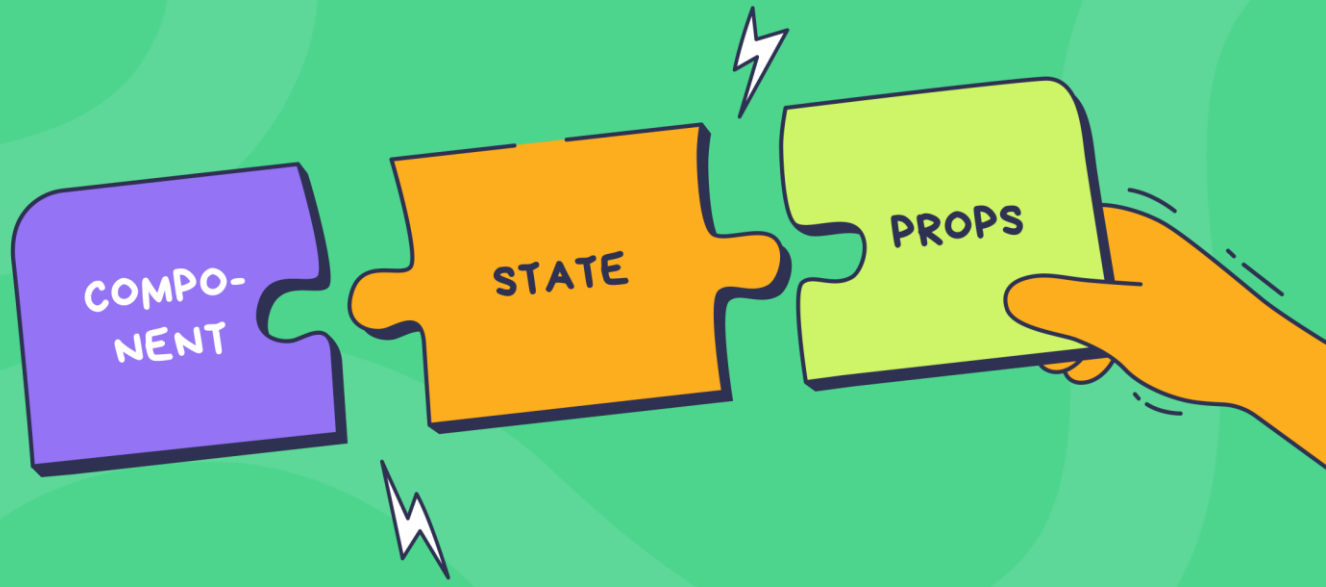
JAVARUSH

Путь к карьере Frontend Fullstack разработчика

MODULE 3. React
Уровень 8 Введение в React Query — кэширование данных и оптимизация запросов



JAVARUSH

Введение

Мы уже умеем управлять клиентским состоянием UI с помощью useState, useReducer и useContext. Но большинство приложений взаимодействуют с сервером для получения и изменения данных. **Это серверное состояние.**

Проблемы "ручного" управления серверным состоянием (через useEffect + useState или Redux):

- Кэширование: как эффективно хранить полученные данные, чтобы не запрашивать их снова и снова?
- Инвалидация кеша: как обновлять кеш, когда данные на сервере изменились?
- Состояние запроса: отслеживание флагов isLoading, isError, хранение объекта error.
- Фоновое обновление: обновление данных без блокировки UI.
- Много шаблонного кода для каждой сущности API.

Для решения этих (и многих других) проблем существуют специализированные библиотеки.

JAVARUSH

Что такое React Query

React Query (TanStack Query) – это мощная библиотека для **управления серверным состоянием** в React-приложениях.

Это НЕ замена Redux, Zustand и т.п. для управления клиентским состоянием UI. React Query **дополняет** их, специализируясь на взаимодействии с асинхронными данными, получаемыми с сервера.

Ключевые возможности "из коробки":

- Запрос данных (Fetching) и их автоматическое обновление.
- Кэширование с гибкой настройкой времени жизни.
- Управление состоянием запроса.
- Мутации (POST, PUT, DELETE) с автоматической инвалидацией кеша.
- Пагинация и бесконечная прокрутка.
- Оптимистичные обновления.
- Инструменты разработчика (React Query Devtools).

JAVARUSH

Настройка React Query

QueryClient хранит кеш и управляет запросами.
QueryClientProvider делает его доступным в приложении.
Это делается в корневом файле (например, main.tsx).

Логика: создаем клиент -> Оборачиваем приложение в Провайдер с этим клиентом.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
// 1. Импортируем QueryClient и QueryClientProvider
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

// 2. Создаем экземпляр QueryClient
const queryClient = new QueryClient({
   // defaultOptions: { queries: { staleTime: ... } }
   // Глобальные опции (позже)
});

ReactDOM.createRoot(document.getElementById('root')!).render(
   <React.StrictMode>
       {/* 3. Оборачиваем приложение в QueryClientProvider */}
       <QueryClientProvider client={queryClient}>
           <App />
           {/* (Опционально) <ReactQueryDevtools /> */}
       </QueryClientProvider>
   </React.StrictMode>
);
```

JAVARUSH

Хук useQuery

useQuery – это основной хук React Query для **запроса (fetch), кеширования и управления состоянием асинхронных данных.**

queryKey (Обязательный): массив, уникально идентифицирующий этот запрос. Используется React Query для кеширования данных. Если ключ меняется, запрос выполняется заново.
queryFn (Обязательный): асинхронная функция, которая выполняет фактический запрос данных и возвращает Promise с результатом (или выбрасывает ошибку).

```
const {
   data,          // TData | undefined: Успешно загруженные данные
   error,         // TError | null: Объект ошибки
   isLoading,     // boolean: Идет первоначальная загрузка (нет кеша)
   isFetching,    // boolean: Идет запрос (даже если есть кеш, для фонового обновления)
   isSuccess,     // boolean: Запрос успешно завершен и есть данные
   isError,       // boolean: Произошла ошибка
   refetch,       // () => void: Функция для принудительного повторного запроса
   // ... и другие полезные свойства
} = useQuery({
   queryKey: ['уникальный', 'ключ', 'запроса'], // Array - для кеширования
   // Должна вернуть Promise
   queryFn: async () => { /* ... ваша функция для fetch данных ... */ return data; },
});
```

Типизация запросов и ответов

useQuery является generic-хуком, что позволяет строго типизировать его результаты.

Сигнатура (основные generic-типы):
useQuery<TQueryFnData, TError, TData, TQueryKey>

- TQueryFnData (самый важный) - тип данных, который возвращает ваша queryFn (например, Promise<Post[]> -> TQueryFnData это Post[]). Это определяет тип data при успехе.
- TError - тип объекта ошибки (по умолчанию Error или unknown).
- TData (редко используется) - тип данных в data после опциональной трансформации через select.
- TQueryKey - тип для queryKey.

Всегда явно указывайте как минимум TQueryFnData (тип результата queryFn) и TError.

JAVARUSH

Управление кэшем

React Query активно **кеширует** результаты запросов. Два ключевых параметра управляют этим

staleTime: number (в миллисекундах)

- Время, в течение которого данные считаются "свежими" (fresh).
- Пока данные "свежие", useQuery не будет выполнять повторный запрос к сети при монтировании компонента, фокусе окна и т.д., а вернет данные из кеша немедленно.
- По умолчанию: 0. Это значит, что данные сразу считаются "устаревшими" (stale). При "устаревших" данных, useQuery сначала возвращает их из кеша (если есть), а затем в фоне запускает queryFn для обновления.

cacheTime: number (в миллисекундах)

- Время, в течение которого неактивные (не используемые ни одним активным useQuery с таким queryKey) данные хранятся в кеше перед тем, как будут удалены сборщиком мусора.
- По умолчанию: 5 * 60 * 1000 (5 минут).

Логика:

- Запрос -> Данные в кеше (fresh).
- Прошло staleTime -> Данные в кеше (stale).
- Компонент монтируется -> useQuery видит stale данные -> Возвращает их из кеша + запускает фоновый refetch.
- Прошло cacheTime после того, как не осталось активных useQuery -> Данные удаляются из кеша.

StaleTime vs CacheTime

Данные сначала "свежие", затем "устаревшие" (но еще в кеше), и только потом удаляются из кеша, если не используются.

JAVARUSH

Мутации

Для операций, **изменяющих данные на сервере** (POST, PUT, PATCH, DELETE), используется хук useMutation.

```
const {
   mutate,        // (variables, options?) => void: Функция для запуска мутации
   mutateAsync,   // (variables, options?) => Promise<TData>: То же, но возвращает Promise
   data,          // TData | undefined: Данные, возвращенные успешной mutationFn
   error,         // TError | null: Объект ошибки
   isLoading,     // boolean: Мутация в процессе выполнения
   isSuccess,     // boolean: Мутация успешно завершена
   isError,       // boolean: Произошла ошибка при мутации
   reset,         // () => void: Сбросить состояние мутации (ошибки, данные)
   // ... и другие
} = useMutation({
   mutationFn: async (variables: TVariables) => { /* ... ваша асинхронная функция
(fetch/axios) для изменения данных ... */ return result; },
   // ... другие опции (onSuccess, onError, onSettled, retry)
});
```

mutationFn (Обязательный): асинхронная функция, которая выполняет операцию изменения и возвращает Promise с результатом (или ошибкой). Принимает один аргумент – variables (данные для мутации).
mutate(variables): вызывает mutationFn с переданными variables.
Коллбэки onSuccess, onError, onSettled позволяют реагировать на результат мутации.

JAVARUSH

Типизация мутаций

Хук useMutation также является generic и позволяет строго типизировать данные.

Сигнатура (основные generic-типы): useMutation<TData, TError, TVariables, TContext>

- TData - тип данных, который возвращает ваша mutationFn при успехе (и который будет в data хука).
- TError - тип объекта ошибки, который может выбросить mutationFn (и который будет в error хука). По умолчанию Error или unknown.
- TVariables - тип объекта переменных, который принимает ваша mutationFn (и который передается в mutate(variables)).
- TContext (редко используется) - тип для контекста оптимистичных обновлений.

Явно типизировать как минимум TData, TError, TVariables.

JAVARUSH

Обновление кеша

После успешной мутации кешированные данные, связанные с этим ресурсом, становятся неактуальными.

Способ 1: Инвалидация кеша (queryClient.invalidateQueries)

Принцип: сообщить React Query, что данные с определенным queryKey "устарели".
Действие: если есть активные useQuery с этим ключом, они будут автоматически перезапрошены (refetched) с сервера.
Плюсы: просто, надежно, гарантирует свежие данные с сервера.

```
// В onSuccess мутации

// Инвалидировать все списки постов
queryClient.invalidateQueries({ queryKey: ['posts'] });
// Инвалидировать конкретный пост
queryClient.invalidateQueries({ queryKey: ['posts', postId] });
// Можно использовать предикат: queryClient.invalidateQueries({ predicate: query => ... })
```

JAVARUSH

Обновление кеша

Способ 2: Прямое обновление кеша (queryClient.setQueryData)

Принцип: вручную обновить данные в кеше для определенного queryKey без повторного запроса к серверу.
Когда полезно: если API мутации возвращает полный обновленный объект или список. Можно сразу поместить его в кеш, экономя запрос.

```
// В onSuccess мутации, где `newOrUpdatedPost` - данные от сервера

// Обновить кеш для одного поста
queryClient.setQueryData(['posts', newOrUpdatedPost.id], newOrUpdatedPost);

queryClient.setQueryData(['posts'], (oldData: Post[] | undefined) => { // Обновить список
   if (!oldData) return [newOrUpdatedPost];
   return oldData.map(post => post.id === newOrUpdatedPost.id ? newOrUpdatedPost : post);
   // Или добавить новый: return [...oldData, newOrUpdatedPost];
});
```

JAVARUSH

Refetch

Хук useQuery возвращает функцию refetch().

Вы можете вызвать refetch() в любой момент (например, по клику на кнопку "Обновить"), чтобы принудительно перезапросить данные для этого queryKey.

```
const { data, refetch, isFetching } = useQuery(...);
// <button onClick={() => refetch()}
disabled={isFetching}>Обновить</button>
```

JAVARUSH

Автоматическое обновление (Refetching)

React Query имеет несколько встроенных стратегий для поддержания данных "свежими" без явного вмешательства:

- refetchOnMount: boolean | "always" (по умолч. true) - перезапросить данные при монтировании компонента, если они "устаревшие" (stale).
- refetchOnWindowFocus: boolean | "always" (по умолч. true) - перезапросить данные, когда окно браузера снова получает фокус, если они "устаревшие".
- refetchOnReconnect: boolean | "always" (по умолч. true): - перезапросить данные при восстановлении сетевого соединения, если они "устаревшие".
- refetchInterval: number | false (по умолч. false): - периодически перезапрашивать данные через указанный интервал в миллисекундах.
- refetchIntervalInBackground: boolean - позволяет refetchInterval работать, даже если вкладка браузера неактивна.

Эти опции можно настроить глобально в QueryClient или индивидуально для каждого useQuery.

JAVARUSH

React Query Devtools

Для отладки и инспектирования состояния React Query существует официальный инструмент – React Query Devtools.

```
# Установка
npm install @tanstack/react-query-devtools
```

Возможности Devtools:

- Просмотр всех активных запросов (useQuery).
- Состояние каждого запроса: fresh, fetching, stale, inactive.
- Инспектирование кешированных данных для каждого queryKey.
- Ручной запуск refetch, инвалидации кеша (invalidate), сброса кеша (reset).
- Отслеживание истории запросов.

JAVARUSH

Домашнее задание

Уровень 8 Введение в React Query — кэширование данных и оптимизация запросов

MODULE 3. REACT