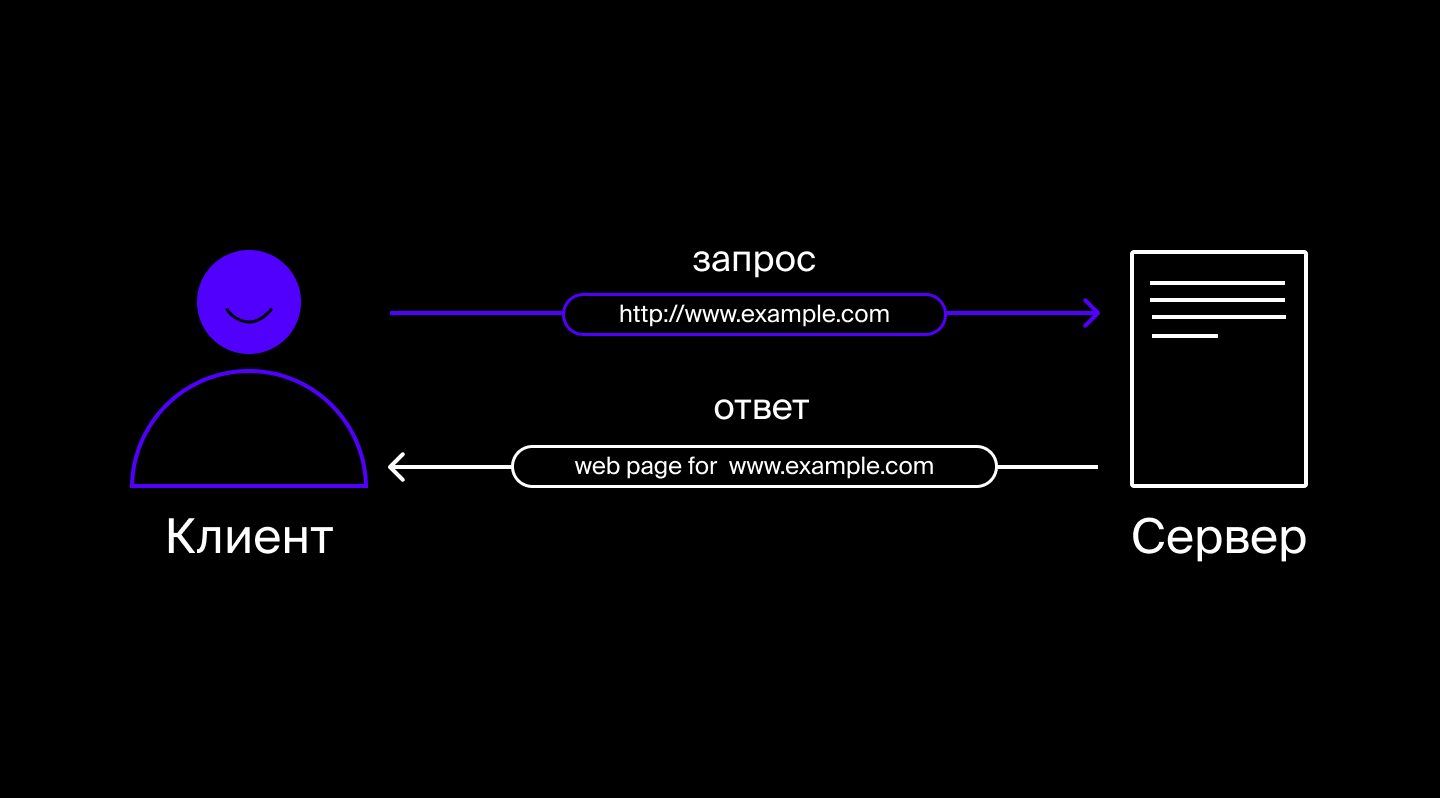
**Маршрутизация на стороне сервера**

Первый появившийся в мире подход к роутингу — роутинг на сервере.

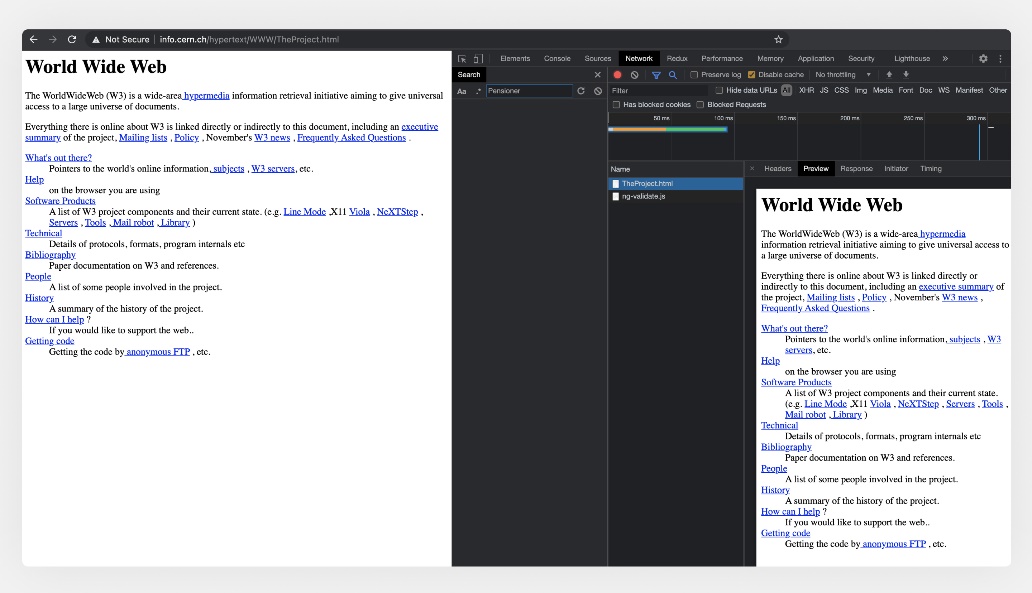
Алгоритм его работы выглядит так:

1. Пользователь вводит адрес сайта в браузерную строку.
2. На сервер отправляется запрос на получение нужной страницы.
3. Сервер формирует HTML и возвращает его клиенту.
4. Пользователь видит готовую страницу.



*А так можно визуализировать роутинг на стороне сервера.*

Такой способ маршрутизации используется на многих сайтах, в том числе и [на самом первом](http://info.cern.ch/), который появился в интернете. Если в панели разработчика открыть вкладку “Network” и нажать на любую ссылку страницы, например “Browse the first website”, можно увидеть запрос и ответ:



\*В 1991 году ещё не придумали тизерную рекламу, и сайты выглядели кристально чисто. \*

Роутинг на стороне сервера хорош тем, что:

* при первой загрузке сайта весь контент моментально доступен пользователю, так как сервер присылает готовую HTML-страницу;
* сайт с маршрутизацией на сервере хорошо обрабатывается поисковыми системами, так как контент внутри body уже известен.

Но у такого подхода есть и недостатки, к примеру:

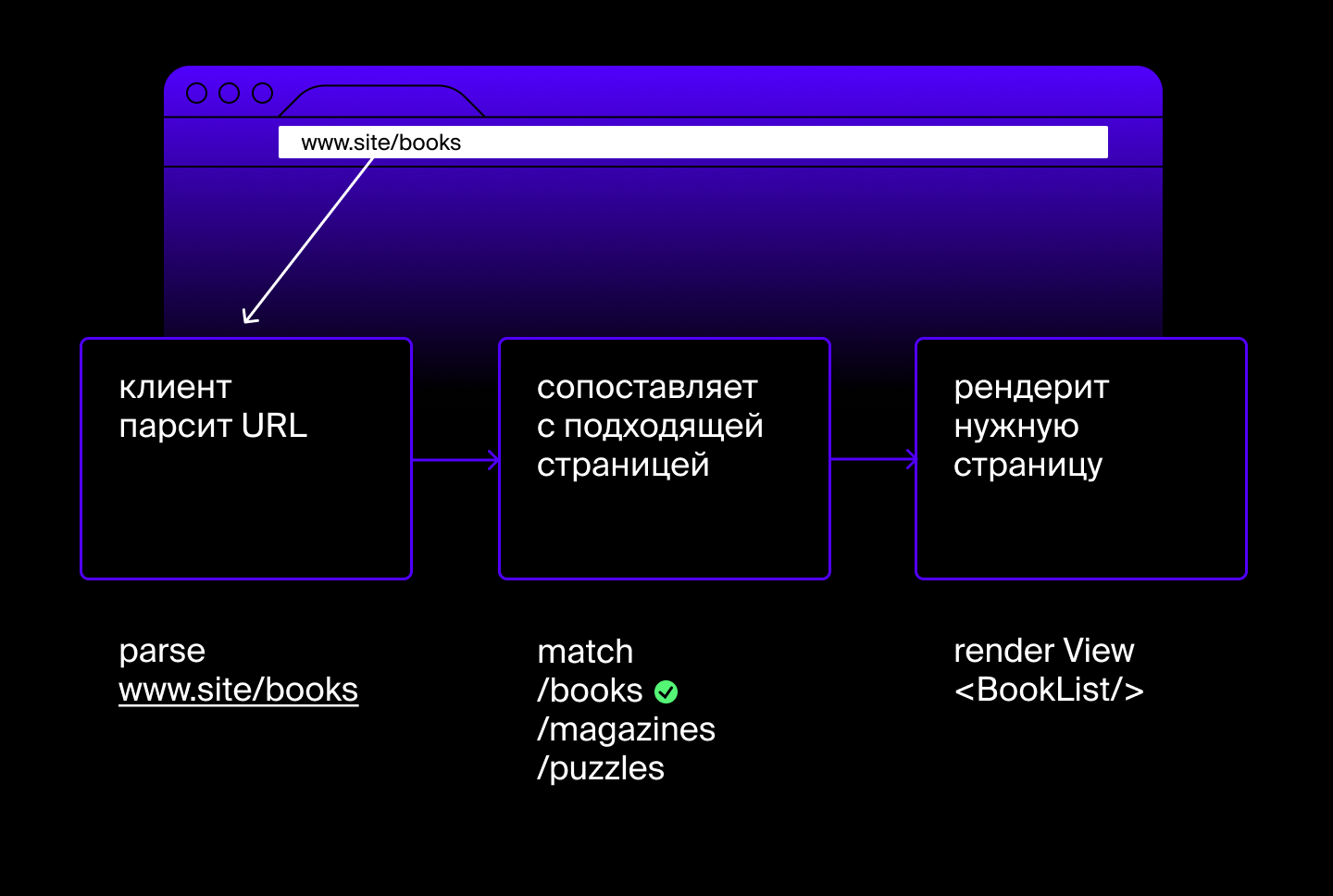
* страница полностью перерисовывается. Представьте сайт интернет-магазина, в котором на каждой странице одинаковые header и footer. Тогда при переходе между страницами сервер будет возвращать её полный HTML-код, и страница будет перерисовываться целиком;
* ответ очень долго ждать. Страницы могут быть довольно тяжёлыми, а если у пользователя ещё и скорость соединения с интернетом 1Kbps, ответ от сервера придётся ждать много световых лет, и ничего кроме белого экрана пользователь не увидит.

Для сайтов с единым шаблоном (к примеру сайдбаром, шапкой и футером), где информация от страницы к странице меняется лишь частично, оперативнее было бы перерисовывать только динамическую часть, а не всю HTML-страницу. Так появился новый подход — роутинг на стороне клиента. Расскажем про него в следующем уроке.

# Маршрутизация на стороне клиента

В предыдущем уроке мы рассмотрели, как осуществлялась навигация со времён первых сайтов. Маршрутизация на сервере отлично подходит для статических сайтов, что не скажешь про сайты с динамическими данными. В последних изменение одного заголовка на странице повлечёт за собой полную её перерисовку. А при большом количестве запросов к серверу нагрузка на сеть существенно возрастёт. Поэтому появился подход с клиентской маршрутизацией, который частично решает эти проблемы.

При таком подходе вся логика по сопоставлению URL с нужной страницей описывается на стороне клиента с помощью JavaScript:

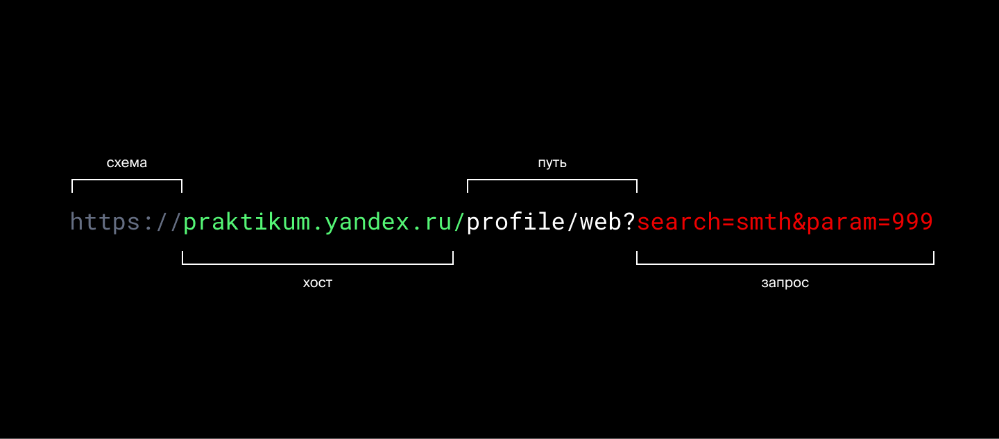


Роутинг на стороне клиента

Клиент разбивает адрес на определённые части — составляющие URL. Чтобы с этим лучше разобраться, рассмотрим, из чего может состоять адрес страницы.

## Части URL

Вы уже знаете, что URL — указатель на веб-ресурсы. Помимо страниц, он может указывать на изображения, аудиофайлы, материалы внутри страниц и другие интересные вещи. Давайте рассмотрим, из каких же частей состоит URL, что по ним происходит сопоставление с нужным ресурсом:



URL может состоять из таких элементов:

* Схема (scheme) — обязательная часть URL, она указывает, какой протокол передачи используется.
* Хост (host) — название, по которому можно найти сайт в интернете. Изначально адрес сайта, или IP-адрес, состоит из цифр и точек, и только потом к нему привязывается человекопонятное название сайта (доменное имя). Так, например, IP-адресу 127.0.0.1 соответствует имя localhost, и вы можете открыть локально запущенный проект любым способом. А в нашем примере [praktikum.yandex.ru](http://praktikum.yandex.ru/) — имя в системе доменных имён (Domain Name System, DNS), которое можно рассмотреть по частям:
  + praktikum — поддомен (subdomain), или домен, который не является корневым. Про [praktikum.yandex.ru](http://praktikum.yandex.ru/) можно сказать, что это домен третьего уровня;
  + [yandex.ru](http://yandex.ru/) — домен второго уровня, который в свою очередь — поддомен корневого домена ru;
  + ru — домен верхнего уровня (Top-level Domain, TLD) — самый высокий уровень в иерархии доменных имён.
* Порт (port) — порт хоста для подключения. Обычно эта часть в URL скрыта, но вы наверняка её замечали, когда локально запускали проекты. К примеру, в <http://localhost:3000/> 3000 — порт. Кстати, по умолчанию у HTTP 80-й порт, а у HTTPS 443.
* Путь (path) — путь к файлам или директориям на сервере.
* Запрос (query) — строка запроса, которая начинается с ? и состоит из пары параметр=значение. Если таких пар несколько, их соединяют &. Query — вспомогательное средство, часто используют для фильтрации или сортировки объектов. К примеру, из строки запроса https://любой-интернет-магазин/страница-товара?sortPrice=ASC можно понять, что применяется сортировка товаров по возрастанию цены. Строка запроса помогает сохранить состояние фильтра в случае перезагрузки страницы.
* Якорь (fragment) — ссылка внутри страницы. С помощью якорных ссылок можно переместиться к нужной части документа. Для этого в конец URL надо добавить #id-элемента-к-которому-хотите-перейти.

Именно эти части URL клиент сопоставляет с ресурсом, а затем возвращает нужную страницу.

Из этого урока вы узнали, что представляют собой URL и роутинг на клиенте. Теперь нужно получше разобраться в механизмах маршрутизации. Для этого в следующем уроке мы напишем свой клиентский роутер.

# Клиентский роутинг и History API

Чтобы разобраться, как работает роутинг на стороне клиента, напишем свой простейший роутер — сделаем интернет-портал для домовых.

Для этого потребуется HTML-разметка страницы. Создадим index.html и добавим панель навигации со ссылками на страницы и main для отображения основного контента:

Скопировать кодHTML

<nav>

<ul class="router">

<li><a class="route" id="dashboard" href="/home">Home</a></li>

<li><a class="route" id="shop" href="/shop">Shop</a></li>

</ul>

</nav>

<main>

<div id="content"></div>

</main>

Теперь повесим на каждую ссылку обработчик события onclick:

Скопировать кодJSX

window.onload = () => {

const navLinks = document.getElementsByClassName("route");

for (let link of navLinks) {

link.addEventListener("click", push);

}

};

В методе push будет происходить магия роутинга. Добавим его:

Скопировать кодJSX

const push = (event) => {

// Отменяем действие по умолчанию, которое должно произойти по клику на ссылку

event.preventDefault();

const {

target: { id, classList }

} = event;

const url = event.target.getAttribute("href");

// Изменяем заголовок страницы <title></title>

document.title = id;

// Меняем URL страницы

window.history.pushState({ id }, "", url);

};

За роутинг в нашем коде отвечает строка window.history.pushState({ id }, "", url);. Разберём её подробнее.

## History API

Свойство history глобального объекта window предоставляет доступ к History API и позволяет работать с историей браузера, по которой мы можем навигироваться или даже изменять её.

Существует несколько вариантов навигации по сайту:

* переход на определённую страницу сайта,
* возврат на шаг назад,
* переход на шаг вперёд.

Но чтобы перемещаться по сайту, надо записать страницы в историю браузера. Это делается в процессе перемещения по сайту, чтобы сохранить определённую последовательность посещений, по которой можно вернуться назад (например при помощи браузерных стрелок «Вперед» и «Назад»). О навигации по истории поговорим чуть позже, сначала разберёмся с добавлением страниц.

### Добавление новой записи в историю

Сделать это можно двумя способами:

* window.history.pushState(state, title [, url]); — добавить новую запись,
* window.history.replaceState(state, title [, url]) — заменить существующую.

В оба этих метода можно передать три параметра:

* state — состояние для новой записи в истории. Так можно пробрасывать параметры на страницы и получать их именно в том месте, куда передали. Если параметры не нужны, можно передать null. В примере выше мы передали уникальный id из тега a, теперь мы сможем его использовать в качестве заголовка страницы. В следующих уроках мы более подробно разберём, как можно работать с состоянием истории.
* title — заголовок страницы. Пока все браузеры, кроме Safari, игнорируют этот параметр, но когда-то могут начать его использовать, поэтому лучше передавать пустую строку.
* url — URL страницы, которую хотим показать. Это необязательный параметр. Иногда не нужно добавлять новую запись в историю браузера, требуется лишь изменить state текущей страницы. В этом случае url можно опустить.

Метод pushState подойдёт для добавления новых шагов в историю браузера. Если мы будем возвращаться назад по истории с помощью браузерной кнопки «Назад», пройдём эти шаги. Например, на сайтах с иерархией, как и наш интернет-портал для домовых, необходимо сохранять историю посещений. Если домовой зайдёт в “Shop”, потом в категорию «Отопительные приборы» и выберет товар «Печь», назад он должен возвращаться по тому же пути.

Но бывает, что пользователь не должен попадать на предыдущие шаги. Например, экран ввода кода из смс не отображается каждый раз, когда пользователь возвращается на сайт. Удалить роуты мы не можем, зато можем переписать историю. Для этого и подходит второй метод — replaceState.

### Навигация по истории посещений

Чтобы перемещаться по посещённым страницам, History API предоставляет несколько методов:

* history.back() — перемещение на предыдущую страницу истории;
* history.forward() — перемещение на следующую страницу истории;
* history.go(n) — универсальное перемещение. С помощью этого метода можно переместиться на любую страницу истории, нужно лишь вместо n передать необходимое число. Вызвав history.go(2), вы переместитесь на 2 страницы вперёд от текущей, а history.go(1), на одну страницу вперёд, что соответствует history.forward(). Кстати, n может принимать отрицательные значения, и history.go(-1) соответствует history.back().

Теперь рассмотрим, как работать с этими методами на практике, — вернёмся к порталу с домовыми. Если домовой перейдёт с Home на Shop и нажмёт на браузерную кнопку «Назад», опять попадёт на страницу Home. Тут как раз следует использовать history.back(); или history.go(-1);. Добавим на страницу кнопку и по клику на неё будем вызывать history.back():

Скопировать кодHTML

<main>

<div id="content"></div>

<button id="backButton">Back</button>

</main>

Скопировать кодJSX

window.onload = () => {

...

const back = document.getElementById("backButton");

back.onclick = () => window.history.back();

};

Вот и всё, мы написали простой клиентский роутинг на JavaScript. В этом уроке вы познакомились с главными составляющими History API: методами, с помощью которых можно добавлять страницы в историю, — pushState, replaceState, и методами для перемещения по этим страницам — back, forward, go.

Если вам показалось, что клиентский роутинг в разы круче серверного, это не совсем так. Например, при первоначальной загрузке сайта с клиентским роутингом пользователю придётся немного потупить в пустой экран из-за отсутствия готовой HTML-страницы для отображения. Но уже при последующей навигации по сайту всё загрузится моментально.

В общем, у обоих подходов есть свои достоинства и недостатки. Какой из них лучше использовать, решать вам — просто отталкивайтесь от поставленной задачи.

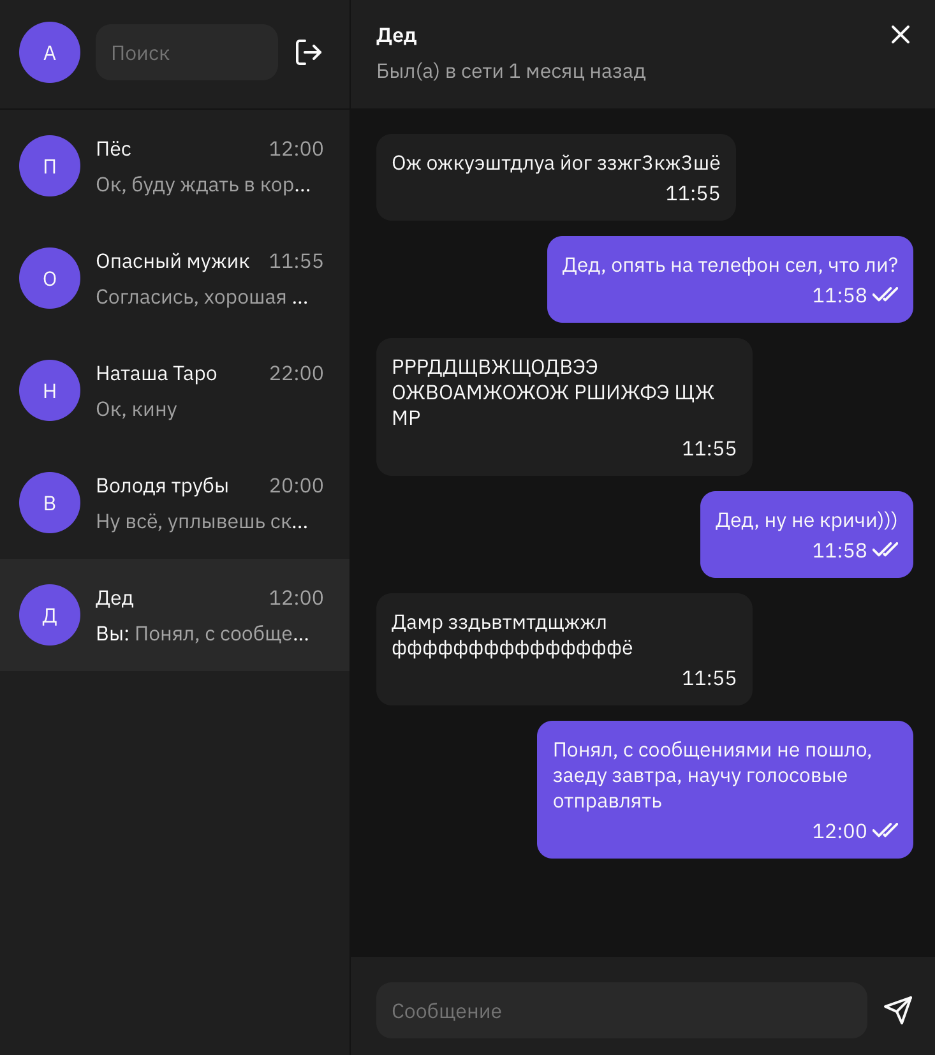
Уже в следующих уроках познакомимся с роутингом в React и библиотекой React Router.

Всё просто: React Router ускоряет процесс разработки и добавляет в проект наглядности. Например, позволяет использовать роутинг, который основан на файловой структуре вашего приложения, когда каждой странице соответствует папка или файл.

# Установка и подключение React Router

В предыдущих уроках вы познакомились с подходом маршрутизации на клиенте и для этого использовали History API — window.history. В React-приложениях для клиентского роутинга обычно используют библиотеку React Router. Под капотом она так же работает с History API, только ещё предоставляет различные компоненты, функции и хуки для удобной организации роутинга. Кроме этого, разработчики библиотеки позаботились о решении распространённых проблем и нам не придётся тратить на них своё время.

В этом уроке вы научитесь устанавливать и настраивать React Router в проекте и познакомитесь с возможностями библиотеки. Мы сделаем чат, который под конец спринта будет выглядеть как-то так:



Кажется, ради деда придётся добавить в чат и голосовые сообщения. Но это не точно.

Сперва создадим новый проект с помощью CRA. Для этого в командной строке введите npx create-react-app chat.

Чтобы установить библиотеку React Router, откройте главную директорию проекта и введите npm i react-router-dom@5.2.1. Эта версия React Router предназначена для маршрутизации в браузерных приложениях.

Важно установить именно эту версию библиотеки —5.2.1. Сейчас доступна более новая мажорная версия, но она пока не стала популярной в больших проектах.

Теперь надо организовать роутинг. Чтобы задать определённые адреса для страниц приложения, используем два компонента Route и BrowserRouter.

Компонент Route — главный строительный блок. Он отвечает за привязку [path](https://www.notion.so/3-58dcca55e1664ba397685356ee7ce0f2) к компоненту. Чтобы Route работал правильно, нужно убедиться, что он вложен в компонент BrowserRouter.

Компонент BrowserRouter отслеживает историю навигации в процессе работы. Когда пользователь переходит в браузере назад или вперёд, BrowserRouter показывает нужный контент.

Теперь самое время добавить пару маршрутов. Сделаем это в следующем уроке!

# Создание маршрутов

Пора добавить в приложение несколько маршрутов. В папке pages мы будем хранить все экраны приложения, для которых создадим маршруты. Добавим первый экран login.jsx:

Скопировать кодJSX

*// login.jsx*

import React from 'react';

import styles from './login.module.css';

export function LoginPage() {

return (

<div className={styles.wrapper}>

<form className={styles.form}>

<h1 className={styles.heading}>Вход</h1>

</form>

</div>

);

}

Теперь сделаем маршрут для страницы логина. Для этого импортируем компоненты Route и LoginPage в файл App.js. Внутрь Route вложим компонент, который отрисовывается, когда адрес соответствует пропсу path. И не забудем обернуть всё в BrowserRouter:

Скопировать кодJSX

*// App.js*

import { BrowserRouter as Router, Route } from 'react-router-dom';

import './App.css';

import { LoginPage } from './pages';

export default function App() {

return (

<Router>

<Route path="/login">

<LoginPage />

</Route>

</Router>

);

}

Готово. Если запустить приложение и перейти в браузере на /login, вы увидите содержимое LoginPage.

## Больше маршрутов!

Пока наше приложение содержит всего один экран — экран входа. Добавим главную страницу со списком чатов. В list.jsx сделаем запрос на сервер и отрендерим список чатов:

Скопировать кодJSX

*// list.jsx*

import React, { useEffect, useState } from 'react';

import { ChatsList } from '../components/chats-list';

import listStyles from './list.module.css';

export const ListPage = () => {

const [data, setData] = useState([]); *// здесь будем хранить список чатов*

const [loading, setLoading] = useState(false); *// состояние для загрузки данных*

const content = loading ? (

'loading'

) : data && data.length ? (

<ChatsList chats={data} />

) : null;

return (

<div className={listStyles.container}>

<div className={listStyles.list}>

{content}

</div>

</div>

);

};

Добавим метод для загрузки списка чатов с сервера:

Скопировать кодJSX

*// list.jsx*

...

import { getChats } from '../services/api';

export const ListPage = () => {

...

const getChatsData = async () => {

*// добавим состояние загрузки*

*// большой объём данных может подгружаться довольно долго*

setLoading(true);

try {

const chats = await getChats();

setData(chats);

} finally {

setLoading(false);

}

};

useEffect(() => {

*// загрузим список чатов*

getChatsData();

}, []);

return (

);

};

Затем перейдём в App.js и добавим роут для списка чатов. Чтобы в браузере отображался только один компонент, обернём маршруты в Switch:

Скопировать кодJSX

*// App.js*

import { BrowserRouter as Router, Switch, Route } from 'react-router-dom';

import './App.css';

import { LoginPage, ListPage } from './pages';

export default function App() {

return (

<Router>

<Switch>

<Route path="/">

<ListPage />

</Route>

<Route path="/login">

<LoginPage />

</Route>

</Switch>

</Router>

);

}

Теперь, если вы перейдёте на <http://localhost:3000/login>, увидите ListPage.

Всё просто. Компонент Switch проверяет маршруты сверху вниз и рендерит самый первый из подходящих, так как значение пропса path сравнивается с путём в URL не по принципу равенства, а по принципу «начинается с». Иными словами, маршрут с путём / будет активным для любого пути в URL, ведь любой путь всегда начинается с /. Даже если перейти по localhost:3000/login, всё равно отобразится ListPage, ведь в названии маршрута тоже стоит /.

Решить эту проблему можно несколькими способами:

* поставить <Route path="/"> самым последним в Switch;
* добавить <Route path="/" exact={true}>, где exact указывает на полное соответствие путей.

Мы рассмотрели работу Switch и exact в деле, теперь для наглядности заменим URL для списка чатов на /list:

Скопировать кодJSX

*// App.js*

export default function App() {

return (

<Router>

<Switch>

<Route path="/list">

<ListPage />

</Route>

<Route path="/login">

<LoginPage />

</Route>

</Switch>

</Router>

);

}

## Всё хорошо, что хорошо маршрутизируется

Чтобы использовать React Router в проекте, нужно импортировать необходимые компоненты. Если хотите добавить Route в приложение, убедитесь, что он обёрнут в компонент BrowserRouter. Компонент Route устанавливает связь между путём, указанным в пропсе path, и URL-адресом, который в данный момент используется браузером. При каждом обращении к этому URL будет отображаться компонент внутри маршрута. Компонент Switch применяют в тех случаях, когда нужно отрисовать только один маршрут.

В следующем уроке вы узнаете, как в React Router работать со ссылками и навигацией внутри приложения.

# Навигация и ссылки. Основа

Надо признать, что принуждать пользователей вручную вводить URL-адрес, чтобы перемещаться по приложению, — не самое мудрое решение. Вы уже знаете про тег <a> и о том, как его можно использовать, чтобы помочь пользователям находить нужную информацию.

React Router предлагает свой набор инструментов навигации, и в этом уроке мы их рассмотрим.

## Недостающая ссылка

React Router предоставляет компонент Link, который выполняет операцию, аналогичную знакомым тегам <a> из стандартного HTML. В действительности Link — тег <a> с дополнительными характеристиками для работы с React Router.

Импортируем компонент Link из react-router-dom. Затем добавим ссылку на компонент ListPage, чтобы пользователи могли перейти на страницу чатов. Сам процесс авторизации в приложении мы сделаем чуть позже.

Скопировать кодJSX

*// login.jsx*

import React from 'react';

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

import styles from './login.module.css';

export function LoginPage() {

return (

<div className={styles.wrapper}>

<form className={styles.form}>

<h1 className={styles.heading}>Вход</h1>

<Link to='/list'>

Войти

</Link>

</form>

</div>

);

}

У компонента Link есть пропс to для описания URL, по которому следует перейти.

## Не отклоняйтесь от маршрута

Вы уже научились реализовывать базовую маршрутизацию и навигацию в одностраничных приложениях. Чтобы настроить маршруты, нужно использовать BrowserRouter и Route. Можно сделать так, чтобы только один соответствующий маршрут отрисовывался с помощью компонента Switch, а с помощью exact указать, что путь должен быть точным. Также в этом уроке вы узнали, как с помощью компонента Link сделать удобную навигацию.

Пора усложнять задачу. В следующих уроках изменим текущую реализацию навигации — добавим выделение на выбранный чат, а также сделаем программную навигацию.

# Навигация и ссылки. Продвинутое использование

Вы уже знаете, что для создания удобной навигации нужно добавить компонент Link в ChatsList:

Скопировать кодJSX

*// импортируем Link*

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

...

export const ChatsList = ({ chats }) => {

const normalizedData = React.useMemo(() => normalizeData(chats), [chats]);

return (

<ul>

{normalizedData.map(({ recipientName, id, timestamp, lastMessage }, index) => (

<li key={index}>

{/\* добавим ссылку на чат \*/}

<Link

to={{ pathname: `/list/${id}` }}

className={styles.chat}

>

<Avatar name={recipientName} />

<div className={styles.chatContent}>

<div className={styles.info}>

<span>{recipientName}</span>

<span>{getTimeFromTimestamp(timestamp)}</span>

</div>

<div className={styles.lastMessage}>

<span>{lastMessage.text}</span>

</div>

</div>

</Link>

</li>

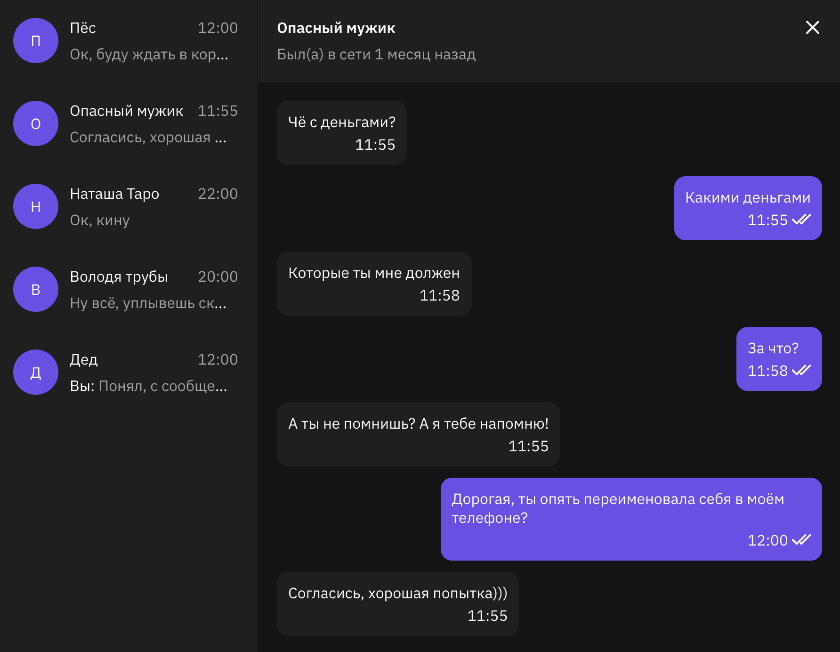
))}

</ul>

);

};

Теперь мы можем посмотреть переписку:



Кажется, тут разворачивается драма...

Вот только незаметно, какой из чатов выбран. Лучше бы его выделить. Для этого нужно заменить Link на NavLink.

## Компонент NavLink

Этот компонент отличается от Link тем, что у него есть пропс activeClassName. В качестве значения он принимает селектор CSS-класса. Указанный в activeClassName CSS-класс будет добавлен к тому элементу NavLink, маршрут которого совпадает с текущим URL в браузере. Так вы сможете применять собственные стили к ссылкам для текущей активной страницы. Это полезно в панели навигации, поскольку помогает пользователю проследить его текущее местоположение в приложении.

Скопировать кодJSX

<NavLink

to={{ pathname: `/list/${id}` }}

className={styles.chat}

activeClassName={styles.activeChat}

>

## Программная навигация

Пока что для навигации мы использовали Link и NavLink. Если посмотреть код собранного приложения, вы увидите ссылку — обычный тег a. Ссылкой мы можем сделать любые текстовые блоки. Но что делать, если на странице есть кнопка и по клику на неё нам нужно перейти на другую страницу? Если обернуть её в Link, при сборке получим <a><button.... Но у кнопки есть обработчик события onClick и правильнее в нём осуществить переход на другую страницу при помощи программной навигации. Тогда и не придётся оборачивать кнопку в ссылку.

Используем хук useHistory, он возвращает объект history, который помогает программно контролировать навигацию пользователя. Объект history содержит множество свойств и методов, подробнее с ними можно ознакомиться [в официальной документации библиотеки](https://reactrouter.com/web/api/history). Для реализации программного перехода на другую страницу мы используем метод replace(path, [state]), который заменяет существующую запись в истории на новую.

Сейчас на странице логина есть ссылка Link, стилизованная под кнопку. Не будем изобретать велосипед и заменим её на обычную кнопку:

Скопировать кодJSX

<Button onClick={login} primary={true}>

Войти

</Button>

Теперь импортируем useHistory в файл:

Скопировать кодJSX

import { useHistory } from 'react-router-dom';

И вызовем хук внутри функционального компонента. Для этого присвоим его константе history:

Скопировать кодJSX

const history = useHistory();

Затем в методе login мы можем вызвать метод history.replace и войти в наше приложение с чатами:

Скопировать кодJSX

const login = useCallback(

() => {

history.replace({ pathname: '/list' });

},

[history]

);

В этом уроке вы узнали про NavLink, который удобно использовать, когда надо стилизовать активную ссылку. А также рассмотрели хук useHistory, с помощью которого можно программно навигироваться в приложении, если не нужно оборачивать кнопку в Link и NavLink.

import { Button } from '../components/button';

export function HomePage() {

const history = useHistory();

const onClick = useCallback(

() => {

history.replace({ pathname: '/list' });

},

[history]

);

return (

<div className={styles.wrapper}>

<div className={styles.container}>

<form className={styles.form}>

<h1 className={styles.heading}>Nobel Prize Library</h1>

<Button onClick={onClick} primary={true}>

View catalog

</Button>

</form>

<p>1901-2020</p>

</div>

</div>

);

} "id": 1,

# Динамические маршруты

В предыдущих уроках вы познакомились с основами навигации и маршрутизации внутри приложений и попрактиковались в применении инструментов React Router. В этом уроке перейдём на новый уровень и разберём продвинутые возможности этой библиотеки.

Пока в приложении есть только список чатов, который мы получаем от сервера в формате JSON:

Скопировать кодJSON

"chats": [

{

"id": 1,

"recipientName": "Glenna",

"lastSeen": 1619082000,

"recipientId": 34,

"messages": [

{

"id": 1,

"senderId": 34,

"text": "Привет, как дела?",

"timestamp": 1619082000

}

]

},

{

"id": 2,

...

},

...

]

Нам нужно добавить роуты для просмотра переписок. Для этого мы могли бы использовать id чатов и добавить новые маршруты в App.js так:

Скопировать кодJSX

<Route path='/list/1'>

<ChatPage />

</Route>

<Route path='/list/2'>

<ChatPage />

</Route>

Только чатов может быть бесконечное множество, и id могут быть какими угодно. Получается, приложение совсем не масштабируемо, поскольку описанные в нём маршруты не умеют работать с динамическими данными. Поправим это с помощью React Router — добавим Route с динамическим path:

Скопировать кодJSX

*// list.jsx*

import React, { useEffect, useState } from 'react';

*// добавим импорт компонентов для создания нового роута*

import { Route, Switch } from 'react-router-dom';

import { ChatPage } from './chat';

...

export const ListPage = () => {

const [data, setData] = useState([]); *// список чатов*

const [loading, setLoading] = useState(false);

...

return (

<div className={listStyles.container}>

<div className={listStyles.list}>

{content}

</div>

{/\* добавим роут для просмотра чата \*/}

<Switch>

<Route path='/list/:chatId'>

<ChatPage />

</Route>

</Switch>

</div>

);

};

Мы добавили Route именно в ListPage, а не в App.js. Это нужно, чтобы экран с перепиской отображался сбоку от списка чатов.

Новый компонент ChatPage содержит переписку и отображается по динамическому маршруту /list/:chatId.

Теперь перед нами встают две задачи:

1. Отобразить переписку в ChatPage. Для этого нужно получить из URL id чата при помощи хука useParams.
2. Не хардкодить путь для ChatPage. Если путь для компонента ListPage поменять в App, к примеру с /list на /chats, роутинг для ChatPage сломается, ведь для него прописан путь /list/:chatId. Чтобы этого избежать, нужно в ListPage получить текущий path, который указан в Route. Для этого следует воспользоваться хуком useRouteMatch.

Рассмотрим, как решить эти задачи на практике.

## Параметры URL. Хук useParams

useParams возвращает объект, который содержит ключ-значение для каждого параметра в адресе. В нашем случае этот хук вернёт объект с ключом chatId и значением id переписки.

Например, когда мы находимся по адресу /list/777, вызов useParams вернёт такой объект:

Скопировать кодJSON

{

chatId: "777"

}

С помощью хука useParams мы получаем значения параметров адреса, в который обёрнут компонент ChatPage. В нашем случае мы можем использовать эти значения, чтобы отобразить нужный чат.

## Совпадение маршрутов. Хук useRouteMatch

Хук useRouteMatch применяется, чтобы получить данные о том, как Route сравнивает текущий адрес с path, переданным в Route. Хук возвращает [объект match](https://reactrouter.com/web/api/match), который и содержит все необходимые данные:

* params — объект с параметрами из path, такой же, как возвращает useParams;
* isExact — true, если адрес совпадает с path в роуте;
* path — сам path, который задан в Route;
* url — текущий url от корня сайта.

Есть несколько способов вызова useRouteMatch:

1. Без параметров. Так мы получаем объект match с данными о сравнении адреса с path роута, в который обёрнут компонент.

В этом случае объект match чаще используется для получения path и url.

url используется для создания ссылок относительно текущего пути:

Скопировать кодJSX

const { url } = useRouteMatch();

<Link to={`${url}`/some-nestend-url}>

A path используется для создания роутов в компоненте, в который вложен другой роут. Как раз то, что нам нужно — получим в ListPage path:

Скопировать кодJSX

export const ListPage = () => {

*// path === /list*

const { path } = useRouteMatch();

...

return (

<div className={listStyles.container}>

<div className={listStyles.list}>

{content}

</div>

{/\* добавим роут для просмотра чата \*/}

<Switch>

<Route path={`${path}/:chatId`}>

<ChatPage />

</Route>

</Switch>

</div>

);

};

1. С параметрами. В качестве параметра в useRouteMatch передаётся строка или объект, которые описывают path. Такой вызов хука применяется для другой задачи — проверки соответствия текущего адреса с каким-то path. Так без использования компонента Route мы можем проверить такое соответствие:

Скопировать кодJSX

const match = useRouteMatch("/blog/:slug");

const match = useRouteMatch({

path: "/BLOG/:slug/",

strict: true,

sensitive: true

});

В результате мы получаем объект сравнения текущего адреса с переданным path в useRouteMatch. Так мы без использования Route можем проверить, подходит ли текущий адрес под заданный path. Если адрес не подходит, useRouteMatch вернёт null, если подходит — объект match.

Пример:

export default function App() {

return (

<Router>

<Switch>

<Route path="/" exact={true}>

<HomePage />

</Route>

<Route path="/list" exact={true}>

<ListPage />

</Route>

<Route path="/list/:country">

<CountryPage />

</Route>

</Switch>

</Router>

);

export const CountryPage = () => {

const [laureates, setLaureates] = useState([]);

const [countryTitle, setCountryTitle] = useState('');

const { country } = useParams();

const loadCountryInfo = useCallback(

() => {

loadCountries().then(countries => {

const currentCountry = countries.find(({ code }) => code === country);

setCountryTitle(currentCountry && currentCountry.name ? currentCountry.name : country);

});

},

[country]

);

} });

export const CountryList = ({ countries }) => {

const { url } = useRouteMatch();

const { code } = useParams();

return (

<ul>

{countries.map(({ name, code, count }, index) => (

<li key={index}>

<Link to={{ pathname: `${url}/${code}` }} className={styles.link}>

<div className={styles.country}>

<div className={styles.info}>

<span>{name}</span>

</div>

<div className={styles.info}>

<span>{count}</span>

</div>

</div>

</Link>

</li>

))}

</ul>

);

};

;

# Фильтрация и сортировка значений

Кажется, вы знаете про роутинг всё: можете настроить динамические маршруты, переходы по ссылкам и программную навигацию. Но в этом уроке мы раскроем ещё один секрет URL. Расскажем, как [строка запроса (query)](https://practicum.yandex.ru/learn/react/courses/36f94820-19c9-4dfb-beed-5c4858c587e2/sprints/9586/topics/b6061376-13a6-4959-9f1c-6b1303f69da1/lessons/46453f36-5536-465b-896b-b5519244c157/) может помочь при фильтрации и сортировке значений.

Представьте, что вы создали сайт строительного интернет-магазина: настроили в нём динамические маршруты, навигацию и ссылки. Пользователи начали им пользоваться: настраивать множество фильтров в поисках той самой извёстки. Но каждый раз, когда они случайно перезагружали страницу, все фильтры сбрасывались. Ремонт — и так дело нервное, а тут ещё и неуклюжий пользовательский интерфейс подъехал. Чтобы уберечь нервы пользователей, нужно использовать query-параметры — они сохраняются в адресе при перегрузке страницы, в то время как стейт приложения очищается.

Рассмотрим query-параметры в деле на примере нашего чата. Мы можем сделать поиск по чатам, то есть отфильтровать их по наибольшему совпадению в названии. Добавим на страницу с чатами input для поиска контакта:

Скопировать кодJSX

// list.tsx

...

return (

<div className={listStyles.container}>

<div className={listStyles.list}>

<div className={listStyles.searchbar}>

<Avatar name={'A'} />

<Input placeholder="Поиск" onChange={onChange} value={searchValue} />

<img alt="logout" src={logoutImage} onClick={logout} />

</div>

...

Подпишемся на событие onChange и, по мере ввода символов, будем записывать их в стейт searchValue:

Скопировать кодJSX

const [searchValue, setSearch] = useState('');

...

const onChange = e => {

setSearch(e.target.value);

};

Теперь самое интересное: нужно добавить searchValue к строке запроса. Если пользователь начнёт вводить в инпут «дед», строка запроса должна выглядеть так:

Скопировать код

https://localhost:3000/list?q=дед

Добавить к адресу строку запроса можно с помощью хука useHistorу, с которым мы знакомились [в предыдущих уроках](https://praktikum.yandex.ru/trainer/react/lesson/714d83ed-aea6-41cd-849b-c6c40dd9e0b3).

Для начала импортируем useHistory в файл:

Скопировать кодJSX

import { useHistory } from 'react-router-dom';

Затем вызовем хук внутри функционального компонента. Для этого присвоим его константе history:

Скопировать кодJSX

const history = useHistory();

Теперь query в адресе можно заменить с помощью метода replace. Передадим в метод объект с ключом search и значением, которое и будет соответствовать строке запроса:

Скопировать кодJSX

useEffect(

() => {

if (searchValue) {

history.replace({ С });

} else {

history.replace({ search: '' });

}

},

[searchValue, history, search]

);

В случае, если пользователь что-то ввёл в инпут, мы изменим URL, а если инпут пуст, то необходимо очистить строку запроса { search: '' }.

Для реализации поиска по чатам нам осталось только отследить изменения query-параметров в коде. Для этого используем другой хук — useLocation. Он предоставляет всю информацию о текущем URL — query-параметры, якорь и другие интересные вещи.

Добавим его в импорт:

Скопировать кодJSX

import { useHistory, useLocation } from 'react-router-dom';

Весь объект location нам не нужен, поэтому получим только search:

Скопировать кодJSX

const { search } = useLocation();

Если обнаружим строку запроса, десериализуем её. То есть превратим строку в объект — "?q=ad" ⇒ { q: "ad" }:

Скопировать кодJSX

useEffect(

() => {

if (search) {

const { q } = deserializeQuery(search);

setSearch(q);

}

},

[search]

);

Дело за  фильтрацией:

Скопировать кодJSX

const preparedData = useMemo(

() => {

return data.filter(

chat => chat.recipientName.toLocaleLowerCase().indexOf(searchValue.toLocaleLowerCase()) > -1

);

},

[data, searchValue]

);

Теперь preparedData передадим в ChatsList:

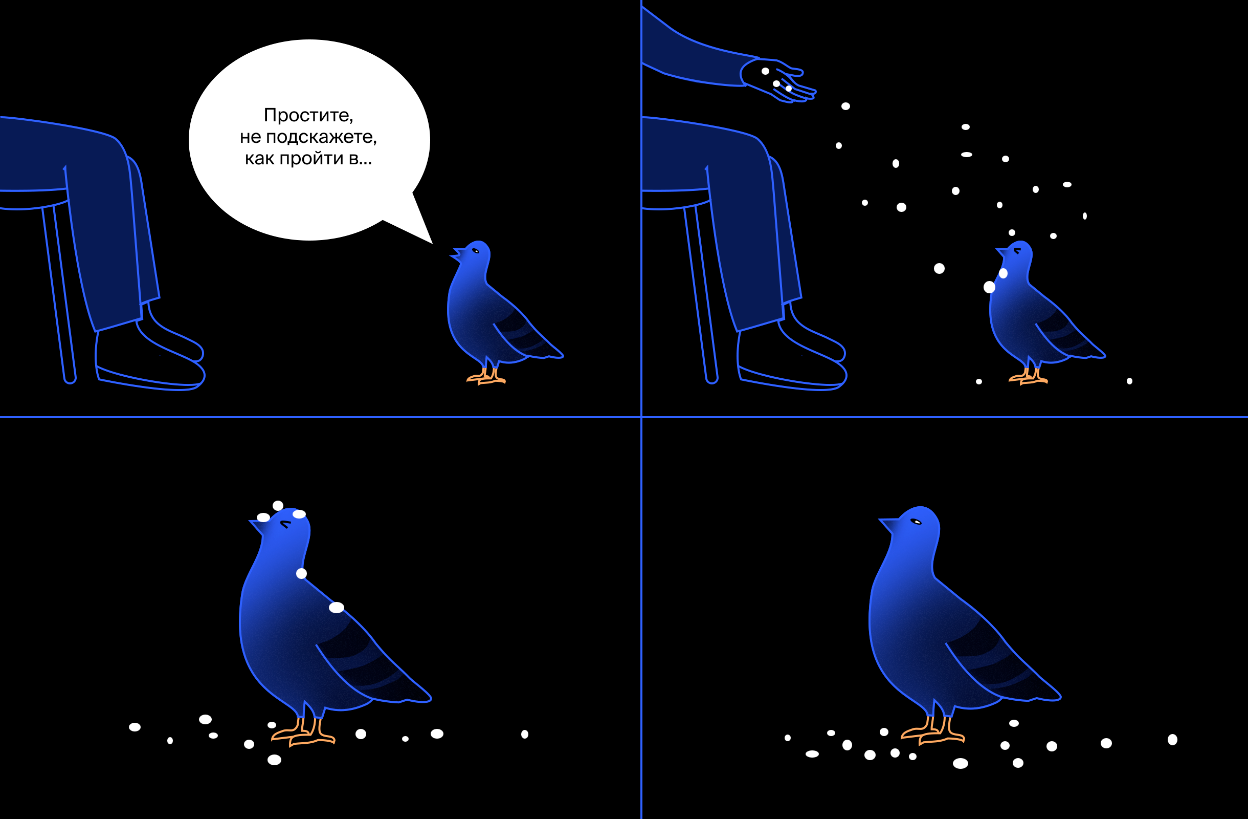
Скопировать кодJSX

<ChatsList chats={preparedData} />

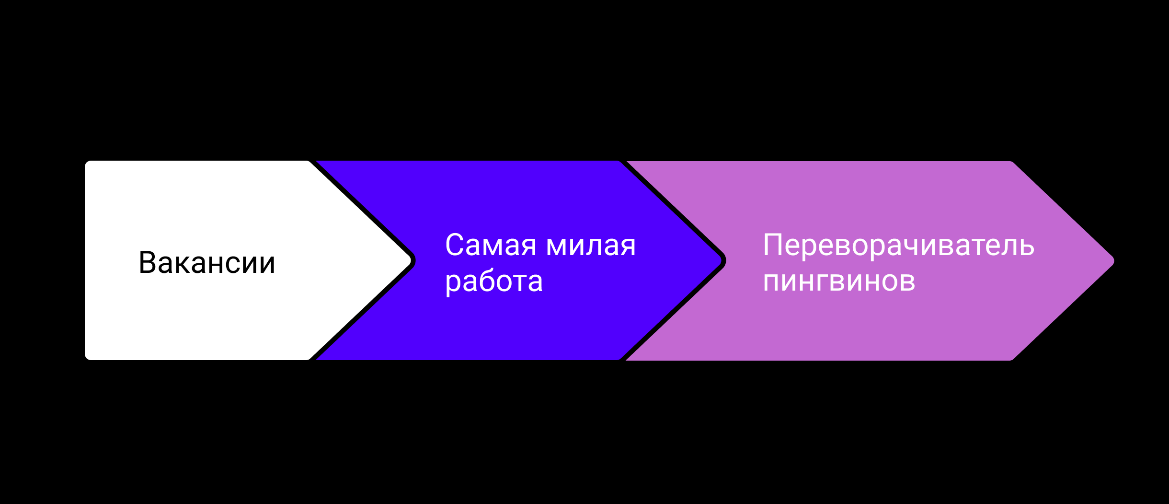
Фильтрация чатов готова! Вы узнали, как можно использовать строку запроса, осталось закрепить эти знания на практике.

# Хлебные крошки

Хлебные крошки (Breadcrumbs) — навигационное меню, которое показывает, где сейчас находится пользователь.



Вы наверняка встречали на сайтах такое меню:



Обычно навигационная цепочка начинается с главной страницы, а последний элемент всегда указывает текущее местоположение пользователя и не является ссылкой, в отличие от его родителей. По этой же причине навигационное меню никогда не размещают на главной странице, так как цепочка будет включать всего один некликабельный элемент.

## Способ организации меню

Хлебные крошки можно разделить по способу их организации:

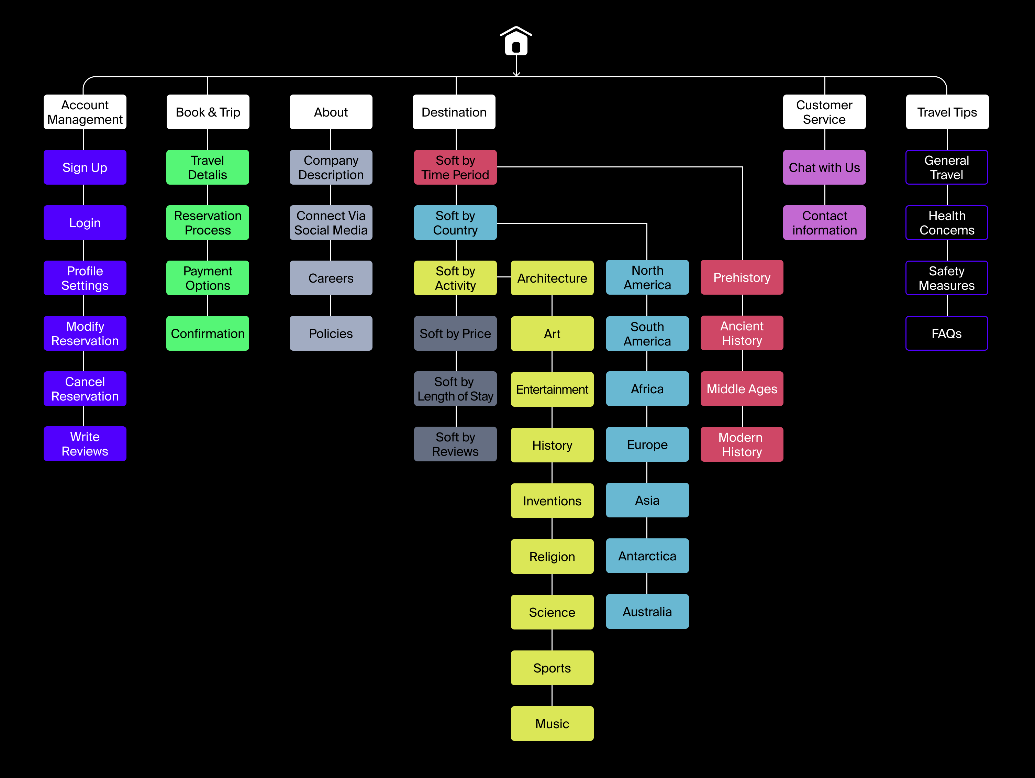
1. **На основе иерархии.** Это самый распространённый тип. Меню строится на основе вложенности разделов и страниц сайта. То есть когда заранее известны все категории, например все жанры книг в магазине.
2. **Динамическое меню.** В этом случае навигационная цепочка зависит от пути, который проходит пользователь. Например, на сайте художественного салона товар «Кисть» может быть в категориях «Живопись» и «Товары для детей». Тогда в хлебных крошках нужно указать именно ту категорию, из которой был выбран товар.

Такое навигационное меню — удобная штука. Пользователь видит весь путь, который прошёл до текущей страницы, и может легко вернуться в любой из просмотренных разделов. Но при этом хлебные крошки подойдут не для всех сайтов.

## Когда навигационное меню подойдёт

Хлебные крошки стоит использовать на масштабных проектах, где есть определённая иерархия. В том же художественном салоне наберётся не одна категория и подкатегория товаров.

К примеру, если сайт выглядит так, без хлебных крошек явно не обойтись:

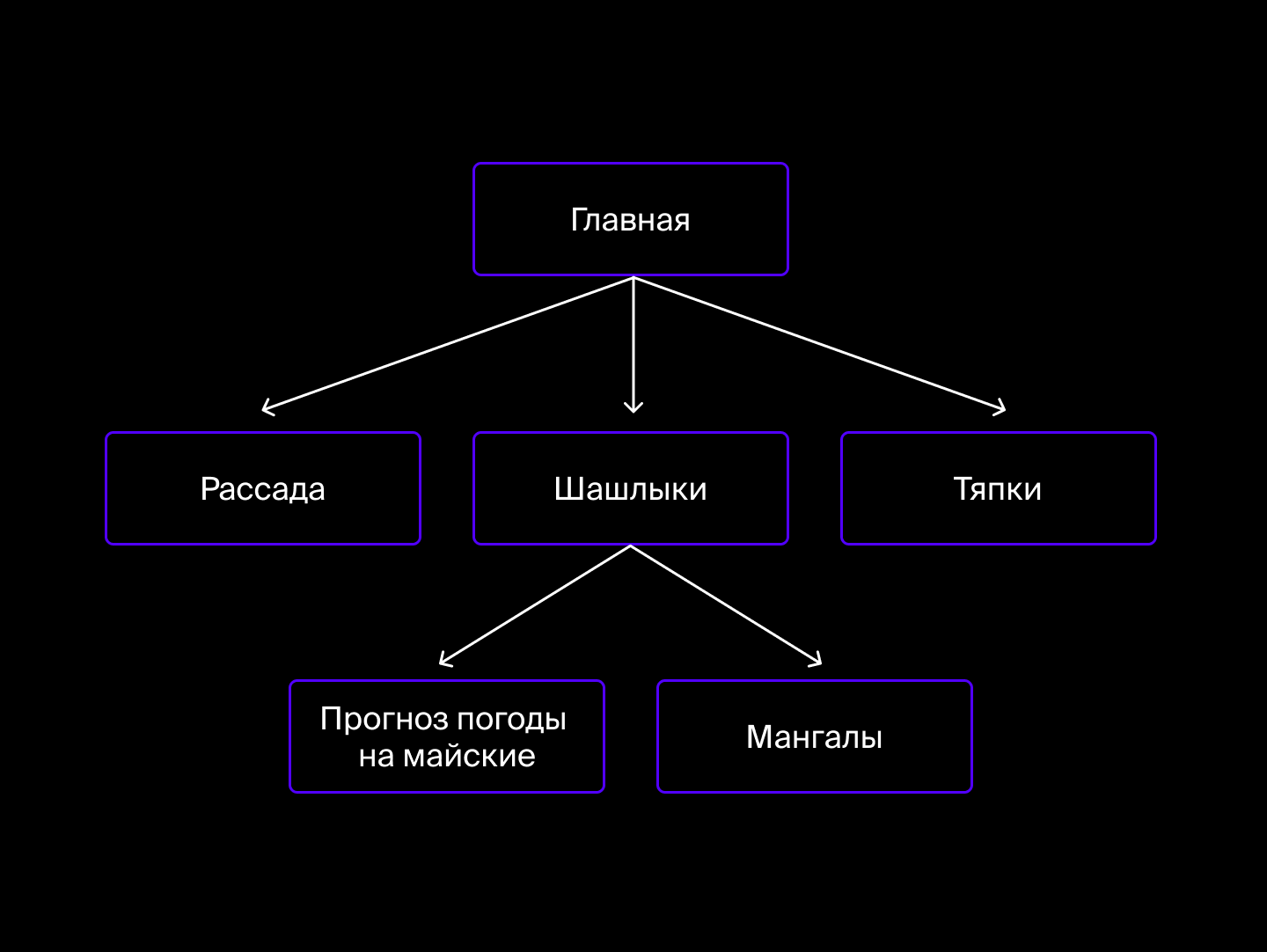


Слишком сложно, слишком...

## Когда навигационное меню не подойдёт

Если вы создаёте сайт, где нет сложной иерархии, хлебные крошки могут быть избыточны. А если сомневаетесь, нужны ли вам крошки, можете схематично нарисовать навигацию сайта.

Вот пример простого информационного сайта для дачников, где можно обойтись без навигационного меню, так как во все разделы легко вернуться из основного:



Дорогу со страницы с мангалами на страницу с рассадой можно найти и без хлебных крошек.

В этом уроке вы узнали, что хлебные крошки могут быть не только на кухне, но и на сайте. Это полезный элемент интерфейса, который помогает пользователям навигироваться по страницам сайта. Ещё немного практики и можно двигаться дальше!

# Создание страницы 404

Если пользователь перейдёт по ссылке, которой нет в приложении, на экране ничего не отрисуется. Хороший тон — проинформировать пользователя, что он перешёл на неопределённый маршрут.

Для этого с помощью React Router сделаем маршрут «Страница не найдена».

Создадим NotFound404 в директории pages/:

Скопировать кодJSX

import React from 'react';

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

import styles from './not-found.module.css';

import pageNotFound from "../images/404.svg";

export const NotFound404 = () => {

return (

<div className={styles.container}>

<div className={styles.content}>

<img alt="page not found" src={pageNotFound} />

<br />

<Link to='/' className={styles.link}>Перейти в список чатов</Link>

</div>

</div>

);

};

Импортируем новый компонент в корневой компонент App нашего чата.

Внутри App уже создано несколько компонентов Route в Switch:

Скопировать кодJSX

// маршруты внутри файла App.js

<Switch>

<Route path="/login">

<LoginPage />

</Route>

<Route path="/list">

<ListPage />

</Route>

</Switch>

Важен порядок компонентов Route внутри Switch: только первый совпадающий маршрут будет отрисован. Когда Switch обнаружит, что URL браузера не совпадает с /login или /list, то отрисует компонент NotFound404. Для этого в конце Switch добавим ещё один компонент Route.

Скопировать кодJSX

// маршруты внутри файла App.js

<Switch>

<Route path="/login">

<LoginPage />

</Route>

<Route path="/list">

<ListPage />

</Route>

<Route>

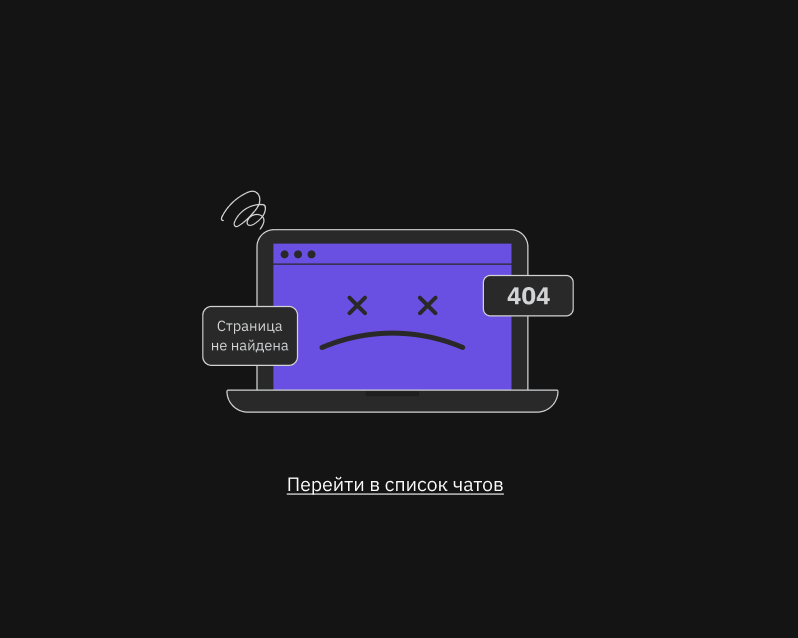
<NotFound404 />

</Route>

</Switch>

Мы могли бы указать path="\*". Символ \* обозначает «любой». Но это значение можно опустить.

Теперь, если запустить проект на локальном сервере и перейти на несуществующий маршрут, например localhost:3000/zdesnichegonet, приложение отрисует страницу 404:



Ну вот:(

В этой теме вы узнали много нового о библиотеке React Router и её возможностях. Ещё немного практики и можно двигаться дальше!

navigate(

`/ingredients/${item.\_id}`,

{state: { modal: true }},

);

<Navigate to="/login" state={{ from: location.pathname } } replace/>