1) <https://nextjs.org/>

React framevork.

Роутинг, серверные компоненты.

На сервере генерируется готовый html, что хорошо для СЕО.

<https://www.youtube.com/watch?v=VGiphTEApfo&t=185s>



Можно указать конкретную версию:



На май 2023 года.

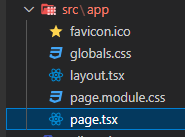
Если последнюю, то дописываем: latest



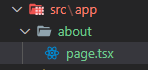
2) Для автоматического форматирования можно создать prettierrc и заполним его пустым объектом:



Уже есть одна созданная страница:



Если мы хотим создать новую страницу:



Будем запускать в режиме.

По умолчанию сборка работает на webpacke.

Но компания Vercel, которая делает Next JS, разрабатывает свой сборщик, он называется: turbopack.

Запустим ем в бета режиме:

  "scripts": {

    "dev": "next dev --turbo",

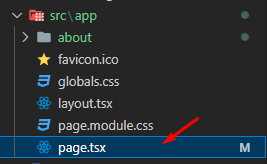
    "build": "next build",

    "start": "next start",

    "lint": "next lint"

  },

3) Базовая страница:



// import Image from 'next/image'

// import styles from './page.module.css'

export default function Home() {

  return (

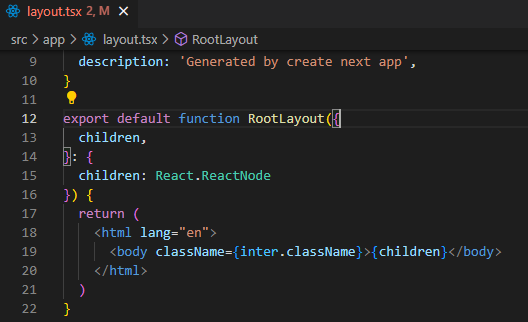
    <h1>Welcome to NaextJS world</h1>

  )

}

4) Также есть отдельный **layout.tsx**

Один layout в корне всегда должен лежать. Так называемый корневой **layout**



Нужно это для того, что Next автоматическит не добавляет html/body во все файлы, поэтому нужно задать layout, чтобы они были у нас всегда.

сhildren – любая страница

Если мы на главной, то сhildren будет главной страницей, если about, то сhildren about



Также экспортируется объект metadata:

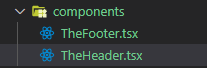
export const metadata: Metadata = {

  title: 'Create Next App',

  description: 'Generated by create next app',

}

5) Создадим два компонента:



TheHeader:

import Link from 'next/link';

function TheHeader() {

  return (

    <header>

        <Link href="/">Home</Link>

        <Link href="/blog">Blog</Link>

        <Link href="/about">About</Link>

    </header>

  )

}

export default TheHeader

Два эти компонента добавим в базовый Layout:

В итоге у нас есть серверный рендеринг SSR. SEO это учитывает.

6) Динамический роутинг



Можем написать id:





Благодаря id, мы можем получать этот параметр:

**page.tsx**

// опишем параметры как отдельный тип

type Props = {

  params: {

    id: string; // id - так как мі так создали папку

  };

};

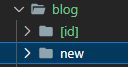
export default function Post({ params: { id } }: Props) {

  return <h1>Post page {id}</h1>;

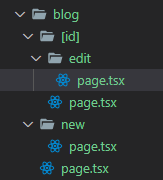
}



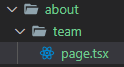
7) Можем рядом создать еще одну динамическую папку:



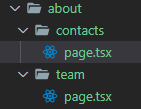
8) Вложенность может быть любая. Можем создать страницу для редактирования:



9) Для about сделаем вложенные страницы



Контакты

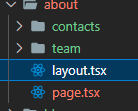


10) Можем создавать любое количество layout.

Они будут вложенны относительно друг друга.

Для about создадим вложенное меню.

И в папке about создадим еще один layout:



import Link from '@/node\_modules/next/link';

export default function AboutLayout({

  children,

}: {

  children: React.ReactNode;

}) {

  return (

    <div>

      <h1>About US</h1>

      <ul>

        <li>

          <Link href="/about/contacts">Contacts</Link>

        </li>

        <li>

          <Link href="/about/team">Team</Link>

        </li>

      </ul>

      {children}

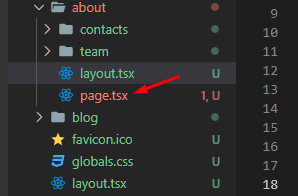
    </div>

  );

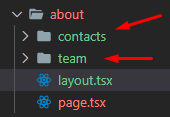
}

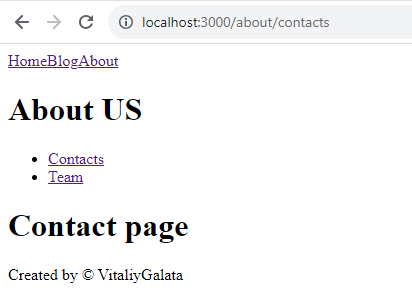
childrem – это конкретная страница

Т.е. та, которая лежит рядом:



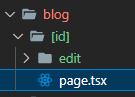
Либо одна из двух:





11) Для создания meta данных для динамических страниц:

**page.tsx:**



export function generateMetadata({ params: { id } }: Props) {

  return {

    title: id,

  };

}

Теперь тайтл будет у нас динамически генерироваться, так как мы передаем в объекте id.

Но данные мы будем получать из сервера.

export async function generateMetadata({

  params: { id },

}: Props): Promise<Metadata> {

  return {

    title: id,

  };

}

12) **Серверный компонент**

Это не просто SSR, который кинул нам HTML, а потом на эту HTML-ку натянул наш JS и мы получили реакт компонент.

Все компоненты, которые мы создавали – они серверные (по умолчанию в Next JS).

У них есть как плюсы, так и минусы.

Главным плюсом серверного компонента является то, что на фронт-енд не поставляется js.

Если нужно нарисовать страницу, на которой всегда есть статические данные, т.е. мы получаем данные по id, но вполне конкретные страницы, с конкретным url, - это у нас статика.

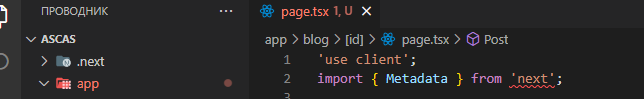
Мы по этой статике можем отдать какой-либо html без js, так как никакое пользовательское взаимодействие не требуется. У пользователя меньший бандл, и приложение загружается быстрее.

Как минус, мы ничего не можем использовать, связанное со стейтом.

Если нам нужен стейт, методы жизненного цикла, хуки и .т.д., то нам нужен клиентский компонент.

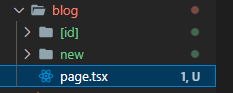
Он начинается с:

'use client';



Если нужно будет делать строку поиска, то клиентский компонент понадобится.

13) Будем получать реальные данные на блоговой странице.



Вспомогательные функции мы не экспортируем.

async function getData() {

  const responce = await fetch('https://jsonplaceholder.typicode.com/posts');

  return responce.json();

}

Серверные компоненты могут быть асинхронными:

export default async function Blog() {

  return <h1>Blog page</h1>;

}

export default async function Blog() {

  const posts = await getData();

  return (

    <>

      <h1>Blog page</h1>

      <ul>

        {posts.map((post: any) => (

          <li key={post.id}>

            <Link href={`blog/${post.id}`}>{post.title}</Link>

          </li>

        ))}

      </ul>

    </>

  );

  return <h1>Blog page</h1>;

}



На сервере произошел запрос, мы получили данные, можем переходить по этим страничкам.

Запрос случился только один раз.

Если опять перейдем на блог, данные загрузятся мгновенно.

Это связанно с кешированием.

Т.е. с помощью фетчи получили данные и они в кеше.

Next предоставляет возможность управления кешированием.

async function getData() {

  const responce = await fetch('https://jsonplaceholder.typicode.com/posts',{

    next:{

      revalidate: 60, // время в секундах - с каким промежутком мы хотим делать запросы

    }

  });

  return responce.json();

}

Т.е. обновление будет идти не для каждого пользователя.

А для сервера в целом.

**Вывод**:

Данные мы получаем на сервере:

   <ul>

        {posts.map(

          (post: any) => console.log(post)

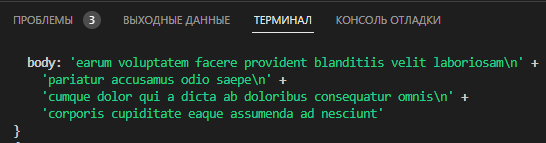
          // <li key={post.id}>

          //   <Link href={`blog/${post.id}`}>{post.title}</Link>

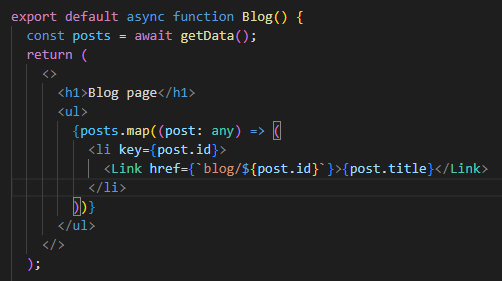
          // </li>

        )}

      </ul>

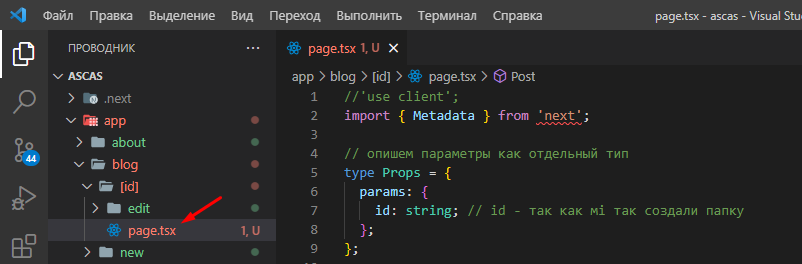


Т.е. это код не будет на фронте рендерится:



Так как серверные компоненты не поставляют на фронт-енд js.

14) На уровне динамической страницы тоже будем получать данные.



// функция получения данных по id

async function getData(id: string) {

  const response = await fetch(

    `https://jsonplaceholder.typicode.com/posts/${id}`,

    {

      next: {

        revalidate: 60, // время в секундах - с каким промежутком мы хотим делать запросы

      },

    }

  );

  return response.json();

}

export default async function Post({ params: { id } }: Props) {

  const post = await getData(id);

  return (

    <>

      <h1>{post.title}</h1>

      <p>{post.body}</p>

    </>

  );

}



Для пользователя это будет просто эта разметка:

 <h1>{post.title}</h1>

 <p>{post.body}</p>

Никакого другого скрипта пользователю в браузер не прийдет.

15) Также можем генерировать данные на уровне meta даты.

// асинхронная функция будет возращать промис с метаданными

export async function generateMetadata({

  params: { id },

}: Props): Promise<Metadata> {

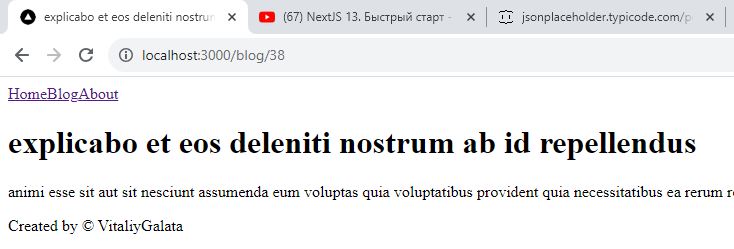
  const post = await getData(id);

  return {

    title: post.title,

  };

}



16) На серверном компоненте консоли будут отрабатывать на сервере

17) Мы можем создавать страницу ошибки и прелоадинга



function LoadingPots() {

  return <h1>Loading...</h1>;

}

export default LoadingPots;

Это зарезервированный файл.

И файл ошибки:

'use client';

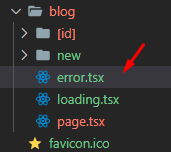
function Error({ error }: { error: Error }) {

  // принимаем error, которій по факту является объектом error

  return <h1>Oops!!! {error.message}</h1>;

}

export default Error;



import type { Metadata } from 'next';

import Link from '@/node\_modules/next/link';

async function getData() {

  const responce = await fetch('https://jsonplaceholder.typicode.com/posts1', {

    next: {

      revalidate: 60, // время в секундах - с каким промежутком мы хотим делать запросы

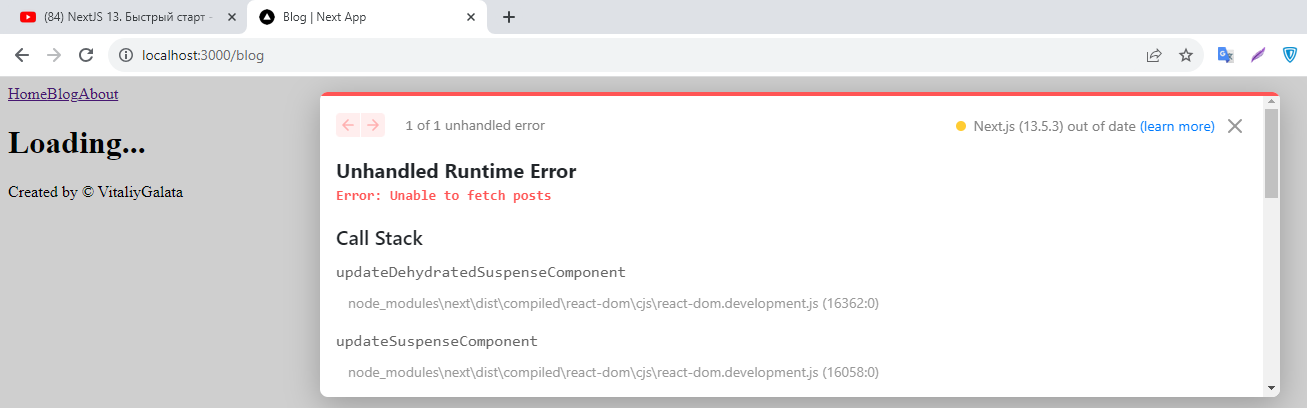
    },

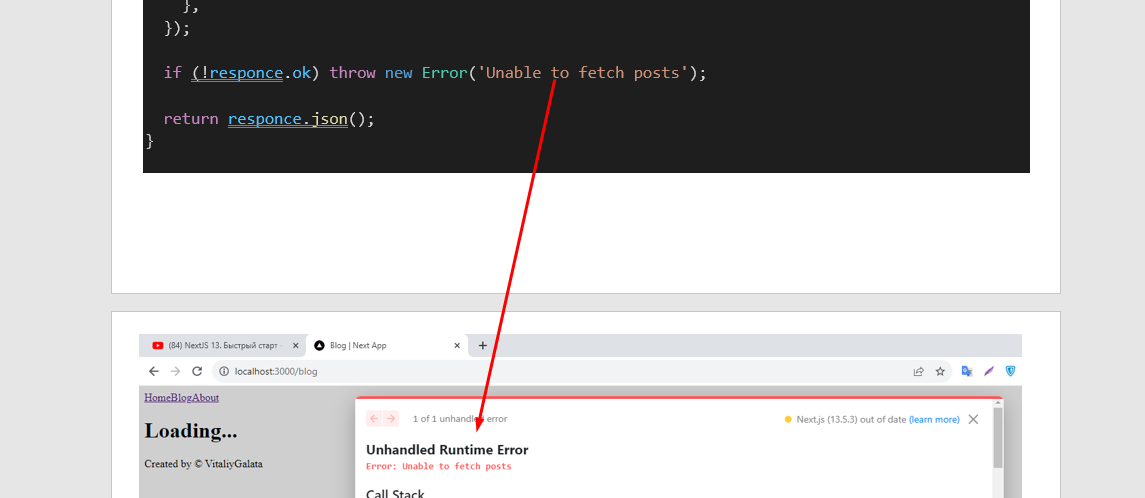
  });

  if (!responce.ok) throw new Error('Unable to fetch posts');

  return responce.json();

}





Т.е. эта ошибка попала в

**{error.message}**