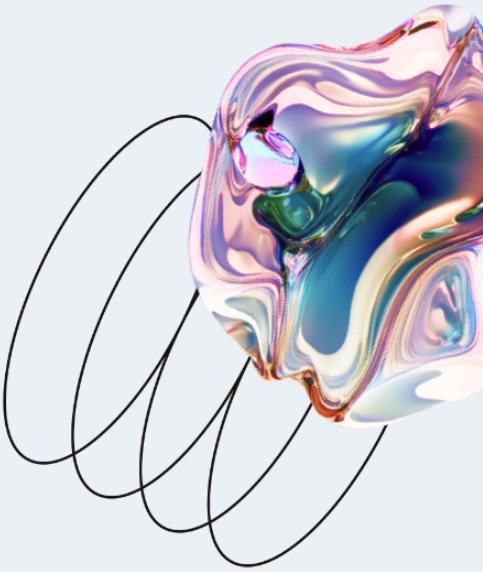
GeekBrains



Redux middlewares. Redux-persist

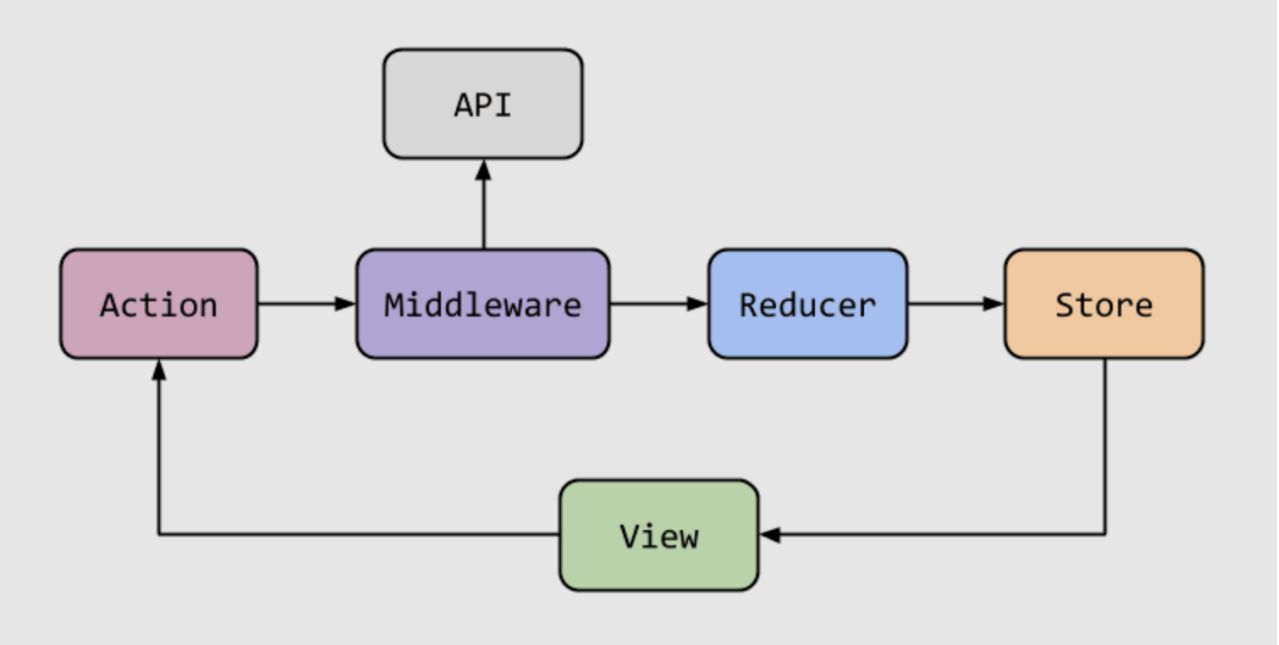
React

Оглавление

Теория урока

Redux middlewares

Рассматривая поток данных в компоненте (действие пользователя - обновление данных стейта - обновление визуального представления), можно провести аналогию с потоком данных redux (компонент диспатчит action - reducer обновляет стор - обновление данных в компоненте). Компонент, при этом, предоставляет возможность выполнять сайд-эффекты (useEffect или componentDidMount/didUpdate). Такую “точку доступа” к побочным эффектам предоставляет и redux - она называется middlewares.



Middleware (мидлвар) можно рассматривать как посредника, который может либо просто передать экшен далее в редьюсер (или по цепочке в следующий мидлвар), либо выполнить некоторое действие, как правило - асинхронное. Здесь и далее под асинхронными действиями понимаются все действия, которые будут выполняться не в тот же момент, когда был задиспатчен экшен, а через некоторое, возможно, неизвестное на данный момент, время (к примеру, миддлвар может сделать запрос к серверу, дождаться ответа и отправить в стор данные - для этой операции время ответа сервера нам неизвестно; или поставить таймаут, дождаться его окончания и отправить в стор данные).

По большому счету, мидлвар представляет из себя просто функцию определенного вида:

```
const middleware = store => next => action => {
    console.log('We can do side effects here!');

    setTimeout(() => {
      console.log('timeouts, api calls etc');
    }, 1000);

    return next(action)
}
```

Для работы данного миддлвара необходимо подключить его к стору:

```
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

export const store = createStore(
  combineReducers({
    chats: chatsReducer,
    profile: profileReducer,
    messages: messagesReducer,
  }),
  composeEnhancers(applyMiddleware(middleware))
);
```

Здесь использование composeEnhancers необходимо для работы redux-devtools.

Если такой мидлвар будет подключен к redux, то при диспатче экшена redux вызовет этот миддлвар, передав аргументом стор, затем вызовет результирующую функцию, передав туда аргументом специальную функцию next, а затем снова вызовет результирующую функцию, передав туда сам экшен. Данный мидлвар может совершать любые, в т.ч. асинхронные действия, но для передачи экшена в редьюсер (или следующий мидлвар) он должен вернуть результат вызова функции next, с action в качестве аргумента. Если миддлвар не вернет вызов next (т. е., будет отсутствовать строка return next(action)), то данный экшен не пойдет дальше по цепочке - в редьюсеры или следующие миддлвары. В этом случает до редьюсера

экшены не дойдут - миддлвар как бы “проглотит” их, включая и те экшены, которые диспатчатся из самого же миддлвара.

Это может быть допустимо (и иногда необходимо) для некоторых из экшенов, которые мы диспатчим - в таком случае следует обернуть return next(action) в некоторое условие, например:

```
const middleware = store => next => (action) => {
    if (action.type !== ADD_MESSAGE) {
       return next(action)
    }
    // ... асинхронные действия, которые будут вызваны только
    // если action.type === ADD_MESSAGE
}
```

К примеру, мы можем использовать следующий миддлвар, чтобы перенести в redux логику ответа робота:

```
const middleware = store => next => (action) => {
    if (action.type === ADD_MESSAGE && action.message.author !== AUTHORS.BOT)
    {
      const botMessage = {/* ... */};
      setTimeout(() => store.dispatch(addMessage(botMessage)), 2000);
    }

    return next(action)
}
```

Здесь миддлвар, при получении нового экшена типа ADD_MESSAGE проверяет автора сообщения, и при необходимости устанавливает таймаут на добавление нового сообщения от бота.

Однако, на практике для таких распространенных задач как асинхронные запросы, используются библиотеки, предоставляющие готовые миддлвары - наиболее популярными являются redux-thunk и redux-saga.

К стору можно подключать любое количество мидлваров - redux будет вызывать их в том порядке, в котором они подключены:

```
const store = createStore(
  reducer,
  applyMiddleware(
    rafScheduler,
    timeoutScheduler,
    thunk,
    vanillaPromise,
    readyStatePromise,
    logger,
    crashReporter
  )
)
```

Redux-thunk

Redux-thunk - один из самых простых миддлваров. Thunk (искаженный глагол think в прошедшем времени) - это некоторая функция, служащая оберткой для выражения, и откладывающая его вычисление.

```
// немедленное вычисление
const x = 1 + 2;

// отложенное вычисление,
// будет выполнено только при вызове foo
const foo = () => 1 + 2;
```

Redux-thunk позволяет использовать action creators, которые возвращают не объект экшен, а функцию. Далее под thunk понимается именно такая функция-обертка, а под redux-thunk - сама библиотека, используемая в качестве миддлвара.

Рассмотрим работу с redux-thunk на примере задачи с ответом робота.

Сперва установим библиотеку:

```
npm i --save redux-thunk
```

Затем подключим thunk к стору:

```
import thunk from 'redux-thunk';
// ...

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

export const store = createStore(
  combineReducers({
    chats: chatsReducer,
    profile: profileReducer,
    messages: messagesReducer,
  }),
  composeEnhancers(applyMiddleware(thunk))
);
```

Идея использования этой библиотеки заключается в следующем: если в стор диспатчится экшен, который является объектом (как во всех предыдущих примерах с экшенами), то redux-thunk просто пропустит ее дальше - следующему миддлвару или редьюсеру - т.е., стандартный синхронный порядок действий. Однако, если в качестве экшена будет передана функция, то redux-thunk вызовет ее, передавая ей аргументами функции dispatch и getState.

Упрощенно миддлвар thunk, подключаемый к стору, выглядит так (сравните с миддлваром из предыдущего раздела):

```
const thunk = (store) => (next) => (action) => {
    if (typeof action === 'function') {
      return action(store.dispatch, store.getState);
    }

    return next(action);
  };
}
```

К примеру, при использовании redux-thunk мы можем использовать в качестве action creator следующую конструкцию:

```
const addMessageWithThunk = (chatId, message) => (dispatch, getState) => {
      dispatch(addMessage(chatId, message));
      if (message.author !== AUTHORS.BOT) {
         const botMessage = {/* ... */};
```

```
            setTimeout(() => dispatch(addMessage(chatId, botMessage)), 2000);
        }
    }
}
```

Без использования thunk такой экшен вызовет ошибку - так как редьюсер ожидает сериализуемый объект.

Так как thunk, названный addMessageWithThunk, имеет доступ к стору (через переданные ей dispatch и getState), мы имеем возможность, как и в миддлваре из предыдущего раздела, запустить некоторые асинхронные действия, дождаться их окончания, и передать эти данные в стор с помощью нового экшена. Обратите внимание, что новый экшен, запущенный из нашего thunk, также попадет в redux-thunk, но, если он не является функцией (addMessage(chatId, message) возвращает объект), то будет просто передан дальше.

Теперь логика ответа бота, ранее находившаяся в useEffect, может быть удалена из компонента - вместо нее будем просто диспатчить thunk:

```
const onAddMessage = useCallback((message) => {
  dispatch(addMessageWithThunk(chatId, message));
}, [chatId, dispatch]);
```

Redux-saga

Другой распространенной библиотекой для миддлвар является redux-saga. Она несколько сложнее в использовании, однако предоставляет несколько больше возможностей.

Redux-saga использует функции-генераторы для работы с асинхронными запросами. Подключение и использование этой библиотеки выглядит следующим образом:

store/index.js

```
import { createStore, applyMiddleware } from 'redux'
import createSagaMiddleware from 'redux-saga'

import reducer from './reducers'
import mySaga from './sagas'

const sagaMiddleware = createSagaMiddleware();

const store = createStore(
  reducer,
  applyMiddleware(sagaMiddleware)
);

sagaMiddleware.run(mySaga);
```

sagas.js

```
import { call, put, takeLatest, delay } from 'redux-saga/effects'

function* onAddMessageWithSaga(action) {
   yield put(addMessage(action));
   if (action.message.author !== AUTHORS.BOT) {
      const botMessage = {/* ... */};
      yield delay(2000);
      yield put(addMessage(chatId, botMessage));
   }
}


function* mySaga() {
  yield takeLatest("MESSAGES::ADD_MESSAGE_WITH_SAGA", onAddMessageWithSaga);
}

export default mySaga;
```

Здесь с помощью takeLatest мы указали миддлвару, какой экшен необходимо “перехватить” (в нашем случае - экшен с типом “MESSAGES::ADD_MESSAGE_WITH_SAGA”). При диспатче этого экшена будет вызвана функция-генератор onAddMessageWithSaga, причем аргументом ей будет передан “перехваченный” экшен.

Далее, с помощью ключевого слова yield и т.н. эффектов саги мы указываем, какие действия необходимо совершить. К примеру, select позволяет получить данные из стора, put - диспатчит в стор переданный экшен, call - вызывает переданную

функцию, delay - останавливает выполнение на указанный промежуток времени и т.д.

Используя для перехвата экшена takeLatest, мы указываем, что для запуска генератора будет всегда использован последний экшен. Если после начала работы генератора будет задиспатчен ещё один подходящий экшен, то работа генератора будет перезапущена с начала - такой подход особенно удобен, к примеру, при отправке сетевых запросов. Помимо этого эффекта, экшены можно перехватывать и другими - например, takeEvery будет запускать генератор на каждый подходящий экшен.

Особенно удобными становятся при работе с генераторами как раз асинхронные действия - к примеру, задержку на 2 секунды мы задаем, используя эффект delay. Эффект retry позволяет задать автоматические повторные вызовы функции при возникновении ошибки. Можно запускать действия параллельно - с помощью race или all.

Компоненты при использовании саг работают аналогично рассмотренным в примерах с thunk, за исключением того, что из компонента будет диспатчиться не thunk, а дополнительный экшен, вызывающий работу саги.

Redux-observable

Еще одна библиотека для создания миддлваров redux - redux-observable. Она основана на использовании паттерна observer и использует библиотеку RxJS. Принципиальное отличие ее от redux-thunk заключается в том, что мы, аналогично redux-saga, сперва настраиваем т.н. epic (упрощенно - канал, по которому будут “проходить” экшены), описывая действия, которые необходимо с этими экшенами совершить:

```
const pingEpic = action$ => action$.pipe(
  filter(action => action.type === 'PING'),
  mapTo({ type: 'PONG' })
);
```

```
// и в компоненте:
dispatch({ type: 'PING' });

// пример выше эквивалентен следующей записи:
dispatch({ type: 'PING' });
dispatch({ type: 'PONG' });
```

Redux-persist

Итак, мы перенесли данные в стор. Однако, при обновлении страницы все данные в сторе очищаются - при каждом обновлении стор создается заново, не сохраняя данные. Для решения этой проблемы будем использовать библиотеку redux-persist.

Redux-persist сохраняет данные в локальном хранилище (по умолчанию используется localStorage, однако можно использовать и другие локальные хранилища). При запуске приложения redux-persist проверяет локальное хранилище на наличие данных, и, в случае, если они есть, наполняет ими стор.

Установим библиотеку:

```
npm i --save redux-persist
```

И изменим создание стора следующим образом:

```
import thunk from 'redux-thunk';
import { createStore } from 'redux';
import { persistStore, persistReducer } from 'redux-persist';
import storage from 'redux-persist/lib/storage'; // localStorage

// ...

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

// создаем объект конфигурации для persist
const persistConfig = {
  key: 'root',
  storage,
}

// комбинируем редьюсеры
```

```
const rootReducer = combineReducers({
    chats: chatsReducer,
    profile: profileReducer,
    messages: messagesReducer,
});

// оборачиваем редьюсеры в persist
const persistedReducer = persistReducer(persistConfig, rootReducer);

// создаем store с использованием persistedReducer
export const store = createStore(
    persistedReducer,
    composeEnhancers(applyMiddleware(thunk))
);

// создаем persistor
export const persistor = persistStore(store);
```

Объект persistConfig содержит конфигурацию для redux-persist. При создании этого объекта необходимо указать, как минимум, свойства key и storage. Свойство key указывает ключ, по которому в локальном хранилище будут сохраняться и получаться данные. Свойство storage указывает хранилище, которое будет использовано для хранения данных.

Помимо этих свойств, redux-persist позволяет указывать и другие настройки - например, св-ва blacklist и whitelist служат для указания названий редьюсеров, состояние которых будет сохраняться в хранилище или игнорироваться соответственно.

Свойство stateReconciler служит для указание способа, которым восстановленые из хранилища данные будут слиты с текущим состоянием стора (к примеру, по умолчанию используется autoMergeLevel1 - т.е. данные из localStorage перепишут ту часть стора, ключи которой совпадают с ключами данных в хранилище):

```
Данные из хранилища: { foo: incomingFoo }
Данные в сторе: { foo: initialFoo, bar: initialBar }
Результат: { foo: incomingFoo, bar: initialBar }
```

Level1 здесь показывает, что сравнения объектов глубже одного уровня не произойдет.

Помимо autoMergeLevel1, redux-persist предоставляет hardSet и autoMergeLevel2. Первый производит полное перезаписывание данных из начального состояния:

```
Данные из хранилища: { foo: incomingFoo }
Данные в сторе: { foo: initialFoo, bar: initialBar }
Результат: { foo: incomingFoo }
```

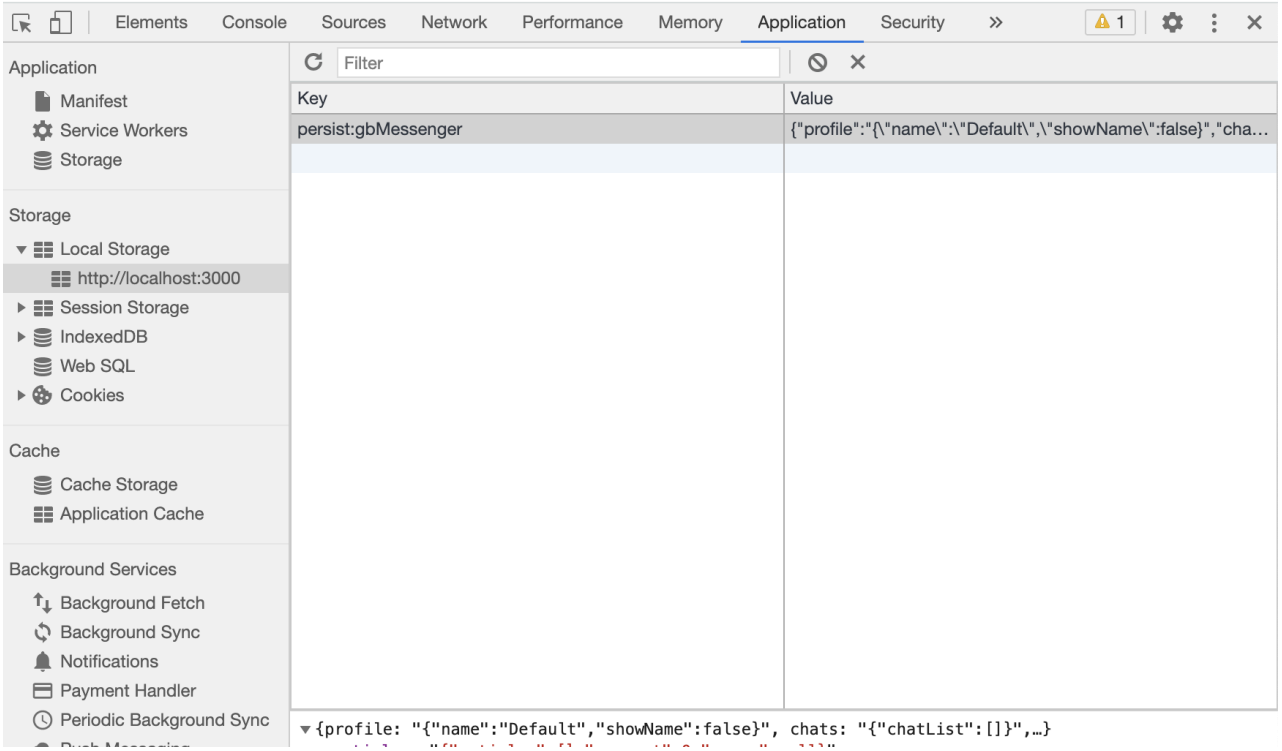
Второй - слияние с проверкой вложенности до двух уровней.

После создания стора необходимо обернуть компоненты приложения в компонент PersistGate (PersistGate должен находится в иерархии ниже Provider):

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

Этому компоненту пропсом передается объект persistor. Также возможно указать (необязательный) проп loading - в него передают компонент, который будет использоваться, пока в сторе не будут восстановлены данные из хранилища. При использовании localStorage восстановление данных, как правило, происходит настолько быстро, что данный компонент не будет виден, однако это может быть полезно при использовании хранилища с асинхронным доступом (например, AsyncStorage в react-native).

Кроме того, в localSttorage теперь находится запись с ключом “gbMessenger” (ключ, указанный в persistConfig):



Теперь при обновлении страницы данные, хранящиеся в сторе, не потеряются, а будут восстановлены из localStorage.

Redux-persist служит для сохранения данных на стороне клиента (localStorage, sessionStorage, filesystem и т.п.). Следует иметь ввиду, что пользователь имеет доступ к этим данным, и может в любой момент удалить их. Кроме того, следует помнить, что все перечисленные выше хранилища не являются безопасными (к примеру, доступ к localStorage может иметь любой скрипт, выполняющийся на странице) - не следует хранить в них важные данные в незашифрованном виде.

Презентационные компоненты и компоненты-контейнеры

В нашем приложении данные хранятся в сторе, обрабатываются в миддлваре, а компоненты практически не содержат существенной логики - в основном вся логика в них направлена на получение данных для отображения. Вместе с тем, логика и разметка находятся в компонентах вместе - то есть, например,

MessageField отвечает и за разметку (включая стили), и за связь со стором через useDispatch/useSelector.

Такое смешение задач внутри одного компонента может привести к ненужному усложнению кода, его низкой читабельности, а также затруднить тестирование и переиспользование компонентов. Для решения такой проблемы в React часто используется паттерн “Компонент-контейнер и презентационный компонент” (containers and presentational components, также containers and dummy components). Смысл его заключается в следующем - компонент, содержащий большое количество логики и разметки, разделяется на два - один отвечает за логическую часть (получение, обработка, отправка данных и т.п.), другой - за презентационную - то, как компонент выглядит (разметка, стили и т.п.). Первый компонент рендерит второй, передавая ему данные (и, при необходимости, некоторые функции) в виде пропсов. Первый компонент, соответственно, является контейнером, второй - презентационным компонентом.

Презентационный компонент:

1. Отвечает за внешний вид
2. Не зависят от других частей приложения (таких как redux)
3. Содержат свою разметку и свои стили
4. Не содержат свой стейт
5. Получают данные и коллбэки только через пропсы

Компонент-контейнер:

1. Отвечает за логику работы с данными (связь со стором, обработка данных и т.п.)
2. Может быть связан с внешним источником данных
3. Не содержит стилей и DOM-элементов, за исключением, при необходимости, элементов-оберток (как правило div или фрагмент React)
4. Может иметь стейт
5. Передает данные и коллбэки пропсами презентационным компонентам

Как презентационный компонент, так и компонент-контейнер может рендерить и контейнеры, и другие презентационные компоненты.

К примеру, при использовании данного паттерна компонент MessageField можно разделить на два следующих компонента:

MessageFieldContainer.js

```
const MessageFieldContainer = () => {
  const { chatId } = useParams();

  const chats = useSelector(getChatList);
  const messageList = useSelector(selectMessageList);
  const dispatch = useDispatch();

  const onAddMessage = useCallback(
    (message) => {
      dispatch(addMessageWithThunk(chatId, message));
    },
    [chatId]
  );

  const onAddChat = useCallback((newChatName) => {
    dispatch(addChat(newChatName));
  }, []);

  if (!chatId) {
    return (
      <>
        <ChatList chats={chats} chatId={null} onAddChat={() => {}} />
      </>
    );
  }

  if (!chats[chatId]) {
    return <Redirect to="/nochat" />;
  }

  return (
    <MessageField
      chatId={chatId}
      messages={messageList[chatId]}
      onAddChat={onAddChat}
      onAddMessage={onAddMessage}
    />
  );
};
```

MessageField.js

```
const MessageField = ({ chatId, messages, onAddMessage, onAddChat }) => (
  <section className="message-field">
    <header className="message-header">Header</header>
    <div className="wrapper">
      <div>
        <ChatList chatId={chatId} onAddChat={onAddChat} />
      </div>
      <div>
        <MessagesList messages={messages} />
        <Input onAddMessage={onAddMessage} />
      </div>
    </div>
  </section>
);
```

Такой подход следует принципу разделения ответственностей, упрощает чтение, отладку и тестирование кода. Однако, это не означает, что следует применять его для всех компонентов - в первую очередь его использование должно быть продиктовано необходимостью упростить код. К примеру, рассмотрим компонент Header, который выглядит следующим образом:

```
export const Header = () => {
  const name = useSelector(selectName);

  return <header className="header">hello, {name}</header>
}
```

Количество логики и разметки в нем минимально, и, хотя, технически, в нем можно провести разделение на логическую и презентационную часть - разделение такого компонента может привести к излишнему усложнению кода.

В целом, необходимо помнить, что, хотя такой подход является полезным для больших компонентов, его “бездумное” применение для каждого компонента может принести больше вреда, чем пользы.

Использованная литература

1. Официальный сайт https://reactrouter.com/en/6.14.2
2. https://ru.legacy.reactjs.org/docs/glossary.html#propschildren