**Redux Essentials, Part 6: Performance and Normalizing Data**

## Adding User Pages

Над добавить стр юзеров.

Компонент

features/users/UsersList.js

import React from 'react'  
import { useSelector } from 'react-redux'  
import { Link } from 'react-router-dom'  
import { selectAllUsers } from './usersSlice'  
  
export const UsersList = () => {  
 const users = useSelector(selectAllUsers)  
  
 const renderedUsers = users.map(user => (  
 <li key={user.id}>  
 <Link to={`/users/${user.id}`}>{user.name}</Link>  
 </li>  
 ))  
  
 return (  
 <section>  
 <h2>Users</h2>  
  
 <ul>{renderedUsers}</ul>  
 </section>  
 )  
}

Слайс

eatures/users/usersSlice.js

export default usersSlice.reducer  
  
export const selectAllUsers = state => state.users  
  
export const selectUserById = (state, userId) =>  
 state.users.find(user => user.id === userId)

Страница отдельного юзера

features/users/UserPage.js

import React from 'react'  
import { useSelector } from 'react-redux'  
import { Link } from 'react-router-dom'  
  
import { selectUserById } from '../users/usersSlice'  
import { selectAllPosts } from '../posts/postsSlice'  
  
export const UserPage = ({ match }) => {  
 const { userId } = match.params  
  
 const user = useSelector(state => selectUserById(state, userId))  
  
 const postsForUser = useSelector(state => {  
 const allPosts = selectAllPosts(state)  
 return allPosts.filter(post => post.user === userId)  
 })  
  
 const postTitles = postsForUser.map(post => (  
 <li key={post.id}>  
 <Link to={`/posts/${post.id}`}>{post.title}</Link>  
 </li>  
 ))  
  
 return (  
 <section>  
 <h2>{user.name}</h2>  
  
 <ul>{postTitles}</ul>  
 </section>  
 )  
}

Новый путь

App.js

<Route exact path="/posts/:postId" component={SinglePostPage} />  
 <Route exact path="/editPost/:postId" component={EditPostForm} />  
 <Route exact path="/users" component={UsersList} />  
 <Route exact path="/users/:userId" component={UserPage} />  
 <Redirect to="/" />

Ну ии в меню

app/Navbar.js

export const Navbar = () => {  
 return (  
 <nav>  
 <section>  
 <h1>Redux Essentials Example</h1>  
  
 <div className="navContent">  
 <div className="navLinks">  
 <Link to="/">Posts</Link>  
 <Link to="/users">Users</Link>  
 </div>  
 </div>  
 </section>  
 </nav>  
 )  
}

## Adding Notifications

Нужно добавить оповещения, причем свежие. У нас они тоже фейк

Слайс

features/notifications/notificationsSlice.js

import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'

import { client } from '../../api/client'

export const fetchNotifications = createAsyncThunk(

'notifications/fetchNotifications',

async (\_, { getState }) => {

const allNotifications = selectAllNotifications(getState())

const [latestNotification] = allNotifications

const latestTimestamp = latestNotification ? latestNotification.date : ''

const response = await client.get(

`/fakeApi/notifications?since=${latestTimestamp}`

)

return response.data

}

)

const notificationsSlice = createSlice({

name: 'notifications',

initialState: [],

reducers: {},

extraReducers(builder) {

builder.addCase(fetchNotifications.fulfilled, (state, action) => {

state.push(...action.payload)

// Sort with newest first

state.sort((a, b) => b.date.localeCompare(a.date))

})

}

})

export default notificationsSlice.reducer

export const selectAllNotifications = state => state.notifications

### Thunk Arguments

Что-то про виды второго аргумента

Вторым аргументом нашего создателя полезной нагрузки является thunkAPIобъект, содержащий несколько полезных функций и фрагментов информации:

* dispatchи getState: актуальные dispatchи getStateметоды из нашего магазина Redux. Вы можете использовать их внутри переходника для отправки дополнительных действий или получения последнего состояния хранилища Redux (например, чтения обновленного значения после отправки другого действия).
* extra: «дополнительный аргумент», который можно передать в промежуточное программное обеспечение Thunk при создании хранилища. Обычно это своего рода оболочка API, например набор функций, которые знают, как выполнять API-вызовы к серверу вашего приложения и возвращать данные, так что вашим переходникам не обязательно содержать все URL-адреса и логику запросов непосредственно внутри.
* requestId: уникальное случайное значение идентификатора для этого вызова Thunk. Полезно для отслеживания статуса отдельного запроса.
* signal: AbortController.signalфункция, которую можно использовать для отмены текущего запроса.
* rejectWithValue: утилита, которая помогает настроить содержимое действия rejected, если преобразователь получает ошибку.

### Adding the Notifications List

Список уведомлений

Компонент

features/notifications/NotificationsList.js

import React from 'react'  
import { useSelector } from 'react-redux'  
import { formatDistanceToNow, parseISO } from 'date-fns'  
  
import { selectAllUsers } from '../users/usersSlice'  
  
import { selectAllNotifications } from './notificationsSlice'  
  
export const NotificationsList = () => {  
 const notifications = useSelector(selectAllNotifications)  
 const users = useSelector(selectAllUsers)  
  
 const renderedNotifications = notifications.map(notification => {  
 const date = parseISO(notification.date)  
 const timeAgo = formatDistanceToNow(date)  
 const user = users.find(user => user.id === notification.user) || {  
 name: 'Unknown User'  
 }  
  
 return (  
 <div key={notification.id} className="notification">  
 <div>  
 <b>{user.name}</b> {notification.message}  
 </div>  
 <div title={notification.date}>  
 <i>{timeAgo} ago</i>  
 </div>  
 </div>  
 )  
 })  
  
 return (  
 <section className="notificationsList">  
 <h2>Notifications</h2>  
 {renderedNotifications}  
 </section>  
 )  
}

Меню

import React from 'react'

import { useDispatch } from 'react-redux'

import { Link } from 'react-router-dom'

import { fetchNotifications } from '../features/notifications/notificationsSlice'

export const Navbar = () => {

const dispatch = useDispatch()

const fetchNewNotifications = () => {

dispatch(fetchNotifications())

}

return (

<nav>

<section>

<h1>Redux Essentials Example</h1>

<div className="navContent">

<div className="navLinks">

<Link to="/">Posts</Link>

<Link to="/users">Users</Link>

<Link to="/notifications">Notifications</Link>

</div>

<button className="button" onClick={fetchNewNotifications}>

Refresh Notifications

</button>

</div>

</section>

</nav>

)

}

App.js

// omit imports

import { NotificationsList } from './features/notifications/NotificationsList'

function App() {

return (

<Router>

<Navbar />

<div className="App">

<Switch>

<Route exact path="/notifications" component={NotificationsList} />

// omit existing routes

<Redirect to="/" />

</Switch>

</div>

</Router>

)

}

### Showing New Notifications

Каждый раз, когда мы нажимаем «Обновить уведомления», в наш список будет добавляться еще несколько записей уведомлений.

Давайте добавим немного логики, чтобы отслеживать, какие уведомления были прочитаны, а какие из них «новые». Это позволит нам отображать количество «Непрочитанных» уведомлений в виде значка на вкладке «Уведомления» на панели навигации, а также отображать новые уведомления другим цветом.

Наш поддельный API уже отправляет обратно записи уведомлений с полями isNewи read, поэтому мы можем использовать их в нашем коде.

Во-первых, мы обновим, notificationsSliceчтобы иметь редуктор, который помечает все уведомления как прочитанные, и некоторую логику для обработки маркировки существующих уведомлений как «не новых»:

Слайс

const notificationsSlice = createSlice({

name: 'notifications',

initialState: [],

reducers: {

allNotificationsRead(state, action) {

state.forEach(notification => {

notification.read = true

})

}

},

extraReducers(builder) {

builder.addCase(fetchNotifications.fulfilled, (state, action) => {

state.push(...action.payload)

state.forEach(notification => {

// Any notifications we've read are no longer new

notification.isNew = !notification.read

})

// Sort with newest first

state.sort((a, b) => b.date.localeCompare(a.date))

})

}

})

export const { allNotificationsRead } = notificationsSlice.actions

export default notificationsSlice.reducer

Мы хотим помечать эти уведомления как прочитанные при каждом <NotificationsList>рендеринге нашего компонента, либо потому, что мы щелкнули вкладку, чтобы просмотреть уведомления, либо потому, что она уже открыта и мы только что получили несколько дополнительных уведомлений. Мы можем сделать это, отправляя allNotificationsReadкаждый раз повторный рендеринг этого компонента. Чтобы избежать обновления старых данных при обновлении, мы отправим действие в виде useLayoutEffectперехватчика. Мы также хотим добавить дополнительное имя класса ко всем записям списка уведомлений на странице, чтобы выделить их:

функции/уведомления/NotificationsList.js

import React, { useLayoutEffect } from 'react'  
import { useSelector, useDispatch } from 'react-redux'  
import { formatDistanceToNow, parseISO } from 'date-fns'  
import classnames from 'classnames'  
  
import { selectAllUsers } from '../users/usersSlice'  
  
import {  
 selectAllNotifications,  
 allNotificationsRead  
} from './notificationsSlice'  
  
export const NotificationsList = () => {  
 const dispatch = useDispatch()  
 const notifications = useSelector(selectAllNotifications)  
 const users = useSelector(selectAllUsers)  
  
 useLayoutEffect(() => {  
 dispatch(allNotificationsRead())  
 })  
  
 const renderedNotifications = notifications.map(notification => {  
 const date = parseISO(notification.date)  
 const timeAgo = formatDistanceToNow(date)  
 const user = users.find(user => user.id === notification.user) || {  
 name: 'Unknown User'  
 }  
  
 const notificationClassname = classnames('notification', {  
 new: notification.isNew  
 })  
  
 return (  
 <div key={notification.id} className={notificationClassname}>  
 <div>  
 <b>{user.name}</b> {notification.message}  
 </div>  
 <div title={notification.date}>  
 <i>{timeAgo} ago</i>  
 </div>  
 </div>  
 )  
 })  
  
 return (  
 <section className="notificationsList">  
 <h2>Notifications</h2>  
 {renderedNotifications}  
 </section>  
 )  
}

приложение/Navbar.js

// omit imports  
import { useDispatch, useSelector } from 'react-redux'  
  
import {  
 fetchNotifications,  
 selectAllNotifications  
} from '../features/notifications/notificationsSlice'  
  
export const Navbar = () => {  
 const dispatch = useDispatch()  
 const notifications = useSelector(selectAllNotifications)  
 const numUnreadNotifications = notifications.filter(n => !n.read).length  
 // omit component contents  
 let unreadNotificationsBadge  
  
 if (numUnreadNotifications > 0) {  
 unreadNotificationsBadge = (  
 <span className="badge">{numUnreadNotifications}</span>  
 )  
 }  
 return (  
 <nav>  
 // omit component contents  
 <div className="navLinks">  
 <Link to="/">Posts</Link>  
 <Link to="/users">Users</Link>  
 <Link to="/notifications">  
 Notifications {unreadNotificationsBadge}  
 </Link>  
 </div>  
 // omit component contents  
 </nav>  
 )  
}

[Улучшение производительности рендеринга](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#improving-render-performance)

у нас есть пара недостатков в том, когда и как перерисовываются наши компоненты. Давайте рассмотрим эти проблемы и поговорим о некоторых способах улучшения производительности.

"функции/UserPage.js

export const UserPage = ({ match }) => {  
 const { userId } = match.params  
  
 const user = useSelector(state => selectUserById(state, userId))  
  
 const postsForUser = useSelector(state => {  
 const allPosts = selectAllPosts(state)  
 return allPosts.filter(post => post.user === userId) – фильтр возвр новый массив  
 })  
  
 // omit rendering logic  
}

Мы знаем, что он useSelectorбудет выполняться повторно каждый раз, когда отправляется действие, и что он заставляет компонент перерисовываться, если мы возвращаем новое ссылочное значение.

Мы вызываем filter()внутри нашего useSelectorхука, чтобы вернуть только список сообщений, принадлежащих этому пользователю. К сожалению, **это означает, что он useSelector *всегда* возвращает ссылку на новый массив, и поэтому наш компонент будет повторно отображаться после *каждого* действия, даже если данные сообщений не изменились!** .

* [Запоминание функций выбора](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#memoizing-selector-functions)
* state.postsили userIdизменился. Если они *не изменились* , мы хотим вернуть ту же ссылку на отфильтрованный массив, что и в прошлый раз.
* Эта идея называется «мемоизация». Мы хотим сохранить предыдущий набор входных данных и вычисленный результат, и, если входные данные одинаковы, вернуть предыдущий результат вместо его повторного расчета.
* [**Reselect**](https://github.com/reduxjs/reselect)**— это библиотека для создания запоминаемых функций селектора** , специально разработанная для использования с Redux. У него естьcreateSelectorфункция, которая генерирует запоминаемые селекторы, которые пересчитывают результаты только при изменении входных данных. Redux Toolkit[экспортирует createSelectorфункцию](https://redux-toolkit.js.org/api/createSelector) , поэтому она у нас уже есть.
* Давайте создадим новую selectPostsByUserфункцию выбора, используя Reselect, и воспользуемся ею здесь.
* функции/сообщения/postsSlice.js
* import { createSlice, createAsyncThunk, createSelector } from '@reduxjs/toolkit'  
    
  // omit slice logic  
    
  export const selectAllPosts = state => state.posts.posts  
    
  export const selectPostById = (state, postId) =>  
   state.posts.posts.find(post => post.id === postId)  
    
  export const selectPostsByUser = createSelector(  
   [selectAllPosts, (state, userId) => userId],  
   (posts, userId) => posts.filter(post => post.user === userId)  
  )

export const UserPage = ({ match }) => {  
 const { userId } = match.params  
  
 const user = useSelector(state => selectUserById(state, userId))  
  
 const postsForUser = useSelector(state => selectPostsByUser(state, userId))  
  
 // omit rendering logic  
}

* [Исследование списка сообщений](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#investigating-the-posts-list)

Если мы вернемся к нашему <PostsList>и попытаемся нажать кнопку реакции на одном из сообщений во время захвата трассировки профилировщика React, мы увидим, что не только отрисовываются <PostsList>обновленный <PostExcerpt>экземпляр, но и все компоненты <PostExcerpt>:

[**Поведение React по умолчанию заключается в том, что при рендеринге родительского компонента React рекурсивно визуализирует все дочерние компоненты внутри него!**](https://blog.isquaredsoftware.com/2020/05/blogged-answers-a-mostly-complete-guide-to-react-rendering-behavior/) .

Есть несколько способов оптимизировать это поведение в <PostsList>.

Во-первых, мы могли бы обернуть <PostExcerpt>компонент в [React.memo()](https://react.dev/reference/react/memo), что будет гарантировать, что компонент внутри него будет перерисовываться только в том случае, если реквизиты действительно изменились. На самом деле это будет работать очень хорошо — попробуйте и посмотрите, что произойдет:

"функции/сообщения/PostsList.js

let PostExcerpt = ({ post }) => {  
 // omit logic  
}  
  
PostExcerpt = React.memo(PostExcerpt)

другой предложенный вариант сильно сложный, не стану описывать

Последний вариант — найти способ заставить наш редуктор хранить отдельный массив идентификаторов для всех сообщений и изменять этот массив только при добавлении или удалении сообщений, а также выполнять ту же самую перезапись <PostsList>и <PostExcerpt>. Таким образом, <PostsList>повторную визуализацию необходимо выполнять только при изменении массива идентификаторов.

Удобно, что в Redux Toolkit есть createEntityAdapterфункция, которая поможет нам сделать это.

[Нормализация данных](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#normalizing-data)

Вы видели, что большая часть нашей логики ищет элементы по их полю ID. Поскольку мы храним наши данные в массивах, это означает, что нам придется перебирать все элементы массива, array.find()пока не найдем элемент с искомым идентификатором.

На самом деле это не займет много времени, но если бы у нас были массивы с сотнями или тысячами элементов внутри, просмотр всего массива в поисках одного элемента стал бы напрасной тратой усилий. Нам нужен способ поиска отдельного элемента по его идентификатору напрямую, без необходимости проверять все остальные элементы. Этот процесс известен как «нормализация».

**Нормализованная структура состояния**

«Нормализованное состояние» означает, что:

* В нашем штате есть только одна копия каждого конкретного фрагмента данных, поэтому дублирования нет.
* Нормализованные данные хранятся в справочной таблице, где идентификаторы элементов являются ключами, а сами элементы — значениями.
* Также может существовать массив всех идентификаторов для определенного типа элемента.

{

users: {

ids: ["user1", "user2", "user3"],

entities: {

"user1": {id: "user1", firstName, lastName},

"user2": {id: "user2", firstName, lastName},

"user3": {id: "user3", firstName, lastName},

}

}

}

### Управление нормализованным состоянием createEntityAdapter[с помощью](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#managing-normalized-state-with-createentityadapter)

API Redux Toolkit createEntityAdapterпредоставляет стандартизированный способ хранения ваших данных в срезах, беря коллекцию элементов и помещая их в форму { ids: [], entities: {} }. Наряду с этой предопределенной формой состояния он генерирует набор функций-редюсеров и селекторов, которые знают, как работать с этими данными.

* [Обновление фрагмента сообщений](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#updating-the-posts-slice)
* Имея это в виду, давайте обновим наш код postsSliceдля использования createEntityAdapter:
* функции/сообщения/postsSlice.js
* import {  
   createEntityAdapter  
   // omit other imports  
  } from '@reduxjs/toolkit'  
    
  const postsAdapter = createEntityAdapter({  
   sortComparer: (a, b) => b.date.localeCompare(a.date)  
  })  
    
  const initialState = postsAdapter.getInitialState({  
   status: 'idle',  
   error: null  
  })  
    
  // omit thunks  
    
  const postsSlice = createSlice({  
   name: 'posts',  
   initialState,  
   reducers: {  
   reactionAdded(state, action) {  
   const { postId, reaction } = action.payload  
   const existingPost = state.entities[postId]  
   if (existingPost) {  
   existingPost.reactions[reaction]++  
   }  
   },  
   postUpdated(state, action) {  
   const { id, title, content } = action.payload  
   const existingPost = state.entities[id]  
   if (existingPost) {  
   existingPost.title = title  
   existingPost.content = content  
   }  
   }  
   },  
   extraReducers(builder) {  
   // omit other reducers  
    
   builder  
   .addCase(fetchPosts.fulfilled, (state, action) => {  
   state.status = 'succeeded'  
   // Add any fetched posts to the array  
   // Use the `upsertMany` reducer as a mutating update utility  
   postsAdapter.upsertMany(state, action.payload)  
   })  
   // Use the `addOne` reducer for the fulfilled case  
   .addCase(addNewPost.fulfilled, postsAdapter.addOne)  
   }  
  })  
    
  export const { postAdded, postUpdated, reactionAdded } = postsSlice.actions  
    
  export default postsSlice.reducer  
    
  // Export the customized selectors for this adapter using `getSelectors`  
  export const {  
   selectAll: selectAllPosts,  
   selectById: selectPostById,  
   selectIds: selectPostIds  
   // Pass in a selector that returns the posts slice of state  
  } = postsAdapter.getSelectors(state => state.posts)  
    
  export const selectPostsByUser = createSelector(  
   [selectAllPosts, (state, userId) => userId],  
   (posts, userId) => posts.filter(post => post.user === userId)  
  )
* [Оптимизация списка сообщений](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#optimizing-the-posts-list)

Теперь, когда наш фрагмент сообщений использует createEntityAdapter, мы можем обновить его <PostsList>, чтобы оптимизировать поведение рендеринга.

// omit other imports  
  
import {  
 selectAllPosts,  
 fetchPosts,  
 selectPostIds,  
 selectPostById  
} from './postsSlice'  
  
let PostExcerpt = ({ postId }) => {  
 const post = useSelector(state => selectPostById(state, postId))  
 // omit rendering logic  
}  
  
export const PostsList = () => {  
 const dispatch = useDispatch()  
 const orderedPostIds = useSelector(selectPostIds)  
  
 // omit other selections and effects  
  
 if (postStatus === 'loading') {  
 content = <Spinner text="Loading..." />  
 } else if (postStatus === 'succeeded') {  
 content = orderedPostIds.map(postId => (  
 <PostExcerpt key={postId} postId={postId} />  
 ))  
 } else if (postStatus === 'error') {  
 content = <div>{error}</div>  
 }  
  
 // omit other rendering  
}

[Преобразование других фрагментов](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#converting-other-slices)

функции/пользователи/usersSlice.js

import {  
 createSlice,  
 createAsyncThunk,  
 createEntityAdapter  
} from '@reduxjs/toolkit'  
import { client } from '../../api/client'  
  
const usersAdapter = createEntityAdapter()  
  
const initialState = usersAdapter.getInitialState()  
  
export const fetchUsers = createAsyncThunk('users/fetchUsers', async () => {  
 const response = await client.get('/fakeApi/users')  
 return response.users  
})  
  
const usersSlice = createSlice({  
 name: 'users',  
 initialState,  
 reducers: {},  
 extraReducers(builder) {  
 builder.addCase(fetchUsers.fulfilled, usersAdapter.setAll)  
 }  
})  
  
export default usersSlice.reducer  
  
export const { selectAll: selectAllUsers, selectById: selectUserById } =  
 usersAdapter.getSelectors(state => state.users)

* [Преобразование фрагмента уведомлений](https://redux.js.org/tutorials/essentials/part-6-performance-normalization#converting-the-notifications-slice)

import {

createSlice,

createAsyncThunk,

createEntityAdapter

} from '@reduxjs/toolkit'

import { client } from '../../api/client'

const notificationsAdapter = createEntityAdapter({

sortComparer: (a, b) => b.date.localeCompare(a.date)

})

// omit fetchNotifications thunk

const notificationsSlice = createSlice({

name: 'notifications',

initialState: notificationsAdapter.getInitialState(),

reducers: {

allNotificationsRead(state, action) {

Object.values(state.entities).forEach(notification => {

notification.read = true

})

}

},

extraReducers(builder) {

builder.addCase(fetchNotifications.fulfilled, (state, action) => {

notificationsAdapter.upsertMany(state, action.payload)

Object.values(state.entities).forEach(notification => {

// Any notifications we've read are no longer new

notification.isNew = !notification.read

})

})

}

})

export const { allNotificationsRead } = notificationsSlice.actions

export default notificationsSlice.reducer

export const { selectAll: selectAllNotifications } =

notificationsAdapter.getSelectors(state => state.notifications)

SUMMARY

* **Memoized selector functions can be used to optimize performance**
  + Redux Toolkit re-exports the createSelector function from Reselect, which generates memoized selectors
  + Memoized selectors will only recalculate the results if the input selectors return new values
  + Memoization can skip expensive calculations, and ensure the same result references are returned
* **There are multiple patterns you can use to optimize React component rendering with Redux**
  + Avoid creating new object/array references inside of useSelector - those will cause unnecessary re-renders
  + Memoized selector functions can be passed to useSelector to optimize rendering
  + useSelector can accept an alternate comparison function like shallowEqual instead of reference equality
  + Components can be wrapped in React.memo() to only re-render if their props change
  + List rendering can be optimized by having list parent components read just an array of item IDs, passing the IDs to list item children, and retrieving items by ID in the children
* **Normalized state structure is a recommended approach for storing items**
  + "Normalization" means no duplication of data, and keeping items stored in a lookup table by item ID
  + Normalized state shape usually looks like {ids: [], entities: {}}
* **Redux Toolkit's createEntityAdapter API helps manage normalized data in a slice**
  + Item IDs can be kept in sorted order by passing in a sortComparer option
  + The adapter object includes:
    - adapter.getInitialState, which can accept additional state fields like loading state
    - Prebuilt reducers for common cases, like setAll, addMany, upsertOne, and removeMany
    - adapter.getSelectors, which generates selectors like selectAll and selectById