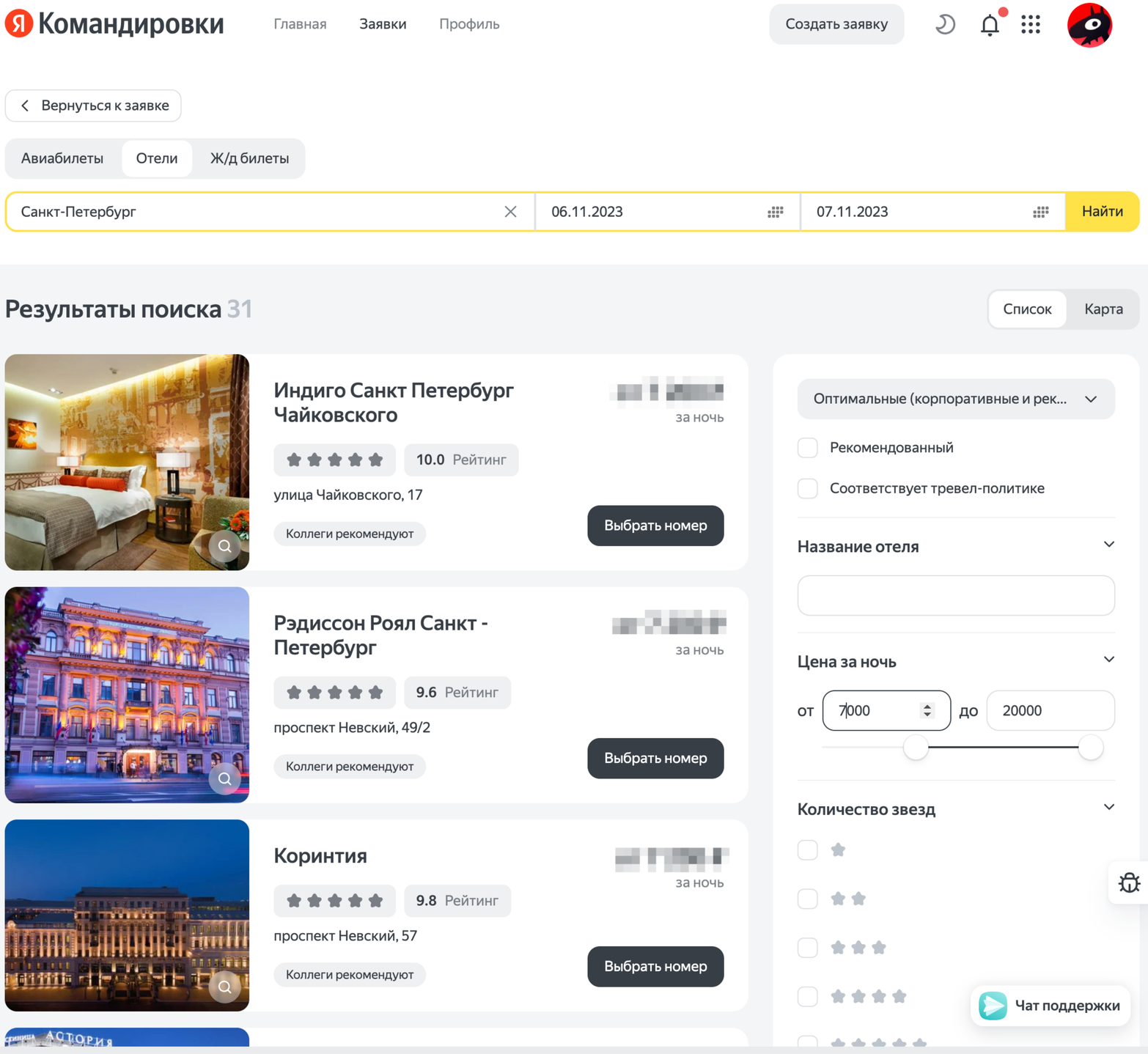
Яндекс Командировки

# Как Яндекс Командировки уже три года живут без state-менеджера

Баранов Валера, руководитель фронтенд-разработки Яндекс Командировок

# О себе

- Последние 4 года в Яндексе
- Разработал и запустил фронтенд Яндекс Командировок

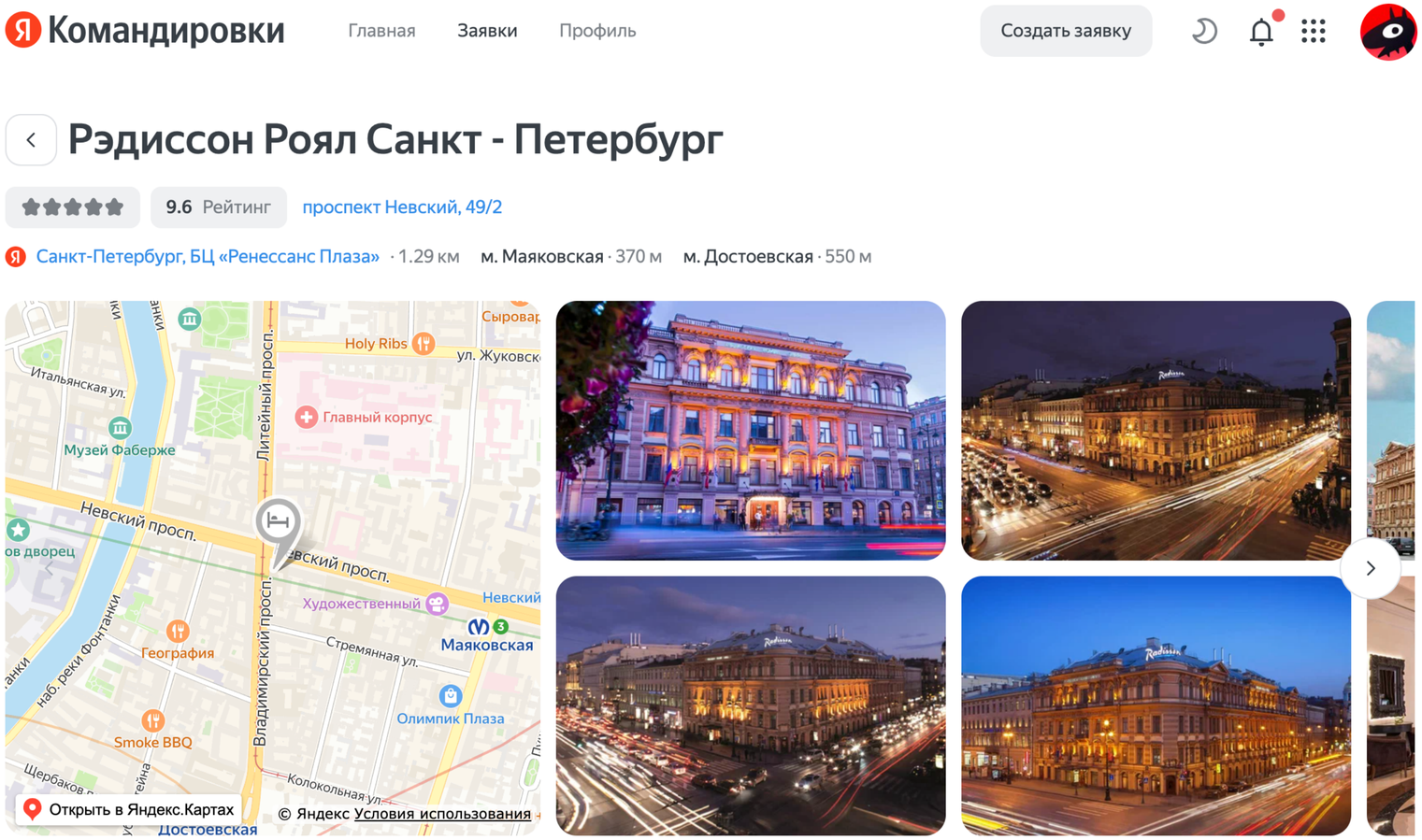



HR Tech

# О сервисе

- Командировки: авиабилеты, жд, отели
- React + Typescript, статика на S3
- 3 года разработки 3+ фронтендера
- 200k+ строк кода
- Rest API

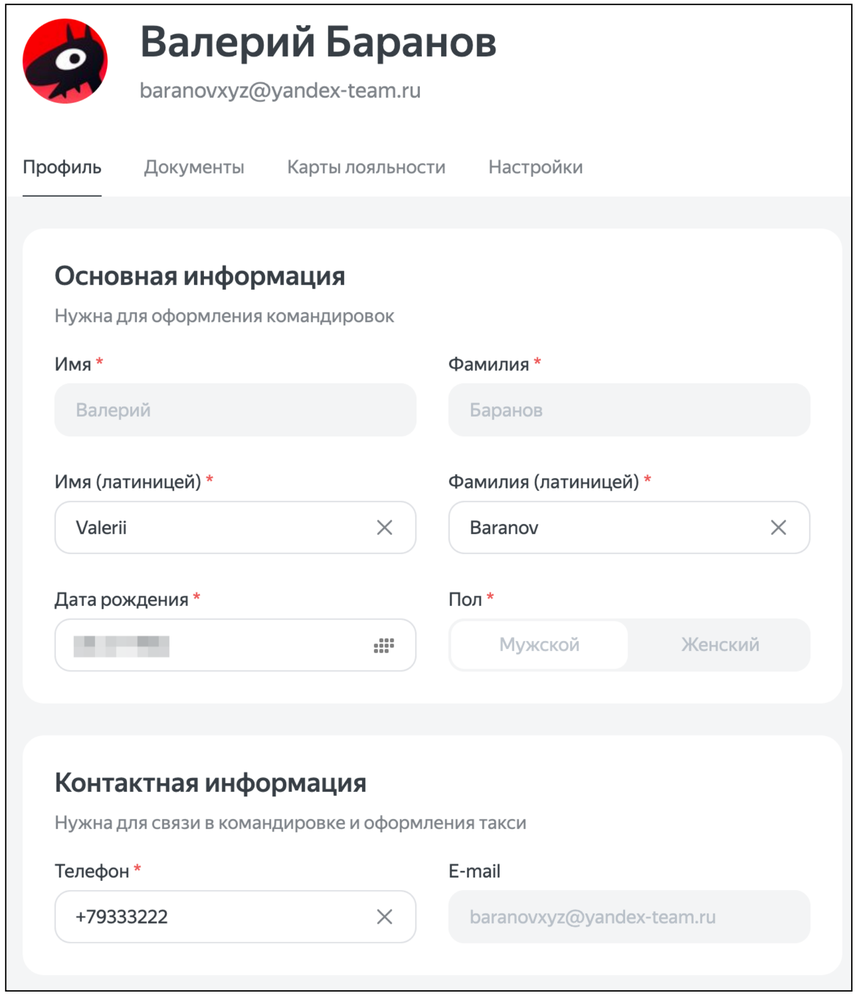
HR Tech



```
const UI = fn(state)
```

# Данные для страницы профиля

https://trip.yandex-team.ru/profile/001




```
const state = {
  config: { … },
  router: { … },
  meta: { … },
  profile: { … },
}
```

# Данные для страницы профиля

- Зависит от окружения
- Зашивается при сборке
- Доступен глобально
- Синхронный доступ

```
const state = {
  config: {
    apiHost: 'trip.yandex-team.ru'
    ...
  },
  router: { ... },
  meta: { ... },
  profile: { ... }
}
```

# Данные для страницы профиля

- Определяет страницу
- Содержит параметры
- Доступен глобально *
- Синхронный доступ

```
const state = {
  config: { … },
  router: {
    pathname: '/profile/001',
    params: { personId: '001' },
    searchParams: {},
  },
  meta: { … },
  profile: { … }
}
```

# Данные для страницы профиля

```
const state = {
  config: { … },
  router: { … },
  meta: {
    uid: '001',
    isCoordinator: true,
    …
  },
  profile: { … }
}
```

/api/meta

- Асинхронный доступ
- Хранятся на сервере
- Могут измениться
- Нужны глобально

# Данные для страницы профиля

```
const state = {
  config: { … },
  router: { … },
  meta: { … },
  profile: {
    name: ‘Valera’,
    email: valera@ya.ru,
    …
  }
}
```

/api/persons/001/details

- Асинхронный доступ
- Хранятся на сервере
- Могут измениться
- Нужны на странице профиля *

# Данные для страницы профиля

Данные клиента:
- Хранятся в памяти
- Синхронный доступ
- Доступ у приложения
- Всегда актуальны

```
const state = {
  config: { ... },
  router: { ... },
  meta: { ... },
  profile: { ... }
}
```

Данные сервера:
- Хранятся на сервере
- Асинхронный доступ
- Совместный доступ
- Снапшот

# Как было у нас

```
1  import { createAction, handleActions } from 'redux-actions';  4.4K (gzipped: 1.8K)
2  import { ofType, combineEpics } from 'redux-observable';  8.4K (gzipped: 2.7K)
3  import { map, switchMap } from 'rxjs/operators';  13.9K (gzipped: 3.9K)
4
5  import { Store } from 'ducks/store';
6  import { SwaggerApi, Purpose } from 'services/SwaggerApi';
7  import { fromApiCall } from 'utils/fromApiCall';
8  import { getValueOrNull, initial, RemoteData } from 'utils/Loadable';
9
10 const REQUEST = 'trip/purposes/REQUEST';
11 const PURPOSES =
12
13 export const requ
14 const purposes = (
15
16 export type Purpos
17
18 export const redu
19     [PURPOSES]: (
20 }, initial());
21
22 const apiPurposes = new SwaggerApi().purpose_list_api_purposes__get;
23
24 export const epic = combineEpics(action$ => action$.pipe(
25     ofType(REQUEST),
26     switchMap(() => fromApiCall(apiPurposes)().pipe(map(purposes))),
27 ));
28
29 export const selectPurposes = (state: Store) => state.purposes;
30 export const selectPurposesData = (state: Store) => getValueOrNull(state.purposes);
```


Почему вы вообще используете createAction, когда могли бы всё упростить?

Что это за бессмысленные константы REQUEST и PURPOSES?

Неужели вам показалось, что такой reducer удобно читать и поддерживать?

getValueOrNull – и это, по вашему, хороший способ обработки данных?

Зачем нужен этот combineEpics, когда есть масса других инструментов?

"Ducks"? Вероятно, у вас ещё и "frogs" где-то скрыты?

ofType, switchMap – сколько ещё операторов вы собрались тут накидать?

Что это за кривой fromApiCall? Вы думаете, такой способ делает ваш код лучше?

На что вы рассчитывали, используя handleActions с таким подходом?

Каким бессмысленным образом selectPurposes отличается от selectPurposesData?

# Как стало

```
import { createUseQueryHook } from './createUseQueryHook';

export const usePurposes = createUseQueryHook(
    'purpose_list',
);
```

# Использовать state-менеджер для серверных данных?

# Опыт Яндекс Командировок

1 | React Query

2 | Какие стратегии кеширования в Командировках

3 | Мутирующие запросы в Командировках

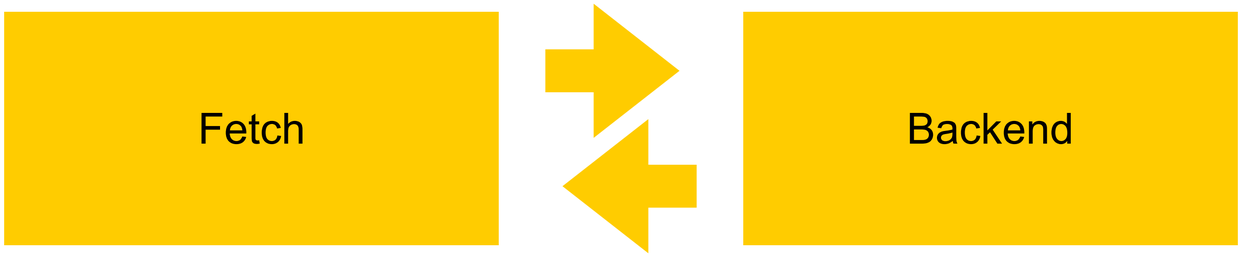
4 | А как в Командировках сделано…

5 | Специфика Командировок: обертки и кастомные хуки

6 | Подводим итоги и делаем выводы

# Как работает React Query

# Как мы обычно загружаем данные



# Как мы загружаем данные в React Query

# Пример работы

```
<TripInfo />
export const TripInfo = ({ tripId: id }) => {
  const trip = useQuery(['trip', id], getTrip(id));

  if (status === 'loading') return <Spinner />;

  return <div>{...trip.data}<div>;
}
```

```
Debugger

trip: {
  status: 'success',
  data: { ... }
}
```

```
const getTrip = id => fetch(`/api/trips/${id}`);
```

Mounted:

Stale-while-revalidate

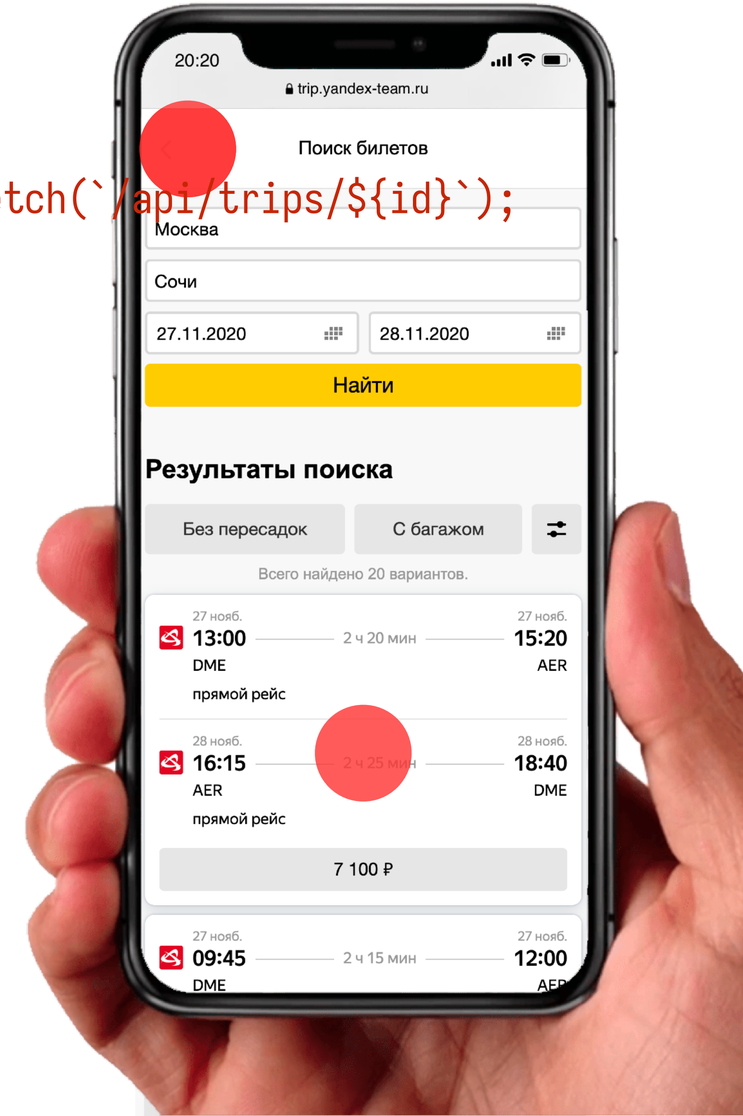
QueryCache (garbage collected)

5 минут

Дедупликация запросов

Structural sharing

Backend

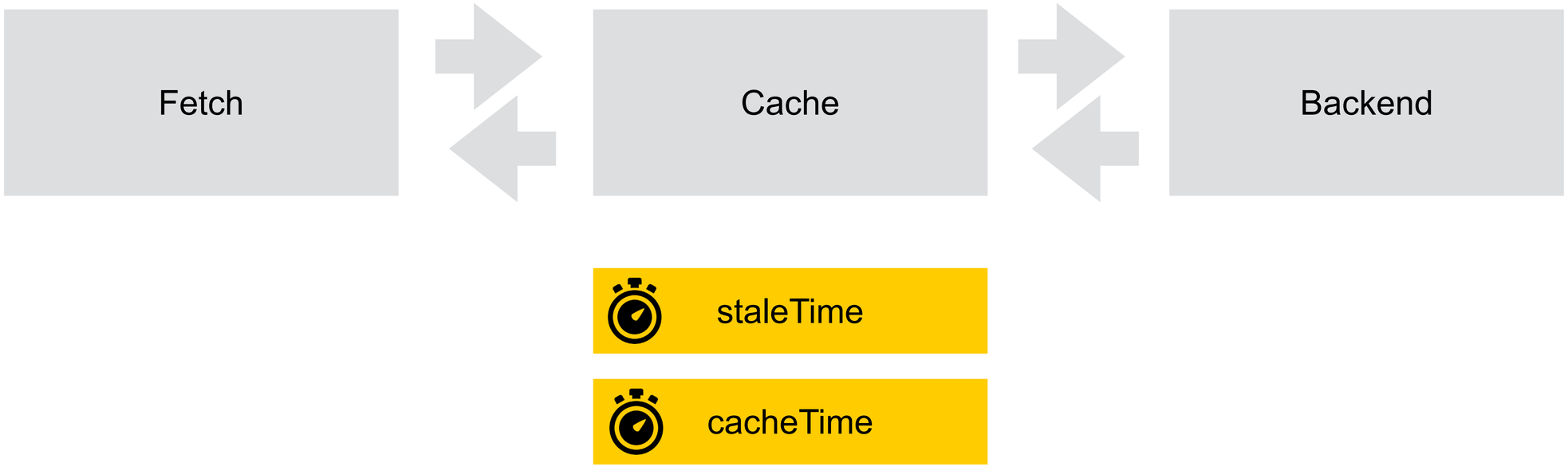


# React Query

- Нет бойлерплейта
- Кеширование
- Дедупликация запросов
- Нет лишних ререндеров
- Декларативно

# Какие стратегии кеширования в Командировках

# Настройки кеширования



# staleTime ⏱

как долго загруженные данные считаются «свежими»

- Частота обновления данных
- Насколько важна актуальность данных
- Динамическая настройка (состояние сети)
- Нагрузка на сервер при низких значениях
- Информировать пользователя при высоких значениях

# cacheTime ⏱

как долго загруженные данные хранятся в кэше

- Даже если данные "устарели", они будут храниться в кэше
- Оптимизация памяти при больших ответах
- Могут ли данные использоваться повторно
- Бесконечное кеширование (справочники)
- Динамическая настройка (тип запроса)
- Можно удалять из кэша вручную

# Stale-While-Revalidate (SWR)
# 80%

Моментально показываем данные из кэша,
в фоне обновляем.

Для данных, актуальность которых может быстро измениться:
- информация об услуге (авиаперелет, жд, отель)
- информация о командировке

```
useQuery([‘trip’, tripId], fetchTrip, {
    staleTime: 0, // всегда в фоне обновляем данные
    cacheTime: 1000 * 60 * 15 // для защиты от утечек памяти
})
```

# Cache-First (Network Fallback)
# 20%

Моментально показываем данные из кэша,
загружаем если нет кэша

Для данных, которые вряд ли изменятся за время сессии:
- справочники (цели командировки)
- саджест по городам
- данные о поиске (не изменяемые у поставщика)

```
useQuery('purposes', fetchPurposes, {
    staleTime: Infinity,
    cacheTime: Infinity | 1000 * 60 * 15,
})
```

# Network-Only

Всегда запрос в api за актуальными данными, игнорируем кэш.

В Командировках не используем: нет настолько критичных данных

```
useQuery('accountBalance', fetchAccountBalance, {
    cacheTime: 0,
    staleTime: 0,
})
```

# Cache-Only

Используем только кэш, не делаем сетевых запросов.

Сценарии использования:
- глобальные данные
- оффлайн режим (если сохранять данные запросов в IndexedDB и восстанавливать при повторном открытии приложения)

В Командировках не используем: нет такой потребности

```
queryClient.setQueryData(‘some’, data);
queryClient.getQueryData(‘some’);
```

# Мутирующие запросы для методов POST, PUT, PATCH и DELETE

# useMutation для POST, PUT, PATCH и DELETE

- Можно использовать колбеки onSuccess, onError, onSettled
- Надо инвалидировать часть данных

```
useMutation(mutationFn, {
    onSuccess: () => {
        queryClient.invalidateQueries(['trips', 1])
    },
});
```

# А как в Командировках сделано…

# Частые вопросы

- Динамические запросы
- Зависимые запросы
- Условные запросы
- Предзагрузка данных
- Оптимистичные обновления

# enabled (useQuery)

false предотвратит автоматический запуск запроса

- Динамические запросы
- Зависимые запросы
- Условные запросы

# Динамические запросы

Когда запрос зависит от другой переменной

```
const responseAvia = createUseQueryHook('get_avia_search_status', {
    enabled: type === 'Avia',
    refetchInterval,
})(searchId, { person_id, trip_id });
const responseHotel = createUseQueryHook('get_hotel_search_status', {
    enabled: type === 'Hotel',
    refetchInterval,
})(searchId, { person_id, trip_id });
const responseRail = createUseQueryHook('get_rail_search_status', {
    enabled: type === 'Rail',
    refetchInterval,
})(searchId, { person_id, trip_id });
```

# Зависимые запросы

Когда второй запрос зависит от данных из первого

```
const { data: service, status } = useQuery(..., fetchServiceFn);

const { data: providerService } = useQuery(..., fetchProviderServiceFn, {
    enabled: status === 'success',
});
```

… или водопадная загрузка компонентов

# Условные запросы

Если запрос зависит от состояния булевой переменной

```
const members = useQuery([groupMembers, 1], fetchGroupMembers, {
    enabled: isGroupTrip,
});
```

# Предзагрузка данных в Командировках

на примере мобилки с WebView

Табы:

〉 Главная: список активных командировок

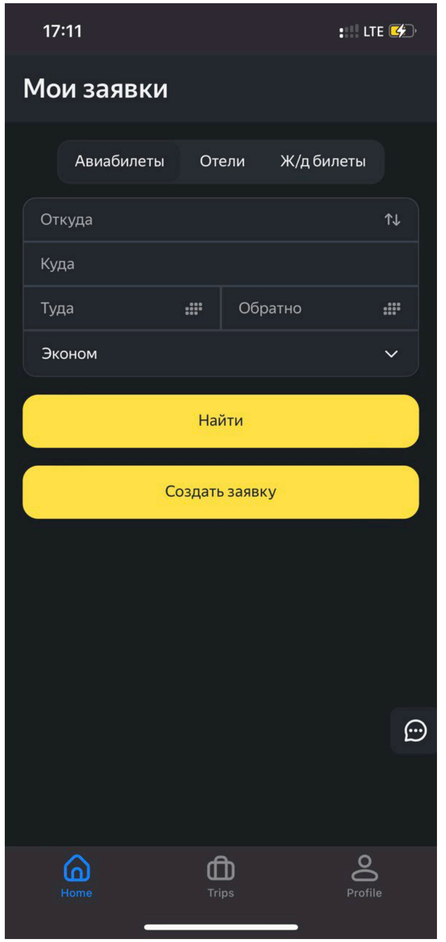
〉 Командировки: 10 последних командировок

〉 Профиль: детали пользователя

```
queryClient.prefetchQuery(['activeTrips'], fetchActiveTrips, {
    staleTime: 0,
    cacheTime: Infinity,
});
```

Результат:

〉 Табы переключаются моментально (или быстрее)



# Optimistic updates

# Optimistic updates

- Действия бинарного типа (лайк, добавить в избранное)
- Действия не должны быть связаны с другими частями интерфейса
- Быстрый отклик API
- Успешность выполнения метода API близка к 100%

# Optimistic updates (проблемы)

〉 Быстрый ответ может сбить пользователя с толку (важные изменения)
〉 Усложнение разработки / ухудшение DX

# Optimistic updates в Командировках

Не используем.

Кейсы:
〉Простой: отображать командировки в профиле (важный тоггл)
〉Сложный: отправить услугу на оформление (логика на беке)

Декларативность снижается, рассуждать что и как происходит в коде становится сложнее.

# Специфика Командировок

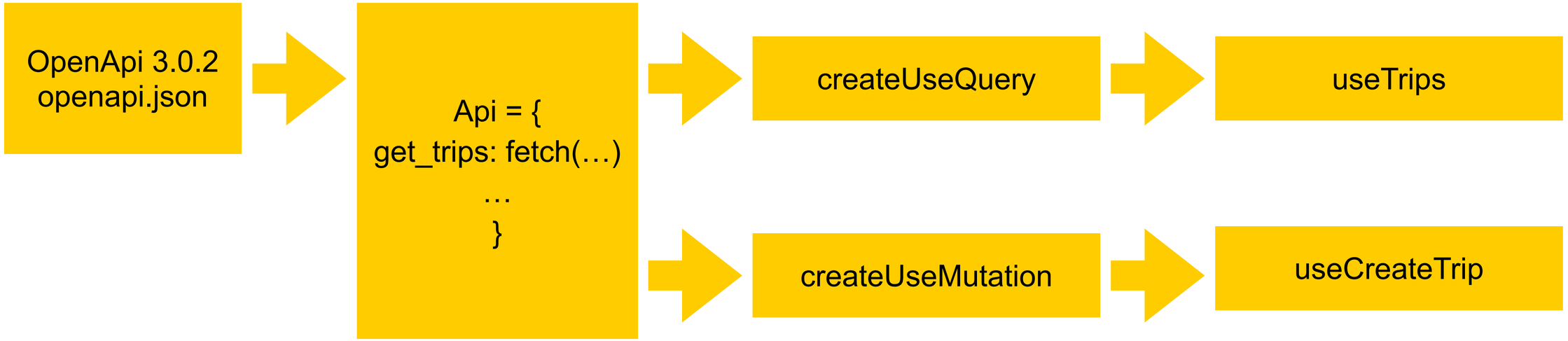
# RemoteData<A, E>

```ts
type RemoteData<A, E> =
    | { status: 'initial' }
    | { status: 'loading' }
    | { status: 'success', value: A }
    | { status: 'failure', error: E }
    ;

type TripState = RemoteData<{ trip_id: number }, string>;
```

# <Remote data={RemoteData} render={FC} /> HOC withRemote

```tsx
export function Remote<T>(props: RemoteProps<T>): JSX.Element | null {
    const { data, render, children } = props;
    const Component = render || children;

    switch (data.status) {
        case 'success':
            return <Component data={data.value} />;
        case 'failure':
            return <Error />;
        default:
            return <Spinner />;
    }
}
```

# Серверные данные в Командировках



# Кастомный генератор на базе OpenApi 3

```
export class SwaggerApi extends SwaggerApiTemp {
  service_create(
      payload: Types.ServiceCreate,
      options?: RequestInit
  ) {
      return apiFetchTypedErrors<
          ResponseOnStatus<201, Types.ServiceId> |
          ResponseOnStatus<422, Types.HTTPValidationError>,
          Types.ServiceCreate,
          null
      >('/api/services/', null, [201, 422], 'POST', payload, options);
  },
  ...
}
```

# createUseQuery

вернет данные в формате RemoteData

```
export function createUseQuery<
    K extends keyof Api,
    Args extends Parameters<Api[K]>,
>(
    key: K,
    queryConfig?: UseQueryOptions,
) {
    return (...args: Args): RemoteData<TSuccess, FetchError> =>
        ...
}
```

# usePersonDocuments

```
import { createUseQuery } from ‘./createUseQuery';

export const usePersonDocuments = createUseQuery(
    ‘person_document_list',
    { staleTime: 0, cacheTime: Infinity },
);

export const TabDocuments: FC<TabDocumentsProps> = props => {
    const { personId } = props;
    const documents = usePersonDocuments(personId);

    return <Documents documents={documents} personId={personId} />;
};
```

# createUseMutation

RemoteData и декларативная инвалидация

```
export function createUseMutation<
    K extends keyof Api,
    Args extends Parameters<Api[K]>,
>(
    key: K,
    invalidateQueriesFn: (...args: Args) => unknown[][],
) {
    return ({ onSuccess, onError, options }) =>
        ...
}
```

# Декларативная инвалидация данных

〉 зависимости описаны в одном файле
〉 декларативно указано, что изменится

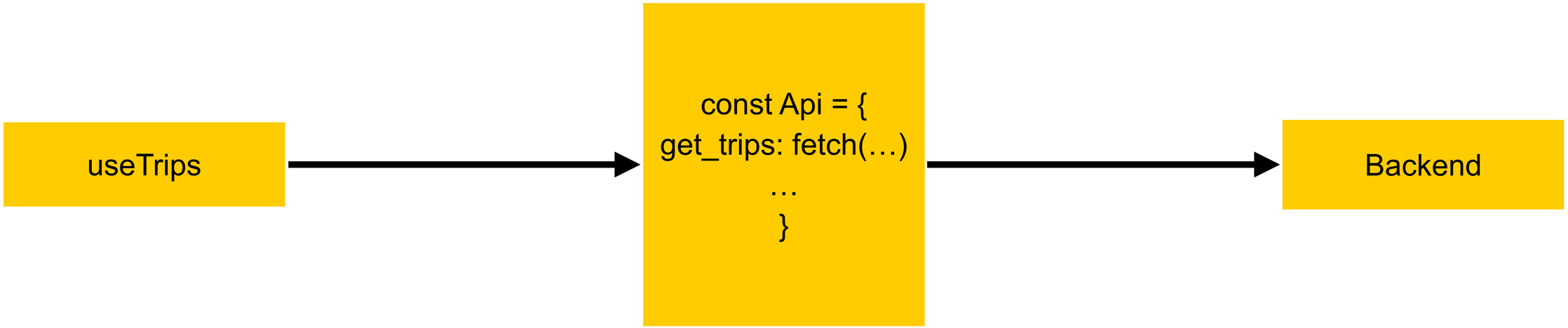
```
import { createUseMutation } from './createUseMutation';

export const useEditGroup = createUseMutation(
    'edit_group',
    groupId => [
        ['get_group_details', groupId],
    ],
);
```

# Данные роутера на уровне страниц

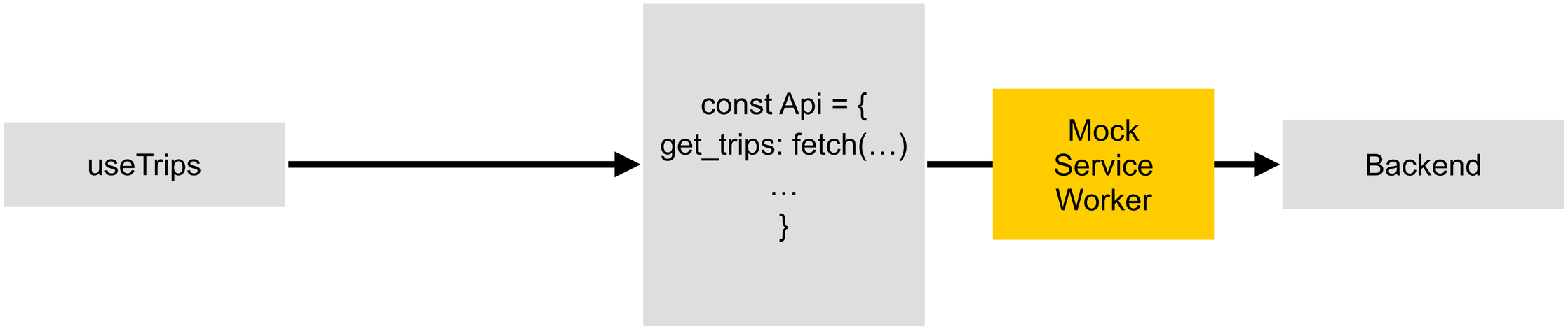
- Декодер для каждого роута обрабатывает location и предоставляет типизированные данные в компонент страницы
- В другие компоненты передаем через пропсы
- Обеспечивает безопасность использования данных из роута

```
export const PageProfile: Page<RouteProfile> = props => {
    const {
        params: { personId },
        search: { tab = 'profile' },
    } = props;
    ...
```

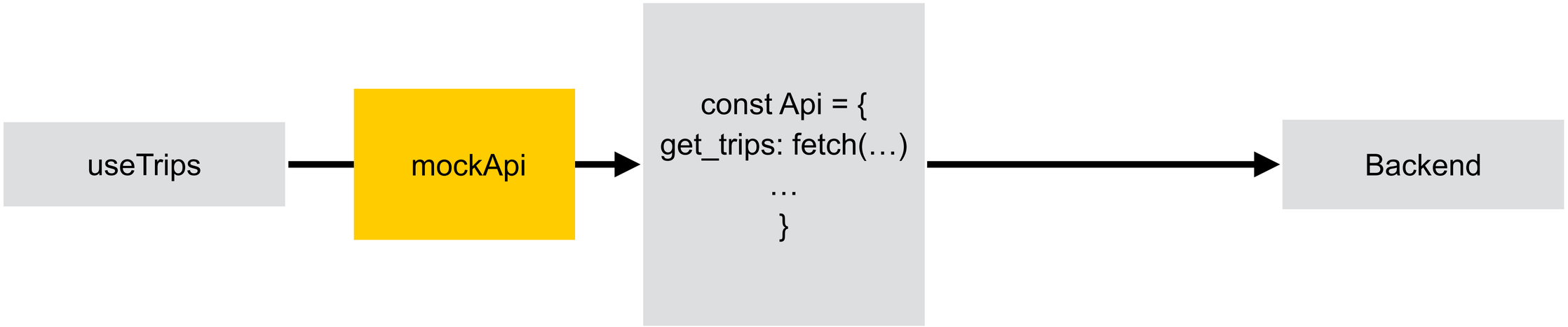
# Серверные данные в тестах и сторибуках



# Серверные данные в тестах и сторибуках



# Серверные данные в тестах и сторибуках



# Дебаг

〉 React Query DevTools (вначале захочется подебажить)
〉 Вкладка “Сетевые запросы” в инструментах разработчика

# Какие выводы сделали

# С точки зрения UX

〉 Приложение обновляется “само”, обновлять не требуется
〉 Держим высокий уровень оценок по опросам пользователей

# С точки зрения DX

- Нет бойлерплейта
- Сложный код оберток, простое API работы с ними
- Декларативные зависимости описаны в хуках
- Команда отмечает, что с данными работать просто

# Усложнять — просто,
# упрощать — сложно.

**Баранов Валерий**

Руководитель фронтенд-разработки
в Яндекс Командировках

HR Tech